

汇总计算评估结果，左边使用稠密向量，右边使用稀疏向量，对比显示如下。

#### Knn with PCA Dense

##### Metrics:

Accuracy:0.9755 Macro F1:0.9753

Micro F1:0.9755 Kappa:0.9728

Pred\Real	9	8	7	...	2	1	0
9 972	4	10	...	0	1	0	
8	7 937	0	...	5	0	0	
7	3	3 994	...	10	0	1	
...	...	...	...	...	...	...	...
2	2	4	7	...	1008	3	1
1	3	0	13	...	1 1131	1	
0	6	5	0	...	5	0 975	

1 minutes 33 seconds 676.0 milliseconds.

#### Knn with PCA Sparse

##### Metrics:

Accuracy:0.9755 Macro F1:0.9753

Micro F1:0.9755 Kappa:0.9728

Pred\Real	9	8	7	...	2	1	0
9 972	4	10	...	0	1	0	
8	7 937	0	...	5	0	0	
7	3	3 994	...	10	0	1	
...	...	...	...	...	...	...	...
2	2	4	7	...	1008	3	1
1	3	0	13	...	1 1131	1	
0	6	5	0	...	5	0 975	

27 seconds 183.0 milliseconds.

可以观察到如下信息。

- 稠密向量数据和稀疏向量数据在评估指标上是完全一致的。
- 使用了PCA后，放入分类器的特征个数减少了，但评估指标非但没有下降，还有少许上升，这可能是由于提取主成分后减少了一些噪声干扰。
- 稠密向量使用PCA前后，计算时间由5分17秒降到1分33秒。
- 稀疏向量使用PCA前后，计算时间由1分1秒降到27秒。
- 对于PCA+KNN的Pipeline，稠密向量数据与稀疏向量数据的计算时间比例大约是3:1。

如果有已训练好的PCA模型，我们可以直接将PCAModel放入Pipeline中，PCAModel不需要进行训练，直接通过`setModelData`方法载入训练好的模型。左边使用稠密向量，右边使用稀疏向量，具体代码如下。

```
new Pipeline()
    .add(
        new PCAModel()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(VECTOR_COL_NAME)
            .setModelData(
                new AkSourceBatchOp()
                    .setFilePath(DATA_DIR + PCA_MODEL_FILE)
            )
    )
    .add(
        new KnnClassifier()
            .setK(3)
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
    )
}
```

```
new Pipeline()
    .add(
        new PCAModel()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(VECTOR_COL_NAME)
            .setModelData(
                new AkSourceBatchOp()
                    .setFilePath(DATA_DIR + PCA_MODEL_FILE)
            )
    )
    .add(
        new KnnClassifier()
            .setK(3)
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
    )
}
```

```

.setPredictionCol(PREDICTION_COL_NAME)
)
.fit(dense_train_data)
.transform(dense_test_data)
.link(
    new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("Knn PCAModel Dense")
);

```

---

```

.setPredictionCol(PREDICTION_COL_NAME)
)
.fit(sparse_train_data)
.transform(sparse_test_data)
.link(
    new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("Knn PCAModel Sparse")
);

```

汇总计算评估结果，左边使用稠密向量，右边使用稀疏向量，对比显示如下。

### Knn PCAModel Dense

#### Metrics:

Accuracy:0.9755 Macro F1:0.9753

Micro F1:0.9755 Kappa:0.9728

Pred\Real	9	8	7	...	2	1	0
9	972	4	10	...	0	1	0
8	7	937	0	...	5	0	0
7	3	3	994	...	10	0	1
...	...	...	...	...	...	...	...
2	2	4	7	...	1008	3	1
1	3	0	13	...	1	1131	1
0	6	5	0	...	5	0	975

22 seconds 951.0 milliseconds.

### Knn PCAModel Sparse

#### Metrics:

Accuracy:0.9755 Macro F1:0.9753

Micro F1:0.9755 Kappa:0.9728

Pred\Real	9	8	7	...	2	1	0
9	972	4	10	...	0	1	0
8	7	937	0	...	5	0	0
7	3	3	994	...	10	0	1
...	...	...	...	...	...	...	...
2	2	4	7	...	1008	3	1
1	3	0	13	...	1	1131	1
0	6	5	0	...	5	0	975

20 seconds 425.0 milliseconds.

可以观察到如下信息。

- 直接加载模型的方式，对评估指标没有影响。
- 计算时间明显缩短，稠密向量由 1 分 33 秒降到 22 秒，稀疏向量由 27 秒降到 20 秒。
- PCA 的模型是由稀疏向量计算的，但在本实验中应用到稠密向量，仍然可以正常使用。



## 超参数搜索

在机器学习领域，超参数（Hyper-parameter）是在模型训练过程开始之前设置的参数。超参数一般是根据经验确定的变量（譬如学习速率、迭代次数、神经网络的层数）、随机森林中决策树的棵数，K-Means 聚类中的聚类数目，等等。

超参数值的选择会影响模型的效果。例如决策树的树深度参数，如果参数值太小，则不能对一些普遍场景进行细分，分类器太简单，即机器学习常说的欠拟合（Under-fitting），对测试集的分类效果差；如果参数值太大，那么模型对训练数据有很好的细分能力，但是影响模型泛化能力，即过拟合（Over-fitting），在测试数据集上分类效果也会比较差。

分类评估和回归评估有很多评估指标，各有侧重，用户可以根据具体的问题选择适合的评估指标，再针对目标评估指标选择适合的超参数值。

Alink 的模型选择组件是将一个基本的估算器（Estimator）或一个管道（Pipeline）作为输入，针对具体的训练数据选择适合的评估指标，在超参数空间内找到最佳组合超参数。

我们提供了两种常用的模型选择方式。

(1) 网格搜索（Grid Search）：针对所要搜索的每个参数，指定其可以选择的数值列表。需要搜索的超参数组合就是各参数可选值的“网格”（笛卡儿积）组合。

(2) 随机搜索（Random Search）：针对所要搜索的每个参数，指定其分布范围（可以是离散值数组，也可以是服从某种分布的连续区间），然后在各超参数的分布范围内抽样固定次数，得到需要进行搜索的超参数组合。该方法用来避免网格搜索中的组合爆炸。更多内容可以参考 Bergstra 和 Bengio 的文章“Random Search for Hyper-Parameter Optimization”。

评估指标验证的方式有如下两种。

(1) K 折交叉验证 (Cross Validation, CV)：机器学习领域经典的验证方式，对一个超参数组合需要计算  $K$  次，取  $K$  次实验的平均值。

(2) 训练集和验证集拆分 (Train-Validation Split)：简单指定训练数据与验证数据的比例，其好处是对一个超参数组合需要计算 1 次，虽然在评估指标计算的稳定性方面不如 K 折交叉验证，但更适合注重计算时间和资源的场景。

针对不同的问题提供了 4 个调参评估器。

- 二分类调参评估器 (BinaryClassificationTuningEvaluator)。
- 多分类调参评估器 (MultiClassClassificationTuningEvaluator)。
- 回归调参评估器 (RegressionTuningEvaluator)。
- 聚类调参评估器 (ClusterTuningEvaluator)。

## 20.1 示例一：尝试正则系数

使用网格搜索的代码如下所示。

---

```

LogisticRegression lr = new LogisticRegression()
    .setFeatureCols(new_features)
    .setLabelCol(Chap04.LABEL_COL_NAME)
    .setPredictionCol(Chap04.PREDICTION_COL_NAME)
    .setPredictionDetailCol(Chap04.PRED_DETAIL_COL_NAME);

Pipeline pipeline = new Pipeline().add(lr);

GridSearchCV gridSearch = new GridSearchCV()
    .setNumFolds(5)
    .setEstimator(pipeline)
    .setParamGrid(
        new ParamGrid()
            .addGrid(lr, LogisticRegression.L_1,
                new Double[] {0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0, 10.0})
    )
    .setTuningEvaluator(
        new BinaryClassificationTuningEvaluator()
            .setLabelCol(Chap04.LABEL_COL_NAME)
            .setPredictionDetailCol(Chap04.PRED_DETAIL_COL_NAME)
            .setTuningBinaryClassMetric(TuningBinaryClassMetric.AUC)
    )
    .enableLazyPrintTrainInfo();

```

---

---

```
GridSearchCVModel bestModel = gridSearch.fit(train_data);
```

---

这里使用的是 5 折交叉验证，设置要评估的估算器为 lr，并列举全部候选正则系数的取值。GridSearchCV 执行 fit 方法后得到 GridSearchCVModel，即使用搜索出的最优超参数对全部训练数据进行训练而得到的模型。

运行结果如下，前面是超参数搜索的输出结果，显示了各个参数对应的指标值；后面是应用当前最优超参数对整个数据集进行训练得到的模型评估结果。

```
Metric information:
  Metric name: AUC
  Larger is better: true
Tuning information:
+-----+-----+-----+
|      AUC |      stage | param | value |
+-----+-----+-----+
| 0.7952618317852187 | LogisticRegression | 11 | 1.0E-5 |
| 0.793731422403357 | LogisticRegression | 11 | 1.0E-6 |
| 0.7819673037230401 | LogisticRegression | 11 | 0.001 |
| 0.7796277279311166 | LogisticRegression | 11 | 0.01 |
| 0.7715346390692316 | LogisticRegression | 11 | 1.0E-7 |
| 0.7632430058789119 | LogisticRegression | 11 | 1.0E-4 |
| 0.7441272544405644 | LogisticRegression | 11 | 0.1 |
|          0.5 | LogisticRegression | 11 | 1.0 |
|          0.5 | LogisticRegression | 11 | 10.0 |

GridSearchCV
+-----+-----+-----+
| Metrics: |          |
+-----+-----+-----+
AUC: 0.7905 Accuracy: 0.78   Precision: 0.5882   Recall: 0.566   F1: 0.5769   LogLoss: 0.4799
|Pred\Real| 2 | 1 |
+-----+-----+-----+
|          2 | 30 | 21 |
|          1 | 23 | 126 |
```

## 20.2 示例二：搜索GBDT超参数

这里使用随机搜索，同时为了减少计算次数，因此没有使用 K 折交叉验证，而是直接按指定的比例（TrainRatio=0.8）划分训练集与验证集，随机搜索中的搜索参数可以设置为在指定数组中选择或者在指定取值区间内随机取值。具体代码如下。

---

```
GbdtClassifier gbdt = new GbdtClassifier()
.setFeatureCols(featuresColNames)
.setLabelCol(Chap05.LABEL_COL_NAME)
.setPredictionCol(Chap05.PREDICTION_COL_NAME)
.setPredictionDetailCol(Chap05.PRED_DETAIL_COL_NAME);
```

---

```

RandomSearchTVSplit randomSearch = new RandomSearchTVSplit()
    .setNumIter(20)
    .setTrainRatio(0.8)
    .setEstimator(gbdt)
    .setParamDist(
        new ParamDist()
            .addDist(gbdt, GbdtClassifier.NUM_TREES, ValueDist.randArray(new Integer[] {50, 100}))
            .addDist(gbdt, GbdtClassifier.MAX_DEPTH, ValueDist.randInteger(4, 10))
            .addDist(gbdt, GbdtClassifier.MAX_BINS, ValueDist.randArray(new Integer[] {64, 128, 256,
512}))
            .addDist(gbdt, GbdtClassifier.LEARNING_RATE, ValueDist.randArray(new Double[] {0.3, 0.1,
0.01}))
    )
    .setTuningEvaluator(
        new BinaryClassificationTuningEvaluator()
            .setLabelCol(Chap05.LABEL_COL_NAME)
            .setPredictionDetailCol(Chap05.PRED_DETAIL_COL_NAME)
            .setTuningBinaryClassMetric(TuningBinaryClassMetric.F1)
    )
    .enableLazyPrintTrainInfo();
}

RandomSearchTVSplitModel bestModel = randomSearch.fit(train_sample);

```

运行结果如下，可以看出最优指标所对应的各个参数值。

#### Metric information:

Metric name: F1  
Larger is better: true

#### Tuning information:

	F1	stage	param value	stage 2  param 2 value 2	stage 3 param 3 value 3	stage 4  param 4 value 4
0.44776119402985076   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  4   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.3						
0.410958904109589   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  8   GbdtClassifier maxBins  128   GbdtClassifier learningRate  0.3						
0.39285714285714285   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  4   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.1						
0.38095238095238093   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  6   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.1						
0.3561643835616438   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  6   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.3						
0.3492063492063492   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  4   GbdtClassifier maxBins  64   GbdtClassifier learningRate  0.3						
0.3384615384615385   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  8   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.3						
0.3333333333333333   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  10   GbdtClassifier maxBins  64   GbdtClassifier learningRate  0.3						
0.3283582089552239   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  4   GbdtClassifier maxBins  128   GbdtClassifier learningRate  0.1						
0.2933333333333333   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  5   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.3						
0.2909090909090909   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  9   GbdtClassifier maxBins  128   GbdtClassifier learningRate  0.01						
0.2807017543859649   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  7   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.3						
0.2666666666666666   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  4   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.1						
0.25800451612903225   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  5   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.1						
0.22641509433962265   GbdtClassifier numTrees  100   GbdtClassifier maxDepth  7   GbdtClassifier maxBins  128   GbdtClassifier learningRate  0.01						
0.0   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  10   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.01						
0.0   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  6   GbdtClassifier maxBins  64   GbdtClassifier learningRate  0.01						
0.0   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  8   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.01						
0.0   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  6   GbdtClassifier maxBins  256   GbdtClassifier learningRate  0.01						
0.0   GbdtClassifier numTrees  50   GbdtClassifier maxDepth  5   GbdtClassifier maxBins  512   GbdtClassifier learningRate  0.01						

## 20.3 示例三：最佳聚类个数

对于聚类问题，需要使用聚类调参评估器（ClusterTuningEvaluator），其他方面的设置与

20.1 节和 20.2 节相似，具体代码如下。

---

```
KMeans kmeans = new KMeans()
    .setVectorCol(VECTOR_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME);

GridSearchCV cv = new GridSearchCV()
    .setNumFolds(4)
    .setEstimator(kmeans)
    .setParamGrid(
        new ParamGrid()
            .addGrid(kmeans, KMeans.K, new Integer[] {2, 3, 4, 5, 6})
            .addGrid(kmeans, KMeans.DISTANCE_TYPE,
                new DistanceType[] {DistanceType.EUCLIDEAN, DistanceType.COSINE})
    )
    .setTuningEvaluator(
        new ClusterTuningEvaluator()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setTuningClusterMetric(TuningClusterMetric.RI)
    )
    .enableLazyPrintTrainInfo();

GridSearchCVModel bestModel = cv.fit(source);
```

---

搜索结果如下，最优的参数为使用余弦距离，选择聚类个数为 3。

Metric information:

Metric name: ri

Larger is better: true

Tuning information:

ri	stage	param	value	stage 2	param 2	value 2
0.8994191559981034	KMeans	distanceType	COSINE	KMeans	k	3
0.8700608503240082	KMeans	distanceType	EUCLIDEAN	KMeans	k	4
0.8698237711395606	KMeans	distanceType	COSINE	KMeans	k	5
0.8688952110004742	KMeans	distanceType	EUCLIDEAN	KMeans	k	5
0.8517662399241347	KMeans	distanceType	COSINE	KMeans	k	4
0.8423028291449345	KMeans	distanceType	COSINE	KMeans	k	6
0.8401296032874981	KMeans	distanceType	EUCLIDEAN	KMeans	k	6
0.8264185237869449	KMeans	distanceType	EUCLIDEAN	KMeans	k	3
0.7933854907539117	KMeans	distanceType	COSINE	KMeans	k	2
0.7813734787418998	KMeans	distanceType	EUCLIDEAN	KMeans	k	2

利用最佳参数所得模型的聚类评估结果如下。

Metrics:

k:3

VRC:469.9545 DB:0.7682 SilhouetteCoefficient:0.6398  
ARI:0.9222 NMI:0.9144 Purity:0.9733

Label\Cluster	2	1	0
Iris-virginica	0	50	0
Iris-versicolor	0	4	46
Iris-setosa	50	0	0



## 文本分析

我们接触到的很多数据是文本形式的，譬如新闻的标题和内容、电影评论、商品描述等。如何从这些文本数据中寻找相似的内容？新闻中的关键词是什么？用户评论了一段话，表达的意思是喜欢还是不喜欢？本章会讨论如何解决这样的问题。

本章主要介绍文本分析中的基本内容，譬如分词、词频统计、TF-IDF 等。后面的两章会基于文本内容构造特征，使用机器模型进行情感分析和预测。

### 21.1 数据探索

本章使用的数据来自今日头条中的新闻，参见链接 21-1。下载压缩文件，解压后得到 57MB 的文本文件，用文本编辑器打开，如图 21-1 所示。

A screenshot of a text editor showing a list of news items. Each item consists of a unique ID followed by a URL pattern: '6551700932785387022\_!\_101\_!\_news\_culture\_!\_...'. The URLs point to various news articles on the Toutiao platform, such as '京城最值得你来场文化之旅的博物馆\_!', '保利集团,马未都,中国科学技术馆,博物馆,新中国6552368441838272771\_!', '发醉床的垫料种类有哪些?哪种更好?\_!', '6552407965343678723\_!\_101\_!\_news\_culture\_!\_上联:黄山黄河黄皮肤黄土高原。怎么对下联?\_!', '6552332417753940238\_!\_101\_!\_news\_culture\_!\_林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣?\_!', '6552475601595269390\_!\_101\_!\_news\_culture\_!\_黄杨木是什么树?\_!', '6552387648126714125\_!\_101\_!\_news\_culture\_!\_上联:草根登上星光大道,怎么对下联?\_!', '6552271725814350087\_!\_101\_!\_news\_culture\_!\_什么是超写实绘画?\_!', '6552452982015787268\_!\_101\_!\_news\_culture\_!\_松涛听雨莺婉转,下联?\_!', '6552400379038536455\_!\_101\_!\_news\_culture\_!\_上联:老子骑牛读书,下联怎么对?\_!'

图 21-1

可以看到，每条记录为一行，各数据列间使用“`_!`”进行分隔。使用 Alink CSV 格式数据源组件 `CsvSourceBatchOp`，经过如下配置即可读取。

```
static final String DATA_DIR = "/Users/yangxu/alink/data/news_toutiao/";
static final String ORIGIN_TRAIN_FILE = "toutiao_cat_data.txt";
static final String FIELD_DELIMITER = "_!";
static final String SCHEMA_STRING =
    "id string, category_code int, category_name string, news_title string, keywords string";

private static BatchOperator getSource() {
    return new CsvSourceBatchOp()
        .setFilePath(DATA_DIR + ORIGIN_TRAIN_FILE)
        .setSchemaStr(SCHEMA_STRING)
        .setFieldDelimiter(FIELD_DELIMITER);
}
```

我们使用两个函数获取数据的基本信息，具体代码如下。

```
getSource()
.lazyPrint(10)
.lazyPrintStatistics();
```

获取前 10 条数据，输出如下。

id	category_code	category_name	news_title	keywords
6551700932705387022	101	news_culture	京城最值得你来场文化之旅的博物馆	保利集团, 马未都, 中国科学技术馆, 博物馆, 新中国
6552368441838272771	101	news_culture	发酵床的垫料种类有哪些？哪种更好？	null
6552407965343678723	101	news_culture	上联：黄山黄河黄皮肤黄土高原。怎么对下联？	null
6552332417753940238	101	news_culture	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？	null
6552475601595269390	101	news_culture	黄杨木是什么树？	null
6552387648126714125	101	news_culture	上联：草根登上星光道，怎么对下联？	null
6552271725814350087	101	news_culture	什么是超写实绘画？	null
6552452982015787268	101	news_culture	松涛听雨莺婉转，下联？	null
6552400379030536455	101	news_culture	上联：老子骑牛读书，下联怎么对？	null
6552339283632455939	101	news_culture	上联：山水醉人何须酒。如何对下联？	null

输出的全表基本统计结果如下。

colName	count	missing	sum	mean	variance	min	max
id	382688	0	NaN	NaN	NaN	NaN	NaN
category_code	382688	0	41163656	107.5645	22.4489	100	116
category_name	382688	0	NaN	NaN	NaN	NaN	NaN
news_title	382688	0	NaN	NaN	NaN	NaN	NaN
keywords	382688	122453	NaN	NaN	NaN	NaN	NaN

总记录数为 382688 条，`keywords` 列有缺失值，缺失比例约为 1/3。

我们再看一下类别分布情况，数据中提供了 `category_code` 和 `category_name` 两项，分别表示分类代码和分类名称。一般来说，它们应该是一致的，保险起见，我们也一并进行验证。同

时对 category\_code、category\_name 两列使用 groupBy，并进行计数操作，具体代码如下。

```
getSource()
    .groupBy("category_code, category_name", "category_code, category_name, COUNT(category_name)
AS cnt")
    .orderBy("category_code", 100)
    .lazyPrint(-1);
```

结果如表 21-1 所示。

表 21-1 Category 代码与名称

category_code	category_name	cnt
100	news_story	6273
101	news_culture	28031
102	news_entertainment	39396
103	news_sports	37568
104	news_finance	27085
106	news_house	17672
107	news_car	35785
108	news_edu	27058
109	news_tech	41543
110	news_military	24984
112	news_travel	21422
113	news_world	26909
114	stock	340
115	news_agriculture	19322
116	news_game	29300

显然，category\_code 列与 category\_name 列的值是一一对应的；类别分布并不平均，最大的类别 news\_tech 有 41543 条数据，最小的类别只有 340 条数据，相差 100 多倍。

## 21.2 分词

相比于单个的字符，单词可以表达更明确的意义，一些数字和成语经过分词后还是作为一个整体存在的。

## 21.2.1 中文分词

中文分词是指基于词法分析系统，对指定的文本内容进行分词，分词后的各个词语间以空格作为分隔符。

我们先构造 3 个句子，然后使用分词组件 SegmentBatchOp，设置选择进行分词操作的文本列 SelectedCol，并设置分词结果输出列 OutputCol，如下代码所示。

```
String[] strings = new String[] {
    "大家好！我在学习、使用 Alink。",
    "【流式计算和批式计算】、(Alink)",
    "《人工智能》，“机器学习”？2020"
};

MemSourceBatchOp source = new MemSourceBatchOp(strings, "sentence");

source.link(
    new SegmentBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("words")
).lazyPrint(-1, "< Segment >");
```

输出结果如表 21-2 所示。

表 21-2 分词结果

sentence	words
大家好！我在学习、使用 Alink。	大家 好！ 我 在 学习、 使用 alink。
【流式计算和批式计算】、(Alink)	【 流式 计算 和 批式 计算 】 、 (alink)
《人工智能》，“机器学习”？2020	《 人工 智能 》 ， “ 机 器 学习 ” ？ 2020

左边是原始文本，右边是分词结果，显然，单词之间、单词与标点符号之间都以空格分隔，英文字母变成了小写的，“机器学习”被拆成了两个单词——“机器”与“学习”，但我们希望把其当作整体看待，这就需要使用分词组件的自定义词典功能。使用方法 UserDefinedDict，输入需要被当作整体看待的单词，具体代码如下。

```
source.link(
    new SegmentBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("words")
        .setUserDefinedDict("流式计算", "机器学习")
).print();
```

运行结果如表 21-3 所示，右边是分词后的结果，我们发现“流式计算”和“机器学习”没有被继续拆分，已经被当作单词看待。

表 21-3 使用自定义词典的分词结果

sentence	words
大家好！我在学习、使用 Alink。	大家 好！ 我 在 学习、 使用 alink。
【流式计算和批式计算】、(Alink)	【 流式计算 和 批式 计算 】 、 (alink)
《人工智能》，“机器学习”？2020	《 人工智能 》，“ 机器学习 ”？ 2020

接下来，演示停用词过滤组件 StopWordsRemoverBatchOp，它经常与分词组件配合使用，基于分词的结果，过滤掉常用的标点符号，还要把一些助词、副词、代词等经常出现且不能单独体现意义的词过滤掉。分词组件 SegmentBatchOp 后面连接停用词过滤组件 StopWordsRemoverBatchOp，并设置 SelectedCol，选择输入分词结果列，并设置输出列，具体代码如下。

```
source.link(
    new SegmentBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("words")
        .setUserDefinedDict("流式计算", "机器学习")
).link(
    new StopWordsRemoverBatchOp()
        .setSelectedCol("words")
        .setOutputCol("left_words")
).print();
```

运行结果如表 21-4 所示。

表 21-4 停用词过滤的结果

sentence	words	left_words
大家好！我在学习、使用 Alink。	大家 好！ 我 在 学习、 使用 alink。	大家 好 学习 使用 alink
【流式计算和批式计算】、(Alink)	【 流式计算 和 批式 计算 】 、 (alink)	流式计算 批式 计算 alink
《人工智能》，“机器学习”？2020	《 人工智能 》，“ 机器学习 ”？ 2020	人工智能 机器学 习 2020

停用词过滤结果在最右边一列，过滤后的内容没有了标点符号，“我”“在”“和”这些

单词也被过滤掉了。

我们同样也可以定义新的停用词，使用方法 `setStopWords`，新增了两个停用词——“计算”和“2020”，如下代码所示。

---

```
source.link(
    new SegmentBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("words")
        .setUserDefinedDict("流式计算", "机器学习")
).link(
    new StopWordsRemoverBatchOp()
        .setSelectedCol("words")
        .setOutputCol("left_words")
        .setStopWords("计算", "2020")
).print();
```

---

运行结果如表 21-5 所示，可以看到“计算”和“2020”这两个词都被过滤掉了。

表 21-5 扩展停用词

sentence	words	left_words
大家好！我在学习、使用 Alink。	大家 好！ 我 在 学习 、 使用 alink 。	大家 好 学习 使用 alink
【流式计算和批式计算】、(Alink)	【 流式计算 和 批式 计算 】 、 (alink )	流式计算 批式 alink
《人工智能》，“机器学习”？ 2020	《 人工智能 》 ， “ 机器学习 ” ？ 2020	人工智能 机器学习

我们再对本章的数据集进行分词操作，具体代码如下。

---

```
getSource()
    .select("news_title")
    .link(
        new SegmentBatchOp()
            .setSelectedCol("news_title")
            .setOutputCol("segmented_title")
    )
    .firstN(10)
    .print();
```

---

运行结果如表 21-6 所示。

表 21-6 新闻标题的分词结果

news_title	segmented_title
京城最值得你来场文化之旅的博物馆	京城 最 值得 你 来场 文化 之 旅 的 博 物 馆
发酵床的垫料种类有哪些？哪种更好？	发 酵 床 的 垫 料 种 类 有 哪 些 ？ 哪 种 更 好 ？
上联：黄山黄河黄皮肤黄土高原。怎么对下联？	上 联： 黄 山 黄 河 黄 皮 肤 黄 土 高 原 。 怎 么 对 下 联 ？
林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？	林 徽 因 什 么 理 由 拒 绝 了 徐 志 摩 而 选 择 梁 思 成 为 终 生 伴 侣 ？
黄杨木是什么树？	黄 杨 木 是 什 么 树 ？
上联：草根登上星光道，怎么对下联？	上 联： 草 根 登 上 星 光 道 ， 怎 么 对 下 联 ？
什么是超写实绘画？	什 么 是 超 写 实 绘 画 ？
松涛听雨莺婉转，下联？	松 涛 听 雨 莺 婉 转 ， 下 联 ？
上联：老子骑牛读书，下联怎么对？	上 联： 老 子 骑 牛 读 书 ， 下 联 怎 么 对 ？
上联：山水醉人何须酒。如何对下联？	上 联： 山 水 醉 人 何 须 酒 。 如 何 对 下 联 ？

## 21.2.2 Tokenizer 和 RegexTokenizer

Tokenizer 针对英文文本，将文档（譬如句子）分解为单元个体（譬如单词、数字、标点符号等）。使用时只需指定输入列和输出列，不需要设置其他参数。Tokenizer 会将文档内容转化为小写的，并以 `\s+` 为分隔符（即以单个或连续的多个空格作为分隔符）将文档分割为多个单元个体，并用空格符再将各单元个体连接为一个大字符串。

RegexTokenizer 是 Tokenizer 的升级版，可以通过正则表达式匹配完成更高级的操作。其中有两个参数非常重要——`gaps` 和 `pattern`。参数 `gaps` 是布尔型变量，表明正则匹配针对的对象，默认值为 `true`，即对分割符进行正则匹配，具体的正则表达式为参数 `pattern` 的内容；参数 `pattern` 的默认值为 `\s+`，所以在默认参数情况下，RegexTokenizer 是 Tokenizer 的输出是一致的。将参数 `gaps` 设置为 `false` 时，则是对要提取的内容进行正则匹配，找到所有出现的匹配并以空格符连接，作为输出结果。

常用的正则表达式字符如表 21-7 所示。

表 21-7 常用的正则表达式字符

字符	含义
<code>+</code>	规定其前导字符必须在目标对象中出现一次或连续多次
<code>*</code>	规定其前导字符必须在目标对象中出现零次或连续多次
<code>?</code>	规定其前导对象必须在目标对象中出现零次或一次

续表

字符	含义
\s	用于匹配单个空格符，包括 Tab 键和换行符
\S	用于匹配除单个空格符之外的所有字符
\d	用于匹配从 0 到 9 的数字
\w	用于匹配字母、数字或下画线字符
\W	用于匹配所有与\w 不匹配的字符
.	用于匹配除换行符之外的所有字符

容易看出，正则表达式\s+可以用于匹配目标对象中的一个或多个空格字符。而\s、\S、\w、\W 都可以看作相反的操作，再考虑到参数 gaps 取值 true 和 false 时会影响正则匹配的对象是分隔符还是分隔出的内容，所以表 21-8 列出了等效的参数设置。

表 21-8 等效的参数设置

参数设置	等效的参数设置
gaps=true;pattern="\W"	gaps=false;pattern="\w"
gaps=true;pattern="\s"	gaps=false;pattern="\S"

下面通过实验看看各种组件和参数组合的结果。

首先来看 TokenizerBatchOp 组件和 RegexTokenizerBatchOp 组件在默认参数情况下的处理结果。这里构造了两个字符串，包含了多种元素，具体代码如下。

```

String[] strings = new String[] {
    "Hello!      This is Alink!",
    "Flink,Alink..AI#ML@2020"
};

MemSourceBatchOp source = new MemSourceBatchOp(strings, "sentence");

source
    .link(
        new TokenizerBatchOp()
            .setSelectedCol("sentence")
            .setOutputCol("tokens")
    )
    .link(
        new RegexTokenizerBatchOp()
            .setSelectedCol("sentence")
            .setOutputCol("regex_tokens")
    ).lazyPrint(-1);

```

运行结果如表 21-9 所示，显然，在默认参数的情况下，两个组件的处理结果一致。我们也注意到，第一个字符串中有连续的多个空格，在分词结果中没有出现；在默认参数情况下，标点符号没有与单词分离开。

表 21-9 英文 Tokenizer 的结果

sentence	tokens	regex_tokens
Hello! This is Alink!	hello! this is alink!	hello! this is alink!
Flink,Alink..AI#ML@2020	flink,alink..ai#ml@2020	flink,alink..ai#ml@2020

下面的实验基于 RegexTokenizerBatchOp 组件，看看参数 `gaps=true;pattern="\W"` 与 `gaps=false;pattern="\w+"` 的结果是否一致。注意，组件的默认 `gaps` 参数值为 `true`，具体代码如下。

```
source
    .link(
        new RegexTokenizerBatchOp()
            .setSelectedCol("sentence")
            .setOutputCol("tokens_1")
            .setPattern("\W+")
    )
    .link(
        new RegexTokenizerBatchOp()
            .setSelectedCol("sentence")
            .setOutputCol("tokens_2")
            .setGaps(false)
            .setPattern("\w+")
    )
).lazyPrint(-1);
```

运行结果如表 21-10 所示，两种参数设置得到的结果是一致的。结果中将单词与标点符号分离了，并去除了标点符号。

表 21-10 不同参数对 Tokenizer 的影响

sentence	tokens_1	tokens_2
Hello! This is Alink!	hello this is alink	hello this is alink
Flink,Alink..AI#ML@2020	flink alink ai ml 2020	flink alink ai ml 2020

我们注意到，结果中的单词都是小写的，这是因为 RegexTokenizerBatchOp 组件的参数 `ToLowerCase` 的默认值为 `true`。如下代码所示，进行对比实验。

```
source
    .link(
```

```

new RegexTokenizerBatchOp()
    .setSelectedCol("sentence")
    .setOutputCol("tokens_1")
    .setPattern("\W+")
)
.link(
    new RegexTokenizerBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("tokens_2")
        .setPattern("\W+")
        .setToLowerCase(false)
).lazyPrint(-1);

```

运行结果如表 21-11 所示，最右边一列中的单词没有被转换为小写形式。

表 21-11 小写形式转换

sentence	tokens_1	tokens_2
Hello! This is Alink!	hello this is alink	Hello This is Alink
Flink,Alink..AI#ML@2020	flink alink ai ml 2020	Flink Alink AI ML 2020

对于 Tokenize 的结果，我们同样可以使用停用词过滤组件 StopWordsRemoverBatchOp，如下代码所示。

```

source
.link(
    new RegexTokenizerBatchOp()
        .setSelectedCol("sentence")
        .setOutputCol("tokens")
        .setPattern("\W+")
)
.link(
    new StopWordsRemoverBatchOp()
        .setSelectedCol("tokens")
        .setOutputCol("left_tokens")
).lazyPrint(-1);

```

运行结果如表 21-12 所示，可以看到，单词“this”和“is”都被过滤掉了。

表 21-12 英文停用词过滤

sentence	tokens	left_tokens
Hello! This is Alink!	hello this is alink	hello alink
Flink,Alink..AI#ML@2020	flink alink ai ml 2020	flink alink ai ml 2020

### 21.3 词频统计

Alink 提供了 WordCountBatchOp 组件，统计各个单词在数据集中出现的总数。为了便于比对，这里从数据集中选出了 10 条数据，先进行分词，然后连接 WordCountBatchOp 组件，组件中只需要设置文本数据所在列的名称，其输出的格式是固定的，共 2 列（word 和 cnt），随后按个数进行降序排序并输出。具体代码如下。

```
BatchOperator titles = getSource()
    .firstN(10)
    .select("news_title")
    .link(
        new SegmentBatchOp()
            .setSelectedCol("news_title")
            .setOutputCol("segmented_title")
            .setReservedCols(new String[] {})
    );
titles
    .link(
        new WordCountBatchOp()
            .setSelectedCol("segmented_title")
    )
    .orderBy("cnt", 100, false)
    .lazyPrint(-1, "WordCount");
```

运行结果如表 21-13 所示，其中显示了出现次数较多的单词。问号出现的次数最多，“下联”在单词中出现的次数最多。

表 21-13 单词出现的次数

word	cnt
?	10
下联	5
对	4
上联	4
:	4
,	3
怎么	3
什么	3

续表

word	cnt
。	2
是	2
的	2
写实	1
博物馆	1
发酵	1
.....	.....

进一步，如果想知道各单词在每个文本数据中的出现次数，可以使用组件 DocWordCountBatchOp。将每个文本数据看作一个 Doc，这里直接将文本列当作 ID 列，相关代码如下。

```
titles
    .link()
        new DocWordCountBatchOp()
            .setDocIdCol("segmented_title")
            .setContentCol("segmented_title")
    )
    .lazyPrint(-1, "DocWordCount");
```

运行结果如表 21-14 所示，此时的 cnt 列是左边文本中出现相应单词的次数。

表 21-14 DocWordCount 组件的运行结果

segmented_title	word	cnt
.....	.....	.....
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	种类	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	的	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	更好	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	垫料	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	哪种	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	有	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	床	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	哪些	1

续表

segmented_title	word	cnt
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	发酵	1
发酵 床 的 垫料 种类 有 哪些 ？ 哪种 更好 ？	？	2
.....	.....	.....

## 21.4 单词的区分度

下面首先介绍几个重要概念。

词频 (Term Frequency, TF)，即单词在该文档中出现的频率。

$$TF = \frac{\text{单词在该文档出现的次数}}{\text{该文档中单词的总数}}$$

显然，TF 的值越大，表示这个单词越重要。

逆文本频率指数 (Inverse Document Frequency, IDF)，是指在一个文档库中，一个分词出现在的文档数越少，那么它越能和其他文档区别开来。

$$IDF = \ln\left(\frac{\text{总文档数}}{\text{出现该单词的文档数}}\right)$$

TF-IDF (Term Frequency-Inverse Document Frequency) 通过计算单词在文档中的频率，进而得到单词的权重，可以用来评估单词对于一个文档集或一个语料库中的其中一份文档的重要程度。单词的重要性与它在文档中出现的次数成正比，但同时会与它在语料库中出现的频率成反比。

计算 TF-IDF 的值就是将 TF 和 IDF 的值相乘。对于某一特定文档内的高频(即 TF 的值较高)单词，并且包含该单词的文档数目在整个文档集合中所占的比例较低(即 IDF 的值较高)，则可以产生高权重的 TF-IDF。所以，使用 TF-IDF 可以过滤掉常见的单词，保留能区分该文档的单词。

Alink 提供了两个组件 DocCountVectorizer 和 DocHashCountVectorizer，都可以用于计算上述的指标。二者的区别在于，DocCountVectorizer 是把每个单词对应一个向量分量，单词的总个数就是向量的维度；DocHashCountVectorizer 是指定向量维度，示例中的 setNumFeatures 方法就限定了向量维度为 100，各个单词通过 hash 的方式映射到向量的相应分量。这里会存在多个

单词映射到同一个分量的问题，使用大的向量维度可以降低产生这种问题的概率。

```

for (String featureType : new String[] {"WORD_COUNT", "Binary", "TF", "IDF", "TF_IDF"}) {
    new DocCountVectorizer()
        .setFeatureType(featureType)
        .setSelectedCol("segmented_title")
        .setOutputCol("vec")
        .fit(titles)
        .transform(titles)
        .lazyPrint(-1, "DocCountVectorizer + " + featureType);
}

for (String featureType : new String[] {"WORD_COUNT", "Binary", "TF", "IDF", "TF_IDF"}) {
    new DocHashCountVectorizer()
        .setFeatureType(featureType)
        .setSelectedCol("segmented_title")
        .setOutputCol("vec")
        .setNumFeatures(100)
        .fit(titles)
        .transform(titles)
        .lazyPrint(-1, "DocHashCountVectorizer + " + featureType);
}

```

输出内容较多，我们选择一条经过分词操作的文本数据（“发酵 床 的 垫料 种类 有 哪些？ 哪种 更好？”），使用组件 DocCountVectorizer 在各种方式下生成的结果向量汇总见表 21-15。

表 21-15 向量生成方式及结果

向量生成方式	发酵 床 的 垫料 种类 有 哪些？ 哪种 更好？
WORD_COUNT	\$61\$0:2.0 10:1.0 20:1.0 22:1.0 23:1.0 24:1.0 28:1.0 33:1.0 35:1.0 43:1.0
BINARY	\$61\$0:1.0 10:1.0 20:1.0 22:1.0 23:1.0 24:1.0 28:1.0 33:1.0 35:1.0 43:1.0
TF	\$61\$0:0.181818181818182 10:0.090909090909091 20:0.090909090909091 22:0.090909090909091 23:0.090909090909091 24:0.090909090909091 28:0.090909090909091 33:0.090909090909091 35:0.090909090909091 43:0.090909090909091
IDF	\$61\$0:0.09531017980432493 10:1.2992829841302609 20:1.7047480922384253 22:1.7047480922384253 23:1.7047480922384253 24:1.7047480922384253 28:1.7047480922384253 33:1.7047480922384253 35:1.7047480922384253 43:1.7047480922384253
TF_IDF	\$61\$0:0.017329123600786353 10:0.11811663492093281 20:0.15497709929440232 22:0.15497709929440232 23:0.15497709929440232 24:0.15497709929440232 28:0.15497709929440232 33:0.15497709929440232 35:0.15497709929440232 43:0.15497709929440232

选择另一条文本数据（“京城 最 值得 你 来场 文化 之旅 的 博物馆”），使用 DocHashCountVectorizer 组件，并设置参数 NumFeatures=100，在各种方式下生成的结果向量汇总见表 21-16。注意 WORD\_COUNT 方法生成的结果向量，索引号 43 对应的值为 2.0，而该文本中并没有相同的单词，应该是某两个单词被 hash 到同一个向量分量相同导致的。

表 21-16 有向量维度限制的向量生成结果

向量生成方式	京城 最 值得 你 来场 文化 之旅 的 博物馆
WORD_COUNT	\$100\$8:1.0 21:1.0 26:1.0 38:1.0 42:1.0 43:2.0 56:1.0 93:1.0
BINARY	\$100\$8:1.0 21:1.0 26:1.0 38:1.0 42:1.0 43:1.0 56:1.0 93:1.0
TF	\$100\$8:0.1111111111111111 21:0.1111111111111111 26:0.1111111111111111 38:0.1111111111111111 42:0.1111111111111111 43:0.2222222222222222 56:0.1111111111111111 93:0.1111111111111111
IDF	\$100\$8:1.2992829841302609 21:1.7047480922384253 26:1.0116009116784799 38:1.7047480922384253 42:1.2992829841302609 43:1.2992829841302609 56:1.7047480922384253 93:0.6061358035703155
TF_IDF	\$100\$8:0.14436477601447342 21:0.18941645469315835 26:0.11240010129760887 38:0.18941645469315835 42:0.14436477601447342 43:0.28872955202894685 56:0.18941645469315835 93:0.06734842261892394

## 21.5 抽取关键词

文档中的关键词是对详细文档内容的提炼和总结，可以体现文档的基本内容，帮助用户迅速了解文档所关注的领域和所持的基本观点。本小节会从常用的关键内容计算原理入手，帮助用户更深刻地理解“关键”的含义，然后使用关键词抽取组件对示例数据进行操作。

### 21.5.1 原理简介

关键词和关键句的抽取常用到 TextRank 算法，该算法可以看作著名的网页排名 PageRank 算法在文本领域的应用和改进。本节会首先简要地介绍 PageRank 算法，然后把文本的特点融入其中，以便于我们很自然地理解 TextRank 算法。

在网页排名方面，Google 的 PageRank 算法很好地体现了网页的相关性和重要性，一个网页的重要性不但和有多少个网页链接它相关，还与每个链接它的网页的重要性相关。图 21-2 所

示的 PageRank 卡通演示图（来自维基百科）比较形象地说明了这一点，图标个头越大，表示它越重要。

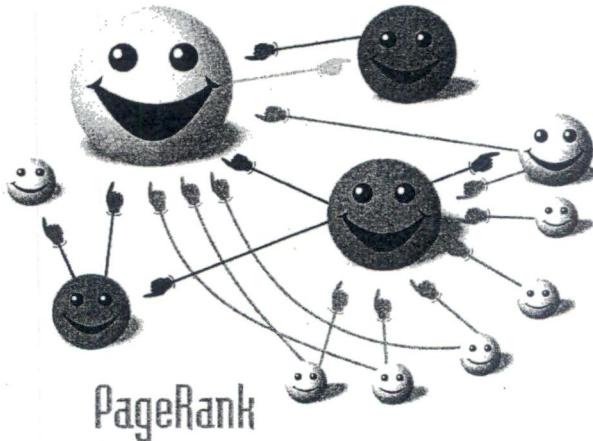


图 21-2

PageRank 的计算公式如下。

$$\text{PR}(P_i) = (1 - d) + d \sum_{P_k \in \text{In}(P_i)} \frac{1}{|\text{Out}(P_k)|} \text{PR}(P_k)$$

其中， $\text{PR}(P_i)$  是网页  $P_i$  的 PageRank 值； $d$  是阻尼系数，一般设置为 0.85； $\text{In}(P_i)$  是存在指向网页  $P_k$  的链接的网页集合； $\text{Out}(P_k)$  是网页  $P_k$  中存在的链接指向的网页的集合。

TextRank 算法可以被看作 PageRank 在自然语言处理领域的应用，主要用于为文本生成关键字和摘要。TextRank 在 PageRank 的基础上引入了边的权值的概念，代表两个单词或句子的相似度，公式如下。

$$\text{TR}(T_i) = (1 - d) + d \cdot \sum_{T_k \in \text{In}(T_i)} \frac{w_{ki}}{\sum_{T_j \in \text{Out}(T_k)} w_{kj}} \text{TR}(T_k)$$

其中， $\text{TR}(T_i)$  是单词或句子  $T_i$  的 TextRank 值； $d$  是阻尼系数，一般设置为 0.85； $\text{In}(T_i)$  是指向  $T_i$  的单词或句子集合； $\text{Out}(T_k)$  是被  $T_k$  指向的单词或句子集合。

在计算关键词时，首先要将文档进行分词，并采取停用词过滤等操作。单词之间的联系是通过是否在句子中相邻出现来定义的，具体做法是，定义一个窗口长度  $k$ ，对于每个句子的单词

$$w_1, w_2, w_3, \dots, w_{m-1}, w_m$$

考查各个窗口

$$\begin{aligned} & w_1, w_2, \dots, w_k \\ & w_2, w_3, \dots, w_{k+1} \\ & \vdots \\ & w_{m-k+1}, w_{m-k+2}, \dots, w_m \end{aligned}$$

在每个窗口中出现的任意两个单词间定义一个双向的边，每条边出现的次数即为其权重。

在定义了单词和联系后，就可以使用 TextRank 算法进行计算，并根据 rank 值确定单词的重要性。

## 21.5.2 示例

很多人都听过相声《报菜名》，那么菜单中的关键词是什么呢？下面我们就用关键词提取组件计算一下，相关代码如下。

```
String[] strings = new String[] {
    "蒸羊羔、蒸熊掌、蒸鹿尾儿、烧花鸭、烧雏鸡、烧子鹅、卤猪、卤鸭、酱鸡、腊肉、松花小肚儿、晾肉、香肠儿、什锦苏盘、熏鸡白肚儿、清蒸八宝猪、江米酿鸭子、罐儿野鸡、罐儿鹌鹑。"
    + "卤什件儿、卤子鹅、山鸡、兔脯、菜蟒、银鱼、清蒸哈什蚂、烩鸭丝、烩鸭腰、烩鸭条、清拌鸭丝、黄心管儿、
    焖白鳝、焖黄鳝、豆豉鲇鱼、锅烧鲤鱼、烀烂甲鱼、抓炒鲤鱼、抓炒对儿虾。"
    + "软炸里脊、软炸鸡、什锦套肠儿、卤煮寒鸦儿、麻酥油卷儿、熘鲜蘑、熘鱼脯、熘鱼肚、熘鱼片儿、醋熘肉片儿、
    烩三鲜、烩白蘑、烩鸽子蛋、炒银丝、烩鳗鱼、炒白虾、炝青蛤、炒面鱼。"
    + "炒竹笋、芙蓉燕菜、炒虾仁儿、烩虾仁儿、烩腰花儿、烩海参、炒蹄筋儿、锅烧海参、锅烧白菜、炸木耳、炒
    肝尖儿、桂花翅子、清蒸翅子、炸飞禽、炸汁儿、炸排骨、清蒸江瑶柱。"
    + "糖熘芡仁米、拌鸡丝、拌肚丝、什锦豆腐、什锦丁儿、糟鸭、糟熘鱼片儿、熘蟹肉、炒蟹肉、烩蟹肉、清拌蟹
    肉、蒸南瓜、酿倭瓜、炒丝瓜、酿冬瓜、烟鸭掌儿、焖鸭掌儿、焖笋、炝茭白。"
    + "茄子晒炉肉、鸭羹、蟹肉羹、鸡血汤、三鲜木樨汤、红丸子、白丸子、南煎丸子；四喜丸子、三鲜丸子、汆丸
    子、鲜虾丸子、鱼脯丸子、饹炸丸子、豆腐丸子、樱桃肉、马牙肉、米粉肉。"
    + "一品肉、栗子肉、坛子肉、红焖肉、黄焖肉、酱豆腐肉、晒炉肉、炖肉、黏糊肉、烀肉、扣肉、松肉、罐儿肉、
    烧肉、大肉、烤肉、白肉、红肘子、白肘子、熏肘子、水晶肘子、蜜蜡肘子。"
    + "锅烧肘子、扒肘条、炖羊肉、酱羊肉、烧羊肉、烤羊肉、清羔羊肉、五香羊肉、汆三样儿、爆三样儿、炸卷果
    儿、烩散丹、烩酸燕儿、烩银丝、烩白杂碎、汆节子、烩节子、炸绣球。"
    + "三鲜鱼翅、栗子鸡、汆鲤鱼、酱汁鲫鱼、活钻鲤鱼、板鸭、筒子鸡、烩脐肚、烩南荠、爆肚仁儿、盐水肘花儿、
    锅烧猪蹄儿、拌稂子、炖吊子、烧肝尖儿、烧肥肠儿、烧心、烧肺。"
    + "烧紫盖儿、烧连帖、烧宝盖儿、油炸肺、酱瓜丝儿、山鸡丁儿、拌海蜇、龙须菜、炝冬笋、玉兰片、烧鸳鸯、
    烧鱼头、烧槟子、烧百合、炸豆腐、炸面筋、炸软巾、糖熘饹儿。"
    + "拔丝山药、糖焖莲子、酿山药、杏仁儿酪、小炒螃蟹、汆大甲、炒荤素儿、什锦葛仙米、鳎目鱼、八代鱼、海
    鲫鱼、黄花鱼、鲫鱼、带鱼、扒海参、扒燕窝、扒鸡腿儿、扒鸡块儿。"
    + "扒肉、扒面筋、扒三样儿、油泼肉、酱泼肉、炒虾黄、熘蟹黄、炒子蟹、炸子蟹、佛手海参、炸烹儿、炒芡子
    米、奶汤、翅子汤、三丝汤、熏斑鸠、卤斑鸠、海白米、烩腰丁儿。"
    + "火烧茨菰、炸鹿尾儿、焖鱼头、拌皮渣儿、汆肥肠儿、炸紫盖儿、鸡丝豆苗、十二台菜、汤羊、鹿肉、驼峰、
```

---

```

    鹿大哈、插根儿、炸花件儿，清拌粉皮儿、炝莴笋、烹芽韭、木樨菜。”
    + “烹丁香、烹大肉、烹白肉、麻辣野鸡、炝酸蓄、熘脊髓、咸肉丝儿、白肉丝儿、荸荠一品锅、素炝春不老、清
    焖莲子、酸黄菜、烧萝卜、脂油雪花儿菜、炝银耳、炒银枝儿。”
    + “八宝榛子酱、黄鱼锅子、白菜锅子、什锦锅子、汤圆锅子、菊花锅子、杂烩锅子、煮饽饽锅子、肉丁辣酱、炒
    肉丝、炒肉片儿、炝酸菜、炝白菜、炝豌豆、焖扁豆、汆毛豆、炒豇豆，外加腌苤蓝丝儿。”,
};

new MemSourceBatchOp(strings, "doc")
    .link(
        new SegmentBatchOp()
            .setSelectedCol("doc")
            .setOutputCol("words")
    )
    .link(
        new StopWordsRemoverBatchOp()
            .setSelectedCol("words")
    )
    .link(
        new KeywordsExtractionBatchOp()
            .setMethod(Method.TEXT_RANK)
            .setSelectedCol("words")
            .setOutputCol("extract_keywords")
    )
    .select("extract_keywords")
    .print();

```

---

菜名文档较长，只保留了头尾；对文档进行分词、停用词过滤处理，然后设置 KeywordsExtractionBatchOp 组件，选择使用 TEXT\_RANK 方法，选择输入的文本数据列（空格分隔的分词结果形式），并设置输出数据列名称。

计算结果如下。

---

extract_keywords
炝 儿 炒 肉 烧 熘 炸 余 焖 丸 子

---

关键词以烹饪方法为主，有“炝、炒、烧、熘、炸、余、焖”，排在第 2 位的是儿化音“儿”；作为重要材料的“肉”排在第 4 位；还有一个关键词“丸子”，也在菜谱中多次与其他词组合出现。

TextRank 算法更适合用在长文本中，比如本章中的新闻标题数据，使用 TF-IDF 指标进行判断更为有效。与前面的示例类似，对文档进行分词、停用词过滤处理，然后设置 KeywordsExtractionBatchOp 组件，选择使用 TF\_IDF 方法，设置最多输出关键词的个数为 5，选择输入的文本数据列（空格分隔的分词结果形式），并设置输出数据列的名称。

---

```

getSource()
    .link(
        new SegmentBatchOp()
            .setSelectedCol("news_title")

```

---

```

        .setOutputCol("segmented_title")
    )
.link(
    new StopWordsRemoverBatchOp()
    .setSelectedCol("segmented_title")
)
.link(
    new KeywordsExtractionBatchOp()
    .setTopN(5)
    .setMethod(Method.TF_IDF)
    .setSelectedCol("segmented_title")
    .setOutputCol("extract_keywords")
)
.select("news_title, extract_keywords")
.firstN(10)
.print();

```

运行结果如表 21-17 所示，右边一列为提取出来的关键词。

表 21-17 关键词提取

news_title	extract_keywords
京城最值得你来场文化之旅的博物馆	来场 京城 博物馆 之旅 文化
发酵床的垫料种类有哪些？哪种更好？	垫料 种类 床 发酵 哪种
上联：黄山黄河黄皮肤黄土高原。怎么对下联？	黄皮肤 黄土高原 黄河 黄山 下联
林徽因什么理由拒绝了徐志摩而选择梁思成成为终身伴侣？	终身伴侣 梁思成 徐志摩 林徽因 理由
黄杨木是什么树？	黄杨木 树
上联：草根登上星光道，怎么对下联？	草根 星光 登上 道 下联
什么是超写实绘画？	写实 绘画 超
松涛听雨莺婉转，下联？	听雨莺 婉转 松涛 下联
上联：老子骑牛读书，下联怎么对？	骑牛 老子 读书 下联 上联
上联：山水醉人何须酒。如何对下联？	何须 醉人 山水 酒 下联

## 21.6 文本相似度

文本不能像数字那样被定义大小，但是我们可以看出哪些文本更接近，这种“接近”是通过文本间的距离和相似度来衡量的。常用的有编辑距离（Levenshtein 距离）、最长公共子串（Longest Common Substring，LCS）距离、余弦相似度、Jaccard 相似度等。

本节会使用编辑距离进行文本比较。编辑距离是由苏联科学家 Vladimir Levenshtein 在 1965 年提出的，故又叫 Levenshtein 距离，指的是在两个字符串之间，由一个字符串转换成另一个字符串所需要的最少的编辑操作次数。

将编辑操作定义为以下 3 种类型。

- (1) 插入一个字符。
- (2) 删除一个字符。
- (3) 将一个字符替换成另一个字符。

为了加深对此距离的理解，举例如下。

- ab 和 ab 需要有 0 个操作。字符串同理。
- ab 和 a 需要有 1 个删除操作。
- a 和 ab 需要有 1 个插入操作。
- ab 和 ac 需要有 1 个替换操作。

编辑距离相似度的公式如下，即单位 1 减去编辑距离与最大字符串长度的比值。

$$\text{Simi}_{\text{Edit}} = 1 - \frac{\text{EditDistance}}{\max(\text{Length1}, \text{Length2})}$$

### 21.6.1 文本成对比较

本小节使用的数据为构造的 5 对文本数据，如表 21-18 所示。

表 21-18 构造的 5 对文本数据

col1	col2
机器学习	机器学习
批式计算	流式计算
Machine Learning	ML
Flink	Alink
Good Morning!	Good Evening!

如下代码所示，将要比较的两列数据分别命名为 col1 和 col2，并分别构造批式数据源和流式数据源。

---

```
Row[] rows = new Row[] {
    Row.of("机器学习", "机器学习"),
    Row.of("批式计算", "流式计算"),
```

---

```

Row.of("Machine Learning", "ML"),
Row.of("Flink", "Alink"),
Row.of("Good Morning!", "Good Evening!")
};

MemSourceBatchOp source = new MemSourceBatchOp(rows, new String[] {"col1", "col2"});
MemSourceStreamOp source_stream = new MemSourceStreamOp(rows, new String[] {"col1", "col2"});

```

以字符为单位来比较字符串，可以使用字符串相似度成对比较组件 `StringSimilarityPairwiseBatchOp`，就是将文本看作字符串，以字符为单位进行比较，如下代码所示。

```

source
.link(
    new StringSimilarityPairwiseBatchOp()
        .setSelectedCols("col1", "col2")
        .setMetric("LEVENSHTEIN")
        .setOutputCol("LEVENSHTEIN")
)
...
.link(
    new StringSimilarityPairwiseBatchOp()
        .setSelectedCols("col1", "col2")
        .setMetric("JACCARD_SIM")
        .setOutputCol("JACCARD_SIM")
)
.lazyPrint(-1, "\n## StringSimilarityPairwiseBatchOp ##");

```

调用该组件 5 次，每次都换一种度量方式，并根据选择的度量方式命名结果数据列。

计算结果如表 21-19 所示。

表 21-19 各种句子的相似度指标

col1	col2	LEVENSHTEIN	LEVENSHTEIN_SIM	LCS	LCS_SIM	JACCARD_SIM
机器学习	机器学习	0.0000	1.0000	4.0000	1.0000	1.0000
批式计算	流式计算	1.0000	0.7500	3.0000	0.7500	0.6000
Machine Learning	ML	14.0000	0.1250	2.0000	0.1250	0.1250
Flink	Alink	1.0000	0.8000	4.0000	0.8000	0.6667
Good Morning!	Good Evening!	3.0000	0.7692	10.0000	0.7692	0.6250

可以观察到如下信息。

- LEVENSHTEIN 和 LCS 计算的是距离。
- LEVENSHTEIN\_SIM、LCS\_SIM 和 JACCARD\_SIM，即以“\_SIM”结尾的为相似度指标，取值范围为[0, 1]。

以单词为单位来比较字符串，可以使用文本相似度成对比较组件 TextSimilarityPairwiseBatchOp，就是将文本看作由单词构成且以单词为单位的，如下代码所示。

```

source
    .link(
        new SegmentBatchOp()
            .setSelectedCol("col1")
    )
    .link(
        new SegmentBatchOp()
            .setSelectedCol("col2")
    )
    .link(
        new TextSimilarityPairwiseBatchOp()
            .setSelectedCols("col1", "col2")
            .setMetric("LEVENSHTEIN")
            .setOutputCol("LEVENSHTEIN")
    )
    ...
    .link(
        new TextSimilarityPairwiseBatchOp()
            .setSelectedCols("col1", "col2")
            .setMetric("JACCARD_SIM")
            .setOutputCol("JACCARD_SIM")
    )
    .lazyPrint(-1, "\n## TextSimilarityPairwiseBatchOp ##");

```

首先进行分词操作，由于分词组件每次只能处理一列，所以分两次进行调用，分别处理 col1 和 col2。调用该组件 5 次，每次都换一种度量方式，并根据选择的度量方式命名结果数据列。

计算结果如表 21-20 所示。

表 21-20 各种文本的相似度指标

col1	col2	LEVENSHTEIN	LEVENSHTEIN_SIM	LCS	LCS_SIM	JACCARD_SIM
机器 学习	机器 学习	0.0000	1.0000	2.0000	1.0000	1.0000
批式 计算	流式 计算	1.0000	0.5000	1.0000	0.5000	0.3333
machine learning	ml	4.0000	0.0000	0.0000	0.0000	0.0000
flink	alink	1.0000	0.0000	0.0000	0.0000	0.0000
good morning !	good evening !	1.0000	0.8000	4.0000	0.8000	0.6667

可以观察到如下信息。

- 由于以单词为基本单位进行比较，每对文本包含的元素个数大幅下降。
- 从单词角度来看，fline 与 alink 是不同的，3 个相似度指标都为 0，但是它们的大部分字符是相同的，所以在以字符为单位进行计算时，3 个相似度指标都是比较高的。

流式数据的处理与批式类似，要使用相应的流式字符串相似度成对比较组件 StringSimilarityPairwiseStreamOp，如下代码所示。

```
source_stream
    .link(
        new StringSimilarityPairwiseStreamOp()
            .setSelectedCols("col1", "col2")
            .setMetric("LEVENSHTEIN")
            .setOutputCol("LEVENSHTEIN")
    )
    ...
    .link(
        new StringSimilarityPairwiseStreamOp()
            .setSelectedCols("col1", "col2")
            .setMetric("JACCARD_SIM")
            .setOutputCol("JACCARD_SIM")
    )
    .print();
StreamOperator.execute();
```

调用该组件 5 次，每次都换一种度量方式，并根据选择的度量方式命名结果数据列。

计算结果输出如表 21-21 所示，对比批式计算结果，在数值上是一样的。

表 21-21 流式相似度的计算结果

col1	col2	LEVENSHTEIN	LEVENSHTEIN_SIM	LCS	LCS_SIM	JACCARD_SIM
Machine Learning	ML	14.0000	0.1250	2.0000	0.1250	0.1250
Flink	Alink	1.0000	0.8000	4.0000	0.8000	0.6667
批式计算	流式计算	1.0000	0.7500	3.0000	0.7500	0.6000
Good Morning!	Good Evening!	3.0000	0.7692	10.0000	0.7692	0.6250
机器学习	机器学习	0.0000	1.0000	4.0000	1.0000	1.0000

## 21.6.2 最相似的TopN

给定一个文本，如何从若干文本中找到 N 个最相近的那些文本，就是本小节关注的问题。

Alink 提供了字符串最相似 TopN 计算组件 StringNearestNeighbor（以字符为单位，将文本数据看作字符串）和 TextNearestNeighbor（以单词为单位，将文本数据看作文档）。为了提高

计算效率，会对作为对比的原始数据进行一些预算操作，并保留预算计算的结果，我们也可将此看作对原始数据的“训练”，而预算计算结果就是“模型”。

从原始数据中取出 4 条新闻标题，构造目标数据集，具体代码如下。

```
Row[] rows = new Row[] {
    Row.of("林徽因什么理由拒绝了徐志摩而选择梁思成成为终身伴侣?"),
    Row.of("发酵床的垫料种类有哪些？哪种更好？"),
    Row.of("京城最值得你来场文化旅游的博物馆"),
    Row.of("什么是超写实绘画?"),
};

MemSourceBatchOp target = new MemSourceBatchOp(rows, new String[] {TXT_COL_NAME});

BatchOperator <?> source = getSource();
```

## 1. 基本算法

我们以字符为单位，将文本数据看作字符串，使用 StringNearestNeighbor 组件进行计算。该组件需要输入两个数据，一个是文本数据所在的列，另一个是用来显示结果的，直接罗列各个相似文本。如果每个文本较长，结果会比较臃肿，所以单独提供了参数 IdCol，用于设置文本对应的 ID 列，计算结果会显示 ID 信息。鉴于本章的文本数据为新闻标题信息，比较短小，所以 IdCol 列仍然设置为标题列，便于我们查看结果。

```
for (String metric : new String[] {"LEVENSHTEIN", "LCS", "SSK", "COSINE"}) {
    new StringNearestNeighbor()
        .setMetric(metric)
        .setSelectedCol(TXT_COL_NAME)
        .setIdCol(TXT_COL_NAME)
        .setTopN(5)
        .setOutputCol("similar_titles")
        .fit(source)
        .transform(target)
        .lazyPrint(-1, "StringNearestNeighbor " + metric.toString());
    BatchOperator.execute();
}
```

计算结果如表 21-22 所示，左边一列为 StringNearestNeighbor 组件的度量参数 metric 值。

表 21-22 字符串最近邻计算结果

metric 值	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣?
String : LEVENSHTEIN	{"ID":"\[\"林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣?\",\"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么?\",.....]","METRIC":"[1.0,12.0,19.0,20.0,20.0]"} .....]

续表

metric 值	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？
String : LCS	{"ID":"\[\"林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\",\"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？\",.....,\"NBA 球员因什么打篮球？他们理由各不同，邓肯因怕鲨鱼而打球\",\".....]","METRIC":"[24.0,19.0,6.0,6.0,6.0]"}]
String : SSK	{"ID":"\[\"林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\",\"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？\",.....]","METRIC":"[0.9786452262557865,0.7201100809120543,0.17344669301981408,0.17344669301981408]"}]
String : COSINE	{"ID":"\[\"林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\",\"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？\",.....]","METRIC":"[0.9789450103725609,0.7089490077940542,0.18860838403857944,0.16718346377260584,0.16718346377260584]"}]

可以观察到如下信息。

- 原数据中包含这个新闻标题，每个算法都找到了完全一致的那个文本，并且排在了第一位。
- “林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？”是目标文本的一个改写，意思相同，大部分文字也相同，各个算法都把它找出来了，而且都排在了第二位。这也说明了各种算法的有效性。
- 对于第三个及后面的选项，各算法的差异很大。

我们再以单词为单位，将文本数据进行分词，再使用 TextNearestNeighbor 组件进行计算。与 StringNearestNeighbor 组件类似，需要输入两个数据，一个是文本数据所在的列，另一个是 IdCol 列名称。

---

```
for (String metric : new String[] {"LEVENSHTEIN", "LCS", "SSK", "COSINE"}) {
    new Pipeline()
        .add(
            new Segment()
                .setSelectedCol(TXT_COL_NAME)
                .setOutputCol("segmented_title")
        )
        .add(
            new TextNearestNeighbor()
                .setMetric(metric)
                .setSelectedCol("segmented_title")
                .setIdCol(TXT_COL_NAME)
                .setTopN(5)
```

---

```

        .setOutputCol("similar_titles")
    )
    .fit(source)
    .transform(target)
    .lazyPrint(-1, "TextNearestNeighbor + " + metric.toString());
BatchOperator.execute();
}

```

运行结果如表 21-23 所示，左边一列为 TextNearestNeighbor 组件的度量参数 metric 值。

表 21-23 文本最近邻计算结果

metric 值	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？
Text : LEVENSHTEIN	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？ \", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？ \", \"什么造就了今年的季后赛隆多\", \"是什么因素阻止了伊涅斯塔加盟中超？ \", \"是什么让你选择留在鄂州？ \"], \"METRIC\": [1.0, 8.0, 9.0, 9.0, 9.0]}
Text : LCS	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？ \", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？ \", .....\", \"小米就要港股上市了，那么为什么选择香港而没有选择上海？ \"], \"METRIC\": [11.0, 8.0, 3.0, 3.0, 3.0]}
Text : SSK	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？ \", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？ \", .....], \"METRIC\": [0.9519716503395643, 0.5072036680937279, 0.1021051630028303, 0.08756718614037064, 0.08194015972362995]}
Text : COSINE	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？ \", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？ \", .....], \"METRIC\": [0.9534625892455924, 0.45643546458763845, 0.1318760946791574, 0.12909944487358055, 0.10540925533894598]}

可以观察到如下信息。

- 排在第一、二位的结果都一致，与前面基于字符计算得到的前两名结果也一样，说明了各种算法的有效性。
- 在第三位及后面的选项中，各算法差异明显，而且与基于字符计算的结果差异更大。“林徽因”作为一个单词或者三个字符，在相似度计算中所占的比重有很大差别，所以在以单词为单位的 TopN 计算结果中，和“林徽因”相关的其他标题没有排到前五位。

我们再将“京城最值得你来场文化之旅的博物馆”相关的结果汇总起来，如表 21-24 所示，左边一列为所用组件（StringNearestNeighbor 或者 TextNearestNeighbor）的度量参数 metric 值。

表 21-24 最近邻计算结果

metric 值	京城最值得你来场文化之旅的博物馆
String : LEVENSHTEIN	{"ID": "["京城最值得你来场文化之旅的博物馆", "全国教师文化之旅 第一季", "《刮痧》文化之间的差异", "青岛长沙路小学学生走进海关博物馆", "中国传统文化之楼阁"]", "METRIC": "[0.0,12.0,12.0,13.0,13.0]"}
String : LCS	{"ID": "["京城最值得你来场文化之旅的博物馆", "第十四届文博会书画艺术文化之旅活动在深圳合正艺术博物馆开幕", "除了故宫,北京还有哪些值得去的博物馆?", "“金城文化名家”王作宝丝绸之路风情画 被敦煌市博物馆收藏", "彰显着青海深厚的历史文化的博物馆都在这了!"]", "METRIC": "[16.0,7.0,7.0,7.0,6.0]"}]
String : SSK	{"ID": "["京城最值得你来场文化之旅的博物馆", "杭州有哪些值得参观的博物馆?", "吐鲁番博物馆, 最值得看的文物有哪些?", "青州 5A 级景区值得你来", "彰显着青海深厚的历史文化的博物馆都在这了!"]", "METRIC": "[1.0,0.25123248562505274,0.23162579414326565,0.22439441671450996,0.22091900671835765]"}]
String : COSINE	{"ID": "["京城最值得你来场文化之旅的博物馆", "杭州有哪些值得参观的博物馆?", "吐鲁番博物馆, 最值得看的文物有哪些?", "青州 5A 级景区值得你来", "超级链接的博物馆: 93 项活动亮相北京 518 博物馆日"]", "METRIC": "[1.0,0.2864459496157732,0.2504897164340598,0.24494897427831783,0.2439750182371333]"}]
Text : LEVENSHTEIN	{"ID": "["京城最值得你来场文化之旅的博物馆", "教你判断 P2P 平台的背景", "庄浪; 你是人间五月的天堂", "诗词与企业文化的关系", "圣迭戈有哪些地方你不容错过的?"]", "METRIC": "[0.0,7.0,7.0,7.0,7.0]"}]
Text : LCS	{"ID": "["京城最值得你来场文化之旅的博物馆", "6 款最值得买的合资品牌 SUV 全在这儿, 有没有你等的?", "美媒报道中国 40 个最美丽最值得游览的景点, 这里有你喜欢的吗?", "初夏, 最值得你去的旅行地——安康, 有 4 处神秘之地, 你去过几个?", "世界上最奇葩的水果, 你未必都见过, 吃过其中的一种我就服你!"]", "METRIC": "[9.0,4.0,4.0,4.0,3.0]"}]
Text : SSK	{"ID": "["京城最值得你来场文化之旅的博物馆", "初夏, 最值得你去的旅行地——安康, 有 4 处神秘之地, 你去过几个?", "彰显着青海深厚的历史文化的博物馆都在这了!", "最值得购买的 5 款中级轿车", "最值得购买的 5 款 MPV"]", "METRIC": "[1.0,0.1313804715200644,0.13063966852069117,0.11823132268957977]"}]

续表

metric 值	京城最值得你来场文化之旅的博物馆
Text : COSINE	{"ID": "[\"京城最值得你来场文化之旅的博物馆\", \"初夏，最值得你去的旅行地——安康，有 4 处神秘之地，你去过几个\", \"广西什么地方最值得去旅游\", \"兴义哪个景色最值得去？\", \"最值得购买的 5 款中级轿车\"]", "METRIC": "[1.0,0.1543033499620919,0.14433756729740646,0.14433756729740646,0.14433756729740646]"}}

可以观察到如下信息。

- 针对原数据中包含的这个新闻标题，每个算法都找到了与其完全一致的那个文本，并且排在了第一位。
- 在以字符为单位的 Top5 结果中，以“文化之旅”和“博物馆”为主；在以单位为单位的 Top5 结果中，以“最值得”为主。

最后看一下标题“什么是超写实绘画？”相关的 Top5 结果汇总情况，如表 21-25 所示，StringNearestNeighbor 或者 TextNearestNeighbor 的结果差异小了一些。

表 21-25 最近邻计算结果

metric 值	什么是超写实绘画？
String : LEVENSHTEIN	{"ID": "[\"什么是超写实绘画？\", \"什么是文人画？\", \"什么是区块链？\", \"什么是区块链？\", \"什么是爱情？\"]", "METRIC": "[0.0,4.0,5.0,5.0,5.0]"}}
String : LCS	{"ID": "[\"什么是超写实绘画？\", \"超写实绘画只是在描绘现实吗？\", \"QQ 飞车手游中什么段位是实力的分界线？\", \"你怎么看吕建军的写实油画？\", \"QQ 飞车手游中什么段位是实力的分界线？\"]", "METRIC": "[9.0,6.0,5.0,5.0,5.0]"}}
String : SSK	{"ID": "[\"什么是超写实绘画？\", \"超写实绘画只是在描绘现实吗？\", \"什么是文人画？\", \"解答：什么是超载？\", \"什么是人情，什么是世故？\"]", "METRIC": "[1.0,0.3911627417302326,0.3727231335822584,0.3525653656235633,0.3268835910175176]"}}
String : COSINE	{"ID": "[\"什么是超写实绘画？\", \"什么是文人画？\", \"超写实绘画只是在描绘现实吗？\", \"解答：什么是超载？\", \"什么是人情，什么是世故？\"]", "METRIC": "[1.0,0.43301270189221935,0.3922322702763681,0.375,0.3651483716701107]"}}
Text : LEVENSHTEIN	{"ID": "[\"什么是超写实绘画？\", \"什么是区块链？\", \"什么是区块链？\", \"什么是爱情？\", \"什么是伊朗核协议？\"]", "METRIC": "[0.0,3.0,3.0,3.0,3.0]"}}

续表

metric 值	什么是超写实绘画？
Text : LCS	{"ID": "["什么是超写实绘画？", "超写实绘画只是在描绘现实吗？", "什么是方法论？怎么掌握方法论？", "什么是以租代购？以租代购过户吗？车子什么有保障吗？", "什么是混改，为什么要联通混改，而不是电信或者移动呢？"], "METRIC": "[6.0,4.0,3.0,3.0,3.0]"}]
Text : SSK	{"ID": "["什么是超写实绘画？", "超写实绘画只是在描绘现实吗？", "什么是四维空间？", "什么是教育？", "什么是化性"], "METRIC": "[1.0,0.2872548320403265,0.2672519210676725,0.26725192106725,0.2672519210676725]"}]
Text : COSINE	{"ID": "["什么是超写实绘画？", "超写实绘画只是在描绘现实吗？", "什么是四维空间？", "什么是教育？", "什么是化性"], "METRIC": "[1.0,0.31622776601683794,0.31622776601683794,0.31622776601683794,0.31622776601683794]"}]

## 2. 近似算法

文本相似度算法虽然能获得较好的分类效果，但是计算时间较长。在实际的大规模文本比较的场景中，我们常常希望在效率和效果中有所折中，那么近似相似度就派上了用场。

Alink 提供了 StringApproxNearestNeighbor 组件，以字符为单位，可以选择多种近似计算相似 TopN 的方式。与 StringNearestNeighbor 组件类似，需要输入两个数据，一个是文本数据所在的列，另一个是 IdCol 列名称。如下代码所示，循环尝试各种近似相似度计算。

```
for (String metric : new String[] {"JACCARD_SIM", "MINHASH_JACCARD_SIM", "SIMHASH_HAMMING_SIM",}) {
    new StringApproxNearestNeighbor()
        .setMetric(metric)
        .setSelectedCol(TXT_COL_NAME)
        .setIdCol(TXT_COL_NAME)
        .setTopN(5)
        .setOutputCol("similar_titles")
        .fit(source)
        .transform(target)
        .lazyPrint(-1, "StringApproxNearestNeighbor + " + metric.toString());
    BatchOperator.execute();
}
```

Alink 提供了 TextApproxNearestNeighbor 组件，以单词为单位，可以选择多种近似计算相似 TopN 的方式。与 TextNearestNeighbor 组件类似，需要先将文本数据进行分词，该组件需要输入两个数据，一个是文本数据所在的列，另一个是 IdCol 列名称。具体代码如下，尝试各种

近似度量，将输出样例结果进行比较。

```

for (String metric : new String[] {"JACCARD_SIM", "MINHASH_JACCARD_SIM", "SIMHASH_HAMMING_SIM"}) {
    new Pipeline()
        .add(
            new Segment()
                .setSelectedCol(TXT_COL_NAME)
                .setOutputCol("segmented_title")
        )
        .add(
            new TextApproxNearestNeighbor()
                .setMetric(metric)
                .setSelectedCol("segmented_title")
                .setIdCol(TXT_COL_NAME)
                .setTopN(5)
                .setOutputCol("similar_titles")
        )
        .fit(source)
        .transform(target)
        .lazyPrint(-1, "TextApproxNearestNeighbor + " + metric.toString());
    BatchOperator.execute();
}

```

我们先将新闻标题“林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？”在上面两个组件和各种度量情况下的计算结果汇总如表 21-26 所示。

表 21-26 近似最近邻计算结果

metric 值	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？
String : JACCARD_SIM	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？\", .....\", \"林黛玉为什么忽然成了肺结核\\n\"]", "METRIC": "[0.96, 0.7241379310344828, 0.20588235294117646, 0.2, 0.1935483870967742]"}]
String : MINHASH_JACCARD_SIM	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\", \"林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？\", .....\", \"绝地求生：蓝战非终于完成了摩托车特技？摩托车脾气上来直接自爆\", \"中国股市终于有人把布林线指标浓缩成精髓了，少走二十年弯路\\n\"]", "METRIC": "[1.0, 0.9, 0.4, 0.4, 0.3]"}]
String : SIMHASH_HAMMING_SIM	{"ID": "["林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？\", \"迷你世界愤怒的小鸟\", \"《极限挑战》除了张艺兴还缺席了一个人，但观众却都很高兴\", \"扬州一中学探索因材施教 老师分层次布置作业\", \"忽而今夏：白宇为她设计了一款游戏，女生最羡慕的异地恋游戏！\\n\"]", "METRIC": "[0.953125, 0.921875, 0.90625, 0.90625, 0.890625]"}]

续表

metric 值	林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？
Text : JACCARD_SIM	{"ID": "T", "林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？", "林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？", "小米香港上市将为小米带来了什么？", "公务员考试失利后你选择了做什么？", "OPPO 为迎接 5G 时代做了些什么？"}, "METRIC": "[0.9166666666666666, 0.6, 0.1666666666666666, 0.1666666666666666, 0.1666666666666666]"}]
Text : MINHASH_JACCARD_SIM	{"ID": "T", "林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？", "林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？", "孝顺儿子接我进城养老，准备走时接了个电话，我找个理由不去了", "汽车停产理由多，而这几款车，只因长得太丑就被停产了"}, "METRIC": "[1.0, 0.7, 0.3, 0.3, 0.3]"}]
Text : SIMHASH_HAMMING_SIM	{"ID": "T", "林徽因什么理由拒绝了徐志摩而选择梁思成为终身伴侣？", "日本为什么不敢研发干线客机？俄军专家：发动机不仅要买且被锁死", "果农注意了，近期柑橘溃疡病大爆发！", "重庆市民投资对象变化：理财产品成首选 房地产退居第四"}, "METRIC": "[0.9375, 0.78125, 0.765625, 0.75, 0.734375]"}]

对于原始数据包含的同样的标题，在各种度量下都能被找出，并排在第一位。而对与之相近的标题“林徽因拒绝徐志摩而选择梁思成为终身伴侣的原因是什么？”，在多数度量中都排进了前五位。

再看另一个标题“什么是超写实绘画？”，汇总结果见表 21-27。

表 21-27 近似最近邻计算结果

metric 值	什么是超写实绘画？
String : JACCARD_SIM	{"ID": "T", "什么是超写实绘画？", "超写实主义画作存在的意义是什么？", "什么是文人画？", "超写实绘画只是在描绘现实吗？", "鲁本斯绘画的长处是什么？"}, "METRIC": "[1.0, 0.47058823529411764, 0.454545454545453, 0.4375, 0.4]"}]
String : MINHASH_JACCARD_SIM	{"ID": "T", "什么是超写实绘画？", "为什么子弹超音速时没有音爆？", "什么是妖股？", "什么是财务自由？", "合肥的市花是什么？"}, "METRIC": "[1.0, 0.6, 0.6, 0.6, 0.6]"}]
String : SIMHASH_HAMMING_SIM	{"ID": "T", "什么是超写实绘画？", "叙利亚的军事实力如何？", "安徽省临近江浙沪，为什么发展却跟不上？", "电子维修有前途吗？", "郴州有哪些比较好的技校？"}, "METRIC": "[1.0, 0.9375, 0.90625, 0.90625, 0.90625]"}]

续表

metric 值	什么是超写实绘画？
Text : JACCARD_SIM	{"ID": "[\"什么是超写实绘画？\", \"鲁本斯绘画的长处是什么？\", \"什么是咖喱？\", \"什么是都江堰？\", \"什么是妖股？\", \"\"]", "METRIC": "[1.0, 0.4444444444444444, 0.42857142857142855, 0.42857142857142855, 0.42857142855]"}]
Text : MINHASH_JACCARD_SIM	{"ID": "[\"什么是超写实绘画？\", \"什么是妖股？\", \"web 服务器是什么？\", \"什么是都江堰？\", \"什么是子网？什么是子网掩码？\", \"\"]", "METRIC": "[1.0, 0.7, 0.7, 0.7, 0.7]"}]
Text : SIMHASH_HAMMING_SIM	{"ID": "[\"什么是超写实绘画？\", \"不同国家的军人在一起，军衔还论高低吗？\", \"现代海战中，能否俘获敌方完整战舰？已经绝无可能？\", \"共享单车好骑吗？\", \"刀郎的徒弟是谁？\", \"\"]", "METRIC": "[1.0, 0.84375, 0.84375, 0.84375, 0.84375]"}]

### 3. 计算性能

本节将分别选取计算近似与非近似相似度算法来测试模型训练时间，并针对不同数据量的测试预测时间，以便读者对算法性能有一个初步的印象，便于在实际使用中选择合适的方法。

非近似算法组件 StringNearestNeighbor 选择 LEVENSHTEIN（编辑距离）度量，近似算法组件 StringApproxNearestNeighbor 选择 JACCARD\_SIM 度量。构建 Pipeline 的代码如下。

```
Pipeline snn = new Pipeline()
    .add(
        new StringNearestNeighbor()
            .setMetric("LEVENSHTEIN")
            .setSelectedCol(TXT_COL_NAME)
            .setIdCol(TXT_COL_NAME)
            .setTopN(5)
            .setOutputCol("similar_titles")
    );

Pipeline approx_snn = new Pipeline()
    .add(
        new StringApproxNearestNeighbor()
            .setMetric("JACCARD_SIM")
            .setSelectedCol(TXT_COL_NAME)
            .setIdCol(TXT_COL_NAME)
            .setTopN(5)
            .setOutputCol("similar_titles")
    );
```

计算非近似算法模型训练的时间，代码如下，将模型保存到文件路径。

---

```
static final String SNN_MODEL_FILE = "snn_model.ak";
...
snn.fit(source)
    .save(DATA_DIR + SNN_MODEL_FILE);
BatchOperator.execute();
```

---

得到任务执行时间为 12.587 秒，模型文件大小为 52.7MB。

计算近似算法模型训练的时间，代码如下，将模型保存到文件路径。

---

```
static final String APPROX_SNN_MODEL_FILE = "approx_snn_model.ak";
...
approx_snn
    .fit(source)
    .save(DATA_DIR + APPROX_SNN_MODEL_FILE);
BatchOperator.execute();
```

---

任务执行时间为 21.218 秒，模型文件大小为 356.7MB。

下面我们看预测时间。首先构造两个预测集，希望数据量适中，既可以演示差异，也不会运行时间过长。在 21.1 节中，我们知道各个新闻类别的数据条数，其中新闻类“stock”有 340 条，新闻类“news\_story”有 6273 条。我们可以分别将其过滤出来作为测试集，具体代码如下。

---

```
BatchOperator <?> target_stock = source.filter("category_name = 'stock'");
BatchOperator <?> target_news_story = source.filter("category_name = 'news_story'");
```

---

测试非近似算法模型对 340 条 stock 数据的预测时间，代码如下。

---

```
PipelineModel
    .load(DATA_DIR + SNN_MODEL_FILE)
    .transform(target_stock)
    .lazyPrint(10, "StringNearestNeighbor + LEVENSHTEIN");
BatchOperator.execute();
```

---

运行时间为 4 分 35 秒。

测试近似算法模型对 340 条 stock 数据的预测时间，代码如下。

---

```
PipelineModel
    .load(DATA_DIR + APPROX_SNN_MODEL_FILE)
    .transform(target_stock)
    .lazyPrint(10, "MINHASH_SIM + stock");
BatchOperator.execute();
```

---

运行时间为 33 秒，去除模型加载的时间，实际预测的时间会更短。我们可以尝试更大的数据集。

测试非近似算法模型对 6273 条 news\_story 数据的预测时间，代码如下。

```
PipelineModel
    .load(DATA_DIR + APPROX_SNN_MODEL_FILE)
    .transform(target_news_story)
    .lazyPrint(10, "MINHASH_SIM + news_story");
BatchOperator.execute();
```

运行时间为 2 分 59 秒。

综上，使用近似文本相似算法可以在预测阶段大幅提升性能。

## 4. 流式预测

我们将做两个流式预测实验，一个是使用非近似算法模型预测 4 条数据，与批式结果相比，看看结果是否相同；另一个是使用近似算法模型预测 stock 数据，并将流式预测的速度与批式预测相比较。

沿用前面实验中的 4 条新闻标题，使用组件 `MemSourceStreamOp` 构造流式数据源，通过 `PipeModel` 加载非近似算法模型文件，然后对流式数据源进行预测，输出预测结果，便于和前面批式预测的结果进行对比。具体代码如下。

```

Row[] rows = new Row[] {
    Row.of("林徽因什么理由拒绝了徐志摩而选择梁思成成为终身伴侣？"),
    Row.of("发酵床的垫料种类有哪些？哪种更好？"),
    Row.of("京城最值得你来场文化之旅的博物馆"),
    Row.of("什么是超写实绘画？")
};
StreamOperator <?> stream_target
= new MemSourceStreamOp(rows, new String[] {TXT_COL_NAME});

PipelineModel
.load(DATA_DIR + SNN_MODEL_FILE)
.transform(stream_target)
.print();
StreamOperator.execute();

```

运行结果如下，与前面计算的批式预测结果完全一致。

news title|similar titles

林徽因什么理由拒绝了徐志摩而选择梁思成成为终身伴侣 | {"ID": "\\\\"林徽因什么理由拒绝了徐志摩而选择梁思成成为终身伴侣？\\\""}  
林徽因拒绝徐志摩而选择梁思成成为终身伴侣的原因是什么？

\", .....]\", \"METRIC\":[1,0,12,0,19,0,20,0,20,0]}]

发酵床的垫料种类有哪些？哪种更好？ | {"ID": "\\"发酵床的垫料种类有哪些？哪种更好？\\", \"发酵饲料有哪些好处\\", \"\\教育的分类有哪些？\\", \"\\烤猪蹄的佐料有哪些？\\", \"\\普洱茶的原料有哪些讲究？\\", \"\\"}]

```
\"]", "METRIC": "[0, 0, 10, 0, 11, 0, 11, 0, 11, 0]"}]
```

京城最值得你来场文化之旅的博物馆 | {"ID": "\\"京城最值得你来场文化之旅的博物馆\\", \"全国教师文化之旅 第一季\", \"\\《刮痧》文化之间的差异\\", \"\\青岛长沙路小学学生走进海关博物馆\\", \"\\中国传统文化之楼阁

```
\"]", "METRIC": "[0.0, 12.0, 12.0, 13.0, 13.0]"}  
什么是超写实绘画? | {"ID": "\\"什么是超写实绘画?\", \\"什么是文人画?\", \\"什么是区块链?\", \\"什么是区块链?\", \\"什么是爱情?\"]", "METRIC": "[0.0, 4.0, 5.0, 5.0, 5.0]"}  
什么是超写实绘画?
```

我们再用流的方式读取原始数据文件，并对流式数据进行过滤，保留类别为“stock”的数据作为流式预测的数据。利用 PipeModel 加载近似算法模型文件，然后对流式数据源进行预测，由于预测条数较多，因此进行了流式采样操作（采样率为 0.02），随后输出采样的预测结果。具体代码如下。

---

```
StreamOperator.setParallelism(1);  
... ...  
StreamOperator <?> stream_target_stock  
= getStreamSource().filter("category_name = 'stock'");  
  
sw.reset();  
sw.start();  
PipelineModel  
.load(DATA_DIR + APPROX_SNN_MODEL_FILE)  
.transform(stream_target_stock)  
.sample(0.02)  
.print();  
StreamOperator.execute();  
sw.stop();  
System.out.println(sw.getElapsedSpan());
```

---

注意，由于近似算法模型很大，而每个流式预测 worker 都会保留一份完整模型，这里设置流式任务的并发度为 1，即只有一个 worker 进行预测；由于构造的流式数据量有限，预测完成后会自动终止流式任务，在流式任务的执行前后设置了计时，统计任务运行时间。

这里选取一条输出的新闻标题“三大股指暴跌，后市该如何走？”，预测的相似标题结果如下。

```
{"ID": "\\"三大股指暴跌，后市该如何走?\", \\"周三（3月28日）股市大跌，明天如何走?\", \\"大盘筑底，个股冷不丁暴跌，该如何选股?\", \\"股票开盘跌停，该如何卖出?\", \\"央行加息，股市应该如何投资?\"]", "METRIC": "[1.0, 0.4166666666666667, 0.375, 0.35, 0.3333333333333333]"}  
什么是超写实绘画?
```

流式任务的总运行时间为 1 分 7 秒，考虑到其流式任务的并发度为 1，而对应的批式预测任务的并发度为 2，运行时间为 35 秒。可以看出，批式和流式方式对于单条数据的预测速度差不多是一样的。

## 21.7 主题模型

本章所用的数据集是新闻文档的集合，虽然每篇新闻的内容不同，但是我们看过若干篇后，

会发现有些新闻讲的是同一方面的事情，譬如都是关于电影的或者足球意大利甲级联赛的，我们称之为“主题”。本节要讲的内容，是让计算机直接从文本内容上计算其中有多少主题。为了控制主题的粒度，我们使用一个参数，即指定主题的总个数。

计算主题模型(Topic Model)有几个经典的方法：潜在语义分析(Latent Semantic Analysis, LSA)、概率潜在语义分析(Probabilistic Latent Semantic Analysis, PLSA)，以及本节重点介绍的潜在狄里克莱分布(Latent Dirichlet Allocation, LDA)。这些方法的共同点是，我们会忽略每个文档中单词出现的顺序，文档是由出现了哪些单词、每个单词出现的个数决定的，即词袋(bag of words)。

### 21.7.1 LDA模型

LDA 模型将贝叶斯估计的思想引入话题模型，使得参数具备了概率分布。LDA 是关于多个参数的分布，为了使读者容易理解此模型，我们先介绍简单的情形，即两个参数的情形，与此相关的两个分布分别是二项分布和 Beta 分布。

#### 1. 二项分布 (Binomial Distribution)

二项分布即重复  $n$  次独立的伯努利试验。在每次试验中只有两种可能的结果，而且两种结果发生与否是互相对立的，并且相互独立，与其他各次试验结果无关，事件发生与否的概率在每一次独立试验中都保持不变，则这一系列试验总称为  $n$  重伯努利实验。譬如投硬币的实验，出现正面的概率为  $x$ ，则出现负面的概率为  $1 - x$ ，那么，出现  $\alpha_1$  次正面， $\alpha_2$  次负面的概率函数为

$$P(\alpha_1, \alpha_2; x) = \binom{\alpha_1 + \alpha_2}{\alpha_1} x^{\alpha_1} (1 - x)^{\alpha_2}$$

#### 2. Beta分布

给定参数  $\alpha_1, \alpha_2 > 0$ ，取值范围为  $[0,1]$  的随机变量  $x$  的概率密度函数为

$$\begin{aligned} P(x; \alpha_1, \alpha_2) &= \frac{1}{B(\alpha_1, \alpha_2)} x^{\alpha_1-1} (1-x)^{\alpha_2-1} \\ &= \frac{\Gamma(\alpha_1 + \alpha_2)}{\Gamma(\alpha_1)\Gamma(\alpha_2)} x^{\alpha_1-1} (1-x)^{\alpha_2-1} \end{aligned}$$

其中， $B(\alpha_1, \alpha_2)$  为 Beta 函数，可以用 Gamma 函数  $\Gamma(\cdot)$  表示。如图 21-3 所示，列出 3 个 Beta 分布的概率密度分布图，参数的数值越大，在局部的概率密度会越大。

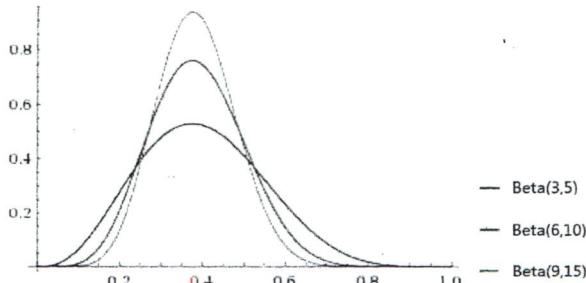


图 21-3

还是以投硬币的场景为例，我们来看这两种分布。二项分布是把概率  $x$  作为参数，即在概率  $x$  的值确定的情况下，不同的正反面次数  $\alpha_1$  和  $\alpha_2$  对应的概率分布；Beta 分布是把  $\alpha_1$  和  $\alpha_2$  作为参数，即在正反面出现次数确定的情况下，给出概率  $x$  对应的概率密度分布。

由贝叶斯理论，Beta 分布为二项分布的共轭先验分布。“轭”为驾车时搁在牛马颈上的曲木，可以起到束缚、控制的作用，使牛马行走同步。“共轭”可以理解为按照一定规律匹配的一对。

下面我们将“二项”推广为“多项”，将投硬币的场景扩展到投骰子的场景。在“二项”的时候，只给出一种结果的概率  $x$ ，譬如硬币为正面的概率，另一种结果的概率可以用  $1 - x$  表示。但在“多项”的情况下， $K$  种结果的概率要分别以  $x_1, x_2, \dots, x_K$  来表示，还要满足一个限制条件——所有结果的概率和为 1，即

$$x_1 + x_2 + \dots + x_K = 1$$

### 3. 多项分布 (Multinomial Distribution)

某随机实验如果有  $K$  个可能结局  $A_1, A_2, \dots, A_K$ ，譬如骰子有 6 个面，各种结果出现的概率分别为  $x_1, x_2, \dots, x_K$ ，那么在  $\alpha_1 + \alpha_2 + \dots + \alpha_K$  次试验结果中， $A_1$  出现  $\alpha_1$  次、 $A_2$  出现  $\alpha_2$  次、……、 $A_K$  出现  $\alpha_K$  次的概率，即多项分布的概率密度函数为

$$P(\alpha_1, \alpha_2, \dots, \alpha_K; x_1, x_2, \dots, x_K) = \frac{(\sum_{k=1}^K \alpha_k)!}{\prod_{k=1}^K \alpha_k!} \cdot \prod_{k=1}^K x_k^{\alpha_k}$$

其中

$$\sum_{k=1}^K x_k = 1, \quad x_k \geq 0, \quad k = 1, 2, \dots, K$$

#### 4. Dirichlet 分布

参数为  $K$  个正数  $\alpha_1, \alpha_2, \dots, \alpha_K$ ，其概率密度函数为

$$P(x_1, x_2, \dots, x_K; \alpha_1, \alpha_2, \dots, \alpha_K) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \cdot \prod_{k=1}^K x_k^{\alpha_k - 1}$$

其中，变量  $x_1, x_2, \dots, x_K$  的定义域为

$$\sum_{k=1}^K x_k = 1, \quad x_k \geq 0, k = 1, 2, \dots, K$$

以投骰子的场景为例，多项分布是把各面出现的概率  $x_1, x_2, \dots, x_6$  作为参数，即在概率确定的情况下，给出各面出现次数  $\alpha_1, \alpha_2, \dots, \alpha_6$  对应的概率分布。Dirichlet 分布是把各面出现的次数  $\alpha_1, \alpha_2, \dots, \alpha_6$  作为参数，即在各面出现次数确定的情况下，给出各面出现概率  $x_1, x_2, \dots, x_6$  对应的概率密度分布。

LDA 主题模型如图 21-4 所示，详细内容请参考 David M. Blei、Andrew Y. Ng 和 Michael I. Jordan 在 2003 年发表的论文（参见链接 21-2）。其中，带阴影的圆圈表示可以观测到的变量，即文档内容；不带阴影的圆圈都是潜在变量。每篇文章的主题分布是按照参数为  $\alpha$  的 Dirichlet 分布采样得到的多项分布，每个主题的词分布是按照参数为  $\beta$  的 Dirichlet 分布采样得到的多项分布。

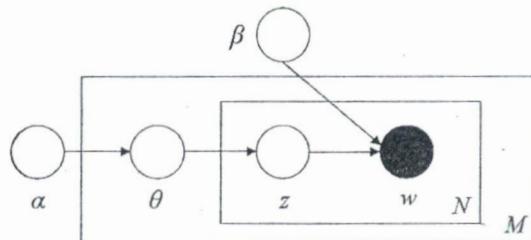


图 21-4

#### 21.7.2 新闻的主题模型

下面以头条新闻数据集为例，根据新闻标题内容计算主题模型。首先，在原始数据中选出本节需要的两列数据，新闻标题列用来计算主题模型，在 21.8 节将通过标签列来验证主题模型的效果。因为 LDA 模型是基于分词结果计算的，所以还需使用分词和停用词组件，具体代码如下。

```

BatchOperator <?> docs = getSource()
    .select(LABEL_COL_NAME + ", " + TXT_COL_NAME)
    .link(new SegmentBatchOp().setSelectedCol(TXT_COL_NAME))
    .link(new StopWordsRemoverBatchOp().setSelectedCol(TXT_COL_NAME));

docs.lazyPrint(10);

```

处理后的结果数据如表 21-28 所示。

表 21-28 新闻标题进行数据处理后的结果

category_name	news_title
news_sports	詹姆斯 吃饭 睡觉 猛龙 猛龙 很萌
news_sports	上港 球迷 远征 亚冠 客场 扬 国旗 举 围巾 霸气 呐喊助威
news_finance	沪 指 重新 站上 3100 点 市场 氛围 转好 优质 票 可选
news_finance	需 小心 地雷 炸伤 大盘 积极 搭台 创业板 唱戏 反弹 围绕 一点 位 展开
news_finance	人民日报 欧盟 预算 因何 众口难调
news_finance	天津 七 一二 通信 广播 股份 有限公司 召开 2017 年度 业绩 现金 分红 说明会 公告
news_finance	股票 不能 碰
news_finance	券商 四大 传统 业务 承压 期货 海外 业务 加速 成长
news_finance	文安 产业 新城 签约 追梦 教育 集团 助力 文安 教育 发展
news_finance	泰达 股份 北方 信托 混改 有利于 优化 公司 竞争力

下面讲解本节的重点——LDA 模型训练。使用 LDA 模型训练组件，设置 Topic 个数为 10，设置迭代次数 NumIter=200，设置词汇表的容量为 20000（会根据各单词在整个语料中出现的频率进行排序，取前 20000 个单词）；选择输出 LDA 模型的信息，并将 LDA 模型保存到文件中。另外，训练组件的 0 号侧输出（SideOutput[0]）还会输出单词主题概率表（单词属于各 Topic 的概率），也将其保存到文件，便于后面分析使用。相关代码如下。

```

static final String LDA_MODEL_FILE = "lda_model.ak";
static final String LDA_PWZ_FILE = "lda_pwz.ak";
.....
LdaTrainBatchOp lda = new LdaTrainBatchOp()
    .setTopicNum(10)
    .setNumIter(200)
    .setVocabSize(20000)
    .setSelectedCol(TXT_COL_NAME)
    .setRandomSeed(123);

docs.link(lda);

```

---

```

lda.lazyPrintModelInfo();

lda.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + LDA_MODEL_FILE)
);

lda.getSideOutput(0)
    .link(
        new AkSinkBatchOp().setFilePath(DATA_DIR + LDA_PWZ_FILE)
    );

BatchOperator.execute();

```

---

显示 LDA 模型信息如下，可以看到 Topic 的个数、词汇表的个数，以及两个重要指标 logPerplexity 和 logLikelihood。

----- LdaModelInfo -----

topic number	vocabulary size	logPerplexity	logLikelihood
10. 0	20000	1247. 7366	-24954732. 9439

### 21.7.3 主题与原始分类的对比

我们首先使用 LDA 模型预测各新闻标题所属的主题（Topic），具体代码如下。

---

```

new LdaPredictBatchOp()
    .setSelectedCol(TXT_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol("predinfo")
    .linkFrom(
        new AkSourceBatchOp().setFilePath(DATA_DIR + LDA_MODEL_FILE),
        docs
    )
    .lazyPrint(5)
    .link(
        new EvalClusterBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics()
    );

```

---

使用 LdaPredictBatchOp 组件，设置参数 SelectedCol 为标题文本数据，设置预测结果列 PredictionCol，并设置预测详细信息列 PredictionDetailCol（其输出为一个向量，各分量对应着

属于各 Topic 的概率）。该预测组件需要输入两个数据，一个为 LDA 模型，另一个为所要预测的数据。预测结果中会输出 5 条数据，由于主题模型的预测结果与聚类预测结果相似，可以使用聚类评估组件，看看预测结果与原始标签的联系。

预测结果如表 21-29 所示，右边两列分别为主题索引值和该新闻属于各主题的概率。

表 21-29 LDA 计算结果

category_name	news_title	pred	predinfo
news_house	山东 最 近 几 个 月 好 地 段 楼 盘 几 乎 翻 一 倍 看	9	0.08846173326746984 0.08790198371425065 0.08695652173913526 0.11132784130175362 0.08704271106524446 0.0963784707694184 0.10143139656072783 0.087337078787098 0.10146142566592847 0.15170083712897348
news_house	中国 两 百 万 以 上 资 产 家 庭 大 概	9	0.09230769231252751 0.1076919729680564 0.09230769783775432 0.09230769273934723 0.0923077056600504 0.10769231195776426 0.0927928683155056 0.09230769230822215 0.10769728241983208 0.12258708348094007
news_house	买 二 手 房 要 交 税 最 全 计 算 方 式 一 览	9	0.08955223880597014 0.11107794092234603 0.09083815954230492 0.09070069094553912 0.11264249230503576 0.09498002328302255 0.10447764165673845 0.08955223910814286 0.10294956612073876 0.11322900731016139
news_house	广 东 惠 州 买 房 升 值 空 间	9	0.09230769230769281 0.0923076923579364 0.09231051293661477 0.09230769245814259 0.09230769232219097 0.09230769239314511 0.107701241625694 0.09230805945158871 0.10769998096127562 0.13844174318571903
news_car	a2 驾 照 能 开 23 座 客 车	6	0.09096819616171696 0.09096908235666334 0.10775203096378642 0.10633489746936109 0.09595360701940348 0.09443902433286673 0.13459232009550584 0.09593454722532924 0.09099265512255543 0.09206363925281145

输出评估结果如下，我们主要关注标签值与主题索引值间的交叉矩阵。

Label\Cluster	9	8	7	6	5	4	3	2	1	0
stock	83	151	8	16	23	12	11	29	3	4
news_world	911	773	529	541	3547	554	4236	12875	889	2054
news_travel	2121	2876	486	1250	4895	648	753	974	5829	1590
news_tech	4248	6215	720	2643	2158	18977	2986	2093	759	744
news_story	400	61	58	100	645	147	118	123	963	3658
news_sports	782	750	28915	565	1417	616	707	1095	761	1960
news_military	592	778	497	756	2487	470	13919	3949	716	820
news_house	11619	1528	197	980	903	639	241	832	535	198
news_game	1133	6492	2220	953	2313	1745	7274	1497	1220	4453
news_finance	5814	10018	609	1546	2176	1319	1398	3149	665	391
news_entertainment	855	611	1360	621	2421	711	1090	891	3590	27246
news_edu	1497	12657	551	1336	1752	4846	761	989	1761	908
news_culture	697	1577	586	522	3392	784	1059	1102	17085	1227
news_car	1313	1105	309	23783	3049	2418	716	1182	1028	882
news_agriculture	3134	3961	279	827	5706	1043	818	1055	1902	597

接着导入单词主题概率表，先输出几条数据，了解数据的格式，然后找出主题中概率最高的 20 个单词，这些单词上可以帮助我们理解各主题的内容。具体代码如下。

```
AkSourceBatchOp pwz = new AkSourceBatchOp().setFilePath(DATA_DIR + LDA_PWZ_FILE);

pwz.sample(0.001).lazyPrint(10);

for (int t = 0; t < 10; t++) {
    pwz.select("word, topic_" + t)
        .orderBy("topic_" + t, 20, false)
        .lazyPrint(-1, "topic" + t);
}
```

单词主题概率表中的数据如表 21-30 所示，第一列为单词，后面是属于各主题的概率。可以看到，横向的概率和为 1，单词“顶配”在 topic\_6 上的概率最大，为 0.7334；再从前面的交叉矩阵中可以看到，topic\_6 中 news\_car 的新闻出现最多。

表 21-30 单词主题概率表

word	topic_0	topic_1	topic_2	topic_3	topic_4	topic_5	topic_6	topic_7	topic_8	topic_9
顶配	0.0214	0.0214	0.0214	0.0532	0.0426	0.0214	0.7334	0.0214	0.0426	0.0214
问询	0.0140	0.0140	0.1943	0.0695	0.0556	0.0417	0.0695	0.0556	0.4717	0.0140
鉴别	0.2118	0.2118	0.0607	0.0305	0.2420	0.0305	0.0607	0.0607	0.0305	0.0607
这款	0.0033	0.0044	0.0011	0.0111	0.0066	0.0033	0.9557	0.0011	0.0078	0.0055

续表

word	topic_0	topic_1	topic_2	topic_3	topic_4	topic_5	topic_6	topic_7	topic_8	topic_9
花卉	0.0862	0.0346	0.0690	0.2239	0.0690	0.1723	0.0518	0.0862	0.1207	0.0862
犯下	0.0288	0.0288	0.2282	0.1427	0.1142	0.0573	0.0573	0.0858	0.1427	0.1142
沪市	0.0646	0.0325	0.0968	0.1611	0.0325	0.0646	0.0968	0.0646	0.2897	0.0968
宿迁	0.0572	0.1000	0.0144	0.1428	0.1000	0.1285	0.0144	0.0715	0.2712	0.1000
姐	0.1311	0.1475	0.0656	0.0329	0.1147	0.1147	0.0493	0.0656	0.2293	0.0493
同一个	0.1939	0.0449	0.1939	0.1045	0.0896	0.0896	0.0896	0.0300	0.0896	0.0747

各主题中概率最高的 20 个单词，计算汇总如表 21-31 所示。

表 21-31 各主题高频单词

topic_0	topic_1	topic_2	topic_3	topic_4
word topic_0	word topic_1	word topic_2	word topic_3	word topic_4
----- -----	----- -----	----- -----	----- -----	----- -----
网友 0.9967	上联 0.9973	美国 0.9959	以色列 0.9892	手机 0.9976
岁 0.9904	下联 0.9969	世界 0.9926	求生 0.9880	联想 0.9946
活动 0.9871	黄山 0.9774	特朗普 0.9917	俄罗斯 0.9868	老师 0.9933
穿 0.9849	泰山 0.9729	退出 0.9904	战机 0.9866	学生 0.9899
现身 0.9837	赵本山 0.9729	伊核 0.9900	俄 0.9843	看待 0.9886
出席 0.9834	对联 0.9611	协议 0.9854	击落 0.9842	华为 0.9885
戛纳 0.9781	春风 0.9601	伊朗 0.9832	绝地 0.9824	5g 0.9879
rng 0.9763	接 0.9565	国家 0.9821	称 0.9813	高通 0.9876
kz 0.9745	写 0.9535	黄金 0.9811	警告 0.9811	响个 0.9868
范冰冰 0.9744	公鸡 0.9526	宣布 0.9768	普京 0.9808	家长 0.9866
机场 0.9706	故事 0.9503	伊朗核 0.9736	将会 0.9782	事 0.9864
儿子 0.9687	打架 0.9466	会 0.9693	军事基地 0.9765	不停 0.9850
女儿 0.9625	人生 0.9408	成为 0.9677	链 0.9743	苹果 0.9844
粉丝 0.9620	甄 0.9315	制裁 0.9639	轰炸 0.9736	赔 0.9823
腿 0.9580	求下联 0.9301	影响 0.9616	叙利亚 0.9711	发票 0.9815
鹿晗 0.9567	千金 0.9257	总统 0.9591	航母 0.9694	小米 0.9808
赵丽颖 0.9550	头 0.9253	事件 0.9585	美军 0.9667	上课时 0.9795
谢娜 0.9544	嬛 0.9250	走 0.9581	战场 0.9658	票 0.9790
明星 0.9519	经典 0.9208	今后 0.9522	歼 0.9606	摔 0.9781
红毯 0.9517	一夜 0.9203	要求 0.9493	币 0.9589	投给 0.9776

续表

topic_5	topic_6	topic_7	topic_8	topic_9
word topic_5	word topic_6	word topic_7	word topic_8	word topic_9
说 0.9930	万 0.9941	火箭 0.9918	荣耀 0.9923	月 0.9940
人 0.9873	suv 0.9895	詹姆斯 0.9890	王者 0.9899	日 0.9906
没有 0.9801	车 0.9877	勇士 0.9850	万元 0.9765	房子 0.9905
中国 0.9795	奥迪 0.9856	nba 0.9849	2017 0.9740	马云 0.9903
日本 0.9792	买 0.9846	球迷 0.9842	股份 0.9727	5 0.9902
象棋 0.9759	上市 0.9789	球员 0.9839	教育 0.9717	房价 0.9878
印度 0.9738	奔驰 0.9771	骑士 0.9836	有限公司 0.9674	言论 0.9821
古代 0.9719	款 0.9768	保罗 0.9819	营收 0.9666	惊人 0.9808
大象 0.9680	大众 0.9767	决赛 0.9810	亿元 0.9658	8 0.9806
源自 0.9646	宝马 0.9737	1 0.9810	专业 0.9625	买房 0.9798
农村 0.9613	座 0.9702	西决 0.9809	招生 0.9618	后 0.9797
说法 0.9602	国产 0.9698	恒大 0.9806	净赚 0.9591	可信 0.9786
方舟子 0.9588	新车 0.9690	比赛 0.9800	大学 0.9587	不值钱 0.9752
很多 0.9551	仅 0.9685	猛龙 0.9783	公告 0.9587	年 0.9722
吃 0.9508	车型 0.9664	0 0.9779	建设 0.9574	银行 0.9644
俗语 0.9388	万起 0.9664	中超 0.9749	产业 0.9554	上涨 0.9634
股市 0.9170	丰田 0.9664	主场 0.9743	项目 0.9548	9 0.9614
有人 0.9166	1 0.9656	联赛 0.9737	公司 0.9511	支付宝 0.9570
认为 0.9076	全新 0.9639	哈登 0.9730	教师 0.9468	钱 0.9558
这句 0.8928	性价比 0.9621	分 0.9713	新 0.9440	出 0.9552

有了这个主题单词表，再加上前面的标签值与主题索引值间的交叉矩阵，我们就能更容易地了解各主题的内容。这里抛砖引玉，分析以下两个主题。

- 前面的讨论涉及了 topic\_6，其中 news\_car 的新闻占了绝大部分，而且其中概率较高的单词有“SUV、车、奥迪、奔驰、大众、宝马、新车”等，显然是关于汽车的主题。
- topic\_0 中 news\_entertainment 的新闻占了绝大部分，高概率出现的单词有“戛纳、范冰冰、鹿晗、赵丽颖、谢娜、粉丝、明星、红毯”等，非常明显，这是娱乐新闻。

## 21.8 组件使用小结

本节用思维导图描绘前面介绍的各种文本组件的调用关系及所需的数据依赖，如图 21-5 所示。

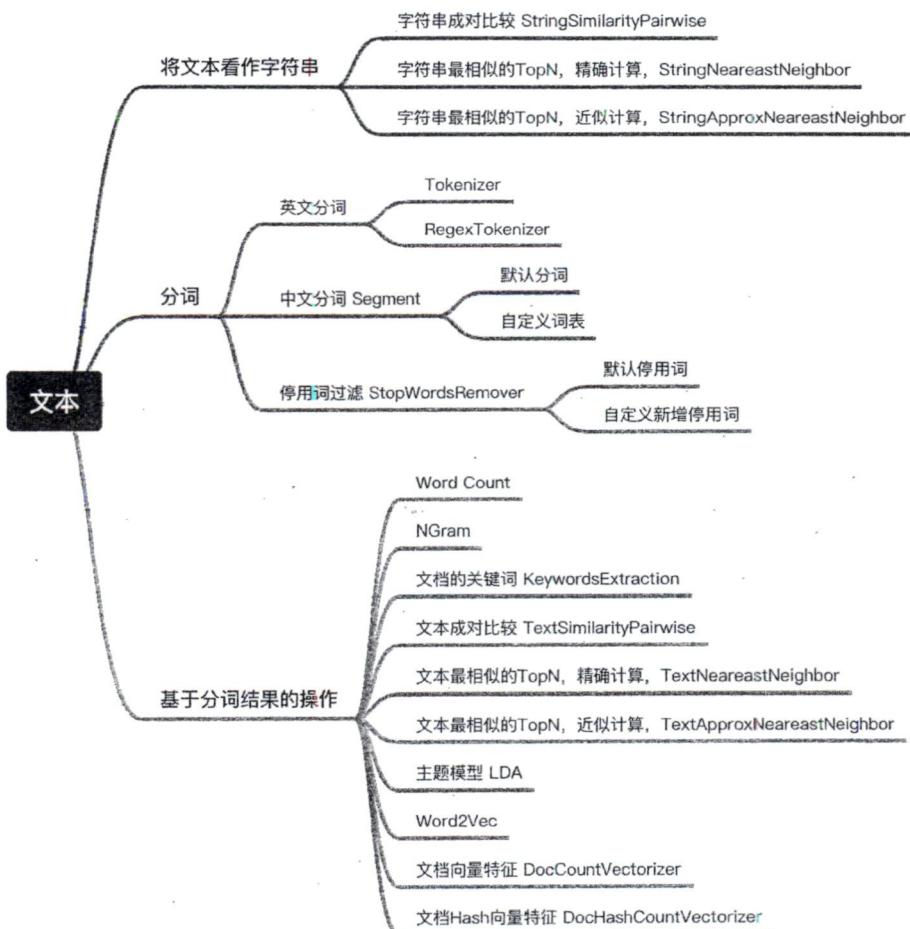


图 21-5

## 单词向量化

单词是文本的基本单位，单词向量化是文本向量化的基础。通过向量化将文本转化为机器学习所需的向量特征，也可以通过计算向量间的距离判断单词或文本间的相似度。

把一个单词表示成一个向量，主要有两种方法。

- 独热 (One-Hot) 编码表示：这是最简单的表示方式，仅与词典相关。为词典中的每个单词赋予一个序号，对于每个单词，向量维度是词典中单词的个数，将该单词序号的位置设置为 1，其余位置都设置为 0。整个向量中只有一个位置不为 0，只有一个热点，即 One-Hot。
- 嵌入式 (Embedding) 表示：这种表示方式不仅需要词典，还需要训练语料，根据每个单词出现的上下文 (context)，需要指定向量的维度（一般选 50、100），显然，对比独热编码表示，维度数显著减少，每个维度不再对应一个单词，可以看作某个隐含变量。每个单词会通过各个隐含变量上的权重值表示出来，也称为“Word Embedding”。对于该种表示方法得出的结果，可以通过向量间的距离判断单词间的相似度。

独热编码的优势是简单直观，各个单词对应一个向量分量的位置，在文本中各个单词所在向量分量的位置设置数值，就得到了文本的向量表示。向量分量值可以设置为 1，表示文档中是否出现了该单词，也可以设置为单词出现的个数，或者更能表达单词区分度的指标（譬如 TF、IDF、TF-IDF 等）。在实际应用中，还可以使用 hash 的方式，简化单词到向量位置的映射操作。

单词表示为嵌入式向量，需要对语料库数据进行训练才能得到。而在不同的语料库数据和不同的训练参数情况下，每个词得到的单词向量表示可能是不一样的。嵌入式单词向量将单词映射到高维空间上的点，点之间距离可以看作对应的两个词之间的“距离”，即两个词在语法、

语义上的相似性。

## 22.1 单词向量预训练模型

这里我们选择了两个容易查到的英文的单词向量预训练模型。

- 一个模型是文件尺寸小一些的，便于下载，参见链接 22-1，由维基百科相关（Wikipedia Dependency）的一些语料训练而成。
- 另一个模型来自 GloVe，也是选择其中较小的一个下载，参见链接 22-2，其中包含了多个维度的向量模型。

### 22.1.1 加载模型

将 Wiki 单词向量预训练模型文件（参见链接 22-1）下载到本地，并解压为 deps.words，使用文本编辑器打开，如图 22-1 所示。

deps.words
0.0123964806212 0.0226242564799 -0.0842967628244 0.0358073334299 0.06602465833
-0.0481858908712 0.011024201281 0.102462772857 0.0988509622331 -0.0436883298619
0.0276317616644 -0.0128436928669 -0.132208601694 0.156152813645 -0.135469464093
-0.0585674591764 1.38760182098e-05 -0.136834333342 0.135753055052 -0.0018218430884
-0.117289867134 -0.0456775129422 0.0521139588947 0.011946884564 -0.0845349668663
-0.0477418585444 0.02427755252797 0.03115791136754 0.0164370013441 0.0459145614499
0.0329795181825 0.0621171160382 -0.101167388688 0.0514603401971 0.0374672511512
0.0345697667693 -0.07892240982 -0.0132637312593 0.042577891911 -0.0764559490052
-0.0451158236122 0.0688976764386 0.0399068271962
rito -0.0345849277658 0.0151980538933 0.0350979861837 0.0418954895262 0.116777372302
0.0807744177182 0.0370075493482 -0.0370190046444 0.0347609227711 -0.0103548935096
-0.0217747824361 -0.0343617230549 -0.0183114645499 0.0240502200188 -0.0083623662307
-0.0675924959571 -0.0622275989008 -0.0103795397529 -0.0570931962893 -0.00779446124252
-0.08665273798189 0.00546799472219 -0.0476897866251 0.0454237207575 0.0850628640614
0.0990758154927 -0.0357672531863 -0.0388386610902 0.0484107760257 0.0977483896536
0.014095568416 -0.0209486125886 0.0297608595424 0.0171489255501 0.0757553444009
-0.00569224082368 -0.0461429745072 0.0335702662261 -0.02170088437061 -0.00422700429991
-0.0265839240614 0.054578585207 0.0640722487201 -0.0516835194387 0.0104277908491
0.0224765796764 0.0642808739631 -0.0446598462487 0.00356363851054 0.0122599439816
0.0237837848911 -0.0293949843242 -0.005345804896 -0.0135911882528 0.0434169611404

图 22-1

数据格式比较简单，单词与向量之间用空格隔开，向量的各分量间使用空格分隔（这与 Alink 调密向量的序列化为字符串的格式一样）。可以通过组件 TextSourceBatchOp 将一行看作一条文本数据，然后以第一个出现的空格为界，划分为 word 和 vec 两部分，如下代码所示。

```
static BatchOperator<?> getWikiDependency() {
    return new TextSourceBatchOp().setFilePath(DATA_DIR + WIKI_DEPENDENCY)
        .setTextCol("txt")
```

```

.select("SUBSTRING(txt FROM 1 FOR POSITION(' ' IN txt)-1 ) AS word, "
+ "SUBSTRING(txt FROM POSITION(' ' IN txt) + 1 ) AS vec");
}

```

将 glove.6B 单词向量预训练模型文件（参见链接 22-2）下载到本地，解压后会出现 4 个文本文件，名称分别为 glove.6B.50d.txt、glove.6B.100d.txt、glove.6B.200d.txt 和 glove.6B.300d.txt。从名称上就可以看出，各文件的差别在于对应向量的维度。我们选择 100 维的情况作为试验的目标。使用文本编辑器打开，如图 22-2 所示。

```

glove.6B.100d.txt
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.843953 -0.39141 0.3344 -0.57545
0.087459 0.28787 -0.06731 0.38986 -0.26384 -0.13231 0.28757 0.33395 -0.33848 -0.31743
-0.48336 0.1464 -0.37384 0.34577 0.052841 0.44946 0.02628 -0.54155 -0.15518
-0.14187 -0.039722 0.28277 0.14393 0.23464 -0.31021 0.086173 0.28397 0.52624 0.17164
-0.082378 -0.71787 -0.41531 0.20335 -0.12763 0.41367 0.55187 0.57908 -0.33477 -0.36559
-0.54857 -0.062892 0.26584 0.30285 0.99775 -0.80481 -3.0243 0.01254 -0.35942 2.2167
0.72281 -0.24978 0.92136 0.034514 0.46745 1.1079 -0.19358 -0.074575 0.23353 -0.052062
-0.22044 0.057162 -0.15886 -0.30798 -0.41625 0.37972 0.15006 -0.53212 -0.2055 -1.2526
0.071624 0.78565 0.49744 -0.42063 0.26148 -1.538 -0.30223 -0.073438 -0.28312 0.37104
-0.25217 0.016215 -0.817899 -0.38984 0.87424 -0.72569 -0.51858 -0.52028 -0.1459 0.8278
0.27062
-0.18767 0.11053 0.59812 -0.54361 0.67396 0.10663 0.038867 0.35481 0.06351 -0.094189
0.15786 -0.81665 0.14172 0.21939 0.58505 -0.52158 0.22783 -0.16642 -0.68228 0.3587
0.42568 0.19021 0.91963 0.57555 0.46185 0.42363 -0.895399 -0.42749 -0.16567 -0.056842
-0.29595 0.26037 -0.26606 -0.070484 -0.27662 0.15821 0.69825 0.43881 0.27952 -0.45437
-0.33801 -0.58184 0.22364 -0.5778 -0.26862 -0.20425 0.56394 -0.58524 -0.14365 -0.64218
0.0854697 -0.35248 0.16162 1.1796 -0.47674 -2.7553 -0.1321 -0.047729 1.0655 1.1034
-0.2208 0.18669 0.13177 0.15117 0.7131 -0.35215 0.91348 0.61783 0.70992 0.23955
-0.14571 -0.37859 -0.045959 -0.47368 0.2385 0.28536 -0.18996 0.32587 -1.1112 -0.36341
0.94674 -0.084776 -0.54088 0.11796 -1.0194 -0.24424 0.17771 0.014884 0.080774 -0.35414

```

图 22-2

数据格式与 deps.words 一样，单词与向量之间用空格隔开，向量的各分量间使用空格隔开（这与 Alink 稠密向量的序列化为字符串的格式一样）。仍然通过 TextSourceBatchOp 组件将一行看作一条文本数据，然后以第一个出现的空格为界，划分为 word 和 vec 两部分。

```

static BatchOperator<?> getGlove6B100d() {
    return new TextSourceBatchOp() {
        .setFilePath(DATA_DIR + GLOVE_6B_100D)
        .setTextCol("txt")
        .select("SUBSTRING(txt FROM 1 FOR POSITION(' ' IN txt)-1 ) AS word, "
            + "SUBSTRING(txt FROM POSITION(' ' IN txt) + 1 ) AS vec");
    }
}

```

经过上述处理得到的数据集包含两列，分别为 word 和 vec，与 Alink 提供的单词向量模型的数据格式完全一样。

### 22.1.2 查找相似的单词

可以使用向量最近邻组件 VectorNearestNeighbor，其使用方式与第 21 章介绍的 StringNearestNeighbor 组件类似，只是把用来计算和比较的内容换成了向量。具体代码如下。

```

BatchOperator.setParallelism(1);
for (BatchOperator <?> word2vec : new BatchOperator <?>[] {getWikiDependency(), getGlove6B100d()}) {
    for (String metric : new String[] {"EUCLIDEAN", "COSINE"}) {
        new VectorNearestNeighbor()
            .setIdCol("word")
            .setSelectedCol("vec")
            .setMetric(metric)
            .setOutputCol("similar_words")
            .setTopN(7)
            .fit(word2vec)
            .transform(word2vec.filter("word='king'"))
            .select("word, similar_words")
            .lazyPrint(-1, metric);
        BatchOperator.execute();
    }
}

```

外层循环选择不同的单词向量模型，内层循环使用不同的距离度量（EUCLIDEAN 与 COSINE），每次计算的目标都是单词“king”对应的向量。

先看 WikiDependency 向量模型计算出来的结果，如下所示。

EUCLIDEAN	{"ID": "[\"king\", \"norodom\", \"songtsän\", \"queen\", \"bhumibol\", \"monarch\", \"archduke\"]", "METRIC": "[2.9802322387695312E-8, 0.8055086190497484, 0.8063912079032907, 0.8214690626774828, 0.8229473168473302, 0.8253910038487656, 0.8260947822801127]"}]
COSINE	{"ID": "[\"king\", \"norodom\", \"songtsän\", \"queen\", \"bhumibol\", \"monarch\", \"archduke\"]", "METRIC": "[-7.771561172376096E-16, 0.32442206768187054, 0.3251333900919796, 0.3374057104681471, 0.338621143153173, 0.340635154617282, 0.3412162946552998]"}]

可以看到，使用两种度量方式计算出来的前 7 个词都是一样的，甚至顺序都一样，但两种方式下的度量值不同。被搜索的词库中含有该单词，所以该单词被排在了第一位，单词 queen（皇后）、monarch（帝王）和 archduke（大公）显然和 king 有紧密的联系；对于其他 3 个单词，通过搜索引擎获取其含义如表 22-1 所示，都是著名的“国王”。

表 22-1 3 个单词的含义

norodom	songtsän	bhumibol
诺罗敦	Songtsen Gampo (松赞干布)	Bhumibol Adulyadej (普密蓬·阿杜德)
诺罗敦，柬埔寨国王，1860 年至 1904 年在位。本名安瓦戴。他是柬埔寨诺罗敦王室的始祖	松赞干布，悉补野氏，是吐蕃雅鲁王朝第 33 任赞普，也是吐蕃帝国的建立者，约 629 年至 650 年在位。他是前任赞普囊日论赞的儿子。原名赤松赞，“松赞干布”是其尊号，意思是“心胸深邃的松赞”	普密蓬·阿杜德，亦称普密蓬大帝，泰国却克里王朝第九代国王，亦称拉玛九世。1946 年 6 月 9 日即位，2016 年 10 月 13 日驾崩，过世时为世界上在任时间第二长的国家元首，亦为泰国历史上统治时间最长的国君，总共在位 70 年 126 天

再看 GLOVE\_6B\_100D 向量模型计算出来的结果，如下所示。

EUC	{"ID": "[\"king\", \"prince\", \"queen\", \"monarch\", \"brother\", \"uncle\", \"nephew\"]", "METRIC": "[1.6858739404357614E-7, 4.092165813215177, 4.281252149113389, 4.474171532331215, 4.536668410815254, 4.668967636875079, 4.695680189674975]"}]
COS INE	{"ID": "[\"king\", \"prince\", \"queen\", \"son\", \"brother\", \"monarch\", \"throne\"]", "METRIC": "[2.220446049250313E-16, 0.23176711289070218, 0.2492309206376152, 0.29791115594092876, 0.301422441664613, 0.85, 0.3022109473347163, 0.3080009470368401]"}]

可以看到，使用两种度量方式计算出来的前 3 个词都是一样的，都为 king、prince 和 queen。后面的词主要有两类：monarch（帝王）和 throne（君主）；brother（兄弟）、uncle（叔伯）、nephew（侄子）和 son（儿子）。

综合上面的测试结果可知，不同语料来源的单词向量模型是有区别的；单词的相似度可以通过向量距离来衡量，但“相似”的含义受具体单词向量模型的影响。

### 22.1.3 单词向量

本节我们会聚焦 4 个单词：man、woman、king、queen。

首先，我们提取并输出这 4 个单词所对应的向量，具体代码如下。

```
getWikiDependency()
    .filter("word IN ('man', 'woman', 'king', 'queen')")
    .lazyPrint(-1);

getGlove6B100d()
    .filter("word IN ('man', 'woman', 'king', 'queen')")
    .lazyPrint(-1);

BatchOperator.execute();
```

输出 WikiDependency 中这 4 个单词的向量值，见表 22-2。

表 22-2 WikiDependency 中 4 个单词的向量值

word	vec
man	-0.00220404170083 0.0678135463787 ..... -0.0164630158472 -0.0441532936764
king	0.0330381329194 0.0665698704227 ..... -0.013237880883 -0.0142381059934
woman	-0.0923953735088 0.062354665309 ..... -0.0518072294457 0.0179050510523
queen	0.0199859507483 0.1526205478 ..... 0.0267236279156 0.0352932794572

输出 Glove6B100d 中这 4 个单词的向量值，见表 22-3。

表 22-3 Glove6B100d 中 4 个单词的向量值

word	vec
man	0.37293 0.38503 ..... 0.039138 -0.53911
king	-0.32307 -0.87616 ..... 0.16483 -0.98878
woman	0.59368 0.44825 ..... 0.15162 -0.30754
queen	-0.50045 -0.70826 ..... 0.13347 -0.56075

Alink 定义了稠密向量 (DenseVector) 类型，可以处理向量计算。下面将由提取出来的向量字符串构造 DenseVector，随后从向量角度进行分析。

先分析 WikiDependency 中这 4 个单词的向量值，具体代码如下。

```
DenseVector vec_man = VectorUtil.parseDense(
    "-0.00220404170083 0.0678135463787 ... ... -0.0164630158472 -0.0441532936764");

DenseVector vec_woman = VectorUtil.parseDense(
    "-0.0923953735088 0.062354665309 ... ... -0.0518072294457 0.0179050510523");

DenseVector vec_king = VectorUtil.parseDense(
    "0.0330381329194 0.0665698704227 ... ... -0.013237880883 -0.0142381059934");

DenseVector vec_queen = VectorUtil.parseDense(
    "0.0199859507483 0.1526205478 ... ... 0.0267236279156 0.0352932794572");

System.out.println("'man' vector normL2 : " + vec_man.normL2());
System.out.println("'woman' vector normL2 : " + vec_woman.normL2());
System.out.println("'king' vector normL2 : " + vec_king.normL2());
System.out.println("'queen' vector normL2 : " + vec_queen.normL2());
System.out.println("'man - woman' normL2 : " + vec_man.minus(vec_woman).normL2());
System.out.println("'king - queen' normL2 : " + vec_king.minus(vec_queen).normL2());
System.out.println("(|man - woman| - |king - queen|) normL2 : " +
    vec_man.minus(vec_woman).minus(vec_king.minus(vec_queen)).normL2());
```

运行结果如下。

```
'man' vector normL2 : 0.999999999999892
'woman' vector normL2 : 0.9999999999997525
'king' vector normL2 : 0.9999999999997967
'queen' vector normL2 : 1.0000000000000957
'man - woman' normL2 : 0.6768403103559452
'king - queen' normL2 : 0.8214690626774831
(|man - woman| - |king - queen|) normL2 : 0.9766210180698275
```

显然，这 4 个单词向量的 2-范数都为 1；“man”与“woman”的向量差“man - woman”的 2-范数为 0.6768403103559452；“king”与“queen”的向量差“king - queen”的 2-范数为 0.8214690626774831。

由于 2-范数不同，向量“man-woman”与“king-queen”应该不同，但差异有多大呢？我们计算这两个向量的差，即“(man-woman)-(king-queen)”，其 2-范数为 0.9766210180698275，说明差异不小。所以从向量值的角度，有

$$\text{man-woman} \neq \text{king-queen}$$

但是向量“man-woman”所指向的方向是否能给我们带来一些有意思的结论呢？在下面的实验中，我们会构造两个新的向量“king-man+woman”与“queen-woman+man”，看看有哪些与之相似的单词？具体代码如下。

```
new VectorNearestNeighbor()
    .setIdCol("word")
    .setSelectedCol("vec")
    .setMetric(Metric.EUCLIDEAN)
    .setOutputCol("similar_words")
    .setTopN(5)
    .fit(
        getWikiDependency()
    )
    .transform(
        new MemSourceBatchOp(
            new Row[] {
                Row.of("king", vec_king),
                Row.of("king-man+woman", vec_king.minus(vec_man).plus(vec_woman)),
                Row.of("queen", vec_queen),
                Row.of("queen-woman+man", vec_queen.minus(vec_woman).plus(vec_man)),
            },
            new String[] {"word", "vec"}
        )
    )
    .select("word, similar_words")
    .print();
```

运行结果如表 22-4 所示。

表 22-4 相似单词

word	similar_words
king	{"ID": ["king", "norodom", "songtsän", "queen", "bhumibol"], "METRIC": [2.9802322387695312E-8, 0.8055086190497484, 0.8063912079032907, 0.8214690626774828, 0.8229473168473302]}
king-man+woman	{"ID": ["king", "queen", "monarch", "princess", "norodom"], "METRIC": [0.6768403103559449, 0.976621018069827, 0.9920983527502455, 1.023152990110213, 1.0314874003333323]}

续表

word	similar_words
queen	{"ID": ["queen", "tsarina", "princess", "king", "tsaritsa"], "METRIC": [2.9802322387695312E-8, 0.8076690691199051, 0.8195939124642033, 0.8214690626774828, 0.8629289238663901]}
queen-woman+man	{"ID": ["queen", "king", "tsarina", "knave", "overlord"], "METRIC": [0.6768403103559447, 0.9766210180698268, 1.0186863394683976, 1.0294727515416788, 1.0405324064187942]}

对比“king-man+woman”与“king”，最相邻的有皇后“queen”和公主“princess”；而和“queen-woman+man”最相邻的是皇后“queen”和国王“king”。

我们继续做这个实验，分析Glove6B100d中这4个单词的向量值，具体代码如下。

```
DenseVector vec_man = VectorUtil.parseDense(
    "0.37293 0.38503 0.71086 ... ... -0.93711 0.039138 -0.53911");

DenseVector vec_woman = VectorUtil.parseDense(
    "0.59368 0.44825 0.5932 ... ... -0.54648 0.15162 -0.30754");

DenseVector vec_king = VectorUtil.parseDense(
    "-0.32307 -0.87616 0.21977 ... ... -0.52881 0.16483 -0.98878");

DenseVector vec_queen = VectorUtil.parseDense(
    "-0.50045 -0.70826 0.55388 ... ... -0.36032 0.13347 -0.56075");

System.out.println("'man' vector normL2 : " + vec_man.normL2());
System.out.println("'woman' vector normL2 : " + vec_woman.normL2());
System.out.println("'king' vector normL2 : " + vec_king.normL2());
System.out.println("'queen' vector normL2 : " + vec_queen.normL2());
System.out.println("'man - woman' normL2 : " + vec_man.minus(vec_woman).normL2());
System.out.println("'king - queen' normL2 : " + vec_king.minus(vec_queen).normL2());
System.out.println("(man - woman) - (king - queen) normL2 : " +
    vec_man.minus(vec_woman).minus(vec_king.minus(vec_queen)).normL2());
```

运行结果如下。

```
'man' vector normL2 : 5.593625640618958
'woman' vector normL2 : 5.961704845063365
'king' vector normL2 : 6.1176004287966546
'queen' vector normL2 : 6.006716940168814
'man - woman' normL2 : 3.364067897373334
'king - queen' normL2 : 4.281252149113388
(man - woman) - (king - queen) normL2 : 4.081078579600476
```

显然，这4个单词向量的2-范数都在6左右；向量“(man-woman) - (king-queen)”的2-范

数为 4.081078579600476，说明差异不小。所以从向量值的角度，有

$$\text{man-woman} \neq \text{king-queen}$$

与 WikiDependency 单词向量模型计算两个新的向量“king-man+woman”和“queen-woman+man”的代码类似，我们同样可计算基于 Glove6B100d 单词向量模型的情形，这里略去具体代码，只显示运行结果，如表 22-5 所示。

表 22-5 基于 Glove6B100d 单词向量模型的相似单词

word	similar_words
king	{"ID": "["king", "prince", "queen", "monarch", "brother"]", "METRIC": "[1. 6858739404357614E-7, 4. 092165813215177, 4. 281252149113389, 4. 474171532331215, 4. 536668410815254]"}]
king-man+woman	{"ID": "["king", "queen", "monarch", "throne", "elizabeth"]", "METRIC": "[3. 364067897373334, 4. 081078579600477, 4. 642907390892788, 4. 905500707490234, 4. 921558914642319]"}]
queen	{"ID": "["queen", "princess", "elizabeth", "king", "lady"]", "METRIC": "[0. 0, 3. 8532465383426704, 4. 159615241047596, 4. 281252149113389, 4. 467098262191517]"}]
queen-woman+man	{"ID": "["queen", "king", "prince", "royal", "majesty"]", "METRIC": "[3. 364067897373337, 4. 0810785796004785, 5. 034758702597461, 5. 069448877059333, 5. 189214516972033]"}]

对比“king-man+woman”与“king”的结果，最相邻的有皇后“queen”和“elizabeth”；对比“queen-woman+man”与“queen”的结果，最相邻的是国王“king”和王子“prince”。

## 22.2 单词映射为向量

本节以四大名著之一的《三国演义》为例，将整本书的内容作为语料，使用 Word2Vec 算法将单词映射为向量，并使用向量距离分析各人物。

《三国演义》文本文件的下载地址为链接 22-3，里面不仅有《三国演义》，还有《水浒传》、《红楼梦》和《西游记》。感兴趣的读者可以仿照本节的方法，使用机器学习“读”一下其他的名著。

下载到本地后，使用文本编辑器打开，如图 22-3 所示。

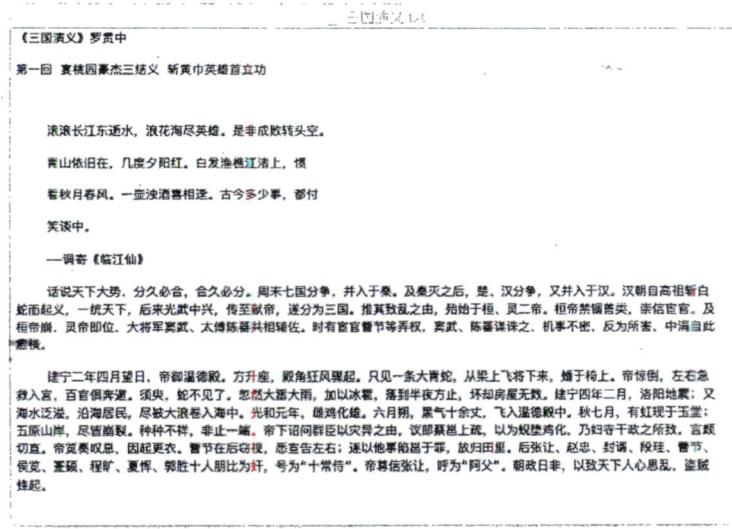


图 22-3

有了原始数据文件，如何读取它，使之成为数据源呢？我们可以将文档中的每个段落作为一条记录，即每个换行符分隔一条记录。Alink 提供了一个文本数据源组件，可以直接读取文本文件。

```
static final String ORIGIN_FILE = "三国演义.txt";
TextSourceBatchOp source = new TextSourceBatchOp().setFilePath(DATA_DIR + ORIGIN_FILE);
source.lazyPrint(8);
```

输出前 8 条数据，显示如下。

《三国演义》罗贯中  
第一回 宴桃园豪杰三结义 斩黄巾英雄首立功

滚滚长江东逝水，浪花淘尽英雄。是非成败转头空。  
青山依旧在，几度夕阳红。白发渔樵江渚上，惯  
看秋月春风。一壶浊酒喜相逢。古今多少事，都付  
笑谈中。  
——调寄《临江仙》

接下来做一下分析前的预处理工作，先使用分词组件将句子拆分为单词，然后使用停用词组件过滤标点符号。我们先使用 WordCountBatchOp 组件统计单词出现的频率，具体代码如下。

列出一些常用的人名，存放在字符串数组 CHARACTER\_DICT 中，用于为分词的自定义词典参数 UserDefinedDict 的赋值；计算完词频后，按词频降序选择前 100 个并输出。

```
final String[] CHARACTER_DICT = new String[]{
    "曹操", "孔明", "玄德", "刘玄德", "刘备", "关羽", "张飞",
    "赵云", "曹孟德", "诸葛亮", "张郃", "孙权", "张辽", "鲁肃"
};

source
    .link(
        new SegmentBatchOp()
            .setSelectedCol("text")
            .setUserDefinedDict(CHARACTER_DICT)
    )
    .link(
        new StopWordsRemoverBatchOp()
            .setSelectedCol("text")
    )
    .link(
        new WordCountBatchOp()
            .setSelectedCol("text")
    )
    .orderBy("cnt", 100, false)
    .print();
```

运行结果如表 22-6 所示，“曰”出现的次数最多，有 7558 次；在各人物中，“曹操”出现的次数最多，有 908 次，而“操”也常用来指代“曹操”，出现了 665 次，可见曹操是三国演义中出场次数最多的人物；紧随其后的是“玄德”和“孔明”。通过词频统计结果可以看出，我们使用的停用词过滤主要针对现代文，对于古文中的一些词，例如“曰”“吾”“皆”等，还需要扩充停用词词典再进行处理。

表 22-6 单词出现的频率

Cword	cnt
曰	7558
吾	1916
皆	1132
去	1100
曹操	908
不	890
见	826
遂	818

续表

C cword	cnt
人	796
将军	759
玄德	752
孔明	731
操	665
今	664
欲	651

下面是本节的核心内容，使用 Word2VecTrainBatchOp 组件训练单词向量模型。扩充了停用词表，设置参数 MinCount=10，即只计算词频数在 10 以上的单词，选择训练迭代次数为 50，并将结果保存为 AK 格式数据文件。具体代码如下。

```

source
    .link(
        new SegmentBatchOp()
            .setSelectedCol("text")
            .setUserDefinedDict(CHARACTER_DICT)
    )
    .link(
        new StopWordsRemoverBatchOp()
            .setSelectedCol("text")
            .setStopWords(
                "亦", "曰", "遂", "吾", "已", "去", "二人", "今", "使", "中", "知",
                "不", "见", "都", "令", "却", "欲", "请", "人", "谓", "不可", "闻",
                "前", "后", "皆", "便", "问", "日", "时", "耳", "不敢", "问", "回", "才",
                "之事", "之人", "之时", "料", "今日", "令人", "受", "说", "出", "已毕",
                "不得", "使人", "众", "何不", "不知", "再", "处", "无", "即日", "诸", "此时",
                "只", "下", "还", "上", "杀", "将军", "却说", "兵", "汝", "走", "言", "寨",
                "不能", "斩", "死", "商议", "听", "军士", "军", "左右", "军马", "引兵", "次日",
                "二", "看", "耶", "退", "更", "毕", "正", "一人", "原来", "大笑", "车胄", "口",
                "引", "大喜", "其事", "助", "事", "未", "大", "至此", "迄", "心中", "敢"
            )
    )
    .link(
        new Word2VecTrainBatchOp()
            .setSelectedCol("text")
            .setMinCount(10)
            .setNumIter(50)
    )
}

```

```

)
.link(
    new AkSinkBatchOp()
        .setFilePath(DATA_DIR + W2V_MODEL_FILE)
);

```

训练完成后，我们再导入模型数据，使用向量最近邻组件，看一下这些人物的关系。具体代码如下。

```

AkSourceBatchOp word2vec
= new AkSourceBatchOp().setFilePath(DATA_DIR + W2V_MODEL_FILE);

new VectorNearestNeighbor()
.setIdCol("word")
.setSelectedCol("vec")
.setTopN(20)
.setOutputCol("similar_words")
.fit(word2vec)
.transform(
    word2vec.filter("word IN ('曹操', '操', '玄德', '刘备', '孔明', "
        + "'亮', '卧龙', '周瑜', '吕布', '貂蝉', '云长', '孙权')")
)
.select("word, similar_words")
.print();

```

整理输出内容如表 22-7 所示，为了节省篇幅，这里省略了每个单词对应的距离指标，并将“曹操”和“操”、“玄德”和“刘备”的结果放在相邻位置，便于比对。

表 22-7 相似单词的计算结果

word	similar_words
曹操	{"ID": "["曹操", "玄德", "玄德曰", "徐州", "刘备", "怒", "其言", "郡", "云长", "吕布", "以为", "进兵", "守", "大军", "朝廷", "东吴", "此事", "许都", "操", "之兵"]"]}
操	{"ID": "["操", "玄德曰", "天子", "曹操", "玄德", "许都", "何如", "朝廷", "何故", "主公", "公", "徐州", "刘备", "大事", "破", "投", "忽", "吕布", "英雄", "坐"]"]}
玄德	{"ID": "["玄德", "玄德曰", "曹操", "徐州", "云长", "吕布", "许都", "投", "操", "坐", "刘备", "教", "丞相", "只得", "主公", "大事", "相见", "关公", "命", "引军"]"]}
刘备	{"ID": "["刘备", "徐州", "曹操", "主公", "玄德曰", "将士", "书", "急", "袁术", "孙策", "起兵", "之计", "攻", "朝廷", "郡", "东吴", "玄德", "其言", "进兵", "袁绍"]"]}
孔明	{"ID": "["孔明", "魏延", "先主", "孔明曰", "曹真", "东吴", "司马懿", "魏兵", "蜀", "军中", "马岱", "祁山", "奏", "关兴", "姜维", "懿", "魏", "以为", "成都", "中原"]"]}

续表

word	similar_words
亮	{"ID": "[\"亮\", \"权\", \"东吴\", \"甚\", \"都督\", \"鲁肃\", \"恪\", \"孙权\", \"诸葛\", \"内\", \"孙\", \"周瑜\", \"先主\", \"瑜\", \"道\", \"不见\", \"何故\", \"先生\", \"年\", \"罢\"]"}
卧龙	{"ID": "[\"卧龙\", \"叹\", \"先生\", \"年\", \"汉室\", \"道\", \"军中\", \"张\", \"瑜\", \"罢\", \"后人\", \"鲁肃\", \"玄德\", \"诗\", \"周瑜\", \"真\", \"入城\", \"城中\", \"马\", \"必\"]"}
吕布	{"ID": "[\"吕布\", \"布\", \"军中\", \"徐州\", \"玄德\", \"陈宫\", \"玄德曰\", \"张辽\", \"曹操\", \"曹兵\", \"城\", \"主公\", \"投\", \"相见\", \"云长\", \"只得\", \"城中\", \"袁术\", \"城下\", \"三军\"]"}
云长	{"ID": "[\"云长\", \"张飞\", \"玄德曰\", \"玄德\", \"何故\", \"张\", \"急\", \"关公\", \"徐州\", \"曹操\", \"飞\", \"吕布\", \"只见\", \"救\", \"主公\", \"厮杀\", \"关\", \"之计\", \"贼\", \"大叫\"]"}
孙权	{"ID": "[\"孙权\", \"江东\", \"东吴\", \"魏兵\", \"鲁肃\", \"三军\", \"提兵\", \"江南\", \"玄德曰\", \"权\", \"汉中\", \"救\", \"无不\", \"陆逊\", \"主公\", \"奏\", \"周瑜\", \"蜀\", \"之计\", \"先主\"]"}
周瑜	{"ID": "[\"周瑜\", \"鲁肃\", \"瑜\", \"之计\", \"军中\", \"曹操\", \"吴侯\", \"孙权\", \"东吴\", \"孙策\", \"刘备\", \"救\", \"江东\", \"将士\", \"命\", \"汉中\", \"云长\", \"厮杀\", \"提兵\", \"道\"]"}
貂蝉	{"ID": "[\"貂蝉\", \"卓\", \"妾\", \"董卓\", \"入\", \"罢\", \"允\", \"玄德曰\", \"布\", \"而来\", \"立于\", \"忽见\", \"军中\", \"吕布\", \"间\", \"叱\", \"岂\", \"天下\", \"肃\", \"否\"]"}

可以观察到如下信息。

- “曹操”和“操”的距离比较近，相近的单词内容也比较相似，都出现了“玄德”“玄德曰”“许都”“朝廷”“徐州”“吕布”。
- “玄德”和“刘备”的距离较近，都关联着“曹操”和“徐州”；但有趣的是，关联的单词中没有“诸葛亮”。
- 我们看一下诸葛亮的 3 个称呼：“孔明”“亮”和“卧龙”。“孔明”出现的频率最高，与其最相似的单词为“魏延”“司马懿”“曹真”“东吴”，还有一个词比较特殊——“先主”，应该是指刘备。“亮”最相近的单词有“权”“东吴”“鲁肃”“周瑜”，可能“亮”这个称呼在赤壁大战前后使用的比较多。与“卧龙”相近的词中出现了“玄德”，它和“先生”“汉室”的距离也较近。
- 与“云长”最近的是“张飞”“玄德曰”“玄德”，也有“关公”，当然也少不了“曹操”。

# 23

## 情感分析

本章主要讲解一个文本分析领域非常有用的应用——情感分析。我们在网上购买商品（或浏览电影信息、寻找饭馆信息）时，经常看到一个区域显示着用户的评论。我们会习惯性地参考其他人的评论，如果大家都说好，就更坚定了我们买下此商品的信心；如果有人给了差评，那么我们会很自然地想到，我要是购买了此商品是否也会遇到同样的问题。如何通过机器学习算法高效地判断评论的情感倾向呢？

我们选择一个在情感分析方面常用的公开数据集（参见链接 23-1），是用户对电影的评论，并有标签标明用户评论的核心意思是喜欢还是不喜欢。每条评论都比较长，这里选择了一条完整内容，显示如下。虽然有“good”和“laughed loud”等好评的词，但整篇读下来，可以知道用户并不看好这部电影，由此可见情感分析的难度。

Oh dear. good cast, but to write and direct is an art and to write wit and direct wit is a bit of a task. Even doing good comedy you have to get the timing and moment right. Im not putting it all down there were parts where i laughed loud but that was at very few times. The main focus to me was on the fast free flowing dialogue, that made some people in the film annoying. It may sound great while reading the script in your head but getting that out and to the camera is a different task. And the hand held camera work does give energy to few parts of the film. Overall direction was good but the script was not all that to me, but I'm sure you was reading the script in your head it would sound good. Sorry.

将数据文件[下载并解压](#)，目录结构如图 23-1 所示。

aclImdb		2019年5月19日 17:11	--	文件夹
imdb.vocab		2011年4月13日 01:14	846 KB	文档
imdbEr.txt		2011年6月12日 06:54	903 KB	纯文本文档
README		2011年6月26日 08:18	4 KB	文本编辑 文稿
test		2019年5月19日 17:13	--	文件夹
labeledBow.feat		2011年4月13日 01:25	20.2 MB	文档
neg		2011年4月12日 17:48	--	文件夹
pos		2011年4月12日 17:48	--	文件夹
urls_neg.txt		2011年4月12日 17:48	613 KB	纯文本文档
urls_pos.txt		2011年4月12日 17:48	613 KB	纯文本文档
train		2019年5月19日 17:11	--	文件夹
labeledBow.feat		2011年4月13日 01:17	21 MB	文档
neg		2011年4月12日 17:47	--	文件夹
pos		2011年4月12日 17:47	--	文件夹
unsup		2011年4月12日 17:47	--	文件夹
unsupBow.feat		2011年4月13日 01:22	41.3 MB	文档
urls_neg.txt		2011年4月12日 17:48	613 KB	纯文本文档
urls_pos.txt		2011年4月12日 17:48	613 KB	纯文本文档
urls_unsup.txt		2011年4月12日 17:47	2.5 MB	纯文本文档

图 23-1

其中，README 文件对数据集进行了详细介绍。数据集中包括原始文本数据，也包括从原始数据中使用词袋（bag of words）方式提取的特征数据。本章会先使用已提取特征的数据快速进行实验，得到模型效果；然后，对原始的英文数据进行预处理，展示完整的情感分析流程。

## 23.1 使用提供的特征

下面了解一下数据，使用词袋的方式将每个词作为一个特征，而该词在样本中出现的个数作为特征值。在具体的实现方式上，需要将所有出现的单词作为词汇表，放在文件 imdb.vocab 中，内容如图 23-2 所示。每行为一个单词，将词汇表的单词总数作为稀疏特征向量的维度，每个单词在词汇表的排序位置（从 0 开始）即为其在特征向量的索引位置。

imdb.vocab	
the	
and	
a	
of	
to	
is	
it	
in	
i	
this	
that	
was	

图 23-2

我们再用文本编辑器打开 train/labeledBow.feat 文件，如图 23-3 所示。

```
labeledBow.feat
9 0:9 1:1 2:4 3:4 4:6 5:4 6:2 7:2 8:4 10:4 12:2 26:1 27:1 28:1 29:2 32:1 41:1 45:1 47:1 50:1 54:2 57:1
59:1 63:2 64:1 66:1 68:2 70:1 72:1 78:1 100:1 106:1 113:1 122:1 125:1 136:1 140:1 142:1 150:1 167:1
183:1 201:1 207:1 208:1 213:1 217:1 230:1 255:1 321:5 343:1 357:1 370:1 390:2 468:1 514:1 571:1 619:1
671:1 766:1 877:1 1057:1 1179:1 1192:1 1402:2 1416:1 1477:2 1940:1 1941:1 2096:1 2243:1 2285:1 2379:1
2934:1 2938:1 3520:1 3647:1 4938:1 5138:4 5715:1 5726:1 5731:1 5812:1 8319:1 8567:1 10488:1 14239:1
26684:1 22489:1 24551:1 47384:1
7 0:7 1:4 2:2 3:2 5:4 6:1 8:2 9:2 14:1 16:1 18:1 20:1 22:1 24:1 27:1 29:1 34:1 36:3 37:2 41:1 42:1
48:3 52:2 79:1 91:1 100:1 106:1 119:1 154:1 166:1 172:1 207:1 288:1 321:1 336:1 353:1 365:1 383:1
386:2 390:1 429:1 504:1 517:1 706:1 729:1 748:1 911:1 1107:1 1195:1 1332:1 1433:1 1620:1 1662:1 2047:1
2365:1 2390:1 2673:1 3111:1 3238:1 3513:1 4065:1 4357:1 5138:1 5365:1 5884:1 6617:1 11823:1 11392:1
11798:1 12318:1 12831:1 13645:1 18316:1 22409:1 22496:1 27245:1 28840:1 30533:2 32289:1 33606:1
44792:1 48240:1 48538:1 53958:1 54537:1 54675:1 62726:1 71743:1 7281:1
9 0:4 1:4 2:4 3:7 4:2 5:1 6:1 7:1 9:1 10:1 13:1 14:1 19:1 20:2 22:1 25:1 27:1 29:1 34:1 39:1 42:1 44:1
49:1 76:1 91:1 97:1 101:1 118:1 123:1 148:1 156:1 171:1 176:1 237:1 242:1 253:1 276:1 288:1 321:2
342:1 345:1 370:1 390:1 509:1 680:1 682:1 840:2 868:1 872:2 916:1 972:1 1009:1 1010:1 1188:1 1195:1
1205:1 1477:1 1531:1 1790:1 1873:1 1998:1 2053:1 2067:1 2081:1 2099:1 2453:1 2511:1 2714:1 2924:1
3108:1 3531:1 3533:1 3781:1 4030:1 4263:1 5068:1 5138:1 5254:1 5268:2 6097:1 6611:1 8373:1 8950:1
15061:1 15594:1 20571:1 22409:2 23279:1 30533:1 48240:1 48538:1
10 0:10 1:2 2:2 4:3 5:2 6:4 7:2 9:1 10:4 11:1 16:1 17:1 19:3 20:3 24:2 26:1 27:1 28:1 29:2 32:1 36:1
39:2 46:2 49:1 53:1 55:1 61:4 69:1 70:1 73:1 75:1 89:1 128:1 125:2 133:1 137:1 138:1 139:1 151:1 174:1
292:3 295:1 371:1 394:1 499:2 566:1 563:1 592:1 673:1 693:1 709:2 862:1 1306:1 1341:1 1642:2 1755:1
2149:1 2210:1 2778:1 2860:1 4648:1 5480:1 7117:1 7384:1 7826:1 9364:1
```

图 23-3

这是 LIBSVM 格式的文件，关于该格式的详细介绍请参考 3.2.2 节。

可以看到，train/labeledBow.feat 文件中的单词索引是从 0 开始的，在词汇表文件 imdb.vocab 中可以找到与索引相对应的单词。譬如，文件中第一行出现的 0:9，表示词汇表 imdb.vocab 中的第一个单词（the）在该评论中出现了 9 次。

我们可以方便地使用 Alink 组件读取数据，需要设置文件的路径，因为 LIBSVM 文件通常以 1 作为起始索引，但是该数据以 0 作为起始索引，所以还需设置参数 StartIndex 为 0。具体代码如下所示。

```
static String ORIGIN_DATA_DIR = DATA_DIR + "aclImdb" + File.separator;
.....
BatchOperator <?> train_set = new LibSvmSourceBatchOp()
.setFilePath(ORIGIN_DATA_DIR + "train" + File.separator + "labeledBow.feat")
.setStartIndex(0);

train_set.lazyPrint(10, "train_set");
```

最后一行代码用于输出前 10 行数据，图 23-4 显示了部分结果。

label	features
8.0000	0:12.0 1:8.0 2:3.0 3:2.0 4:10.0 5:2.0 6:9.0 7:6.0 8:6.0 9:2.0 10:6.0 11:6.0 12:1.0 13:3.0
10.0000	0:17.0 1:18.0 2:14.0 3:15.0 4:8.0 5:10.0 7:5.0 9:1.0 10:4.0 12:8.0 13:5.0 14:5.0 16:1.0 1
8.0000	0:12.0 1:7.0 2:3.0 3:4.0 4:3.0 7:3.0 8:1.0 9:3.0 11:3.0 12:1.0 13:5.0 14:5.0 15:3.0 18:2.0
9.0000	0:18.0 1:16.0 2:9.0 3:10.0 4:10.0 5:8.0 6:7.0 7:12.0 8:4.0 9:7.0 10:3.0 11:2.0 12:6.0 13:2
7.0000	0:5.5 1:1.0 3:2.0 5:3.0 7:1.0 9:3.0 10:1.0 13:1.0 15:2.0 16:1.0 22:1.0 26:1.0 34:1.0 45:1.
7.0000	0:28.0 1:18.0 2:10.0 3:11.0 4:7.0 5:7.0 6:7.0 7:4.0 8:1.0 9:1.0 10:2.0 12:1.0 13:3.0 14:4.0
7.0000	0:15.0 1:6.0 2:8.0 3:10.0 4:5.0 5:4.0 6:3.0 7:6.0 8:5.0 9:5.0 10:2.0 11:2.0 12:3.0 14:1.0
7.0000	0:29.0 1:18.0 2:13.0 3:16.0 4:12.0 5:7.0 6:5.0 7:9.0 10:6.0 11:4.0 12:3.0 13:3.0 14:4.0 16
8.0000	0:66.0 1:34.0 2:21.0 3:29.0 4:29.0 5:19.0 6:5.0 7:9.0 9:3.0 10:13.0 11:7.0 12:11.0 13:6.0
7.0000	0:4.0 1:6.0 2:3.0 3:3.0 5:5.0 6:2.0 9:2.0 12:2.0 14:1.0 16:2.0 19:1.0 22:1.0 26:2.0 31:1.0

图 23-4

显然，数据分为两列，features 列为稀疏特征向量，label 列记录的是评分值。我们再看一下 label 列数据的分布情况，代码如下。

---

```
train_set
  .groupBy("label", "label, COUNT(label) AS cnt")
  .orderBy("label", 100)
  .lazyPrint(-1, "labels of train_set");
```

---

显示结果如下。

```
label|cnt
----|---
1.0000|5100
2.0000|2284
3.0000|2420
4.0000|2696
7.0000|2496
8.0000|3009
9.0000|2263
10.0000|4732
```

可以看到，最低分 1.0 出现的次数最多，为 5100；最高分 10.0 出现的次数也很多，为 4732；其他分值出现的次数比较平均；无法明显体现情感倾向的中间评分 5.0 和 6.0 均未出现。

同样，我们看一下预测集的情况。

---

```
BatchOperator <?> test_set = new LibSvmSourceBatchOp()
  .setFilePath(ORIGIN_DATA_DIR + "test" + File.separator + "labeledBow.feat")
  .setStartIndex(0);

test_set
  .lazyPrint(10, "test_set")
  .groupBy("label", "label, COUNT(label) AS cnt")
  .orderBy("label", 100)
  .lazyPrint(-1, "labels of test_set");
```

---

预测集 test\_set 的评分分布如下。

```
label|cnt
----|---
1.0000|5022
2.0000|2302
3.0000|2541
4.0000|2635
7.0000|2307
8.0000|2850
9.0000|2344
10.0000|4999
```

其评分分布情况与训练集大致相同，以 1.0 和 10.0 居多，其他分值出现的次数比较平均，无法明显体现情感倾向的中间评分 5.0 和 6.0 均未出现。

我们的目标是二分类问题，需要按评分多少转换为“pos”（正向评论、好评）或“neg”（负向评论、差评）标签值。使用字符串常量 `VECTOR_COL_NAME` 来统一向量特征列的命名。

```
private static final String LABEL_COL_NAME = "label";
private static final String VECTOR_COL_NAME = "vec";
...
train_set = train_set.select("CASE WHEN label>5 THEN 'pos' ELSE 'neg' END AS label, "
    + "features AS " + VECTOR_COL_NAME);
test_set = test_set.select("CASE WHEN label>5 THEN 'pos' ELSE 'neg' END AS label, "
    + "features AS " + VECTOR_COL_NAME);

train_set.lazyPrint(10, "train_set");
```

看一下变换后的训练数据集，向量数据较长，图 23-5 显示了部分数据。

label	vec
neg 0:7.0 1:2.0 3:3.0 4:1.0 5:2.0 6:5.0 9:1.0 10:1.0 15:1.0 22:1.0 24:1.0 27:4.0 28:1.0 33:2.0 34:1.0 35:1.0 37:1.	
neg 0:6.0 1:4.0 2:2.0 3:1.0 6:2.0 8:2.0 9:5.0 10:3.0 11:3.0 12:1.0 14:1.0 15:3.0 16:2.0 17:1.0 18:1.0 19:1.0 20:1.	
neg 0:1.0 1:1.0 3:3.0 4:1.0 5:2.0 6:1.0 7:1.0 9:3.0 10:1.0 13:2.0 15:1.0 16:1.0 22:1.0 24:1.0 25:1.0 31:3.0 33:1.0	
neg 0:3.0 1:3.0 2:3.0 3:2.0 4:2.0 6:1.0 7:1.0 8:2.0 9:4.0 10:1.0 11:3.0 12:1.0 13:3.0 15:2.0 16:3.0 17:2.0 18:1.0	
neg 0:2.0 1:6.0 2:4.0 3:2.0 4:1.0 5:1.0 6:2.0 8:5.0 9:2.0 10:3.0 14:1.0 15:3.0 16:1.0 18:1.0 19:1.0 21:1.0 24:1.0	
neg 0:12.0 1:8.0 2:11.0 3:7.0 4:6.0 5:12.0 6:5.0 7:6.0 8:2.0 9:6.0 10:3.0 11:2.0 12:2.0 13:3.0 14:2.0 15:8.0 16:2.	
neg 0:6.0 2:8.0 4:5.0 5:2.0 6:3.0 7:2.0 8:2.0 11:4.0 14:2.0 15:1.0 16:5.0 18:1.0 19:1.0 20:8.0 21:1.0 22:1.0 25:1.	
neg 0:12.0 1:7.0 2:5.0 3:3.0 4:8.0 5:2.0 6:7.0 7:8.0 9:1.0 10:1.0 12:1.0 13:1.0 14:1.0 15:2.0 16:2.0 17:1.0 18:1.0	
neg 0:24.0 1:12.0 2:11.0 3:10.0 4:8.0 5:5.0 6:5.0 7:1.0 9:4.0 10:6.0 11:1.0 12:3.0 14:4.0 15:1.0 16:5.0 17:4.0 18:	
neg 0:10.0 1:5.0 2:8.0 3:6.0 4:5.0 5:7.0 6:4.0 7:2.0 8:1.0 9:3.0 10:4.0 12:5.0 13:2.0 14:4.0 15:5.0 16:4.0 19:2.0	

图 23-5

### 23.1.1 使用朴素贝叶斯方法

针对文本的朴素贝叶斯分类器与通用的朴素贝叶斯分类器是有区别的，一个明显的不同之处是输入的数据格式不同，通用的朴素贝叶斯分类器支持数据列的输入，每个数据列都是数值型或枚举型数据；针对文本的朴素贝叶斯分类器输入的数据为稀疏向量形式，向量的维度较大，一般为语料中不同单词的个数。

针对文本的朴素贝叶斯算法有如下两种。

- 多项式朴素贝叶斯（Multinomial Naive Bayes）：特征向量  $x = (x_1, x_2, \dots, x_m)$ ，其中各分量为对应单词在该文档中出现的个数；设  $p_{ki}$  为第  $i$  个单词出现在第  $k$  个类别的概率，则  $x$  属于第  $k$  个类别的可能性为

$$P(x|C_k) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_i p_{ki}^{x_i}$$

- 伯努利朴素贝叶斯 (Bernoulli Naive Bayes)：特征向量  $\mathbf{x} = (x_1, x_2, \dots, x_m)$ ，其中各分量的取值为 0 或 1，代表对应的单词是否在该文档中出现；设  $p_{ki}$  为第  $i$  个单词出现在第  $k$  个类别的概率，则  $\mathbf{x}$  属于第  $k$  个类别的可能性为

$$P(\mathbf{x}|C_k) = \prod_i p_{ki}^{x_i} (1 - p_{ki})^{(1-x_i)}$$

可以看到它们的差异，在输入数据的形式方面，多项式朴素贝叶斯以单词出现的次数为特征值，为非负整数；伯努利朴素贝叶斯以 0 或 1 为特征值，表示单词是否出现。从类别可能性的公式上看，在多项式朴素贝叶斯中，单词出现的次数对分类可能性有影响；而在伯努利朴素贝叶斯中，单词出现 1 次和多次对分类可能性的影响是一样的。

当前训练、预测数据集的特征向量的各个分量值正是对应单词在当前文档中出现的个数，所以我们可以直接使用 `NaiveBayesTextClassifier` 组件，设置 `ModelType` 参数为 “Multinomial”，尝试多项式朴素贝叶斯算法。

---

```

new NaiveBayesTextClassifier()
  .setModelType("Multinomial")
  .setVectorCol(VECTOR_COL_NAME)
  .setLabelCol(LABEL_COL_NAME)
  .setPredictionCol(PREDICTION_COL_NAME)
  .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
  .enableLazyPrintModelInfo()
  .fit(train_set)
  .transform(test_set)
  .link(
    new EvalBinaryClassBatchOp()
      .setPositiveLabelValueString("pos")
      .setLabelCol(LABEL_COL_NAME)
      .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
      .lazyPrintMetrics("NaiveBayesTextClassifier + Multinomial")
  );
BatchOperator.execute();

```

---

贝叶斯模型信息显示如下。

---

```

----- NaiveBayesTextModelInfo -----
===== model meta info =====
{model type: Multinomial, vector size: 89527, vector col name: vec}
===== label proportion information =====
|neg|pos|
|---|---|
|0.5|0.5|

```

---

```
===== feature probability information =====
```

vector index	neg	pos
0	0.0552	0.0573
1	0.0251	0.0296
2	0.0268	0.0276
...	...	...
89524	0	0
89525	0	0
89526	0	0

计算结果如下。

Metrics:		
AUC: 0.8905	Accuracy: 0.8136	Precision: 0.8591
Recall: 0.7504	F1: 0.8011	LogLoss: 1.6084
Pred\Real	pos	neg
pos	9380	1539
neg	3120	10961

再尝试使用伯努利朴素贝叶斯算法，仍然使用 NaiveBayesTextClassifier 组件，需要设置 ModelType 参数为“Bernoulli”。因为伯努利朴素贝叶斯算法要求特征向量的分量值只能为 0 或 1，所以我们需要加上预处理组件 Binarizer，以某个阈值为界，将数据转化为 0 和 1（小于等于阈值会被转化为 0，大于阈值则为 1），该组件默认的阈值为 0。另外，为了查看该组件处理结果，可以使用 enableLazyPrintTransformData 方法。

---

```

new Pipeline()
    .add(
        new Binarizer()
            .setSelectedCol(VECTOR_COL_NAME)
            .enableLazyPrintTransformData(5, "After Binarizer")
    )
    .add(
        new NaiveBayesTextClassifier()
            .setModelType("Bernoulli")
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .enableLazyPrintModelInfo()
    )
    .fit(train_set)
    .transform(test_set)
    .link()

```

---

---

```

new EvalBinaryClassBatchOp()
    .setPositiveLabelValueString("pos")
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .lazyPrintMetrics("Binarizer + NaiveBayesTextClassifier + Bernoulli")
);

```

---

模型信息显示如下。

```

===== NaiveBayesTextModelInfo =====

===== model meta info =====
{model type: Bernoulli, vector size: 89527, vector col name: vec}
===== label proportion information =====

| pos | neg |
| --- | --- |
| 0.5 | 0.5 |

===== feature probability information =====

| vector index | pos | neg |
| --- | --- | --- |
| 0 | 0.0068 | 0.0069 |
| 1 | 0.0067 | 0.0066 |
| 2 | 0.0066 | 0.0067 |
| ... | ... | ... |
| 89524 | 0 | 0 |
| 89525 | 0 | 0 |
| 89526 | 0 | 0 |

```

计算结果如下，各项指标要优于使用多项式朴素贝叶斯算法。

```

Metrics:
AUC:0.908 Accuracy:0.8274 Precision:0.8674 Recall:0.7729 F1:0.8174 LogLoss:1.0788
|Pred\Real| pos | neg |
| --- | --- | --- |
| pos | 9661 | 1477 |
| neg | 2839 | 11023 |

```

### 23.1.2 使用逻辑回归算法

使用朴素贝叶斯文本分类器可以快速获取一个不错的模型，我们也可以尝试使用 LogisticRegression 分类器，具体代码如下。

---

```

new LogisticRegression()
    .setVectorCol(VECTOR_COL_NAME)

```

---

```

.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.setPredictionDetailCol(PRED_DETAIL_COL_NAME)
.enableLazyPrintTrainInfo("< LR train info >")
.enableLazyPrintModelInfo("< LR model info >")
.fit(train_set)
.transform(test_set)
.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("pos")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics("LogisticRegression")
);

```

计算结果如下，相比于文本朴素贝叶斯算法，各项指标又有所提升。

Metrics:		
AUC:0.94	Accuracy:0.873	Precision:0.8649
Recall:0.8842	F1:0.8744	LogLoss:0.8491
Pred\Real	pos	neg
pos	11052	1727
neg	1448	10773

逻辑回归训练过程的信息如下。

train meta info			
{model name: Logistic Regression, num feature: 89527}			
train importance info			
colName	importanceValue	colName	weightValue
427	1.82374809	6126	11.24306434
77	1.78636621	9808	10.48445417
240	1.63787453	8898	9.63473157
...	...	...	...
21750	0.00000059	8157	-10.27395642
44833	0.00000050	7339	-11.29044369
46409	0.00000036	4111	-13.70695642
train convergence info			
step:0	loss:0.69288129	gradNorm:0.05158780	learnRate:0.40000000
step:1	loss:0.58087868	gradNorm:0.05154487	learnRate:1.60000000
step:2	loss:0.42480889	gradNorm:0.05234910	learnRate:4.00000000
...	...	...	...
step:97	loss:0.01554132	gradNorm:0.00131275	learnRate:4.00000000
step:98	loss:0.01439589	gradNorm:0.00141001	learnRate:4.00000000
step:99	loss:0.01365725	gradNorm:0.00162553	learnRate:4.00000000

在第三部分中，显示了每次迭代中 3 个重要的指标值（loss、gradNorm、learnRate）。此次训练是到了最大迭代次数（默认值 100）而终止的，而在终止的时候，loss 的值还在不断下降，

gradNorm 和 learnRate 的值也不小。这时可能需要调高迭代次数，使 loss 值收敛；可能训练过程有些“过拟合”，过多次的迭代可以获得更低的 loss 值，但我们关心的 AUC、Accuracy 等指标不一定是最优的。此时最好的办法是进行超参数搜索，确定较优的参数。

使用交叉验证方式网格搜索参数，具体代码如下。

```
AlinkGlobalConfiguration.setPrintProcessInfo(true);

LogisticRegression lr = new LogisticRegression()
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME);

GridSearchCV gridSearch = new GridSearchCV()
    .setEstimator(
        new Pipeline().add(lr)
    )
    .setParamGrid(
        new ParamGrid()
            .addGrid(lr, LogisticRegression.MAX_ITER,
                new Integer[] {10, 20, 30, 40, 50, 60, 80, 100})
    )
    .setTuningEvaluator(
        new BinaryClassificationTuningEvaluator()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .setTuningBinaryClassMetric(TuningBinaryClassMetric.AUC)
    )
    .setNumFolds(6)
    .enableLazyPrintTrainInfo();

GridSearchCVModel bestModel = gridSearch.fit(train_set);

bestModel
    .transform(test_set)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("pos")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("LogisticRegression")
    );
BatchOperator.execute();
```

单独定义 LogisticRegression 实例 lr，便于后面指定调参对象。通过组件 GridSearchCV 进行网格搜索，并设置交叉验证的折数（NumFolds）为 6，设置搜索参数的列表，最大迭代次数

(`MAX_ITER`) 会尝试{10, 20, 30, 40, 50, 60, 80, 100}。还有一个重要参数就是调参的目标，在 `TuningEvaluator` 中设置，这里我们选择了 AUC。最后，使用搜索出来的最优参数对整个训练数据集进行训练，得到 `bestModel`，再用测试集进行评估。

需要注意的是，网格搜索所需时间较长，我们一般会允许组件输出中间结果信息，便于我们看到搜索的进展，运行 `AlinkGlobalConfiguration` 的 `setPrintProcessInfo(true)` 方法即可。需要屏幕显示网格搜索结果时可以使用 `enableLazyPrintTrainInfo` 方法。

网格搜索的结果如下，最大迭代次数为 30 时，AUC 的指标更高。

```
Metric information:
Metric name: AUC
Larger is better: true
Tuning information:
+-----+-----+-----+
|       AUC|      stage| param|value|
+-----+-----+-----+
| 0.9495710102806689|LogisticRegression|maxIter| 30|
| 0.9480044722419051|LogisticRegression|maxIter| 40|
| 0.9478236485040631|LogisticRegression|maxIter| 20|
| 0.9474024334078258|LogisticRegression|maxIter| 60|
| 0.9471031004488054|LogisticRegression|maxIter| 50|
| 0.943939344834118|LogisticRegression|maxIter| 80|
| 0.9426986353478771|LogisticRegression|maxIter| 100|
| 0.9293614922795674|LogisticRegression|maxIter| 10|
```

对于使用最大迭代次数 30 训练出来的 `bestModel`，其评估结果如下，明显优于前面使用默认最大迭代次数 100 得到的结果。

```
Metrics:
AUC:0.9472 Accuracy:0.8814 Precision:0.8868    Recall:0.8744   F1:0.8806 LogLoss:0.3727
+-----+-----+
|Pred\Real| pos | neg |
+-----+-----+
| pos | 10930 | 1395 |
| neg | 1570 | 11105 |
```

另外，如果将调参目标变为 Accuracy，可以得到如下的搜索结果，仍然是当最大迭代次数为 30 时结果最优。

```
Metric information:
Metric name: Accuracy
Larger is better: true
Tuning information:
+-----+-----+-----+
|       Accuracy|      stage| param|value|
+-----+-----+-----+
| 0.8880398353537382|LogisticRegression|maxIter| 30|
| 0.8859196401109802|LogisticRegression|maxIter| 40|
```

0.8833198384935733 LogisticRegression maxIter  60
0.8821201265012576 LogisticRegression maxIter  50
0.8815198768736043 LogisticRegression maxIter  80
0.8809601136853843 LogisticRegression maxIter  100
0.8781601840551746 LogisticRegression maxIter  20
0.8568795725338613 LogisticRegression maxIter  10

## 23.2 如何提取特征

在23.1节中，我们对如何基于已经抽取的特征进行了建模和预测。本节我们会进一步讲解如何从原始文本数据中提取特征。原始评论数据的每条样本都是一个文本文件，正负样本分别位于pos文件夹和neg文件夹中。提取文本内容和标签后，将数据保存为AK格式文件，便于后续多次调用该训练和测试数据集。具体代码如下。

```
ArrayList<Row> trainRows = new ArrayList<>();
ArrayList<Row> testRows = new ArrayList<>();

for (String label : new String[] {"pos", "neg"}) {
    File subfolder = new File(ORIGIN_DATA_DIR + "train" + File.separator + label);
    for (File f : subfolder.listFiles()) {
        trainRows.add(Row.of(label, readFileContent(f)));
    }
}
for (String label : new String[] {"pos", "neg"}) {
    File subfolder = new File(ORIGIN_DATA_DIR + "test" + File.separator + label);
    for (File f : subfolder.listFiles()) {
        testRows.add(Row.of(label, readFileContent(f)));
    }
}

new MemSourceBatchOp(trainRows, COL_NAMES)
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + TRAIN_FILE)
    );
new MemSourceBatchOp(testRows, COL_NAMES)
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + TEST_FILE)
    );
BatchOperator.execute();
```

使用前面保存的 AK 格式文件构建训练和测试数据源，并输出 5 条数据，查看数据内容。具体代码如下。

```
AkSourceBatchOp train_set = new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE);
AkSourceBatchOp test_set = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);

train_set.lazyPrint(5);
```

由于每条样本的文本内容较多，这里只显示其中的两条，如表 23-1 所示。

表 23-1 两条样本数据

label	review
pos	This was a good film with a powerful message of love and redemption. I loved the transformation of the brother and the repercussions of the horrible disease on the family. Well-acted and well-directed. If there were any flaws, I'd have to say that the story showed the typical suburban family and their difficulties again. What about all people of all cultural backgrounds? I would love to see a movie where all of these cultures are shown - like in real life. Nevertheless, the film soared in terms of its values and its understanding of the how a disease can bring someone closer to his or her maker. Loved the film and it brought tears to my eyes
pos	Hello. This movie is.....well.....okay. Just kidding! ITS AWESOME! It's NOT a Block Buster smash hit. It's not meant to be. But its a big hit in my world. And my sisters. We are rockin' Rollers. GO RAMONES!!!! This is a great movie..... For ME!

有了英文文本数据，构造 23.1 节中所用的特征向量就很简单了。首先使用组件 RegexTokenizer，得到空格分隔的英文单词序列；然后使用组件 DocCountVectorizer，计算各文档中单词出现的数目，并根据指定的类型产生特征向量，这里选择的是“WORD\_COUNT”，即计算文档内各单词出现的次数；最后，直接对生成的特征向量进行建模，通过模型的评估指标验证我们所生成的特征向量的有效性。具体代码如下。

```
new Pipeline()
    .add(
        new RegexTokenizer()
            .setPattern("\\\\W+")
            .setSelectedCol(TXT_COL_NAME)
    )
    .add(
        new DocCountVectorizer()
            .setFeatureType("WORD_COUNT")
            .setSelectedCol(TXT_COL_NAME)
            .setOutputCol(VECTOR_COL_NAME)
            .enableLazyPrintTransformData(5)
    )
    .add(
```

---

```

new LogisticRegression()
    .setMaxIter(30)
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
)
.fit(train_set)
.transform(test_set)
.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("pos")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics("DocCountVectorizer")
);

```

---

计算评估结果如下，与直接使用已提供的特征向量训练得到的模型的效果相当，从而验证了我们构造的特征向量的有效性。

Metrics:		
AUC:0.9479	Accuracy:0.882	Precision:0.8754
Recall:0.8909	F1:0.8831	LogLoss:0.3467
Pred\Real	pos	neg
pos	11136	1585
neg	1364	10915

另外，我们还可以通过 hash 方式同样实现单词到向量索引的映射。使用 DocHashCountVectorizer 组件，同样是用模型的评估指标验证我们所生成的特征向量的有效性，具体代码如下。

---

```

new Pipeline()
    .add(
        new RegexTokenizer()
            .setPattern("\\W+")
            .setSelectedCol(TXT_COL_NAME)
    )
    .add(
        new DocHashCountVectorizer()
            .setFeatureType("WORD_COUNT")
            .setSelectedCol(TXT_COL_NAME)
            .setOutputCol(VECTOR_COL_NAME)
            .enableLazyPrintTransformData(5)
    )
    .add(
        new LogisticRegression()
            .setMaxIter(30)
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
    )

```

---

---

```

.setPredictionCol(PREDICTION_COL_NAME)
.setPredictionDetailCol(PRED_DETAIL_COL_NAME)
)
.fit(train_set)
.transform(test_set)
.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("pos")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics("DocHashCountVectorizer")
);
BatchOperator.execute();

```

---

使用 hash 方式计算得到的评估结果如下，在评估指标上，与精确匹配方式得到的结果相当，这就验证了使用 hash 方式生成特征向量的有效性。

---

Metrics:		
AUC: 0.9477	Accuracy: 0.883	Precision: 0.8816
Recall: 0.8849	F1: 0.8833	LogLoss: 0.351
Pred\Real	pos	neg
-----	-----	-----
pos	11061	1485
neg	1439	11015

---

### 23.3 构造更多特征

特征生成方法主要利用了单词是否在该文档中出现的性质，并没有考虑各单词之间的先后关系，本节将利用 NGram 构造新的特征。

将文档看作单词的序列，NGram 是将序列中相邻的  $n$  个单词作为一个片段，从片段的出现次数上可以看出哪些单词组合在语料中最为常用。就像每个单词可以作为特征向量的一个分量，每个片段也可以作为特征向量的一个分量。先进行分词，按单词的出现次数生成特征向量；然后使用 NGram 组件，设置参数  $N=2$ ，即相邻两个单词作为一个片段；再将片段当作单词看待，为了对产生的数据有一个直观印象，这里激活了该组件处理结果的输出功能。同样可以使用 DocCountVectorizer 组件生成特征向量，最后使用 VectorAssembler 组件，将两个生成的特征向量组装为一个；接着和前面介绍的流程一样，使用逻辑回归算法进行训练、预测和评估模型，具体代码如下。为了节省篇幅，这里略去一些前面章节出现过的代码。

---

```

New Pipeline()
.add(

```

---

```

new RegexTokenizer()
    .setPattern("\w+")
    .setSelectedCol(TXT_COL_NAME)
)
.add(
    new DocCountVectorizer()
        .setFeatureType("WORD_COUNT")
        .setSelectedCol(TXT_COL_NAME)
        .setOutputCol(VECTOR_COL_NAME)
)
.add(
    new NGram()
        .setN(2)
        .setSelectedCol(TXT_COL_NAME)
        .setOutputCol("v_2")
        .enableLazyPrintTransformData(1, "2-gram")
)
.add(
    new DocCountVectorizer()
        .setFeatureType("WORD_COUNT")
        .setSelectedCol("v_2")
        .setOutputCol("v_2")
)
.add(
    new VectorAssembler()
        .setSelectedCols(VECTOR_COL_NAME, "v_2")
        .setOutputCol(VECTOR_COL_NAME)
)
.add(new LogisticRegression()... ...)
.fit(train_set)
.transform(test_set)
.link(new EvalBinaryClassBatchOp()... ...);
BatchOperator.execute();

```

2-gram 操作后，输出了一条数据，整理如表 23-2 所示，左边是其所在的数据列名称，右边是数据值。

表 23-2 一条数据的各列取值

label	pos
review	i really like this show it has drama romance and comedy all rolled into one i am 28 and i am a married mother so i can identify both with lorelei s and rory s experiences in the show i have been watching mostly the repeats on the family channel lately so i am not up to date on what is going on now i think females would like this show more than males but i know some men out there would enjoy it i really like that is an hour long and not a half hour as th hour seems to fly by when i am watching it give it a chance if you have never seen the show i think lorelei and luke are my favorite characters on the show though mainly because of the way they are with one another how could you not see something was there or take that long to see it i guess i should say br br happy viewing

续表

label	pos
vec	\$74629\$0:6.0 1:5.0 2:3.0 3:1.0 4:3.0 5:2.0 6:2.0 7:5.0 8:1.0 9:13.0 10:2.0 11:2.0 12:2.0 13:1.0 14:1.0 16:2.0 18:1.0 21:2.0 22:4.0 23:3.0 25:2.0 27:2.0 29:2.0 30:1.0 32:1.0 33:1.0 34:1.0 36:2.0 38:3.0 39:2.0 41:1.0 44:1.0 45:1.0 46:1.0 47:1.0 48:1.0 50:1.0 51:1.0 53:1.0 55:1.0 59:1.0 60:2.0 64:2.0 65:2.0 72:1.0 76:1.0 82:1.0 85:1.0 86:1.0 93:1.0 96:1.0 101:2.0 102:1.0 107:1.0 113:1.0 120:5.0 121:1.0 132:1.0 141:1.0 142:1.0 147:2.0 148:1.0 150:1.0 160:1.0 169:1.0 185:1.0 191:1.0 194:2.0 197:1.0 201:1.0 208:1.0 213:1.0 242:4.0 319:1.0 336:1.0 356:1.0 420:1.0 451:1.0 480:1.0 508:1.0 528:3.0 569:1.0 636:1.0 660:1.0 822:1.0 872:1.0 1006:1.0 1273:1.0 1289:1.0 1423:1.0 2031:1.0 2211:1.0 2486:1.0 3481:1.0 4466:1.0 5027:1.0 5232:1.0 5721:1.0 6263:1.0 7568:1.0 10706:1.0 10978:1.0
v_2	i_really really_like like_this this_show show_it it_has has_drama drama_romance romance_and and_comedy comedy_all all_rolled rolled_into into_one one_i i_am am_28 28_and and_i i_am am_a a_married married_mother mother_so so_i i_can can_identify identify_both both_with with_lorelei lorelei_s s_and and_rory rory_s s_experiences experiences_in in_the the_show show_i i_have have_been been_watching watching_mostly mostly_the the_repeats repeats_on on_the the_family family_channel channel_lately lately_so so_i_am am_not not_up up_to to_date date_on on_what what_is is_going going_on on_now now_i i_think think_females females_would would_like like_this this_show show_more more_than than_males males_but but_i i_know know_some some_men men_out out_there there_would would_enjoy enjoy_it it_i i_really really_like like_that that_is is_an an_hour hour_long long_and and_not not_a a_half half_hour hour_as as_th th_hour hour_seems seems_to to_fly fly_by by_when when_i i_am am_watching watching_it it_give give_it it_a a_chance chance_if if_you you_have have_never never_seen seen_the the_show show_i i_think think_lorelei lorelei_and and_luke luke_are are_my my_favorite favorite_characters characters_on on_the the_show show_though though_mainly mainly_because because_of of_the the_way way_they they_are are_with with_one one_another another_how how_could could_you you_not not_see see_something something_was was_there there_or or_take take_that that_long long_to to_see see_it it_i i_guess guess_i i_should should_say say_br br br_happy happy_viewing

其中 v\_2 列就是对原始文本数据进行 2-gram 的结果，相邻的两个单词用下画线“\_”连接起来，成为一个片段，片段之间使用空格分隔。这里将片段看作“单词”，还可以使用 DocCountVectorizer 组件生成特征向量。

最后的评估结果如下，AUC 和 Accuracy 指标有明显提升，说明了新特征的有效性。

## Metrics:

AUC:0.9603	Accuracy:0.893	Precision:0.9407	Recall:0.839	F1:0.8869	LogLoss:0.6826
<hr/>					
Pred\Real	pos	neg			
----- ----- -----					
pos   10487   661					
neg   2013   11839					

既然 2-gram 产生了作用，那么再增加片段长度，譬如 3-gram，是否能获得更好的效果呢？

在 2-gram 代码的基础上，增加 3-gram 特征向量的生成，并和其他两个特征向量一起组装成一个向量，具体代码如下。

---

```

new Pipeline()
    .add(new RegexTokenizer()... ...)
    .add(new DocCountVectorizer()... ...)
    .add(new NGram()... ...)
    .add(new DocCountVectorizer()... ...)
    .add(
        new NGram()
            .setN(3)
            .setSelectedCol(TXT_COL_NAME)
            .setOutputCol("v_3")
    )
    .add(
        new DocCountVectorizer()
            .setFeatureType("WORD_COUNT")
            .setVocabSize(10000)
            .setSelectedCol("v_3")
            .setOutputCol("v_3")
    )
    .add(
        new VectorAssembler()
            .setSelectedCols(VECTOR_COL_NAME, "v_2", "v_3")
            .setOutputCol(VECTOR_COL_NAME)
    )
    .add(new LogisticRegression()... ...)
    .fit(train_set)
    .transform(test_set)
    .link(new EvalBinaryClassBatchOp()... ...);
BatchOperator.execute();

```

---

注意，由于不同的 3-gram 片段的数量较大，我们只使用出现频次较高的那些片段，在 DocCountVectorizer 组件设置了 VocabSize 为 10000。

运行得到的评估结果如下，AUC 和 Accuracy 指标再一次被提升。

## Metrics:

AUC:0.9641	Accuracy:0.9057	Precision:0.9039	Recall:0.9078	F1:0.9059	LogLoss:0.4472
<hr/>					
Pred\Real	pos	neg			
----- ----- -----					
pos   11348   1206					
neg   1152   11294					

## 23.4 模型保存与预测

前面的实验流程都是基于 Pipeline 的，而 Pipeline 执行 fit 方法就得到了 PipelineModel，我们可以使用 save 方法将模型保存到文件系统。这里省略了 Pipeline 构建的代码，save 方法不会触发任务执行，所以在其后显式调用 execute 函数，具体代码如下。

```
static String PIPELINE_MODEL = "pipeline_model.ak";
...
new Pipeline()
    .add(new RegexTokenizer()... ...)
    ...
    .add(new LogisticRegression()... ...)
    .fit(train_set)
    .save(DATA_DIR + PIPELINE_MODEL);
BatchOperator.execute();
```

运行成功后，我们可以在相应路径中看到新生成的模型文件。

### 23.4.1 批式/流式预测任务

对于批式/流式预测任务，我们需要从模型文件中导入 PipelineModel，可以使用如下代码。

```
PipelineModel pipeline_model = PipelineModel.load(DATA_DIR + PIPELINE_MODEL);
```

对于批式数据，可以直接使用 PipelineModel 的 transform 方法进行预测，后接预测评估组件，检验模型效果，具体代码如下。

```
AkSourceBatchOp test_set = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);
pipeline_model
    .transform(test_set)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("pos")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("NGram 2 and 3")
    );
BatchOperator.execute();
```

其评估结果如下所示，与前面批式实验的结果基本一致。

---

Metrics:					
AUC: 0.9644	Accuracy: 0.9056	Precision: 0.9185	Recall: 0.8901	F1: 0.9041	LogLoss: 0.4387

---

Pred\Real	pos	neg
pos	11126	987
neg	1374	11513

对于流式数据，我们也可以直接使用 PipelineModel 的 transform 方法，输入为流式数据，输出的预测结果也为流式数据。由于数据较多，我们进行了采样（采样率为 0.001），并选择输出最主要的 3 列，具体代码如下。

```
AkSourceStreamOp test_stream = new AkSourceStreamOp().setFilePath(DATA_DIR + TEST_FILE);
pipeline_model
    .transform(test_stream)
    .sample(0.001)
    .select(PREDICTION_COL_NAME + ", " + LABEL_COL_NAME + ", " + TXT_COL_NAME)
    .print();
StreamOperator.execute();
```

可以看到，随着流式任务的运行，会不断输出预测结果。

### 23.4.2 嵌入式预测

在实际应用中，我们经常需要将机器学习模型预测嵌入应用中，通过 SDK 方式直接调用。Alink 提供的 LocalPredictor 就是为了解决此问题的，LocalPredictor 的输入和输出都为 Flink Row 类型。

PipelineModel 提供了 collectLocalPredictor 方法，可以直接得到 LocalPredictor 的实例。在 Pipeline 中，通过数据列名称来指定所要处理的数据，我们输入的 Row 类型数据没有列名信息，所以需要在构建 LocalPredictor 时加入当前输入数据的列名和类型信息，一般使用 Schema String 方式描述（详见 2.8 节）。具体代码如下。

```
String str
= "Oh dear. good cast, but to write and direct is an art and to write wit and direct wit is a bit of a "
+ "task. Even doing good comedy you have to get the timing and moment right. I'm not putting it all down "
+ "there were parts where I laughed loud but that was at very few times. The main focus to me was on the "
+ "fast free flowing dialogue, that made some people in the film annoying. It may sound great while "
+ "reading the script in your head but getting that out and to the camera is a different task. And the "
+ "hand held camera work does give energy to few parts of the film. Overall direction was good but the "
+ "script was not all that to me, but I'm sure you was reading the script in your head it would sound good"
+ ". Sorry.";

Row pred_row;

LocalPredictor local_predictor = pipeline_model.collectLocalPredictor("review string");

System.out.println(local_predictor.getOutputSchema());
```

---

```
pred_row = local_predictor.map(Row.of(str));
System.out.println(pred_row.getField(4));
```

---

其中，使用 `getOutputSchema` 方法获得当前预测输出结果的 Schema，内容如下。

```
root
|— review: STRING
|— vec: LEGACY(GenericType<com.alibaba.alink.common.linalg.Vector>)
|— v_2: LEGACY(GenericType<com.alibaba.alink.common.linalg.SparseVector>)
|— v_3: LEGACY(GenericType<com.alibaba.alink.common.linalg.SparseVector>)
|— pred: STRING
|— predinfo: STRING
```

“`pred`”列是预测标签列，其前面有 4 列，可以按索引值 4 从预测输出 `pred_row` 中获取预测标签值，即 `pred_row.getField(4)`。输出结果如下。

---

`neg`

---

输入的电影评论表达的是用户不喜欢该影片。

从 `PipelineModel` 获取 `LocalPredictor` 更符合我们的直观理解，但这需要启动 Flink 任务才能完成。在实际使用时，我们可以用另一种更轻量的方法——直接使用 `LocalPredictor` 的构造方法，第一个参数为模型文件地址，第二个参数为 `SchemaStr`，具体代码如下。

---

```
LocalPredictor local_predictor_2
= new LocalPredictor(DATA_DIR + PIPELINE_MODEL, "review string");

System.out.println(local_predictor_2.getOutputSchema());
pred_row = local_predictor_2.map(Row.of(str));
System.out.println(pred_row.getField(4));
```

---

运行结果如下。

```
root
|— review: STRING
|— vec: LEGACY(GenericType<com.alibaba.alink.common.linalg.Vector>)
|— v_2: LEGACY(GenericType<com.alibaba.alink.common.linalg.SparseVector>)
|— v_3: LEGACY(GenericType<com.alibaba.alink.common.linalg.SparseVector>)
|— pred: STRING
|— predinfo: STRING
```

`neg`



## 构建推荐系统

本章讨论的主题是“推荐”，即通过参考已知的数据，对每个用户所关注的商品或信息给出个性化、精准的推荐。本章会围绕一个电影推荐的具体场景展开，需要用到两个数据集——电影信息和用户对电影的评分记录，二者均来自 MovieLens 网站，下载地址为链接 24-1。

电影信息数据集的数据表名为 movielens\_movies，包含 1700 部电影的信息，内容如图 24-1 所示。

movieid	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action Crime Thriller
7	Sabrina (1995)	Comedy Romance
8	Tom and Huck (1995)	Adventure Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action Adventure Thriller
11	American President, The (1995)	Comedy Drama Romance
12	Dracula: Dead and Loving It (1995)	Comedy Horror
13	Balto (1995)	Animation Children
14	Nixon (1995)	Drama
15	Cutthroat Island (1995)	Action Adventure Romance
16	Casino (1995)	Crime Drama

图 24-1

其中共有 3 个数据列，第一列名称为 movieid，即电影 ID，在下面的数据集中，通过电影

ID 关联评论信息；第二列名称为 title，是电影名称，电影名称后括号内为电影发行的年份；第三列名称为 genres，是电影类别（喜剧、动作片、爱情喜剧等）。

用户对电影的评分记录集，其数据表名为 movielens\_ratings，包含 100000 条用户对电影的打分记录，内容如图 24-2 所示。

userid	movieid	rating
1	122	5
1	185	5
1	231	5
1	292	5
1	316	5
1	329	5
1	355	5
1	356	5
1	362	5
1	364	5

图 24-2

同前面的数据表一样，评分记录表也只有 3 列，第一列名称为 userid，用一个数字 ID 代表某个用户；第二列名称为 movieid，是电影 ID，在前面介绍数据集时已经提过，可以使用此 ID 与具体的电影信息对应；第三列名称为 rating，是用户为该影片的打分，范围为 0.5、1、1.5、……、4.5、5。

## 24.1 与推荐相关的组件介绍

Alink 提供了一系列与推荐相关的组件，从组件使用的角度来看，需要重点关注如下 3 个方面。

### 1. 算法选择

推荐领域有很多算法，常用的有基于物品/用户的协同过滤、ALS、FM 算法等。对于不同的数据场景，算法也会在计算方式上有很大的变化，譬如 ALS 算法针对数据是显式评分还是隐式反馈的情况，会采用不用的目标函数进行模型求解，得出截然不同的模型。

### 2. 推荐方式

输入信息可以有多种选择，输入结果也有多种情况。

- (1) 同时输入一个用户的信息和一个物品的信息，计算用户对此物品的评分。
- (2) 输入用户的信息，可以推荐适合此用户的相关物品，也可以计算与其相似的用户。

(3) 输入物品的信息，推荐给可能喜欢该物品的用户，也可以计算与其相似的物品。

### 3. 使用方法

在应用推荐引擎时，可能是在离线任务中进行批量推荐，也可能是在实时任务中对流式数据进行推荐，还可以通过使用 Alink Java SDK 将推荐引擎嵌入用户的应用系统中。

算法选择和推荐方式是通过“推荐模型”这个桥梁连接起来的，如图 24-3 所示。选择适合的推荐算法，基于训练数据集得到推荐模型，基于训练出来的模型可以执行多种推荐方式。

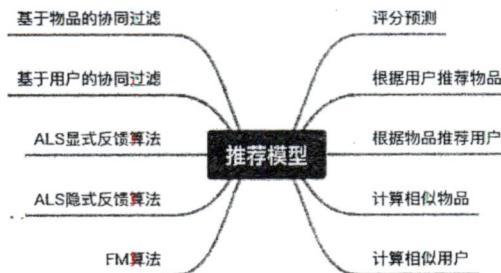


图 24-3

下面以基于物品的协同过滤（Item-based Collaborative Filtering）算法为例，看一下 Alink 相关的组件。模型训练为离线批式训练，对应组件为 ItemCfTrainBatchOp，得到 ItemCf 模型。基于此模型可以进行多种推荐，但不是每种推荐方式使用该 ItemCf 算法都能得到较好的效果。我们只提供了适合该算法的推荐方式——评分预测（ItemCfRate）、根据用户推荐物品（ItemCfItemsPerUser）、计算相似物品（ItemCfSimilarItems），没有提供“根据物品推荐用户”和“计算相似用户”组件。考虑到每种推荐需要支持多种使用方式，每种方法都提供了 3 种组件——批式推荐（RecommBatchOp）、流式推荐（RecommStreamOp）和 Pipeline 节点（后缀为 Recommender，并可由此获得 LocalPredictor，嵌入用户的应用中推荐）。如图 24-4 所示。

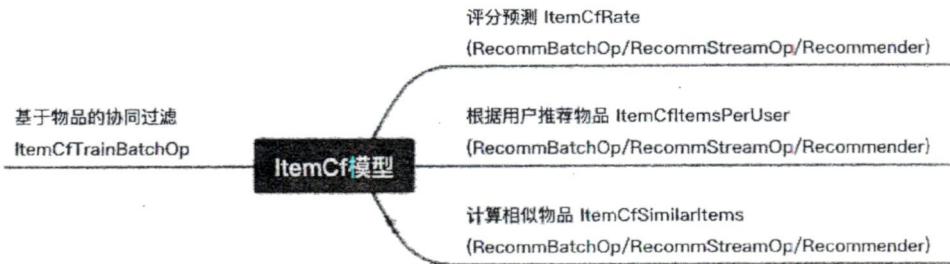


图 24-4

再看另一个有代表性的推荐算法——交替最小二乘法，其基本思路为交替固定用户特征向量和物品特征向量的值，每次求解一个最小二乘问题，直到满足求解条件。根据用户—物品矩阵中值的含义是评分值，还是行为次数、观看/收听时长，分别选用显式反馈算法（AlsTrainBatchOp）与隐式反馈算法（AlsImplicitTrainBatchOp）。两种计算方式得到 ALS 模型格式是一样的，后面可以选用 5 种推荐方式，而且每种方法都提供了 3 种组件——批式推荐（RecommBatchOp）、流式推荐（RecommStreamOp）和 Pipeline 节点（后缀为 Recommender，并可由此获得 LocalPredictor，进行嵌入用户的应用中推荐）。如图 24-5 所示。



图 24-5

Alink 在推荐方面提供的组件较多，但是规律性很强。

- 模型训练组件一律是“算法名+TrainBatchOp”。
- 推荐组件的名称为“算法名+推荐方式+使用方法”。
- 现在支持的算法名如下。
  - 基于物品的协同过滤（ItemCf）
  - 基于用户的协同过滤（UserCf）
  - ALS 显式反馈算法（Als）
  - ALS 隐式反馈算法（AlsImplicit）
  - FM 算法（Fm）
- 推荐方法如下。
  - 评分预测（Rate）

- 根据用户推荐物品 ( ItemsPerUser )
- 根据物品推荐用户 ( UsersPerItem )
- 计算相似物品 ( SimilarItems )
- 计算相似用户 ( SimilarUsers )
- 使用方法如下。
  - 批式推荐 ( RecommBatchOp )
  - 流式推荐 ( RecommStreamOp )
  - Pipeline 节点 ( 后缀为 Recommender, 并可由此获得 LocalPredictor, 进行嵌入用户的应用中推荐 )

## 24.2 常用推荐算法

本节介绍两种常用的推荐算法，分别是协同过滤和交替最小二乘法。

### 24.2.1 协同过滤

协同过滤 ( Collaborative Filtering ) 是一种利用集体智慧的方法，从众多用户的历史行为中收集目标用户的相似信息，从而发现用户潜在的兴趣偏好。

我们的日常生活中就在使用协同过滤的思想，比如你想看电影，可能会询问与自己品味相似的朋友，他们最近看了什么；你喜欢的电影如果出了续集，也可能吸引你走进电影院。这就引出了两种常用的协同过滤方法。

(1) 基于用户 ( User-Based ) 的协同过滤。即找到与你相似的用户，他们喜欢的很有可能也是你喜欢的，例如，可以通过在已经看过的影片历史记录中寻找和你的记录相似的用户，作为“相似用户”。

(2) 基于物品 ( Item-Based ) 的协同过滤。由历史记录出发，同时被更多人观看过的影片，相似度更高。也就是根据你的观影记录，将关联相似度更高的影片推荐给你。

在实际处理中，用户与物品的关系用矩阵来表示，如图 24-6 所示。用户有  $M$  个，记为  $u_1, u_2, \dots, u_M$ ，物品有  $N$  个，记为  $i_1, i_2, \dots, i_N$ 。如果用户  $u_m$  购买过物品  $i_n$ ，则可在矩阵的第  $m$  行  $n$  列记录购买次数，没有购买行为的位置可以标记为 0，这样我们就得到了用户与物品关系的矩阵。

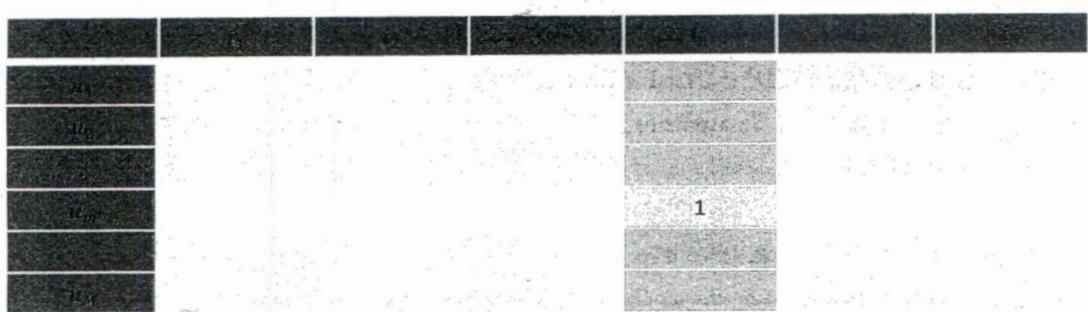


图 24-6

由用户与物品关系的矩阵，我们可以衡量用户间的相似程度、物品间的相似程度。具体的方法是：对于每个用户，将其所在矩阵的行看作其向量值，用户间的距离或相似程度可以由其对应的向量进行计算；对于每个物品，将其所在的矩阵的列看作其向量值，物品间的相似程度可以由其对应的向量进行计算。

向量间比较常用的是余弦相似度，公式如下。

$$\text{CosSim}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2} = \frac{\sum a_k b_k}{\sqrt{(\sum a_k^2)(\sum b_k^2)}}$$

如果我们忽略向量的具体数值，而在意每个位置是否为 0，则可以使用集合的 Jaccard 相似度。

$$\text{JaccardSim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

### 24.2.2 交替最小二乘法

本小节简单介绍交替最小二乘法（Alternating Least Square, ALS）。设用户的打分矩阵为  $R_{M \times N}$ ，每个用户和物品都可以用  $K$  维向量表示，用户矩阵为  $U_{M \times K}$ ，物品矩阵为  $I_{N \times K}$ 。评分  $R_{mn}$  可以表示为用户向量  $U_m$  和物品向量  $I_n$  的内积，即

$$R_{mn} \approx U_m \cdot I_n^T$$

我们希望计算用户矩阵  $U_{M \times K}$  和物品矩阵  $I_{N \times K}$ ，使其乘积逼近，采用最小平方误差定义损失函数。

$$L(\mathbf{U}, \mathbf{I}) = \sum_{R_{mn} \text{ is known}} (R_{mn} - \mathbf{U}_m \cdot \mathbf{I}_n^T)^2$$

为了防止过拟合，加入正则项系数 $\lambda$ 。

$$L(\mathbf{U}, \mathbf{I}) = \sum_{R_{mn} \text{ is known}} (R_{mn} - \mathbf{U}_m \cdot \mathbf{I}_n^T)^2 + \lambda \left( \sum_m |\mathbf{U}_m|^2 + \sum_n |\mathbf{I}_n|^2 \right)$$

固定 $\mathbf{I}_{N \times K}$ ，对 $\mathbf{U}_m$ 求导，得

$$\frac{\partial L}{\partial \mathbf{U}_m} = -2 \sum_{R_{mn} \text{ is known}} \mathbf{I}_n (R_{mn} - \mathbf{U}_m \cdot \mathbf{I}_n^T) + 2\lambda \mathbf{U}_m$$

令 $\frac{\partial L}{\partial \mathbf{U}_m} = 0$ ，可以得到相应的 $\mathbf{U}_m$ ，进而得到新的 $\mathbf{U}_{M \times K}$ 。

同理，固定 $\mathbf{U}_{M \times K}$ ，可以得到新的 $\mathbf{I}_{N \times K}$ 。如此交替执行，直到误差满足阈值条件或者达到最大迭代次数上限。

上面处理的是显式反馈的情况，对于隐式反馈， $R_{mn}$ 不再是评分值，而是表示动作次数，譬如购买、加购物车、收藏、点击的次数，或者观看/收听的时长等，取值范围是 $[0, +\infty)$ 。

定义损失函数如下：

$$L(\mathbf{U}, \mathbf{I}) = \sum_{m,n} C_{mn} (P_{mn} - \mathbf{U}_m \cdot \mathbf{I}_n^T)^2 + \lambda \left( \sum_m |\mathbf{U}_m|^2 + \sum_n |\mathbf{I}_n|^2 \right)$$

其中

$$P_{mn} = \begin{cases} 1 & \text{if } R_{mn} > 0 \\ 0 & \text{if } R_{mn} = 0 \end{cases}$$

$$C_{mn} = 1 + \alpha R_{mn}$$

求解方式与显式反馈一样，对 $\mathbf{U}_{M \times K}$ 和 $\mathbf{I}_{N \times K}$ 交替固定求解，直到误差满足阈值条件或者达到最大迭代次数上限。

## 24.3 数据探索

本章会围绕一个电影推荐的具体场景，需要用到两个数据集——电影信息和用户对电影的

评分记录，二者均来自 MovieLens 网站，下载地址为链接 24-2。

根据评论数的多少，MovieLens 提供了多个数据集。

- MovieLens 100K Dataset
- MovieLens 1M Dataset
- MovieLens 10M Dataset
- MovieLens 20M Dataset
- MovieLens 25M Dataset
- MovieLens 1B Synthetic Dataset

最小的数据集中有 10 万条评论，最大的数据集中有 2500 万条评论，利用向量生成方式得到的更大数据集中有 10 亿条评论。

除了浏览其主页上的介绍，还可以从链接 24-3 中看到可以访问的全部数据，如图 24-7 所示。

Index of /datasets/movielens				
	Name	Last modified	Size	Description
Parent Directory				
[ ]	ml-1m-README.txt	2019-12-03 11:14	5.4K	
[ ]	ml-1m.zip	2019-12-03 11:14	5.6M	
[ ]	ml-1m.zip.md5	2019-12-03 11:14	51	
[ ]	ml-10m-README.html	2019-12-03 11:14	11K	
[ ]	ml-10m.zip	2019-12-03 11:14	63M	
[ ]	ml-10m.zip.md5	2019-12-03 11:14	52	
[ ]	ml-20m-README.html	2019-12-03 11:14	12K	
[ ]	ml-20m-youtube-README...	2019-12-03 11:14	2.5K	
[ ]	ml-20m-youtube.zip	2019-12-03 11:14	639K	
[ ]	ml-20m-youtube.zip.md5	2019-12-03 11:14	56	
[ ]	ml-20m.zip	2019-12-03 11:15	189M	
[ ]	ml-20m.zip.md5	2019-12-03 11:14	52	
[ ]	ml-20mx16x32-README.txt	2019-12-03 11:14	5.0K	
[ ]	ml-20mx16x32.tar	2019-12-03 11:15	3.1G	
[ ]	ml-20mx16x32.tar.md5	2019-12-03 11:14	51	
[ ]	ml-25m-README.html	2019-12-03 11:14	12K	
[ ]	ml-25m.zip	2019-12-03 11:14	250M	
[ ]	ml-25m.zip.md5	2019-12-03 11:14	45	
[ ]	ml-100k-README.txt	2019-12-03 11:14	6.6K	
[ ]	ml-100k.zip	2019-12-03 11:14	4.7M	
[ ]	ml-100k.zip.md5	2019-12-03 11:14	53	
[ ]	ml-100k/	2019-05-08 11:18	-	
[ ]	ml-latest-README.html	2019-12-03 11:14	11K	
[ ]	ml-latest-small-READ...	2019-12-03 11:14	9.7K	
[ ]	ml-latest-small.zip	2019-12-03 11:14	955K	
[ ]	ml-latest-small.zip.md5	2019-12-03 11:14	61	
[ ]	ml-latest.zip	2019-12-03 11:15	264M	
[ ]	ml-latest.zip.md5	2019-12-03 11:15	55	
[ ]	old/	2019-05-08 11:18	-	

图 24-7

每个数据集包括多个数据文件，所有数据集都提供了压缩文件，可以下载到本地；100K 的小数据集还单独提供了文件夹 ml-100K，可以直接访问其数据文件。

在网页浏览器上访问链接 24-4，如图 24-8 所示。

Index of /datasets/movielens/ml-100k			
	Name	Last modified	Size Description
<a href="#">Parent Directory</a>			
	allbut.pl	2019-12-03 11:14	716
	mku.sh	2019-12-03 11:14	643
	u.data	2019-12-03 11:14	1.9M
	u.genre	2019-12-03 11:14	202
	u.info	2019-12-03 11:14	36
	u.item	2019-12-03 11:14	231K
	u.occupation	2019-12-03 11:14	193
	u.user	2019-12-03 11:14	22K
	u1.base	2019-12-03 11:14	1.5M
	u1.test	2019-12-03 11:14	383K
	u2.base	2019-12-03 11:14	1.5M
	u2.test	2019-12-03 11:14	386K
	u3.base	2019-12-03 11:14	1.5M
	u3.test	2019-12-03 11:14	387K
	u4.base	2019-12-03 11:14	1.5M
	u4.test	2019-12-03 11:14	388K
	u5.base	2019-12-03 11:14	1.5M
	u5.test	2019-12-03 11:14	388K
	ua.base	2019-12-03 11:14	1.7M
	ua.test	2019-12-03 11:14	182K
	ub.base	2019-12-03 11:14	1.7M
	ub.test	2019-12-03 11:14	182K

图 24-8

可以看到，该路径下有多个文件，下面我们选择 3 个常用的文件进行介绍。

### 1. 文件 u.user

表示用户属性信息，图 24-9 是数据文件的部分内容。

u.user			
1 24 M technician 85711			
2 53 F other 94043			
3 23 M writer 32067			
4 24 M technician 43537			
5 33 F other 15213			
6 42 M executive 98101			
7 57 M administrator 91344			
8 36 M administrator 05201			
9 29 M student 01002			
10 53 M lawyer 90703			

图 24-9

每个属性之间用“|”分隔，包含 5 个属性。

- (1) user\_id: 用户 ID, 对应 u.data 数据中的用户 ID 属性。
  - (2) age: 年龄。
  - (3) gender: 性别。
  - (4) occupation: 职业。
  - (5) zip code: 邮政编码。

获取用户信息数据源的代码如下。

```
static final String USER_FILE = "u.user";

static final String USER_SCHEMA_STRING
    = "user_id long, age int, gender string, occupation string, zip_code string";
...
...

static CsvSourceBatchOp getSourceUsers() {
    return new CsvSourceBatchOp()
        .setFieldDelimiter("|")
        .setFilePath(DATA_DIR + USER_FILE)
        .setSchemaStr(USER_SCHEMA_STRING);
}
```

## 2. 文件u.item

表示物品（电影）属性信息，图 24-10 是数据文件的部分内容。

图 24-10

每个属性之间用“|”分隔，包含“5+19”个属性。

- (1) item\_id: 物品(电影)ID, 对应 u.data 数据中物品(电影)ID 属性。
  - (2) title: 电影名称。
  - (3) release\_date: 电影上映日期。
  - (4) video\_release\_date: 视频发布日期。
  - (5) imdb\_url: IMDB 链接。
  - (6) 最后 19 个字段是电影流派(未知、动作、冒险、动画、儿童、喜剧、犯罪、纪录片、戏剧、幻想、黑色电影、恐怖、音乐、神秘、浪漫、科幻、惊悚、战争、西部), 1 表示电影是这种流派的, 0 表示电影不是这种流派的; 每部电影可以同时属于多种流派。

获取影片信息数据源的代码如下。

```

static final String ITEM_FILE = "u.item";
static final String ITEM_SCHEMA_STRING = "item_id long, title string, "
+ "release_date string, video_release_date string, imdb_url string, "
+ "unknown int, action int, adventure int, animation int, "
+ "children int, comedy int, crime int, documentary int, drama int, "
+ "fantasy int, film_noir int, horror int, musical int, mystery int, "
+ "romance int, sci_fi int, thriller int, war int, western int";
...
}

static CsvSourceBatchOp getSourceItems() {
    return new CsvSourceBatchOp()
        .setFieldDelimiter("|")
        .setFilePath(DATA_DIR + ITEM_FILE)
        .setSchemaStr(ITEM_SCHEMA_STRING);
}

```

### 3. 文件 u.data

表示用户对电影的评级。其中包括 943 个用户对 1682 个物品（电影）进行的 10 万次评论；每个用户至少有 20 部电影；用户和物品（电影）都是从 1 开始编号的。图 24-11 是数据文件的部分内容。

u.data			
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817
6	86	3	883603013

图 24-11

每个属性之间用制表符 “\t” 分隔，包含 4 个属性。

- (1) user\_id: 用户 ID。
- (2) item\_id: 物品（电影）ID。
- (3) rating: 评分。
- (4) ts: Unix 时间戳，是自 1/1/1970 UTC 以来的 Unix 秒数。

获取评分信息数据源的代码如下。

```

static final String RATING_FILE = "u.data";
static final String RATING_SCHEMA_STRING
    = "user_id long, item_id long, rating float, ts long";
...

static TsvSourceBatchOp getSourceRatings() {
    return new TsvSourceBatchOp()

```

---

```

.setFilePath(DATA_DIR + RATING_FILE)
.setSchemaStr(RATING_SCHEMA_STRING);
}

```

---

目录下还有已拆分的训练与测试数据集。数据集 ua.base、ua.test、ub.base 和 ub.test 是由 u.data 拆分的训练集和测试集，测试集中每个用户正好有 10 个评级记录，且 ua.test 和 ub.test 不相交。数据集 u1.base 和 u1.test 到 u5.base 和 u5.test 是 5 折交叉验证数据集。

## 24.4 评分预测

本节的目标是基于已知的评分信息建立推荐模型，针对给定的用户 ID 和影片 ID 预测评分结果。

我们使用已拆分好的训练集 ua.base 和测试集 ua.test，使用评分数据的 SchemaStr，通过 TsvSourceBatchOp 组件获取数据源。

---

```

TsvSourceBatchOp train_set = new TsvSourceBatchOp()
.setFilePath(DATA_DIR + RATING_TRAIN_FILE)
.setSchemaStr(RATING_SCHEMA_STRING);

TsvSourceBatchOp test_set = new TsvSourceBatchOp()
.setFilePath(DATA_DIR + RATING_TEST_FILE)
.setSchemaStr(RATING_SCHEMA_STRING);

```

---

训练数据连接到 AlsTrainBatchOp 组件，需要设置用户列名、影片列名和评分值；获得 ALS 推荐模型，并将其保存为 AK 格式文件，具体代码如下。

---

```

train_set
.link(
    new AlsTrainBatchOp()
        .setUserCol(USER_COL)
        .setItemCol(ITEM_COL)
        .setRateCol(RATING_COL)
        .setLambda(0.1)
        .setRank(10)
        .setNumIter(10)
)
.link(
    new AkSinkBatchOp()
        .setFilePath(DATA_DIR + ALS_MODEL_FILE)
);

```

---

下面以 ID 为 1 的用户为例，演示评分预测过程。

- 评分预测需要使用 ALS 算法相应的评分推荐组件 `AlsRateRecommender`，该组件需要用到已经训练好的 ALS 模型，所以使用 `setModelData` 方法，从 AK 文件数据源获取模型。
- 为了便于理解评分结果，需要将影片 ID 转换为名称，我们调用组件 `Lookup`，其需要一个映射表，即原始影片信息数据表（前面已经介绍过，可通过封装的 `getSourceItems` 方法获取），设置映射的 Key 和 Value 所在列名。
- 对测试数据集 `test_set` 进行过滤，保留 “`user_id=1`” 的数据，然后选择 4 个主要数据列，并对评分值进行排序，便于我们查看和对比。

具体代码如下。

```
new PipelineModel
(
    new AlsRateRecommender()
        .setUserCol(USER_COL)
        .setItemCol(ITEM_COL)
        .setRecommCol(RECOMM_COL)
        .setModelData(
            new AkSourceBatchOp()
                .setFilePath(DATA_DIR + ALS_MODEL_FILE)
        ),
    new Lookup()
        .setSelectedCols(ITEM_COL)
        .setOutputCols("item_name")
        .setModelData(getSourceItems())
        .setMapKeyCols("item_id")
        .setMapValueCols("title")
),
.transform(
    test_set.filter("user_id=1")
)
.select("user_id, rating, recomm, item_name")
.orderBy("rating, recomm", 1000)
.lazyPrint(-1);
```

运行结果如下。

user_id	rating	recomm	item_name
1 2.0000	2.5695	Dirty Dancing (1987)	
1 3.0000	3.3675	Rock, The (1996)	
1 3.0000	4.3740	Grand Day Out, A (1992)	
1 4.0000	3.4306	Desperado (1995)	
1 4.0000	3.6392	Angels and Insects (1995)	
1 4.0000	3.7185	Glengarry Glen Ross (1992)	
1 4.0000	3.7284	Hunt for Red October, The (1990)	
1 4.0000	4.3246	Three Colors: White (1994)	

```
1|5.0000|3.9633|Groundhog Day (1993)
1|5.0000|4.5148|Delicatessen (1991)
```

在对 10 部影片的预测结果中，最低分 2.5695 对应的影片为 Dirty Dancing (1987)，原始的评分为 2.0 分；最高分 4.5148 对应的影片为 Delicatessen (1991)，原始的用户评分为 5.0 分；其他预测结果与原始评分的差距似乎不大，要衡量其差距，可以使用回归评估组件。

我们对全部测试集 test\_set 的数据进行推荐评分，然后使用回归评估组件 EvalRegressionBatchOp，详细代码如下。

---

```
new AlsRateRecommender()
    .setUserCol(USER_COL)
    .setItemCol(ITEM_COL)
    .setRecommCol(RECOMM_COL)
    .setModelData(
        new AkSourceBatchOp()
            .setFilePath(DATA_DIR + ALS_MODEL_FILE)
    )
    .transform(test_set)
    .link(
        new EvalRegressionBatchOp()
            .setLabelCol(RATING_COL)
            .setPredictionCol(RECOMM_COL)
            .lazyPrintMetrics()
    );
}
```

---

运行结果如下。

```
Metrics:
MSE:0.9092 RMSE:0.9535      MAE:0.7529 MAPE:28.0715      R2:0.2751
```

这里我们更关心每个预测值与原始用户评分的差距，即平均绝对误差（MAE）0.7529。

## 24.5 根据用户推荐影片

在本节和 24.6 节中，我们将使用基于物品的协同过滤算法建立推荐模型，并根据不同问题选择使用相应的推荐预测组件。

基于物品的协同过滤模型的训练与前面 ALS 算法的类似，需要设置用户列名、影片列名和评分值，随后将模型保存为 AK 格式文件，便于在后面的推荐过程中直接导入模型进行使用。具体代码如下。

---

```
getSourceRatings()
    .link(
        new ItemCfTrainBatchOp()
```

---

```

.setUserCol(USER_COL)
.setItemCol(ITEM_COL)
.setRateCol(RATING_COL)
)
.link(
    new AkSinkBatchOp()
        .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
);
}

```

推荐任务如下代码所示。

```

MemSourceBatchOp test_data = new MemSourceBatchOp(new Long[] {1L}, "user_id");

new ItemCfItemsPerUserRecommender()
.setUserCol(USER_COL)
.setRecommCol(RECOMM_COL)
.setModelData(
    new AkSourceBatchOp()
        .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
)
.transform(test_data)
.print();
}

```

先使用 MemSourceBatchOp 组件构造一个只有一列 “user\_id”的批式数据表，包含一条数据，即所要预测的用户 ID；接着使用根据用户推荐物品的组件，设置用户列名、推荐结果列名；还需要使用 setModelData 方法，从 AK 文件数据源获取推荐模型。

输出结果如表 24-1 所示，推荐结果中包括影片 ID 和相应的推荐分值。

表 24-1 推荐结果

user_id	recomm
1	{"item_id": "[174, 56, 176, 98, 50, 195, 172, 96, 181, 204]", "score": "[0.5446295202769522, 0.5421344455706749, 0.5310766611215199, 0.5306864538037679, 0.5261969546534264, 0.5179894090961199, 0.5168003816848815, 0.5124562838810977, 0.5076640796772237, 0.5065063397097225]"}

上面演示了批式推荐任务，同样可以使用相应的流式组件执行流式任务。接下来，演示如何使用 LocalPredictor 进行推荐，使推荐过程可以嵌入 Java 应用中。

对 ItemCfItemsPerUserRecommender 组件调用其 collectLocalPredictor 方法。注意：该方法必须输入预测数据的 SchemaStr。最后，我们再将 LocalPredictor 的输出 Schema 显示出来，具体代码如下。

```

LocalPredictor recomm_predictor = new ItemCfItemsPerUserRecommender()
.setUserCol(USER_COL)
.setRecommCol(RECOMM_COL)
.setK(20)

```

---

```

.setModelData(
    new AkSourceBatchOp()
        .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
)
.collectLocalPredictor("user_id long");

System.out.println(recomm_predictor.getOutputSchema());

```

---

输出的 Schema 如下，共两列，一列为输入的用户 ID，另一列为推荐预测结果。

```

root
|--- user_id: BIGINT
|--- recomm: STRING

```

同样，定义从影片 ID 到名称映射的 kv\_predictor，并将其输出的 Schema 显示出来，代码如下。

---

```

LocalPredictor kv_predictor = new Lookup()
    .setSelectedCols(ITEM_COL)
    .setOutputCols("item_name")
    .setModelData(getSourceItems())
    .setMapKeyCols("item_id")
    .setMapValueCols("title")
    .collectLocalPredictor("item_id long");

System.out.println(kv_predictor.getOutputSchema());

```

---

输出结果如下，一列为影片 ID，另一列为影片名称。

```

root
|--- item_id: BIGINT
|--- item_name: STRING

```

下面的代码演示了整个预测过程。

- 输入的是 Row 类型数据，只有一个 Long 型字段，可以写为 Row.of(1L)。
- 对 recomm\_predictor 的 map 方法进行预测，结果为 2 个字段的 Row 类型数据，由前面输出的 Schema 信息可知，其索引号为 1 的字段为推荐结果（字符串类型）。
- 使用 JsonConverter 解析出推荐的影片 ID。
- 将每个推荐的影片 ID 映射到影片名称，并输出。

具体代码如下。

---

```

String recommResultStr = (String) recomm_predictor.map(Row.of(1L)).getField(1);

System.out.println(recommResultStr);

List <Long> recomm_ids = JsonConverter
    .fromJson(
        JsonConverter
            .<Map <String, String>>fromJson(

```

---

```

        recommResultStr,
        Map.class
    )
    .get("item_id"),
    new TypeReference <List <Long>>() {}.getType()
);

for (Long id : recomm_ids) {
    System.out.println(kv_predictor.map(Row.of(id)));
}

```

输出内容如下，前面为推荐结果，后面是各个推荐影片 ID 和名称。

```

{"item_id": "[174, 56, 176, 98, 50, 195, 172, 96, 181, 204, 183, 89, 168, 173, 79, 228, 210, 234, 423, 144]", "score": "[0.5
446295202769522, 0.5421344455706749, 0.5310766611215199, 0.5306864538037679, 0.5261969546534264, 0.51798940
90961199, 0.5168003816848815, 0.5124562838810977, 0.5076640796772237, 0.5065063397097225, 0.504903666688330
5, 0.5037182261191273, 0.503485951777993, 0.4974758183778898, 0.48814994943889534, 0.4855370682711127, 0.484
72553094018794, 0.48321375585798926, 0.48163621419180747, 0.48119185189692276]"}
174, Raiders of the Lost Ark (1981)
56, Pulp Fiction (1994)
176, Aliens (1986)
98, Silence of the Lambs, The (1991)
50, Star Wars (1977)
195, Terminator, The (1984)
172, Empire Strikes Back, The (1980)
96, Terminator 2: Judgment Day (1991)
181, Return of the Jedi (1983)
204, Back to the Future (1985)
183, Alien (1979)
89, Blade Runner (1982)
168, Monty Python and the Holy Grail (1974)
173, Princess Bride, The (1987)
79, Fugitive, The (1993)
228, Star Trek: The Wrath of Khan (1982)
210, Indiana Jones and the Last Crusade (1989)
234, Jaws (1975)
423, E.T. the Extra-Terrestrial (1982)
144, Die Hard (1988)

```

在判断上面的推荐是否准确前，我们需要先了解 1 号用户 (`user_id=1`) 对影片的喜爱。一个简单的办法是列出被 1 号用户评为高分的影片，就大概知道他喜欢什么类型影片，具体代码如下。

```

new Lookup()
.setSelectedCols(ITEM_COL)
.setOutputCols("item_name")
.setModelData(getSourceItems())
.setMapKeyCols("item_id")
.setMapValueCols("title")
.transform()

```

```

    .getSourceRatings().filter("user_id=1 AND rating>4")
)
.select("item_name")
.orderBy("item_name", 1000)
.lazyPrint(-1);

```

从原始的评分数据中，过滤用户 ID 为 1 且评分大于 4 分的记录，并将其中的影片 ID 映射为影片名称。

输出结果如表 24-2 所示。

表 24-2 1 号用户喜欢的影片列表

12 Angry Men (1957)	Manon of the Spring (Manon des sources) (1986)
Alien (1979)	Mars Attacks! (1996)
Aliens (1986)	Maya Lin: A Strong Clear Vision (1994)
Amaeus (1984)	Mighty Aphrodite (1995)
Antonia's Line (1995)	Monty Python and the Holy Grail (1974)
Back to the Future (1985)	Monty Python's Life of Brian (1979)
Big Night (1996)	Mr. Holland's Opus (1995)
Blade Runner (1982)	Mystery Science Theater 3000: The Movie (1996)
Bound (1996)	Nightmare Before Christmas, The (1993)
Brazil (1985)	Nikita (La Femme Nikita) (1990)
Breaking the Waves (1996)	Pillow Book, The (1995)
Chasing Amy (1997)	Postino, Il (1994)
Chasing Amy (1997)	Priest (1994)
Cinema Paradiso (1988)	Princess Bride, The (1987)
Clerks (1994)	Professional, The (1994)
Contact (1997)	Raiders of the Lost Ark (1981)
Crumb (1994)	Remains of the Day, The (1993)
Cyrano de Bergerac (1990)	Return of the Jedi (1983)
Dead Man Walking (1995)	Ridicule (1996)
Dead Poets Society (1989)	Searching for Bobby Fischer (1993)
Delicatessen (1991)	Shanghai Triad (Yao a yao yao dao waipo qiao) (1995)
Dolores Claiborne (1994)	Shawshank Redemption, The (1994)
Eat Drink Man Woman (1994)	Sleeper (1973)
Empire Strikes Back, The (1980)	Sling Blade (1996)
Fargo (1996)	Star Trek: The Wrath of Khan (1982)
French Twist (Gazon maudit) (1995)	Star Wars (1977)
Full Monty, The (1997)	Swingers (1996)
Gattaca (1997)	Terminator 2: Judgment Day (1991)
Godfather, The (1972)	Terminator, The (1984)
Good, The Bad and The Ugly, The (1966)	Three Colors: Blue (1993)
Graduate, The (1967)	Three Colors: Red (1994)
Groundhog Day (1993)	Toy Story (1995)
Haunted World of Edward D. Wood Jr., The (1995)	Truth About Cats & Dogs, The (1996)
Henry V (1989)	Usual Suspects, The (1995)
Hoop Dreams (1994)	Wallace & Gromit: The Best of Aardman Animation (1996)
Horseman on the Roof, The (Hussard sur le toit, Le) (1995)	Welcome to the Dollhouse (1995)
	When Harry Met Sally... (1989)

续表

Hudsucker Proxy, The (1994) Jean de Florette (1986) Jurassic Park (1993) Kids in the Hall: Brain Candy (1996) Kolya (1996) Lone Star (1996)	Wrong Trousers, The (1993) Young Frankenstein (1974)
--	---

我们对比推荐结果，发现很多是用户评价为 5 分的影片，推荐的影片的确都是用户喜欢的。但在实际使用中，用户更希望我们推荐新的影片，也就是用户没看过的影片。在 Alink 推荐组件中提供了一个方法 `setExcludeKnown(true)`，该方法的默认值为 `false`。构建一个新的 LocalPredictor，推荐用户没评价过的影片，如下代码所示。

```
...
LocalPredictor recomm_predictor_2 = new ItemCfItemsPerUserRecommender()
    .setUserCol(USER_COL)
    .setRecommCol(RECOMM_COL)
    .setK(20)
    .setExcludeKnown(true)
    .setModelData(
        new AkSourceBatchOp()
            .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
    )
    .collectLocalPredictor("user_id long");
...

```

推荐结果如下，这次排在第 1 位的是 *E.T. the Extra-Terrestrial* (1982)，在上次的推荐结果中它排在十几位，前面都是用户评价过的影片。

```
{"item_id": "[423, 318, 357, 655, 568, 385, 684, 496, 403, 566, 732, 433, 651, 527, 432, 474, 405, 550, 588, 435]", "score": "[0.48163621419180747, 0.4564074931610021, 0.4425355598672598, 0.4360613380052011, 0.42484215121153635, 0.41393159715117167, 0.41252565376658934, 0.4028040283264125, 0.3992900166944061, 0.39697125072097783, 0.39490277720105643, 0.3880766917892268, 0.3843606351882687, 0.3813688255025348, 0.3749809650446883, 0.37404643823803746, 0.37202533566821233, 0.369066737917194, 0.36636674389371593, 0.36300743588081913]"}
423, E.T. the Extra-Terrestrial (1982)
318, Schindler's List (1993)
357, One Flew Over the Cuckoo's Nest (1975)
655, Stand by Me (1986)
568, Speed (1994)
385, True Lies (1994)
684, In the Line of Fire (1993)
496, It's a Wonderful Life (1946)
403, Batman (1989)
566, Clear and Present Danger (1994)
732, Dave (1993)
```

433,Heathers (1989)  
 651,Glory (1989)  
 527,Gandhi (1982)  
 432,Fantasia (1940)  
 474,Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1963)  
 405,Mission: Impossible (1996)  
 550,Die Hard: With a Vengeance (1995)  
 588,Beauty and the Beast (1991)  
 435,Butch Cassidy and the Sundance Kid (1969)

## 24.6 计算相似影片

本节以星球大战( Star Wars (1977), item\_id=50 )为例，通过推荐模型计算与其相似的影片。  
 推荐任务如下代码所示。

```
MemSourceBatchOp test_data = new MemSourceBatchOp(new Long[] {50L}, ITEM_COL);

new ItemCfSimilarItemsRecommender()
.setItemCol(ITEM_COL)
.setRecommCol(RECOMM_COL)
.setModelData(
    new AkSourceBatchOp()
        .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
)
.transform(test_data)
.print();
```

先使用 MemSourceBatchOp 组件，构造一个只有一列 “item\_id” 的批式数据表，包含一条数据，即所要预测的影片 ID；使用根据计算物品相似度的组件，设置用户列名、推荐结果列名，设置输出 Top10 的影片；还需要使用 setModelData 方法，从 AK 文件数据源获取模型。

输出结果如表 24-3 所示，推荐结果中包括影片 ID 和相应的推荐分值。

表 24-3 推荐结果

item_id	recomm
50	{"item_id": "[181, 174, 172, 1, 127, 121, 210, 100, 98, 222]", "similarities": "[0.8844757466 059665, 0.7648851255036908, 0.7498192415368896, 0.7345720560109783, 0.697331842805255 6, 0.6928373036216864, 0.6893433161183706, 0.6865325410591914, 0.6764284324255825, 0.6 739748837432759]"}

下面我们换成使用 LocalPredictor 的方式计算相似的影片，并使用 Lookup 组件将影片 ID

转化为影片名称，具体的电影名能给我们更直观的感觉，以便判断计算出来的影片是否很相似。具体代码如下，其使用方式与24.5节类似，这里就不详细介绍。

```

LocalPredictor recomm_predictor = new ItemCfSimilarItemsRecommender()
    .setItemCol(ITEM_COL)
    .setRecommCol(RECOMM_COL)
    .setK(10)
    .setModelData(
        new AkSourceBatchOp()
            .setFilePath(DATA_DIR + ITEMCF_MODEL_FILE)
    )
    .collectLocalPredictor("item_id long");

LocalPredictor kv_predictor = new Lookup()
    .setSelectedCols(ITEM_COL)
    .setOutputCols("item_name")
    .setModelData(getSourceItems())
    .setMapKeyCols("item_id")
    .setMapValueCols("title")
    .collectLocalPredictor("item_id long");

String recommResultStr = (String) recomm_predictor.map(Row.of(50L)).getField(1);

List <Long> recomm_ids = JsonConverter
    .fromJson(
        JsonConverter
            .<Map <String, String>>fromJson(
                recommResultStr,
                Map.class
            )
            .get("item_id"),
        new TypeReference <List <Long>>() {}.getType()
    );
}

for (Long id : recomm_ids) {
    System.out.println(kv_predictor.map(Row.of(id)));
}

```

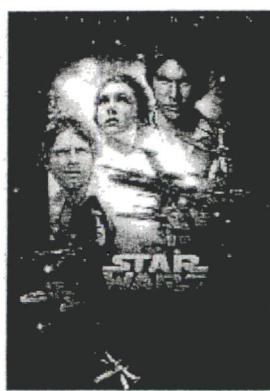
计算结果如下。

```

181, Return of the Jedi (1983)
174, Raiders of the Lost Ark (1981)
172, Empire Strikes Back, The (1980)
1, Toy Story (1995)
127, Godfather, The (1972)
121, Independence Day (ID4) (1996)
210, Indiana Jones and the Last Crusade (1989)
100, Fargo (1996)
98, Silence of the Lambs, The (1991)
222, Star Trek: First Contact (1996)

```

从图 24-12 中可以看到，推荐结果中排在第 1 位的是“星战系列”的《绝地反击》（Return of the Jedi），排在第 3 位的是“星战系列”的《帝国反击战》（Empire Strikes Back, The）。



Star Wars  
(1977)



Empire Strikes Back, The  
(1980)



Return of the Jedi  
(1983)

图 24-12

排在第 2 位的是《夺宝奇兵》（Raiders of the Lost Ark），其编剧乔治·卢卡斯是《星球大战》（Star Wars）的编剧和导演，主演哈里森·福特是《星球大战》的主演。

A black and white movie poster for Indiana Jones and the Raiders of the Lost Ark (1981). It features Harrison Ford as Indiana Jones, wearing his signature fedora and trench coat, standing in a desert landscape with a temple in the background. The title "RAIDERS OF THE LOST ARK" is at the top.	<p><b>夺宝奇兵 (Raiders of the Lost Ark)</b></p> <p>导演：史蒂文·斯皮尔伯格 编剧：劳伦斯·卡斯丹/乔治·卢卡斯/菲利普·考夫曼 主演：哈里森·福特/凯伦·阿兰/保罗·弗里曼/约翰·瑞斯·戴维斯</p>
---	---

## 24.7 根据影片推荐用户

在本节及 24.8 节中，我们将使用基于用户的协同过滤算法，其模型训练与前面基于物品的

协同过滤算法类似。所需的训练数据格式是一样的，使用模型训练组件 UserCfTrainBatchOp，设置用户列名、影片列名和评分值，随后将模型保存为 AK 格式文件，便于在后面的推荐过程中直接导入模型进行使用。具体代码如下。

```
getSourceRatings()
    .link(
        new UserCfTrainBatchOp()
            .setUserCol(USER_COL)
            .setItemCol(ITEM_COL)
            .setRateCol(RATING_COL)
    )
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + USERCF_MODEL_FILE)
    );
};
```

推荐任务如下代码所示。

```
MemSourceBatchOp test_data = new MemSourceBatchOp(new Long[] {50L}, ITEM_COL);

new UserCfUsersPerItemRecommender()
    .setItemCol(ITEM_COL)
    .setRecommCol(RECOMM_COL)
    .setModelData(
        new AkSourceBatchOp()
            .setFilePath(DATA_DIR + USERCF_MODEL_FILE)
    )
    .transform(test_data)
    .print();
```

先使用 MemSourceBatchOp 组件，构造一个只有一列“user\_id”的批式数据表，包含一条数据，即所要预测的影片 ID；使用根据物品推荐用户的组件 UserCfUsersPerItemRecommender，设置用户列名、推荐结果列名，还需要使用 setModelData 方法从 AK 文件数据源获取模型。

输出结果如表 24-4 所示，推荐结果中包括用户 ID 和相应的推荐分值。

表 24-4 推荐结果

item_id	recomm
50	{"item_id": "[276, 429, 222, 864, 194, 650, 896, 303, 749, 301]", "score": "[0.2911498078185388, 0.28980307729545673, 0.2896787402948555, 0.2867326943225667, 0.28158684138333434, 0.2815710577453545, 0.2803413050267188, 0.27966251684722754, 0.27960936969364464, 0.27907226770541305]"}

我们通过一个简单的实验来检验预测的情况，看看评分数据集中是否已有推荐用户对目标影片（ID=15）的评分。使用 filter 方法，用 Flink SQL 语法描述过滤条件即可，具体代码如下。

```

getSourceRatings()
    .filter("user_id IN (276,429,222,864,194,650,896,303,749,301) AND item_id=50")
    .print();

```

运行结果如表 24-5 所示，10 个推荐的用户都对该电影给出过评分，8 人给的是 5 分，给 4 分和 3 分的各 1 人。

表 24-5 10 个推荐的原始评分

user_id	item_id	rating	ts
429	50	5.0000	882384553
301	50	5.0000	882074647
864	50	5.0000	877214085
650	50	5.0000	891372232
276	50	5.0000	880913800
194	50	3.0000	879521396
303	50	5.0000	879466866
749	50	5.0000	878846978
222	50	4.0000	877563194
896	50	5.0000	887159211

如果我们希望将电影推荐给那些没看过该电影的新用户，那么需要在原有的推荐预测流程中配置一个新的参数 `ExcludeKnown=true`，该参数的默认值为 `false`。相关代码如下。

```

new UserCfUsersPerItemRecommender()
    .setItemCol(ITEM_COL)
    .setRecommCol(RECOMM_COL)
    .setExcludeKnown(true)
    .setModelData(
        new AkSourceBatchOp()
            .setFilePath(DATA_DIR + USERCF_MODEL_FILE)
    )
    .transform(test_data)
    .print();

```

运行结果如表 24-6 所示。

表 24-6 推荐结果

item_id	recomm
50	{"item_id": "[16, 788, 932, 442, 207, 90, 627, 543, 532, 911]", "score": "[0.2350168159886009, 4, 0.2342959922966516, 0.2285404610116206, 0.22245913526644714, 0.22095385722564706, 0.2198506485812082, 0.21495692781445608, 0.2145426338815538, 0.20757353674629853, 0.1931964040290373]"}]

## 24.8 计算相似用户

本节以 user\_id=1 的用户为例，通过推荐模型计算与其相似的用户。

推荐任务如下代码所示。

```
MemSourceBatchOp test_data = new MemSourceBatchOp(new Long[] {1L}, USER_COL);

new UserCfSimilarUsersRecommender()
.setUserCol(USER_COL)
.setRecommCol(RECOMM_COL)
.setModelData(
    new AkSourceBatchOp()
    .setFilePath(DATA_DIR + USERCF_MODEL_FILE)
)
.transform(test_data)
.print();
```

先使用 MemSourceBatchOp 组件，构造一个只有一列“user\_id”的批式数据表，包含一条数据，即所要预测的用户 ID；使用计算相似用户的推荐组件，设置用户列名、推荐结果列名；还需要使用 setModelData 方法，从 AK 文件数据源获取基于用户的协同过滤模型。

输出结果如表 24-7 所示，推荐结果中包括推荐的相似用户 ID 和相应的推荐分值。

表 24-7 推荐结果

user_id	recomm
1	{"item_id": "[916, 864, 268, 92, 435, 457, 738, 429, 303, 276]", "similarities": "[0.56906573 15279888, 0.5475482621940828, 0.5420770475201064, 0.5405335611842348, 0.5386645318853 762, 0.5384759750393853, 0.5270310735011106, 0.525949926718084, 0.525717734084985, 0.5 245225229720628]"}

进一步，我们可以在用户信息数据表中过滤这些推荐的用户，查看他们的信息，具体代码如下。

```
getSourceUsers()
.filter("user_id IN (1, 916, 864, 268, 92, 435, 457, 738, 429, 303, 276)")
.print();
```

运行结果如表 24-8 所示，可以看到，1 号用户的性别为男性（Male），年龄 24 岁，职业为 technician，被推荐的相似用户绝大部分也为男性，年龄在 19 ~ 35 岁，职业方面主要是学生和工程技术人士。

表 24-8 过滤的用户信息

user_id	age	gender	occupation	zip_code
864	27	M	programmer	63021
916	27	M	engineer	N2L5N
429	27	M	student	29205
435	24	M	engineer	60007
457	33	F	salesman	30011
738	35	M	technician	95403
268	24	M	engineer	19422
276	21	M	student	95064
303	19	M	student	14853
1	24	M	technician	85711
92	32	M	entertainment	80525

业界力荐

Alink将众多的机器学习算法以标准组件的方式结合在一起，力图使对机器学习感兴趣的业务工程师可以迅速将这些算法和生产数据集结合在一起，验证效果，进行参数的调优，并最终将这些算法嵌入业务流程当中。

贾扬清

阿里巴巴集团副总裁

阿里巴巴开源技术委员会负责人

Apache Flink作为新一代大数据计算引擎，凭借其流批一体以及高效的迭代计算能力，不仅能够利用SQL对海量数据进行实时分析，还能够在机器学习领域进行包括特征工程、模型训练、验证和推理在内的全链路计算。Alink在Apache Flink强大计算能力的基础上，封装了一套实时离线一体化的机器学习算法库，并在阿里巴巴集团内部进行了大量验证。通过学习Alink技术，算法人员可以快速搭建起业界强大的机器学习算法平台。

王峰

Apache Flink中文社区发起人

阿里巴巴开源大数据平台负责人

Apache Flink被大家熟知的往往是其流计算的能力，其实在设计之初Flink还有一个重要的特性——迭代计算。Alink正是利用Apache Flink迭代计算能力构建的高效机器学习算法库，把算法和大数据处理无缝结合起来，大大提高了智能算法的研发效率，帮助企业实现了更多业务智能化场景的落地。

林伟

阿里巴巴研究员

阿里巴巴机器学习平台PAI负责人



读者服务

微信扫码回复：42058

- 获取书中链接地址
- 加入“人工智能与大数据”读者交流群，与更多读者互动
- 获取【百场业界大咖直播合集】（持续更新），仅需1元



责任编辑：刘皎

方迎投稿

邮箱：Ljiao@phei.com.cn

电话：010-88254395

封面设计：吴海燕

上架建议：机器学习

ISBN 978-7-121-42058-0



9 787121 420580 >

定价：149.00元