

图 7-1

我们遇到的很多数据都近似服从正态分布，那些 Z Score 的绝对值很大时，其是异常点的可能性也很大。

示例代码如下。这里使用了 StandardScaler 组件，通过 fit 方式，计算所需的统计量，得到模型。然后，使用 transform 方法进行预测。对执行标准化操作前后的数据分别打印统计结果，以便对照、理解数据的变化。

---

```
source.lazyPrintStatistics("< Origin data >");

StandardScaler scaler = new StandardScaler().setSelectedCols(FEATURE_COL_NAMES);

scaler
    .fit(source)
    .transform(source)
    .lazyPrintStatistics("< after Standard Scale >");

BatchOperator.execute();
```

---

结果如下。在执行标准化操作后，四个数值列的均值都为 0，方差都是 1。

< Origin data >							
colName	count	missing	sum	mean	variance	min	max

sepal_length  150  0  876.5  5.8433  0.6857  4.3  7.9
sepal_width  150  0  458.1  3.054  0.188  2  4.4
petal_length  150  0  563.8  3.7587  3.1132  1  6.9
petal_width  150  0  179.8  1.1987  0.5824  0.1  2.5
category  150  0  NaN  NaN  NaN  NaN  NaN

< after Standard Scale >							
colName	count	missing	sum	mean	variance	min	max
sepal_length	150	0	-0	-0	1	-1.8638	2.4837
sepal_width	150	0	0	0	1	-2.4308	3.1043
petal_length	150	0	0	0	1	-1.5635	1.7804
petal_width	150	0	0	0	1	-1.4396	1.7052
category	150	0	NaN	NaN	NaN	NaN	NaN

### 7.3.2 MinMaxScale

将最小值映射为 0，最大值映射为 1，然后按比例将所有数据映射到区间[0,1]。即对数据  $X_1, X_2, \dots, X_n$ ，进行如下变换：

$$X'_i = \frac{X_i - X_{\min}}{X_{\max} - X_{\min}}$$

由最大值、最小值的定义，可知

$$0 \leq X_i - X_{\min} \leq X_{\max} - X_{\min}$$

所以， $0 \leq X'_i \leq 1$ ，即得到的数据  $X'_1, X'_2, \dots, X'_n$  都在 0 与 1 之间。这就是“归一”化处理的结果。再将此结果数据存入结果表中。

示例代码如下。这里使用了 `MinMaxScaler`，通过 `fit` 方式，计算所需的统计量，得到模型。然后，使用 `transform` 方法进行预测。对执行该操作前后的数据分别打印统计结果，以便对照、理解数据的变化。

---

```
source.lazyPrintStatistics("< Origin data >");

MinMaxScaler scaler = new MinMaxScaler().setSelectedCols(FEATURE_COL_NAMES);

scaler
    .fit(source)
    .transform(source)
    .lazyPrintStatistics("< after MinMax Scale >");

BatchOperator.execute();
```

---

结果如下。在执行 `MinMaxScale` 操作后，四个数值列的最小值都为 0，最大值都是 1。

< Origin data >						
	colName	count	missing	sum	mean	variance
					min	max
	sepal_length	150	0	876.5	5.8433	0.6857
	sepal_width	150	0	458.1	3.054	0.188
	petal_length	150	0	563.8	3.7587	3.1132
	petal_width	150	0	179.8	1.1987	0.5824
	category	150	0	NaN	NaN	NaN NaN NaN

< after MinMax Scale >						
	colName	count	missing	sum	mean	variance
					min	max
	sepal_length	150	0	64.3056	0.4287	0.0529
	sepal_width	150	0	65.875	0.4392	0.0326
	petal_length	150	0	70.1356	0.4676	0.0894
	petal_width	150	0	68.6667	0.4578	0.1011
	category	150	0	NaN	NaN	NaN NaN NaN

### 7.3.3 MaxAbsScale

以 0 为中心，将数据缩放至区间 [-1,1]，在整个变化过程中，数值 0 不会改变。

定义所有  $X_i$  的绝对值的最大值为  $X_{\text{maxabs}}$ ，即

$$X_{\text{maxabs}} = \max(|X_1|, |X_2|, \dots, |X_n|)$$

如果已经计算出了  $X_{\text{min}}$  和  $X_{\text{max}}$ ，也可将  $X_{\text{maxabs}}$  的计算简化为

$$X_{\text{maxabs}} = \max(|X_{\text{min}}|, |X_{\text{max}}|)$$

绝对值最大值的伸缩变换为

$$X'_i = \frac{X_i}{X_{\text{maxabs}}}$$

示例代码如下。这里使用了 MaxAbsScaler，通过 fit 方式，计算所需的统计量，得到模型。然后，使用 transform 方法进行预测，并对执行该操作前后的数据分别打印统计结果。

---

```
source.lazyPrintStatistics("< Origin data >");

MaxAbsScaler scaler = new MaxAbsScaler().setSelectedCols(FEATURE_COL_NAMES);

scaler
    .fit(source)
```

---

---

```
.transform(source)
.lazyPrintStatistics("< after MaxAbs Scale >");

BatchOperator.execute();
```

---

结果如下。在执行 MaxAbsScaler 操作后，四个数值列的最大值都变换为 1，各列的最小值也相应地按比例变化。

< Origin data >						
	colName	count	missing	sum	mean	variance
sepal_length	150	0	876.5	5.8433	0.6857	4.3
sepal_width	150	0	458.1	3.054	0.188	2
petal_length	150	0	563.8	3.7587	3.1132	1
petal_width	150	0	179.8	1.1987	0.5824	0.1
category	150	0	NaN	NaN	NaN	NaN

< after MaxAbs Scale >						
	colName	count	missing	sum	mean	variance
sepal_length	150	0	110.9494	0.7397	0.011	0.5443
sepal_width	150	0	104.1136	0.6941	0.0097	0.4545
petal_length	150	0	81.7101	0.5447	0.0654	0.1449
petal_width	150	0	71.92	0.4795	0.0932	0.04
category	150	0	NaN	NaN	NaN	NaN

## 7.4 向量的尺度变换

本节将介绍四种关于向量的尺度变换：StandardScale、MinMaxScale、MaxAbsScale 和 Normalize。前三种变换（StandardScale、MinMaxScale、MaxAbsScale）在确定变换尺度的时候，都考虑到了整个向量集合，并针对每个向量分量计算整体的变换尺度。但 Normalize（规范化）操作是对于一个向量来说的，其无须考虑其他向量。

### 7.4.1 StandardScale、MinMaxScale、MaxAbsScale

StandardScale、MinMaxScale、MaxAbsScale 这三个变换均分别对向量中的每个分量进行变换。单个分量的变换可直接参考 7.3 节的相应内容，这里不再赘述。

我们构造一个示例，对原始数据分别做这三种变换。从变换前后的统计量变化上，可以看出其变换效果。仍然使用前面所用的 iris 数据集，但需要将多列的数据转化为一个向量。使用

向量组装组件 VectorAssemblerBatchOp，选择将所有特征列组装成向量，并保留标签列。使用 VectorSummarizerBatchOp 计算向量中各个分量的统计指标。先对原始数据进行统计，随后分别对向量列进行 StandardScale、MinMaxScale、MaxAbsScale 操作，并分别统计操作结果。

```
new CsvSourceBatchOp()
    .setFilePath(DATA_DIR + ORIGIN_FILE)
    .setSchemaStr(SCHEMA_STRING)
    .link(
        new VectorAssemblerBatchOp()
            .setSelectedCols(FEATURE_COL_NAMES)
            .setOutputCol(VECTOR_COL_NAME)
            .setReservedCols(LABEL_COL_NAME)
    );
source.link(
    new VectorSummarizerBatchOp()
        .setSelectedCol(VECTOR_COL_NAME)
        .lazyPrintVectorSummary("< Origin data >")
);
new VectorStandardScaler()
    .setSelectedCol(VECTOR_COL_NAME)
    .fit(source)
    .transform(source)
    .link(
        new VectorSummarizerBatchOp()
            .setSelectedCol(VECTOR_COL_NAME)
            .lazyPrintVectorSummary("< after Vector Standard Scale >")
    );
new VectorMinMaxScaler()
    .setSelectedCol(VECTOR_COL_NAME)
    .fit(source)
    .transform(source)
    .link(
        new VectorSummarizerBatchOp()
            .setSelectedCol(VECTOR_COL_NAME)
            .lazyPrintVectorSummary("< after Vector MinMax Scale >")
    );
new VectorMaxAbsScaler()
    .setSelectedCol(VECTOR_COL_NAME)
    .fit(source)
    .transform(source)
    .link(
        new VectorSummarizerBatchOp()
            .setSelectedCol(VECTOR_COL_NAME)
            .lazyPrintVectorSummary("< after Vector MaxAbs Scale >")
    );
BatchOperator.execute();
```

运行结果如下。执行 StandardScale 操作后，向量的四个分量的均值都为 0，方差都是 1；执行 MinMaxScale 操作后，四个分量的最小值都为 0，最大值都是 1；执行 MaxAbsScaler 操作后，四个数值列的最大值都变换为 1，各列的最小值也相应地按比例变化。

< Origin data >

Summary

id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2
0	150	876.5000	5.8433	0.6857	0.8281	4.3000	7.9000	876.5000	72.2762
1	150	458.1000	3.0540	0.1880	0.4336	2.0000	4.4000	458.1000	37.7763
2	150	563.8000	3.7587	3.1132	1.7644	1.0000	6.9000	563.8000	50.8232
3	150	179.8000	1.1987	0.5824	0.7632	0.1000	2.5000	179.8000	17.3868

< after Vector Standard Scale >

Summary

id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2
0	150	-0.0000	-0.0000	1.0000	1.0000	-1.8638	2.4837	124.5472	12.2066
1	150	-0.0000	-0.0000	1.0000	1.0000	-2.4308	3.1043	115.2321	12.2066
2	150	0.0000	0.0000	1.0000	1.0000	-1.5635	1.7804	132.7847	12.2066
3	150	-0.0000	-0.0000	1.0000	1.0000	-1.4396	1.7052	129.5140	12.2066

< after Vector MinMax Scale >

Summary

id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2
0	150	64.3056	0.4287	0.0529	0.2300	0.0000	1.0000	64.3056	5.9541
1	150	65.8750	0.4392	0.0326	0.1807	0.0000	1.0000	65.8750	5.8132
2	150	70.1356	0.4676	0.0894	0.2991	0.0000	1.0000	70.1356	6.7911
3	150	68.6667	0.4578	0.1011	0.3180	0.0000	1.0000	68.6667	6.8191

< after Vector MaxAbs Scale >

Summary

id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2
0	150	110.9494	0.7397	0.0110	0.1048	0.5443	1.0000	110.9494	9.1489
1	150	104.1136	0.6941	0.0097	0.0985	0.4545	1.0000	104.1136	8.5855
2	150	81.7101	0.5447	0.0654	0.2557	0.1449	1.0000	81.7101	7.3657
3	150	71.9200	0.4795	0.0932	0.3053	0.0400	1.0000	71.9200	6.9547

## 7.4.2 正则化

对于向量  $x = (x_1, x_2, \dots, x_m)^T$ ,  $p$ -范数的定义如下：

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_m|^p)^{\frac{1}{p}}$$

当  $p$  取 1、2、 $\infty$  时，分别对应以下三种最简单的情形：

- 1-范数： $\|x\|_1 = |x_1| + |x_2| + \cdots + |x_m|$
- 2-范数： $\|x\|_2 = (x_1^2 + x_2^2 + \cdots + x_m^2)^{\frac{1}{2}}$
- $\infty$ -范数： $\|x\|_\infty = \max(|x_1|, |x_2|, \dots, |x_m|)$

正则化指的是根据用户指定的  $p$  值，让每个变换后的向量的  $p$ -范数都为 1。显然，经过正则化操作，原先向量的 0 值分量经过变换后还是 0，并没有破坏向量的稀疏性。

为了方便检验结果，我们选择了 1-范数，然后选择 5 条数据进行打印。示例代码如下：

---

```
source
    .link(
        new VectorNormalizeBatchOp()
            .setSelectedCol(VECTOR_COL_NAME)
            .setP(1.0)
    )
    .firstN(5)
    .print();
```

---

运行结果如下。在此可以看出四个分量的和为 1，符合我们选择 1-范数进行正则化的预期。

category	vec
Iris-setosa	0.5 0.3431372549019608 0.13725490196078433 0.019607843137254905
Iris-setosa	0.5157894736842106 0.3157894736842105 0.14736842105263157 0.021052631578947368
Iris-setosa	0.5 0.3404255319148936 0.13829787234042554 0.02127659574468085
Iris-setosa	0.4893617021276596 0.3297872340425532 0.15957446808510642 0.021276595744680854
Iris-setosa	0.4901960784313726 0.35294117647058826 0.13725490196078433 0.019607843137254905

## 7.5 缺失值填充

缺失值填充指的是将空值或者一个指定的值替换为最大值、最小值、均值或者一个自定义的值。缺失值填充组件的参数如表 7-2 所示。可以使用如下填充方法填充值：

- 通过给定一个缺失值的配置列表，实现将输入表的缺失值用指定的值来填充的目的。
- 将数值型的空值替换为最大值、最小值、均值或者一个自定义的值。
- 将字符型的空值、空字符串或其他指定值替换为一个自定义的值。
- 缺失值若选择空字符串，则填充的目标列应是 string 类型的值。

标准化操作及归一化操作需要所处理数据列的整体统计信息，标准化操作需要均值和方差；归一化操作需要最大值和最小值；在进行缺失值填充时，如果需要使用数据的最大值、最小值或均值，也需要事先计算出统计量。

表 7-2 缺失值填充组件的参数

名称	描述
strategy	【可选】缺失值填充的规则 缺失值填充的规则，支持使用 mean、max、min 或者 value 等值进行填充。选择 value 时，需要读取 fillValue 的值。其默认值为 mean
fillValue	【可选】填充缺失值 自定义的填充值。当参数 strategy 为 value 时，读取 fillValue 的值
selectedCols	【必填】选择的列名 计算列对应的列名列表
outputCols	【可选】输出结果列的列名数组 输出结果列的列名数组，默认值为 null，即在原数据列上直接填充

下面通过示例进行说明，代码如下。首先构造一个包含 5 条数据的数据集，最后一条数据的各项都是 null。第一列是字符串类型的，选择指定填充值为“e”；第二列和第三列都选择使用均值进行填充。

```
Row[] rows = new Row[] {
    Row.of("a", 10.0, 100),
    Row.of("b", -2.5, 9),
    Row.of("c", 100.2, 1),
    Row.of("d", -99.9, 100),
    Row.of(null, null, null)
};

MemSourceBatchOp source
= new MemSourceBatchOp(rows, new String[] {"col1", "col2", "col3"});

source.lazyPrint(-1, "< origin data >");

Pipeline pipeline = new Pipeline()
    .add(
        new Imputer()
            .setSelectedCols("col1")
            .setStrategy(Strategy.VALUE)
            .setFillValue("e")
    )
    .add(
        new Imputer()
            .setSelectedCols("col2", "col3")
    )
}
```

```

.setStrategy(Strategy.MEAN)
);
pipeline.fit(source)
.transform(source)
.print();

```

在此可以看到运行情况，原始数据如表 7-3 所示。第 5 行的数据都为 null，需要被填充。

表 7-3 带缺失值的原始数据

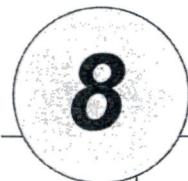
col1	col2	col3
a	10.0000	100
b	-2.5000	9
c	100.2000	1
d	-99.9000	100
null	null	null

填充结果如表 7-4 所示。

表 7-4 缺失值填充后的结果

col1	col2	col3
a	10.0000	100
b	-2.5000	9
c	100.2000	1
d	-99.9000	100
e	1.9500	52

第一列正如组件所设定，填充了“e”；第二列为双精度浮点类型的数值，填充值为均值 1.9500；第三列为整型的数值，其均值为  $210/4$ ，按整数计算规则得到 52，故填充值为 52。



## 线性二分类模型

本章将介绍纸钞的真假判断示例。根据给定钞票照片的 4 个度量值，预测其是真钞还是假钞。从机器学习的角度来看，这是典型的分类问题，而且分类目标为两个，即二分类问题。这里使用的数据特征已经被很好地数字化，可以直接套用常用的逻辑回归、线性 SVM 模型进行训练、预测。

本章的开头会介绍一些理论知识。从线性模型的角度，可以将逻辑回归模型和线性 SVM 模型（线性支持向量机模型）统一起来，这样更容易理解它们之间的关联与差异。有了二分类模型后，还需要知道如何评估该模型的效果，8.2 节会具体介绍相关的指标和曲线。

从 8.3 节开始，通过纸钞的真假判断示例，展示数据探索、训练线性模型、使用模型进行预测的完整过程。通过对比评估指标，我们可以看到调整模型参数、特征处理对模型效果的影响。最后介绍线性模型的一个扩展：因子分解机（FM）模型。

### 8.1 线性模型的基础知识

本节首先从如何评估预测值与真实值入手，介绍线性模型，再说明其常用的两个具体形式：逻辑回归模型与线性 SVM 模型。

#### 8.1.1 损失函数

损失函数（Loss Function）或代价函数（Cost Function）用来度量预测值 $f(x)$ 与真实值 $y$ 的不一致程度，将预测误差量化为非负实数，记作 $L(y, f(x))$ 。

分类损失函数的情况比较复杂，为了方便描述问题，我们只考虑分类标签值为 $\pm 1$ 的情况， $y = \pm 1$ 。常用的分类损失函数如下：

- 0-1 损失函数（0-1 Loss Function）

$$L(y, f(x)) = \begin{cases} 1, & y \cdot f(x) < 0 \\ 0, & y \cdot f(x) \geq 0 \end{cases}$$

- 对数损失函数（Logarithmic Loss Function）

$$L(y, f(x)) = \log_2(1 + \exp(-y \cdot f(x)))$$

注意：这里是以 2 为底的对数，当 $y \cdot f(x) = 0$ 时，可以保证 $L(y, f(x))$ 的值为 1，与大部分损失函数在 0 点的值相同。

- 指数损失函数（Exponential Loss Function）

$$L(y, f(x)) = \exp\{-y \cdot f(x)\}$$

- 合页损失函数（Hinge Loss Function）

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

当 $f(x)$ 的符号与 $y$ 相同，且 $|f(x)| \geq 1$ 时，损失函数的值为 0。当 $y = +1$ ，且 $f(x) \geq 1$ 时，对应的损失函数 $L(y, f(x)) = 0$ ；当 $y = -1$ ，且 $f(x) \leq -1$ 时，对应的损失函数 $L(y, f(x)) = 0$ 。

- 感知损失函数（Perceptron Loss Function）

$$L(y, f(x)) = \max(0, -y \cdot f(x))$$

该函数可以被看作合页损失函数（Hinge Loss Function）的变种。如果 $f(x)$ 的符号与 $y$ 相同，损失函数的值为 0。

我们可以看到，在前面定义的分类损失函数中， $y$ 和 $f(x)$ 往往是作为一个整体 $y \cdot f(x)$ 出现的。 $y \cdot f(x)$ 为正，意味着预测正确； $y \cdot f(x)$ 为负，则意味着预测错误。负值越大，意味着模型在当前数据中的预测结果越差。

我们将 $y \cdot f(x)$ 整体看作一个自变量，则可以得到各损失函数的函数曲线，如图 8-1 所示。

感兴趣的读者可以自己尝试作图，访问链接 8-1，在函数输入框中填如下表达式：

---

```
0.5*(1+sign(-x)), log(1+exp(-x))/log(2), exp(-x), max(0,1-x), max(0,-x)
```

---

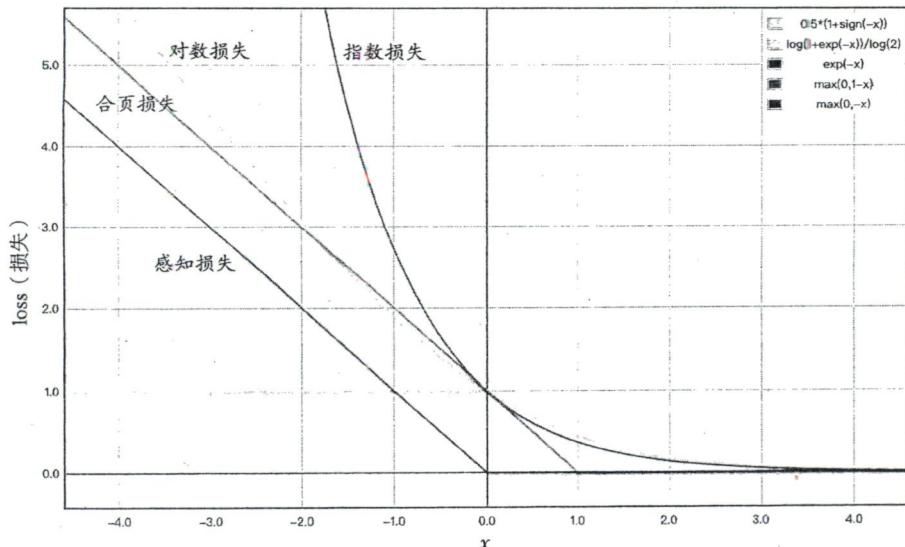


图 8-1

从图 8-1 中看到，在 0 点右侧，各曲线迅速趋近 X 轴，即损失值趋近 0。而在 0 点左侧，各损失函数的增长情况明显不同；损失值增长速度的大小关系如下：

指数损失 > 对数损失 > 合页损失 > 感知损失 > 0-1 损失

我们常用的逻辑回归算法使用的是对数损失函数，线性支持向量机算法使用的是合页损失函数，二者都是增长速度位于中位的损失函数。

### 8.1.2 经验风险与结构风险

下面先介绍一些理论上的定义。风险函数 (Risk Function)，又称期望损失 (Expected Loss)。顾名思义，“期望损失”就是损失函数计算出来的损失值的数学期望，即

$$\int_{x,y} L(y, f(x)) P(x, y) dx dy$$

其中， $P(x, y)$  为属性变量  $x$  与标签变量  $y$  的联合概率。但实际上联合概率  $P(x, y)$  是未知的（若已知，则可以直接求出条件概率  $P(y|x)$ ，也无须再训练模型  $f(x)$  了）。这就需要一个更实用、

更可操作的指标。

设样本集有  $N$  个样本，即  $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)\}$ ，则模型  $f(x)$  关于此样本集的平均损失被称为经验风险（Empirical Risk）或经验损失（Empirical Loss），即

$$L(f) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i))$$

模型的经验风险是一个量化的指标，可以将经验风险最小的模型看作最优的模型，即模型寻优的策略是经验风险最小的。

引入正则（Regularizer）项或者罚（Penalty）项的概念，记作  $\Omega(f)$ ，用来表示模型的复杂度。比如，在线性场景下， $\Omega(f)$  为模型中各权重参数构成向量的范数。

结构风险，记为  $J(f)$ ，是经验风险与正则项的加权和。即

$$J(f) = L(f) + \lambda \cdot \Omega(f)$$

其中  $\lambda$  被称为超参数（Hyperparameter），可用来平衡经验风险  $L(f)$  和正则项  $\Omega(f)$  对于目标函数的影响。这里  $\lambda \geq 0$ ； $\lambda$  增大，则正则项  $\Omega(f)$  的重要程度增加； $\lambda$  越趋近 0，则正则项  $\Omega(f)$  的重要程度越弱。特别地，当  $\lambda = 0$  时，则可以完全忽略正则项的影响。

### 8.1.3 线性模型与损失函数

考虑线性模型的情形。设训练集有  $m$  个特征  $x = \{x_1, \dots, x_m\}$ ，分类变量  $y$  的取值范围为  $\{0, 1\}$ ，对于权重参数  $w = \{w_0, w_1, \dots, w_m\}$ ，线性函数为

$$\eta(w, x) = w_0 + w_1 x_1 + \dots + w_m x_m$$

所以，线性模型的预测值函数为

$$f(x^{(i)}) = \eta(w, x^{(i)}) = w_0 + w_1 x_1^{(i)} + \dots + w_m x_m^{(i)}$$

即，线性模型的预测值函数是由参数  $w$  定义的，前面介绍的不少概念定义都可以被转化为与参数  $w$  的关系，如下所示：

- 损失函数  $L(y_i, f(x_i)) = L(w, x^{(i)}, y^{(i)})$
- 经验损失函数  $L(f) = L(w)$
- 正则项  $\Omega(f) = \Omega(w)$
- 结构风险  $J(f) = J(w)$

为了表述起来更简单、直接，我们使用如下定义：

经验损失函数为

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n L(\mathbf{w}, \mathbf{x}^{(i)}, y^{(i)})$$

其结构风险函数为

$$J(\mathbf{w}) = L(\mathbf{w}) + \lambda \cdot \Omega(\mathbf{w})$$

$\Omega(\mathbf{w})$  为衡量模型复杂程度的正则项，经常使用  $\mathbf{w}$  的 L1 范数和 L2 范数。

- L1 范数：

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = |w_0| + |w_1| + |w_2| + \cdots + |w_m|$$

- L2 范数：

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2 = w_0^2 + w_1^2 + w_2^2 + \cdots + w_m^2$$

下面将针对两个常用的线性分类模型（逻辑回归模型、线性支持向量机模型），详细介绍其损失函数。

#### 8.1.4 逻辑回归与线性支持向量机 (Linear SVM)

逻辑回归 (Logistic Regression) 算法使用的是对数损失函数  $\log_2(1 + e^{-x})$ ，分类标签取值  $y = \pm 1$  时，可得

$$L(\mathbf{w}, \mathbf{x}^{(i)}, y^{(i)}) = \log_2 \left( 1 + \exp \left( -y^{(i)} \cdot \eta(\mathbf{w}, \mathbf{x}^{(i)}) \right) \right)$$

逻辑回归模型的计算输出是分类值为 1 的概率：

$$P(1|x) = \frac{1}{1 + \exp\{-\eta(\mathbf{w}, \mathbf{x})\}}$$

最终的模型预测结果，会根据概率值是否大于 0.5，输出标签值 +1 或者 -1。

注意：在一些介绍逻辑回归算法的资料里，也有另外的表示方法，其本质是对取值范围的假设不同。

当标签取值  $y = \{0, 1\}$  时，由上面的正负例对应的损失值，可对应到正负例为  $y = 1$  和  $y = 0$  的情形：

- $y = 1$ , 为正例时, 损失值为  $\log(1 + \exp(-\eta(w, x)))$ 。
- $y = 0$ , 为负例时, 损失值为  $\log(1 + \exp(\eta(w, x)))$ 。

可以用一个表达式, 将这两种情况表示出来:

$$y \cdot \log(1 + \exp(-\eta(w, x))) + (1 - y) \log(1 + \exp(\eta(w, x)))$$

线性支持向量机 (Linear SVM) 算法使用的是合页损失函数  $\max(0, 1 - x)$ , 分类标签取值  $y = \pm 1$  时

$$L(w, x^{(i)}, y^{(i)}) = \max(0, 1 - y^{(i)} \cdot \eta(w, x^{(i)}))$$

线性支持向量机模型的计算输出为权重与特征值的乘积和:

$$f(x) = \eta(w, x) = w_0 + w_1 x_1 + \cdots + w_m x_m$$

最终的模型预测结果, 会根据计算值是否大于 0, 输出标签值 +1 或者 -1。

特别地, 取线性支持向量机的正则项为 L2 范数正则项, 则线性支持向量机的结构风险为

$$J(w) = L(w) + \lambda \cdot \|w\|_2^2$$

如果数据集正好可以被完全区分开, 则可以对应一个较直观的描述。

我们将  $x^{(i)}$  对应到  $m + 1$  维空间的点  $z^{(i)} = (1, x_1^{(i)}, \dots, x_m^{(i)})$ , 则

$$f(x^{(i)}) = \eta(w, x^{(i)}) = w_0 + w_1 x_1^{(i)} + \cdots + w_m x_m^{(i)} = w \cdot z^{(i)}$$

数据集正好可以被完全区分开。这就意味着, 可以找到  $w$ , 使下列情形成立:

- 对于所有  $y^{(i)} = -1$ , 对应的  $x^{(i)}$  满足  $f(x^{(i)}) \leq -1$ 。
- 对于所有  $y^{(i)} = +1$ , 对应的  $x^{(i)}$  满足  $f(x^{(i)}) \geq 1$ 。

对应到空间的点上, 就是

- 对于所有  $y^{(i)} = -1$ , 对应的  $z^{(i)}$  满足  $w \cdot z^{(i)} \leq -1$ 。
- 对于所有  $y^{(i)} = +1$ , 对应的  $z^{(i)}$  满足  $w \cdot z^{(i)} \geq 1$ 。

即,  $\{z^{(1)}, \dots, z^{(n)}\}$  被超平面  $w \cdot z = 0$  分隔开。更确切地说, 所有满足  $y^{(i)} = -1$  的点都在  $w \cdot z = -1$  的下方; 所有满足  $y^{(i)} = +1$  的点都在  $w \cdot z = 1$  的上方。超平面  $w \cdot z = -1$  与  $w \cdot z = 1$  之间的距离越大, 则模型的分隔效果越好。

我们注意到,  $\frac{w}{\|w\|_2}$  的 L2 范数为 1, 且是与超平面垂直的向量, 并且

$$w \cdot \left(z + \frac{w}{\|w\|_2} \cdot \frac{1}{\|w\|_2}\right) = w \cdot z + \frac{w \cdot w}{\|w\|_2^2} = w \cdot z + 1$$

有了这个等式，我们就很容易理解超平面  $\mathbf{w} \cdot \mathbf{z} = -1$  与  $\mathbf{w} \cdot \mathbf{z} = 1$  之间的距离为

$$\frac{2}{\|\mathbf{w}\|_2}$$

要使距离最大，就是要让  $\frac{2}{\|\mathbf{w}\|_2}$  最大，这等价于求  $\|\mathbf{w}\|_2$  的最小值。

我们回头再看线性支持向量机的结构风险：

$$J(\mathbf{w}) = L(\mathbf{w}) + \lambda \cdot \|\mathbf{w}\|_2^2$$

在数据集能被完全区分开的情况下，在迭代求解的过程中，会找到适合的一系列  $\mathbf{w}$ ，使得损失函数为 0，即  $L(\mathbf{w}) = 0$ ，随后的迭代求解最小  $J(\mathbf{w})$  就相当于在求  $\|\mathbf{w}\|_2$  的最小值，从而使分隔超平面的间隔最大化。

在实际的分类问题中，标签值往往为“是/否”“通过/不通过”等。Alink 的逻辑回归组件和线性 SVM 组件都会在训练前、预测后自动进行实际标签值与  $\pm 1$  的转换，用户使用时无须关注标签值是否为  $\pm 1$  形式的问题。

逻辑回归使用的对数损失函数  $\log(1 + e^{-x})$  与线性支持向量机使用的合页损失函数  $\max(0, 1 - x)$  的函数对比图像如图 8-2 所示。在图 8-2 的左图中，可以看到在 0 点附近两个函数的差异；图 8-2 的右图则体现了在自变量  $x$  为更大的负值时，损失函数间的差异。

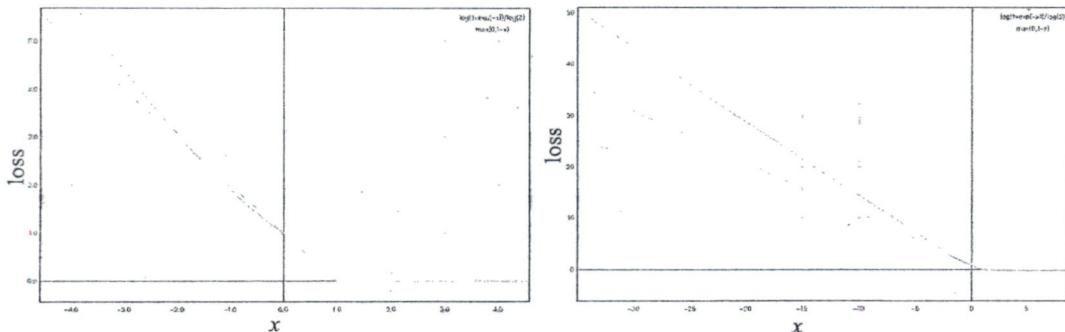


图 8-2

## 8.2 二分类评估方法

对于二分类问题，即实际分类值为“是”（Yes）或者“否”（No），机器学习模型预测会给出阳性（Positive）或阴性（Negative）的判断，从而产生四种不同的情况：真阳性（True Positive，

TP)、假阳性 (False Positive, FP)、真阴性 (True Negative, TN) 和假阴性 (False Negative, FN)，如表 8-1 所示。

表 8-1 实际值与预测结果

预测结果	实际值	
	是 (Yes)	否 (No)
阳性 (Positive)	真阳性 (TP)	假阳性 (FP)
阴性 (Negative)	假阴性 (FN)	真阴性 (TN)

Yes 实例指的是实际值为 Yes 的实例；No 实例指的是实际值为 No 的实例。考虑到实例可能被预测为阳性或阴性，结合表 8-1 的分类，Yes 实例是真阳性或假阴性的实例；No 实例是假阳性或真阴性的实例。

### 8.2.1 基本指标

由基本概念的定义，很容易理解下面这些基本指标。

真阳性率 (True Positive Rate, TPR)，也被称为灵敏度 (Sensitivity)、召回率 (Recall)，计算公式为

$$TPR = \frac{TP}{TP + FN}$$

描述的是分类器所识别出阳性的 Yes 实例占所有 Yes 实例的比例。

真阴性率 (True Negative Rate, TNR)，也被称为特异度 (Specificity)，计算公式为

$$TNR = \frac{TN}{FP + TN}$$

假阳性率 (False Positive Rate, FPR)，也被称为 1 - 特异度 (1 - Specificity)，其计算公式为

$$FPR = \frac{FP}{FP + TN} = 1 - \frac{TN}{FP + TN} = 1 - TNR$$

假阳性率计算的是分类器错认为阳性的 No 实例占所有 No 实例的比例。

准确率 (Precision)，或被称为 Positive Predictive Value (PPV)，定义如下：

$$PPV = \frac{TP}{TP + FP}$$

Negative Predictive Value (NPV) 的定义如下：

$$NPV = \frac{TN}{TN + FN}$$

False Discovery Rate (FDR) 的定义如下：

$$FDR = \frac{FP}{FP + TP} = 1 - PPV$$

Miss Rate 或 False Negative Rate (FNR) 的定义如下：

$$FNR = \frac{FN}{FN + TN}$$

将上面的定义汇总到表 8-2 中（注意，有几个上面未涉及的使用频次较低的定义也被放到了表 8-2 中，便于读者对比、清晰地了解各指标间的关系）。

表 8-2 指标汇总

预测结果	实际值			
	是 (Yes) TP + FN	否 (No) FP + TN		
Positive (阳性) TP + FP	True Positive (TP) (真阳性) TP	False Positive (FP) (假阳性) FP	Positive Predictive Value (PPV) 准确率 (Precision) $PPV = \frac{TP}{TP + FP}$	False Discovery Rate (FDR) $FDR = \frac{FP}{TP + FP}$
Negative (阴性) FN + TN	False Negative (FN) (假阴性) FN	True Negative (TN) (真阴性) TN	False Omission Rate (FOR) $FOR = \frac{FN}{FN + TN}$	Negative Predictive Value (NPV) $NPV = \frac{TN}{FN + TN}$
	True Positive Rate (TPR) 灵敏度 (Sensitivity) 召回率 (Recall) $TPR = \frac{TP}{TP + FN}$	False Positive Rate (FPR) 1-特异度 (1-Specificity) $FPR = \frac{FP}{FP + TN}$	Positive Likelihood Ratio (LR+) $LR+ = \frac{TPR}{FPR}$	Diagnostic Odds Ratio (DOR) $DOR = \frac{LR+}{LR-}$
	False Negative Rate (FNR) 丢失率 (Miss Rate) $FNR = \frac{FN}{TP + FN}$	True Negative Rate (TNR) 特异度 (Specificity) $TNR = \frac{TN}{FP + TN}$	Negative Likelihood Ratio (LR-) $LR- = \frac{FNR}{TNR}$	

在医学上，灵敏度 (Sensitivity) 指的是在实际的患病人群中，被诊断为阳性的比例；特异度 (Specificity) 指的是在实际没有患病的人群中，被诊断为阴性的比例。在信息检索领域，召回率 (Recall) 指的是检索到的相关文件数量与相关文件总数量的比值；准确率 (Precision) 指的是检索到的相关文件数量与检索到的文件总数量的比值。

## 8.2.2 综合指标

### 1. AUC指标

AUC (Area Under the Curve) 指的是 ROC 曲线下方的面积。ROC 曲线在 8.2.3 节中会具体介绍。随机分类的 AUC 为 0.5，而完美分类的 AUC 等于 1；实际上大多数分类模型的 AUC 介于 0.5 和 1 之间。

### 2. 精确度 (Accuracy)

精确度 (Accuracy, 缩写为 ACC) 被定义为预测正确的样本在整个样本中的比例，即

$$ACC = \frac{TP + TN}{TP + FN + FP + TN}$$

注意：精确度是一个很容易理解，并且常用的指标。读者要了解其局限性，避免在某些场景下误导自己的判断。比如，这里以选择 100 个样本为例，表 8-3 所示的模型预测情况如下：我们看到数据集中的负样本（实际值为 No 的样本）在数量上占优，有 95 个，达到了样本总数的 95%。分类模型只是简单地将所有样本都预测为阴性 (Negative)，精确度 Accuracy 就可以达到 95%，但对我们更为关注的正样本（实际值为 Yes 的样本），计算召回率为 0%。显然，这个模型并不是我们想要的。

表 8-3 模型预测情况示例

单位：样本个数

预测结果	实际值		
	是 (Yes)	否 (No)	合计
阳性 (Positive)	0	0	0
阴性 (Negative)	5	95	100
合计	5	95	100

### 3. $F_1$ -Score

下面先介绍一个更泛化的概念， $F$ -Score ( $F_a$ ) 又被称为  $F$ -Measure，可用来拟合准确率 (Precision) 与召回率 (Recall)，即将准确率与召回率的加权调和平均作为指标：

$$F_a = \frac{1}{\frac{1}{(a^2 + 1)} \times \frac{1}{Precision} + \frac{a^2}{(a^2 + 1)} \times \frac{1}{Recall}} = \frac{(a^2 + 1) \times Precision \times Recall}{a^2 \times Precision + Recall}$$

当参数 $\alpha = 1$ 时，就是最常见的 $F_1$ -Score，即

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

由于 $\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$ ,  $\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$

因此，又可以写作

$$F_1 = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

$F_1$ 指标（其也常常被称作 F1）指的是准确率与召回率的调和平均值（Harmonic Mean）。调和平均值又被称为倒数平均数，是总体各统计变量倒数的算术平均数的倒数。

我们做一下类比，以便轻松理解 $F_1$ 与准确率和召回率的关系。小学数学中的有些应用题，实际就是调和平均值的实际应用示例。例题如下：A、B两地分别位于河流的上下游，路程为 $S$ ，由于水流的关系，A地到B地的速度为 $v_1$ ，B地到A地的速度为 $v_2$ ，求A地到B地再返回A地的平均速度 $x$ 。

由

$$\frac{2S}{x} = \frac{S}{v_1} + \frac{S}{v_2}$$

可得：

$$x = \frac{2}{\frac{1}{v_1} + \frac{1}{v_2}}$$

所以，平均速度 $x$ 为速度为 $v_1$ 和 $v_2$ 的调和平均值。

#### 4. Kappa系数

1960年Cohen等人提出用Kappa系数（用希腊字母 $\kappa$ 表示）作为评判诊断试验一致性程度的指标。

对于二分类评估，可以使用Kappa系数来测量实际分类情况与预测结果的一致性程度。Kappa系数定义为

$$\kappa = \frac{P_a - P_e}{1 - P_e}$$

其中 $P_a$ 为精确度（Accuracy），表明实际分类情况与预测分类结果的实际一致率，即

$$P_a = \frac{TP + TN}{TP + FN + FP + TN}$$

$P_e$  为实际分类情况与预测分类结果的期望一致率，定义如下：

$$P_e = \frac{(TP + FN) \times (TP + FP) + (FP + TN) \times (FN + TN)}{(TP + FN + FP + TN) \times (TP + FN + FP + TN)}$$

Kappa 系数  $\kappa$  与精确度  $P_a$  的关系如下：

$$\kappa \leq P_a$$

证明过程很简单：

$$P_a - \kappa = P_a - \frac{P_a - P_e}{1 - P_e} = \frac{P_e(1 - P_a)}{1 - P_e}$$

由于  $0 \leq P_a \leq 1$ ,  $0 \leq P_e \leq 1$ , 因此

$$\kappa \leq P_a$$

使用 Kappa 系数的建议参考标准如下：

- $\kappa$  的值域为  $[-1; +1]$ 。
- 若  $0.75 \leq \kappa$ , 说明一致性较好。
- 若  $0.4 \leq \kappa < 0.75$ , 说明一致性一般。
- 若  $\kappa < 0.4$ , 说明一致性较差。

前面介绍精确度（Accuracy）时曾讲过一个例子，如表 8-3 所示。将所有 100 个样本都预测为阴性（Negative），虽然计算出的精确度（Accuracy）很高，有 95%，但是我们认为其分类模型很差。现在计算该模型的 Kappa 系数。

计算

$$P_a = \frac{95+0}{100} = 0.95, P_e = \frac{95 \times 100 + 5 \times 0}{100 \times 100} = 0.95$$

所以，Kappa 系数为

$$\kappa = \frac{P_a - P_e}{1 - P_e} = \frac{0.95 - 0.95}{1 - 0.95} = 0$$

我们看到，在这种情形下，模型的 Kappa 系数为 0。这说明精确度（Accuracy）和 Kappa 系数结合起来，对分类模型的评估更准确。

### 5. LogLoss

该指标与逻辑回归算法的经验损失函数相关。设分类标签取值 $y = \{0, 1\}$ ,  $P^{(i)}$ 指的是：逻辑回归模型的计算输出是分类值为 1 的概率：

$$P^{(i)} = P(1|x^{(i)}) = \frac{1}{1 + \exp\{-\eta(w, x^{(i)})\}}$$

则，损失函数

$$\begin{aligned} L(w, x^{(i)}, y^{(i)}) &= y^{(i)} \cdot \log(1 + \exp(-\eta(w, x^{(i)}))) + (1 - y^{(i)}) \log(1 + \exp(\eta(w, x^{(i)}))) \\ &= -y^{(i)} \cdot \log\left(\frac{1}{1 + \exp(-\eta(w, x^{(i)}))}\right) - (1 - y^{(i)}) \log\left(1 - \frac{1}{1 + \exp(-\eta(w, x^{(i)}))}\right) \\ &= -y^{(i)} \cdot \log(P^{(i)}) - (1 - y^{(i)}) \log(1 - P^{(i)}) \end{aligned}$$

所以，经验损失函数为

$$L(w) = \frac{1}{n} \sum_{i=1}^n L(w, x^{(i)}, y^{(i)}) = \frac{1}{n} \sum_{i=1}^n [-y^{(i)} \cdot \log(P^{(i)}) - (1 - y^{(i)}) \log(1 - P^{(i)})]$$

此经验损失函数值就是 LogLoss 指标，即

$$\text{LogLoss} = \frac{1}{n} \sum_{i=1}^n [-y^{(i)} \cdot \log(P^{(i)}) - (1 - y^{(i)}) \log(1 - P^{(i)})]$$

注意：对于分类标签 $y$ 为其他取值的情况，在 Alink 二分类评估组件中可以输入参数 PositiveLabelValueString，即指定 $y^{(i)} = 1$ 所对应的标签值；在 Alink 的预测详情信息中包含了正、负分类标签值及其概率。

### 8.2.3 评估曲线

对于二分类问题（分类值为“是”（Yes）或者“否”（No）），使用机器学习模型可以给每个实例预测出一个数值（不同分类模型的返回值有差异，比如，使用逻辑回归模型返回的“是”（Yes）的概率，值域为 $[0, 1]$ ；使用线性支持向量机模型得到的线性变换的值，值域为 $(-\infty, +\infty)$ ）。假设已确定一个阈值，比如 0.5，大于这个值的实例被预测为阳性，小于这个值

的实例则被预测为阴性，由公式可以计算出 FPR、TPR。如果增加阈值（如增加到 0.8），则会减少识别出的 Yes 实例，也就是降低了识别出的 Yes 实例占所有 Yes 实例的比例，即降低了 TPR；但同时也会有更少的 No 实例被预测为阳性，即降低了 FPR。如果减小阈值（如减少到 0.4），虽然能识别出更多的 Yes 实例，也就是提高了 TPR，但同时也将更多的 No 实例预测为阳性，即提高了 FPR。

根据我们的需求，如何确定最优的阈值？如何评估各种分类方法的优劣？

下面即将介绍的 ROC 曲线等提供了形象化的图形展示，便于我们做出决策。

### 1. ROC ( Receiver Operating Characteristic ) 曲线

ROC ( Receiver Operating Characteristic ) 曲线最初用来评价雷达性能，被称为接收者操作特性曲线。ROC 曲线指的是，根据一系列不同的阈值，得到相应的一系列二分类预测方式，每一个预测结果以一个点表示，每个点以真阳性率 (TPR) 为纵坐标，以假阳性率 (FPR) 为横坐标，再将所有的点连接起来，绘制成曲线。真阳性率代表获利能力，其值越高，获得的利益越多。假阳性率可以被看作成本，其值越高，成本就越高。在 ROC 曲线中，斜率较高的一段可获得较大的利益，同时付出较小的成本。

ROC 曲线的范围是由 (0,1)、(1,1)、(1,0)、(0,0) 四点构成的单位正方形区域。最好的可能预测方式在左上角，即 (0,1) 点，这代表 100% 灵敏（没有假阴性）和 100% 特异（没有假阳性）。一个完全随机预测会得到一条从左下到右上对角线（也叫无识别率线）上的一个点。一个最直观的采用随机预测的方式做决定的例子就是抛硬币。这条斜线将 ROC 空间划分为两个区域，在这条线以上的点代表一个好的分类结果，而在这条线以下的点代表差的分类结果。

为了将 ROC 曲线概括成单一的数量，可以选用曲线下方的面积 (Area Under Curve, AUC)。一般来说，AUC 值越大，分类预测效果越好。AUC 也可以被看作，分类预测将任意抽取的 Yes 实例排列在任意抽取的 No 实例之前的概率。

为了便于理解该算法，下面引用一个例子（参见链接 8-2）。表 8-4 是一个逻辑回归预测结果，它将得到的实数值按从大到小的顺序划分成 10 个个数大致相同的部分。

表 8-4 逻辑回归预测结果

百分位数	实例数	Yes 实例数	(1-特异度) / %	敏感度 / %
10	6180	4879	2.73	34.64
20	6180	2804	9.80	54.55
30	6180	2165	18.22	69.92
40	6180	1506	28.01	80.62
50	6180	987	38.90	87.62

续表

百分位数	实例数	Yes 实例数	(1-特异度) / %	敏感度 / %
60	6180	529	50.74	91.38
70	6180	365	62.93	93.97
80	6180	294	75.26	96.06
90	6180	297	87.59	98.17
100	6177	258	100.00	100.00

将逻辑回归得到的结果按从大到小的顺序排列，若以前 10% 的数值作为阈值，即将前 10% 的实例都预测为阳性，共 6180 个。其中，Yes 实例的个数为 4879 个，占所有 Yes 实例的百分比如下： $4879/14084 \times 100\% = 34.64\%$ （即敏感度）。另外，可得  $6180 - 4879 = 1301$  个 No 实例被错误地预测为阳性，占所有 No 实例的百分比如下： $1301/47713 \times 100\% = 2.73\%$ （即 1-特异度）。以这两组值分别作为  $x$  值和  $y$  值，在 Excel 中制作散点图，得到 ROC 曲线，横轴为 1-特异度（%），纵轴为敏感度（%），如图 8-3 所示。

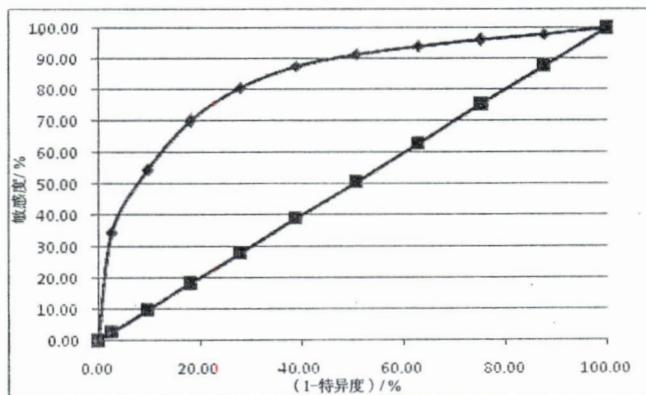


图 8-3

## 2. KS (Kolmogorov-Smirnov) 曲线

对于每个阈值点  $\alpha_i$ ，其真阳性率  $TPR_i = \frac{TP_i}{TP_i + FN_i}$  和假阳性率  $FPR_i = \frac{FP_i}{FP_i + TN_i}$  可以被看作正样本集合与负样本集合中，被预测为阳性的比例。如果采用随机采样模型，则正负样本集合中被采样到（即预测为阳性）的比例是相同的；这两个比例的差异越大，模型的效率就越高。

KS 曲线的横轴为  $\alpha_i$  的值，纵轴为真阳性率和假阳性率的差值，即  $TPR_i - FPR_i$ 。

该曲线纵轴绝对值的最大值，就等于 KS (Kolmogorov-Smirnov) 检验的统计量，所以，该曲线被命名为 KS 曲线。进一步，可以使用 KS 检验判断，在正样本集合与负样本集合中，样本

评分的分布是否有显著差异。

### 3. PR ( Precision Recall ) 曲线

PR 曲线很好理解，就如其名称，纵坐标和横坐标分别对应准确率（Precision）和召回率（Recall）。

对于每个阈值点 $\alpha_i$ ，计算曲线上的点。

- 横坐标为召回率（Recall）：

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

- 纵坐标为准确率（Precision）：

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

很多算法会用指标 Precision 和 Recall 来评估自身的好坏，但事实上这两者在某些情况下是此消彼长的关系。可以通过绘制 Precision-Recall 曲线来分析、获取适合的阈值。

### 4. Lift ( 提升 ) 图

如果我们从训练样本中随机采样，则得到的样本为阳性的概率，就是训练样本中的正样本的概率值 ( $\pi_+ = \frac{\text{TP} + \text{FN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}}$ )。现在我们使用模型的预测效果应该更好，即预测为阳性的样本中是正样本的概率值（即  $\text{Precision} = \text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}}$ ）会更大。Lift 就是这两个概率值的比值，表示提升的倍数，即

$$\text{Lift} = \frac{\text{Precision}}{\pi_+} = \frac{\frac{\text{TP}}{\text{TP} + \text{FP}}}{\frac{\text{TP} + \text{FN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}}}$$

显然，Lift 值越大，模型的效果就越好。当 Lift 值为 1 时，表示效果没有提升，依靠模型预测的效果与随机采样的效果是一样的。

### 5. CAP ( Cumulative Accuracy Profile ) 曲线

CAP 曲线上各点坐标的定义如下：

- X 轴代表预测结果为阳性的样本占总样本的比例：

$$\text{PR}_i = \frac{\text{TP}_i + \text{FP}_i}{\text{TP}_i + \text{FN}_i + \text{FP}_i + \text{TN}_i}$$

- Y轴代表正样本中被预测为阳性的比例，即真阳性率 (TPR)、召回率 (Recall)：

$$\text{Recall}_i = \text{TPR}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

对于每个阈值点 $\alpha_i$ ，可以得到相应的 $\{\text{TP}_i, \text{FN}_i, \text{FP}_i, \text{TN}_i\}$ 。下面将 5 种曲线所需要的指标总结如下。

- (1) 真阳性率 (TPR)、召回率 (Recall)：

$$\text{Recall}_i = \text{TPR}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

- (2) 假阳性率 (FPR)：

$$\text{FPR}_i = \frac{\text{FP}_i}{\text{FP}_i + \text{TN}_i}$$

- (3) 准确率 (Precision)、Positive Predictive Value (PPV)：

$$\text{Precision}_i = \text{PPV}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

- (4) 预测结果为阳性的样本占总样本的比例：

$$\text{PR}_i = \frac{\text{TP}_i + \text{FP}_i}{\text{TP}_i + \text{FN}_i + \text{FP}_i + \text{TN}_i}$$

- (5) 训练样本中正样本的概率值：

$$\pi_+ = \frac{\text{TP}_i + \text{FN}_i}{\text{TP}_i + \text{FN}_i + \text{FP}_i + \text{TN}_i}$$

注意： $\pi_+$ 可以被看作一个常量， $\text{TP}_i$ 与 $\text{FN}_i$ 的和为正样本的总数，是不随阈值点 $\alpha_i$ 而改变的；同样，分母上 4 个数的和为全体样本的总数，也是不随阈值点 $\alpha_i$ 而改变的。

基于上面汇总的指标，可以更清晰地看到 5 种评估曲线的关联与区别，如表 8-5 所示。

表 8-5 5 种评估曲线的对比

名称	横坐标 (X 轴)	纵坐标 (Y 轴)
ROC 曲线	$\text{FPR}_i$	$\text{TPR}_i$
KS 曲线	$\alpha_i$	$\text{TPR}_i - \text{FPR}_i$

续表

名称	横坐标 (X 轴)	纵坐标 (Y 轴)
PR 曲线	$\text{Recall}_i$	$\text{Precision}_i$
Lift (提升) 图	$\text{PR}_i$	$\frac{\text{Precision}_i}{\pi_+}$
CAP 曲线	$\text{PR}_i$	$\text{Recall}_i$

### 8.3 数据探索

纸钞认证数据集( Banknote Authentication Data Set )来源于 UC Machine Learning Repository，可从链接 8-3 下载。

其数据是从真钞和假钞的图像样本中提取出来的。每个图像均为 400 像素×400 像素。这些图像是分辨率约为 660dpi 的灰度图像。可通过小波变换从图像中提取 4 个特征。一共有 1372 个样本，每个样本有 4 个输入变量和 1 个输出变量。变量名如下：

- (1) variance: 小波变换图像的方差。
- (2) skewness: 小波变换图像的偏度。
- (3) kurtosis: 小波变换图像的峰度。
- (4) entropy: 图像熵。
- (5) class: 类别 (0 代表真钞，1 代表假钞)。

我们先下载数据，之后使用文本编辑器即可打开该数据文件，如图 8-4 所示。

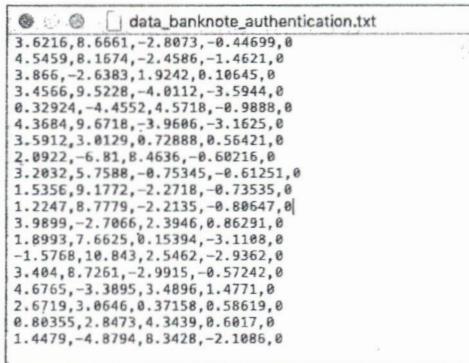


图 8-4

我们看到每行为一条数据，各列数据间以逗号分隔。这是一个典型的 CSV 格式数据，可以使用 CsvSourceBatchOp 组件直接读取。关于 CsvSourceBatchOp 等数据文件相关操作的详细介绍，请参阅 3.2.1 节。

首先，定义相关的一些常量、原始数据及后续处理数据所在的文件夹路径 DATA\_DIR（该路径在数据根文件夹下的 banknote 子文件夹中）、原始数据文件的名称、SCHEMA\_STRING（CsvSourceBatchOp 组件所需的参数）。将各列的名称和类型使用字符串形式表示出来，此内容在 2.8 节中已有详细介绍，这里不再展开说明。

---

```
static final String DATA_DIR = Utils.ROOT_DIR + "banknote" + File.separator;
static final String ORIGIN_FILE = "data_banknote_authentication.txt";
static final String SCHEMA_STRING
    = "variance double, skewness double, kurtosis double, entropy double, class int";
```

---

然后，使用 CsvSourceBatchOp 组件读取数据，并打印显示信息，代码如下：

---

```
CsvSourceBatchOp source =
new CsvSourceBatchOp()
.setFilePath(DATA_DIR + ORIGIN_FILE)
.setSchemaStr(SCHEMA_STRING);

System.out.println("schema of source:");
System.out.println(source.getSchema());

System.out.println("column names of source:");
System.out.println(ArrayUtils.toString(source.getColNames()));

System.out.println("column types of source:");
System.out.println(ArrayUtils.toString(source.getColTypes()));

source.firstN(5).print();
```

---

前 4 行代码定义了 CsvSourceBatchOp 的实例 source；随后，展示了可以从 source 中获取 Schema 信息、列名数组信息、列类型数组信息；最后一行的 firstN(5)方法，是从批式组件中取前 5 行数据，然后选择打印方法输出结果。

```
schema of source:
root
|--- variance: DOUBLE
|--- skewness: DOUBLE
|--- kurtosis: DOUBLE
|--- entropy: DOUBLE
|--- class: INT

column names of source:
```

```

{variance, skewness, kurtosis, entropy, class}
column types of source:
{Double, Double, Double, Double, Integer}
variance|skewness|kurtosis|entropy|class
-----|-----|-----|-----|-----
3.6216|8.6661|-2.8073|-0.4470|0
4.5459|8.1674|-2.4586|-1.4621|0
3.8660|-2.6383|1.9242|0.1065|0
3.4566|9.5228|-4.0112|-3.5944|0
0.3292|-4.4552|4.5718|-0.9888|0

```

前面打印输出的 Schema 信息、列名数组信息、列类型数组信息，都和我们对数据的理解相一致，不用进行说明。后面打印的 5 行数据内容，采用的是 Markdown 表格的形式。可以通过 Markdown 编辑器将这些内容转换为一般的表格形式，以便更好地展现数据，如表 8-6 所示。

表 8-6 5 条 iris 数据

variance	skewness	kurtosis	entropy	class
3.6216	8.6661	-2.8073	-0.4470	0
4.5459	8.1674	-2.4586	-1.4621	0
3.8660	-2.6383	1.9242	0.1065	0
3.4566	9.5228	-4.0112	-3.5944	0
0.3292	-4.4552	4.5718	-0.9888	0

### 8.3.1 基本统计

使用 SummarizerBatchOp 可以轻松计算数据表各列数据的基本统计信息。下列代码给出了 SummarizerBatchOp 的基本使用方式：

---

```

TableSummary summary = new SummarizerBatchOp().linkFrom(source).collectSummary();
System.out.println("Count of data set : " + summary.count());
System.out.println("Max value of entropy : " + summary.max("entropy"));
System.out.println(summary);

```

---

运行结果如下，先打印输出记录的总条数，随后输出“entropy”列的最大值，最后执行 `println(summary)` 操作，将 `summary.toString()` 的结果打印了出来：

```

Count of data set : 1372
Max value of entropy : 2.4495
+---+---+---+---+---+
| colName|count|missing|      sum|    mean|variance|      min|      max|
+---+---+---+---+---+
| variance| 1372|      0| 595.0848| 0.4337| 8.0813| -7.0421| 6.8248|

```

skewness	1372	0	2637.4685	1.9224	34.4457	-13.7731	12.9516
kurtosis	1372	0	1917.5444	1.3976	18.5764	-5.2861	17.9274
entropy	1372	0	-1634.9527	-1.1917	4.4143	-8.5482	2.4495
class	1372	0	610	0.4446	0.2471	0	1

上述代码通过 collectSummary 方法，触发 SummarizerBatchOp 及其上游关联组件的执行，得到任务运行结果（TableSummary 类的实例 summary），从而可以获取、打印输出任何基本统计量。Alink 提供了 Lazy 机制（详见 2.5 节中的相关内容），SummarizerBatchOp 组件不会马上被触发执行，而会和其他组件共同在一个任务中执行，一起返回结果。这样就节省了多次启动任务的时间，也可以减少重复计算，如此一来极大地提高了我们处理批式任务的效率。使用 lazyCollectSummary 方法的代码如下，通过 Java 的 Consumer 接口，异步接收处理 summary 的返回结果：

---

```
source
  .link(
    new SummarizerBatchOp()
      .lazyCollectSummary(
        new Consumer<TableSummary>() {
          @Override
          public void accept(TableSummary tableSummary) {
            System.out.println("Count of data set : " + tableSummary.ccount());
            System.out.println("Max value of entropy : " + tableSummary.max("entropy"));
            System.out.println(tableSummary);
          }
        }
      )
  );

```

---

上述代码可以获得与前面示例代码一样的输出，这里就不重复显示了。

基本统计的输出表包含了常用的信息，我们还可以简化代码，如下所示：

---

```
source
  .link(
    new SummarizerBatchOp()
      .lazyPrintSummary()
  );

```

---

这里显示的就是基本统计的信息，以表格形式呈现。与前面使用 lazyCollectSummary 方法相比，这种写法简单了很多。

基本统计是常用操作。在任务中多次调用基本统计操作，可以帮助我们了解整个流程中发生的变化。基本统计的调用能否再简单一点呢？Alink 为每个批式组件提供了 lazyPrintStatistics 方法，该方法就是对 new SummarizerBatchOp().lazyPrintSummary() 的包装。其用法更加简单，示例代码如下。这里对原始数据进行了统计，然后取前 5 条数据，再对取出的这 5 条数据进行统计，

最后打印输出。注意：经过 lazyPrintStatistics 操作，数据没有改变。

---

```
source
    .lazyPrintStatistics("<- origin data ->")
    .firstN(5)
    .lazyPrintStatistics("<- first 5 data ->")
    .print();
```

---

运行结果如下。第一次对原始数据进行统计，共有 1372 条数据。执行 firstN(5)后，第二次统计结果显示的是 5 条数据。经过 lazyPrintStatistics 操作，数据没有改变，最后打印输出的仍是 5 条数据。

```
<- origin data ->
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance |
+-----+-----+-----+-----+-----+-----+
| variance | 1372 | 0 | 595.0848 | 0.4337 | 8.0813 | -7.0421 | 6.8248 |
| skewness | 1372 | 0 | 2637.4685 | 1.9224 | 34.4457 | -13.7731 | 12.9516 |
| kurtosis | 1372 | 0 | 1917.5444 | 1.3976 | 18.5764 | -5.2861 | 17.9274 |
| entropy | 1372 | 0 | -1634.9527 | -1.1917 | 4.4143 | -8.5482 | 2.4495 |
| class | 1372 | 0 | 610 | 0.4446 | 0.2471 | 0 | 1 |
```

```
<- first 5 data ->
+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance |
+-----+-----+-----+-----+-----+
| variance | 5 | 0 | 4.0304 | 0.8061 | 3.0154 | -1.2528 | 3.5458 |
| skewness | 5 | 0 | 17.2018 | 3.4404 | 47.9574 | -4.4552 | 10.2036 |
| kurtosis | 5 | 0 | 7.8499 | 1.57 | 10.8332 | -4.0351 | 4.5718 |
| entropy | 5 | 0 | -11.7274 | -2.3455 | 5.3836 | -5.6038 | -0.1935 |
| class | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
```

```
variance|skewness|kurtosis|entropy|class
-----+-----+-----+-----+
-1.2528|10.2036|2.1787|-5.6038|0
0.5195|-3.2633|3.0895|-0.9849|0
0.3292|-4.4552|4.5718|-0.9888|0
0.8887|5.3449|2.0450|-0.1936|0
3.5458|9.3718|-4.0351|-3.9564|0
```

### 8.3.2 相关性

前面，我们通过基本统计对单个数值列有了一定的了解。下面，我们关注各数据列之间的关系，尤其是特征数据列与类别数据列之间的关系。这里先计算和显示一下相关系数，代码如下。使用 collectCorrelation 触发任务执行，返回相关性结果，从中抽取所需信息进行输出。

---

```
CorrelationResult correlation = new CorrelationBatchOp().linkFrom(source).collectCorrelation();
String[] colNames = correlation.getColNames();
System.out.print("Correlation of " + colNames[0] + " with " + colNames[1]);
System.out.println(" is " + correlation.getCorrelation()[0][1]);
System.out.println(correlation.getCorrelationMatrix());
```

---

显示结果如下：

---

```
Correlation of variance with skewness is 0.26402552997043593
mat[5,5]:
1. 0, 0.26402552997043593, -0.3808499720462522, 0.2768166960053636, -0.7248431424446056
0.26402552997043593, 1. 0, -0.7868952243065793, -0.5263208425437146, -0.44468775759659307
-0.3808499720462522, -0.7868952243065793, 1. 0, 0.31884088768744584, 0.15588323600923013
0.2768166960053636, -0.5263208425437146, 0.31884088768744584, 1. 0, -0.023423678954851614
-0.7248431424446056, -0.44468775759659307, 0.15588323600923013, -0.023423678954851614, 1. 0
```

---

与基本统计操作类似，我们可以使用 Lazy 方式提高效率。具体代码如下：

---

```
source
.link(
    new CorrelationBatchOp()
        .lazyCollectCorrelation(new Consumer<CorrelationResult>() {
            @Override
            public void accept(CorrelationResult correlationResult) {
                String[] colNames = correlationResult.getColNames();
                System.out.print("Correlation of " + colNames[0] + " with " + colNames[1]);
                System.out.println(" is " + correlationResult.getCorrelation()[0][1]);
                System.out.println(correlationResult.getCorrelationMatrix());
            }
        })
);
```

---

运行结果与前面使用 collectCorrelation 方法得到的结果相同。

相关性计算结果可以通过把列名信息与相关性矩阵整合为一张表格来清晰地展现。相关性组件提供的 lazyPrintCorrelation 方法，便可直接打印结果表格，具体 Java 代码如下所示。特别地，我们默认使用 Pearson 相关系数，另外一种 Spearman 相关系数可通过设置 Method 参数进行切换。很明显，使用 lazyPrintCorrelation 方法比使用 lazyCollectCorrelation 方法的代码简化了很多。

---

```
source
.link(
    new CorrelationBatchOp()
        .lazyPrintCorrelation("< Pearson Correlation >")
);
source.link()
```

---

```

new CorrelationBatchOp()
.setMethod(Method.SPEARMAN)
.lazyPrintCorrelation("< Spearman Correlation >")
);

BatchOperator.execute();

```

运行结果如下：

```

< Pearson Correlation >
----- Correlation -----
colName|variance|skewness|kurtosis|entropy|class
-----|-----|-----|-----|-----|-----
variance|1.0000|0.2640|-0.3808|0.2768|-0.7248
skewness|0.2640|1.0000|-0.7869|-0.5263|-0.4447
kurtosis|-0.3808|-0.7869|1.0000|0.3188|0.1559
entropy|0.2768|-0.5263|0.3188|1.0000|-0.0234
class|-0.7248|-0.4447|0.1559|-0.0234|1.0000
< Spearman Correlation >
----- Correlation -----
colName|variance|skewness|kurtosis|entropy|class
-----|-----|-----|-----|-----|-----
variance|1.0000|0.2551|-0.3267|0.2415|-0.6404
skewness|0.2551|1.0000|-0.7294|-0.5725|-0.3834
kurtosis|-0.3267|-0.7294|1.0000|0.4333|0.0694
entropy|0.2415|-0.5725|0.4333|1.0000|-0.0139
class|-0.6404|-0.3834|0.0694|-0.0139|1.0000

```

结合两种相关系数可知，类别 class 列与 variance 列的相关性最强，类别 class 列与 skewness 列的相关性次之；类别 skewness 列与 kurtosis 列之间有较强的负相关性。

可使用 Python 的工具函数，使用多变量图来展示出各变量间两两的关系。

如图 8-5 所示，对角线上为单个变量的直方分布图。由于选择了分类值，因此每个柱子是按分类值叠加展示的。在此可以看出每个柱子所代表区间内，不同分类的比例。variance 所对应的直方图在图 8-5 的左上角。不同分类值所对应的分布有显著不同。分类值 1 所对应的灰色部分偏向左侧，而分类值 0 所对应的黑色部分偏向右边。再看与 variance 相关的散点图，灰色部分与黑色部分有交叠，但也有较大的纯色区域。

在前面的散点图中，选择两个特征，比如 variance 和 skewness，数据点间的交叠部分较小。随着更多特征的加入，数据点应该有更好的区分，但是我们没法形象地“看到”其在高维空间如何分布。这就需要借助数据降维与可视化方法，将数据展示在平面或三维空间中。使用 t-SNE 方法得到的可视化图像参见图 8-6。

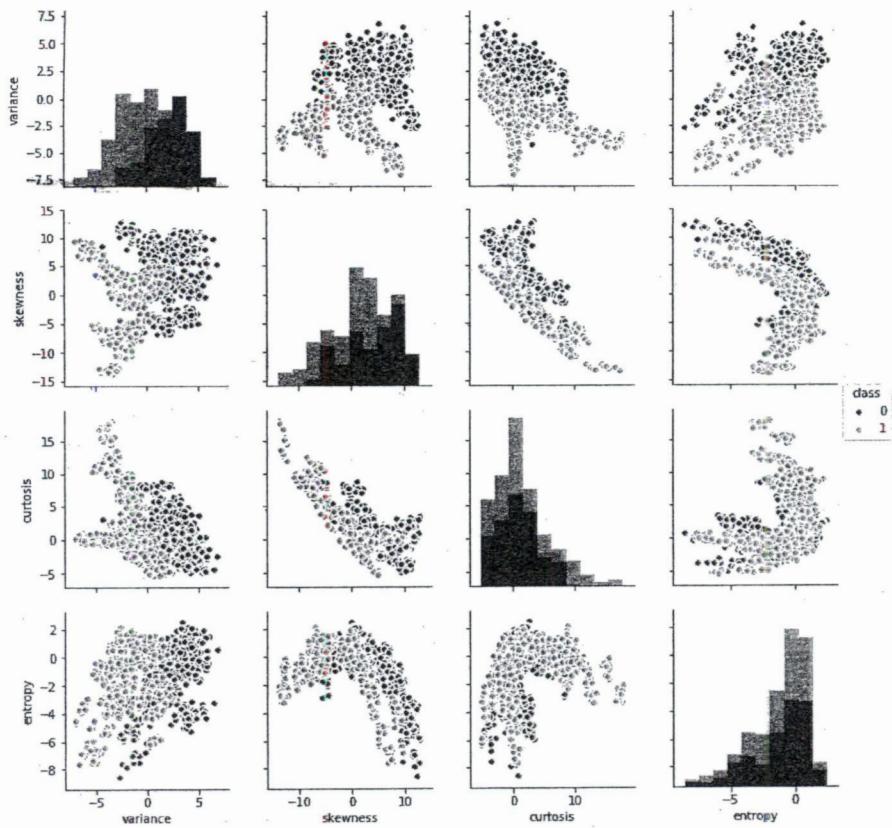


图 8-5

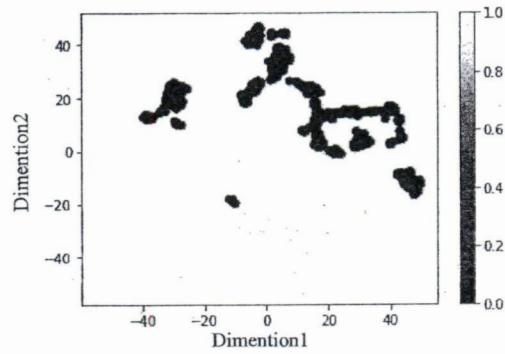


图 8-6

从图 8-6 中可以看到，两个类别间似乎有一个分隔，大部分样本都会被分到正确的类别。这也给了我们很强的信心：后面我们会尝试各种分类方法，争取拿到更好的分类模型。

## 8.4 训练集和测试集

将原始的数据集分为训练集和测试集两部分。训练集用来建立分类模型，然后，使用分类模型对测试集的数据进行预测，并对比预测值与原始分类标记值，给出模型评估指标。

本节内容涉及两个重要组件：

- 数据划分组件 SplitBatchOp，详见 7.2 节。
- 批式 AK 文件格式导出组件 AkSinkBatchOp，详见 3.2.3 节。

由于数据划分在后面的章节中频繁出现，因此我们在工具类 Utils 中封装定义了相关的函数。具体代码如下。原始数据 source 会被分成两部分，第一部分为 ratio，其余在第二部分。第一部分会被看作训练集，另一部分被看作测试集。我们会直接把划分出来的数据集分别保存为本地数据文件。在数据格式方面选择 AK 格式，以使数据读取操作更加简便。

---

```

public static void splitTrainTestIfNotExist(
    BatchOperator <?> source,
    String trainFilePath,
    String testFilePath,
    double ratio
) throws Exception {
    if ((!new File(trainFilePath).exists()) && (!new File(testFilePath).exists())) {
        SplitBatchOp spliter = new SplitBatchOp().setFraction(ratio);

        source.link(spliter);

        spliter
            .link(
                new AkSinkBatchOp()
                    .setFilePath(trainFilePath)
            );
    }

    spliter.getSideOutput(0)
        .link(
            new AkSinkBatchOp()
                .setFilePath(testFilePath)
        );
}

BatchOperator.execute();
}
}

```

---

使用如下代码进行数据划分，其中训练集占总数据量的 80%。训练集和测试集的数据文件都以 “.ak” 为扩展名，即均为 AK 格式的数据文件。

---

```
static final String TRAIN_FILE = "train.ak";
static final String TEST_FILE = "test.ak";

Utils.splitTrainTestIfNotExist(
    source,
    DATA_DIR + TRAIN_FILE,
    DATA_DIR + TEST_FILE,
    0.8
);
```

---

有了训练集和测试集的数据，每次实验的时候就可以直接从文件路径读取。定义相应的 AkSourceBatchOp，代码如下：

---

```
AkSourceBatchOp train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE);
AkSourceBatchOp test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);
```

---

## 8.5 逻辑回归模型

这里主要介绍对于当前的二分类问题，通过逻辑回归训练组件和预测组件来实现建模和预测。

首先设置逻辑回归训练（LogisticRegressionTrainBatchOp）组件和逻辑回归预测（LogisticRegressionPredictBatchOp）组件，代码如下。训练组件需要指定特征列和标签列；预测组件需要指定预测结果列名，即参数 predictionCol，分类预测结果会被放在该列。另外，如果想要获取更多的信息，了解预测过程中对各分类情况的概率值，可以通过设置预测详情列名，即参数 predictionDetailCol 来完成。设置了该列名，预测结果数据集中就会有这样一行：该列为字符串类型值，给出了属于各分类值的概率，可用于二分类评估。

---

```
LogisticRegressionTrainBatchOp lrTrainer =
    new LogisticRegressionTrainBatchOp()
        .setFeatureCols(FEATURE_COL_NAMES)
        .setLabelCol(LABEL_COL_NAME);

LogisticRegressionPredictBatchOp lrPredictor =
    new LogisticRegressionPredictBatchOp()
        .setPredictionCol(PREDICTION_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME);
```

---

然后，通过 link 方法，建立起整个实验流程，代码如下。我们通过这两行代码的描述，可以想象出整个流程：训练数据源 train\_data 连接逻辑回归训练组件 lrTrainer；预测过程需要模型，

也需要待预测的数据，所以逻辑回归预测组件 `lrPredictor` 需要使用 `linkFrom` 方法来同时连接 `lrTrainer` 输出的模型和待预测的数据源 `test_data`；`lrPredictor` 组件的输出即预测结果。

---

```
train_data.link(lrTrainer);
lrPredictor.linkFrom(lrTrainer, test_data);
```

---

在训练过程中，我们还希望看到训练信息（`TrainInfo`）和模型信息（`ModelInfo`），以及最终的预测结果。我们希望能看到几条数据，对运行结果有一个直观的了解。以上这些信息最好能运行一次任务就可全部得到。因此，这里选择了 Lazy 方式。具体代码如下：

---

```
lrTrainer.lazyPrintTrainInfo().lazyPrintModelInfo();
lrPredictor.lazyPrint(5, "< Prediction >");
BatchOperator.execute()
```

---

训练信息如下：

```
----- train meta info -----
{model name: Logistic Regression, num feature: 4}
----- train importance info -----
| colName|importanceValue| colName|weightValue|
|-----|-----|-----|-----|
| skewness|    24.05967904| entropy|-0.68830239|
| kurtosis|   22.17694646|skewness|-4.14845206|
| variance|  21.81129809|kurtosis|-5.20781697|
| entropy|     1.47179204|variance|-7.68311606|
----- train convergence info -----
step:0 loss:0.67538777 gradNorm:0.42586577 learnRate:0.40000000
step:1 loss:0.50807036 gradNorm:0.41072258 learnRate:1.60000000
step:2 loss:0.27567906 gradNorm:0.26515855 learnRate:4.00000000
...
step:32 loss:0.02060997 gradNorm:0.00055431 learnRate:4.00000000
step:33 loss:0.02060759 gradNorm:0.00009935 learnRate:4.00000000
step:34 loss:0.02060757 gradNorm:0.00000829 learnRate:4.00000000
```

在训练信息中给出了对于训练数据计算出来的特征重要性（`importance`）。在此可以看到特征 `skewness`、`kurtosis` 和 `variance` 的重要性数值差别较小，特征 `entropy` 的重要性数值比前面几个特征的重要性数值要小得多。随后可看到逻辑回归迭代训练中每步的主要指标。在此可看到本次训练是在第 34 轮中结束的，属于满足终止条件、提前结束迭代训练的情况。

模型信息输出如下，在此可以看到各个模型的权重系数：

```
----- model meta info -----
{hasInterception: true, model name: Logistic Regression, num feature: 4}
----- model weight info -----
|intercept|  variance|  skewness|  kurtosis|  entropy|
|-----|-----|-----|-----|-----|
|  6.9651| -7.68311606| -4.14845206| -5.20781697| -0.68830239|
```

预测结果的前 5 项如下所示。预测结果项返回的是预测标签值，详细信息列返回的是各标签值的概率。

```
< Prediction >
variance|skewness|kurtosis|entropy|class|pred|predinfo
-----|-----|-----|-----|-----|-----|-----
-2.4953|11.1472|1.9353|-3.4638|0|0|{"0": "0.9999999999991567", "1": "8.433254095052689E-13"}
-1.3000|10.2678|-2.9530|-5.8638|0|0|{"0": "0.9980296917461061", "1": "0.001970308253893882"}
5.2423|11.0272|-4.3530|-4.1013|0|0|{"0": "1.0", "1": "0.0"}
2.0843|6.6258|0.4838|-2.2134|0|0|{"0": "1.0", "1": "0.0"}
3.8200|10.9279|-4.0112|-5.0284|0|0|{"0": "1.0", "1": "0.0"}
```

从这 5 条数据上看，预测得比较准确。

## 8.6 线性SVM模型

本节将介绍如何使用线性 SVM 训练组件和线型 SVM 预测组件来解决二分类问题。

首先设置线性 SVM 训练组件（LinearSvmTrainBatchOp）组件和线型 SVM 预测组件（LinearSvmPredictBatchOp），代码如下。与逻辑回归算法相似，线性 SVM 训练组件需要指定特征列和标签列；线性 SVM 预测组件需要指定预测结果列名，在此还可以通过设置预测详情列名，列出属于各分类值的概率，用于后面的二分类评估。

---

```
LinearSvmTrainBatchOp svmTrainer =
    new LinearSvmTrainBatchOp()
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME);

LinearSvmPredictBatchOp svmPredictor =
    new LinearSvmPredictBatchOp()
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME);
```

---

建立起整个实验流程：训练数据源 train\_data 连接线性 SVM 训练组件 svmTrainer；预测过程需要线性 SVM 模型，也需要待预测的数据，所以预测组件 svmPredictor 需要使用 linkFrom 方法来同时连接 svmTrainer 输出的模型和待预测的数据源 test\_data；svmPredictor 组件的输出即预测结果。代码如下：

---

```
train_data.link(svmTrainer);
svmPredictor.linkFrom(svmTrainer, test_data);
```

---

在训练过程中，我们还仿照前面的逻辑回归算法，使用 Lazy 方式打印输出训练信息（TrainInfo）和模型信息（ModelInfo），以及最终的预测结果。具体代码如下：

---

```
svmTrainer.lazyPrintTrainInfo().lazyPrintModelInfo();

svmPredictor.lazyPrint(5, "< Prediction >");

BatchOperator.execute();
```

---

训练信息如下：

```
----- train meta info -----
{model name: Linear SVM, num feature: 4}
----- train importance info -----
| colName|importanceValue| colName|weightValue|
|-----|-----|-----|-----|
|skewness|     8.43892212| entropy|-0.22797316|
|kurtosis|      7.74791237| skewness|-1.45506778|
|variance|      7.62486689| kurtosis|-1.81944388|
| entropy|      0.48747337| variance|-2.68588953|
----- train convergence info -----
step:0 loss:0.43263605 gradNorm:0.85130570 learnRate:0.40000000
step:1 loss:0.26491660 gradNorm:0.73496319 learnRate:1.60000000
step:2 loss:0.06516288 gradNorm:0.45340859 learnRate:4.00000000
...
step:24 loss:0.01192161 gradNorm:0.00012197 learnRate:4.00000000
step:25 loss:0.01192159 gradNorm:0.00003354 learnRate:4.00000000
step:26 loss:0.01192159 gradNorm:0.00000841 learnRate:4.00000000
```

在此可从特征重要性的角度来说（参考训练信息的 importanceValue 项），特征 skewness、kurtosis 和 variance 的重要性数值差别较小，特征 entropy 的重要性数值比前面几个特征的重要性数值要小得多。本次训练是在第 26 轮中满足终止条件，提前结束的。在上一个实验中，逻辑回归训练是在第 34 轮结束的。

模型信息如下：

```
----- model meta info -----
{hasInterception: true, model name: Linear SVM, num feature: 4}
----- model weight info -----
|intercept| variance| skewness| kurtosis|   entropy|
|-----|-----|-----|-----|-----|
| 2.4171|-2.68588953|-1.45506778|-1.81944388|-0.22797316|
```

预测结果如下：

```
< Prediction >
variance|skewness|kurtosis|entropy|class|pred|predinfo
```

----- ----- ----- ----- ----- -----
0.8887 5.3449 2.0450 -0.1936 0 0 {"0": "0.9999890670812613", "1": "1.0932918738659758E-5"}
2.6799 3.1349 0.3407 0.5849 0 0 {"0": "0.9999587398819237", "1": "4.12601180762584E-5"}
4.5459 8.1674 -2.4586 -1.4621 0 0 {"0": "0.999999528621162", "1": "4.7137883818493265E-8"}
4.4338 9.8870 -4.6795 -3.7483 0 0 {"0": "0.9999995002912768", "1": "4.997087231783937E-7"}
-0.7829 11.3603 -0.3764 -7.0495 0 0 {"0": "0.9999398211526664", "1": "6.0178847333558494E-5"}

## 8.7 模型评估

二分类模型评估需要在模型预测结果数据集中包含如下信息：

- 标签列——即原始的分类标记结果，用来验证预测的正确性。
- 预测详情列——该列包含了该条记录预测为各分类值的概率，概率最大的就是分类预测结果。

二分类评估组件 EvalBinaryClassBatchOp 需要两个必选参数，即所要评估的预测结果数据集的标签列名称和预测详情列名称。此外，还有一个可选参数 PositiveLabelValueString（作为正例的标签值，选择不同的标签值作为正例会影响召回率等指标的值。当用户没有输入该参数时，默认会使用标签值降序排列。位于首位的标签值作为正例标签值。注意：为了统一输入类型，这里使用的是正例标签值转为字符串的结果）。

从二分类评估组件 EvalBinaryClassBatchOp 中获取评估指标。这里使用的是该组件的 collectMetrics 方法。在此沿用 Flink 的风格，collect 方法和 print 方法都会触发运行相应的任务。任务结束后返回结果，所以我们无须再调用 execute 方法。

具体代码如下。将前面的逻辑回归预测结果存储于 AK 格式的文件 DATA\_DIR + LR\_PRED\_FILE 中。在此可以先定义评估组件 EvalBinaryClassBatchOp，然后使用 linkFrom 方法，将预测结果数据源与评估组件连接起来。

```
BinaryClassMetrics lr_metrics =
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("1")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .linkFrom(
            new AkSourceBatchOp().setFilePath(DATA_DIR + LR_PRED_FILE)
        )
        .collectMetrics();
```

接下来，从评估结果中抽取我们需要的指标。首先抽取常用的指标值：

```
StringBuilder sbd = new StringBuilder();
sbd.append("<LR>\n")
```

---

```

.append("AUC : ").append(lr_metrics.getAuc())
.append("\t Accuracy : ").append(lr_metrics.getAccuracy())
.append("\t Precision : ").append(lr_metrics.getPrecision())
.append("\t Recall : ").append(lr_metrics.getRecall())
.append("\n");
System.out.println(sbd.toString());

```

---

运行结果如下：

```

< LR >
AUC: 0.9999457847655191 Accuracy : 0.9963503649635036 Precision : 1.0 Recall : 0.9915966386554622

```

直接对评估结果对象 lr\_metrics 进行打印，会显示几个常用的指标，并输出混淆矩阵。代码如下：

---

```
System.out.println(lr_metrics);
```

---

运行结果如下。其中，横向为 Real（测试数据中真实的标签值），纵向为 Pred（预测结果的标签值），标签值为 1 和 0，中间交叉的部分即混淆矩阵。

Metrics:		
{Accuracy: 0.9964, LogLoss: 0.0089, Precision: 1, Recall: 0.9916, F1: 0.9958, Auc: 0.9999}		
Pred\Real	1	0
----- ----- -----		
1 118  0		
0  1 155		

二分类评估中的各种评估曲线也是非常重要的。Alink 提供了将曲线保存为图片的方法，相关代码如下：

---

```

lr_metrics.saveRocCurveAsImage(DATA_DIR + "lr_roc.jpg", true);
lr_metrics.saveRecallPrecisionCurveAsImage(DATA_DIR + "lr_recallprec.jpg", true);
lr_metrics.saveLiftChartAsImage(DATA_DIR + "lr_lift.jpg", true);
lr_metrics.saveKSAsImage(DATA_DIR + "lr_ks.jpg", true);

```

---

运行成功后，可以在指定的路径下，看到评估曲线图片。比如，ROC 曲线图像文件打开后，如图 8-7 所示。

至此，我们展示了二分类评估组件的常用功能。在实际使用中，我们除了使用 collectMetrics 方法直接触发任务执行、获取评估指标，也可以采用 Lazy 方式，将评估组件与任务中的其他组件一起执行，提高效率。参考代码如下。以评估 SVM 预测结果为例，其测试集采用了 AK 数据文件格式，保存在 DATA\_DIR + SVM\_PRED\_FILE 中。首先建立预测结果数据源，然后连接二分类评估组件 EvalBinaryClassBatchOp，并设置标签列名称和预测详情列名称。使用 lazyPrintMetrics 方法打印二分类评估常用的指标及混淆矩阵；再使用 lazyCollectMetrics 方法，设置通过 Java 的 Consumer 接口来异步接收处理 BinaryClassMetrics 的返回结果。这里略去了打

印输出评估指标的代码。

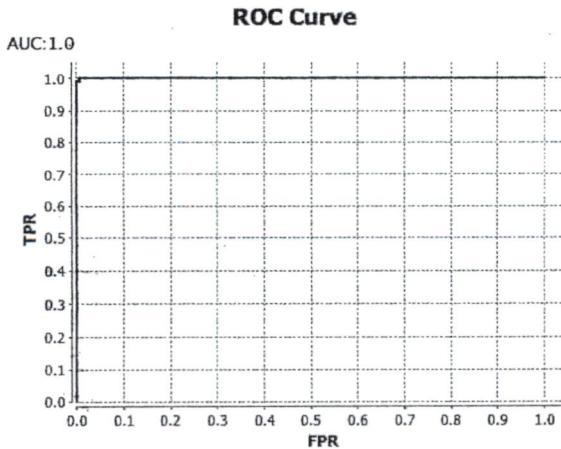


图 8-7

```

new AkSourceBatchOp()
.setFilePath(DATA_DIR + SVM_PRED_FILE)
.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("1")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics()
        .lazyCollectMetrics(new Consumer<BinaryClassMetrics>() {
            @Override
            public void accept(BinaryClassMetrics binaryClassMetrics) {
                try {
                    binaryClassMetrics.saveRocCurveAsImage(
                        DATA_DIR + "svm_roc.jpg", true);
                    binaryClassMetrics.saveRecallPrecisionCurveAsImage(
                        DATA_DIR + "svm_recallprec.jpg", true);
                    binaryClassMetrics.saveLiftChartAsImage(
                        DATA_DIR + "svm_lift.jpg", true);
                    binaryClassMetrics.saveKSAsImage(
                        DATA_DIR + "svm_ks.jpg", true);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        })
);
}
);

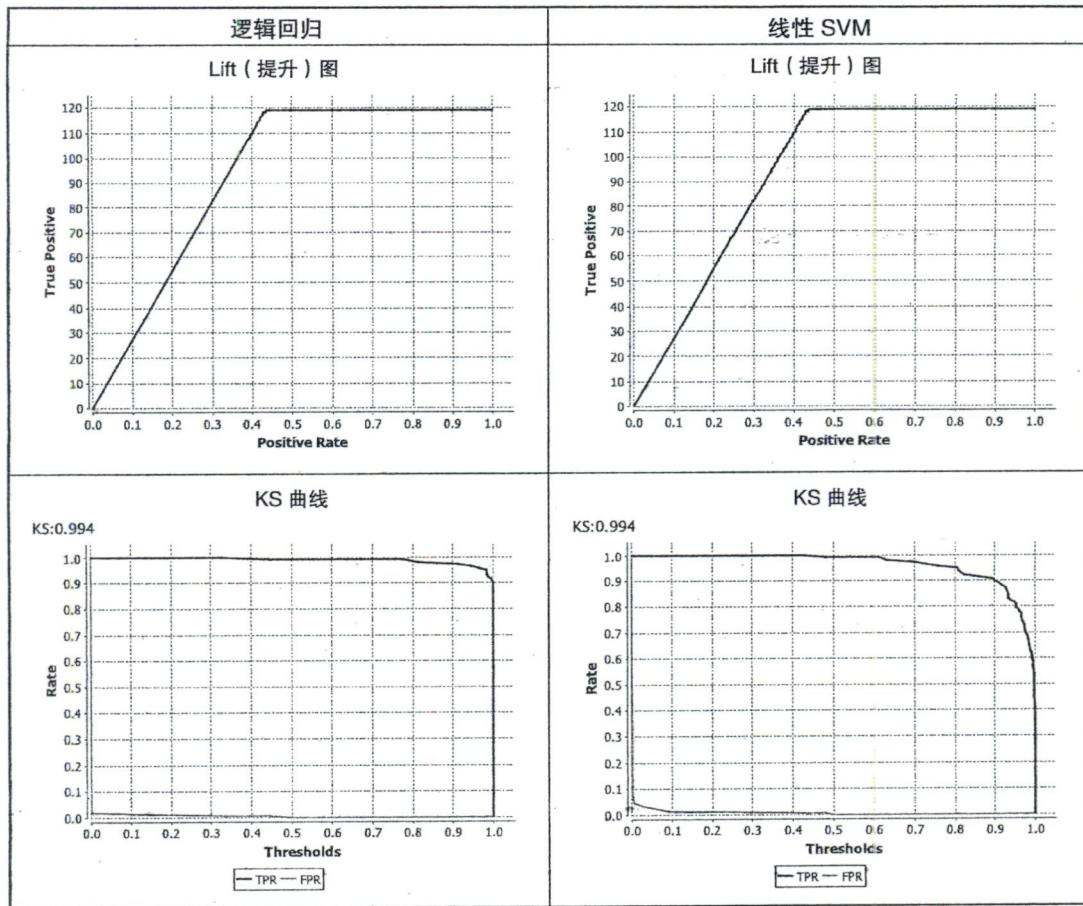
```

前面，我们针对逻辑回归和线性 SVM 的预测结果，分别进行了二分类评估。表 8-7 将输出的各种指标及曲线汇总起来，进行对比分析。

表 8-7 逻辑回归与线性 SVM 的二分类评估指标对比

逻辑回归	线性 SVM																		
AUC:0.9999 Accuracy:0.9964 Precision:1 Recall:0.9916 F1:0.9958 LogLoss:0.0089	AUC:0.9999 Accuracy:0.9964 Precision:1 Recall:0.9916 F1:0.9958 LogLoss:0.0233																		
混淆矩阵： <table border="1"> <tr> <td>Pred\Real</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>118</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>155</td> </tr> </table>	Pred\Real	1	0	1	118	0	0	1	155	混淆矩阵： <table border="1"> <tr> <td>Pred\Real</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>118</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>155</td> </tr> </table>	Pred\Real	1	0	1	118	0	0	1	155
Pred\Real	1	0																	
1	118	0																	
0	1	155																	
Pred\Real	1	0																	
1	118	0																	
0	1	155																	
ROC 曲线 AUC:1.0 	ROC 曲线 AUC:1.0 																		
PR 曲线 PRC:1.0 	PR 曲线 PRC:1.0 																		

续表



逻辑回归和线性 SVM 算法对当前的问题都非常有效，其测试集中的数据都只有一个预测错误。在表 8-7 中显示的几个指标中，两者的数值都相同；在 ROC 曲线、PR 曲线和 Lift 图上，二者看不出差别；但在 KS 曲线上二者有些差别，逻辑回归算法略好。

## 8.8 特征的多项式扩展

前面介绍的线性模型形式如下：

$$y = w_0 + \sum_{i=1}^m w_i x_i$$

在此基础上，定义二阶多项式模型如下：

$$y = w_0 + \sum_{i=1}^m w_i x_i + \sum_{i=1}^m \sum_{j=i}^m w_{ij} x_i x_j$$

在实际应用中，我们可以将原始特征  $(x_1, x_2, \dots, x_m)$ ，通过特征间交叉相乘的方式加入二阶特征，扩展为  $(x_1, x_2, \dots, x_m, x_1^2, x_1 x_2, \dots, x_{m-1} x_m, x_m^2)$ ，从而将二阶多项式模型训练转化为关于新特征的线性模型训练。

基于这种想法，使用 Alink 的向量多项式扩展组件 VectorPolynomialExpand 可以实现特征的展开，设置多项式的阶数为 2，即参数 Degree=2。由于该组件需要输入向量形式的数据，因此在运行前需要使用向量组装组件 VectorAssembler，将各数值列合并为向量。具体的调用代码如下：

```
PipelineModel featureExpand = new Pipeline()
    .add(
        new VectorAssembler()
            .setSelectedCols(FEATURE_COL_NAMES)
            .setOutputCol(VEC_COL_NAME + "_0")
    )
    .add(
        new VectorPolynomialExpand()
            .setSelectedCol(VEC_COL_NAME + "_0")
            .setOutputCol(VEC_COL_NAME)
            .setDegree(2)
    )
    .fit(train_data);

train_data = featureExpand.transform(train_data);
test_data = featureExpand.transform(test_data);

train_data.lazyPrint(1);
```

运行结果如下：

variance	skewness	kurtosis	entropy	class	vec_0	vec
2.5328	7.5280	-0.4193	-2.6478	0	2.5328	7.528 -0.41929 -2.6478   2.5328 6.41507584 7.528 19.0669184
56.67078399999999	-0.41929	-1.061977712	-3.1564151199999997	0	1.061977712	0.1758041041 -2.6478 -6.70634784 -19.9326384
1.110196062	7.010844840000001					

每个特征的数值在打印输出时会保留小数点后 4 位，各数值列合并成的向量为 (2.5328 7.5280 -0.4193 -2.6478)；二阶扩展后得到的向量为 (2.5328 6.41507584 7.528 19.0669184

56.67078399999999 -0.41929 -1.061977712 -3.156415119999997 0.1758041041 -2.6478 -6.70634784  
 -19.9326384 1.110196062 7.010844840000001)，其中包含了初始的4个特征，再加上10个二阶项特征，共14个特征。

下面，我们对新特征向量试用线性SVM算法，看看效果如何。具体代码如下，使用Pipeline的机制简化代码，LinearSvm为Pipeline的Estimator（详细内容可参考2.4节）。

---

```
new LinearSvm()
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("LinearSVM")
    );
}
```

---

模型评估结果如下：

```
LinearSVM
----- Metrics: -----
Auc:1 Accuracy:1 Precision:1 Recall:1 F1:1 LogLoss:0.0066
|Pred\Real| 1| 0|
|-----|---|---|
| 1|119| 0|
| 0| 0|155|
```

测试集的数据均被正确分类！特征的扩展直接提升了模型分类效果。

再看另外一个线性模型——逻辑回归模型，具体代码如下：

---

```
new LogisticRegression()
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("LogisticRegression")
    );
}
```

---

运行结果如下：

```
LogisticRegression
Metrics:
Auc:0.9967 Accuracy:0.9964 Precision:0.9917   Recall:1   F1:0.9958  LogLoss:0.1261
|Pred\Real| 1| 0|
|-----|---|---|
| 1|119| 1|
| 0| 0|154|
```

这里有一个数据被错分类，其分类效果比线性 SVM 模型稍差。

下面做一个实验，调整一下 LogisticRegression 组件的优化方法选择参数 OptimMethod。逻辑回归算法默认使用 OptimMethod.LBFGS 优化方法，这是一种在内存使用、收敛速度等方面综合表现最优的算法；而 OptimMethod.Newton 方法在收敛性方面表现得更好，只是其内存使用量为特征数的平方量级，不适合特征数很多的场景。这次的特征数很少，只有 14 个，因此可以尝试采用 OptimMethod.Newton 方法，具体代码如下，

```
new LogisticRegression()
    .setOptimMethod(OptimMethod.Newton)
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("LogisticRegression + OptimMethod.Newton")
    );
};
```

运行结果如下，这种方法也做到了完全分类：

```
LogisticRegression + OptimMethod.Newton
Metrics:
Auc:1 Accuracy:1 Precision:1   Recall:1   F1:1 LogLoss:0
|Pred\Real| 1| 0|
|-----|---|---|
| 1|119| 0|
| 0| 0|155|
```

从本节内容可以看到，特征扩展直接提升了分类器的效果。在后面的章节中，我们还会陆续介绍特征工程方面的方法。

## 8.9 因子分解机

前面介绍的二阶多项式模型如下：

$$y = w_0 + \sum_{i=1}^m w_i x_i + \sum_{i=1}^m \sum_{j=i}^m w_{ij} x_i x_j$$

二阶多项式模型比线性模型具有更强的分类能力。可以将原始特征  $(x_1, x_2, \dots, x_m)$ ，加入二阶特征，扩展为  $(x_1, x_2, \dots, x_m, x_1^2, x_1 x_2, \dots, x_{m-1} x_m, x_m^2)$ 。

二阶多项式模型中参数的个数为特征数的平方级。当特征数较多的时候，比如有 10 万个特征，则模型参数就要到百亿量级。利用因子分解机（Factorization Machine, FM）算法，可在模型效果和模型规模间取得一个很好的平衡。FM 模型的公式如下：

$$y = w_0 + \sum_{i=1}^m w_i x_i + \sum_{i=1}^{m-1} \sum_{j=i+1}^m \langle v^{(i)}, v^{(j)} \rangle x_i x_j$$

即，每个特征  $x_i$  对应一个向量  $v^{(i)}$ ，向量的维度为  $k$ ，使用向量内积  $\langle v^{(i)}, v^{(j)} \rangle$  替代参数  $w_{ij}$ 。实际应用中对于千万级特征，向量的维度可以取 100 维左右。FM 模型参数的个数约等于特征数乘以向量的维度。

注意：为了降低计算复杂度，FM 模型公式在实际计算时会采用另外一种形式。在 FM 模型公式定义中，计算量集中在交叉项：

$$\sum_{i=1}^{m-1} \sum_{j=i+1}^m \langle v^{(i)}, v^{(j)} \rangle x_i x_j$$

其计算复杂度为  $O(km^2)$ ，将此交叉项进行等式变换：

$$\begin{aligned} & \sum_{i=1}^{m-1} \sum_{j=i+1}^m \langle v^{(i)}, v^{(j)} \rangle x_i x_j \\ &= \sum_{i=1}^{m-1} \sum_{j=i+1}^m \langle x_i \cdot v^{(i)}, x_j \cdot v^{(j)} \rangle \end{aligned}$$

$$\begin{aligned}
 &= \frac{1}{2} \left( \left\langle \sum_{i=1}^m x_i \cdot v^{(i)}, \sum_{j=1}^m x_j \cdot v^{(j)} \right\rangle - \sum_{i=1}^m \left\langle x_i \cdot v^{(i)}, x_i \cdot v^{(i)} \right\rangle \right) \\
 &= \frac{1}{2} \left( \left\| \sum_{i=1}^m x_i \cdot v^{(i)} \right\|_2^2 - \sum_{i=1}^m \|x_i \cdot v^{(i)}\|_2^2 \right)
 \end{aligned}$$

可知，变换后表达式的计算复杂度为  $O(km)$ 。

下面将此算法应用到本章的数据，代码如下。设置向量的维度为 2，即参数 NumFactor=2；参数 NumEpochs 表示训练几遍数据，这里设置 NumEpochs=10；并设置学习率 LearnRate=0.5。参数 NumEpochs 和 LearnRate 是需要根据具体数据进行调节的。我们使用了 enableLazyPrintTrainInfo 方法来打印输出训练过程的数据：

```

new FmClassifier()
    .setNumEpochs(10)
    .setLearnRate(0.5)
    .setNumFactor(2)
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .enableLazyPrintTrainInfo()
    .enableLazyPrintModelInfo()
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("FM"))
);

```

训练过程的信息如下：

```

----- train meta info -----
{numFeature: 4, numFactor: 2, hasLinearItem: true, hasIntercept: true}
----- train convergence info -----
step: 0 loss: 0.08713016 auc: 0.99632587 accuracy: 0.96083789
step: 1 loss: 0.05184192 auc: 0.99943728 accuracy: 0.98633880
step: 2 loss: 0.04138949 auc: 1.00000000 accuracy: 0.99635701
...
step: 7 loss: 0.02687007 auc: 1.00000000 accuracy: 0.99726776
step: 8 loss: 0.02532989 auc: 1.00000000 accuracy: 1.00000000
step: 9 loss: 0.02396747 auc: 1.00000000 accuracy: 1.00000000

```

我们可以看到经过每一个 Epoch 后主要指标 loss、auc 和 accuracy 的变化情况。参考这些信息有助于我们调整参数 NumEpochs 和 LearnRate。

模型信息如下。其中，bias 为模型的 0 阶截距 (Intercept) 系数，每个特征都对应一个一阶系数 (linearItem) 和一个向量系数。

```
----- model meta info -----
{numFeature: 4, numFactor: 2, bias: 1.8750569560538486, hasLinearItem: true, hasIntercept: true}
----- model label values -----
[1, 0]
----- model info -----
| colName| linearItem | factor |
| --- | --- | --- |
| variance| -2.22505520 | 0.34821299 0.31662868 |
| skewness| -0.93356183 | 0.10569094 0.14378680 |
| kurtosis| -1.53638153 | -0.05461301 -0.23339019 |
| entropy| 0.11982146 | 0.22809465 0.04826799 |
```

二分类评估结果如下，测试集实现了全部正确分类：

```
----- Metrics: -----
Auc:1 Accuracy:1 Precision:1 Recall:1 F1:1 LogLoss:0.0173
|Pred\Real| 1 | 0 |
| --- | --- | --- |
| 1 | 119 | 0 |
| 0 | 0 | 155 |
```

一般来说，FM 的分类效果要优于逻辑回归和线性 SVM，但在使用 FM 时，需要在调节训练参数方面多花费一些时间。



# 9

## 朴素贝叶斯模型与决策树模型

前面介绍了线性模型，本章将重点介绍两种常用的非线性模型：朴素贝叶斯模型与决策树模型。

9.1 节和 9.2 节将分别对这两种算法模型进行介绍。从 9.3 节开始，以蘑菇是否有毒的分类为例，介绍如何使用模型、如何查看模型。

### 9.1 朴素贝叶斯模型

朴素贝叶斯（Naive Bayes）是基于“朴素”的假设（各特征属性相互独立）和贝叶斯定理的分类算法。它是一种简单、常用的分类算法，“朴素”的假设不但使计算量大幅减少，而且对分类效果的影响也不大。对于现实中的一些复杂场景，该算法也表现得很好。

设有 $k$ 个分类值 $\{c_1, c_2, \dots, c_k\}$ ，共有 $M$ 个特征属性 $a_1, a_2, \dots, a_M$ ，使用朴素贝叶斯算法预测一个待分类项，分为两步：

(1) 计算出该属性值对于各分类值的条件概率，即

$$P(c_1|a_1, a_2, \dots, a_M), P(c_2|a_1, a_2, \dots, a_M), \dots, P(c_k|a_1, a_2, \dots, a_M)$$

(2) 最大概率值所对应的分类值即预测结果。

可以看出计算核心是 $P(c_k|a_1, a_2, \dots, a_M)$ 。这里需要用到贝叶斯公式及朴素(Naive)的假设，下面将详细解释。

对于概率 $P(c_k|a_1, a_2, \dots, a_M)$ ，由贝叶斯公式，可得

$$P(c_k | a_1, a_2, \dots, a_M) = \frac{P(a_1, a_2, \dots, a_M | c_k) P(c_k)}{P(a_1, a_2, \dots, a_M)}$$

再利用“朴素”的假设（各个特征属性是相互独立的），则其联合概率可以表示为更简单的，也更容易计算的形式：

$$P(a_1, a_2, \dots, a_M | c_k) = P(a_1 | c_k) P(a_2 | c_k) \cdots P(a_{M-1} | c_k) P(a_M | c_k)$$

所以，

$$P(c_k | a_1, a_2, \dots, a_M) = \frac{P(a_1 | c_k) P(a_2 | c_k) \cdots P(a_{M-1} | c_k) P(a_M | c_k) P(c_k)}{P(a_1, a_2, \dots, a_M)}$$

我们要比较的k个概率：

$$P(c_1 | a_1, a_2, \dots, a_M), P(c_2 | a_1, a_2, \dots, a_M), \dots, P(c_k | a_1, a_2, \dots, a_M)$$

都是以 $P(a_1, a_2, \dots, a_M)$ 为分母的，所以 $P(a_1, a_2, \dots, a_M)$ 的具体值对k个概率的大小顺序是没有影响的，即

$$P(c_k | a_1, a_2, \dots, a_M) \propto P(a_1 | c_k) P(a_2 | c_k) \cdots P(a_{M-1} | c_k) P(a_M | c_k) P(c_k)$$

因此，我们只要计算 $P(a_i | c_k)$ 与 $P(c_k)$ ，带入上面的公式，就可知道哪个分类的概率最大。我们从训练数据中很容易计算出

$$P(c_k) = \frac{\text{分类为 } c_k \text{ 的训练样本数}}{\text{总训练样本数}}$$

在条件概率 $P(a_i | c_k)$ 的计算中，将训练数据按其分类值分为k个集合，对每个子集计算各个特征属性的概率分布。特征属性 $a_i$ 为连续特征或者离散特征时，计算方法是不一样的，分别介绍如下：

(1) 若特征属性 $a_i$ 为连续特征，则在所有分类为 $c_k$ 的训练样本上，计算其均值 $\mu_{ik}$ 与标准差 $\sigma_{ik}$ ，并假设 $a_i$ 服从正态分布的，则概率密度如下：

$$P(a_i | c_k) = \frac{1}{\sigma_{ik} \sqrt{2\pi}} \exp\left(-\frac{(a_i - \mu_{ik})^2}{2\sigma_{ik}^2}\right)$$

(2) 若特征属性 $a_i$ 为离散特征，共有 $s_i$ 个离散值 $a_i^{(1)}, a_i^{(2)}, \dots, a_i^{(s_i)}$ ，则关于分类值 $c_k$ 的条件概率的极大似然估计如下：

$$P(a_i^{(j)}|c_k) = \frac{\text{在分类为 } c_k \text{ 的训练样本中, } a_i = a_i^{(j)} \text{ 的样本数}}{\text{分类为 } c_k \text{ 的训练样本数}}$$

尽管这个表达式很容易理解，但是在实际使用中却会遇到问题。因为在某个分类所对应的数据集中，有的属性的属性值可能不出现。即，可能存在  $c_k$  与  $a_i^{(j)}$ ，使得：

在分类为  $c_k$  的训练样本中， $a_i = a_i^{(j)}$  的样本数为 0

则对应的

$$P(a_i^{(j)}|c_k) = 0$$

当使用模型预测新的数据时，如果正好遇到预测的数据的属性值为  $a_i^{(j)}$ ，则因为该项的概率为 0，会使整体的概率为 0。这样无论其他属性的概率有多高，都对分类结果没有影响。为了避免出现这种情况，我们希望给  $P(a_i^{(j)}|c_k)$  赋个很小的概率。这样既体现了该属性的作用，又可以整体用到其他属性的概率信息。

下面使用离散数据的平滑技术：加法平滑 [ Additive Smoothing，或称拉普拉斯平滑 (Laplace Smoothing) ] 来解决这个问题，即，定义平滑参数  $\lambda \geq 0$ ，条件概率如下：

$$P(a_i^{(j)}|c_k) = \frac{\text{在分类为 } c_k \text{ 的训练样本中, } a_i = a_i^{(j)} \text{ 的样本数} + \lambda}{\text{分类为 } c_k \text{ 的训练样本数} + s_i \cdot \lambda}$$

常取参数  $\lambda = 1$ 。在表达式的分子中加  $\lambda$ ，分母中加  $s_i \cdot \lambda$ ，这样可保证条件概率的和为 1，即

$$\sum_{j=1}^{s_i} P(a_i^{(j)}|c_k) = 1$$

在实际应用中，遍历计算所有离散值关于分类值  $c_k$  的条件概率，并保存到模型中。在预测新的数据时，对于其属性值  $a_i$ ，可以通过查表的方式找到  $P(a_i|c_k)$ 。

## 9.2 决策树模型

决策树模型再现了人们做决策的过程，该过程由一系列的判断构成，后面的判断基于前面判断的结果，不断缩小范围，最终推出结果。我们先看一个使用决策树模型的简单示例。通过

天气状态判断某一天是否适合打高尔夫球，决策的过程如下。首先看天气的整体情况：晴天？阴天？下雨？如果是阴天的话，可以直接给出结论。对于晴天的情况，需要根据湿度情况进行进一步判断；对于雨天的情况，需要根据是否刮风，给出最终的结论。整个决策过程如图 9-1 所示，用树状结构可以形象地展示。

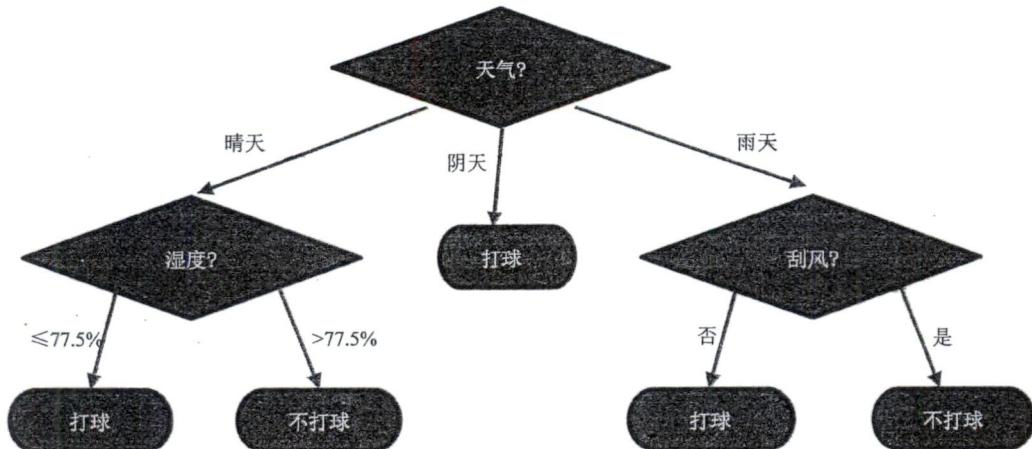


图 9-1

在此可以看到：

- 树的叶节点，给出结论。
- 树的枝节点，做出判断。
  - 作用：产生子节点（可能是枝节点或叶节点），即根据本轮的判断，引申出进一步的判断或得出结论。
  - 方式：针对某个特征（分裂特征），看特征值属于哪种划分方式（分裂方式）。比如，湿度按 $\leq 77.5\%$ 与 $>77.5\%$ 进行划分；天气按晴天、阴天和雨天进行划分。

在上述例子中，是否适合打高尔夫球的场景，就对应着 Golf 数据集，如表 9-1 所示。表 9-1 记录了天气状态 [天气的整体情况 (Outlook)、温度 (Temperature, 本例采用的是华氏度)、湿度 (Humidity)、是否刮风 (Windy) ] 及当天是否适合打高尔夫球。注意：本示例 Golf 数据集中的湿度百分比号省略。

表 9-1 Golf 数据集

序号	Outlook	Temperature	Humidity	Windy	Play
1	sunny	85	85	false	no
2	sunny	80	90	true	no
3	overcast	83	78	false	yes

续表

序号	Outlook	Temperature	Humidity	Windy	Play
4	rainy	70	96	false	yes
5	rainy	68	80	false	yes
6	rainy	65	70	true	no
7	overcast	64	65	true	yes
8	sunny	72	95	false	no
9	sunny	69	70	false	yes
10	rainy	75	80	false	yes
11	sunny	75	70	true	yes
12	overcast	72	90	true	yes
13	overcast	81	75	false	yes
14	rainy	71	80	true	no

决策树的构建可以被看作一个递归的过程，具体过程如下所示，先确定根节点的分裂特征和分裂方式，然后从其最左边的节点开始：

(1) 设节点的数据集为  $D$ ，

- 如果所有数据都属于同一个类别  $C_j$ ，则将该节点作为叶节点；
- 如果满足某种条件，该数据集不能再划分，则将  $D$  中实例数最大的类  $C_j$  作为该节点的类别；
- 否则，考虑所有特征及其可能的划分方式。找到使分裂指标最优的划分方式，定为该节点的分裂特征和分裂方式。此节点作为枝节点，每个分裂方式对应一个子节点  $S_i$ ，并按此分裂特征进行判断，每个子节点  $S_i$  均对应一个非空子集  $D_i$ 。

(2) 对于子节点  $S_i$ （相应的数据集  $D_i$ ）递归调用(1)。

(3) 汇总所有计算出的枝节点和叶节点，生成决策树。

对于 Golf 数据集，我们可以得到如图 9-2 所示的决策树，每个节点的 Instances 项为属于该节点的样本个数，并表明了标签值为 yes 或 no 的样本数量和比例。

注意：图 9-2 由 Alink 决策树组件生成，在 9.2.5 节中有相关的具体示例。

在实际操作中还要考虑其他因素，以控制树的生长，常用下面一些判断：

(1) 当决策树达到一定的高度时，就让决策树停止生长。

(2) 到达某节点的训练样本具有相同的特征属性，即使这些样本不属于同一类，也可以让决策树停止生长。

- (3) 当某节点的训练样本个数小于叶节点最低样本数的时候，也可以让决策树停止生长。  
(4) 如果分裂指标的增量小于某个阈值，也可以让决策树停止生长。

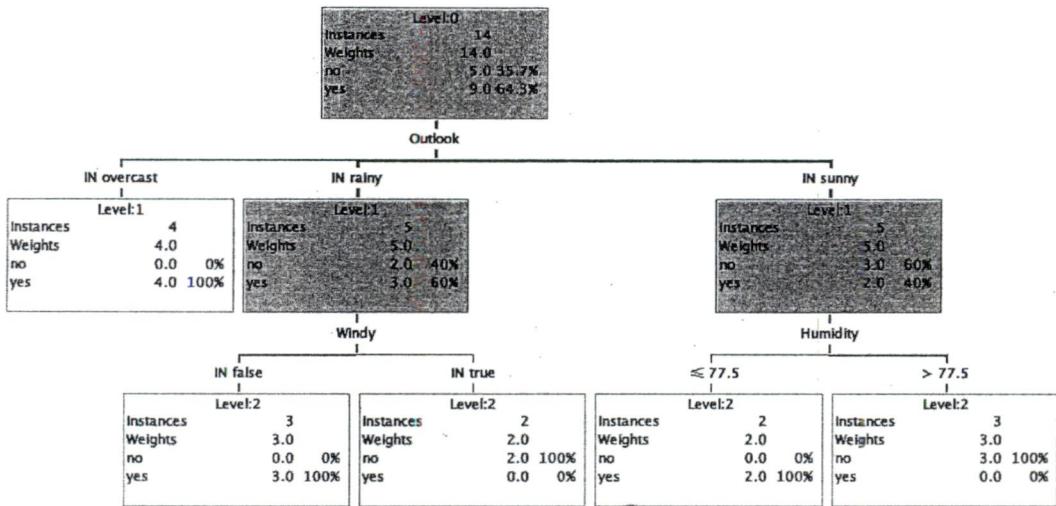


图 9-2

### 9.2.1 决策树的分裂指标定义

对于离散随机变量 $X$ ，有 $k$ 个状态 $\{x_1, x_2, \dots, x_k\}$ ，其每个状态对应的概率为 $\{p_1, p_2, \dots, p_k\}$ ，则 $X$ 的信息熵定义如下：

$$H(X) = - \sum_{i=1}^k p_i \log_a(p_i)$$

其中 $a$ 为对数的底，当 $a = 2$ 时，信息熵的单位为比特（bit）；当 $a = e$ 时，信息熵的单位为奈特（nat）。由定义可以看出，离散随机变量 $X$ 的信息熵为“信息量的加权平均”。

设 $Y$ 有 $m$ 个取值 $\{y_1, y_2, \dots, y_m\}$ 。条件熵表示在已知随机变量 $X$ 的条件下随机变量 $Y$ 的不确定性，其定义为在 $X$ 给定的条件下， $Y$ 的条件概率分布对 $X$ 的数学期望：

$$H(Y|X) = \sum_{i=1}^k P(x_i) H(Y|X = x_i)$$

设训练集为 $C$ ，用 $|\cdot|$ 表示集合的元素个数，则 $|C|$ 为样本容量，即样本个数。

设训练数据共有 $k$ 个分类，按分类划分为 $k$ 个样本集合 $\{C_1, C_2, \dots, C_k\}$ ，满足：

$$\begin{cases} C_i \cap C_j = \emptyset, \forall i \neq j \\ C = C_1 \cup C_2 \cup \dots \cup C_k \end{cases}$$

对于各分类的样本个数 $|C_i|$ , 有

$$|C| = \sum_{i=1}^k |C_i|$$

对于特征 $A$ , 按照某种规则, 可将 $C$ 划分为 $m$ 个子集 $\{A_1, A_2, \dots, A_m\}$ , 满足:

$$\begin{cases} A_i \cap A_j = \emptyset, \forall i \neq j \\ C = A_1 \cup A_2 \cup \dots \cup A_m \end{cases}$$

对于各子集的样本个数 $|A_i|$ , 有

$$|C| = \sum_{i=1}^m |A_i|$$

信息增益(Information Gain, IG)表示得知特征 $A$ 的信息而使分类 $C$ 的信息的熵减少的程度。

$$IG(C, A) = H(C) - H(C|A)$$

由前面介绍的内容可知, 信息增益也是特征 $A$ 与分类 $C$ 的互信息。

$$IG(C, A) = I(C; A) = H(C) + H(A) - H(C, A)$$

各类别的子集为 $\{C_1, C_2, \dots, C_k\}$ , 相应的概率为 $\{\frac{|C_1|}{|C|}, \frac{|C_2|}{|C|}, \dots, \frac{|C_k|}{|C|}\}$ 。由信息熵和条件熵的定义, 数据集 $C$ 的信息熵如下:

$$H(C) = - \sum_{i=1}^k \frac{|C_i|}{|C|} \ln \frac{|C_i|}{|C|}$$

根据特征 $A$ 的取值, 将数据集划分为 $m$ 个子集 $\{A_1, A_2, \dots, A_k\}$ , 则特征 $A$ 对于数据集 $C$ 的条件熵如下:

$$H(C|A) = - \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|}$$

所以, 信息增益

$$\text{IG}(C, A) = H(C) - H(C|A) = \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|} - \sum_{i=1}^k \frac{|C_i|}{|C|} \ln \frac{|C_i|}{|C|}$$

信息增益率 (Information Gain Ratio, IGR) 为信息增益  $\text{IG}(C, A)$  与特征  $A$  的信息熵的比值, 即

$$\text{IGR}(C, A) = \frac{\text{IG}(C, A)}{H(A)}$$

这里, 假设数据集  $C$  中属于各类别的子集为  $\{C_1, C_2, \dots, C_k\}$ , 并根据特征  $A$  的值将数据集划分为  $m$  个子集  $\{A_1, A_2, \dots, A_k\}$ , 则  $H(A)$  和  $\text{IG}(C, A)$  的计算公式如下:

$$H(A) = - \sum_{i=1}^k \frac{|A_i|}{|C|} \ln \frac{|A_i|}{|C|}$$

$$\text{IG}(C, A) = \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|} - \sum_{i=1}^k \frac{|C_i|}{|C|} \ln \frac{|C_i|}{|C|}$$

样本集合  $C$  的基尼指数 (Gini Index, 或称基尼系数) 与其分类情况相关:

$$\text{Gini}(C) = 1 - \sum_{i=1}^k \left( \frac{|C_i|}{|C|} \right)^2$$

根据属性特征  $A$ , 若样本集合  $C$  被划分为两个非空子集  $A_1$  和  $A_2$ , 则

$$\text{Gini}(C, A) = \frac{|A_1|}{|C|} \text{Gini}(A_1) + \frac{|A_2|}{|C|} \text{Gini}(A_2)$$

其中,

$$\text{Gini}(A_j) = 1 - \sum_{i=1}^k \left( \frac{|A_j \cap C_i|}{|A_j|} \right)^2$$

## 9.2.2 常用的决策树算法

常用的决策树算法包括 ID3、C4.5 和 CART, 各算法的简介如表 9-2 所示。

表 9-2 常用的决策树算法简介

算法	简介	作者	年份
ID3	Iterative Dichotomiser 3 的缩写	Ross Quinlan	1986 年

续表

算法	简介	作者	年份
C4.5	对 ID3 算法的改进	Ross Quinlan	1993 年
CART	Classification And Regression Tree 的缩写	L. Breiman, J. Friedman, R. Olshen, C. Stone	1984 年

决策树对于离散特征和连续特征，处理方法有些差异，下面将分别讨论。

### 1. 离散特征

对于离散特征，直接套用公式，计算分裂指标即可，参见表 9-3。

表 9-3 离散特征的决策树分裂指标

算法	分裂指标
ID3	<p>使用信息增益 (Information Gain, IG)</p> $IG(C, A^{(i)}) = H(C) - H(C A^{(i)})$ <p>选择使 <math>IG(C, A^{(i)})</math> 最大的特征属性 <math>A^{(i)}</math></p> <p>划分方法：将枚举特征属性 <math>A^{(i)}</math> 的每一个取值 <math>a_j^{(i)}</math>，分别作为一个划分子集</p>
C4.5	<p>使用信息增益率 (Information Gain Ratio, IGR)</p> $IGR(C, A^{(i)}) = \frac{IG(C, A^{(i)})}{H(A^{(i)})}$ <p>选择使 <math>IGR(C, A^{(i)})</math> 最大的特征属性 <math>A^{(i)}</math></p> <p>划分方法：将枚举特征属性 <math>A^{(i)}</math> 的每一个取值 <math>a_j^{(i)}</math>，分别作为一个划分子集</p>
CART	<p>使用基尼指数 (Gini Index)</p> <p>需要选择特征属性 <math>A^{(i)}</math> 是否取某一个可能值 <math>a_j^{(i)}</math>，将数据分割成 <math>A_1^{(i)}</math> 和 <math>A_2^{(i)}</math> 两部分，并计算</p> $Gini(C, A^{(i)} = a_j^{(i)})$ <p>考虑基尼增量：</p> $\Delta Gini(C) = Gini(C) - Gini(C, A^{(i)} = a_j^{(i)})$ <p>目标是使基尼增量最大，即选择使基尼指数 <math>Gini(C, A^{(i)} = a_j^{(i)})</math> 最小的属性特征 <math>A^{(i)}</math>。每次将是否等于 <math>a_j^{(i)}</math> 作为划分条件，得到两个划分子集。</p>

### 2. 连续特征

处理连续特征的方法如下：确定可能的阈值集合；每个阈值点，可将数据集划分为 2 个部分；计算每个划分的指标（信息增益、信息增益率，或者基尼指数）；找到使指标达到最优的

阈值点，并将该阈值点作为分割阈值点。连续特征的决策树分裂指标如表 9-4 所示。

计算分割阈值点的具体步骤如下：

(1) 确定可能的阈值集合。

(a) 对特征的取值按升序进行排序。

(b) 将两个特征取值之间的中点作为可能的分裂点。（如果有  $n$  条训练样本，那么可能有  $n - 1$  个阈值，过多的阈值会对应大量的指标计算。我们可以通过判断两个相邻特征取值所对应的样本的分类信息是否相同来减少计算次数。如果该分类信息相同，则该点不适合作为分割点，可以略过，这样能显著减少阈值的数量。）

(2) 通过计算分类指标，确定指标最优的阈值，并将其作为分割阈值点。

(a) 对于阈值  $a_j$ ，则  $(-\infty, a_j]$  和  $(a_j, +\infty)$  就是该特征的划分子集，计算分类指标。

(b) 找出指标最优的阈值，并将其作为分割阈值点。

表 9-4 连续特征的决策树分裂指标

算法	分裂指标
ID3	使用信息增益 (Information Gain, IG) $IG(C, A^{(i)}) = H(C) - H(C A^{(i)})$ 选择使 $IG(C, A^{(i)})$ 最大的特征属性 $A^{(i)}$ 以及阈值分割点 $a^{(i)}$
C4.5	使用信息增益率 (Information Gain Ratio, IGR) $IGR(C, A^{(i)}) = \frac{IG(C, A^{(i)})}{H(A^{(i)})}$ 选择使 $IGR(C, A^{(i)})$ 最大的特征属性 $A^{(i)}$ 以及阈值分割点 $a^{(i)}$
CART	使用基尼指数 (Gini Index) $Gini(C, A^{(i)})$ 考虑基尼增量： $\Delta Gini(C) = Gini(C) - Gini(C, A^{(i)} = a_j^{(i)})$ 目标是使基尼增量最大，即选择基尼指数 $Gini(C, A^{(i)} = a_j^{(i)})$ 最小的属性特征 $A^{(i)}$ 以及阈值分割点 $a^{(i)}$

### 9.2.3 指标计算示例

我们以 Golf 数据集（见表 9-1）为例，目标是根据某一天的天气状态（天气的整体情况、

温度、湿度、是否刮风) 来判断这一天是否适合打高尔夫球。

分类  $C$  共有两个分类值 yes 和 no, 对应的分类子集为  $C_1$  和  $C_2$ , 其统计值如表 9-5 所示。

表 9-5 分类值的统计

	个数
yes	9
no	5
总数	14

计算数据集的信息熵如下:

$$H(C) = - \sum_{i=1}^k \frac{|C_i|}{|C|} \ln \frac{|C_i|}{|C|} = - \left( \frac{9}{14} \ln \frac{9}{14} + \frac{5}{14} \ln \frac{5}{14} \right) = 0.6517565611726531$$

计算 Outlook 属性的指标, 该属性的统计值如表 9-6 所示。

表 9-6 Outlook 属性的统计值

	sunny	overcast	rainy
yes	2	4	3
no	3	0	2
总数	5	4	5

$$H(A_{\text{Windy}}) = - \sum_{i=1}^k \frac{|A_i|}{|C|} \ln \frac{|A_i|}{|C|} = 1.0933747175566468$$

$$H(C|A_{\text{Outlook}}) = - \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|} = 0.48072261929232607$$

所以,

$$\text{IG}(C, A_{\text{Outlook}}) = H(C) - H(C|A_{\text{Outlook}}) = 0.17103394188032706$$

$$\text{IGR}(C, A_{\text{Outlook}}) = \frac{\text{IG}(C, A_{\text{Outlook}})}{H(A_{\text{Outlook}})} = 0.1564275624211751$$

与计算 Outlook 属性类似, 可以计算出 Windy 属性, 该属性的统计值如表 9-7 所示。

表 9-7 Windy 属性的统计值

	true	false
yes	3	6
no	3	2
总数	6	8

$$H(A_{\text{Windy}}) = - \sum_{i=1}^k \frac{|A_i|}{|C|} \ln \frac{|A_i|}{|C|} = 0.6829081047004717$$

$$H(C|A_{\text{Windy}}) = - \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|} = 0.6183974457364384$$

所以，

$$\text{IG}(C, A_{\text{Windy}}) = H(C) - H(C|A_{\text{Windy}}) = 0.033359115436214726$$

$$\text{IGR}(C, A_{\text{Windy}}) = \frac{\text{IG}(C, A_{\text{Windy}})}{H(A_{\text{Windy}})} = 0.04884861551152079$$

显然，在信息增益（IG）和信息增益率（IGR）指标上，Outlook 属性均优于 Windy 属性。我们再计算一下连续值属性 Temperature 的指标。

按温度高低对样本进行排序。我们只关注温度与分类列，排序后的结果如下：

64	65	68	69	70	71	72	72	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no

下面我们先找可能的阈值点。我们可以通过下面的两个判断准则来减少候选阈值点的数量：

- (1) 阈值点一定位于两个不同的数值之间。比如，阈值点可以在 64 和 65 中间。
- (2) 对于相邻的不重复的数值，如果它们的分类值一样，则其中点不是候选阈值点。

于是，我们得到候选阈值点的分割显示如下：

64	65	68	69	70	71	72	72	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no

候选阈值点集合：{64.5, 66.5, 70.5, 71.5, 73.5, 77.5, 80.5, 84}。

以候选阈值点 71.5 为例，计算分类指标，该阈值的统计值如表 9-8 所示。

表 9-8 阈值点 71.5 的统计值

	$\leq 71.5$	$> 71.5$
yes	4	5
no	2	3
总数	6	8

$$H(A_{\text{Temperature}} = 71.5) = - \sum_{i=1}^k \frac{|A_i|}{|C|} \ln \frac{|A_i|}{|C|} = 0.6829081047004717$$

$$H(C|A_{\text{Temperature}} = 71.5) = - \sum_{i=1}^m \frac{|A_i|}{|C|} \sum_{j=1}^k \frac{|A_i \cap C_j|}{|A_i|} \ln \frac{|A_i \cap C_j|}{|A_i|} = 0.6508279225023381$$

计算出各个阈值点对应的指标如表 9-9 所示。

表 9-9 各个阈值点对应的指标

$\theta$	$H(A_{\text{Temperature}} = \theta)$	$H(C A_{\text{Temperature}} = \theta)$	$IG(C, A_{\text{Temperature}} = \theta)$	$IGR(C, A_{\text{Temperature}} = \theta)$
64.5	0.25731864054383163	0.618687125099342	0.03306943607331114	0.12851550903354822
66.5	0.410116318288409	0.6446045986184031	0.007151962554249991	0.01743886364750906
70.5	0.6517565611726531	0.6203333076476321	0.031423253525021067	0.04821317558887898
71.5	0.6829081047004717	0.6508279225023381	9.286386703150074E-4	0.0013598296226434664
73.5	0.6829081047004717	0.6508279225023381	9.286386703150074E-4	0.0013598296226434664
77.5	0.5982695885852573	0.6343736959134797	0.01738286525917343	0.029055237957655706
80.5	0.5195798391305154	0.651417286985697	3.392741869561178E-4	6.529779668200214E-4
84	0.25731864054383163	0.5731530718924601	0.078603489280193	0.3054714151841778

注意：对于阈值为 64.5 和 84 的情况（表 9-9 中背景为灰色的区域），在划分的集合中，其中一个划分子集只有一个训练样本。考虑到划分后要满足每个叶节点的样本数大于 1，所以这两种阈值的情况不予考虑。

#### 9.2.4 分类树与回归树

决策树分为两大类：分类树、回归树，其可分别解决机器学习领域中的分类和回归问题。分类树预测分类标签值，回归树用于预测数值。

分类树在每次分枝时，会针对当前枝节点的样本，找出最优的分裂特征和分裂方式，从而得到若干新节点；继续分枝，直到所有样本都被分入类别唯一的叶节点，或者满足设定的终止

条件（比如，叶节点最低样本数的限制，或者决策树层数的限制）。若最终叶节点中的类别不唯一，则以多数样本的类别作为该叶节点的类别。

回归树的建模流程与分类树类似；回归树在分枝时的分裂指标与分类树不同，回归树一般以均方差为分裂指标；回归树的分枝操作很难做到每个叶节点上的数值都一样，在实际中，以最终叶节点上所有样本的平均数值作为该叶节点的预测数值。

### 9.2.5 经典的决策树示例

根据天气因素判断能否打高尔夫球是一个经典的决策树示例。该数据集只有 14 条数据，我们使用内存数据源组件 MemSourceBatchOp，可以直接在代码中定义数据记录 Row 类型的数组，并输入相应的列名称。构造函数代码如下。这里省略了中间的一些数据；最后一句为使用 Lazy 方式进行数据的打印输出，参数-1 表示打印全部数据。

```
MemSourceBatchOp source = new MemSourceBatchOp()
new Row[] {
    Row.of("sunny", 85.0, 85.0, false, "no"),
    ...
    Row.of("overcast", 81.0, 75.0, false, "yes"),
    Row.of("rainy", 71.0, 80.0, true, "no")
},
new String[] {"Outlook", "Temperature", "Humidity", "Windy", "Play"}
);
source.lazyPrint(-1);
```

打印输出数据，如表 9-10 所示。前面四列为特征列，最后一列为是否打高尔夫球的判断结果列，即分类问题的标签列。

表 9-10 Golf 数据集

Outlook	Temperature	Humidity	Windy	Play
sunny	85.0000	85.0000	false	no
sunny	80.0000	90.0000	true	no
overcast	83.0000	78.0000	false	yes
rainy	70.0000	96.0000	false	yes
rainy	68.0000	80.0000	false	yes
rainy	65.0000	70.0000	true	no
overcast	64.0000	65.0000	true	yes
sunny	72.0000	95.0000	false	no
sunny	69.0000	70.0000	false	yes

续表

Outlook	Temperature	Humidity	Windy	Play
rainy	75.0000	80.0000	false	yes
sunny	75.0000	70.0000	true	yes
overcast	72.0000	90.0000	true	yes
overcast	81.0000	75.0000	false	yes
rainy	71.0000	80.0000	true	no

下面，将此数据连接到 C45 决策树训练组件，设置特征数据列，设置其中的离散特征列，并设置标签列。这些参数设置与前面的朴素贝叶斯训练组件类似。决策树还有其他的训练参数，这里暂时使用其默认值。打印输出默认的模型信息。如果需要查看树结构的模型展示，则需要使用 `lazyCollectModelInfo` 方法来获取决策树模型信息。类 `DecisionTreeModelInfo` 中有 `saveTreeAsImage` 方法，其第一个参数为图片路径，第二个参数表示是否覆盖已有图片。

```
source
    .link(
        new C45TrainBatchOp()
            .setFeatureCols("Outlook", "Temperature", "Humidity", "Windy")
            .setCategoricalCols("Outlook", "Windy")
            .setLabelCol("Play")
            .lazyPrintModelInfo()
            .lazyCollectModelInfo(new Consumer<DecisionTreeModelInfo>() {
                @Override
                public void accept(DecisionTreeModelInfo decisionTreeModelInfo) {
                    try {
                        decisionTreeModelInfo.saveTreeAsImage(
                            DATA_DIR + "weather_tree_model.png", true);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            })
    );
});
```

打印输出的模型信息如下：

```
Classification trees modelInfo:
Number of trees: 1
Number of features: 4
Number of categorical features: 2
Labels: [no, yes]
```

```
Categorical feature info:
|feature|number of categorical value|
|-----|-----|
```

outlook	3
Windy	2

Table of feature importance:

feature	importance
Humidity	0.4637
Windy	0.4637
Outlook	0.0725
Temperature	0

决策树的结构如图 9-3 所示。

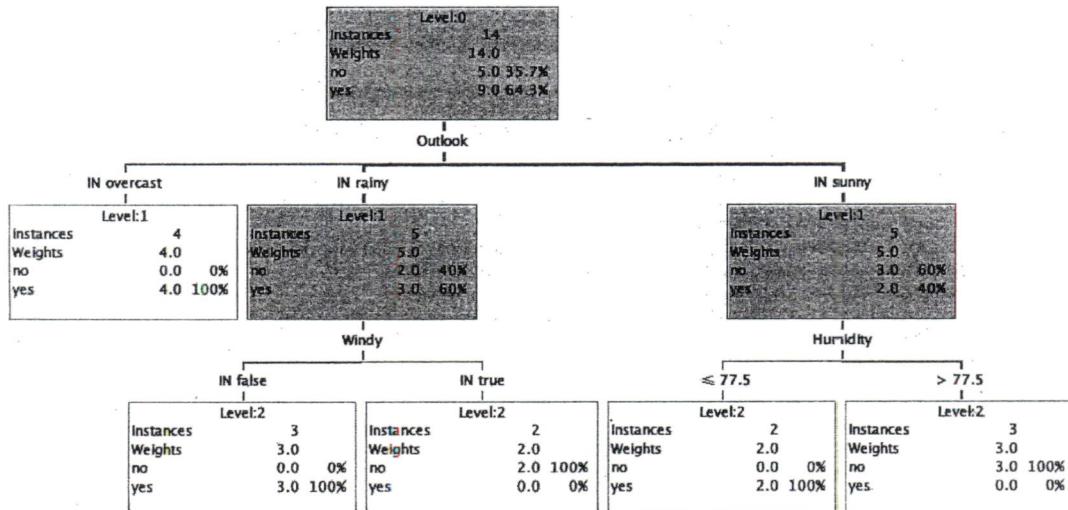


图 9-3

该树结构展示了完整的推断逻辑：

(1) 根节点显示的是整个训练集的情况：14 条样本数据。在没有输入权重列参数的情况下，每条样本的权重都是 1.0，并注明了这些样本数据中各个标签值的分布比例。

(2) 按 Outlook 特征进行判断。对于取值为 overcast 的样本，直接判断结果为 yes；对于其他取值的情况，若两种标签值都存在，则需要继续判断。

(3) 对于 Outlook=sunny 的情况，可根据特征 Humidity 进行判断：该值小于或等于 77.5 时，均判断为 yes；该值大于 77.5 时，均判断为 no。对于 Outlook=rainy 的情况，可根据特征 Windy 进行判断：若值为 true，则判断结果为 no；若值为 false，则判断结果为 yes。

## 9.3 数据探索

本节使用的是 UCI 的 Mushroom 数据集，可通过链接 9-1 下载。其数据来自 *The Audubon Society Field Guide to North American Mushrooms* (1981 年)。

该数据集有 8124 条数据、23 个变量。其中，变量 class 为分类标签，其有 2 个标签值：“e”表示 edible (可食用)，“p”表示 poisonous (有毒)。该数据集共有 22 个特征变量，各特征名称及缩写字母含义如下所示：

---

```

1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
3. cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p,purple=u,red=e,white=w,yellow=y
4. bruises?: bruises=t,no=f
5. odor: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
6. gill-attachment: attached=a,descending=d,free=f,notched=n
7. gill-spacing: close=c,crowded=w,distant=d
8. gill-size: broad=b,narrow=n
9. gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p,purple=u,red=e,
   white=w,yellow=y
10. stalk-shape: enlarging=e,tapering=t
11. stalk-root: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
12. stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
13. stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
14. stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
15. stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p,red=e,white=w,yellow=y
16. veil-type: partial=p,universal=u
17. veil-color: brown=n,orange=o,white=w,yellow=y
18. ring-number: none=n,one=o,two=t
19. ring-type: cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
20. spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r, orange=o,purple=u,white=w,yellow=y
21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d

```

---

这里有几个重要的单词需要说明一下：菌盖 (cap)、菌褶 (gill)、菌柄 (stalk)、菌环 (ring)、气味 (odor)。图 9-4 说明了蘑菇各个部位的名称。

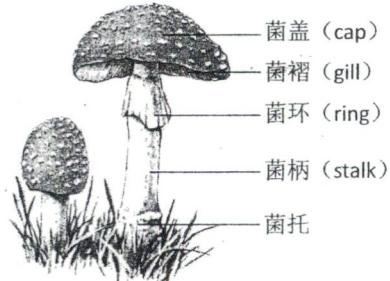


图 9-4

我们将数据文件下载到本地，使用文本编辑器打开该数据文件，数据显示如图 9-5 所示。

agaricus-lepiota.data	
p	x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u
e	x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,g
e	b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,m
p	x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u
e	x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,g
e	x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,n,g
e	b,s,w,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,n,m
e	b,y,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,s,m
p	x,y,w,t,p,f,c,n,p,e,e,s,s,w,w,p,w,o,p,k,v,g
e	b,s,y,t,a,f,c,b,g,e,c,s,s,w,w,p,w,o,p,k,s,m
e	x,y,y,t,l,f,c,b,g,e,c,s,s,w,w,p,w,o,p,n,n,g
e	x,y,y,t,a,f,c,b,n,e,c,s,s,w,w,p,w,o,p,k,s,m

图 9-5

每个特征值都是一个字母，特征值之间以逗号分隔，一行为一条数据。这是典型的 CSV 格式。只需定义好每列的名称，其类型都为 string 类型，便可以使用 CsvSourceBatchOp 组件读取数据了。首先，各列数据的名称和类型如下。标签列为 class 列，除标签列外的各列均为特征列。

```
private static final String DATA_DIR = Chap01.ROOT_DIR + "mushroom" + File.separator;
private static final String ORIGIN_FILE = "agaricus-lepiota.data";
private static final String[] COL_NAMES = new String[] {
    "class",
    "cap_shape", "cap_surface", "cap_color", "bruises", "odor",
    "gill_attachment", "gill_spacing", "gill_size", "gill_color",
    "stalk_shape", "stalk_root", "stalk_surface_above_ring", "stalk_surface_below_ring",
    "stalk_color_above_ring", "stalk_color_below_ring",
    "veil_type", "veil_color",
    "ring_number", "ring_type", "spore_print_color", "population", "habitat"
};

private static final String[] COL_TYPES = new String[] {
    "string",
    "string", "string", "string", "string", "string",
    "string", "string"
};

static final String LABEL_COL_NAME = "class";
static final String[] FEATURE_COL_NAMES = ArrayUtils.removeElement(COL_NAMES, LABEL_COL_NAME);
```

使用 CsvSourceBatchOp 读取并显示前 5 条数据，代码如下：

```
CsvSourceBatchOp source = new CsvSourceBatchOp()
    .setFilePath(DATA_DIR + ORIGIN_FILE)
    .setSchemaStr(Utils.generateSchemaString(COL_NAMES, COL_TYPES));

source.lazyPrint(5, "< origin data >");
```

打印输出如下。在此，可以看到，第一列 class 为分类标签列，随后是各个离散特征列。

```
< origin data >
class|cap_shape|cap_surface|cap_color|bruises|odor|gill_attachment|gill_spacing|gill_size|gill_color|s
talk_shape|stalk_root|stalk_surface_above_ring|stalk_surface_below_ring|stalk_color_above_ring|stalk_c
olor_below_ring|veil_type|veil_color|ring_number|ring_type|spore_print_color|population|habitat
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
p|f|y|y|f|f|c|b|p|e|b|k|k|b|p|p|w|o|l|h|y|d
p|x|f|y|f|f|c|b|p|e|b|k|k|b|n|p|w|o|l|h|y|g
p|x|y|g|f|f|c|b|g|e|b|k|k|n|p|p|w|o|l|h|v|d
p|f|f|g|f|f|f|c|b|g|e|b|k|k|n|b|p|w|o|l|h|y|d
p|f|f|y|f|f|f|c|b|g|e|b|k|k|p|n|p|w|o|l|h|y|p
```

获得原始数据后，我们还需要将其分为训练集和测试集，并按 9 : 1 的比例，分别保存为 AK 格式的文件 TRAIN\_FILE 和 TEST\_FILE，具体代码如下：

```
private static final String TRAIN_FILE = "train.ak";
private static final String TEST_FILE = "test.ak";
...
Utils.splitTrainTestIfNotExist(source, DATA_DIR + TRAIN_FILE, DATA_DIR + TEST_FILE, 0.9);
```

接下来，我们使用卡方特征选择组件 ChiSqSelectorBatchOp，从当前特征列中选出对分类问题最重要的 3 个特征。具体代码如下：

```
new AkSourceBatchOp()
.setFilePath(DATA_DIR + TRAIN_FILE)
.link(
    new ChiSqSelectorBatchOp()
        .setSelectorType(SelectorType.NumTopFeatures)
        .setNumTopFeatures(3)
        .setSelectedCols(FEATURE_COL_NAMES)
        .setLabelCol(LABEL_COL_NAME)
        .lazyPrintModelInfo("< Chi-Square Selector >")
);
}
```

输出结果如下：

```
< Chi-Square Selector >
----- ChisqSelectorModelInfo -----
Number of Selector Features: 3
Number of Features: 22
Type of Selector: NumTopFeatures
Number of Top Features: 3
Selector Indices:
|           ColName|ChiSquare|PValue| DF|Selected|
|-----+-----+-----+-----+-----|
|       odor|6897.9815|     0|   8|    true|
```

spore_print_color 4197. 5627	0  8	true
gill_color 3399. 9436	0  11	true
ring_type 2686. 4367	0  4	false
stalk_surface_above_ring 2542. 7242	0  3	false
stalk_surface_below_ring 2453. 9015	0  3	false
gill_size 2161. 6473	0  1	false
stalk_color_above_ring 2027. 4274	0  8	false
stalk_color_below_ring 1950. 8527	0  8	false
bruises 1874. 4286	0  1	false
population 1769. 4532	0  5	false
habitat 1432. 2396	0  6	false
stalk_root 1254. 6198	0  4	false
gill_spacing  884. 8095	0  1	false
cap_shape  455. 8245	0  5	false
cap_color  350. 5556	0  9	false
ring_number  339. 0443	0  2	false
cap_surface  276. 0812	0  3	false
veil_color  171. 2708	0  3	false
gill_attachment  120. 5918	0  1	false
stalk_shape  70. 8908	0  1	false
veil_type	0  1	0  false

显然，排在前三位的是 `odor`、`spore_print_color` 和 `gill_color`。在后面的示例中，我们会重点观察这三列。我们注意到，排在最后的特征列的卡方值为 0。下面通过一个简单的操作来了解该特征列。我们单独选择该特征列，然后看其有多少不同的值，具体代码如下：

```
new AkSourceBatchOp()
.setFilePath(DATA_DIR + TRAIN_FILE)
.select("veil_type")
.distinct()
.lazyPrint(100);
```

运行结果如下：

```
veil_type
_____
p
```

即特征列 `veil_type` 只有唯一的一个值'p'。显然，该列不会对分类问题有任何贡献，我们可以完全不予考虑。

## 9.4 使用朴素贝叶斯方法

朴素贝叶斯训练组件（`NaiveBayesTrainBatchOp`）和朴素贝叶斯预测组件（`NaiveBayes-`

PredictBatchOp )的参数设置如下面的代码所示。朴素贝叶斯训练组件除了要设置特征列和标签列，还需要指出特征列中的哪些列是离散特征列(对应参数 CategoricalCols )。在当前的分类问题中，所有特征都是离散特征，所以该参数填入所有的特征列 FEATURE\_COL\_NAMES 。朴素贝叶斯预测组件的设置与逻辑回归预测组件和线性 SVM 预测组件一样，需要指定预测结果的列名。朴素贝叶斯预测组件还可以通过设置预测详情的列名，得到属于各分类值的概率，用于后面的二分类评估。

```
NaiveBayesTrainBatchOp trainer =
    new NaiveBayesTrainBatchOp()
        .setFeatureCols(FEATURE_COL_NAMES)
        .setCategoricalCols(FEATURE_COL_NAMES)
        .setLabelCol(LABEL_COL_NAME);

NaiveBayesPredictBatchOp predictor = new NaiveBayesPredictBatchOp()
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME);
```

之后，通过 link 方法，建立起整个实验流程：训练数据源 train\_data 连接朴素贝叶斯训练组件 trainer ；预测过程需要模型，也需要待预测的数据，所以朴素贝叶斯预测组件 predictor 需要使用 linkFrom 方法来同时连接 trainer 输出的模型和待预测的数据源 test\_data ； predictor 组件的输出即预测结果。代码如下：

```
train_data.link(trainer);
predictor.linkFrom(trainer, test_data);
```

为了查看朴素贝叶斯模型的信息，选择训练组件的 lazyPrintModelInfo 方法：

```
trainer.lazyPrintModelInfo();
```

模型信息打印如下，包括特征列的名称、标签值。特别地，对每个标签值给出了其在整个训练集中的比例。标签值'p'(有毒)的比例为 0.4824；标签值'e'(可食用)的比例为 0.5176。然后，给出了各离散特征中每个离散值对应类别'p'和'e'的比例。模型打印出的信息较多，这里只显示了特征 gill\_color 、 spore\_print\_color 和 odor 的信息，略去了其他特征的信息。

---

----- NaiveBayesTextModelInfo -----

```
===== model meta info =====
{label number: 2, feature size: 22, feature col names:
["cap_shape", "cap_surface", "cap_color", "bruises", "odor", "gill_attachment", "gill_spacing", "gill_size", "gill_color", "stalk_shape", "stalk_root", "stalk_surface_above_ring", "stalk_surface_below_ring", "stalk_color_above_ring", "stalk_color_below_ring", "veil_type", "veil_color", "ring_number", "ring_type", "spore_print_color", "population", "habitat"], labels: ["p", "e"]}

===== label proportion information =====
```

Label info:[p, e]

```

proportion:[0.4824, 0.5176]
===== category information =====

categorical features: [stalk_surface_above_ring, habitat, gill_spacing, ..., stalk_color_above_ring,
gill_size, ring_type]
gaussian features: []
===== categorical features proportion information =====

...
The features proportion information of gill_color:
| | p| r| b|...| k| n| o|
|---|---|---|---|---|---|---|---|
| p| 0.4251| 1| 1|...| 0.1432| 0.1065| 0|
| e| 0.5749| 0| 0|...| 0.8568| 0.8935| 1|


The features proportion information of spore_print_color:
| | r| b| u|...| k| n| o|
|---|---|---|---|---|---|---|---|
| p| 1| 0| 0|...| 0.1136| 0.1142| 0|
| e| 0| 1| 1|...| 0.8864| 0.8858| 1|


The features proportion information of odor:
| | p| a| c|...| l| m| n|
|---|---|---|---|---|---|---|---|
| p| 1| 0| 1|...| 0| 1| 0.0339|
| e| 0| 1| 0|...| 1| 0| 0.9661|


...
===== continuous features mean sigma information =====

There is no continuous feature.

```

对于特征 gill\_color、spore\_print\_color 和 odor，每个特征的离散值较多，在此无法完全显示，我们只选择打印输出了 6 个离散值。如果读者希望了解这三个特征的全部离散值相对于分类标签值的概率，可以使用 lazyCollectModelInfo 方法，获取到 NaiveBayesModelInfo 对象，选择输出内容，具体代码如下：

---

```

trainer.lazyCollectModelInfo(new Consumer<NaiveBayesModelInfo>() {
    @Override
    public void accept(NaiveBayesModelInfo naiveBayesModelInfo) {
        StringBuilder sbd = new StringBuilder();
        for (String feature : new String[] {"odor", "spore_print_color", "gill_color"}) {
            HashMap<Object, HashMap<Object, Double>> map2 =
                naiveBayesModelInfo.getCategoryFeatureInfo().get(feature);
            sbd.append("\nfeature:").append(feature);
            for (Entry<Object, HashMap<Object, Double>> entry : map2.entrySet()) {

```

---

---

```

        sbd.append("\n").append(entry.getKey()).append(" : ")
        .append(entry.getValue().toString());
    }
}
System.out.println(sbd.toString());
});
}
});

```

---

运行结果如下。在此可看到很多离散值对应着概率值 1.0，即取到这些值，便可以 100% 判断样本的所属分类。尤其是 odor 特征，当离散值不为 'n' 的时候，均可以完全判断样本的类别；当离散值为 'n' 的时候，属于标签值 'e'（可食用）的概率值也很大，是 96.6%。

```

feature:odor
p : {p=1.0, s=1.0, c=1.0, f=1.0, y=1.0, m=1.0, n=0.033871478315922764}
e : {a=1.0, l=1.0, n=0.9661285216840773}
feature:spore_print_color
p : {r=1.0, w=0.763585694379935, h=0.9720518064076347, k=0.11355529131985731, n=0.11423747889701745}
e : {b=1.0, u=1.0, w=0.23641430562006502, h=0.02794819359236537, y=1.0, k=0.8864447086801427,
n=0.8857625211029826, o=1.0}
feature:gill_color
p : {p=0.42509225092250924, b=1.0, r=1.0, u=0.09610983981693363, w=0.20888468809073724,
g=0.668141592920354, h=0.7216338880484114, y=0.25675675675675674, k=0.14323607427055704,
n=0.10649627263045794}
e : {p=0.5749077490774908, e=1.0, u=0.9038901601830663, w=0.7911153119092628, g=0.33185840707964603,
h=0.2783661119515885, y=0.7432432432432432, k=0.8567639257294429, n=0.8935037273695421, o=1.0}

```

最后，我们打印 10 条预测记录，并根据预测结果集，计算输出朴素贝叶斯二分类模型的评估指标，具体代码如下：

---

```

predictor.lazyPrint(10, "< Prediction >");

predictor
.link(
    new EvalBinaryClassBatchOp()
    .setPositiveLabelValueString("p")
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .lazyPrintMetrics()
);

```

---

```
BatchOperator.execute();
```

结果如下，最右面的列为预测详情列，右边第二列是预测结果列。各项评估指标都比较高，但从混淆矩阵中可看到，仍有一例误判的情况：将一个有毒（标签值为 p）的蘑菇判断为可食用（标签值为 e）的蘑菇：

```
< Prediction >
class|cap_shape|cap_surface|cap_color|bruises|odor|gill_attachment|gill_spacing|gill_size|gill_color|.
```

```
...|-----|-----|-----|-----|-----|-----|-----|-----|.
...|-----|-----|-----|-----|-----|-----|-----|-----|.
e|f|y|n|t|n|f|c|b|p|t|b|s|s|p|w|o|p|k|y|d|e| {"p":3.1915386674427295E-5, "e":0.9999680846133256}
e|f|f|e|t|n|f|c|b|p|t|b|s|s|p|p|w|o|p|k|y|d|e| {"p":8.364168330966485E-5, "e":0.9999163583166915}
p|x|s|w|f|c|f|c|n|p|e|b|s|s|w|w|p|w|o|p|k|v|d|p| {"p":1.0, "e":0.0}
e|f|y|g|t|n|f|c|b|p|t|b|s|s|p|p|w|o|p|n|v|d|e| {"p":4.366093667648151E-4, "e":0.9995633906332355}
p|x|y|y|f|f|f|c|b|g|e|b|k|k|n|b|p|w|o|l|h|y|p|p| {"p":1.0, "e":0.0}
e|x|y|n|t|n|f|c|b|w|t|b|s|s|p|w|p|w|o|p|k|y|d|e| {"p":1.0517174774921838E-5, "e":0.9999894828252253}
e|x|y|w|t|a|f|c|b|w|e|c|s|s|w|w|p|w|o|p|n|s|m|e| {"p":0.0, "e":1.0}
e|x|s|w|f|n|f|w|b|k|t|e|s|s|w|w|p|w|o|e|k|a|g|e| {"p":0.0, "e":1.0}
e|x|s|w|f|n|f|w|b|p|t|e|s|s|w|w|p|w|o|e|n|s|g|e| {"p":3.613017366628178E-6, "e":0.9999963869826345}
e|f|f|e|t|n|f|c|b|u|t|b|s|s|w|g|p|w|o|p|n|v|d|e| {"p":0.0, "e":1.0}

Metrics:
Auc: 0.9998 Accuracy: 0.9889 Precision: 0.9798 Recall: 0.9974 F1: 0.9885 LogLoss: 0.0232
|Pred\Real| p| e|
|-----|---|---|
| p| 388| 8|
| e| 1| 415|
```

前面，我们训练朴素贝叶斯模型时用到了所有特征列。如果减少特征列的个数，会怎么样呢？比如，只关注卡方特征选择组件给出的前三个特征列：gill\_color、spore\_print\_color 和 odor，会出现什么情况？

我们先了解一下这三个特征列。其中，菌褶颜色（gill\_color）和气味（odor），一看名称就能理解，而且这两个特征是大家可以直接看到、闻到的。

孢子印（spore print）指的是蘑菇孢子散落而沉积的菌褶或菌管的印迹。孢子印及其颜色可作为蘑菇分类的依据之一。孢子印的制作方式如下：把菌褶或菌管上的子实层所产生的孢子接收在白纸或黑纸上。孢子印越厚，就越容易得到孢子颜色的准确判断。其制作过程需要数小时。

接下来的实验，我们选择菌褶颜色（gill\_color）和气味（odor）这两个特征，得到的模型有助于我们在实际生活中对蘑菇是否有毒有一个初步判断。整个流程与前面对全部特征进行训练的代码基本相同，只是模型训练参数的设置不同，具体代码如下。训练特征列为“odor”、“gill\_color”，离散特征列为“odor”、“gill\_color”。

---

```
NaiveBayesTrainBatchOp trainer =
    new NaiveBayesTrainBatchOp()
        .setFeatureCols("odor", "gill_color")
        .setCategoricalCols("odor", "gill_color")
        .setLabelCol(LABEL_COL_NAME);
```

---

使用 lazyCollectModelInfo 方法获取特征“odor”和“gill\_color”的概率值，代码如下：

---

```
trainer.lazyCollectModelInfo(new Consumer<NaiveBayesModelInfo>() {
```

---

```

@Override
public void accept(NaiveBayesModelInfo naiveBayesModelInfo) {
    StringBuilder sbd = new StringBuilder();
    for (String feature : new String[] {"odor", "gill_color"}) {
        HashMap <Object, HashMap <Object, Double>> map2 =
            naiveBayesModelInfo.getCategoryFeatureInfo().get(feature);
        sbd.append("\nfeature:").append(feature);
        for (Entry <Object, HashMap <Object, Double>> entry : map2.entrySet()) {
            sbd.append("\n").append(entry.getKey()).append(" : ")
                .append(entry.getValue().toString());
        }
    }
    System.out.println(sbd.toString());
}
});

```

预测结果如下：

```

< Prediction >
class|cap_shape|cap_surface|cap_color|bruises|odor|gill_attachment|gill_spacing|gill_size|gill_color|
...|...|...|...|...|...|...|...|...
...|...|...|...|...|...|...|...|...
e|f|y|n|t|n|f|c|b|p|t|b|s|s|p|w|p|w|o|p|k|y|d|e|{"p":0.027066242618865737, "e":0.9729337573811344}
e|f|f|e|t|n|f|c|b|p|t|b|s|s|p|p|p|w|o|p|k|y|d|e|{"p":0.027066242618865737, "e":0.9729337573811344}
p|x|s|w|f|c|f|c|n|p|e|b|s|s|w|w|p|w|o|p|k|v|d|p|{"p":1.0, "e":0.0}
e|f|y|g|t|n|f|c|b|p|t|b|s|s|p|p|p|w|o|p|n|v|d|e|{"p":0.027066242618865737, "e":0.9729337573811344}
p|x|y|y|f|f|f|c|b|g|e|b|k|k|n|b|p|w|o|1|h|y|p|p|{"p":1.0, "e":0.0}
e|x|y|n|t|n|f|c|b|f|w|t|b|s|s|p|w|p|w|o|p|k|y|d|e|{"p":0.009836338642934176, "e":0.990163661357066}
e|x|y|w|t|a|f|c|b|w|e|c|s|s|w|w|p|w|o|p|n|s|m|e|{"p":0.0, "e":1.0}
e|x|s|w|f|n|f|w|b|k|t|e|s|s|w|w|p|w|o|e|k|a|g|e|{"p":0.0062506869874194345, "e":0.9937493130125807}
e|x|s|w|f|n|f|w|b|p|t|e|s|s|w|w|p|w|o|e|n|s|g|e| {"p":0.027066242618865737, "e":0.9729337573811344}
e|f|f|e|t|n|f|c|b|u|t|b|s|s|w|g|p|w|o|p|n|v|d|e| {"p":0.0039845377693012935, "e":0.9960154622306986}

```

模型评估指标如下：

Metrics:					
Auc:0.9937	Accuracy:0.9901	Precision:1	Recall:0.9794	F1:0.9896	LogLoss:0.0505
Pred\Real	. p	e			
-----	-----	-----			
p   381   0					
e   8   423					

在此可发现整体指标与选择全部特征训练出的模型差异不大，准确率（Precision）为 1.0（即判断为有毒的样本一定是有毒的）；召回率（Recall）为 0.9794，略低于前面模型的 0.9974。这两种模型都没做到将毒蘑菇全部分出来。

我们再看一下打印输出的朴素贝叶斯模型特征“odor”和“gill\_color”的概率值，如下所示：

---

feature:odor

---

```

p : {p=1.0, c=1.0, s=1.0, f=1.0, y=1.0, m=1.0, n=0.033871478315922764}
e : {a=1.0, l=1.0, n=0.9661285216840773}
feature:gill_color
p : {p=0.42509225092250924, b=1.0, r=1.0, u=0.09610983981693363, g=0.668141592920354,
w=0.2088468809073724, h=0.7216338880484114, y=0.25675675675675674, k=0.14323607427055704,
n=0.10649627263045794}
e : {p=0.5749077490774908, e=1.0, u=0.9038901601830663, g=0.33185840707964633,
w=0.7911153119092628, h=0.2783661119515885, y=0.7432432432432432, k=0.8567639257294429,
n=0.8935037273695421, o=1.0}

```

---

对比前面使用全部特征得到的朴素贝叶斯模型，发现这两个特征“odor”和“gill\_color”的概率值没有发生变化（了解一下朴素贝叶斯的原理，就很容易理解这一点了）。

## 9.5 蘑菇分类的决策树

本节会将决策树算法应用到蘑菇分类问题上。3种常用的决策树算法及其对应的分裂指标如表9-11所示。

表9-11 3种常用的决策树算法及其对应的分裂指标

算法	分裂指标
ID3	信息增益（Information Gain）
C4.5	信息增益率（Information Gain Ratio）
CART	基尼指数（Gini Index）

Alink对每种算法都单独提供了训练组件和预测组件：Id3TrainBatchOp、Id3PredictBatchOp、C45TrainBatchOp、C45PredictBatchOp、CartTrainBatchOp、CartPredictBatchOp。比如，前面示例中就使用了C45TrainBatchOp。

Alink还提供了一种使用方式，将3种算法整合为决策树组件（DecisionTreeTrainBatchOp、DecisionTreePredictBatchOp），通过设置树类型（TreeType）参数来选择使用哪种算法。

对于蘑菇数据集，我们会尝试使用3种决策树算法来看看具体分类效果。使用统一的决策树训练组件DecisionTreeTrainBatchOp，各种算法通过调整树类型参数TreeType来进行切换。具体代码如下，决策树训练组件要设置特征列和标签列，还需要指出特征列中哪些列是离散特征列（对应参数CategoricalCols）。在当前的分类问题中，所有特征都是离散特征，所以该参数填入所有的特征列FEATURE\_COL NAMES。决策树预测组件需要指定预测结果列名，还可以通过设置预测详情列名，得到属于各分类值的概率，用于随后的二分类评估。

---

```
BatchOperator train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE);
```

---

```

BatchOperator test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);

for (TreeType treeType : new TreeType[] {TreeType.GINI, TreeType.INFOGAIN, TreeType.INFOGAINRATIO}) {
    BatchOperator <?> model = train_data
        .link(
            new DecisionTreeTrainBatchOp()
                .setTreeType(treeType)
                .setFeatureCols(FEATURE_COL_NAMES)
                .setCategoricalCols(FEATURE_COL_NAMES)
                .setLabelCol(LABEL_COL_NAME)
                .lazyCollectModelInfo(new Consumer<DecisionTreeModelInfo>() {
                    @Override
                    public void accept(DecisionTreeModelInfo decisionTreeModelInfo) {
                        try {
                            decisionTreeModelInfo.saveTreeAsImage(
                                DATA_DIR + "tree_" + treeType.toString() + ".jpg", true);
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                })
        );
}

DecisionTreePredictBatchOp predictor = new DecisionTreePredictBatchOp()
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME);

predictor.linkFrom(model, test_data);

predictor.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("p")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics()
);
}

BatchOperator.execute();

```

3 种决策树算法的评估指标对比如表 9-12 所示。

表 9-12 3 种决策树算法的评估指标对比

ID3	C4.5	CART
TreeType.INFOGAIN	TreeType.INFOGAINRATIO	TreeType.GINI
Auc:1	Auc:1	Auc:1
Accuracy:1	Accuracy:1	Accuracy:1
Precision:1	Precision:1	Precision:1
Recall:1	Recall:1	Recall:1
F1:1	F1:1	F1:1
LogLoss:0	LogLoss:0	LogLoss:0

续表

ID3	C4.5	CART
TreeType.INFOGAIN	TreeType.INFOGAINRATIO	TreeType.GINI
混淆矩阵  Pred\Real  p e  ----- --- ---    p  389  0    e   0 423	混淆矩阵  Pred\Real  p e  ----- --- ---    p  389  0    e   0 423	混淆矩阵  Pred\Real  p e  ----- --- ---    p  389  0    e   0 423

从评估指标上看，这3种决策树算法都对测试集进行了完全正确的分类，比朴素贝叶斯算法更胜一筹。

我们打开树模型图片（见图9-6和见图9-7），观察各算法模型的决策过程。ID3算法(TreeType.INFOGAIN, 图9-6)和C4.5算法(TreeType.INFOGAINRATIO, 图9-7)的决策过程比较相似。先是根据特征odor进行划分，均只有一个节点的样本含有两个标签值，需要继续划分；其他各个节点都满足：节点内的样本标签值都相同，可以作为叶节点给出决策结果。继续进行划分，两种算法的选择还是一样的，都是根据特征spore\_print\_color进行划分。这两级之后，这两种算法展现出明显的差异。

我们再观察CART算法(TreeType.GINI)，如图9-8所示。CART算法每次都是2分划。在此可以看到，首先仍以特征odor做划分，分为两部分。左边的分枝有多个离散值，再次划分这些离散值的时候，可将两种标签值的数据完全分开。这个分枝就相当于前面ID3和C4.5第一层中那些样本标签值都相同的叶节点。右边的分枝是对特征odor=n(即没有气味)的样本继续分类的结果。接下来使用特征stalk\_surface\_below\_ring进行划分。

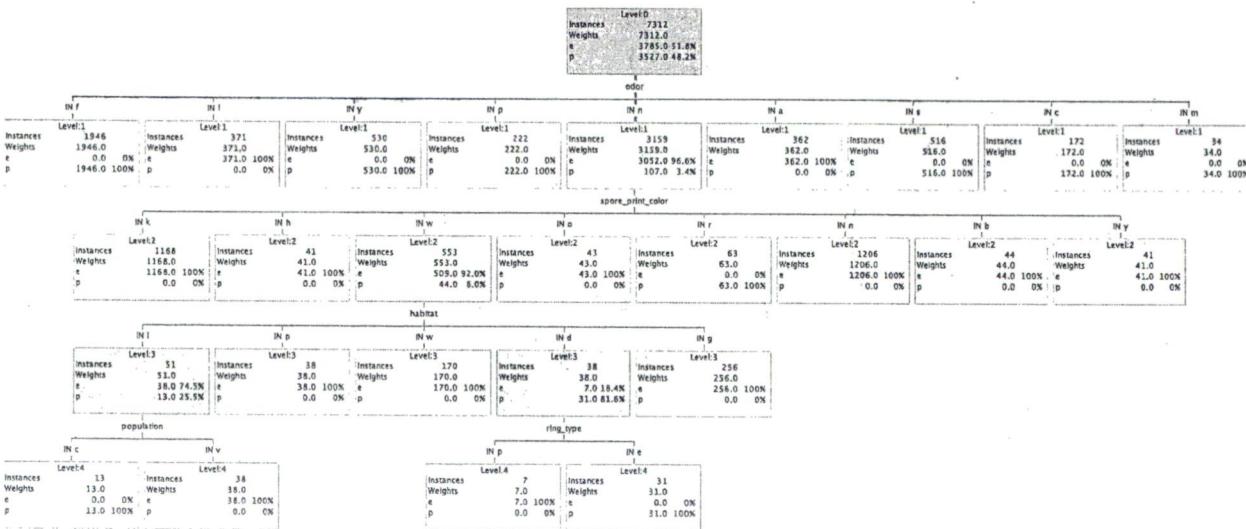


图 9-6

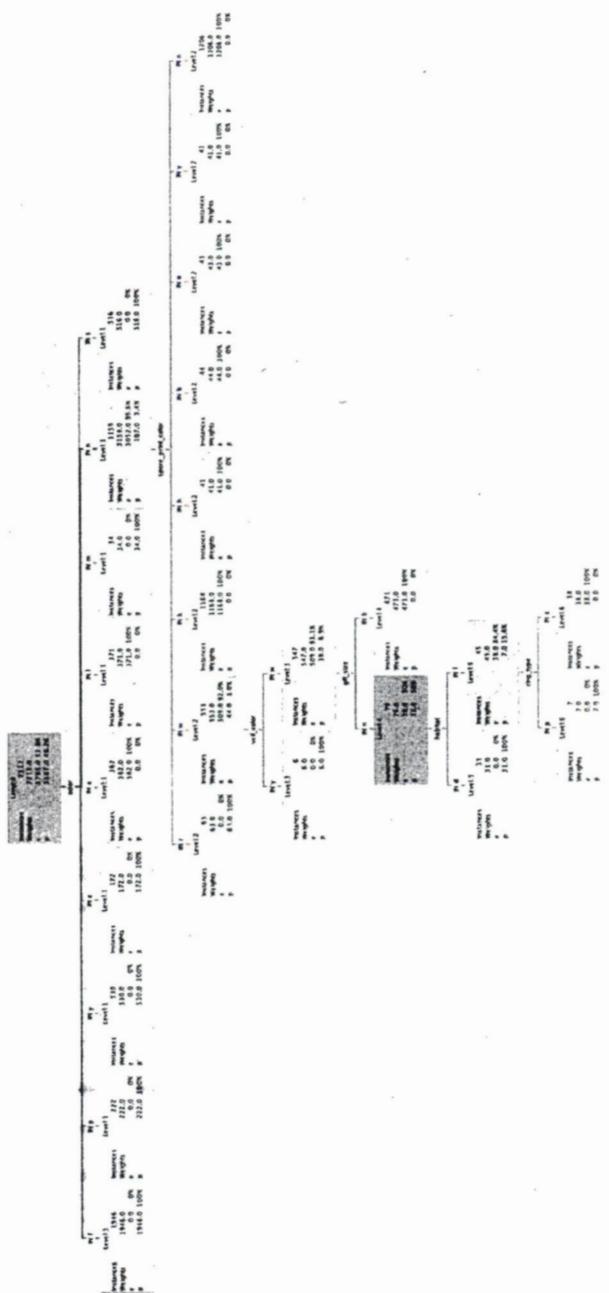


图 9-7

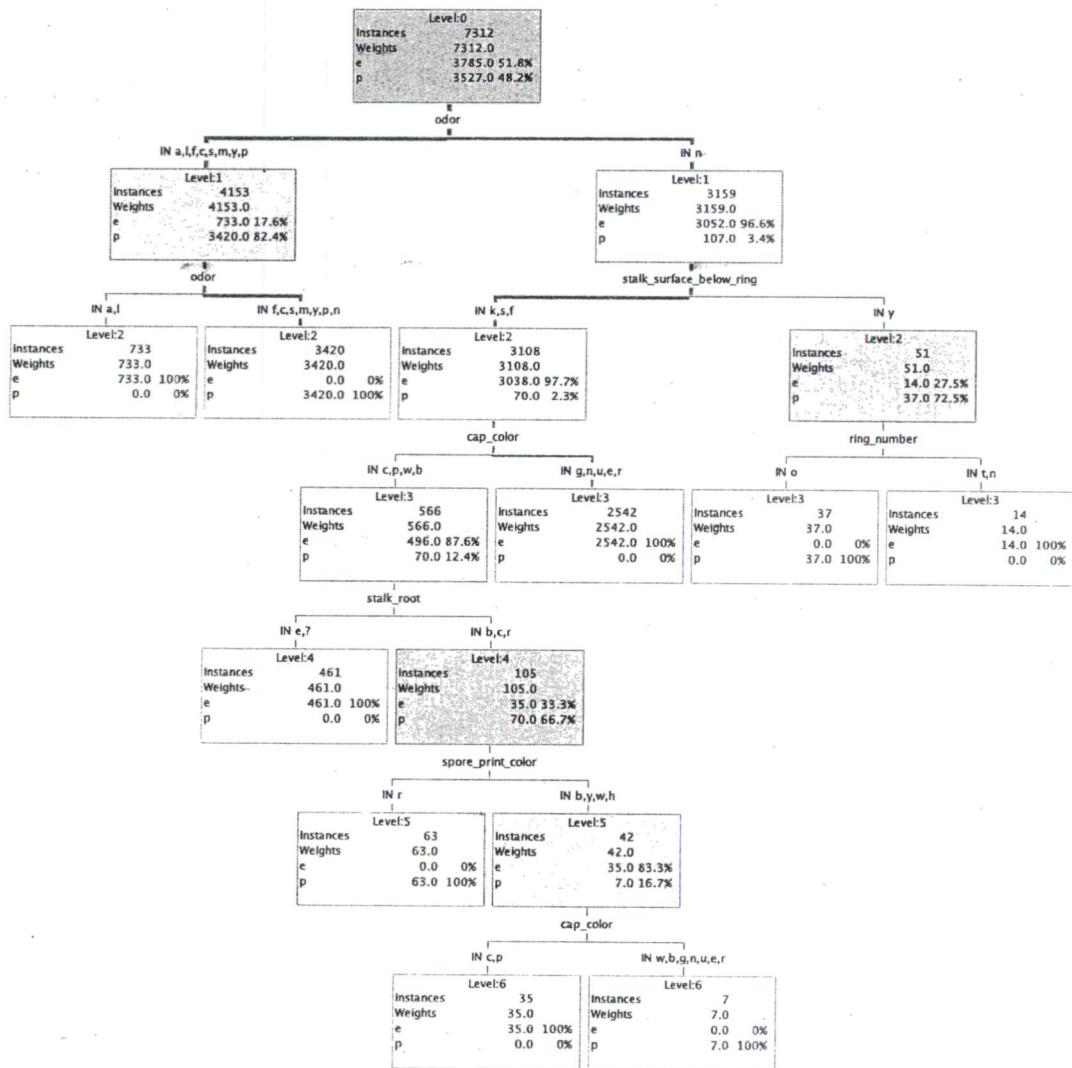


图 9-8

决策树的集成算法（比如，随机森林、GBDT）的分类效果会优于单棵决策树算法的分类效果。

# 10

## 特征的转化

前面介绍了如何使用数值类型的连续特征或枚举类型的离散特征建立二分类模型。大家或许已有了一个笼统的印象：所有分类模型都能处理数值类型的连续特征，部分分类模型能处理枚举类型的离散特征。比如，逻辑回归就是一个形式简单、效果不错的分类方法，但是其只能处理数值类型的数据。是否能通过某种变化或方法，让逻辑回归模型也可使用枚举类型的离散特征呢？

本章讲述特征哑元化的方法，实现特征由枚举类型向数值类型的转化，这样我们就可以使用逻辑回归等分类模型。特征哑元化的方法在给我们带来惊喜的同时，也使我们看到，特征的数量在急剧增长。在众多的特征中，哪些特征更重要呢，是否可以减少模型特征的个数呢？本章将通过实例分析，与大家一起探讨上述问题。

本章使用的数据集为德国信用数据集（German Credit Dataset），读者可从链接 10-1 下载。本数据集记录了 1000 名用户的 20 个属性，且从信用风险的角度，将客户分为好客户（Good=1，按时偿还贷款）与坏客户（Bad=2，违约）两类。

数据特征属性的字段说明，如表 10-1 所示。

表 10-1 数据特征属性的字段说明

属性编号	数据类型	字段名称	含义	备注
1	枚举	status	现有的支票账户的状态	A11: ..... <0 马克 A12: 0 ≤ ..... <200 马克 A13: ..... ≥ 200 马克 A14: 没有支票账户

续表

属性 编号	数据 类型	字段名称	含义	备注
2	数值	duration	持续时间（单位：月）	
3	枚举	credit_history	信用记录	A30: 没有贷款/所有贷款已及时还清 A31: 在这家银行的所有贷款都已及时偿还 A32: 时至今日仍有贷款，但该贷款目前处于及时还款之中 A33: 过去有延迟还款的记录 A34: 关键账户/存在其他贷款（不是在这家银行申请的贷款）
4	枚举	purpose	目的	A40: 汽车（新） A41: 汽车（使用过） A42: 家具/设备 A43: 广播/电视 A44: 家用电器 A45: 维修 A46: 教育 A47: （假期 - 不存在吗？） A48: 培训 A49: 商务 A410: 其他
5	数值	credit_amount	信贷量	
6	枚举	savings	储蓄账户/债券	A61: ..... <100 马克 A62: 100 ≤ ..... <500 马克 A63: 500 ≤ ..... <1000 马克 A64: .. ≥ 1000 马克 A65: 不详/无主的储蓄账户
7	枚举	employment	职业	A71: 失业 A72: ..... <1 年 A73: 1 ≤ ..... <4 年 A74: 4 ≤ ..... <7 年 A75: ..... ≥7 年
8	数值	installment_rate	在可支配收入的百分比中，分期付款的比例	

续表

属性 编号	数据 类型	字段名称	含义	备注
9	枚举	marriage_sex	个人身份和性别	A91: 男: 离婚/分居 A92: 女: 离婚/分居/已婚 A93: 男: 未婚 A94: 男: 已婚/丧偶 A95: 女: 未婚
10	枚举	debtors	其他债务人/担保人	A101: 无 A102: 共同申请人 A103: 担保人
11	数值	residence	目前住处	
12	枚举	property	财产	A121: 房地产 A122: 如果不是 A121 这种情况: 具备建房互助协会的储蓄协定/人寿保险 A123: 如果不是 A121/A122 这种情况: 汽车或其他不在属性 6 中的储蓄账户/债券 A124: 未知/无财产
13	数值	age	年龄	
14	枚举	other_plan	其他分期付款计划	A141: 银行 A142: 存储 A143: 无
15	枚举	housing	住房	A151: 租用 A152: 自有 A153: 免费
16	数值	number_credits	在这家银行的现有贷款数目	
17	枚举	job	工作职位	A171: 无业/不熟练 - 非居民 A172: 不熟练 - 居民 A173: 熟练的员工/公务员 A174: 经理/个体户/高级职员/官员
18	数值	maintenance_num	赡养人数	
19	枚举	telephone	电话	A191: 无 A192: 有, 用客户的姓名进行注册
20	枚举	foreign_worker	外国工作者	A201: 是 A202: 无

该数据可以被看作以空格为字段分隔符的 CSV 格式数据。使用 CsvSourceBatchOp 读取的代码如下。随后打印统计信息，显示前 5 条数据，并将原始数据按 8 : 2 的比例分为训练集和测试集。

```
CsvSourceBatchOp source = new CsvSourceBatchOp()
    .setFilePath(DATA_DIR + ORIGIN_FILE)
    .setSchemaStr(Utils.generateSchemaString(COL_NAMES, COL_TYPES))
    .setFieldDelimiter(" ");

source
    .lazyPrint(5, "< origin data >")
    .lazyPrintStatistics();

BatchOperator.execute();

Utils.splitTrainTestIfNotExist(source, DATA_DIR + TRAIN_FILE, DATA_DIR + TEST_FILE, 0.8);
```

注意：打印输出的原始数据和统计信息都为 Markdown 表格的形式。使用任意 Markdown 编辑器都可以将其转换为一般的表格形式进行展现。本书为了便于读者阅读，直接显示其转换后的内容。

前 5 条原始数据如图 10-1 所示。

status	duration	credit_HI_story	purpose	credit_a_mount	savings	employment_rate	installment_rate	marriage	debtors	residence	property	age	other_employment	housing	number_credits	job	maintenance	telephone	foreign_worker	class
A13	6	A32	A42	2116	A61	A73	2	A93	A101	2	A121	41	A143	A152	1	A173	1	A192	A201	1
A12	9	A21	A40	1437	A62	A74	2	A93	A101	3	A124	29	A143	A152	1	A173	1	A191	A201	2
A14	42	A34	A42	4042	A63	A73	4	A93	A101	4	A121	36	A143	A152	2	A173	1	A192	A201	1
A13	9	A32	A46	3632	A65	A75	1	A93	A101	4	A121	64	A143	A152	1	A172	1	A191	A201	1
A11	34	A32	A43	3660	A61	A73	2	A92	A101	4	A123	28	A143	A152	1	A173	1	A191	A201	1

图 10-1

原始数据的统计信息如图 10-2 所示，共 1000 条数据，没有缺失值，不到一半的特征为数值特征。标签值列 class，均值为 1.3，可知标签值为 2 的样本占 30%，标签值为 1 的样本占 70%；特征 age，最小值为 19，最大值为 75，客户的平均年龄为 35.546；特征 credit\_amount 的取值范围较大，其取值范围为 250 ~ 18 424。

colName	count	missing	sum	mean	variance	min	max
status	1000	0	NaN	NaN	NaN	NaN	NaN
duration	1000	0	20903	20.903	145.415	4	72
credit_history	1000	0	NaN	NaN	NaN	NaN	NaN
purpose	1000	0	NaN	NaN	NaN	NaN	NaN
credit_amount	1000	0	3271258	3271.258	7967843.4709	250	18424
savings	1000	0	NaN	NaN	NaN	NaN	NaN
employment	1000	0	NaN	NaN	NaN	NaN	NaN
installment_rate	1000	0	2973	2.973	1.2515	1	4
marriage_sex	1000	0	NaN	NaN	NaN	NaN	NaN
debtors	1000	0	NaN	NaN	NaN	NaN	NaN
residence	1000	0	2845	2.845	1.2182	1	4
property	1000	0	NaN	NaN	NaN	NaN	NaN
age	1000	0	35546	35.546	129.4013	19	75
other_plan	1000	0	NaN	NaN	NaN	NaN	NaN
housing	1000	0	NaN	NaN	NaN	NaN	NaN
number_credits	1000	0	1407	1.407	0.3337	1	4
job	1000	0	NaN	NaN	NaN	NaN	NaN
maintenance_num	1000	0	1155	1.155	0.1311	1	2
telephone	1000	0	NaN	NaN	NaN	NaN	NaN
foreign_worker	1000	0	NaN	NaN	NaN	NaN	NaN
class	1000	0	1300	1.3	0.2102	1	2

图 10-2

## 10.1 整体流程

将本章所要介绍的特征哑元化操作和特征的重要性等内容都整合到一个流程中，如下所示：

```
BatchOperator <?> train_data =
    new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE).select(CLAUSE_CREATE_FEATURES);

BatchOperator <?> test_data =
    new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE).select(CLAUSE_CREATE_FEATURES);

String[] new_features = ArrayUtils.removeElement(train_data.getColNames(), LABEL_COL_NAME);

train_data.lazyPrint(5, "< new features >");

LogisticRegressionTrainBatchOp trainer = new LogisticRegressionTrainBatchOp()
    .setFeatureCols(new_features)
    .setLabelCol(LABEL_COL_NAME);
```

```

LogisticRegressionPredictBatchOp predictor = new LogisticRegressionPredictBatchOp()
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME);

train_data.link(trainer);

predictor.linkFrom(trainer, test_data);

trainer
    .lazyPrintTrainInfo()
    .lazyCollectTrainInfo(new Consumer<LinearModelTrainInfo>() {
        @Override
        public void accept(LinearModelTrainInfo linearModelTrainInfo) {
            printImportance(
                linearModelTrainInfo.getColNames(),
                linearModelTrainInfo.getImportance()
            );
        }
    });
};

predictor.link(
    new EvalBinaryClassBatchOp()
        .setPositiveLabelValueString("2")
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME)
        .lazyPrintMetrics()
);
BatchOperator.execute();

```

这里需要说明如下几点：

- (1) 获取训练集和测试集的时候，就做了特征哑元化的操作，是通过执行 SQL 操作——`select(CLAUSE_CREATE_FEATURES)`实现的。
- (2) 不仅要对训练数据进行特征哑元化的操作，还应对所要预测的测试数据进行特征哑元化的操作。
- (3) 获取特征哑元化后新数据集的特征名称 `new_features`，用于后续训练预测。
- (4) 随后是设置训练、预测组件参数，连接整个流程。
- (5) 逻辑回归模型为线性模型，可以根据模型中各属性的系数及训练集中各特征的统计指标，得到每个属性的重要性。通过逻辑回归训练组件 `trainer` 的 `lazyPrintTrainInfo` 方法可以获得最重要的 3 个特征信息。若想要知道全部特征的信息，则需要使用 `lazyCollectTrainInfo` 方法，得到 `LinearModelTrainInfo`。
- (6) 最后使用二分类评估组件进行评估，并输出评估结果。

下面将详细介绍算法的细节。

### 10.1.1 特征哑元化

下面首先介绍一下特征哑元化。特征哑元化指的是将枚举类型的特征变量转换为多个二值特征变量。举个例子，如果特征属性  $F$  有 4 个属性值 { $a, b, c, d$ }，则可以通过定义 4 个新的特征变量  $F_a, F_b, F_c, F_d$ ，使得每个新变量只能取两个值，即 0 或 1，表示特征属性  $F$  是否取到了某个属性值。如果某条记录的特征  $F$  的取值为  $b$ ，则特征变量  $F_a=0, F_b=1, F_c=0, F_d=0$ 。

这种变换可以通过批式组件的 `select(String clause)` 方法轻松实现。比如，本例中特征哑元化操作对应的 SQL 语句 `CLAUSE_CREATE_FEATURES` 定义如下：

```
static final String CLAUSE_CREATE_FEATURES
= "(case status when 'A11' then 1 else 0 end) as status_A11,"
+ "(case status when 'A12' then 1 else 0 end) as status_A12,"
+ "(case status when 'A13' then 1 else 0 end) as status_A13,"
+ "(case status when 'A14' then 1 else 0 end) as status_A14,"
+ "duration,"
+ "(case credit_history when 'A30' then 1 else 0 end) as credit_history_A30,"
+ "(case credit_history when 'A31' then 1 else 0 end) as credit_history_A31,"
+ "(case credit_history when 'A32' then 1 else 0 end) as credit_history_A32,"
+ "(case credit_history when 'A33' then 1 else 0 end) as credit_history_A33,"
+ "(case credit_history when 'A34' then 1 else 0 end) as credit_history_A34,"
...
...
+ "age,"
+ "(case other_plan when 'A141' then 1 else 0 end) as other_plan_A141,"
+ "(case other_plan when 'A142' then 1 else 0 end) as other_plan_A142,"
+ "(case other_plan when 'A143' then 1 else 0 end) as other_plan_A143,"
+ "(case housing when 'A151' then 1 else 0 end) as housing_A151,"
+ "(case housing when 'A152' then 1 else 0 end) as housing_A152,"
+ "(case housing when 'A153' then 1 else 0 end) as housing_A153,"
+ "number_credits,"
+ "(case job when 'A171' then 1 else 0 end) as job_A171,"
+ "(case job when 'A172' then 1 else 0 end) as job_A172,"
+ "(case job when 'A173' then 1 else 0 end) as job_A173,"
+ "(case job when 'A174' then 1 else 0 end) as job_A174,"
+ "maintenance_num,"
+ "(case telephone when 'A192' then 1 else 0 end) as telephone,"
+ "(case foreign_worker when 'A201' then 1 else 0 end) as foreign_worker,"
+ "class";
```

这里需要注意 3 个地方：

(1) SQL 语句中的“`case ... when ... then ... else ... end`”为条件选择语句。比如，语句“`case status when 'A11' then 1 else 0 end`”表示当 `status` 的值为'A11'时，选择 1，否则为 0。

(2) 对于具有多个属性值的 `status`，使用判断语句，生成 4 个新属性 `status_A11`、`status_A12`、`status_A13` 和 `status_A14`。

(3) 对于只有 2 个属性值的 telephone 和 foreign\_worker，则可以通过判断是否出现这 2 个属性值中的一个值，将属性转换为用 0-1 表示的二值属性。

我们再通过对比显示，看看该操作的影响。原始的数据如图 10-3 所示；执行哑元化操作后的数据如图 10-4 所示。

status	duration	credit_hi	purpose	credit_a	savings	employm	installme	marriage	debtors	residenc	property	age	other_pl	housing	number_	job	maintena	telephon	foreign_	class
	month	story	amount	amt	amt	ent	mt_rate	_sex	e	e	e	an	on	credits	num	num	num	num	worker	
A13	6	A32	A42	2116	A61	A73	2	A93	A101	2	A121	41	A143	A152	1	A173	1	A192	A201	1
A12	9	A31	A40	1437	A62	A74	2	A93	A101	3	A124	29	A143	A152	1	A173	1	A191	A201	2
A14	42	A32	A42	4042	A63	A73	4	A93	A101	4	A121	36	A143	A152	2	A173	1	A192	A201	1
A14	9	A32	A46	3832	A65	A75	1	A93	A101	4	A121	64	A143	A152	1	A172	1	A191	A201	1
A11	24	A32	A43	2660	A61	A73	2	A92	A101	4	A123	28	A143	A152	1	A173	1	A191	A201	1

图 10-3

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

图 10-4

对比两图可知，执行哑元化操作后，特征的个数增加了很多。比如，原始数据特征 job，经过哑元化操作后，产生了 4 个特征：job\_A171、job\_A172、job\_A173、job\_A174。特征 maintenance\_num 是数值字段，没有受到变换的影响；特征 telephone 和 foreign\_worker 的字段名称没有发生变化，但内容发生了变化，它们都是只有 2 个属性值的特征，可通过判断是否出现这 2 个属性值中的一个值，将属性转换为用 0-1 表示的二值属性。

### 10.1.2 特征的重要性

模型中会涉及很多特征，但这些特征对模型的影响程度是不一样的。逻辑回归训练组件 trainer 使用 lazyPrintTrainInfo 打印的训练信息如下：

```
train meta info
{model name: Logistic Regression, num feature: 61}
train importance info
| colName|importanceValue|          colName|weightValue|
|-----|-----|-----|-----|
| status_A14| 0.50808537| foreign_worker| 1.33227696|
| duration| 0.45647352| credit_history_A31| 0.90477436|
| installment_rate| 0.38513890| status_A11| 0.78638287|
```

...	...	...	...
marriage_sex_A94	0.00244852	status_A14	-1.04164671
purpose_A47	0.00000000	savings_A64	-1.30202577
marriage_sex_A95	0.00000000	purpose_A48	-1.31234705

---

```
train convergence info
step:0 loss:0.67720572 gradNorm:0.40702294 learnRate:0.40000000
step:1 loss:0.57520002 gradNorm:0.38580895 learnRate:1.60000000
step:2 loss:0.46073005 gradNorm:0.24350819 learnRate:4.00000000
...
step:15 loss:0.44737146 gradNorm:0.00066504 learnRate:4.00000000
step:16 loss:0.44737055 gradNorm:0.00059793 learnRate:4.00000000
step:17 loss:0.44737029 gradNorm:0.00042434 learnRate:4.00000000
```

在中间部分中，左侧显示的是各特征的重要性（importance）排名；右侧显示的是各特征在模型中权重值（weight）的排名。线性模型特征重要性中的 weight 指标就是逻辑回归模型系数的绝对值；而 importance 指标，会在 weight 的基础上乘以该特征的标准差，这样可消除特征数值变化所带来的影响。

特征重要性排在前三位的是 status\_A14、duration 和 installment\_rate；排在后三位的是 marriage\_sex\_A95、purpose\_A47 和 marriage\_sex\_A94。如果读者想知道更多特征的重要性数值，需要使用 lazyCollectTrainInfo 方法得到训练信息（LinearModelTrainInfo），从训练信息中获取各特征的重要性（importance）数值，进行排序并打印输出。

```
public static void printImportance(String[] colNames, double[] importance) {
    ArrayList <Tuple2 <String, Double>> list = new ArrayList <>();
    for (int i = 0; i < colNames.length; i++) {
        list.add(Tuple2.of(colNames[i], importance[i]));
    }
    Collections.sort(list, new Comparator <Tuple2 <String, Double>>() {
        @Override
        public int compare(Tuple2 <String, Double> o1, Tuple2 <String, Double> o2) {
            return -(o1.f1).compareTo(o2.f1);
        }
    });
    StringBuilder sbd = new StringBuilder();
    for(int i=0; i<list.size();i++){
        sbd.append(i+1).append(" \t")
            .append(list.get(i).f0).append(" \t")
            .append(list.get(i).f1).append("\n");
    }
    System.out.print(sbd.toString());
}
```

得到的输出结果如下。输出结果有 61 条数据，这里只保留了前后 5 条数据。

```
1 status_A14      0.50808536663665
2 duration       0.45647351612984727
3 installment_rate 0.3851388963592921
```

```

4 status_A11          0.3533059193419253
5 savings_A64         0.2736303892679297
...
57 employment_A75    0.00398066943895036
58 purpose_A49       0.0029410949128441635
59 marriage_sex_A94  0.002448517402189116
60 purpose_A47       0.0
61 marriage_sex_A95  0.0

```

至此，我们可以知道哪些特征是重要的，但是特征 status\_A14、status\_A11 都只是标号，其对应的实际含义是什么？我们可以从原始数据的字段说明（见表 10-1）中，查找出经过哑元化操作后的特征属性字段的含义，如表 10-2 所示。

表 10-2 重要特征属性字段的含义

特征	含义
status_A14	现有支票账户的状态：没有支票账户
duration	持续时间（单位：月）
installment_rate	在可支配收入的百分比中，分期付款的比例
status_A11	现有支票账户的状态：…… <0 马克
savings_A64	储蓄账户/债券：…… $\geq 1000$ 马克

关注模型的效果，二分类评估的结果如下：

```

Auc:0.7904 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|

```

## 10.2 减少模型特征的个数

我们在建模的时候，有时候不仅需要很高的精确度、召回率，还要考虑模型特征的可解释性、模型的大小等因素。本节的重点是如何减少特征的个数，选取少数重要的特征建模，且模型仍具有较高的精确度、召回率等指标。

对于逻辑回归模型，可以通过设置逻辑回归训练的 L1 正则项参数，使模型中一些特征的模型系数为 0。其代码与前面类似，只需要在逻辑回归训练组件 trainer 中设置参数 L1（该参数的默认值为 0，即对模型不起作用）。这里我们设置 L1=0.01，相关代码如下：

---

```
LogisticRegressionTrainBatchOp trainer = new LogisticRegressionTrainBatchOp()
```

---

---

```
.setFeatureCols(new_features)
.setLabelCol(LABEL_COL_NAME)
.setL1(0.01);
```

---

读者会问：为什么将正则项参数 L1 设置为 0.01？本书第 20 章中会介绍如何搜索最佳的 L1 值。

将本节的运行结果与前面不设置 L1 的结果进行对比，通过表 10-3 可知，与不设置 L1 相比，设置 L1 后，Auc、Recall 等指标都有不同程度的下降。但是，与此同时，重要性值为 0 的特征数量明显增加。不设置 L1 的时候只有 2 个重要性值为 0 的特征，现在有 23 个重要性值为 0 的特征。特征的重要性值为 0，就意味着该特征的逻辑回归系数为 0，这对分类结果没有影响。从数据中还可看到，在两次训练中，重要性排在前两位的两个特征没有变化，而排在第 3、4 位的两个特征交换了位置。感兴趣的读者，可以尝试继续调大训练参数 L1 的值，这样还会进一步减小非零模型特征的个数。

表 10-3 逻辑回归 L1 参数的不同设置

逻辑回归训练，不设置 L1	逻辑回归训练，设置 L1=0.01
Auc:0.7904	Auc:0.7804
Accuracy:0.78	Accuracy:0.77
Precision:0.5882	Precision:0.5778
Recall:0.566	Recall:0.4906
F1:0.5769	F1:0.5306
LogLoss:0.4799	LogLoss:0.4817
混淆矩阵： [Pred\Real  2  1  ----- --- ---    2  30  21    1  23  126	混淆矩阵： [Pred\Real  2  1  ----- --- ---    2  26  19    1  27  128
特征的重要性： 1 status_A14 0.50808536663665 2 duration 0.45647351612984727 3 installment_rate 0.3851388963592921 4 status_A11 0.3533059193419253 5 savings_A64 0.2736303892679297 ..... 57 employment_A75 0.00398066943895036 58 purpose_A49 0.0029410949128441635	特征的重要性： 1 status_A14 0.5007428413144325 2 duration 0.42143922884968493 3 status_A11 0.2340152292364477 4 installment_rate 0.20285203615332648 5 purpose_A40 0.19901068608203715 ..... 37 employment_A75 0.0021207607321917523 38 status_A13 2.1658292055416037E-16

续表

逻辑回归训练，不设置 L1	逻辑回归训练，设置 L1=0.01
59 marriage_sex_A94 0.002448517402189116	39 credit_history_A32 0.0
60 purpose_A47 0.0	40 credit_history_A33 0.0
61 marriage_sex_A95 0.0	.....
	60 job_A174 0.0
	61 maintenance_num 0.0

## 10.3 离散特征转化

前面通过 SQL SELECT 语句将离散特征转化为数值特征。本节将介绍另外两种算法，同样可以实现离散特征转化功能。

### 10.3.1 独热编码

独热 (One-Hot) 编码的处理过程如下：对于每一个特征，如果该特征有  $m$  个可能值，那么经过独热编码操作后，就会变成  $m$  个二元特征。并且，这些特征互斥，每次只有一个特征可被激活。在基本形式上又可以有一些变化，比如，因为  $m$  个二元特征的和为 1，则由前  $m-1$  个值可以推断出第  $m$  个值，于是出现了 dropLast 选项；对于某些特征可能出现长尾的情况，允许设置离散个数阈值；输出的基本格式是稀疏向量，也可选择只输出索引值；对多个数据列同时进行 One-Hot 编码时，可以选择将各自生成的多个稀疏向量直接拼接成一个稀疏向量。

独热 (One-Hot) 编码组件的相关参数定义如表 10-4 所示。

表 10-4 独热 (One-Hot) 编码组件的参数

名称	描述
selectedCols	(必选) 计算列所对应的列名列表
reservedCols	算法保留列。默认值为 null，即保留所有数据列
outputCols	输出结果列的列名数组。(可选)，默认值为 null，即和输入数据列名数组一致，直接在原列上输出结果
handleInvalid	未知 Token 的处理策略，包括"keep"、"skip"和"error"。默认值为"keep"
encode	编码方式，包括"INDEX"、"VECTOR"、"ASSEMBLED_VECTOR"。默认采用"ASSEMBLED_VECTOR"方式

续表

名称	描述
dropLast	是否删除最后一个元素。默认值为 true
discreteThresholdsArray	离散个数阈值。每一列对应数组中的一个元素
discreteThresholds	离散个数阈值。低于该阈值的离散样本将不会单独成为一个组。默认值为 Integer.MIN_VALUE，即可理解为没有限制

下面使用独热 (One-Hot) 编码组件进行离散特征的转化，然后用逻辑回归模型进行训练。由于要进行独热 (One-Hot) 编码操作，因此需要对目标数据进行整体扫描，得到 One-Hot 模型，之后才能对数据进行编码。为了简化代码，我们会使用 Pipeline 的方式进行调用 (参见 2.4 节)。

具体代码如下。OneHotEncoder 组件设置了 2 个参数：一个参数是所要编码的数据列名称 (SelectedCols)，这里填入了原数据集中所有离散特征的列名；另一个参数是编码输出方式 (Encode)，这里设置的是 Encode.VECTOR，每个特征都会被转化为一个稀疏向量。如果没有设置各输出列的名称，组件会将其输出列名称默认为输入列名称，即输入列的数据会被替换成编码后的结果。经过 OneHotEncoder 组件的处理，我们得到的数据集各特征列是原始数值，或者是编码后得到的稀疏向量。这时，还需要使用向量组装 (VectorAssembler) 组件，将这些数值和向量顺序拼接为一个向量，设置输出结果向量列名称为 VEC\_COL\_NAME。最后使用逻辑回归组件，设置特征向量列为 VEC\_COL\_NAME，再设置标签列、预测结果列、预测详细信息列。建立 Pipeline 后，通过对训练数据 train\_data 使用 fit 方法，得到 PipelineModel，再对测试数据 test\_data 使用 transform 方法进行预测，预测结果连接二分类评估组件。

```
BatchOperator <?> train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE);
BatchOperator <?> test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);

Pipeline pipeline = new Pipeline()
    .add(
        new OneHotEncoder()
            .setSelectedCols(CATEGORY_FEATURE_COL_NAMES)
            .setEncode(Encode.VECTOR)
    )
    .add(
        new VectorAssembler()
            .setSelectedCols(FEATURE_COL_NAMES)
            .setOutputCol(VEC_COL_NAME)
    )
    .add(
        new LogisticRegression()
            .setVectorCol(VEC_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )

```

```

.setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME)
);

pipeline
.fit(train_data)
.transform(test_data)
.link(
  new EvalBinaryClassBatchOp()
    .setPositiveLabelValueString("2")
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME)
    .lazyPrintMetrics()
);
BatchOperator.execute();

```

我们得到的运行结果如下。与前面使用 SQL SELECT 生成新特征所得模型的评估指标大致相同。

```

Auc:0.7904 Accuracy:0.78 Precision:0.5882 Recall:0.566 F1:0.5769 LogLoss:0.4801
|Pred\Real| 2| 1|
+-----+---+---+
|      | 2| 30| 21|
|      2| 30| 21|
|      1| 23|126|

```

使用 One-Hot 编码的好处是，不用对每个离散特征分别编写逻辑代码进行处理，便于我们快速搭建模型。鉴于独热（One-Hot）编码方式仅针对离散特征，我们在使用该编码方式的同时还经常会配合使用 VectorAssembler，将离散特征和连续特征组合起来。

### 10.3.2 特征哈希

特征哈希（FeatureHasher）指的是，通过使用哈希技术生成特征向量，将离散特征或连续特征映射到向量索引。

- 连续特征：将列名称的哈希值映射到特征向量的索引。
- 离散特征：将字符串"column\_name=value"的哈希值映射到向量索引，指标值为 1.0。

FeatureHasher 组件的参数如表 10-5 所示。

表 10-5 FeatureHasher 组件的参数

名称	描述
selectedCols	【必填】计算列对应的列名列表
outputCol	【必填】输出结果列的列名

续表

名称	描述
categoricalCols	【可选】离散特征列。默认选择 string 类型和 boolean 类型的数据作为离散特征
reservedCols	【可选】算法保留列。默认值为 null，保留所有列
numFeatures	【可选】生成向量的维度，默认值为 $2^{18}$ ，即 262 144

FeatureHasher 组件在使用上更加简单，不像 One-Hot 编码组件那样还需要一个向量拼接组件来配合；但 FeatureHasher 为了降低各 Hash 值间的冲突，往往选择较大的向量维度，这使得特征向量更加稀疏。

下面使用 FeatureHasher 组件进行特征变换，其整体流程与前面介绍的 One-Hot 编码组件类似，二者都是首先建立 Pipeline；然后使用与 Pipeline 相关的 fit 和 transform 方法。具体代码如下。这里主要关注 FeatureHasher 组件的参数设置。选择全部特征列作为输入列；设置参数 CategoricalCols，输入离散数据列的名称；输入列为一个稀疏向量，设置输出向量的列名。

```

BatchOperator <?> train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_FILE);
BatchOperator <?> test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + TEST_FILE);

Pipeline pipeline = new Pipeline()
    .add(
        new FeatureHasher()
            .setSelectedCols(FEATURE_COL_NAMES)
            .setCategoricalCols(CATEGORY_FEATURE_COL_NAMES)
            .setOutputCol(VEC_COL_NAME)
    )
    .add(
        new LogisticRegression()
            .setVectorCol(VEC_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME)
    );
pipeline
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("2")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PREDICTION_DETAIL_COL_NAME)
            .lazyPrintMetrics()
    );
BatchOperator.execute();

```

计算得到的评估结果如下，与前面使用 One-Hot 方法所得的评估结果相似：

Auc:0.7904 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

# 11

## 构造新特征

之前我们一直在实验如何使用已有的特征，包括特征哑元化、特征哈希，以及选择重要的特征。在本章中我们对“特征”的理解将更进一步，学习如何构造新特征。

本章的数据来源于2014年的阿里巴巴公司大数据竞赛第一赛季，下载地址为链接11-1。在天猫平台中，每天都会有数千万用户通过在各品牌下搜索来发现自己喜欢的商品。品牌是连接消费者与商品的最重要纽带。本章的数据为用户4个月在天猫平台中的行为日志，包含4个字段。这些字段的具体说明详见表11-1

表11-1 字段的具体说明

字段	字段说明	提取说明
user_id	用户标记	抽样&字段加密
time	行为时间	精度到天级别&隐藏年份
action_type	用户对品牌的行 <b>为</b> 类型	包括点击、购买、加入购物车、收藏4种行为 (点击:0, 购买:1, 收藏:2, 加入购物车:3)
brand_id	品牌ID	抽样&字段加密

用户对任意商品的行为都会映射为一行数据。其中，所有商品的ID都已汇总为商品对应的品牌ID。用户和品牌都分别做了一定程度的数据抽样，且数字ID均做了加密。所有行为的时间都精确到天级别（隐藏年份）。

本章将根据用户在天猫平台中的行为日志，建立用户的品牌偏好，并预测他们将来对该品牌旗下各商品的购买行为。

我们希望预测的品牌准确率越高越好，也希望覆盖的用户和品牌越多越好，所以用最常用的准确率与召回率作为排行榜的指标。

准确率（Precision）：

$$\text{Precision} = \frac{\sum_{i=1}^N \text{hitBrands}_i}{\sum_{i=1}^N \text{pBrands}_i}$$

其中， $N$ 为预测的用户数； $\text{pBrands}_i$ 的含义是，预测用户*i*会购买的商品品牌列表个数； $\text{hitBrands}_i$ 的含义是，预测用户*i*可能购买的商品品牌列表与用户*i*真实购买的商品品牌交集的个数。

召回率（Recall）或称灵敏度（Sensitivity）：

$$\text{Recall} = \frac{\sum_{i=1}^M \text{hitBrands}_i}{\sum_{i=1}^M \text{bBrands}_i}$$

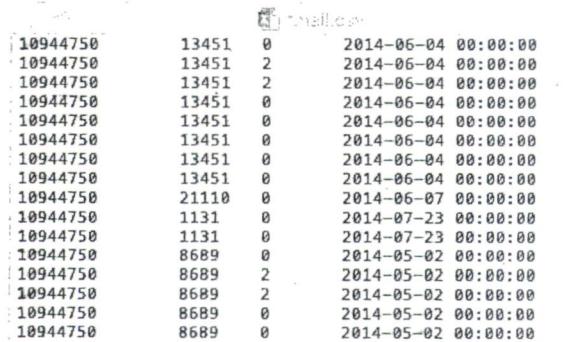
其中， $M$ 为实际产生成交的用户数； $\text{bBrands}_i$ 为用户*i*真实购买的商品品牌个数； $\text{hitBrands}_i$ 为预测的品牌列表与用户*i*真实购买的商品品牌交集的个数。

最后我们用 $F_1$ -Score 来拟合准确率与召回率，并且该大赛最终的比赛成绩排名以 $F_1$ 得分为为准。

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

## 11.1 数据探索

使用文本编辑器，打开下载的数据文件，如图 11-1 所示。该文件共有 4 列，每列均用 Tab 键作为分隔。由于 Tab 键具有对齐作用，因此这些数据看起来比较整齐。



10944750	13451	0	2014-06-04 00:00:00
10944750	13451	2	2014-06-04 00:00:00
10944750	13451	2	2014-06-04 00:00:00
10944750	13451	0	2014-06-04 00:00:00
10944750	13451	0	2014-06-04 00:00:00
10944750	13451	0	2014-06-04 00:00:00
10944750	13451	0	2014-06-04 00:00:00
10944750	13451	0	2014-06-04 00:00:00
10944750	21110	0	2014-06-07 00:00:00
10944750	1131	0	2014-07-23 00:00:00
10944750	1131	0	2014-07-23 00:00:00
10944750	8689	0	2014-05-02 00:00:00
10944750	8689	2	2014-05-02 00:00:00
10944750	8689	2	2014-05-02 00:00:00
10944750	8689	0	2014-05-02 00:00:00
10944750	8689	0	2014-05-02 00:00:00

图 11-1

我们可以使用 CSV 格式的组件进行读取，因为该数据在本章的示例中会被多次用到。我们定义函数 getSource() 如下：

```
static CsvSourceBatchOp getSource() {
    return new CsvSourceBatchOp()
        .setFilePath(DATA_DIR + ORIGIN_FILE)
        .setSchemaStr("user_id long, brand_id long, type int, ts timestamp")
        .setFieldDelimiter("\t");
}
```

对原始数据进行打印、统计，以及计算相似系数，代码如下：

```
BatchOperator<?> source = getSource();
source.lazyPrint(10, "origin file");
source.lazyPrintStatistics("stat of origin file");
source.link(
    new CorrelationBatchOp()
        .setSelectedCols("user_id", "brand_id", "type")
        .lazyPrintCorrelation()
);
});
```

前 10 条数据打印显示如下：

```
Origin file
user_id|brand_id|type|ts
-----|-----|---|-
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|2|2014-06-04 00:00:00.0
10944750|13451|2|2014-06-04 00:00:00.0
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|0|2014-06-04 00:00:00.0
10944750|21110|0|2014-06-07 00:00:00.0
10944750|1131|0|2014-07-23 00:00:00.0
```

各列的基本统计指标如下：

stat of origin file							
Summary:							
colName	count	missing	sum	mean	variance	min	max
user_id	182880	0	1083663976000	5925546.6754	12698777247108.18	19500	12417500
brand_id	182880	0	2595405857	14191.8518	71953590.661	11	29552
type	182880	0	9851	0.0539	0.0692	0	3
ts	182880	0	NaN	NaN	NaN	NaN	NaN

在此可以看到，数据的总数为 182 880 条，没有缺失值；type 的变动范围为 [0, 3]；ts 列没有什么统计指标。我们希望知道 ts 的开始时间和结束时间，后面会用 SQL 语句深入分析。

相关系数结果如下所示，显然，各字段不相关：

Correlation:

colName	user_id	brand_id	type
user_id	1	0.0054	0.004
brand_id	0.0054	1	0.0035
type	0.004	0.0035	1

下面，我们使用 SQL 语句获取更多信息，代码如下。首先获取购买行为发生的时间范围，之后获取 type 字段各取值的分布情况。

---

```
source.select("min(ts) AS min_ts, max(ts) AS max_ts").lazyPrint(-1);
source.groupBy("type", "type, COUNT(*) AS cnt").lazyPrint(-1);
```

---

获得数据集中行为的开始时间、结束时间如下：

min_ts	max_ts
2014-04-15 00:00:00.0	2014-08-15 00:00:00.0

type 字段共有 4 个不同的值（点击：0；购买：1；收藏：2；加入购物车：3），它们各自出现的次数如下，显然，点击行为发生得最多。

type	cnt
0	174539
1	6984
2	1204
3	153

## 11.2 思路

预测用户在未来一个月内对该品牌旗下各商品的购买行为，即判断某个用户在下一个月是否发生购买某品牌商品的行为，也就是机器学习中典型的二分类预测问题。

解决的关键是如何得到关于用户和品牌的各种特征，并训练出二分类模型。

其整体流程主要是构造特征和标签，使用逻辑回归（LR）和随机森林（RF）模型进行训练、预测和评估。

### 11.2.1 用户和品牌的各种特征

某个用户是否购买某个品牌旗下商品的影响因素（特征）有哪些呢？

首先了解用户对品牌的关注度，比如，用户是否点击过该品牌的商品，是否有该品牌商品的购买行为，是否收藏了该品牌商品，以及是否将该品牌商品加入过购物车。而在这些因素中，用户的关注行为离现在越近，其即将购买该品牌商品的可能性就越大，所以我们要依次关注最近 3 天、最近 1 周、最近 1 个月、最近 2 个月、最近 3 个月以及有用户记录以来的情况，于是有了如下的一些特征：

- 最近 3 天的点击数、购买数、收藏数和加入购物车的次数。
- 最近 1 周的点击数、购买数、收藏数和加入购物车的次数。
- 最近 1 个月的点击数、购买数、收藏数和加入购物车的次数。
- 最近 2 个月的点击数、购买数、收藏数和加入购物车的次数。
- 最近 3 个月的点击数、购买数、收藏数和加入购物车的次数。
- 全部点击数、购买数、收藏数和加入购物车的次数。

有了按时间段细分的关注次数还不够，还需要知道该数值的变化率，以刻画该关注的持续程度。我们还可以构造如下特征：

- 最近 3 天点击数的变化率（最近 3 天的点击数 / 最近 4~6 天的点击数）、购买数的变化率、收藏数的变化率、加入购物车次数的变化率。
- 最近 1 周点击数的变化率（最近 1 周的点击数 / 上周的点击数）、购买数的变化率、收藏数的变化率、加入购物车次数的变化率。
- 最近 1 个月点击数的变化率（最近 1 个月的点击数 / 上个月的点击数）、购买数的变化率、收藏数的变化率、加入购物车次数的变化率。

如果用户对该品牌商品曾有过购买行为，我们希望了解，通过多少次点击产生了一次用户购买行为、通过多少次收藏转化为一次用户购买行为，即购买转化率（简称“转化率”）。构造特征如下：

- 最近 3 天的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 周的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 个月的点击转化率、收藏转化率、加入购物车的转化率。
- 整体的点击转化率、收藏转化率、加入购物车的转化率。

其次，我们将注意力放在用户上。需要构造特征，将用户的特点表现出来，重点是了解该用户对所关注的所有品牌的总体行为。用户最近对所有品牌的关注度，有如下特征：

- 最近 3 天的点击数、购买数、收藏数和加入购物车的次数。
- 最近 1 周的点击数、购买数、收藏数和加入购物车的次数。
- 最近 1 个月的点击数、购买数、收藏数和加入购物车的次数。
- 最近 2 个月的点击数、购买数、收藏数和加入购物车的次数。
- 最近 3 个月的点击数、购买数、收藏数和加入购物车的次数。
- 全部点击数、购买数、收藏数和加入购物车的次数。

每个用户都有自己的特点，有的人点击次数很多，却很少购买；有的人关注某商品很久，才会下单购买。这些特点可以用购买转化率来刻画：

- 最近 3 天的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 周的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 个月的点击转化率、收藏转化率、加入购物车的转化率。
- 整体的点击转化率、收藏转化率、加入购物车的转化率。

最后，单独看品牌这个因素的影响。有的热门品牌的关注度很高。我们更关心其近期的情况，包括如下特征：

- 最近 3 天的被点击数、被购买数、被收藏数和被加入购物车的次数。
- 最近 1 周的被点击数、被购买数、被收藏数和被加入购物车的次数。
- 最近 1 个月的被点击数、被购买数、被收藏数和被加入购物车的次数。
- 最近 2 个月的被点击数、被购买数、被收藏数和被加入购物车的次数。
- 最近 3 个月的被点击数、被购买数、被收藏数和被加入购物车的次数。
- 全部被点击数、被购买数、被收藏数和被加入购物车的次数。

有的品牌的受众较少。虽然其没有很高的关注度，但是购买量不少，可以用购买转化率来描述这些特征：

- 最近 3 天的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 周的点击转化率、收藏转化率、加入购物车的转化率。
- 最近 1 个月的点击转化率、收藏转化率、加入购物车的转化率。
- 整体的点击转化率、收藏转化率、加入购物车的转化率。

综上，某个用户是否会购买某品牌商品的特征，由刻画该用户对该品牌商品关注的各种特征、描述该用户的特征及描述该品牌商品的特征共同构成。

## 11.2.2 二分类模型训练

本节的重点是如何构造训练集。训练集中的每一条记录均为某个用户针对某个品牌的数据，

这些记录会包含前面提到的那些特征，但同时还要有一个标签项，表示在这个特征的前提下是否会发生购买行为。

我们知道的数据只是一些带发生时间的行为，比如，开始时间为 2014-4-15，结束时间为 2014-8-15，一共 4 个月。这怎么和特征、标签联系起来呢？

我们需要从一个新的角度来看数据，以 2014-07-16 00:00:00 为界将数据分为两段。由 2014-4-15 到 2014-7-15 这 3 个月内发生的行为，我们可以得到用户与品牌的特征，比如，最近 3 天的点击数、最近 1 周点击数的购买转化率等；而某用户是否会在下个月发生购买某品牌商品的行为，我们可以通过查询其下个月的行为来获知。

这样我们就得到了训练集。对于二分类模型，我们可以有很多选择，比如逻辑回归、随机森林等。我们可以根据二分类问题的特点来选择这些模型。

## 11.3 计算训练集

由于整个构造特征、标签、模型训练和评估的流程比较长，首先以时间 2014-07-16 00:00:00 为界，将数据分为两段，该时间点以前的数据可用来构造特征，该时间点之后的数据可用来统计出在本段时间（2014-07-16 00:00:00 之后的一个月）内哪些用户、品牌有购买行为，用此统计数据来构造标签。

### 11.3.1 原始数据划分

要对数据按时间 2014-07-16 00:00:00 进行划分，相应的代码如下：

---

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

BatchOperator t1 = source.filter("ts < CAST('2014-07-16 00:00:00' AS TIMESTAMP)");
BatchOperator t2 = source.filter("ts >= CAST('2014-07-16 00:00:00' AS TIMESTAMP)");

t1.lazyPrint(3, "[ ts < '2014-07-16 00:00:00' ]")
    .lazyPrintStatistics();

t2.lazyPrint(3, "[ ts >= '2014-07-16 00:00:00' ]")
    .lazyPrintStatistics();

BatchOperator.execute();
```

---

运行结果如下，按时间 2014-07-16 00:00:00 划分的两个数据集，包含的数据条数分别为 131 720 和 51 160。

```
[ ts < '2014-07-16 00:00:00' ]
user_id|brand_id|type|ts
-----|-----|---|-
10944750|13451|0|2014-06-04 00:00:00.0
10944750|13451|2|2014-06-04 00:00:00.0
10944750|13451|2|2014-06-04 00:00:00.0
Summary:
+ colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+-----+-----+
| user_id | 131720 | 0 | 776870757500 | 5897895.2133 | 12780375514149.984 | 19500 | 12417500 |
| brand_id | 131720 | 0 | 1854552448 | 14079.5054 | 72351905.181 | 11 | 29552 |
| type | 131720 | 0 | 6819 | 0.0518 | 0.0649 | 0 | 3 |
| ts | 131720 | 0 | NaN | NaN | NaN | NaN | NaN |
```

```
[ ts >= '2014-07-16 00:00:00' ]
user_id|brand_id|type|ts
-----|-----|---|-
10944750|1131|0|2014-07-23 00:00:00.0
10944750|1131|0|2014-07-23 00:00:00.0
10944750|24955|0|2014-07-26 00:00:00.0
Summary:
+ colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+-----+-----+
| user_id | 51160 | 0 | 306793218500 | 5996740.002 | 12481897307141.959 | 19500 | 12417500 |
| brand_id | 51160 | 0 | 740853409 | 14481.1065 | 70813290.0478 | 15 | 29552 |
| type | 51160 | 0 | 3032 | 0.0593 | 0.08 | 0 | 3 |
| ts | 51160 | 0 | NaN | NaN | NaN | NaN | NaN |
```

### 11.3.2 计算特征

我们所要计算的特征大致分为三类：某时间段内的行为发生数、转化率和变化率。后两类特征可以由相应的行为发生数相除得出，所以特征计算可以分为两个阶段：

(1) 计算出各时间段内点击、购买、收藏和加入购物车的次数。

(2) 计算出相应的转化率和变化率。

在计算次数的时候，我们为了使计算过程清晰，并且在计算的时候有较高的效率，会根据每个用户对品牌的行为类型和发生时间，分为以下情况：是否为最近 3 天发生的点击行为，是否为最近 3 天发生的购买行为，……，是否为最近 1 个月发生的收藏行为……如果判断结果为“是”，该计数项就标识为 1；否则标识为 0。进行标识后，可以将用户和品牌的相同数据聚在同一组，将该组中所有记录对应的“是否为最近 3 天发生的点击行为”列的标识值相加，这样可得到“最近 3 天发生的点击次数”。

## 1. 数据预处理标识

预处理环节的作用是，为后面的特征生成做准备，比如标识了近 1 个月内、近 1 周内是否有点击、购买等行为。将划分出来构造特征的数据集，生成一些新的字段，按如下规则命名：

- (1) 以 is 开头，表明此列是用来标识行为的“是/否”的。
- (2) 之后是行为信息，包括点击 (click)、购买 (buy)、收藏 (collect) 和加入购物车 (cart) 等信息。
- (3) 最后为时间段信息，比如，1m 为最近 1 个月，3m 为最近 3 个月，m2nd 为倒数第 2 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- (4) 各部分之间用下画线 “\_” 连接。

实现过程分为两步：第 1 步是，计算出每条样本的发生时间距离"2014-07-16 00:00:00"的天数，便于后面统计最近 3 天、最近 1 个月等特征；第 2 步是详细计算的过程。相关代码如下：

```
String clausePreProc = "user_id, brand_id, type, ts, past_days,"  
    + "case when type=0 then 1 else 0 end AS is_click,"  
    + "case when type=1 then 1 else 0 end AS is_buy,"  
    + "case when type=2 then 1 else 0 end AS is_collect,"  
    + "case when type=3 then 1 else 0 end AS is_cart,"  
    + "case when type=0 and past_days<=30 then 1 else 0 end AS is_click_1m,"  
    + "case when type=1 and past_days<=30 then 1 else 0 end AS is_buy_1m,"  
    ...  
    + "case when type=1 and past_days>14 and past_days<=21 then 1 else 0 end AS is_buy_w3th,"  
    + "case when type=2 and past_days>14 and past_days<=21 then 1 else 0 end AS is_collect_w3th,"  
    + "case when type=3 and past_days>14 and past_days<=21 then 1 else 0 end AS is_cart_w3th";  
  
BatchOperator t1_preproc = t1  
    .select("user_id, brand_id, type, ts, "  
        + "TIMESTAMPDIFF(DAY, ts, TIMESTAMP '2014-07-16 00:00:00') AS past_days")  
    .select(clausePreProc);
```

## 2. 用户-品牌联合特征

我们所关心的是在前 3 个月的行为数据中出现的用户-品牌对。对于每个用户-品牌对，可根据该用户在前 3 个月针对该品牌的 所有行为数据，汇总统计、计算出如下的特征：

- (1) 计算出各时间段内点击、购买、收藏和加入购物车的次数。
- (2) 计算出相应的转化率和变化率。

我们先介绍第 1 部分的各种“次数”。在前面计算出来的标识数据表基础上，根据 user\_id 和 brand\_id 进行分组 (group)，使用 SQL 中的合计函数 SUM 计算次数。在得到的计数结果数据表中，产生的数据列名满足如下规则：

- (1) 以 cnt 开头，表明此列为计数信息。

(2)之后是行为信息，包括点击 (click)、购买 (buy)、收藏 (collect) 和加入购物车 (cart) 等信息。

(3) 最后为时间段信息，比如，1m 为最近 1 个月，3m 为最近 3 个月，m2nd 为倒数第 2 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。

(4) 各部分之间用下画线 “\_” 连接。

详细的代码如下：

```
String clauseUserBrand = "user_id, brand_id, SUM(is_click) as cnt_click, SUM(is_buy) as cnt_buy, "
+ "SUM(is_collect) as cnt_collect, SUM(is_cart) as cnt_cart, "
+ "SUM(is_click_1m) as cnt_click_1m, SUM(is_buy_1m) as cnt_buy_1m, "
+ "SUM(is_collect_1m) as cnt_collect_1m, SUM(is_cart_1m) as cnt_cart_1m, "
+ "SUM(is_click_2m) as cnt_click_2m, SUM(is_buy_2m) as cnt_buy_2m, "
+ "SUM(is_collect_2m) as cnt_collect_2m, SUM(is_cart_2m) as cnt_cart_2m, "
+ "SUM(is_click_3m) as cnt_click_3m, SUM(is_buy_3m) as cnt_buy_3m, "
+ "SUM(is_collect_3m) as cnt_collect_3m, SUM(is_cart_3m) as cnt_cart_3m, "
+ "SUM(is_click_m2nd) as cnt_click_m2nd, SUM(is_buy_m2nd) as cnt_buy_m2nd, "
+ "SUM(is_collect_m2nd) as cnt_collect_m2nd, SUM(is_cart_m2nd) as cnt_cart_m2nd, "
+ "SUM(is_click_m3th) as cnt_click_m3th, SUM(is_buy_m3th) as cnt_buy_m3th, "
+ "SUM(is_collect_m3th) as cnt_collect_m3th, SUM(is_cart_m3th) as cnt_cart_m3th, "
+ "SUM(is_click_3d) as cnt_click_3d, SUM(is_buy_3d) as cnt_buy_3d, "
+ "SUM(is_collect_3d) as cnt_collect_3d, SUM(is_cart_3d) as cnt_cart_3d, "
+ "SUM(is_click_3d2nd) as cnt_click_3d2nd, SUM(is_buy_3d2nd) as cnt_buy_3d2nd, "
+ "SUM(is_collect_3d2nd) as cnt_collect_3d2nd, SUM(is_cart_3d2nd) as cnt_cart_3d2nd, "
+ "SUM(is_click_3d3th) as cnt_click_3d3th, SUM(is_buy_3d3th) as cnt_buy_3d3th, "
+ "SUM(is_collect_3d3th) as cnt_collect_3d3th, SUM(is_cart_3d3th) as cnt_cart_3d3th, "
+ "SUM(is_click_1w) as cnt_click_1w, SUM(is_buy_1w) as cnt_buy_1w, "
+ "SUM(is_collect_1w) as cnt_collect_1w, SUM(is_cart_1w) as cnt_cart_1w, "
+ "SUM(is_click_w2nd) as cnt_click_w2nd, SUM(is_buy_w2nd) as cnt_buy_w2nd, "
+ "SUM(is_collect_w2nd) as cnt_collect_w2nd, SUM(is_cart_w2nd) as cnt_cart_w2nd, "
+ "SUM(is_click_w3th) as cnt_click_w3th, SUM(is_buy_w3th) as cnt_buy_w3th, "
+ "SUM(is_collect_w3th) as cnt_collect_w3th, SUM(is_cart_w3th) as cnt_cart_w3th";
```

---

```
BatchOperator t1_userbrand = t1_preproc.groupBy("user_id, brand_id", clauseUserBrand);
```

有了计数信息后，下面可以进一步计算其转化率和变化率。我们将刚才计算的计数数据表作为输入，对其中的每一条记录进行变换。由于最终的特征中仍包括这些计算信息，因此在变换结果列中包括了原始的计数列，并将其对应的计算表达式写成该列的列名。新产生的转化率和变化率列，满足如下规则：

- 以 rt (即 rate 的缩写) 开头，表明此列为比率信息。
- 对于变化率，中间为行为信息，包括点击 (click)、购买 (buy)、收藏 (collect) 和加入购物车 (cart) 等信息。
- 对于转化率，中间为行为信息加上 “2buy”，包括点击转化率 (click2buy)、购买转化率 (buy2buy)、收藏转化率 (collect2buy) 和加入购物车的转化率 (cart2buy)。

- 最后为时间段信息，比如 1m 为最近 1 个月，3m 为最近 3 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- 各部分之间用下画线 “\_” 连接。

详细的代码如下：

```
String clauseUserBrand_Rate = "user_id,brand_id,"
+ "cnt_click,cnt_buy,cnt_collect,cnt_cart,"
+ "cnt_click_1m,cnt_buy_1m,cnt_collect_1m,cnt_cart_1m,"
+ "cnt_click_2m,cnt_buy_2m,cnt_collect_2m,cnt_cart_2m,"
+ "cnt_click_3m,cnt_buy_3m,cnt_collect_3m,cnt_cart_3m,"
...
+ "case when cnt_buy_1m=0 then 0.0 when cnt_buy_1m>=30.0*cnt_buy_m2nd then 30.0 else "
+ "cnt_buy_1m*1.0/cnt_buy_m2nd end AS rt_buy_1m,"
+ "case when cnt_collect_1m=0 then 0.0 when cnt_collect_1m>=30.0*cnt_collect_m2nd then 30.0
else "
+ "cnt_collect_1m*1.0/cnt_collect_m2nd end AS rt_collect_1m,"
+ "case when cnt_cart_1m=0 then 0.0 when cnt_cart_1m>=50.0*cnt_cart_m2nd then 50.0 else "
+ "cnt_cart_1m*1.0/cnt_cart_m2nd end AS rt_cart_1m";
t1_userbrand = t1_userbrand.select(clauseUserBrand_Rate);
```

### 3. 用户特征

本节的重点是计算出刻画用户的特征，用户范围为前 3 个月的行为数据中出现的全部用户。可根据每个用户在前 3 个月的所有行为数据，汇总统计、计算出如下的特征：

- (1) 计算出各时间段内点击、购买、收藏和加入购物车的次数。
- (2) 计算出相应的购买转化率。

与前面的计算方法相似，先计算第 1 部分的各种“次数”。在前面计算出来的标识数据表基础上，根据 user\_id 进行分组 (group)，使用 SQL 中的合计函数 SUM 计算次数。产生的数据列名满足如下规则：

- (1) 以 user\_cnt 开头，表明此列为用户计数信息。
- (2) 之后是行为信息，包括点击 (click)、购买 (buy)、收藏 (collect) 和加入购物车 (cart) 等信息。
- (3) 最后为时间段信息，比如，1m 为最近 1 个月，3m 为最近 3 个月，m2nd 为倒数第 2 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- (4) 各部分之间用下画线 “\_” 连接。

详细的代码如下：

```
String clauseUser = "user_id,
+ "SUM(is_click) as user_cnt_click, SUM(is_buy) as user_cnt_buy,
+ "SUM(is_collect) as user_cnt_collect, SUM(is_cart) as user_cnt_cart,
+ "SUM(is_click_1m) as user_cnt_click_1m, SUM(is_buy_1m) as user_cnt_buy_1m,"
```

```

+ "SUM(is_collect_1m) as user_cnt_collect_1m, SUM(is_cart_1m) as user_cnt_cart_1m, "
+ "SUM(is_click_2m) as user_cnt_click_2m, SUM(is_buy_2m) as user_cnt_buy_2m, "
+ "SUM(is_collect_2m) as user_cnt_collect_2m, SUM(is_cart_2m) as user_cnt_cart_2m, "
+ "SUM(is_click_3m) as user_cnt_click_3m, SUM(is_buy_3m) as user_cnt_buy_3m, "
+ "SUM(is_collect_3m) as user_cnt_collect_3m, SUM(is_cart_3m) as user_cnt_cart_3m, "
+ "SUM(is_click_m2nd) as user_cnt_click_m2nd, SUM(is_buy_m2nd) as user_cnt_buy_m2nd, "
+ "SUM(is_collect_m2nd) as user_cnt_collect_m2nd, SUM(is_cart_m2nd) as user_cnt_cart_m2nd, "
+ "SUM(is_click_m3th) as user_cnt_click_m3th, SUM(is_buy_m3th) as user_cnt_buy_m3th, "
+ "SUM(is_collect_m3th) as user_cnt_collect_m3th, SUM(is_cart_m3th) as user_cnt_cart_m3th, "
+ "SUM(is_click_3d) as user_cnt_click_3d, SUM(is_buy_3d) as user_cnt_buy_3d, "
+ "SUM(is_collect_3d) as user_cnt_collect_3d, SUM(is_cart_3d) as user_cnt_cart_3d, "
+ "SUM(is_click_3d2nd) as user_cnt_click_3d2nd, SUM(is_buy_3d2nd) as user_cnt_buy_3d2nd, "
+ "SUM(is_collect_3d2nd) as user_cnt_collect_3d2nd, SUM(is_cart_3d2nd) as user_cnt_cart_3d2nd, "
+ "SUM(is_click_3d3th) as user_cnt_click_3d3th, SUM(is_buy_3d3th) as user_cnt_buy_3d3th, "
+ "SUM(is_collect_3d3th) as user_cnt_collect_3d3th, SUM(is_cart_3d3th) as user_cnt_cart_3d3th, "
+ "SUM(is_click_1w) as user_cnt_click_1w, SUM(is_buy_1w) as user_cnt_buy_1w, "
+ "SUM(is_collect_1w) as user_cnt_collect_1w, SUM(is_cart_1w) as user_cnt_cart_1w, "
+ "SUM(is_click_w2nd) as user_cnt_click_w2nd, SUM(is_buy_w2nd) as user_cnt_buy_w2nd, "
+ "SUM(is_collect_w2nd) as user_cnt_collect_w2nd, SUM(is_cart_w2nd) as user_cnt_cart_w2nd, "
+ "SUM(is_click_w3th) as user_cnt_click_w3th, SUM(is_buy_w3th) as user_cnt_buy_w3th, "
+ "SUM(is_collect_w3th) as user_cnt_collect_w3th, SUM(is_cart_w3th) as user_cnt_cart_w3th";

```

```
BatchOperator t1_user = t1_prepoc.groupBy("user_id", clauseUser);
```

有了计数信息后，再进一步计算其转化率。我们将刚才计算的计数数据表作为输入，对其中的每一条记录进行变换。由于最终的特征中仍包括这些计算信息，因此在变换结果列中包括了原始的计数列，并将其对应的计算表达式写成该列的列名。新产生的购买转化率列满足如下规则：

- 以 user\_rt 开头，表明此列为用户的比率信息。
- 中间为行为信息加上“2buy”，包括点击转化率(click2buy)、购买转化率(buy2buy)、收藏转化率(collect2buy)和加入购物车的转化率(cart2buy)等信息。
- 最后为时间段信息，比如 1m 为最近 1 个月，3m 为最近 3 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- 各部分之间用下画线“\_”连接。

详细的代码如下：

```

String clauseUser_Rate = "user_id AS user_id4join, "
+ "user_cnt_click,user_cnt_buy,user_cnt_collect,user_cnt_cart, "
+ "user_cnt_click_1m,user_cnt_buy_1m,user_cnt_collect_1m,user_cnt_cart_1m, "
+ "user_cnt_click_2m,user_cnt_buy_2m,user_cnt_collect_2m,user_cnt_cart_2m, "
+ "user_cnt_click_3m,user_cnt_buy_3m,user_cnt_collect_3m,user_cnt_cart_3m, "
+ "user_cnt_click_m2nd,user_cnt_buy_m2nd,user_cnt_collect_m2nd,user_cnt_cart_m2nd, "
+ "user_cnt_click_m3th,user_cnt_buy_m3th,user_cnt_collect_m3th,user_cnt_cart_m3th, "
+ "user_cnt_click_3d,user_cnt_buy_3d,user_cnt_collect_3d,user_cnt_cart_3d, "
+ "user_cnt_click_3d2nd,user_cnt_buy_3d2nd,user_cnt_collect_3d2nd,user_cnt_cart_3d2nd, "
+ "user_cnt_click_3d3th,user_cnt_buy_3d3th,user_cnt_collect_3d3th,user_cnt_cart_3d3th, "
+ "user_cnt_click_1w,user_cnt_buy_1w,user_cnt_collect_1w,user_cnt_cart_1w, "
+ "user_cnt_click_w2nd,user_cnt_buy_w2nd,user_cnt_collect_w2nd,user_cnt_cart_w2nd, "
+ "user_cnt_click_w3th,user_cnt_buy_w3th,user_cnt_collect_w3th,user_cnt_cart_w3th, "

```

```

+ "case when user_cnt_buy>user_cnt_click then 1.0 when user_cnt_buy=0 then 0.0 else "
+ "user_cnt_buy*1.0/user_cnt_click end AS user_rt_click2buy,"
+ "case when user_cnt_buy>user_cnt_collect then 1.0 when user_cnt_buy=0 then 0.0 else "
+ "user_cnt_buy*1.0/user_cnt_collect end AS user_rt_collect2buy,"
+ "case when user_cnt_buy>user_cnt_cart then 1.0 when user_cnt_buy=0 then 0.0 else "
+ "user_cnt_buy*1.0/user_cnt_cart end AS user_rt_cart2buy,"
+ "case when user_cnt_buy_3d>user_cnt_click_3d then 1.0 when user_cnt_buy_3d=0 then 0.0 else "
+ "user_cnt_buy_3d*1.0/user_cnt_click_3d end AS user_rt_click2buy_3d,"
+ "case when user_cnt_buy_3d>user_cnt_collect_3d then 1.0 when user_cnt_buy_3d=0 then 0.0 else "
+ "user_cnt_buy_3d*1.0/user_cnt_collect_3d end AS user_rt_collect2buy_3d,"
+ "case when user_cnt_buy_3d>user_cnt_cart_3d then 1.0 when user_cnt_buy_3d=0 then 0.0 else "
+ "user_cnt_buy_3d*1.0/user_cnt_cart_3d end AS user_rt_cart2buy_3d,"
+ "case when user_cnt_buy_1w>user_cnt_click_1w then 1.0 when user_cnt_buy_1w=0 then 0.0 else "
+ "user_cnt_buy_1w*1.0/user_cnt_click_1w end AS user_rt_click2buy_1w,"
+ "case when user_cnt_buy_1w>user_cnt_collect_1w then 1.0 when user_cnt_buy_1w=0 then 0.0 else "
+ "user_cnt_buy_1w*1.0/user_cnt_collect_1w end AS user_rt_collect2buy_1w,"
+ "case when user_cnt_buy_1w>user_cnt_cart_1w then 1.0 when user_cnt_buy_1w=0 then 0.0 else "
+ "user_cnt_buy_1w*1.0/user_cnt_cart_1w end AS user_rt_cart2buy_1w,"
+ "case when user_cnt_buy_1m>user_cnt_click_1m then 1.0 when user_cnt_buy_1m=0 then 0.0 else "
+ "user_cnt_buy_1m*1.0/user_cnt_click_1m end AS user_rt_click2buy_1m,"
+ "case when user_cnt_buy_1m>user_cnt_collect_1m then 1.0 when user_cnt_buy_1m=0 then 0.0 else "
+ "user_cnt_buy_1m*1.0/user_cnt_collect_1m end AS user_rt_collect2buy_1m,"
+ "case when user_cnt_buy_1m>user_cnt_cart_1m then 1.0 when user_cnt_buy_1m=0 then 0.0 else "
+ "user_cnt_buy_1m*1.0/user_cnt_cart_1m end AS user_rt_cart2buy_1m";

```

---

```
t1_user = t1_user.select(clauseUser_Rate);
```

#### 4. 品牌特征

本节重点计算描述品牌的特征，品牌选择范围为前 3 个月的行为数据中出现的全部品牌。对于每个品牌，可根据前 3 个月中所有用户对该品牌商品的行为数据，汇总统计、计算出如下的特征：

- (1) 计算出各时间段内该品牌的商品被点击、被购买、被收藏和被加入购物车的次数。
- (2) 计算出相应的购买转化率。

与计算用户特征的方法相似，先计算第 1 部分的各种“次数”。在前面计算出来的标识数据表基础上，根据 brand\_id 进行分组（group），使用 SQL 中的合计函数 SUM 计算次数。产生的数据列名满足如下规则：

- (1) 以 brand\_cnt 开头，表明此列为用户计数信息。
- (2) 之后是行为信息，包括点击（click）、购买（buy）、收藏（collect）和加入购物车（cart）等信息。
- (3) 最后为时间段信息，比如，1m 为最近 1 个月，3m 为最近 3 个月，m2nd 为倒数第 2 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- (4) 各部分之间用下画线“\_”连接。

详细的代码如下：

```

String clauseBrand = "brand_id, "
+ "SUM(is_click) as brand_cnt_click, SUM(is_buy) as brand_cnt_buy, "
+ "SUM(is_collect) as brand_cnt_collect, SUM(is_cart) as brand_cnt_cart, "
+ "SUM(is_click_1m) as brand_cnt_click_1m, SUM(is_buy_1m) as brand_cnt_buy_1m, "
+ "SUM(is_collect_1m) as brand_cnt_collect_1m, SUM(is_cart_1m) as brand_cnt_cart_1m, "
+ "SUM(is_click_2m) as brand_cnt_click_2m, SUM(is_buy_2m) as brand_cnt_buy_2m, "
+ "SUM(is_collect_2m) as brand_cnt_collect_2m, SUM(is_cart_2m) as brand_cnt_cart_2m, "
+ "SUM(is_click_3m) as brand_cnt_click_3m, SUM(is_buy_3m) as brand_cnt_buy_3m, "
+ "SUM(is_collect_3m) as brand_cnt_collect_3m, SUM(is_cart_3m) as brand_cnt_cart_3m, "
+ "SUM(is_click_m2nd) as brand_cnt_click_m2nd, SUM(is_buy_m2nd) as brand_cnt_buy_m2nd, "
+ "SUM(is_collect_m2nd) as brand_cnt_collect_m2nd, SUM(is_cart_m2nd) as brand_cnt_cart_m2nd, "
+ "SUM(is_click_m3th) as brand_cnt_click_m3th, SUM(is_buy_m3th) as brand_cnt_buy_m3th, "
+ "SUM(is_collect_m3th) as brand_cnt_collect_m3th, SUM(is_cart_m3th) as brand_cnt_cart_m3th, "
+ "SUM(is_click_3d) as brand_cnt_click_3d, SUM(is_buy_3d) as brand_cnt_buy_3d, "
+ "SUM(is_collect_3d) as brand_cnt_collect_3d, SUM(is_cart_3d) as brand_cnt_cart_3d, "
+ "SUM(is_click_3d2nd) as brand_cnt_click_3d2nd, SUM(is_buy_3d2nd) as brand_cnt_buy_3d2nd, "
+ "SUM(is_collect_3d2nd) as brand_cnt_collect_3d2nd, SUM(is_cart_3d2nd) as brand_cnt_cart_3d2nd, "
+ "SUM(is_click_3d3th) as brand_cnt_click_3d3th, SUM(is_buy_3d3th) as brand_cnt_buy_3d3th, "
+ "SUM(is_collect_3d3th) as brand_cnt_collect_3d3th, SUM(is_cart_3d3th) as brand_cnt_cart_3d3th, "
+ "SUM(is_click_1w) as brand_cnt_click_1w, SUM(is_buy_1w) as brand_cnt_buy_1w, "
+ "SUM(is_collect_1w) as brand_cnt_collect_1w, SUM(is_cart_1w) as brand_cnt_cart_1w, "
+ "SUM(is_click_w2nd) as brand_cnt_click_w2nd, SUM(is_buy_w2nd) as brand_cnt_buy_w2nd, "
+ "SUM(is_collect_w2nd) as brand_cnt_collect_w2nd, SUM(is_cart_w2nd) as brand_cnt_cart_w2nd, "
+ "SUM(is_click_w3th) as brand_cnt_click_w3th, SUM(is_buy_w3th) as brand_cnt_buy_w3th, "
+ "SUM(is_collect_w3th) as brand_cnt_collect_w3th, SUM(is_cart_w3th) as brand_cnt_cart_w3th";

```

```
BatchOperator t1_brand = t1_prepoc.groupBy("brand_id", clauseBrand);
```

有了计数信息后，再进一步计算其转化率。我们将刚才计算的计数数据表作为输入，对其中的每一条记录进行变换。由于最终的特征中仍包括这些计算信息，因此在变换结果列中包括了原始的计数列，并将其对应的计算表达式写成该列的列名。新产生的购买转化率列满足如下规则：

- 以 brand\_rt 开头，表明此列为品牌的比率信息。
- 中间为行为信息加上“2buy”，包括点击转化率（click2buy）、购买转化率（buy2buy）、收藏转化率（collect2buy）和加入购物车的转化率（cart2buy）等信息。
- 最后为时间段信息，比如，1m 为最近 1 个月，3m 为最近 3 个月，3d 为最近 3 天，1w 为最近 1 周。若为整个时间段，则没有时间段信息。
- 各部分之间用下画线“\_”连接。

详细的代码如下：

```

String clauseBrand_Rate = "brand_id AS brand_id4join, "
+ "brand_cnt_click,brand_cnt_buy,brand_cnt_collect,brand_cnt_cart, "
+ "brand_cnt_click_1m,brand_cnt_buy_1m,brand_cnt_collect_1m,brand_cnt_cart_1m, "
+ "brand_cnt_click_2m,brand_cnt_buy_2m,brand_cnt_collect_2m,brand_cnt_cart_2m, "
+ "brand_cnt_click_3m,brand_cnt_buy_3m,brand_cnt_collect_3m,brand_cnt_cart_3m, "
+ "brand_cnt_click_m2nd,brand_cnt_buy_m2nd,brand_cnt_collect_m2nd,brand_cnt_cart_m2nd, "
+ "brand_cnt_click_m3th,brand_cnt_buy_m3th,brand_cnt_collect_m3th,brand_cnt_cart_m3th, "
+ "brand_cnt_click_3d,brand_cnt_buy_3d,brand_cnt_collect_3d,brand_cnt_cart_3d, "
+ "brand_cnt_click_3d2nd,brand_cnt_buy_3d2nd,brand_cnt_collect_3d2nd,brand_cnt_cart_3d2nd, "

```

```

+ "brand_cnt_click_3d3th,brand_cnt_buy_3d3th,brand_cnt_collect_3d3th,brand_cnt_cart_3d3th,"
+ "brand_cnt_click_1w,brand_cnt_buy_1w,brand_cnt_collect_1w,brand_cnt_cart_1w,"
+ "brand_cnt_click_w2nd,brand_cnt_buy_w2nd,brand_cnt_collect_w2nd,brand_cnt_cart_w2nd,"
+ "brand_cnt_click_w3th,brand_cnt_buy_w3th,brand_cnt_collect_w3th,brand_cnt_cart_w3th,"
+ "case when brand_cnt_buy>brand_cnt_click then 1.0 when brand_cnt_buy=0 then 0.0 else "
+ "brand_cnt_buy*1.0/brand_cnt_click end AS brand_rt_click2buy,"
+ "case when brand_cnt_buy>brand_cnt_collect then 1.0 when brand_cnt_buy=0 then 0.0 else "
+ "brand_cnt_buy*1.0/brand_cnt_collect end AS brand_rt_collect2buy,"
+ "case when brand_cnt_buy>brand_cnt_cart then 1.0 when brand_cnt_buy=0 then 0.0 else "
+ "brand_cnt_buy*1.0/brand_cnt_cart end AS brand_rt_cart2buy,"
+ "case when brand_cnt_buy_3d>brand_cnt_click_3d then 1.0 when brand_cnt_buy_3d=0 then 0.0 else "
+ "brand_cnt_buy_3d*1.0/brand_cnt_click_3d end AS brand_rt_click2buy_3d,"
+ "case when brand_cnt_buy_3d>brand_cnt_collect_3d then 1.0 when brand_cnt_buy_3d=0 then 0.0 else "
+ "brand_cnt_buy_3d*1.0/brand_cnt_collect_3d end AS brand_rt_collect2buy_3d,"
+ "case when brand_cnt_buy_3d>brand_cnt_cart_3d then 1.0 when brand_cnt_buy_3d=0 then 0.0 else "
+ "brand_cnt_buy_3d*1.0/brand_cnt_cart_3d end AS brand_rt_cart2buy_3d,"
+ "case when brand_cnt_buy_1w>brand_cnt_click_1w then 1.0 when brand_cnt_buy_1w=0 then 0.0 else "
+ "brand_cnt_buy_1w*1.0/brand_cnt_click_1w end AS brand_rt_click2buy_1w,"
+ "case when brand_cnt_buy_1w>brand_cnt_collect_1w then 1.0 when brand_cnt_buy_1w=0 then 0.0 else "
+ "brand_cnt_buy_1w*1.0/brand_cnt_collect_1w end AS brand_rt_collect2buy_1w,"
+ "case when brand_cnt_buy_1w>brand_cnt_cart_1w then 1.0 when brand_cnt_buy_1w=0 then 0.0 else "
+ "brand_cnt_buy_1w*1.0/brand_cnt_cart_1w end AS brand_rt_cart2buy_1w,"
+ "case when brand_cnt_buy_1m>brand_cnt_click_1m then 1.0 when brand_cnt_buy_1m=0 then 0.0 else "
+ "brand_cnt_buy_1m*1.0/brand_cnt_click_1m end AS brand_rt_click2buy_1m,"
+ "case when brand_cnt_buy_1m>brand_cnt_collect_1m then 1.0 when brand_cnt_buy_1m=0 then 0.0 else "
+ "brand_cnt_buy_1m*1.0/brand_cnt_collect_1m end AS brand_rt_collect2buy_1m,"
+ "case when brand_cnt_buy_1m>brand_cnt_cart_1m then 1.0 when brand_cnt_buy_1m=0 then 0.0 else "
+ "brand_cnt_buy_1m*1.0/brand_cnt_cart_1m end AS brand_rt_cart2buy_1m";
t1_brand = t1_brand.select(clauseBrand_Rate);

```

## 5. 整合训练数据的特征

前面分别计算出了用户-品牌联合特征、用户特征和品牌特征。在判断一个用户是否会购买一个品牌的时候，用户对该品牌的关注度是重要的因素，它可以通过用户-品牌联合特征进行刻画。另外，该用户的购买特点和该品牌被购买的情况，也是重要的因素。所以，我们需要将所有的特征整合起来进行分析。

整合的方法如下：以 user\_id 和 brand\_id 作为联合主键，包括用户-品牌联合特征，再使用 SQL 的 JOIN（连接）操作，加入 user\_id 对应的用户特征，并加入 brand\_id 对应的品牌特征。

详细的代码如下：

```

BatchOperator t1_join = new JoinBatchOp()
.setSelectClause("*")
.setJoinPredicate("user_id=user_id4join")
.linkFrom(t1_userbrand, t1_user);

t1_join = new JoinBatchOp()
.setSelectClause("*")
.setJoinPredicate("brand_id=brand_id4join")
.linkFrom(t1_join, t1_brand);

```

### 11.3.3 计算标签

首先计算出最后一个月中发生了购买行为的用户-品牌对。这可以使用 SQL DISTINCT 语句轻松得到。

如果前面计算出来的特征数据表中某一条记录的 user\_id 和 brand\_id 在由原始数据划分出的 t2 中出现，就说明该 user\_id 用户在最后一个月中购买了该 brand\_id 品牌的商品，可以将 label 列赋值为 1；反之该用户就没有购买该品牌的商品。使用 LEFT OUTER JOIN（左连接）方法，将有购买行为的那些记录的 label 列赋值为 1，其他项为缺失值状态。相应的代码如下：

```
BatchOperator t2_label = t2
    .filter("type=1")
    .select("user_id AS user_id4label, brand_id AS brand_id4label, 1 as label")
    .distinct();

BatchOperator feature_label = new LeftOuterJoinBatchOp()
    .setSelectClause("*")
    .setJoinPredicate("user_id = user_id4label AND brand_id = brand_id4label")
    .linkFrom(t1_join, t2_label);
```

然后，我们再使用缺失值填充组件，选择 label 字段，缺失值用 0 填充，具体代码如下：

```
Imputer imputer = new Imputer()
    .setStrategy("value")
    .setFillValue("0")
    .setSelectedCols("label");

feature_label = imputer.fit(feature_label).transform(feature_label);
```

我们再看一下当前数据集的 Schema，代码如下：

```
System.out.println(feature_label.getSchema());
```

结果如下。由于列数较多，因此这里选择了一些有代表性的列名称和类型。

```
root
|   user_id: BIGINT
|   brand_id: BIGINT
|   cnt_click: INT
|   cnt_buy: INT
|   cnt_collect: INT
|   cnt_cart: INT
|   cnt_click_lm: INT
|   cnt_buy_lm: INT
```

---

```

    ...
    |— brand_id4join: BIGINT
    |— brand_cnt_click: INT
    |— brand_cnt_buy: INT
    ...
    |— brand_rt_click2buy_1m: LEGACY(BigDecimal)
    |— brand_rt_collect2buy_1m: LEGACY(BigDecimal)
    |— brand_ft_cart2buy_1m: LEGACY(BigDecimal)
    |— user_id4label: BIGINT
    |— brand_id4label: BIGINT
    |— label: INT

```

---

在此，我们看到两个问题：

- 有些数据列对于后面的分类问题是没用的，比如 brand\_id4join、user\_id4label 等。
- 这里有 LEGACY(BigDecimal)类型，该类型需要转化为基本的 double 类型，方便后面分类器使用。

为解决这两个问题，可以构造 SQL 语句，选取所有需要的特征列，将其类型转化为 double 类型，再加上标签列。具体代码如下。最后将生成的数据集保存起来，供后面的模型训练使用。将整个数据集导出到文件 feature\_label.ak 中。

---

```

String[] featureColNames =
    ArrayUtils.removeElements(
        feature_label.getColNames(),
        new String[] {
            "user_id", "brand_id",
            "user_id4join", "brand_id4join",
            "user_id4label", "brand_id4label",
            LABEL_COL_NAME
        }
    );
}

StringBuilder sbd = new StringBuilder();
for (String name : featureColNames) {
    sbd.append("CAST(").append(name).append(" AS DOUBLE) AS ").append(name).append(", ");
}
sbd.append(LABEL_COL_NAME);

feature_label
    .select(sbd.toString())
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + FEATURE_LABEL_FILE)
            .setOverwriteSink(true)
    );

```

---

## 11.4 正负样本配比

前面已经构造出很多特征，本节将尝试多种二分类模型，并选出最适合的模型。这里需要提醒用户注意的一点是，训练样本中正负样本的比例如果相差太悬殊，会对训练出来的模型产生影响。我们会在模型训练前先关注一下，训练数据的正负样本比例，并进行调整。

前面构造了特征和标签，现在我们先深入了解一下数据：

---

```
AkSourceBatchOp all_data = new AkSourceBatchOp().setFilePath(DATA_DIR + FEATURE_LABEL_FILE);

all_data
    .lazyPrintStatistics()
    .groupBy("label", "label", COUNT(*) AS cnt")
    .print();
```

---

打印统计信息如下，样本总条数为 42 531，没有缺失值；计数类特征的值域变化较大，比率类特征取值为[0, 1]。

colName	count	missing	sum	mean	variance	min	max
cnt_click 42531	0	125865	2.9594	76.9641	0	542	
cnt_buy 42531	0	4971	0.1169	0.3815	0	40	
cnt_collect 42531	0	804	0.0189	0.0324	0	13	
cnt_cart 42531	0	80	0.0019	0.0025	0	4	
cnt_click_1m 42531	0	48025	1.1292	22.1792	0	302	
cnt_buy_1m 42531	0	1880	0.0442	0.1202	0	17	
cnt_collect_1m 42531	0	307	0.0072	0.0115	0	6	
cnt_cart_1m 42531	0	43	0.001	0.0013	0	4	
brand_rt_click2buy_lw 42531	0	600.978	0.0141	0.0036	0	1	
brand_rt_collect2buy_lw 42531	0	6259	0.1472	0.1252	0	1	
brand_cart2buy_lw 42531	0	6288	0.1478	0.126	0	1	
brand_rt_click2buy_1m 42531	0	1243.6492	0.0292	0.0052	0	1	
brand_rt_collect2buy_1m 42531	0	15682.119	0.3687	0.2307	0	1	
brand_rt_cart2buy_1m 42531	0	15860	0.3729	0.2339	0	1	
label 42531	0	259	0.0061	0.0061	0	1	

对标签列进行分组聚合计数，得到如下的结果，正负样本的数量有 2 个数量级的差距。

label cnt
0 42272
1 259