

接下来，我们对数据进行分类建模。首先对数据集进行拆分，得到训练集和测试集，代码如下：

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

随后，使用基本的二分类算法（逻辑回归），试验分类效果：

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

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

new LogisticRegression()
.setFeatureCols(featureColNames)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.setPredictionDetailCol(PRED_DETAIL_COL_NAME)
.fit(train_data)
.transform(test_data)
.link(
    new EvalBinaryClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics("LogisticRegression")
);
BatchOperator.execute();
```

得到二分类评估指标如下。其中，精确度(Accuracy)指标很高，为 0.9945；但是召回率(Recall)指标很低，只有 0.0435，46 个正样本只有 2 个被正确分类。

Metrics:		
Auc: 0.7629	Accuracy: 0.9945	Precision: 0.4
Recall: 0.0435	F1: 0.0784	LogLoss: 0.0342
Pred\Real	1	0
-----	-----	-----
1	2	3
0	44	8457

这里召回率(Recall)指标很低的主要原因在于，训练数据中正负样本的比例相差悬殊。一个解决方法是保持正样本的同时，通过采样来降低负样本的数量。可以使用分层采样组件StratifiedSampleBatchOp，对正负样本分别设置采样率，正样本全部保留，负样本采样 5%，参数setStrataRatios设置为"0:0.05,1:1.0"。将该结果保存到文件DATA_DIR + TRAIN_SAMPLE_FILE 中，方便后续多次试验时调用。相关代码如下：

```
train_data
.link(
    new StratifiedSampleBatchOp()
        .setStrataRatios("0:0.05,1:1.0")
        .setStrataCol(LABEL_COL_NAME)
```

```

)
.link(
    new AkSinkBatchOp()
    .setFilePath(DATA_DIR + TRAIN_SAMPLE_FILE)
);
BatchOperator.execute();

```

我们再对调整了正负样本比例的数据进行逻辑回归训练，并在测试集上进行二分类评估。具体代码如下：

```

AkSourceBatchOp train_sample =
    new AkSourceBatchOp().setFilePath(DATA_DIR + TRAIN_SAMPLE_FILE);

new LogisticRegression()
    .setFeatureCols(featureColNames)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_sample)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
        .lazyPrintMetrics("LogisticRegression with Stratified Sample")
    );
BatchOperator.execute();

```

运行结果如下。其中，召回率（Recall）提升为 0.2391，46 个正样本中有 11 个预测对了。

Metrics:		
Auc: 0.6724	Accuracy: 0.9638	Precision: 0.0387
Recall: 0.2391	F1: 0.0667	LogLoss: 0.19
Pred\Real	1	0
-----	-----	-----
1	11	273
0	35	8187

11.5 决策树

本节将继续对分层采样后的训练数据进行建模。下面我们看看 3 种决策树算法的效果。

使用决策树组件 DecisionTreeClassifier，遍历 3 种决策树类型 TreeType.GINI、TreeType.INFOGAIN、TreeType.INFOGAINRATIO，具体代码如下：

```

for (TreeType treeType : new TreeType[] {TreeType.GINI,
    TreeType.INFOGAIN, TreeType.INFOGAINRATIO}) {

```

```

new DecisionTreeClassifier()
    .setTreeType(treeType)
    .setFeatureCols(featureColNames)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_sample)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics(treeType.toString())
    );
}

```

将运行结果放在一起对比，如表 11-2 所示。

表 11-2 3 种决策树算法的运行结果对比

ID3	C4.5	CART
TreeType.INFOGAIN	TreeType.INFOGAINRATIO	TreeType.GINI
Auc:0.6532	Auc:0.6465	Auc:0.6632
Accuracy:0.8895	Accuracy:0.8655	Accuracy:0.8924
Precision:0.0194	Precision:0.0167	Precision:0.0199
Recall:0.3913	Recall:0.413	Recall:0.3913
F1:0.0369	F1:0.0321	F1:0.0379
LogLoss:2.9092	LogLoss:2.0405	LogLoss:2.3753
混淆矩阵:	混淆矩阵:	混淆矩阵:
Pred\Real 1 0	Pred\Real 1 0	Pred\Real 1 0
----- --- ---	----- --- ---	----- --- ---
1 18 912	1 19 1117	1 18 887
0 28 7548	0 27 7343	0 28 7573

这 3 种决策树算法在各项指标上的差异不大。和前面的逻辑回归算法相比，这 3 种决策树算法在召回率上有一些提升，但由于其 Precision 指标的下降，致使 F1 指标偏低。

11.6 集成学习

本节介绍一个新的思路：集成学习（Ensemble Learning），以便在决策树模型的基础上，

构建更为强大的分类器：随机森林模型和 GBDT 模型。

人们常说“三个臭皮匠赛过诸葛亮”，这种思想在机器学习领域的体现，就是集成学习。

集成学习（Ensemble Learning），用多个弱分类器构成一个强分类器，目的是增强集成模型的预测性能。弱分类器可以由决策树、神经网络、贝叶斯、K 最近邻等构成。

下面着重讨论两种广泛使用的集成技术：Bagging（装袋法）和 Boosting（提升法）。

11.6.1 Bootstrap aggregating

Bootstrap aggregating，经常使用缩写形式“Bagging”（由 Bootstrap AGGREGATING 得来）。由于这种缩写的关系，其也被称为“装袋法”。

“Bootstrap”这个名字来自谚语“pull up by your own bootstraps”，意思是“improve your situation by your own efforts”，即依靠自身努力得到提高。因此，Bootstrap 被称为自助法，或自举法。Bootstrap 是非参数统计中一种重要的估计统计量方差，进而进行区间估计的统计方法，基本步骤如下：

- (1) 采用重复抽样技术，从原始样本中抽取出 N 个样本。
- (2) 根据抽取出的样本计算给定的统计量 T 。
- (3) 重复上述步骤 M 次（一般大于 1000 次），得到 M 个统计量 T 。
- (4) 计算上述 M 个统计量 T 的样本方差，得到统计量的方差。

Bootstrap aggregating（Bagging）将统计学中常用的 Bootstrap 方法应用到分类模型上，成为 Bagging 分类器（Bagging Classifier）。Bagging 分类器的基本思想是对训练集进行有放回的抽样，得到 M 个子训练集，分别对每个子训练集使用若干分类器进行训练，得到 M 个模型；预测时，对 M 个模型预测的结果进行投票，获得最终的分类结果。具体步骤如下：

- (1) 采用重复抽样技术，从训练样本中抽取 N 个样本。
- (2) 对这 N 个样本建立 M 个弱分类器（CART、SVM 等）。
- (3) 重复以上两步 M 次，训练这 M 个弱分类器（CART、SVM 等）。
- (4) 预测时， M 个分类器各自得到分类结果。通过投票方式，确定最终的分类结果。

Bagging 分类器的训练如图 11-2 所示，各个子分类器的训练是可以并发进行的。

在预测阶段，首先是各子分类器分别进行预测，然后对

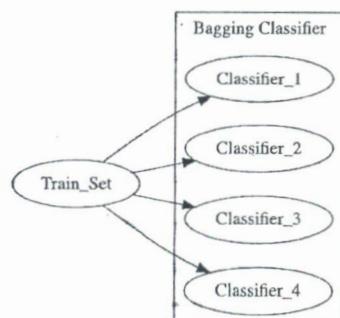


图 11-2 Bagging Classifier 训练示意图

各预测结果通过投票 (Vote) 的方式进行汇总，作为 Bagging 分类器 (Bagging Classifier) 的预测结果，如图 11-3 所示。

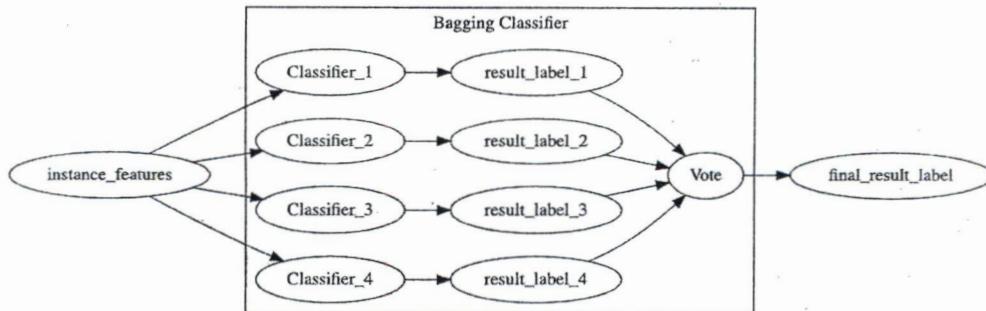


图 11-3 Bagging Classifier 预测示意图

11.6.2 Boosting

前面介绍的 Bagging (装袋法) 采用并行的方法组合多个弱分类器，而 Boosting (提升法) 则采样串行的方式组合多个弱分类器。

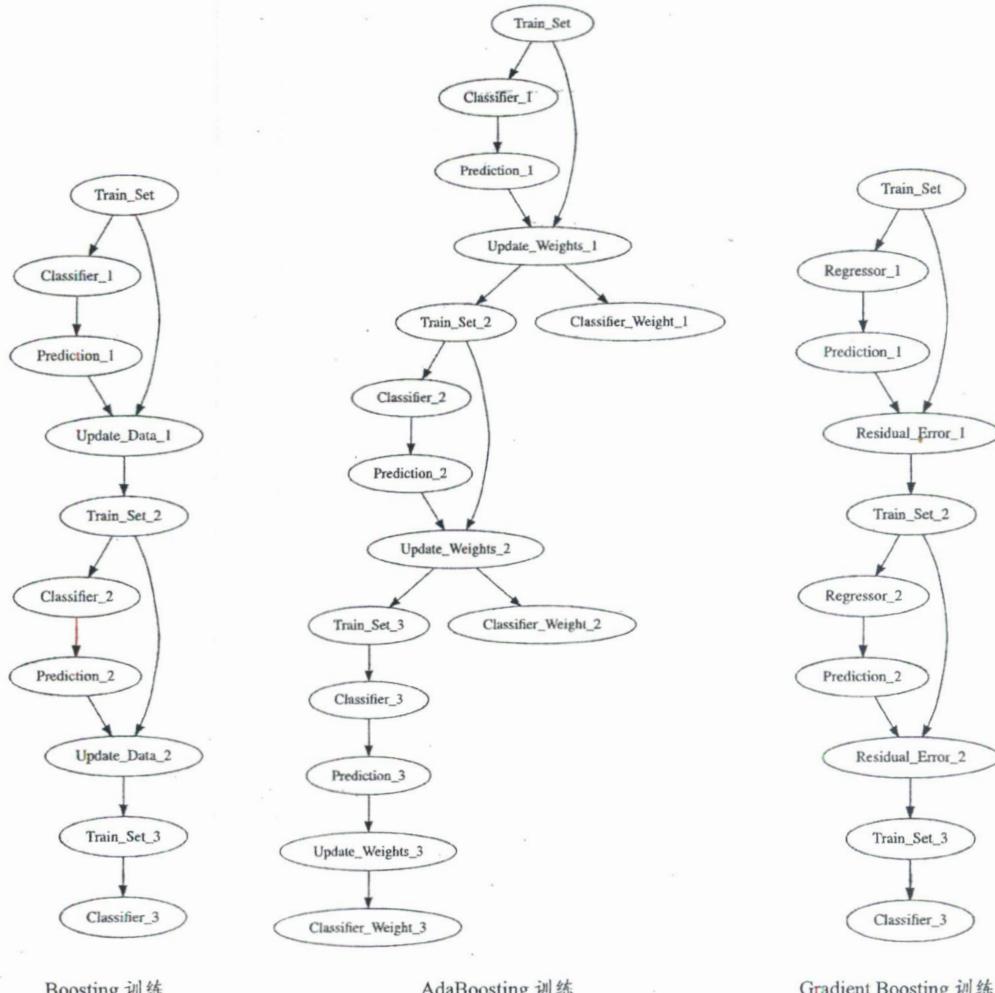
采用 Boosting 方法时，先使用一个弱分类器建立一个模型，以便对训练数据有一个初步的预测；然后针对预测结果，侧重那些被错误预测的样本，通过调整其权重或者关注其残差等方式，重新生成新的训练集；之后再使用一个弱分类器对新生成的训练集进行训练，得到新的模型及新的预测结果；随后重新生成新的训练集……一直重复下去，直到得到指定数量的弱分类器模型。我们可以想象成，Boosting 方法得到了一个弱分类器序列；第一个弱分类器在训练原始数据；其后的每个弱分类器都是针对之前弱分类器没有训练好的情形，进行加强训练。

如图 11-4 的左子图所示，Train_Set 为原始训练数据，训练第一个弱分类器 Classifier_1 并得到一个弱分类器模型；然后对 Train_Set 进行预测得到 Prediction_1，结合训练数据 Train_Set 与预测结果 Prediction_1，通过某种关注训练错误的方式 Update_Data_1，得到新的训练集 Train_Set_2；随后，再由 Train_Set_2 得到 Classifier_2；之后预测出 Prediction_2，使用方式 Update_Data_2 得到新的数据集 Train_Set_3；直到得到最后一个弱分类器 Classifier_3。弱分类器序列 {Classifier_1, Classifier_2, Classifier_3} 就是最终集成的分类器。

使用 Boosting 思想的具体算法主要有 AdaBoost 算法和 Gradient Boosting 算法。

采用 AdaBoost 算法，可通过对训练失败的样本赋以较大权重的方式，得到新的训练集。如图 11-4 的中间子图所示，对原始数据进行训练并得到 Classifier_1 后，结合训练数据 Train_Set 与预测结果 Prediction_1，通过调整训练样本权重 (Update_Weight_1) 的方式，得到新的训练

集 Train_Set_2。此外，还计算出了弱分类器 Classifier_1 的权重参数 Classifier_Weight_1（权重参数会在介绍 AdaBoost 模型如何执行预测操作时说明）；然后基于 Train_Set_2，得到弱分类器 Classifier_2，再结合训练数据 Train_Set_2 与预测结果 Prediction_2，通过调整训练样本权重（Update_Weight_2）的方式，得到新的训练集 Train_Set_3 及模型权重参数 Classifier_Weight_3……直到得到最后一个弱分类器 Classifier_3 的权重参数 Classifier_Weight_3。



训练中得到的各弱分类器及其权重构成了 AdaBoost 模型: { Classifier_1, Classifier_Weight_1; Classifier_2, Classifier_Weight_2; Classifier_3, Classifier_Weight_3}。

下面来看看使用 AdaBoost 模型的预测过程。如图 11-5 所示, 待预测样本的特征值 instance_features 分别被各弱分类器 { Classifier_1, Classifier_2, Classifier_3} 预测, 得到各自的预测结果 { result_label_1, result_label_2, result_label_3}, 然后考虑各模型的权重系数 {Classifier_Weight_1, Classifier_Weight_2, Classifier_Weight_3}, 通过投票的方式得到 AdaBoost 模型的预测结果 final_result_label。

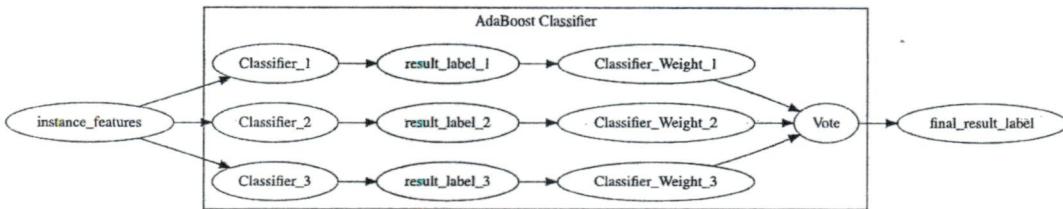


图 11-5 AdaBoost 预测示意图

注意: 想了解 AdaBoost 算法更多细节的读者, 可以参考以下内容:

Freund, Y. & R.E. Schapire (1996), *Experiments with a New Boosting Algorithm, in Proceedings of the Thirteenth International Conference on Machine Learning*, Morgan Kaufmann (参见链接 11-2)。

Gradient Boosting 算法通过损失函数 (loss function) 来度量预测值与真实值的不一致程度。损失函数越小, 则模型预测得越准确。如果随着更多弱分类器的加入, 损失函数能持续下降, 就说明模型在持续改进。最好的方式是, 使损失函数在其梯度 (Gradient) 方向下降。Gradient Boosting 算法将二分类问题看作值为 +1 和 -1 的回归问题, 并最终根据回归值是否大于 0 来将回归值转换输出成分类标签值。

Gradient Boosting 使用的是弱回归器。如图 11-4 的右子图所示, Train_Set 为原始训练数据, 训练第一个弱回归器 Regressor_1 并得到模型; 然后对 Train_Set 进行回归预测并得到 Prediction_1, 结合训练数据 Train_Set 与回归预测结果 Prediction_1, 通过计算残差 Residual_Error_1, 得到新的训练集 Train_Set_2; 随后, 由 Train_Set_2 得到 Regressor_2; 之后预测出 Prediction_2, 计算残差 Residual_Error_2, 得到新的数据集 Train_Set_3……直到得到最后一个弱回归器 Regressor_3。弱回归器模型序列 {Regressor_1, Regressor_2, Regressor_3} 就是 Gradient Boosting 模型。

使用 Gradient Boosting 模型进行预测的流程如图 11-6 所示。待预测样本的特征值

`instance_features` 分别被各弱回归器 { `Regressor_1`, `Regressor_2`, `Regressor_3` } 预测，得到各自的预测结果 { `result_value_1`, `result_value_2`, `result_value_3` }, 求和汇总出 Gradient Boosting 模型的预测结果 `final_result`，根据结果值是否大于 0，便可将结果值转换为二分类标签值。

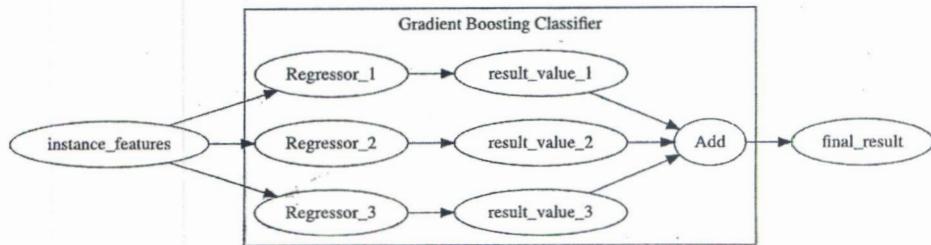


图 11-6 Gradient Boosting 预测示意图

注意：想了解 Gradient Boosting 算法更多细节的读者，可以参考以下内容：

Friedman, J. H., *Greedy Function Approximation: A Gradient Boosting Machine* (February 1999)，参见链接 11-3。

11.6.3 随机森林与GBDT

知道了集成方法，就很容易理解随机森林算法和 GBDT 算法了。

随机森林 (random forests) 算法是由 Leo Breiman 和 Adele Cutler 提出的，并被注册成了商标。随机森林使用了 Bagging 方法，由多个分类决策树 (弱分类器) 构成，每个分类决策树的地位是平等的，并通过投票的方式确定最终的预测结果。关于该算法的详细内容可以参考以下内容：

- Ho, Tin Kam (1995). *Random Decision Forests*. Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14–16 August 1995. pp. 278–282
- Breiman, Leo (2001). "Random Forests". *Machine Learning*, 45 (1): 5–32

GBDT 为 Gradient Boosting Decision Tree 的缩写，该算法又被称为 MART (Multiple Additive Regression Tree) 算法，在其名称中就已经包含了所使用的集成方法 (Gradient Boosting)。GBDT 是由多棵回归树 (Regression Tree) 组成的，所有树的结论累加起来，通过是否大于 0，将回归值转换为二分类标签值。

随机森林与 GBDT 的简单对比如表 11-3 所示。

表 11-3 随机森林算法和 GBDT 算法的对比

算法	集成方法	弱分类/回归器
随机森林	Bagging	分类树
GBDT	Gradient Boosting	回归树

11.7 使用随机森林算法

随机森林是由多棵分类决策树构成的，可以通过调节决策树的棵数、每棵决策树的深度等参数来取得较好的分类效果。

如下代码所示，使用随机森林组件 RandomForestClassifier，对分层采样的训练数据进行训练，并使用测试集进行评估。这里设置树的棵数为 20，每棵树的深度为 4，连续特征会被分成 512 个 bin。

```
new RandomForestClassifier()
    .setNumTrees(20)
    .setMaxDepth(4)
    .setMaxBins(512)
    .setFeatureCols(featureColNames)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_sample)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("RandomForest with Stratified Sample")
    );
};
```

使用随机森林算法的评估结果如下。与逻辑回归算法相比，召回率指标下降，但其他指标都有所提升，F1 指标超过逻辑回归算法中的 F1 数值。

```
----- Metrics: -----
Auc: 0.8428 Accuracy: 0.9763 Precision: 0.0465      Recall: 0.1739   F1: 0.0734 LogLoss: 0.1261
|Pred\Real| 1 | 0 |
|-----|---|---|
| 1 | 8 | 164 |
| 0 | 38 | 8296 |
```

11.8 使用GBDT算法

前面使用随机森林算法获得了较好的效果，这次尝试使用 GBDT 算法。代码如下，使用组件 GbdtClassifier，设置树的棵数为 100，每棵树的深度为 5，连续特征会被分成 256 个 bin。

```
new GbdtClassifier()
    .setNumTrees(100)
    .setMaxDepth(5)
    .setMaxBins(256)
    .setFeatureCols(featureColNames)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .fit(train_sample)
    .transform(test_data)
    .link(
        new EvalBinaryClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .lazyPrintMetrics("GBDT with Stratified Sample")
    );
};
```

使用 GBDT 算法的评估结果如下。与使用随机森林算法相比，使用 GBDT 算法的 F1 指标与其接近，但召回率有较大提升。

Metrics:			
Auc:0.8052	Accuracy:0.9559	Precision:0.0418	
Recall:0.3261	F1:0.0741	LogLoss:0.1386	
Pred\Real	1	0	
-----	-----	-----	
	1	15	344
	0	31	8116

随机森林算法和 GBDT 算法的分类效果较好，但在使用中二者也有不方便的地方，这两种算法在调参方面花费的时间较多。随机森林算法和 GBDT 算法都有多个参数，这些参数对模型效果的影响较大，需要根据具体的问题选择适合的参数值。在第 20 章中会介绍参数搜索方面的内容，以帮助读者掌握这类复杂模型参数的选择方法。

12

从二分类到多分类

本章仍然演示分类的例子，但类别的个数要多于两个，即我们常说的多分类问题。前面介绍的朴素贝叶斯、决策树、随机森林模型都能用在多分类的场景中，本章及第 13 章会在实例分析的过程中陆续介绍一些新的多分类模型。

多分类模型的评估指标包含一些二分类评估指标（例如精确度、Kappa、LogLoss），还有一些基于多个二分类指标的整合指标。评估指标涉及的内容比较通用，与具体分类算法无关，将在 12.1 节中进行介绍。

12.1 多分类模型评估方法

在我们使用测试数据样本集评判模型的分类能力时，每个测试样本都有一个已知的实际分类值，而模型通过对每个样本的特征进行预测，每个测试样本又可以得到其预测分类值。混淆矩阵（Confusion Matrix）就是采用统计上交叉表的做法，将实际分类值与预测分类值分别作为矩阵的 2 个维度，统计各种组合情况下样本的个数作为矩阵值。

假设有 K 个分类，则分别统计测试数据样本中被预测为第 i 个分类，而实际上属于第 j 个分类的样本数 $n_{i,j}$ 。这样我们就得到了如表 12-1 所示的混淆矩阵。

表 12-1 K 分类的混淆矩阵

预测结果	实际值			
	分类 1	分类 2	...	分类 K
分类 1	$n_{1,1}$	$n_{1,2}$...	$n_{1,K}$

续表

预测结果	实际值			
	分类 1	分类 2	...	分类 K
分类 2	$n_{2,1}$	$n_{2,2}$...	$n_{2,K}$
:	:	:	:	:
分类 K	$n_{K,1}$	$n_{K,2}$...	$n_{K,K}$

显然，上述混淆矩阵有如下的性质。

- 第*i*行代表预测结果是分类*i*，该行的总数，即 $\sum_{j=1}^K n_{i,j}$ ，为预测为分类*i*的样本数目。
- 第*j*列代表实际属于分类*j*，该列的总数，即 $\sum_{i=1}^K n_{i,j}$ ，为实际属于分类*j*的样本数目。
- 测试样本的总数为

$$N = \sum_{i=1}^K \sum_{j=1}^K n_{i,j}$$

下面介绍另一种常用的混淆矩阵的展现方式——使用比例来描述实际属于分类*j*的样本，被预测为分类*i*的比例，即矩阵中第*i*行第*j*列的值为

$$\frac{n_{i,j}}{\sum_{i=1}^K n_{i,j}}$$

由比值构成的矩阵如表 12-2 所示。

表 12-2 由比值构成的混淆矩阵

预测结果	实际值			
	分类 1	分类 2	...	分类 K
分类 1	$\frac{n_{1,1}}{\sum_{i=1}^K n_{i,1}}$	$\frac{n_{1,2}}{\sum_{i=1}^K n_{i,2}}$...	$\frac{n_{1,K}}{\sum_{i=1}^K n_{i,K}}$
分类 2	$\frac{n_{2,1}}{\sum_{i=1}^K n_{i,1}}$	$\frac{n_{2,2}}{\sum_{i=1}^K n_{i,2}}$...	$\frac{n_{2,K}}{\sum_{i=1}^K n_{i,K}}$
:	:	:	:	:
分类 K	$\frac{n_{K,1}}{\sum_{i=1}^K n_{i,1}}$	$\frac{n_{K,2}}{\sum_{i=1}^K n_{i,2}}$...	$\frac{n_{K,K}}{\sum_{i=1}^K n_{i,K}}$

由于矩阵对角线上的各项实际上表示的是属于分类*k*，预测结果也为分类*k*的情形，因此这些值越大，模型的精确度越高。

12.1.1 综合指标

本节将介绍三个多分类指标：精确度、Kappa 系数、LogLoss。

后面会陆续介绍使用 Micro、Macro、Weighted 方法，以及基于各标签值的二分类指标给出的整体多分类指标。

1. 精确度

精确度（Accuracy, ACC）的定义为，预测正确的样本在整个样本中的比例。结合混淆矩阵的定义，其公式为

$$ACC = \frac{\sum_{k=1}^K n_{k,k}}{\sum_{i=1}^K \sum_{j=1}^K n_{i,j}}$$

2. Kappa 系数

1960 年，Cohen 等人提出用 Kappa 值（用希腊字母 κ 表示）作为评判诊断试验一致性程度的指标。建议的参考标准如下。

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

多分类情况下，Kappa 系数的定义可以由混淆矩阵来描述：把样本总数 (N) 乘以混淆矩阵对角线 ($n_{k,k}$) 的和，再减去各分类中属于该分类的样本数 ($\sum_{i=1}^K n_{i,k}$) 与被预测为该分类的样本数 ($\sum_{j=1}^K n_{k,j}$) 之积之后，再除以样本总数 (N) 的平方，减去各分类中属于该分类的样本数 ($\sum_{i=1}^K n_{i,k}$) 与被预测为该分类的样本数 ($\sum_{j=1}^K n_{k,j}$) 之积。对所有类别求和的结果为

$$\kappa = \frac{N \sum_{k=1}^K n_{k,k} - \sum_{k=1}^K \{(\sum_{i=1}^K n_{i,k}) \cdot (\sum_{j=1}^K n_{k,j})\}}{N^2 - \sum_{k=1}^K \{(\sum_{i=1}^K n_{i,k}) \cdot (\sum_{j=1}^K n_{k,j})\}}$$

在此定义表达式上进行恒等变换，我们可以得到新的描述方式。将分子、分母同时除以 N^2 ，可以得到

$$\kappa = \frac{\frac{\sum_{k=1}^K n_{k,k}}{N} - \sum_{k=1}^K \left\{ \frac{(\sum_{i=1}^K n_{i,k})}{N} \cdot \frac{(\sum_{j=1}^K n_{k,j})}{N} \right\}}{1 - \sum_{k=1}^K \left\{ \frac{(\sum_{i=1}^K n_{i,k})}{N} \cdot \frac{(\sum_{j=1}^K n_{k,j})}{N} \right\}}$$

变换后，我们可以看到分子、分母中有相同的部分，设

$$P_a = \frac{\sum_{k=1}^K n_{k,k}}{N} \quad \text{和} \quad P_e = \sum_{k=1}^K \left\{ \frac{(\sum_{i=1}^K n_{i,k})}{N} \cdot \frac{(\sum_{j=1}^K n_{k,j})}{N} \right\}$$

则

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

其中， P_a 为实际分类情况与预测分类结果一致的样本数同测试数据样本总数的比值，即分类的精确度，被称为实际一致率； P_e 被称为期望一致率。

3. LogLoss

多分类线性模型 Softmax 的损失函数可以看作二分类逻辑回归模型的对数损失函数的扩展，具体的比较可以参见 12.5 节关于 Softmax 算法的介绍。二分类的 LogLoss 评估指标可以看作来源于逻辑回归，多分类的 LogLoss 评估指标可以看作来源于 Softmax。

Softmax 的经验损失函数为

$$L(W) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=0}^{K-1} [I(y^{(i)} = k) \cdot \log(\phi_k(x^{(i)}))]$$

其中， n 为样本总数， K 为分类数， $x^{(i)}$ 为第 i 个样本的特征向量， $y^{(i)}$ 为第 i 个样本的标签值，标签值的取值范围为 $\{0, 1, 2, \dots, K-1\}$ ，各标签值对应的预测概率为 $\{\phi_0(x), \phi_1(x), \dots, \phi_{K-1}(x)\}$ 。

我们引入如下定义来简化表达式。

$y_{i,k}$ 表示第 i 个样本的真实分类（即标签值）是否为第 k 个类，“是”对应值为 1，“否”对应值为 0。 $P_{i,k}$ 是模型预测出来的第 i 个样本为第 k 个类的概率。

则 LogLoss 指标为

$$\text{LogLoss} = -\frac{1}{n} \sum_{i=1}^n \sum_{k=0}^{K-1} [y_{i,k} \cdot \log(P_{i,k})]$$

12.1.2 关于每个标签值的二分类指标

对于一个多分类问题（分类个数为 K ），可以使用 One-vs-Rest 方法——将一个标签（Label）看作 1，其余标签都看作 0。这样得到 K 个二分类问题，每个二分类问题都能得到基础计数。

- TruePositive(TP): Predicted=True, Actual=True
- TrueNegative(TN): Predicted=False, Actual=False
- FalsePositive(FP): Predicted=True, Actual=False
- FalseNegative(FN): Predicted=False, Actual=True

通过基础计数 TP、FP、TN、FN，就可以计算得到后续指标。

- (1) Sensitivity or TruePositiveRate(TPR): $TPR = \frac{TP}{TP+FN}$
- (2) Specificity(SPC) or TrueNegativeRate(TNR): $SPC = \frac{TN}{FP+TN}$
- (3) Precision or PositivePredictiveValue(PPV): $PPV = \frac{TP}{TP+FP}$
- (4) NegativePredictiveValue(NPV): $NPV = \frac{TN}{TN+FN}$
- (5) Fall-out or FalsePositiveRate(FPR): $FPR = \frac{FP}{FP+TN} = 1 - SPC$
- (6) FalseDiscoveryRate(FDR): $FDR = \frac{FP}{FP+TP} = 1 - PPV$
- (7) MissRate or FalseNegativeRate(FNR): $FNR = \frac{FN}{FN+TN}$
- (8) Accuracy(ACC): $ACC = \frac{TP+TN}{TP+FN+TN+FP}$
- (9) F1-Score: $F1 = \frac{2TP}{2TP+FP+FN}$
- (10) Cohen's Kappa: $\kappa = \frac{P_a - P_e}{1 - P_e}$

其中， $P_a = \frac{TP+TN}{TP+FN+FP+TN}$ ， $P_e = \frac{(TP+FN) \times (TP+FP) + (FP+TN) \times (FN+TN)}{(TP+FN+FP+TN) \times (TP+FN+FP+TN)}$ 。

12.1.3 Micro、Macro、Weighted 计算的指标

12.1.2 节提到，一个多分类（分类个数为 K ）问题使用 One-vs-Rest 方法能得到 K 个二分类问题，每个二分类问题都可以计算出相应的指标。那么如何将 K 个二分类指标整合为一个反映原始多分类的指标呢？

常用的有 Micro、Macro、Weighted 三种算法，下面将分别介绍。为了表示方便，我们引入 B ，代表由二分类的基础计数 TP、FP、TN、FN 可以计算的指标，譬如召回率、F1 等。

(1) Micro 算法：核心想法是将各个二分类的基础计数 TP_i 、 FP_i 、 TN_i 、 FN_i 进行求和，作为整个多分类的 TP、FP、TN、FN，然后就可以利用二分类中的公式计算召回率等各种指标。

$$B_{\text{Micro}} = B \left(\sum_{i=1}^K TP_i, \sum_{i=1}^K FP_i, \sum_{i=1}^K TN_i, \sum_{i=1}^K FN_i \right)$$

(2) Macro 算法：先计算各个二分类的指标，然后将其算术平均值作为多分类的指标。

$$B_{\text{Macro}} = \frac{1}{K} \sum_{i=1}^K B(TP_i, FP_i, TN_i, FN_i)$$

(3) Weighted 算法：可以看作 Macro 算法的扩展，考虑到各类在总体的占比，将加权平均值作为多分类的指标。

$$B_{\text{weighted}} = \sum_{i=1}^K \frac{N_i}{N} B(TP_i, FP_i, TN_i, FN_i)$$

其中， N_i 表示第 i 个标签值的样本条数， N 表示全体评估数据的条数。

前面从不同角度讲了两个指标——Accuracy 和 Micro F1，其实这两个指标的值是相等的。下面是关于此结论的证明。

将混淆矩阵与二分类的基础计数 TP_i, FP_i, TN_i, FN_i 联系起来。混淆矩阵元素 $n_{i,j}$ 被预测为第 i 个分类，而实际上属于第 j 个分类的样本数。基础计数 TP_i, FP_i, TN_i, FN_i 可以用 $n_{i,j}$ 表示。

- TP_k : 将第 k 个分类值看作 1 (正例)，其余为 0 (负例) 时，预测为第 k 个分类，而实际上属于第 k 个分类的样本数，即

$$TP_k = n_{k,k}$$

- FP_k : 将第 k 个分类值看作 1 (正例)，其余为 0 (负例) 时，预测为第 k 个分类，而实际上不属于第 k 个分类的样本数，即

$$FP_k = \sum_{i \neq k} n_{k,i}$$

- FN_k : 为将第 k 个分类值看作 1 (正例)，其余为 0 (负例) 时，没有被预测为第 k 个分类，而实际上是第 k 个分类的样本数，即

$$FN_k = \sum_{i \neq k} n_{i,k}$$

由 Micro 的定义，有

$$\text{Micro F1} = \frac{2 \sum_k TP_k}{2 \sum_k TP_k + \sum_k FP_k + \sum_k FN_k}$$

结合基础计数 TP_i, FP_i, FN_i 与 $n_{i,j}$ 的关系，有

$$\sum_k \text{TP}_k = \sum_k n_{k,k}$$

$$\sum_k \text{FP}_k = \sum_k \sum_{i \neq k} n_{k,i}$$

$$\sum_k \text{FN}_k = \sum_k \sum_{i \neq k} n_{i,k}$$

显然, $\sum_k \text{TP}_k$ 为混淆矩阵中对角线 $\{n_{1,1}, \dots, n_{K,K}\}$ 元素的和, $\sum_k \text{FP}_k$ 和 $\sum_k \text{FN}_k$ 都是混淆矩阵中非对角线 $\{n_{1,1}, \dots, n_{K,K}\}$ 元素的和。于是

$$\begin{aligned} & 2 \sum_k \text{TP}_k + \sum_k \text{FP}_k + \sum_k \text{FN}_k \\ &= 2 \sum_k n_{k,k} + \sum_k \sum_{i \neq k} n_{k,i} + \sum_k \sum_{i \neq k} n_{i,k} \\ &= \left(\sum_k n_{k,k} + \sum_k \sum_{i \neq k} n_{k,i} \right) + \left(\sum_k n_{k,k} + \sum_k \sum_{i \neq k} n_{i,k} \right) \\ &= \sum_k \sum_i n_{k,i} + \sum_k \sum_i n_{i,k} = 2 \sum_k \sum_i n_{k,i} \end{aligned}$$

即

$$\text{Micro F1} = \frac{2 \sum_k n_{k,k}}{2 \sum_k \sum_i n_{k,i}} = \frac{\sum_k n_{k,k}}{\sum_k \sum_i n_{k,i}} = \text{Accuracy}$$

12.2 数据探索

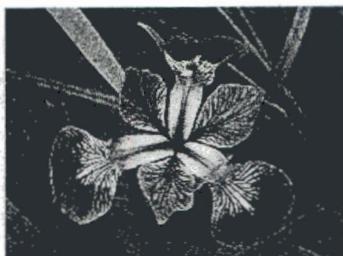
本章使用的数据集是鸢尾花 (Iris) 数据集。该数据集包含鸢尾属下的 3 个亚属（山鸢尾、变色鸢尾和维吉尼亚鸢尾），每个亚属各包含 50 个样本，该数据集共有 150 条数据。数据集有 4 个特征，分别是花萼 (Sepal) 和花瓣 (Petal) 的长度和宽度；包含一列分类信息，取值为亚属的名称 (Iris setosa、Iris virginica 和 Iris versicolor)。该数据集是由 Edgar Anderson 在加拿大加斯帕半岛测量收集的。Fisher, R.A. 在 1936 年以此数据集为例演示了判别分析方法。

为了给读者一些直观的印象，图 12-1 中给出了这 3 种鸢尾花的照片。如果读者想看到更清

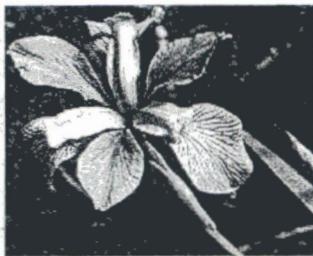
晰的图片并了解更多内容，可以访问链接 12-1。



山鸢尾 (Iris setosa)



变色鸢尾 (Iris virginica)



维吉尼亚鸢尾 (Iris versicolor)

图 12-1

下载数据文件（参见链接 12-2），使用文本编辑器打开，如图 12-2 所示。

```
iris.data
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
5.4,3.4,1.4,0.3,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

图 12-2

该数据文件是典型的 CSV 格式，可以使用 CsvSourceBatchOp 读取，并输出 5 条数据，计算、显示基本统计量和相关系数。相关代码如下。

```
static final String SCHEMA_STRING
    = "sepal_length double, sepal_width double, petal_length double, petal_width double, category
string";

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

source
    .lazyPrint(5, "origin file")
    .lazyPrintStatistics("stat of origin file")
    .link(
        new CorrelationBatchOp()
            .setSelectedCols(FEATURE_COL_NAMES)
            .lazyPrintCorrelation()
    );
}
```

显示 5 条数据如下。

```
origin file
sepal_length|sepal_width|petal_length|petal_width|category
-----|-----|-----|-----|-----
5. 1000|3. 5000|1. 4000|0. 2000|Iris-setosa
4. 9000|3. 0000|1. 4000|0. 2000|Iris-setosa
4. 7000|3. 2000|1. 3000|0. 2000|Iris-setosa
4. 6000|3. 1000|1. 5000|0. 2000|Iris-setosa
5. 0000|3. 6000|1. 4000|0. 2000|Iris-setosa
```

输出统计结果如下，共 150 条数据，没有缺失值。

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

相关系数矩阵如下，petal_length 和 petal_width 的相关系数最高，为 0.9628。

Correlation:

colName	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1	-0.1094	0.8718	0.818
sepal_width	-0.1094	1	-0.4205	-0.3565
petal_length	0.8718	-0.4205	1	0.9628
petal_width	0.818	-0.3565	0.9628	1

在使用 groupBy 时，要看一下标签值的分布情况。

```
source.groupBy(LABEL_COL_NAME, LABEL_COL_NAME + ", COUNT(*) AS cnt").lazyPrint(-1);
```

运行结果如下，各个标签值的分布平均都是 50 条样本。

category	cnt
Iris-setosa	50
Iris-versicolor	50
Iris-virginica	50

在开始实验前，将原始数据按 9:1 的比例分为训练集和测试集，具体代码如下。

```
Utils.splitTrainTestIfNotExist(source, DATA_DIR + TRAIN_FILE, DATA_DIR + TEST_FILE, 0.9);
```

12.3 使用朴素贝叶斯进行多分类

下面使用我们熟悉的朴素贝叶斯算法展开本书的第一次多分类试验。

使用朴素贝叶斯训练组件 `NaiveBayesTrainBatchOp`，与处理二分类问题的设置相似，还是设置特征列和标签列，只不过这里的标签列的标签值多于 2 个；设置朴素贝叶斯训练组件 `NaiveBayesPredictBatchOp`，设置预测结果列名称和预测详细信息列名称。将各组件连接起来，并设置训练组件，输出模型信息，预测组件输出 1 条数据，观察预测结果的具体内容，相关代码如下。

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

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

train_data.link(trainer);

predictor.linkFrom(trainer, test_data);

trainer.lazyPrintModelInfo();

predictor.lazyPrint(1, "< Prediction >");
```

输出的朴素贝叶斯模型信息如下，依次是特征列和标签列的基本信息、各个标签值所占比例、离散特征（categorical feature）和连续特征（gaussian feature）。当前数据集只有连续特征，输出了各特征和标签值所对应的均值和标准差，由这些信息可以计算高斯分布的概率。

```
===== model meta info =====
{label number: 3, feature size: 4, feature col names:
["sepal_length", "sepal_width", "petal_length", "petal_width"], labels:
["Iris-setosa", "Iris-versicolor", "Iris-virginica"]}
===== label proportion information =====

label info:[Iris-versicolor, Iris-virginica, Iris-setosa]
proportion:[0.35, 0.3167, 0.3333]
===== category information =====

categorical features: []
gaussian features: [sepal_length, sepal_width, petal_length, petal_width]
===== categorical features proportion information =====
```

There is no category feature.

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

Mean of features of each label:

	sepal_length	sepal_width	petal_length	petal_width
Iris-setosa	4.9925	3.3875	1.455	0.2425
Iris-versicolor	5.9571	2.7952	4.2857	1.3286
Iris-virginica	6.5211	2.9184	5.4947	2.0079

Std of features of each label:

	sepal_length	sepal_width	petal_length	petal_width
Iris-setosa	0.1242	0.1421	0.0255	0.0124
Iris-versicolor	0.1996	0.0857	0.1917	0.0359
Iris-virginica	0.4048	0.101	0.2973	0.0755

预测结果输出如下。

sepal_length	sepal_width	petal_length	petal_width	category	pred	pred_info
4.4000	2.9000	1.4000	0.2000	Iris-setosa	Iris-setosa	{"Iris-virginica": 6.932034333184479E-25, "Iris-versicolor": 3.811131021451496E-20, "Iris-setosa": 1.0}

右数第二列为预测标签值，可以直接和右数第三列的原始标签值进行比对；右数第一列为预测详细信息列，给出了每个标签值对应的概率，概率最大的那个就是预测结果标签值。

与二分类评估相比，多分类评估没有参数 PositiveLabelValueString（作为正例的标签值）；预测详情列是二分类评估的必选参数，但在多分类评估中是可选的，而且输入该信息只能对少数几个指标的计算有帮助，譬如 LogLoss 指标。所以在使用多分类评估组件时，我们一般只输入如下两个参数：标签列、分类预测结果列。

下面对朴素贝叶斯模型的预测结果进行多分类评估，使用多分类评估组件 EvalMultiClassBatchOp，设置上标签列 LabelCol 和分类预测结果列 PredictionCol，这里也设置了可选参数预测详细信息列 PredictionDetailCol，并选择使用 Lazy 的方式输出评估结果。具体代码如下。

```

predictor
.link(
  new EvalMultiClassBatchOp()
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .lazyPrintMetrics("NaiveBayes")
);

```

运行结果如下，输出了 Accuracy 和 Kappa，还有多分类评估时常关注的 Macro F1 和 Micro F1。

Metrics:			
Accuracy:0.9333 Macro F1:0.9267 Micro F1:0.9333 Kappa:0.8973	LogLoss:0.125		
Pred\Real	Iris-virginica	Iris-versicolor	Iris-setosa
Iris-virginica	12	2	0
Iris-versicolor	0	6	0
Iris-setosa	0	0	10

因为这两个 F1 是通过将所有标签值分别作为正例标签计算再汇总得到的，所以不需要参数指定正例标签值。LogLoss 指标的计算需要依赖 PredictionDetailCol，如果没有输入该参数列，则该指标为 null，就不会被输出。

从上面多分类评估混淆矩阵可以看出，有 2 个标签值为 Iris-versicolor 的样本被错误地预测为标签值 Iris-virginica，其他样本都被正确预测。后面我们还会介绍其他多分类器，看看能否有更好的效果。

12.4 二分类器组合

表 12-3 列出了常用的分类器所支持的分类情况。所有分类器都支持二分类情况，但是只有部分算法同时支持多分类情况。为了描述方便，下面将只支持二分类情况的分类器称为二分类器。本节将介绍如何将二分类器进行组合，解决多分类问题。

表 12-3 常用分类器支持二分类、多分类的情况

分类器名称	二分类	多分类
朴素贝叶斯	✓	✓
逻辑回归	✓	✗
线性 SVM	✓	✗
Softmax	✓	✓
多层感知器	✓	✓
FM	✓	✗
决策树	✓	✓
随机森林	✓	✓
GBDT	✓	✗
KNN	✓	✓

设问题的分类个数为 K ，也就是标签列有不同的标签值，常用的方式有如下两种。

(1) 一对多 (One-vs-Rest 或 One-vs-All)：使用二分类器的数量为 K ，与分类个数相同，但要求每个二分类器不仅给出分类结果，还要给出概率，用来确定最终的多分类结果。

(2) 一对一 (One-vs-One 或 pairwise)：使用二分类器数量较多，为 $\frac{K(K-1)}{2}$ ，但每个二分类器的训练样本数较少，最终由各个二分类器投票确定多分类结果。

举个例子，假设有 3 个类（也就是 3 个 Label 值）要划分，分别是 A、B、C。采用一对多方式，将其分为 3 个二分类子问题，然后将 3 个子问题的分类概率综合得到最终的结果。

- 二分类子问题 1：抽取 A 所对应的样本，对应的标签值改为 1；抽取 B、C 所对应的样本，对应的标签值改为 0。
- 二分类子问题 2：抽取 B 所对应的样本，对应的标签值改为 1；抽取 C、A 所对应的样本，对应的标签值改为 0。
- 二分类子问题 3：抽取 C 所对应的样本，对应的标签值改为 1；抽取 A、B 所对应的样本，对应的标签值改为 0。

分别进行训练后，得到 3 个二分类子模型，分别利用这 3 个子模型进行预测，得到各自对于正例标签值 1 的概率 $P_A(x)$ 、 $P_B(x)$ 、 $P_C(x)$ ，其中概率最大的便是最终的分类结果。

如果采用一对一方式，会将各个类别的数据进行两两组合，对于标签值 A、B、C，会组合成 3 个二分类子问题。

- 二分类子问题 1：将标签值为 A 和 B 的样本放在一起，是关于 A 和 B 的二分类问题。
- 二分类子问题 2：将标签值为 A 和 C 的样本放在一起，是关于 A 和 C 的二分类问题。
- 二分类子问题 3：将标签值为 B 和 C 的样本放在一起，是关于 B 和 C 的二分类问题。

分别进行训练后，得到 3 个二分类子模型，每个模型都会给出一个预测的标签值，然后采用投票的方式得到最终的分类结果。

Alink 提供了 One-vs-Rest 方法的实现，具体代码如下。重点在于 setClassifier 函数，可以在那里设置我们熟悉的二分类器，示例中使用的是逻辑回归算法，按照平常使用逻辑回归组件的方式设置其相应参数。对于 OneVsRest 组件，需要指定其多分类的个数（这个参数是必需的），还要设置组件最终输出的预测结果列，也可选择设置输出预测详细信息列。

```

new OneVsRest()
    .setClassifier(
        new LogisticRegression()
            .setFeatureCols(FEATURE_COL_NAMES)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
    .setNumClass(3)

```

```

    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("OneVsRest_LogisticRegression")
    );

```

评估结果如下，相比于 12.3 节中朴素贝叶斯的评估结果，这次没有设置预测详细信息列参数，所以没有输出 LogLoss 指标。在预测指标的数值上，此次结果较朴素贝叶斯有全面提升，只有 1 个标签值为 Iris-versicolor 的样本被错误地预测为标签值 Iris-virginica。

Metrics:			
Pred\Real	Iris-virginica	Iris-versicolor	Iris-setosa
Iris-virginica	12	1	0
Iris-versicolor	0	7	0
Iris-setosa	0	0	10

如果我们将逻辑回归二分类器换为分类能力更强的 GBDT 二分类器，分类效果是否会更好呢？

试验代码如下。

```

new OneVsRest()
    .setClassifier(
        new GbdtClassifier()
            .setFeatureCols(FEATURE_COL_NAMES)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
    .setNumClass(3)

    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("OneVsRest_GBDT")
    );

```

与使用逻辑回归二分类器的代码相比，只是在 setClassifier 函数内部有变化，设置为 GBDT 二分类器 GbdtClassifier，并可设置其特有的参数。这里使用了参数的默认值，只设置了必需的特征列和标签列名称。

运行结果如下，各项指标与使用逻辑回归二分类器相同。

Metrics:				
Pred\Real	Iris-virginica	Iris-versicolor	Iris-setosa	
Iris-virginica	12	1	0	
Iris-versicolor	0	7	0	
Iris-setosa	0	0	10	

尝试修改了几次 GBDT 参数，但是没能进一步优化分类效果。这时我们尝试使用一个二分类器——LinearSVM。输入 setClassifier 函数，设置为线性 SVM 分类器 LinearSvm，使用参数的默认值，并设置必需的特征列和标签列名称。相关代码如下。

```
new OneVsRest()
.setClassifier(
  new LinearSvm()
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
)
.setNumClass(3)

.fit(train_data)
.transform(test_data)
.link(
  new EvalMultiClassBatchOp()
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .lazyPrintMetrics("OneVsRest_LinearSvm")
);

```

运行结果如下，全部实现正确分类！

Metrics:				
Pred\Real	Iris-virginica	Iris-versicolor	Iris-setosa	
Iris-virginica	12	0	0	
Iris-versicolor	0	8	0	
Iris-setosa	0	0	10	

我们看到了二分类器组合可以解决多分类问题，同时也能感受到，二分类器分类能力的强弱不是绝对的，需要根据具体的问题灵活选择合适的二分类器。

12.5 Softmax算法

Softmax 将多分类问题用一个损失函数描述，当分类数为 2 时，其损失函数与逻辑回归的

损失函数相同。我们可以将 Softmax 算法看作逻辑回归算法在多分类问题上的扩展。

设分类个数为 K , 标签值的取值范围为 $\{0, 1, 2, \dots, K - 1\}$ 。

设示性函数 (Indicative Function) 为

$$I(y = k) = \begin{cases} 1, & \text{当 } y = k \text{ 为真} \\ 0, & \text{当 } y = k \text{ 为假} \end{cases}$$

用一个 $(K - 1) \times (m + 1)$ 维的向量来表示模型权重系数 W , 该向量被等分为 $K - 1$ 段, 分别为

$$w^{(1)}, w^{(2)}, \dots, w^{(K-1)}$$

则

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

定义当 x 分类为标签值 $1 \leq k \leq K - 1$ 时的概率为

$$\phi_k(x) = \frac{e^{\eta(w^{(k)}, x)}}{1 + \sum_{i=1}^{K-1} e^{\eta(w^{(i)}, x)}}$$

并定义

$$\phi_0(x) = \frac{1}{1 + \sum_{i=1}^{K-1} e^{\eta(w^{(i)}, x)}}$$

则 $\{\phi_0(x), \phi_1(x), \dots, \phi_{K-1}(x)\}$ 满足如下等式

$$\sum_{k=0}^{K-1} \phi_k(x) = \frac{1}{1 + \sum_{i=1}^{K-1} e^{\eta(w^{(i)}, x)}} + \frac{\sum_{k=1}^{K-1} e^{\eta(w^{(k)}, x)}}{1 + \sum_{i=1}^{K-1} e^{\eta(w^{(i)}, x)}} = \frac{1 + \sum_{k=1}^{K-1} e^{\eta(w^{(k)}, x)}}{1 + \sum_{i=1}^{K-1} e^{\eta(w^{(i)}, x)}} = 1$$

定义损失函数为

$$L(W, x^{(i)}, y^{(i)}) = -\log \left(\prod_{k=0}^{K-1} (\phi_k(x^{(i)}))^{I(y^{(i)}=k)} \right)$$

注意：

当 $K = 2$ 时, 有

$$L(W, x^{(i)}, y^{(i)}) = -\log \left(\prod_{k=0}^1 (\phi_k(x^{(i)}))^{I(y^{(i)}=k)} \right)$$

当 $y^{(i)} = 1$ 时，有

$$L(W, x^{(i)}, y^{(i)}) = -\log \left(\frac{e^{\eta(w^{(1)}, x)}}{1 + e^{\eta(w^{(1)}, x)}} \right) = \log \left(1 + e^{-\eta(w^{(1)}, x)} \right)$$

当 $y^{(i)} = 0$ 时，有

$$L(W, x^{(i)}, y^{(i)}) = -\log \left(\frac{1}{1 + e^{\eta(w^{(1)}, x)}} \right) = \log \left(1 + e^{\eta(w^{(1)}, x)} \right)$$

将此结果与二分类逻辑回归算法的损失函数（标签值为 0、1 的情形）对比，会发现结果一致，即在分类数 $K = 2$ 时，多分类器的损失函数与逻辑回归的损失函数相同。

Softmax 的经验损失函数为

$$\begin{aligned} L(W) &= \frac{1}{n} \sum_{i=1}^n L(W, x^{(i)}, y^{(i)}) \\ &= \frac{1}{n} \sum_{i=1}^n \left[-\log \left(\prod_{k=0}^{K-1} (\phi_k(x^{(i)}))^{I(y^{(i)}=k)} \right) \right] \\ &= -\frac{1}{n} \sum_{i=1}^n \sum_{k=0}^{K-1} [I(y^{(i)}=k) \cdot \log(\phi_k(x^{(i)}))] \end{aligned}$$

则其结构风险函数为

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

Softmax 算法会输出标签值 $\{0, 1, 2, \dots, K-1\}$ 对应的概率，即

$$\{\phi_0(x), \phi_1(x), \dots, \phi_{K-1}(x)\}$$

并选出概率最大的标签值作为预测结果。在实际的多分类问题中，标签值往往是一些有实际意义的标签，Alink 的 Softmax 组件会在训练前、预测后将实际标签值与 $\{0, 1, 2, \dots, K-1\}$ 进行自动转换，用户在使用时不需要关注标签值的形式问题。

我们使用 Softmax 组件解决多分类问题，设置 Softmax 组件的参数，包括特征列、标签列和预测结果列名称，设置使用 Lazy 方式输出训练过程信息和模型信息，具体代码如下。

```
new Softmax()
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .enableLazyPrintTrainInfo()
    .enableLazyPrintModelInfo()
    .fit(train_data)
```

```

    .transform(test_data)
    .link(
      new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("Softmax")
    );
}

```

训练信息如下。

```

----- train meta info -----
{model name: softmax, num feature: 4}
----- train convergence info -----
step:0 loss:1.03883853 gradNorm:0.78880934 learnRate:0.40000000
step:1 loss:0.79029177 gradNorm:0.72704714 learnRate:1.60000000
step:2 loss:0.46450488 gradNorm:0.45492455 learnRate:4.00000000
...
step:33 loss:0.04431295 gradNorm:0.00016576 learnRate:4.00000000
step:34 loss:0.04431267 gradNorm:0.00007317 learnRate:4.00000000
step:35 loss:0.04431267 gradNorm:0.00000423 learnRate:4.00000000

```

该组件默认迭代次数为 100，可以看到在 step:35 时就已达到了收敛条件。

对于 K 分类问题，Softmax 会有 $K - 1$ 组线性参数，模型信息如下。

```

----- model meta info -----
{hasInterception: true, model name: softmax, num feature: 4}
----- model label values -----
[Iris-setosa, Iris-versicolor, Iris-virginica]
----- model weight info -----
|intercept|sepal_length|sepal_width|petal_length| petal_width|
|-----|-----|-----|-----|-----|
| 115.6539| -22.13402422| 32.05813365| -14.44968333| -31.20994729|
| 43.1648| 1.57043487| 6.21071882| -8.54629750| -17.20480213|

```

对于本章的三分类问题，有 2 组线性参数，详见模型系数信息（model weight info）显示的内容。

模型评估结果如下，也做到了完全分类！

```

----- Metrics: -----
Accuracy:1 Macro F1:1 Micro F1:1 Kappa:1
| Pred\Real|Iris-virginica|Iris-versicolor|Iris-setosa|
|-----|-----|-----|-----|
| Iris-virginica| 12| 0| 0|
| Iris-versicolor| 0| 8| 0|
| Iris-setosa| 0| 0| 10|

```

Softmax 有很好的分类效果，而且使用简单，基本不需要调整参数。在处理多分类问题时，我们可以将其作为首选的算法。

12.6 多层感知器分类器

多层感知器分类器 (Multi-Layer Perceptron Classifier, MLPC) 是一种基于前向神经网络的分类器。使用常用设置等方式，简化了使用前向神经网络进行分类操作的过程，用户只需设置各隐藏层的节点个数。

MLPC 由 $L + 1$ 层节点组成，每层节点都可以对应一个向量。第一层为输入层，对应输入特征向量，节点个数即为特征数，也是特征向量的维度；最后一层为输出层，对应各个分类标签的概率向量，节点个数为分类标签的个数，也是概率向量的维度；中间的 $L - 1$ 层为隐藏层。

每层完全连接到网络中的下一层，第 i 层通过节点权重矩阵 \mathbf{W}_i 和偏差向量 \mathbf{b}_i 的线性组合，并应用激活函数或 Softmax 函数，得到第 $i + 1$ 层节点。对于具有 $L + 1$ 层的 MLPC，可以采用矩阵形式编写，如下所示。

$$f(x) = s_L(\mathbf{W}_L \cdots s_2(\mathbf{W}_2 s_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \cdots) + \mathbf{b}_L$$

其中，对于 $k = 1, 2, \dots, L - 1$ ，即前面 $L - 1$ 层，激活 (Sigmoid) 函数选择 Logistic 函数，即

$$s_k(z_i) = \frac{1}{1 + e^{-z_i}} = \frac{e^{z_i}}{1 + e^{z_i}}$$

Logistic 函数将 $(-\infty, +\infty)$ 的值映射到区间 $(0, 1)$ ，其函数曲线如图 12-3 所示。

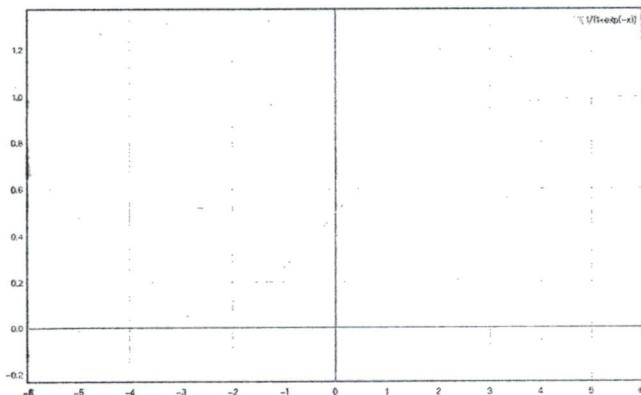


图 12-3

第 L 层使用 Softmax 函数，得到最终的输出层节点，各标签值的概率为

$$s_k(z_i) = \frac{e^{z_i}}{\sum_{i=1}^K e^{z_i}}$$

其中， K 为不同标签值的个数，即分类总数，也是输出层的节点数。

下面通过示例了解多层感知器分类器的使用，具体代码如下。

```
new MultilayerPerceptronClassifier()
.setLayers(new int[] {4, 20, 3})
.setFeatureCols(FEATURE_COL_NAMES)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.fit(train_data)
.transform(test_data)
.link(
    new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("MultilayerPerceptronClassifier [4, 20, 3]")
);
)
```

使用多层感知器分类器组件 `MultilayerPerceptronClassifier`，设置特征列和标签列、预测结果列名称，该组件特有的参数是 `Layers`（神经网络各层神经元的数量）。第一层（输入层）神经元的数量要与特征数相同，Iris 数据集有 4 个特征；最后一层（输出层）神经元的数量等于分类的类别数，Iris 数据集的类别数（标签值个数）为 3；中间为若干个隐层，这里设置了一个隐层，有 20 个神经元。

模型评估结果如下，也做到了完全分类。

Metrics:				
Accuracy	Macro F1	Micro F1	Kappa	
Pred\Real	Iris-virginica	Iris-versicolor	Iris-setosa	
Iris-virginica	12	0	0	
Iris-versicolor	0	8	0	
Iris-setosa	0	0	10	

多层感知器分类器的最后一层与倒数第二层之间使用的激活函数是 `Softmax` 函数。如果没有隐层，那么相当于直接对输入层的原始特征进行线性组合，并使用 `Softmax` 激活函数，等价于 12.5 节的 `Softmax` 算法。多层感知器分类器可以被看作 `Softmax` 分类器的扩展。

我们在前面多层感知器分类器试验的基础上，调整 `Layers` 参数，使其只包含输入层和输出层，即 `Layers={4, 3}`。相关代码如下。

```
new MultilayerPerceptronClassifier()
.setLayers(new int[] {4, 3})
.setFeatureCols(FEATURE_COL_NAMES)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.fit(train_data)
.transform(test_data)
```

```
.link(  
    new EvalMultiClassBatchOp()  
        .setLabelCol(LABEL_COL_NAME)  
        .setPredictionCol(PREDICTION_COL_NAME)  
        .lazyPrintMetrics("MultilayerPerceptronClassifier [4, 3]")  
);
```

运行结果如下，同样得到了 Softmax 分类器的效果。

```
Metrics:  
Accuracy:1 Macro F1:1 Micro F1:1 Kappa:1  
| Pred\Real|Iris-virginica|Iris-versicolor|Iris-setosa|  
|-----|-----|-----|-----|  
| Iris-virginica|      12|         0|       0|  
| Iris-versicolor|       0|        8|       0|  
| Iris-setosa|        0|         0|      10|
```



常用多分类算法

MNIST 是常用的手写数字识别数据集，包含一个 60000 样本的训练集和 10000 样本的测试集。如图 13-1 所示，这些手写数字都位于图像中心，图像尺寸都被规范到 28 像素×28 像素的固定大小。像素值范围为 0~255，0 表示背景（白色），255 表示前景（黑色）。数据集下载地址为链接 13-1。



图 13-1

注意：本章只是为了演示通用的多分类算法，将图像的每个元素都看作一个特征。使用图像方面的深度学习算法可以获得更好的分类效果，但不在本章的讨论范围内，感兴趣的读者可以参阅相关材料。

13.1 数据准备

MNIST 的原始数据是一种自定义的格式，需要按照网站上的格式说明，单独写程序读取。

首先将 4 个数据文件下载到本地文件夹，训练数据的图像数据部分与标签部分是独立的两个文件，预测数据也分为两个文件。

- (1) train-images-idx3-ubyte.gz: training set images (9912422 bytes)
- (2) train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
- (3) t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
- (4) t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

13.1.1 读取MNIST数据文件

在附录代码中，有一段读取 MNIST 数据的代码，是根据指定的图像数据文件与标签文件生成的形式为“特征向量+标签”的数据表，定义为 Alink Source 形式。代码的主体部分如下，构造函数需要输入图像数据文件路径、标签文件路径和参数 isSparse。

```
public static class MnistGzFileSourceBatchOp extends BaseSourceBatchOp
<MnistGzFileSourceBatchOp> {

    private final String imageGzFile;
    private final String labelGzFile;
    private final boolean isSparse;

    public MnistGzFileSourceBatchOp(String imageGzFile, String labelGzFile, boolean isSparse) {
        super(null, null);
        this.imageGzFile = imageGzFile;
        this.labelGzFile = labelGzFile;
        this.isSparse = isSparse;
    }

    @Override
    protected Table initializeDataSource() {
        try {
            ArrayList<Row> rows = new ArrayList<>();
            String[] images = getImages();
            Integer[] labels = getLabels();
            int n = images.length;
            if (labels.length != n) {
                throw new RuntimeException("The size of images IS NOT EQUAL WITH the size of labels.");
            }
            for (int i = 0; i < n; i++) {
                rows.add(Row.of(images[i], labels[i]));
            }
            return new MemSourceBatchOp(rows, new String[] {"vec", "label"}).getOutputTable();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        }
    }
}
```

 } ...

`initializeDataSource` 方法包含了主要流程，使用 `getImages` 方法从文件读取图像数据向量序列化后的 String 类型数据；使用 `getLabels` 方法从文件获取标签信息，然后组装为数据表的形式，返回产生的数据表。

其中，读取数据文件的核心函数如下。

```

private Integer[] getLabels() throws IOException {
    BufferedInputStream bis = new BufferedInputStream(
        new GZIPInputStream(new FileInputStream(this.labelGzFile)));
    ...
    return labels;
}

private String[] getImages() throws IOException {
    BufferedInputStream bis = new BufferedInputStream(
        new GZIPInputStream(new FileInputStream(this.imageGzFile)));
    ...
    String[] images = new String[record_number];
    if (isSparse) {
        ...
        for (int i = 0; i < record_number; i++) {
            ...
            images[i] = new SparseVector(nPixels, pixels).toString();
        }
    } else {
        ...
        for (int i = 0; i < record_number; i++) {
            ...
            images[i] = new DenseVector(image).toString();
        }
    }
    bis.close();
    return images;
}

```

通过先获取原始文件流 `FileInputStream`，然后使用 `GZIPInputStream` 进行解压，再使用 `BufferedInputStream` 提高读数据的性能，之后便是各自按格式进行解析、输出。注意：变量 `isSparse` 决定了生成的向量格式为稀疏或稠密，函数返回向量序列化后的 String 类型。

13.1.2 稠密向量与稀疏向量

基于 `MnistGzFileSourceBatchOp` 可以轻松得到稠密、稀疏向量的训练和预测数据。为了便于后续操作使用，我们直接将其存为 AK 格式，具体代码如下。

```

new MnistGzFileSourceBatchOp
(
    DATA_DIR + "train-images-idx3-ubyte.gz",
    DATA_DIR + "train-labels-idx1-ubyte.gz",
    true
)
.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE)
);
new MnistGzFileSourceBatchOp
(
    DATA_DIR + "t10k-images-idx3-ubyte.gz",
    DATA_DIR + "t10k-labels-idx1-ubyte.gz",
    true
)
.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + SPARSE_TEST_FILE)
);
new MnistGzFileSourceBatchOp
(
    DATA_DIR + "train-images-idx3-ubyte.gz",
    DATA_DIR + "train-labels-idx1-ubyte.gz",
    false
)
.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + DENSE_TRAIN_FILE)
);
new MnistGzFileSourceBatchOp
(
    DATA_DIR + "t10k-images-idx3-ubyte.gz",
    DATA_DIR + "t10k-labels-idx1-ubyte.gz",
    false
)
.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + DENSE_TEST_FILE)
);
BatchOperator.execute();

```

对于稠密训练数据，输出一条数据，看看其具体内容，并对其向量列进行统计，具体代码如下。

```

new AkSourceBatchOp()
.setFilePath(DATA_DIR + DENSE_TRAIN_FILE)
.lazyPrint(1, "MNIST data")
.link(
    new VectorSummarizerBatchOp()
        .setSelectedCol(VECTOR_COL_NAME)
        .lazyPrintVectorSummary()
);

```

数据内容如下。

```
vec|label
---|---
0.0 0.0 0.0 0.0 ... 230.0 132.0 133.0 132.0 132.0 ... 0.0 0.0 0.0 |3
```

右边一列为标签值，这条数据的标签值为 3；左边为稠密向量序列化为字符串的形式，以空格分隔各个数字，由于向量内容较多，这里只保留少量内容进行示意。

向量列统计结果如下。

Summary										
id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2	
0	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
1	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
2	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
...	
774	60000	12026.0000	0.2004	36.5115	6.0425	0.0000	254.0000	12026.0000	1480.8991	
775	60000	5332.0000	0.0889	15.6514	3.9562	0.0000	254.0000	5332.0000	969.3008	
776	60000	2738.0000	0.0456	8.0647	2.8398	0.0000	253.0000	2738.0000	695.7011	
777	60000	1157.0000	0.0193	2.8452	1.6868	0.0000	253.0000	1157.0000	413.1961	
778	60000	907.0000	0.0151	2.8166	1.6783	0.0000	254.0000	907.0000	411.1070	
779	60000	120.0000	0.0020	0.1201	0.3466	0.0000	62.0000	120.0000	84.8999	
780	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
781	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
782	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
783	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	

其中给出了各个维度的统计量，最左边为向量的索引号，后面是该索引位置的各数据的统计指标。

再来看稀疏训练数据，输出一条数据，看看其具体内容，并对其向量列进行统计，具体代码如下。

```
new AkSourceBatchOp()
    .setFilePath(DATA_DIR + SPARSE_TRAIN_FILE)
    .lazyPrint(1, "MNIST data")
    .link(
        new VectorSummarizerBatchOp()
            .setSelectedCol(VECTOR_COL_NAME)
            .lazyPrintVectorSummary()
    );
}
```

数据内容如下。

```
vec|label
---|---
$784$158:124.0 159:253.0 160:255.0 161:63.0 ... 683:220.0 |1
```

右边一列为标签值，这条数据的标签值为1；左边为稀疏向量序列化为字符串的形式，左边以“\$”起始的为稀疏向量维度，向量索引和数值间以“：“间隔，以空格分隔各个索引数值对。由于向量内容较多，这里只保留首尾内容进行展示。

稀疏向量列统计结果如下。

Summary										
id	count	sum	mean	variance	standardDeviation	min	max	normL1	normL2	
0	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
1	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
2	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
3	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
4	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
5	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
6	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
7	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
8	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
9	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
10	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
11	60000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
12	60000	126.0000	0.021	0.2259	0.4753	0.0000	116.0000	126.0000	116.4302	
13	60000	470.0000	0.0078	1.8528	1.3612	0.0000	254.0000	470.0000	333.4247	
...										
776	60000	2738.0000	0.0456	8.0647	2.8398	0.0000	253.0000	2738.0000	695.7011	
777	60000	1157.0000	0.0193	2.8452	1.6868	0.0000	253.0000	1157.0000	413.1961	
778	60000	907.0000	0.0151	2.8166	1.6783	0.0000	254.0000	907.0000	411.1070	
779	60000	120.0000	0.0020	0.1201	0.3466	0.0000	62.0000	120.0000	84.8999	

其中给出了各个维度的统计量，最左边为向量的索引号，后面是该索引位置的数据的统计指标。

对比稀疏向量统计结果与稠密向量统计结果，我们会发现一点差异：稀疏向量统计输出的是数据中实际出现的最大索引值，稠密向量统计输出的是全部索引值。

13.1.3 标签值的统计信息

对稀疏训练数据执行一般的全表统计，向量列会被忽略，代码如下。

```
new AkSourceBatchOp()
    .setFilePath(DATA_DIR + SPARSE_TRAIN_FILE)
    .lazyPrintStatistics()
    .groupBy(LABEL_COL_NAME, LABEL_COL_NAME + ", COUNT(*) AS cnt")
    .orderBy("cnt", 100)
    .lazyPrint(-1);
```

统计结果如下。

colName	count	missing	sum	mean	variance	min	max
vec	60000	0	NaN	NaN	NaN	NaN	NaN
label	60000	0	267236	4.4539	8.3479	0	9

共有 60000 条训练数据，没有缺失值，主要统计了标签值的信息，从 0 到 9，均值为 4.4539。

关于每个标签值对应的样本数，还需要看 groupBy 的结果。

label	cnt
5	5421
4	5842
8	5851
6	5918
0	5923
9	5949
2	5958
3	6131
7	6265
1	6742

显然，数字“5”的样本数最少，数字“1”的样本数最多，各数字的样本数在平均样本数 6000 左右浮动。

13.2 Softmax 算法

对于向量特征的多分类，我们先尝试使用 Softmax 算法。选择稀疏向量格式的训练和预测数据集，在本章后续的试验中也会使用同样的数据集，就不再重复描述。

使用 Softmax 组件，设置向量列名称和标签列名称，以及预测列的名称；选择使用 Lazy 方法输出显示训练过程的信息及模型信息，以便了解更多信息。具体代码如下。

```
AkSourceBatchOp train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);
AkSourceBatchOp test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TEST_FILE);

new Softmax()
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .enableLazyPrintTrainInfo()
    .enableLazyPrintModelInfo()
    .fit(train_data)
```

```

.transform(test_data)
.link(
    new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("Softmax")
);

```

训练信息如下。

```

----- train meta info -----
{model name: softmax, num feature: 784}
----- train convergence info -----
step:0 loss:2.20390243 gradNorm:1.00628876 learnRate:0.40000000
step:1 loss:1.19250130 gradNorm:0.95834731 learnRate:1.60000000
step:2 loss:0.90806251 gradNorm:0.94712724 learnRate:1.60000000
...
step:97 loss:0.24306077 gradNorm:0.00665607 learnRate:4.00000000
step:98 loss:0.24283406 gradNorm:0.00800750 learnRate:4.00000000
step:99 loss:0.24251477 gradNorm:0.00579344 learnRate:4.00000000

```

训练是达到最大迭代轮数（默认值为100）而退出的，loss的变化已经很小了。

模型信息显示如下。

```

----- model meta info -----
{hasInterception: true, model name: softmax, num feature: 784, vector colName: vec}
----- model label values -----
[0, 1, 2, ..., 7, 8, 9]
----- model weight info -----
|intercept| 1| 2| 3| 4| 5| 6| 7| 8|... ...| |
|---|---|---|---|---|---|---|---|---|---|---|
|-1.6538|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|... ...|
| 0.8954|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|... ...|
| -0.0636|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|... ...|
| -0.825|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|0.0000000|... ...|
| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...| ... ...|

```

在模型系数权重信息中显示了部分系数，截距项系数都有数值，但向量前8项的系数都为0.0，是不是算错了呢？我们看一下稀疏向量的统计结果，发现前12项的最大值、最小值都为0.0，即这些项对分类结果都没有影响。

最后，我们看一下分类评估结果。

```

----- Metrics: -----
Accuracy:0.9252 Macro F1:0.9242 Micro F1:0.9252 Kappa:0.9169
|Pred\Real| 9| 8| 7|...| 2| 1| 0|
|-----|---|---|---|---|---|---|---|
| 9|920| 11| 31|...| 5| 0| 0|
| 8| 10|860| 3|...| 35| 11| 1|
| 7| 22| 6|949|...| 9| 2| 4|

```


	2	0	9	22	...	929	4	1
	1	7	10	8	...	8	1111	0
	0	8	8	1	...	3	0	958

精确度为 0.9252，各数字的绝大部分样本被正确预测。

13.3 二分类器组合

下面我们使用二分类器组合 One-vs-Rest 方法，并选择逻辑回归作为二分类器。因为 One-vs-Rest 会将 10 个逻辑回归二分类器连成一个任务，在内存资源消耗上要远高于 Softmax，而且当前训练集的数据量也较多，所以我们通过减小并发度来减少资源的需求。建议在分类数较少的情况下使用 One-vs-Rest 方法。

注意：如何设置并行度（Parallelism）？

设置当前环境执行的并行度，当前任务所涉及的所有批式组件或流式组件都会按此并行度进行并行执行。

向集群提交分布式任务时，并发度是必填参数，整个任务分布运行在多台机器上。

在本地环境执行时，如果不指定并行度，那么它会根据机器硬件配置的不同而被默认指定，并行度的默认值为本地环境的 CPU 核心/线程数。

可以使用如下方法设置批式任务、流式任务的并发度参数。

```
BatchOperator.setParallelism(3);
StreamOperator.setParallelism(3);
```

实验代码如下，需要注意的是，第一行设置了批式组件的并发度为 1。

```
BatchOperator.setParallelism(1);

new OneVsRest()
.setClassifier(
    new LogisticRegression()
        .setVectorCol(VECTOR_COL_NAME)
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
)
.setNumClass(10)

.fit(train_data)
.transform(test_data)
.link(
    new EvalMultiClassBatchOp()
        .setLabelCol(LABEL_COL_NAME)
```

```
.setPredictionCol(PREDICTION_COL_NAME)
.lazyPrintMetrics("OneVsRest - LogisticRegression")
);
```

运行结果如下，精确度为 0.919。

Metrics:

Accuracy:0.919 Macro F1:0.9176 Micro F1:0.919 Kappa:0.91									
Pred\Real	9	8	7	...	2	1	0		
9	896	13	27	...	4	0	2		
8	10	842	4	...	39	12	1		
7	25	10	947	...	9	1	4		
...	
2	1	6	22	...	921	3	1		
1	8	11	8	...	5	1112	0		
0	8	11	3	...	8	0	962		

再将逻辑回归换成线性 SVM，运行结果如下。

Metrics:

Accuracy:0.9214 Macro F1:0.9202 Micro F1:0.9214 Kappa:0.9126									
Pred\Real	9	8	7	...	2	1	0		
9	899	11	25	...	3	0	1		
8	12	853	3	...	40	9	1		
7	23	11	952	...	11	1	3		
...	
2	1	5	21	...	921	3	1		
1	8	10	8	...	7	1113	0		
0	7	11	2	...	8	0	958		

精确度比使用逻辑回归二分类器的精确度高 0.0024，但比 Softmax 低 0.0038。

13.4 多层感知器分类器

前面介绍过，多层感知器分类器（MLPC）在没有隐层时相当于 Softmax 分类器，下面我们就先从没有隐层的网络开始。具体代码如下，输入向量的维度是 784，预测的类别数是 10，设置 Layers={784, 10}。

```
new MultilayerPerceptronClassifier()
.setLayers(new int[] {784, 10})
.setVectorCol(VECTOR_COL_NAME)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
```

```

    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("MultilayerPerceptronClassifier {784, 10}")
    );

```

运行结果如下。

Metrics:									
Accuracy:0.9241 Macro F1:0.923 Micro F1:0.9241 Kappa:0.9156									
Pred\Real	9	8	7	...	2	1	0		
9	920	12	28	...	4	0	0		
8	9	858	4	...	39	11	1		
7	20	7	951	...	7	2	4		
...		
2	1	8	23	...	926	3	0		
1	7	9	7	...	9	1112	0		
0	10	9	1	...	4	0	958		

前面使用 Softmax 算法的精确度是 0.9252，与本次试验的精确度 0.9241 相比，可以说是相近的。感兴趣的读者可以尝试提高迭代次数，即使用 setMaxIter 方法，默认迭代 100 次，提高到 150 次后，得到的精确度是 0.9259。

下面尝试设计多层网络，优化分类效果。设置向量列名称、标签列名称，以及预测列的名称，神经网络的输入层为 784 个特征，输出层对应 10 个标签值，中间设置了 2 个隐层，神经元个数分别为 256 和 128。具体代码如下。

```

new MultilayerPerceptronClassifier()
    .setLayers(new int[] {784, 256, 128, 10})
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("MultilayerPerceptronClassifier {784, 256, 128, 10}")
    );

```

由于该训练的时间较长，大约 9 分钟，建议做如下设置。

```
AlinkGlobalConfiguration.setPrintProcessInfo(true);
```

这样可以看到训练过程中的一些输出内容，了解训练过程的进度。

运行结束后，评估结果如下。

```
Accuracy:0.971 Macro F1:0.9708 Micro F1:0.971 Kappa:0.9678
|Pred\Real| 9| 8| 7|...| 2| 1| 0|
+-----+---+---+---+---+---+---+
| 9|960| 1| 7|...| 0| 1| 2|
| 8| 3|944| 1|...| 4| 5| 2|
| 7|14| 5|1006|...| 6| 0| 3|
| ...|...|...|...|...|...|...|...
| 2| 4| 5| 5|...|1007| 3| 0|
| 1| 3| 0| 3|...| 2|1121| 0|
| 0| 3| 2| 0|...| 4| 0|962|
```

精确度为 0.971，各项指标明显高于 Softmax 算法。

我们也可以尝试更多的迭代次数，评估指标还可以再提高，如表 13-1 所示。

表 13-1 不同迭代次数对应的评估指标对比

迭代次数	Accuracy	Macro F1	Micro F1	Kappa	运行时间
100	0.971	0.9708	0.971	0.9678	9 分
150	0.9774	0.9772	0.9774	0.9749	15 分 37 秒
200	0.9781	0.9779	0.9781	0.9757	21 分 40 秒

13.5 决策树与随机森林

决策树也可以进行多分类，但通常单棵决策树的分类效果不是最优的，实际中常用多棵决策树构成的随机森林算法。

决策树和随机森林算法组件不支持输入向量特征，需要使用 Alink 组件 VectorToColumns 进行一次转换。需要设置输入的向量 VectorCol，设置输出的参数 SchemaStr，选择保留标签列（如果不设置保留列，那么默认会保留所有列），如下代码所示。

```
new VectorToColumns()
.setVectorCol(VECTOR_COL_NAME)
.setSchemaStr(sbd.toString())
.setReservedCols(LABEL_COL_NAME)
.transform(train_sparse)
.link(
    new AkSinkBatchOp().setFilePath(DATA_DIR + TABLE_TRAIN_FILE)
);
```

其中参数 SchemaStr 的信息是这样生成的，使用“_c”作为列名前缀，后面接向量索引值，

如下所示。

```
StringBuilder sbd = new StringBuilder();
sbd.append("c_0 double");
for (int i = 1; i < 784; i++) {
    sbd.append(", c_").append(i).append(" double");
}

```

转换后的数据包含 1 个标签列（label）、784 个特征列（c_0 到 c_783），输出数据后，每行无法完整显示，所以截图显示了左边的部分列，如图 13-2 所示。

图 13-2

最左边为标签列，随后是各特征列。

接下来，我们试验各决策树的多分类效果，由于数据量较大，训练时间偏长，我们定义一个停表对象 `Stopwatch sw = new Stopwatch()`，用来记录各次试验的时间。具体代码如下所示。

```
for (TreeType treeType : new TreeType[] {TreeType.GINI, TreeType.INFOGAIN, TreeType.INFOGAINRATIO}) {
    sw.reset();
    sw.start();
    new DecisionTreeClassifier()
        .setTreeType(treeType)
        .setFeatureCols(featureColNames)
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .enableLazyPrintModelInfo()
        .fit(train_data)
        .transform(test_data)
        .link(
            new EvalMultiClassBatchOp()
                .setLabelCol(LABEL_COL_NAME)
                .setPredictionCol(PREDICTION_COL_NAME)
                .lazyPrintMetrics("DecisionTreeClassifier " + treeType.toString())
        );
    BatchOperator.execute();
    sw.stop();
    System.out.println(sw.getElapsed TimeSpan());
}
```

计算结果汇总如表 13-2 所示。

表 13-2 各决策树的评估指标对比

决策树	Accuracy	Macro F1	Micro F1	Kappa	时间
GINI	0.876	0.8746	0.876	0.8622	1 分 59 秒
INFOGAIN	0.881	0.8793	0.881	0.8677	1 分 32 秒
INFOGAINRATIO	0.863	0.861	0.863	0.8477	5 分 42 秒

可以看到，对于当前数据集，INFOGAIN 类型决策树（即 ID3 算法）的各项指标最高，而且用时最短。单棵决策树的多分类指标与前面介绍的几种算法有很大差距，但是利用多棵树构成随机森林可以获得更好的分类效果。

下面的试验选择了 INFOGAIN 类型决策树，使用 setNumTreesOfInfoGain 方法，调整随机森林中 INFOGAIN 类型决策树的棵数。森林由 2 棵树开始，不断翻倍，直到 128 棵树。设置了当前批式任务的并发度为 4，对于随机森林算法的实现，每个并发的 worker 会执行一棵决策树的训练；在计算任务的前后设置了停表计时；设置了参数 SubsamplingRatio=0.6，每棵决策树会对整个训练数据采样 60% 作为其训练数据。具体代码如下。

```

BatchOperator.setParallelism(4);

.....
for(int numTrees: new int[]{2, 4, 8, 16, 32, 64, 128}){
    sw.reset();
    sw.start();
    new RandomForestClassifier()
        .setSubsamplingRatio(0.6)
        .setNumTreesOfInfoGain(numTrees)
        .setFeatureCols(featureColNames)
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .enableLazyPrintModelInfo()
        .fit(train_data)
        .transform(test_data)
        .link(
            new EvalMultiClassBatchOp()
                .setLabelCol(LABEL_COL_NAME)
                .setPredictionCol(PREDICTION_COL_NAME)
                .lazyPrintMetrics("RandomForestClassifier : InfoGain - " + numTrees)
        );
    BatchOperator.execute();
    sw.stop();
    System.out.println(sw.getElapsedTimeSpan());
}

```

汇总计算结果如表 13-3 所示，可以看到一些特点。

- 从精确度（Accuracy）等指标上看，随着决策树棵数的增加，指标在持续增长，但增速也逐渐放缓。
- 2 棵决策树和 4 棵决策树的训练时间为 55 秒和 59 秒，这是因为每个并发的 worker 负责训练一棵树。我们设置任务并发度为 4，当有 2 棵决策树时，只有 2 个 worker 在运行，另外 2 个在闲置；而当有 4 棵决策树时，全部 4 个 worker 都在运行。任务所用的时间基本上是单棵决策树的训练时间。
- 当有 4 棵决策树及以上时，所有的并发 worker 都在满负荷运行，任务运行的总体时间与决策树棵数成正比。

注意：细心的读者会发现精确度与 Micro F1 在数值上相等，如表 13-3 所示，再看前面章节中的多分类评估的结果，也同样是相等的。其实，这两个指标是可以在理论上证明相等的，具体内容详见 12.1.3 节。

表 13-3 不同决策树棵数对应的评估指标对比

决策树棵数	Accuracy	Macro F1	Micro F1	Kappa	时间
2	0.8797	0.8774	0.8797	0.8663	55 秒
4	0.931	0.9299	0.931	0.9233	59 秒
8	0.9449	0.944	0.9449	0.9388	1 分 42 秒
16	0.9585	0.958	0.9585	0.9539	2 分 50 秒
32	0.9632	0.9628	0.9632	0.9591	5 分 39 秒
64	0.964	0.9636	0.964	0.96	11 分 46 秒
128	0.9654	0.965	0.9654	0.9615	28 分 12 秒

13.6 K最近邻算法

K 最近邻算法（K-Nearest Neighbor, KNN），是基于某个样本最接近的 K 个邻居的类别情况来预测该样本的分类情况。该算法遵循的准则是“近朱者赤，近墨者黑”，即判断一个人的好坏，通过已知的与其关系最近的 K 个人的情况来判断，如果大多数为好人，则判断此人为好人；反之，则认为其为坏人。

使用 K 最近邻算法分类器 KnnClassifier 的代码如下。

```

new KnnClassifier()
    .setK(3)
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("KnnClassifier - 3 - EUCLIDEAN"))
);

```

运行结果如下，KNN 算法虽然很简单，但对当前问题的分类效果还是很好的。

Metrics:									
Accuracy:0.9719 Macro F1:0.9718 Micro F1:0.9719 Kappa:0.9688									
Pred\Real	9	8	7	...	2	1	0		
----- ----- ----- ----- ----- ----- ----- ----- ----- -----									
9 971	4	12	...	1	0	0			
8	4 924	0	...	2	0	0			
7	8	4 991	...	13	0	1			
...			
2	1	3	4	...	994	2	1		
1	4	0	19	...	9 1133	1			
0	4	7	0	...	10	0 974			

KNN 算法默认是使用欧氏 (EUCLIDEAN) 距离，我们也可以尝试使用其他距离，譬如余弦 (COSINE) 距离。设置 DistanceType=COSINE，其他设置、流程与使用默认欧氏距离的示例相同，如下代码所示。

```

new KnnClassifier()
    .setDistanceType(DistanceType.COSINE)
    .setK(3)
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("KnnClassifier - 3 - COSINE"))
);

```

运行结果如下。

Metrics:									
Pred\Real	9	8	7	...	2	1	0		
9	969	3	15	...	0	0	0		
8	3	940	0	...	6	0	0		
7	5	3	993	...	8	0	1		
...	
2	2	2	5	...	1006	3	0		
1	6	2	11	...	2	1130	1		
0	10	6	3	...	8	0	977		

精确度为 0.974，相比欧氏距离有少许提高。

在 KNN 的试验中，我们设置参数 $K=3$ ，其实也可以使用其他值，譬如尝试一下 $K=7$ ，精确度为 0.97，并没有获得比 $K=3$ 更好的分类效果。

下面我们将 KNN 算法与其他算法进行比较，帮助读者理解机器学习的一些概念，并更深入地了解这些算法的本质。

1. KNN 算法与 K-Means 算法；监督学习与非监督学习

- KNN 算法的训练数据标记了每条记录的类别信息，但 K-Means 算法的训练数据不需要标记类别信息。
- 监督学习（Supervised Learning）是对标记的训练数据进行学习。KNN、朴素贝叶斯、逻辑回归、随机森林等分类方法都属于监督学习。非监督学习（Unsupervised Learning）是对没有标记的训练数据进行学习。典型的例子就是 K-Means 等聚类方法。监督学习与非监督学习的本质区别在于训练数据是否必须被标记。

2. KNN 算法与朴素贝叶斯、逻辑回归和随机森林算法；惰性学习与迫切学习

- KNN 算法没有产生模型，对每个新数据的预测都是计算该新数据与各训练数据的距离，再根据距离最近的 K 个数据的标记分类情况，预测新数据的类别。而朴素贝叶斯、逻辑回归和随机森林算法都有模型训练过程，产生分类模型，对于新数据的预测只需要有分类模型，不再需要原始的训练数据；一般来说，预测时需要的计算量更小，预测速度更快。
- 迫切学习（Eager Learning）先利用训练数据进行训练得到一个模型，使用模型对新数据进行预测。朴素贝叶斯、逻辑回归和随机森林算法都属于迫切学习。惰性学习（Lazy Learning）是指直接使用已有的训练数据对新数据进行预测，预测之前没有模型训练过程，KNN 属于这种方式。



在线学习

在线学习 (Online Learning) 是机器学习的一种模型训练方法，可以根据线上数据的变化实时调整模型，使模型能够反映线上的变化，从而提高线上预测的准确率。

为了更好地理解在线学习的概念，我们先介绍与之相对应的概念——批量训练 (Batch Learning)。先确定一个样本训练集，针对训练集的全体数据进行训练，一般需要使用迭代过程，重复使用数据集，不断调整参数。在线学习不需要事先确定训练数据集，训练数据是在训练过程中逐条产生的，每来一个训练样本，就会根据该样本产生的损失函数值、目标函数值及梯度，对模型进行一次迭代。

FTRL (Follow The Regularized Leader) 算法是一种被广泛应用的在线学习算法，由 Google 于 2013 年提出并发表，详见文章 “Ad Click Prediction: a View from the Trenches”。

这里实现的 FTRL 算法，可以用于如下场景。

(1) 在线训练、在线预测功能。计算流式训练数据，不断更新模型，得到“模型流”，并将其输出到流式 Table。通过流式 Table 实时获取模型，进行在线预测。

(2) 支持初始化模型。实际中通常是对历史数据进行批式训练，既可以用 FTRL 算法，也可以用逻辑回归等线性算法得到初始化模型。

(3) 使用该算法训练批式数据得到一个模型，为了区别于前面的“模型流”，这里称之为固定模型 (Fixed Model)。基于此固定模型，可以对批式数据或流式数据进行预测。

14.1 整体流程

我们先关注 FTRL 在线预测组件及 FTRL 流式预测组件。如图 14-1 所示，它们之间通过模

型数据流连接，即 FtrlTrain 不断产生新的模型，流式地传给 FtrlPredict 组件。每当 FtrlPredict 组件接收到一个完整的模型时，便会替换其旧的模型，切换成新的模型。对于在线学习 FtrlTrain，需要两个输入，一个是初始的模型，避免系统冷启动；另一个是流式的训练数据。FtrlTrain 输出的就是模型数据流。FtrlPredict 组件同样需要初始模型，可以在 FtrlTrain 输出模型前对已来到的数据进行预测。

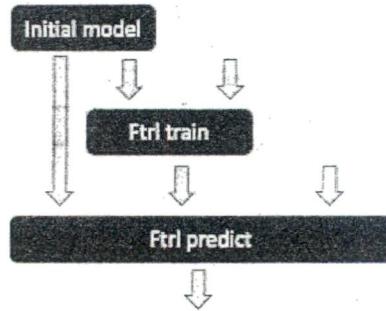


图 14-1

介绍完两个核心组件，我们再看看所需的初始模型、训练数据流和预测数据流是如何准备的。如图 14-2 所示，初始模型是采用传统的离线训练方式，对批式的训练数据进行训练得到的。

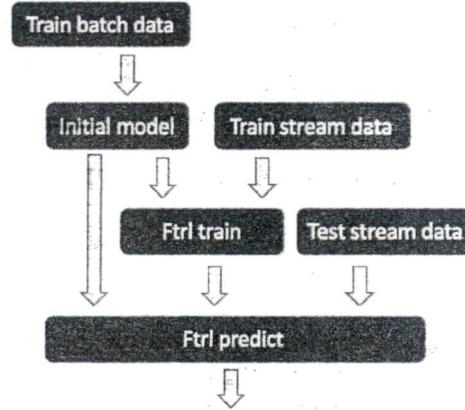


图 14-2

FTRL 算法是线性算法，其输入的数据必须都是数值型的，而原始的数据既有数值型的，也有离散型的，我们需要进行相应的特征工程操作，将原始特征数据变换为向量形式。

如图 14-3 所示，这里我们需要使用特征工程的组件，将批式原始训练数据转化为批式向量训练数据，将流式原始训练数据转化为流式向量训练数据，将流式原始预测数据转化为流式向量预测数据。

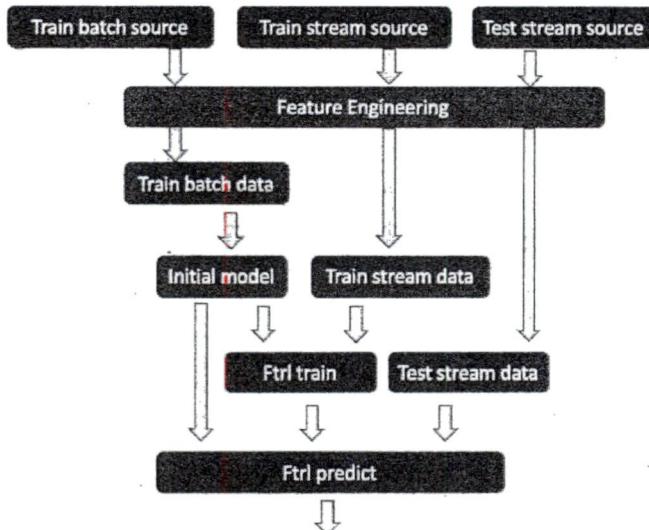


图 14-3

14.2 数据准备

在互联网广告中，点击率（CTR）是衡量广告效果的一个非常重要的指标。因此，点击预测系统在赞助搜索和实时竞价中具有重要的应用价值。

这里使用 Kaggle 比赛的 CTR 数据，原始链接为链接 14-1。Alink 的示例中用到了两个采样数据集（参见链接 14-2、链接 14-3），在国内的访问速度较快。

使用 TextSourceBatchOp 整行读取并输出部分数据，代码如下。

```

new TextSourceBatchOp()
.setFilePath("http://alink-release.oss-cn-beijing.aliyuncs.com/data-files/avazu-small.csv")
.firstN(10)
.print();
  
```

运行结果如图 14-4 所示。

text
1800009438357894273,0,14182100,1005,0,1be01fc,f3a45767,28905ebd,ecad2386,7801e8d9,07d7df22,a99f214a,d2d2926c,44956a24,1,2,15706,320,50,1722,0,35,-1,79
18000169349117063715,0,14182100,1005,0,1be01fc,f3a45767,28905ebd,ecad2386,7801e8d9,07d7df22,a99f214a,96809ac8,711ec220,1,0,15704,320,50,1722,0,35,100084,79
1800037196c2151194066,0,14182100,1005,0,1be01fc,f3a45767,28905ebd,ecad2386,7801e8d9,07d7df22,a99f214a,b3cf8de1,8a4875bd,1,0,15704,320,50,1722,0,35,100084,79
180006467408838376,0,14182100,1005,0,1be01fc,f3a45767,28905ebd,ecad2386,7801e8d9,07d7df22,a99f214a,c02758bf,63324210,1,0,15706,320,50,1722,0,35,100084,79
18000679456417042696,0,14182100,1005,1,fecc448,9166c161,05691928,ecad2386,7801e8d9,07d7df22,a99f214a,9644dbff,779d90c2,1,0,18993,320,50,2161,0,35,-1,157
18000726757801183869,0,14182100,1005,0,d6137915,be1ef334,1028772b,ecad2386,7801e8d9,07d7df22,a99f214a,d5242f26,8a4875bd,1,0,16920,320,50,1899,0,431,100077,117
18000724729968544911,0,14182100,1005,0,81da644b,25d4c1c0,1028772b,ecad2386,7801e8d9,07d7df22,a99f214a,d260c159,be60b107,1,0,2e362,320,50,2333,0,39,-1,157
1800091758742528375,0,14182100,1005,1,c15ic245,7e091613,1028772b,ecad2386,7801e8d9,07d7df22,a99f214a,c6167278,be74e6fc,1,0,20634,320,50,2374,3,39,-1,23
18000949271166023916,2,14182100,1005,1be01fc,f3a45767,28905ebd,ecad2386,7801e8d9,07d7df22,a99f214a,37e8d474,5d8e79b5,1,2,15707,320,50,1722,0,35,-1,79
18001264480619467364,0,14182100,1002,0,84c7ba46,4e18dd8,50e219ed,ecad2386,7801e8d9,07d7df22,c357d0ff,11ac7184,373etdeb,0,0,21e69,320,50,2490,3,167,100191,23

图 14-4

我们看到每条数据包含多个数据项，以逗号分隔。下面是各数据列的定义。

- id: 广告 ID。
- click: 0 表示点击，1 表示不点击。
- hour: 时间格式为 YYMMDDHH，例如 14091123 表示 2014 年 9 月 11 日 23:00。
- C1: 匿名的离散变量。
- banner_pos: 标题位置。
- site_id: 地点 ID。
- site_domain: 地点 Domain。
- site_category: 地点类别。
- app_id: 应用 ID。
- app_domain: 应用 Domain。
- app_category: 应用类别。
- device_id: 设备 ID。
- device_ip: 设备 IP。
- device_model: 设备模型。
- device_type: 设备类型。
- device_conn_type: 设备连接类型。
- C14-C21: 匿名的离散变量。

我们根据各列的定义，组装 SCHEMA_STRING 如下。

```
static final String SCHEMA_STRING
= "id string, click string, dt string, C1 string, banner_pos int, site_id string, site_domain string, "
+ "site_category string, app_id string, app_domain string, app_category string, device_id string, "
+ "device_ip string, device_model string, device_type string, device_conn_type string, C14 int, C15 int, "
+ "C16 int, C17 int, C18 int, C19 int, C20 int, C21 int";
```

接下来，我们就可以通过 CsvSourceBatchOp 读取和显示数据，脚本如下。

```
CsvSourceBatchOp trainBatchData = new CsvSourceBatchOp()
.setFilePath("http://alink-release.oss-cn-beijing.aliyuncs.com/data-files/avazu-small.csv")
```

```
.setSchemaStr(SCHEMA STRING);
```

```
trainBatchData.firstN(10).print();
```

结果如图 14-5 所示。

图 14-5

由于列数较多，我们不易于将数据与列名对应起来。为了更方便地看数据，这里有一个小技巧，输出的文本数据及分隔换行符号正好是 MarkDown 格式，将其复制并粘贴到 MarkDown 编辑器中，即可看到整齐的图片显示，如图 14-6 所示。

图 14-6

14.3 特征工程

14.2 节展示了数据，本节会继续深入了解数据，由数据列的描述信息可知其中含有哪些数值型特征及枚举型特征。具体代码如下所示。

```
static final String[] CATEGORY_COL_NAMES = new String[] {
```

```

"C1", "banner_pos", "site_category", "app_domain",
"app_category", "device_type", "device_conn_type",
"site_id", "site_domain", "device_id", "device_model"};
```

```

static final String[] NUMERICAL_COL_NAMES = new String[] {
    "C14", "C15", "C16", "C17", "C18", "C19", "C20", "C21"};
```

```

static final String LABEL_COL_NAME = "click";

```

click 列标明了是否被点击，是分类问题的标签列。对于数值型特征，各特征的取值范围差异很大，一般需要进行标准化、归一化等操作。枚举类型的特征不能直接应用到 FTRL 模型，需要将枚举值映射到向量值。后面还需要将各列的变换结果合成为一个向量，即后面模型训练的特征向量。

在此示例中，选择对数值类型进行标准化操作，并使用了 FeatureHash 算法组件，在其参数设置中需要指定处理的各列名称，并需要标明哪些是枚举类型，那么没被标明的列就是数值类型的。FeatureHash 操作会将这些特征通过 hash 的方式映射到一个稀疏向量中，可以设置向量的维度，这里设置为 30000。每个数值列都会被 hash 到一个向量项中，该列的数值就会赋给对应的向量项。而每个枚举特征的不同枚举值也会被 hash 到向量项，并被赋值为 1。其实，FeatureHash 同时完成了枚举类型的映射及汇总为特征向量的工作。因为使用了 hash 的方式，所以会存在不同内容被 hash 到同一项的风险，但是该组件使用起来比较简便，因此在示例中使用或者作为实验开始时的组件，快速得到 baseline 指标，FeatureHash 还是很适合的。相关脚本如下所示。

```

static final String VEC_COL_NAME = "vec";
static final int NUM_HASH_FEATURES = 30000;
... ...

// setup feature enginerring pipeline
Pipeline feature_pipeline = new Pipeline()
    .add(
        new StandardScaler()
            .setSelectedCols(NUMERICAL_COL_NAMES)
    )
    .add(
        new FeatureHasher()
            .setSelectedCols(ArrayUtils.addAll(CATEGORY_COL_NAMES, NUMERICAL_COL_NAMES))
            .setCategoricalCols(CATEGORY_COL_NAMES)
            .setOutputCol(VEC_COL_NAME)
            .setNumFeatures(NUM_HASH_FEATURES)
    );

```

我们定义特征工程处理 Pipeline（管道），其中包括 StandardScaler 和 FeatureHasher，对批次训练数据 trainBatchData 执行 fit 方法，并进行训练，得到 PipelineModel（管道模型）。该管

道模型既可以用在批式数据中，也可以应用在流式数据中，生成特征向量。我们先把这个特征工程处理模型保存到本地，文件名为 feature_model.ak，具体代码如下。

```
static final String FEATURE_MODEL_FILE = "feature_model.ak";
...
...
// fit and save feature pipeline model
feature_pipeline
    .fit(trainBatchData)
    .save(DATA_DIR + FEATURE_MODEL_FILE);
BatchOperator.execute();
```

14.4 特征工程处理数据

在 14.3 节中，我们训练并保存了特征工程处理模型。本节需要使用特征工程处理模型，将批式原始训练数据转化为批式向量训练数据，将流式原始训练数据转化为流式向量训练数据，将流式原始预测数据转化为流式向量预测数据，如图 14-7 所示。

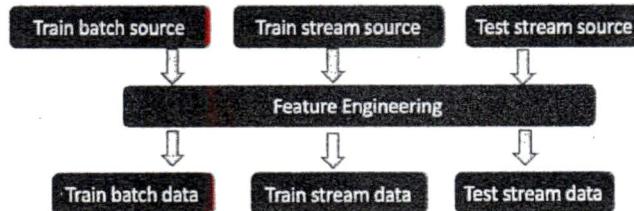


图 14-7

批式原始训练数据如下。

```
CsvSourceBatchOp trainBatchData = new CsvSourceBatchOp()
    .setFilePath("http://alink-release.oss-cn-beijing.aliyuncs.com/data-files/avazu-small.csv")
)
.setSchemaStr(SCHEMA_STRING);
```

我们可以通过定义一个流式数据源，并按 1:1 的比例实时切分数据，从而得到流式原始训练数据、流式原始预测数据。

```
CsvSourceStreamOp data = new CsvSourceStreamOp()
    .setFilePath("http://alink-release.oss-cn-beijing.aliyuncs.com/data-files/avazu-ctr-train-8M.csv")
    .setSchemaStr(SCHEMA_STRING);

SplitStreamOp spliter = new SplitStreamOp().setFraction(0.5).linkFrom(data);
```

利用 PipelineModel.load 方法，可以载入前面保存的特征工程处理模型。

```
PipelineModel feature_pipelineModel = PipelineModel.load(DATA_DIR + FEATURE_MODEL_FILE);
```

Alink 的 PipelineModel 既能预测批式数据，也可以预测流式数据，而且调用方式相同，使用模型实例的 transform 方法即可。

使用如下代码得到批式向量训练数据。

```
feature_pipelineModel.transform(trainBatchData)
```

使用如下代码得到流式向量训练数据。

```
StreamOperator train_stream_data = feature_pipelineModel.transform(splitter);
```

使用如下代码得到流式向量预测数据。

```
StreamOperator test_stream_data = feature_pipelineModel.transform(splitter.getSideOutput(0));
```

进一步地，我们通过批式向量训练数据，可以训练得到一个线性模型作为后面 FTRL 算法的初始模型。首先定义逻辑回归分类器 lr，然后将批式向量训练数据“连接”到此分类器，输出结果便为逻辑回归模型，并将此模型保存到数据文件中，如下脚本所示。

```
// train initial batch model
LogisticRegressionTrainBatchOp lr = new LogisticRegressionTrainBatchOp()
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setWithIntercept(true)
    .setMaxIter(10);

feature_pipelineModel
    .transform(trainBatchData)
    .link(lr)
    .link(
        new AkSinkBatchOp().setFilePath(DATA_DIR + INIT_MODEL_FILE)
    );
BatchOperator.execute();
```

14.5 在线训练

基于前面几节的准备工作，我们已经具备了初始模型、流式向量训练数据、流式向量预测数据。接下来，我们会进入本章的关键内容，演示如何接入 FTRL 在线训练模块及对应的在线预测模块，如图 14-8 所示。

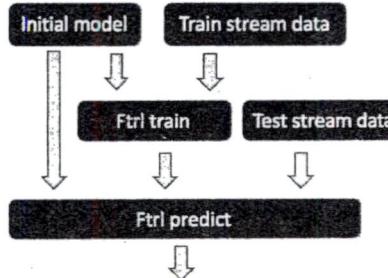


图 14-8

FTRL 在线模型训练的代码如下，在 FtrlTrainStreamOp 的构造函数中输入初始模型 initModel，设置各种参数，并“连接”流式向量训练数据。

```
// ftrl train
FtrlTrainStreamOp model = new FtrlTrainStreamOp(initModel)
.setVectorCol(VEC_COL_NAME)
.setLabelCol(LABEL_COL_NAME)
.setWithIntercept(true)
.setAlpha(0.1)
.setBeta(0.1)
.setL1(0.01)
.setL2(0.01)
.setTimeInterval(10)
.setVectorSize(NUM_HASH_FEATURES)
.linkFrom(train_stream_data);
```

FTRL 在线预测的代码如下，需要“连接” FTRL 在线模型训练输出的模型流，以及流式向量预测数据。

```
FtrlPredictStreamOp predResult = new FtrlPredictStreamOp(initModel)
.setVectorCol(VEC_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.setReservedCols(new String[] {LABEL_COL_NAME})
.setPredictionDetailCol(PRED_DETAIL_COL_NAME)
.linkFrom(model, test_stream_data);
```

我们可以按照如下方式设置输出流式结果，由于数据较多，输出前先对流式数据进行采样，并在每行预测数据前用 SQL 语句增加标识列，显示“Pred Sample”。注意，对于流式的任务，print 方法不能触发流式任务的执行，必须调用 StreamOperator.execute 方法才能开始执行。

```
predResult
.sample(0.0001)
.select("'Pred Sample' AS out_type, *")
.print();
```

在执行的过程中，会先运行批式的初始模型训练，待批式任务执行结束，再启动流式任务。

最后，我们再将预测结果流 predResult 接入流式二分类评估组件 EvalBinaryClassStreamOp，并设置相应的参数。由于每次评估结果是 Json 格式的，为了便于显示，还可以在后面上 Json 内容提取组件 JsonValueStreamOp，并增加标识列，显示相应行输出的内容为评估指标 Eval Metric，代码如下。

```

predResult
    .link(
        new EvalBinaryClassStreamOp()
            .setPositiveLabelValueString("1")
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
            .setTimeInterval(10)
    )
    .link(
        new JsonValueStreamOp()
            .setSelectedCol("Data")
            .setReservedCols(new String[] {"Statistics"})
            .setOutputCols(new String[] {"Accuracy", "AUC", "ConfusionMatrix"})
            .setJsonPath(new String[] {"$.Accuracy", "$.AUC", "$.ConfusionMatrix"})
    )
    .select("'Eval Metric' AS out_type, *")
    .print();

```

注意：流式的组件“连接”完成后，需要调用流式任务执行命令，即 StreamOperator.execute()，开始执行，显示结果如下。

```

out_type|click|pred|pred_info
-----|---|---|-----
out_type|Statistics|Accuracy|AUC|ConfusionMatrix
-----|-----|-----|---|-
Pred Sample[0|0|{"0": "0.9220358860557992", "1": "0.07796411394420077"}]
Pred Sample[1|0|{"0": "0.6017776815075736", "1": "0.3982223184924264"}]
Pred Sample[0|0|{"0": "0.9309052094416868", "1": "0.06909479055831325"}]
Pred Sample[0|0|{"0": "0.9200218813350143", "1": "0.07997811866498572"}]
Pred Sample[0|0|{"0": "0.8557440536248255", "1": "0.14425594637517447"}]
Pred Sample[0|0|{"0": "0.9437830739831977", "1": "0.05621692601680228"}]
Pred Sample[0|0|{"0": "0.7786343432104281", "1": "0.22136565678957187"}]
Pred Sample[0|0|{"0": "0.9682519573861983", "1": "0.03174804261380171"}]
Pred Sample[0|0|{"0": "0.8974879967663411", "1": "0.10251200323365894"}]
Pred Sample[0|0|{"0": "0.8854488604041583", "1": "0.11455113959584173"}]
Eval Metric>window|0.8376583064315868|0.6143809440223419|[ [53, 59], [2556, 13440] ]
Eval Metric/all|0.8376583064315868|0.6143809440223419|[ [53, 59], [2556, 13440] ]
Pred Sample[0|0|{"0": "0.8335324643013561", "1": "0.16646753569864392"}]

```

```

Pred Sample|0|0| {"0": "0.8423454385153902", "1": "0.15765456148460977"}
Pred Sample|0|0| {"0": "0.8452003664043619", "1": "0.1547996335956381"}
Pred Sample|0|0| {"0": "0.9368763856950871", "1": "0.06312361430491287"}
Pred Sample|0|0| {"0": "0.7963950068001105", "1": "0.20360499319988945"}
Eval Metric|window|0.842108940173267|0.6423637093476301|[164, 180], [6600, 35997]
Eval Metric|all|0.8408948500397975|0.6343512339990862|[217, 239], [9156, 49437]
Pred Sample|0|0| {"0": "0.8853896240654119", "1": "0.11461037593458812"}
Pred Sample|0|0| {"0": "0.9144744898328552", "1": "0.08552551016714482"}
Pred Sample|1|0| {"0": "0.8063021776871495", "1": "0.1936978223128505"}
Eval Metric|window|0.8424430394846714|0.7265268800218625|[183, 147], [6457, 35128]
Eval Metric|all|0.8415375777504853|0.67178153459723|[400, 386], [15613, 84565]
...

```

其中输出了两种流式数据，一种是预测数据，各列的名称如下。

out_type	click	pred	pred_info
----------	-------	------	-----------

第 1 列为标识信息，第 2 列是原始的 click 信息，第 3 列为预测结果，第 4 列为预测的详细信息。对应的预测结果形式如下。

```
Pred Sample|0|0| {"0": "0.9220358860557992", "1": "0.07796411394420077"}
```

另一种是评估指标，各列的名称如下。

out_type	Statistics	Accuracy	AUC	ConfusionMatrix
----------	------------	----------	-----	-----------------

其中，Statistics 列有两个值 all 和 window，all 表示从开始运行到现在的所有预测数据的评估结果；window 表示时间窗口（当前设置为 10 秒）的所有预测数据的评估结果。

从输出的信息来看，无论是整体指标，还是窗口指标，Accuracy 和 AUC 都在提升，说明 FTRL 算法是有效的。

14.6 模型过滤

在上节的示例中，在线学习训练器每隔 10 秒就产生一个模型，然后直接推送给流式预测组件。在实际应用中，10 秒内的数据可能有很强的偏向性，从而影响模型的整体预测效果，模型未经验证就直接上线，难免会让人有些担心。本节将介绍模型的过滤机制，对新产生的模型进行验证，只有当评估指标达到了阈值，才将新模型参数传给流式预测组件。

使用 FtrlModelFilterStreamOp 组件 model_filter，设置模型评估精确度的阈值 AccuracyThreshold=

0.83，AUC 的阈值 AucThreshold=0.71，并设置一些模型评估需要用到的参数。将 model_filter 接入在线学习组件输出的模型流，并接入用来验证模型的数据流。这里设置了一个流式输出，将输出过滤后的模型，使用前面的技巧，在每行模型数据前用 SQL 语句增加标识列，显示“Model”。对于流式预测组件 FtrlPredictStreamOp，接入模型过滤组件 model_filter 输出的模型流，并接入所要预测的数据流 test_stream_data。具体代码如下。

```
// ftrl train
FtrlTrainStreamOp model = new FtrlTrainStreamOp(initModel)
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setWithIntercept(true)
    .setAlpha(0.1)
    .setBeta(0.1)
    .setL1(0.01)
    .setL2(0.01)
    .setTimeInterval(10)
    .setVectorSize(NUM_HASH_FEATURES)
    .linkFrom(train_stream_data);

// model filter
FtrlModelFilterStreamOp model_filter = new FtrlModelFilterStreamOp()
    .setPositiveLabelValueString("1")
    .setVectorCol(VEC_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setAccuracyThreshold(0.83)
    .setAucThreshold(0.71)
    .linkFrom(model, train_stream_data);

model_filter
    .select("'Model' AS out_type, *")
    .print();

// ftrl predict
FtrlPredictStreamOp predResult = new FtrlPredictStreamOp(initModel)
    .setVectorCol(VEC_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setReservedCols(new String[] {LABEL_COL_NAME})
    .setPredictionDetailCol(PRED_DETAIL_COL_NAME)
    .linkFrom(model_filter, test_stream_data);
```

输出计算结果如图 14-9 所示，我们看到在前两轮评估前，模型都没有更新，这是因为初始模型的评估指标较低，需要使用更多的训练样本才能达到指定的指标阈值。FTRL 组件虽然每 10 秒就产生一个模型，但是由于评估指标没有到阈值，所以都被过滤掉了。

```

out_type|bid|ntab|model_id|model_info|label_value
-----|---|---|---|---|-----
out_type|click|pred|pred_info
-----|---|---|---|-----
out_type|Statistics|Accuracy|AUC|ConfusionMatrix
-----|---|---|---|-----
Pred Sample[0|0|{"0":"0.9377529917730091","1":"0.06224700822699092"}]
Eval Metric|window|0.8335293287548803|0.612220080597303|[61,88],[3451,17659]]
Eval Metric|all|0.8335293287548803|0.612220080597303|[61,88],[3451,17659]]
Pred Sample[0|0|{"0":"0.843055646316327","1":"0.156944353683673"}]
Pred Sample[0|0|{"0":"0.9129594617918085","1":"0.08704053820819146"}]
Pred Sample[0|0|{"0":"0.9532587643017029","1":"0.046741235698297134"}]
Pred Sample[0|0|{"0":"0.9357957904044711","1":"0.06420420959552886"}]
Pred Sample[0|0|{"0":"0.8641536244526069","1":"0.1358463755473931"}]
Pred Sample[0|0|{"0":"0.7865946845445935","1":"0.21340531545540653"}]
Pred Sample[0|0|{"0":"0.9234134648219126","1":"0.07658653517808744"}]
Eval Metric|window|0.84064184729717774|0.6198697322921425|[142,157],[7058,37855]]
Eval Metric|all|0.8328151614989996|0.6173660384648244|[203,245],[10509,55514]]
Pred Sample[0|0|{"0":"0.9186098221320143","1":"0.08139017786798575"}]
Pred Sample[0|0|{"0":"0.8029715291034955","1":"0.19702847099650455"}]
Model[0|20|0|{"hasInterceptItem":true,"vectorColName":"vec","modelName":"\"Logistic Regression\"","labelCol":n
Model[0|20|1048576|{"featureColNames":null,"featureColTypes":null,"coefVector":{"data":[-0.6784407860097712,0,0,
Model[0|20|1048577|-1.015492755263354E-4,-3.899382446905679E-5,1.135019436935083E-4,0,0,-6.819390973151679E-5,-0
Model[0|20|1048578|687E-5,-0.04365726078229945,0,0,-3.273834796168124E-5,-0.0545121263974209,-1.983135132297495
Model[0|20|1048579|0,0,-0.00506139587506867,-8.595045479817747E-5,1.149032894371878E-4,0,0,0,0,0,0,0,-2.718098
Model[0|20|1048580|4E-5,0,0,0,-3.274184053093851E-5,-1.5153002429807585E-4,-0.017178455686681304,-3.0386246467
Model[0|20|1048581|25848969E-5,-5.6092900005054916E-5,0,0,1.4860481330858388E-4,-0.033398008429795685,-1.0855589
Model[0|20|1048582|0,-7.755721208571834E-5,0,0,-2.6006550017840366E-5,0,0,-2.7342190154608805E-5,1.461146502914
Model[0|20|1048583|-5,0.08310933503473497,-0.05472890571896318,-3.0712463364636266E-5,0,0,0,0,1.3936238214184944
Model[0|20|1048584|876,-2.7898636138064154E-5,-0.05620678923208143,-0.0552696873467867,-2.273291527457903E-5,0,
Model[0|20|1048585|425927,2.79729607179458E-4,-0.0391522948230068,-8.670768826912563E-5,2.2848705357318133E-4,-
Model[0|20|1048586|63405432166148,-3.176819986462355E-5,-3.4186051793327794E-5,0,0,-2.7191147181643166E-5,-2.779
Model[0|20|1048587|836231E-5,0,0,1.4606971848951493E-4,-6.855781569505958E-4,-2.2572380287123274E-5,0,0,0,0,-2.2
Model[0|20|1048588|2345451943E-6,-0.17615794486620295,-1.8779470874865728E-4,-8.415450395865494E-5,-3.4888453510
Model[0|20|1048589|996693038E-4,-3.0812477279027554E-5,-4.784778241234657E-5,0,0,0.00824580250951806,0,0,-0.0786
Model[0|20|1048590|-0.007418252415562256,0,0,1.3567399015933853E-4,-6.798770308079837E-5,-1.0191936079852558E-4,
Model[0|20|1048591|52494017115,0,0,-1.385972170012299E-4,-1.05059711258772E-4,0,0,0,0,-0.012647716635138372,1.48
Model[0|20|1048592|E-5,0,0,1.605422787428073E-4,-0.121684396839641,0,0,0,1.2745802766813677E-4,-5.199610201294
Model[0|20|2251799812636672|null|
Model[0|20|2251799812636673|null|
Pred Sample[0|0|{"0":"0.9835687374543344","1":"0.016431262545665626"}]
```

图 14-9

15

回归的由来

前面的章节都在建立分类或聚类的机器学习模型，模型预测输出的结果都属于类别，所有的类别是平等的，编号 1 的类别并不比编号 100 的类别差。本章同样会建立机器学习模型，但模型预测输出的是数值，即回归模型，数值能够反映不同预测样本在数量多少、品质高低、程度深浅上的差异。

“回归”的概念是由高尔顿 (Francis Galton) 在研究人类遗传问题时提出来的，随着《遗传的身高向平均数方向的回归》¹一文的发表而得到普及。本节将使用当年的数据，利用Alink的分析工具，再现分析过程，通过数据理解“回归”的含义。

在 Kaggle 上有原始数据，为 1078 条父亲与儿子的身高记录，参见链接 15-2。下载解压后为文本格式数据，如图 15-1 所示，第一行为数据列名称，第一列为 Father (父亲身高，单位：英寸)，第二列为 Son (儿子身高，单位：英寸)，中间以制表符分隔。

Father	Son	Pearson's r
65.8	59.8	
63.3	63.2	
65.0	63.3	
65.8	62.8	
61.1	64.3	
63.8	64.2	
65.4	64.1	
64.7	64.0	
66.1	64.6	
67.0	64.0	
59.0	65.2	
62.9	65.4	
63.7	65.7	

图 15-1

¹ 该论文的英文名称为 “Regression towards mediocrity in hereditary stature”，参见链接 15-1。

15.1 平均数

我们先使用 CsvSourceBatchOp 组件读取原始数据，具体代码如下，设列名分别为 father 和 son，都是 DOUBLE 类型；字段分隔符为制表符，即 FieldDelimiter="\t"；原始数据的第一行用于记录数据列名称，在读取数据时应该略过，需要设置参数 IgnoreFirstLine=true。

```
CsvSourceBatchOp source = new CsvSourceBatchOp()
.setFilePath(DATA_DIR + ORIGIN_FILE)
.setSchemaStr("father double, son double")
.setFieldDelimiter("\t")
.setIgnoreFirstLine(true);

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

数据输出结果如下。

father	son
65.0000	59.8000
63.3000	63.2000
65.0000	63.3000
65.8000	62.8000
61.1000	64.3000

为了便于整体观察，我们将这 1000 多个数据以散点图的方式显示出来，如图 15-2 所示。

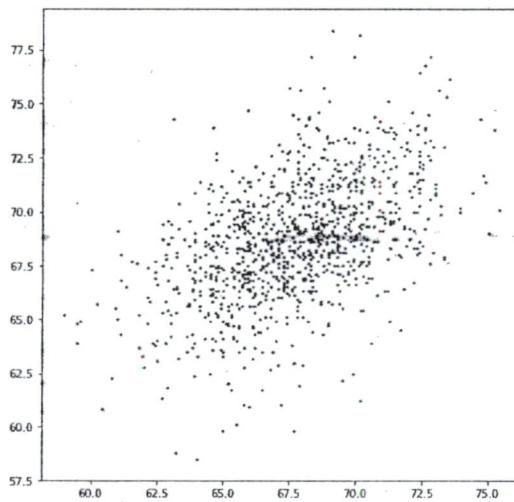


图 15-2

再使用 `lazyPrintStatistics` 方法，对数据进行统计操作，具体代码如下。

```
source.lazyPrintStatistics();
```

计算得到的统计结果如下。

colName	count	missing	sum	mean	variance	min	max
father	1078	0	72966.4	67.6868	7.5396	59	75.4
son	1078	0	74041.6	68.6842	7.9309	58.5	78.4

可知，父亲的平均身高是 67.6868 英寸，儿子的平均身高为 68.6842 英寸，之间相差了 0.9974 英寸，约为 1 英寸。

15.2 向平均数方向的回归

基于 15.1 节的统计结果，儿子的平均身高比父亲的平均身高多 0.9974 英寸（约 1 英寸），我们可以自然地猜测，身高 72 英寸的父亲平均会有身高 73 英寸的儿子，身高 65 英寸的父亲平均会有身高 66 英寸的儿子，即可以用直线 $y = x + 1$ 来近似预测儿子的身高。

我们在原散点图的基础上绘制直线 $y = x + 1$ ，如图 15-3 所示。

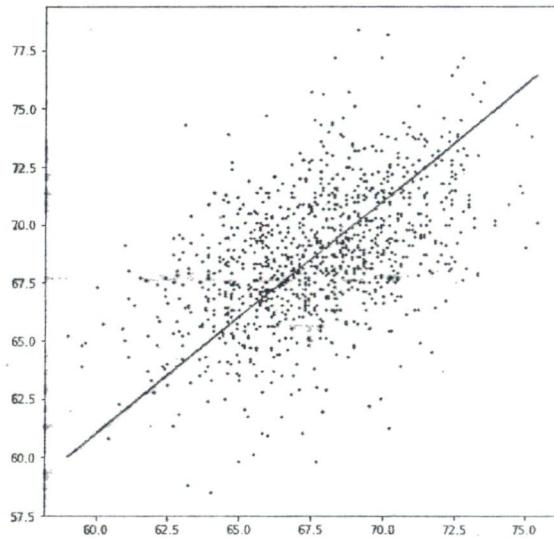


图 15-3

可以发现，在中间区域，数据点平均分布在直线两侧；在左侧区域，似乎直线上方的数据点多一些；在右侧区域，似乎直线下方的数据点多一些。

下面我们选取两个有代表性的父亲身高区域，通过统计进行定量分析。具体代码如下，分别是父亲身高为 72 英寸左右时，儿子身高的统计情况；父亲身高为 65 英寸左右时，儿子身高的统计情况。

```
source.filter("father>=71.5 AND father<72.5").lazyPrintStatistics("father 72");
source.filter("father>=64.5 AND father<65.5").lazyPrintStatistics("father 65");
```

运行结果如下。

father 72	colName	count	missing	sum	mean	variance	min	max
	father	52	0	3745.1	72.0212	0.1037	71.5	72.4
	son	52	0	3675.9	70.6904	5.3542	64.5	76.4

father 65	colName	count	missing	sum	mean	variance	min	max
	father	104	0	6750.5	64.9087	0.087	64.5	65.4
	son	104	0	6986.1	67.174	6.3402	59.8	73.9

可以看出，身高为 65 英寸左右的父亲们的平均身高为 64.9087 英寸，其儿子们的平均身高为 67.174 英寸，提高了 2.2653 英寸；身高为 72 英寸左右的父亲们的平均身高为 72.0212 英寸，其儿子们的平均身高为 70.6904 英寸，降低了 1.3308 英寸。这与我们前面猜测的直线规律 $y = x + 1$ 相差很大。

从这些数据中可以看出，对于矮个子的父亲，其儿子们的平均身高会比父辈高一些；对于高个子的父亲，其儿子们的平均身高会比父辈矮一些，即儿子们的身高会向平均值“回归”。

15.3 线性回归

下面使用线性回归模型对此数据进行建模预测，并与前面的结果进行对比，具体代码如下。

```
LinearRegTrainBatchOp linear_model =
  new LinearRegTrainBatchOp()
    .setFeatureCols("father")
```

```
.setLabelCol("son")
.linkFrom(source);

linear_model.lazyPrintTrainInfo();
linear_model.lazyPrintModelInfo();
```

输出模型信息如下。

```
----- model meta info -----
{hasInterception: true, model name: Linear Regression, num feature: 1}
----- model weight info -----
|intercept|    father|
|-----|-----|
| 33.8928|0.51400662|
```

即，线性回归的表达式为

$$y = 33.8928 + 0.51400662x$$

使用预测组件 `LinearRegPredictBatchOp`，从 `linear_model` 获取线性回归模型，对原始数据进行线性回归预测，相关代码如下。

```
LinearRegPredictBatchOp linear_reg =
new LinearRegPredictBatchOp()
.setPredictionCol("linear_reg")
.linkFrom(linear_model, source);

linear_reg.lazyPrint(5);
```

运行结果如下。

```
father|son|linear_reg
-----|---|
65.0000|59.8000|67.3033
63.3000|63.2000|66.4295
65.0000|63.3000|67.3033
65.8000|62.8000|67.7145
61.1000|64.3000|65.2987
```

最后，将此直线与原始数据及猜测直线放在一张图内，如图 15-4 所示。

两条直线的交点为父亲和儿子身高数据的平均值，即(67.6868, 68.6842)。在其左侧，线性回归值要高于直线 $y=x+1$ ；而在其右侧，线性回归值要低于直线 $y=x+1$ 。

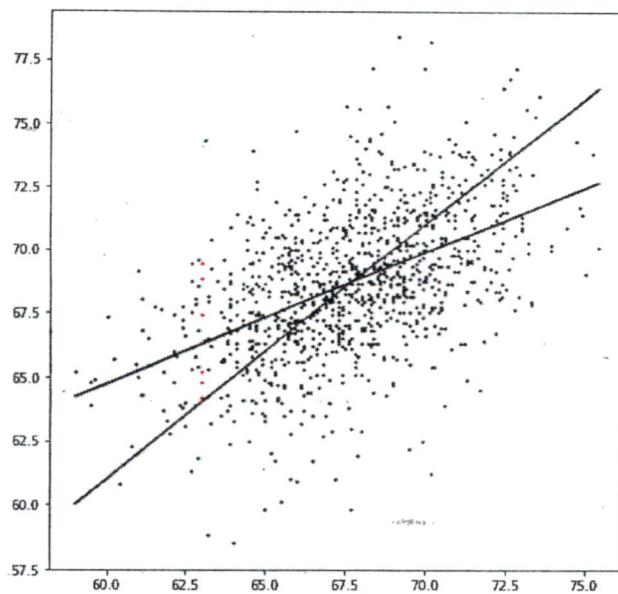


图 15-4

16

常用回归算法

本章以预测葡萄酒品质值为例，演示常用的回归算法。与二分类、多分类模型相似，回归模型也有其评估指标，我们将在 16.1 节中进行介绍。

16.1 回归模型的评估指标

设回归函数为 $f(x)$ ，回归测试样本共 n 条，第 i 条样本的特征为 $x^{(i)}$ ，回归标签值为 $y^{(i)}$ ，并设 \bar{y} 为回归标签值的平均值，即

$$\bar{y} = \frac{1}{n} \sum_{i=1}^n y^{(i)}$$

总平方和（Sum of Squared for Total，SST）为

$$SST = \sum_{i=1}^n (y^{(i)} - \bar{y})^2$$

回归平方和（Sum of Squares for Regression，SSR）为

$$SSR = \sum_{i=1}^n (f(x^{(i)}) - \bar{y})^2$$

误差平方和（Sum of Squares for Error，SSE）为

$$\text{SSE} = \sum_{i=1}^n (f(x^{(i)}) - y^{(i)})^2$$

R^2 判定系数 (Coefficient of Determination) 为

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$$

R 多重相关系数 (Multiple Correlation Coefficient) 为

$$R = \sqrt{R^2}$$

均方误差 (Mean Squared Error, MSE) 为

$$\text{MSE} = \frac{1}{n} \text{SSE} = \frac{1}{n} \sum_{i=1}^n (f(x^{(i)}) - y^{(i)})^2$$

均方根误差 (Root Mean Squared Error, RMSE) 为

$$\text{RMSE} = \sqrt{\text{MSE}}$$

绝对误差 (Sum of Absolute Error/Difference, SAE/SAD) 为

$$\text{SAE} = \sum_{i=1}^n |f(x^{(i)}) - y^{(i)}|$$

平均绝对误差 (Mean Absolute Error/Difference, MAE/MAD) 为

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |f(x^{(i)}) - y^{(i)}|$$

平均绝对百分误差 (Mean Absolute Percentage Error, MAPE) 为

$$\text{MAPE} = \frac{100}{n} \sum_{i=1}^n \left| \frac{f(x^{(i)}) - y^{(i)}}{y^{(i)}} \right|$$

解释方差 (Explained Variance) 为

$$\text{ExplainedVariance} = \frac{\text{SSR}}{n}$$

16.2 数据探索

葡萄酒的品质可以用数值表示，数值越大表示品质越好，而葡萄酒本身的理化指标对其品质是有影响的。下面通过建立回归模型对葡萄酒的品质进行预测，可从链接 16-1 下载数据。葡萄牙的绿酒（Vinho Verde），其酒精度中等，口感清淡鲜酸，特别适合夏季饮用。需要注意的是，绿酒中的“绿”指的并非是成熟或陈年的葡萄酒，绿酒可以是白色的，也可以是红色的。原始数据中分别对两种颜色的绿酒提供了不同的数据集，我们只选择了其中的一个（白色的），数据文件的名称为 winequality-white.csv。

该数据集有 11 个特征字段，都是经过理化测试得到的，具体名称及含义如表 16-1 所示。

表 16-1 特征字段说明

编号	特征字段名称	中文含义	编号	特征字段名称	中文含义
1	fixed acidity	固定酸度	7	total sulfur dioxide	总二氧化硫
2	volatile acidity	挥发性酸度	8	density	密度
3	citric acid	柠檬酸	9	pH	pH 值
4	residual sugar	残糖	10	sulphates	硫酸盐
5	Chlorides	氯化物	11	alcohol	酒精
6	free sulfur dioxide	自由二氧化硫			

该数据集的目标变量为葡萄酒的品质，目标变量名称为 quality，分值在 0 和 10 之间，是基于感官的数据。

将数据文件 winequality-white.csv 下载到本地，使用文本编辑器打开，如图 16-1 所示。

```
"fixed acidity";"volatile acidity";"citric acid";"residual sugar";"chlorides";"free sulfur dioxide";
7;0.27;0.36;20.7;0.045;45;170;1.001;3;0.45;8.8;6
6.3;0.3;0.34;1.6;0.049;14;132;0.994;3.3;0.49;9.5;6
8.1;0.28;0.4;6.9;0.05;30;97;0.9951;3.26;0.44;10.1;6
7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4;9.9;6
7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4;9.9;6
8.1;0.28;0.4;6.9;0.05;30;97;0.9951;3.26;0.44;10.1;6
6.2;0.32;0.16;7;0.045;30;136;0.9949;3.18;0.47;9.6;6
7;0.27;0.36;20.7;0.045;45;170;1.001;3;0.45;8.8;6
6.3;0.3;0.34;1.6;0.049;14;132;0.994;3.3;0.49;9.5;6
8.1;0.22;0.43;1.5;0.044;28;129;0.9938;3.22;0.45;11;6
8.1;0.27;0.41;1.45;0.033;11;63;0.9908;2.99;0.56;12;5
8.6;0.23;0.4;4.2;0.035;17;189;0.9947;3.14;0.53;9.7;5
7;0.28;0.37;20.7;0.045;45;170;1.001;3;0.45;8.8;6
```

图 16-1

第 1 行为各数据列的名称，从第 2 行开始是具体的数据，每行为一条记录，各数值间使用分号分隔。如下代码所示，定义各列的名称，由于列名中不能有空格，所以我们使用“驼峰”

格式将多个单词连在一起；所有数据都是数值类型的，这里我们定义为双精度浮点类型。根据数据的特点，在使用 CsvSourceBatchOp 组件时，需要设置字段间的分隔符为分号，并忽略第 1 行内容，即设置参数 IgnoreFirstLine=true，最后输出 5 条数据。

```
private static final String[] COL_NAMES = new String[] {
    "fixedAcidity", "volatileAcidity", "citricAcid", "residualSugar", "chlorides",
    "freeSulfurDioxide", "totalSulfurDioxide", "density", "pH", "sulphates",
    "alcohol", "quality"
};

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

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

source.lazyPrint(5);
```

输出数据如表 16-2 所示，quality 列为回归标签值列。

表 16-2 葡萄酒品质数据

fixedAcidity	volatileAcidity	citricAcid	residualSugar	chlorides	freeSulfurDioxide	totalSulfurDioxide	density	pH	sulphates	alcohol	quality
5.6000	0.3500	0.4000	6.3000	0.0220	23.0000	174.0000	0.9922	3.5400	0.5000	11.6000	7.0000
8.8000	0.2400	0.2300	10.3000	0.0320	12.0000	97.0000	0.9957	3.1300	0.4000	10.7000	6.0000
6.0000	0.2900	0.2100	15.5500	0.0430	20.0000	142.0000	0.9966	3.1100	0.5400	10.1000	6.0000
6.1000	0.2700	0.3100	1.5000	0.0350	17.0000	83.0000	0.9908	3.3200	0.4400	11.1000	7.0000
7.4000	0.5600	0.0900	1.5000	0.0710	19.0000	117.0000	0.9950	3.2200	0.5300	9.8000	5.0000

了解变量 quality 与 11 个特征变量间的关系，需要计算相关系数矩阵，具体代码如下。

```
source.link(new CorrelationBatchOp().lazyPrintCorrelation());
```

输出结果如表 16-3 所示。

表 16-3 各列的相关系数

colName	fixedAcidity	volatileAcidity	citricAcid	residualSugar	chlorides	freeSulfurDioxide	totalSulfurDioxide	density	pH	sulphates	alcohol	quality
fixedAcidity	1.0000	-0.0227	0.2892	0.0890	0.0231	-0.0494	0.0911	0.2653	-0.4259	-0.0171	-0.1209	-0.1137
volatileAcidity	-0.0227	1.0000	-0.1495	0.0643	0.0705	-0.0970	0.0893	0.0271	-0.0319	-0.0357	0.0677	-0.1947
citricAcid	0.2892	-0.1495	1.0000	0.0942	0.1144	0.0941	0.1211	0.1495	-0.1637	0.0623	-0.0757	-0.0092
residualSugar	0.0890	0.0643	0.0942	1.0000	0.0887	0.2991	0.4014	0.8390	-0.1941	-0.0267	-0.4506	-0.0976
chlorides	0.0231	0.0705	0.1144	0.0887	1.0000	0.1014	0.1989	0.2572	-0.0904	0.0168	-0.3602	-0.2099
freeSulfurDioxide	-0.0494	-0.0970	0.0941	0.2991	0.1014	1.0000	0.6155	0.2942	-0.0006	0.0592	-0.2501	0.0082
totalSulfurDioxide	0.0911	0.0893	0.1211	0.4014	0.1989	0.6155	1.0000	0.5299	0.0023	0.1346	-0.4489	-0.1747
density	0.2653	0.0271	0.1495	0.8390	0.2572	0.2942	0.5299	1.0000	-0.0936	0.0745	-0.7801	-0.3071
pH	-0.4259	-0.0319	-0.1637	-0.1941	-0.0904	-0.0006	0.0023	-0.0936	1.0000	0.1560	0.1214	0.0994
sulphates	-0.0171	-0.0357	0.0623	-0.0267	0.0168	0.0592	0.1346	0.0745	0.1560	1.0000	-0.0174	0.0537
alcohol	-0.1209	0.0677	-0.0757	-0.4506	-0.3602	-0.2501	-0.4489	-0.7801	0.1214	-0.0174	1.0000	0.4356
quality	-0.1137	-0.1947	-0.0092	-0.0976	-0.2099	0.0082	-0.1747	-0.3071	0.0994	0.0537	0.4356	1.0000

为了便于查看，我们借助可视化工具，各字段间的相关系数情况如图 16-2 所示，除了有具体的数字，还用颜色进行了标识，黑色为+1，白色为-1。颜色越深，说明正相关性越强；颜色越浅，说明负相关性越强。

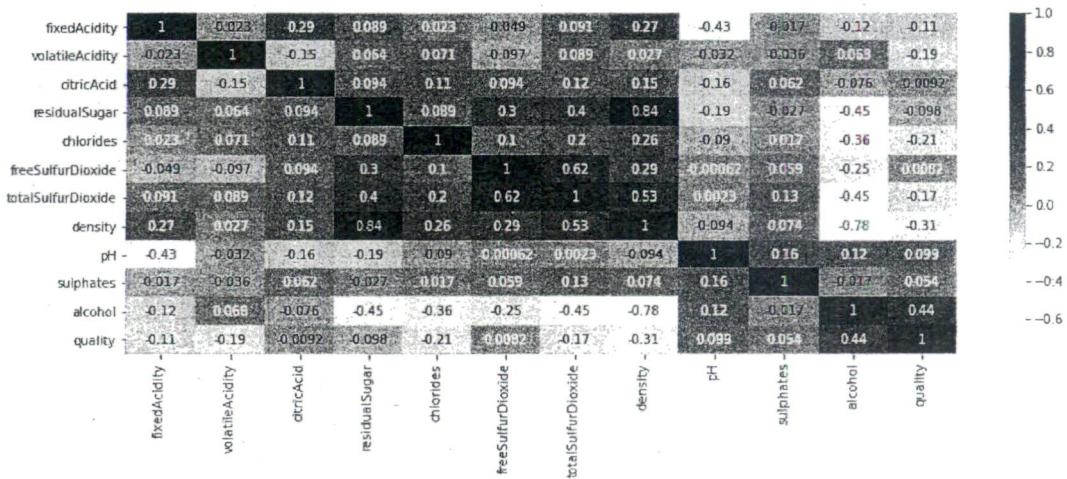


图 16-2

从图 16-2 中可以看出如下内容。

- 特征 alcohol (酒精) 与 quality 的相关系数为 0.44, 正相关性较强; density (密度) 与 quality 的相关系数值为 -0.31, 负相关性较强。
- 特征 citricAcid (柠檬酸) 与 quality 的相关系数为 -0.0092, 接近 0, 是线性无关; 特征 freeSulfurDioxide (自由二氧化硫) 与 quality 的相关系数为 -0.0082, 接近 0, 是线性无关。
- 特征 density (密度) 与 residualSugar (残糖) 的正相关性很强, 相关系数为 0.84; density (密度) 与 alcohol (酒精) 的负相关性很强, 相关系数为 -0.78。

本数据集的标签列 (quality 列) 都是整数值, 可以通过分组计数和排序操作计算其直方图分布, 具体代码如下。

```
source
    .groupBy(LABEL_COL_NAME, LABEL_COL_NAME + ", COUNT(*) AS cnt")
    .orderBy(LABEL_COL_NAME, 100)
    .lazyPrint(-1);
```

运行结果如下。

quality cnt
3.0000 20
4.0000 163
5.0000 1457
6.0000 2198
7.0000 880
8.0000 175
9.0000 5

可以看出 6 的个数最多, 其次是 5 和 7, 4 和 8 的分布更少, 都在 200 以下, 比 6 的个数少一个数量级, 而 3 和 9 的比例更小。

最后, 将原始数据按 8:2 的比例划分为训练集和预测集, 保存为 AK 格式文件。本章后面尝试的各种算法都会基于此训练集和测试集, 以便于比较效果。

16.3 线性回归

首先尝试使用我们比较熟悉的线性回归方法, 相关代码如下。

```
new LinearRegression()
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME)
```

```

.setPredictionCol(PREDICTION_COL_NAME)
.enableLazyPrintTrainInfo()
.enableLazyPrintModelInfo()
.fit(train_data)
.transform(test_data)
.link(
    new EvalRegressionBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("LinearRegression")
);

```

使用线性回归组件，设置特征、标签及预测结果列，其他参数使用默认值；选择使用 Lazy 方式输出训练信息和模型信息；组件 EvalRegressionBatchOp 用于评估回归模型。

训练信息如下。

```

----- train meta info -----
{model name: Linear Regression, num feature: 11}
----- train importance info -----
| colName|importnaceValue| colName| weightValue|
|-----|-----|-----|-----|
| density| 0.44120155| pH| 0.64837239|
| residualSugar| 0.40567559| sulphates| 0.64099725|
| alcohol| 0.24270038| alcohol| 0.19730515|
| ... ...| ... ...| ... ...| ... ...|
| totalSulfurDioxide| 0.01594283| chlorides| -0.32426272|
| chlorides| 0.00728598| volatileAcidity| -1.88292677|
| citricAcid| 0.00313107| density|-147.10297328|
----- train convergence info -----
step:0 loss:14.36145407 gradNorm:5.90856615 learnRate:0.40000000
step:1 loss:5.30536574 gradNorm:5.30978617 learnRate:1.60000000
step:2 loss:1.81822334 gradNorm:3.30213486 learnRate:1.60000000
...
step:12 loss:0.28568962 gradNorm:0.00474675 learnRate:4.00000000
step:13 loss:0.28568641 gradNorm:0.00216498 learnRate:4.00000000
step:14 loss:0.28568635 gradNorm:0.00032419 learnRate:4.00000000

```

最重要的 3 个特征为 density、residualSugar 和 alcohol。整个训练过程在第 14 次迭代时达到收敛条件退出。

下面是模型中各个特征的权重值。

```

----- model meta info -----
{hasInterception: true, model name: Linear Regression, num feature: 11}
----- model weight info -----
| colName[0,9]| intercept|fixedAcidity|volatileAcidity| citricAcid|residualSugar| chlorides|freeSulfurDioxide|totalSulfurDioxide| density| pH|
| weight[0,9]| 147.2227| 0.05561785| -1.88292677| -0.02573757| 0.08013539| -0.32426272| 0.00370713| -0.00037885| -147.10297328| 0.64837239|
| colName[10,11]| sulphates| alcohol| | | | | | |
| weight[10,11]| 0.64099725| 0.19730515| | | | | |

```

回归评估结果如下，我们会把其作为一个基准与后面的方法进行比较。

Metrics:

MSE: 0.5309 RMSE: 0.7286 MAE: 0.5748 MAPE: 10.0995 R2: 0.2655

下面我们希望通过 LASSO 算法减少线性模型中非零特征的数量，使用组件，设置参数 Lambda=0.05，其他设置与线性回归一样。

```

new LassoRegression()
.setLambda(0.05)
.setFeatureCols(FEATURE_COL_NAMES)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.enableLazyPrintTrainInfo()
.enableLazyPrintModelInfo("< LASSO model >")
.fit(train_data)
.transform(test_data)
.link(
    new EvalRegressionBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("LassoRegression")
);

```

输出训练信息如下。

train meta info			
{model name: LASSO, num feature: 11}			
train importance info			
colName	importnaceValue	colName	weightValue
alcohol	0.30556675	alcohol	0.24841285
volatileAcidity	0.14232785	sulphates	0.03046873
residualSugar	0.05538545	residualSugar	0.01094060
...
citricAcid	0.00000000	chlorides	-0.90947378
totalSulfurDioxide	0.00000000	volatileAcidity	-1.40732785
pH	0.00000000	density	-18.19511910

train convergence info			
step:0	loss:14.39385147	gradNorm:5.85882684	learnRate:0.40000000
step:1	loss:4.61560283	gradNorm:5.25846666	learnRate:1.60000000
step:2	loss:0.50012696	gradNorm:2.88726519	learnRate:1.60000000
...			
step:20	loss:0.30128245	gradNorm:0.06450862	learnRate:0.00000000
step:21	loss:0.30128245	gradNorm:0.06450862	learnRate:0.00000000
step:22	loss:0.30128245	gradNorm:0.06450862	learnRate:0.00000000

LASSO 算法选出的最重要的 3 个特征为 alcohol、volatileAcidity 和 residualSugar。对比线性回归的 density、residualSugar 和 alcohol，特征 residualSugar 与 alcohol 都保留在前三位，在前面计算相关系数时，volatileAcidity 与 quality 的相关系数也相对较高，所以 volatileAcidity 的出现

也在意料之中；特征 density 与特征 residualSugar 与 alcohol 的相关性较强，权重值下降显著。

模型信息如下。

model meta info										
hasInterception: true, model name: LASSO, num feature: 11										
model weight info										
colName[0, 9]	intercept	fixedAcidity	volatileAcidity	citricAcid	residualSugar	chlorides	freeSulfurDioxide	totalSulfurDioxide	density	pH
weight[0, 9]	21.7008	-0.01094269	-1.40732785	0.00000000	0.01094060	-0.90947378	0.00132137	0.00000000	-18.19511910	0.00000000
colName[10, 11]	sulphates	alcohol								
weight[10, 11]	0.03046873	0.24841285								

这里出现了 3 个权重为 0 的特征： citricAcid、 totalSulfurDioxide 和 pH。

模型的评估指标如下，相比于线性回归模型，指标变化不大。

Metrics:			
MSE: 0.5555	RMSE: 0.7453	MAE: 0.5904	MAPE: 10.3725
R2: 0.2315			

16.4 决策树与随机森林

本节将尝试使用决策树和随机森林算法，看看能否获得比线性回归更好的效果。

先看决策树回归器 DecisionTreeRegressor 的实验，具体代码如下。

```
new DecisionTreeRegressor()
.setFeatureCols(FEATURE_COL_NAMES)
.setLabelCol(LABEL_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.fit(train_data)
.transform(test_data)
.link(
    new EvalRegressionBatchOp()
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .lazyPrintMetrics("DecisionTreeRegressor")
);
BatchOperator.execute();
```

其中设置了特征、标签及预测结果列，其他参数使用默认值。

运行结果如下。

Metrics:			
MSE: 0.6215	RMSE: 0.7884	MAE: 0.494	MAPE: 8.7346
R2: 0.1401			

各项指标要落后于线性回归算法。

使用多棵决策树构成的随机森林比较容易获得更好的指标。逐渐调整随机森林中树的棵数，看其对回归效果的影响，如下代码所示。

```

for (int numTrees : new int[] {2, 4, 8, 16, 32, 64, 128}) {
    new RandomForestRegressor()
        .setNumTrees(numTrees)
        .setFeatureCols(FEATURE_COL_NAMES)
        .setLabelCol(LABEL_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .fit(train_data)
        .transform(test_data)
        .link(
            new EvalRegressionBatchOp()
                .setLabelCol(LABEL_COL_NAME)
                .setPredictionCol(PREDICTION_COL_NAME)
                .lazyPrintMetrics("RandomForestRegressor - " + numTrees)
        );
    BatchOperator.execute();
}

```

整理运行结果，如表 16-4 所示，可以看到 64 棵树时的回归效果明显优于线性回归；在 64 棵树之前，随着棵树数的增加，各项指标都在提升，但提升的幅度在下降；在 128 棵树时并没有获得更好的效果。因此，在实际应用中，我们需要考虑回归效果和计算代价，选择合适的棵树。

表 16-4 随机森林算法选择不同棵树的参数

棵树	MSE	RMSE	MAE	MAPE	R2
2	0.5126	0.716	0.4945	8.6621	0.2907
4	0.4307	0.6563	0.459	8.0789	0.404
8	0.3866	0.6218	0.4361	7.6881	0.4651
16	0.3705	0.6087	0.428	7.5556	0.4874
32	0.3645	0.6037	0.4256	7.5223	0.4957
64	0.3613	0.601	0.4245	7.4985	0.5002
128	0.3617	0.6014	0.4227	7.4711	0.4996

16.5 GBDT 回归

我们再尝试一个非线性的回归模型——GBDT 模型，流程与随机森林模型相同，使用 GBDT 回归组件 GbdtRegressor，设置特征列为 11 个特征，标签列为 quality，并设置预测结果列。为了获取更好的回归模型，我们需要尝试多个参数，这里列出了一组参数，具体设置如下代码所示。

```
for (int numTrees : new int[] {16, 32, 64, 128, 256, 512}) {
```

```

new GbdtRegressor()
    .setLearningRate(0.05)
    .setMaxLeaves(256)
    .setFeatureSubsamplingRatio(0.3)
    .setMinSamplesPerLeaf(2)
    .setMaxDepth(100)
    .setNumTrees(numTrees)
    .setFeatureCols(FEATURE_COL_NAMES)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .fit(train_data)
    .transform(test_data)
    .link(
        new EvalRegressionBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("GbdtRegressor - " + numTrees)
    );
BatchOperator.execute();
}

```

汇总运行结果如表 16-5 所示，回归指标要优于随机森林。

表 16-5 GBDT 算法选择不同棵数的参数

棵数	MSE	RMSE	MAE	MAPE	R2
16	0.4906	0.7005	0.5465	9.6852	0.3212
32	0.424	0.6511	0.5095	9.0017	0.4134
64	0.3824	0.6184	0.4691	8.2692	0.4709
128	0.3633	0.6027	0.4356	7.6824	0.4974
256	0.3552	0.596	0.4062	7.176	0.5086
512	0.355	0.5958	0.3926	6.9417	0.5088

17

常用聚类算法

聚类就是将若干个对象的集合分割成几个类，每个类内的对象之间是相似的，但与其他类的对象是不相似的。

例如，前面使用过的 Iris 数据集，每个数据有 4 个特征，可以看作四维空间中的点。我们无法显示数据在四维空间的分布情况，但可以将其 4 个特征进行两两组合，分别绘制散点图，并按类别标以不同的颜色，散点图矩阵如图 17-1 所示。

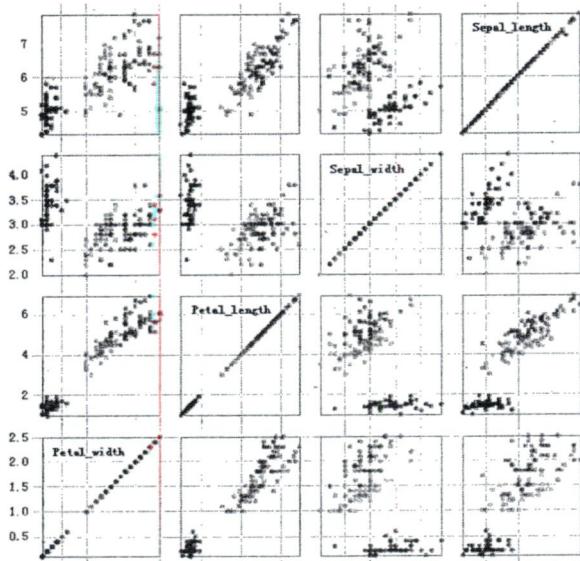


图 17-1

在每个散点图中，这些数据都会明显聚为两簇，根据这些二维平面的投影情况，我们会猜想其在四维空间中应该也会明显地聚合为两类。从类别颜色上看，其中一簇中包含两个类别，它们的界限并不清晰，如果我们的目标是聚为三簇，那么这两个类也会分属于不同的簇，但在每个簇中会以某个类别为主，掺杂其他类别的一些点。

17.1 聚类评估指标

聚类个数为 K 个，整个数据集的中心为 u ，样本总数为 N ，聚类结果为 $\Omega = \{\omega_1, \omega_2, \dots, \omega_K\}$ 。对于聚类簇 ω_k ， u_k 为聚类中心，包含 n_k 条样本，即 $n_k = |\omega_k|$ 。

17.1.1 基本评估指标

本节介绍 5 个基本的评估指标。

(1) 紧密度 (Compactness, CP)

$$CP_k = \frac{1}{|\omega_k|} \sum_{x \in \omega_k} \|x - u_k\|$$

$$CP = \frac{1}{K} \sum_{k=1}^K CP_k$$

CP 越小，意味着类内聚类距离越小。

(2) 分离度 (Separation, SP)

$$SP = \frac{2}{K^2 - K} \sum_{i=1}^K \sum_{j=i+1}^K \|u_i - u_j\|$$

SP 越大，意味着类间聚类距离越大。

(3) Davies-Bouldin (DB) 指数

$$DB = \frac{1}{K} \sum_{i=1}^K \max_{j \neq i} \left(\frac{CP_i + CP_j}{\|u_i - u_j\|} \right)$$

DB 越小，意味着类内距离越小，同时类间距离越大。

(4) 方差比准则 (Calinski-Harabasz Index, VRC)

$$SSB = \sum_{k=1}^K n_k \|u_k - u\|^2$$

$$SSW = \sum_{k=1}^K \sum_{x \in \omega_k} \|x - u_k\|^2$$

$$VRC = \frac{SSB}{SSW} \cdot \frac{N - K}{K - 1}$$

其中, SSB 是组与组之间的平方和误差, SSW 是组内平方和误差。如果 SSW 越小、SSB 越大, 那么 VRC 越大, 聚类效果越好。

(5) 轮廓系数 (Silhouette Coefficient)

用于描述聚类中各簇的轮廓清晰度。将所有样本的轮廓系数求平均值, 就是该聚类结果总的轮廓系数。

下面主要介绍单个样本轮廓系数的计算。

对于属于簇 ω_k 的样本 i , 有以下情况。

- 计算样本 i 到同簇其他样本的平均距离 $a(i)$ 。 $a(i)$ 越小, 说明样本 i 越应该被聚类到该簇。 $a(i)$ 被称为样本 i 的簇内不相似度。
- 计算样本 i 到其他某簇 ω_j 的所有样本的平均距离 $b_j(i)$, 称为样本 i 与簇 ω_j 的不相似度。定义为样本 i 的簇间不相似度为

$$b(i) = \min\{b_1(i), \dots, b_{k-1}(i), b_{k+1}(i), \dots, b_K(i)\}$$

- 根据样本 i 的簇内不相似度 $a(i)$ 和簇间不相似度 $b(i)$, 定义样本 i 的轮廓系数为

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

也可以换一个形式表达, 看上去更清晰。

$$S(i) = \begin{cases} 1 - \frac{a(i)}{b(i)}, & a(i) < b(i) \\ 0, & a(i) = b(i) \\ \frac{b(i)}{a(i)} - 1, & a(i) > b(i) \end{cases}$$

可知轮廓系数的取值范围是 $[-1, 1]$ 。当 $S(i)$ 接近 1 时，样本*i*聚类合理； $S(i)$ 接近-1，则说明样本*i*更应该分类到其他簇；若 $S(i)$ 近似为 0，则说明样本*i*在两个簇的边界上。

上面介绍的指标计算只需要聚类结果，如果数据集还有标记好的类别信息，则还可以提供更多的指标。

17.1.2 基于标签值的评估指标

设样本数据分属于 M 个标签类别 $C = \{c_1, c_2, \dots, c_M\}$ ， $|\omega_k \cap c_m|$ 为聚类 ω_k 中的样本属于类别 c_m 的个数， P_{km} 为聚类 ω_k 中的样本属于类别 c_m 的概率，即

$$P_{km} = \frac{|\omega_k \cap c_m|}{|\omega_k|}$$

则每个聚类的熵（Entropy）可以表示为

$$e_k = - \sum_{m=1}^M P_{km} \log(P_{km})$$

整个聚类划分的熵表示为

$$e = \sum_{k=1}^K \frac{n_k}{N} e_k$$

- 纯度（Purity）

$$\text{purity}(\Omega, C) = \sum_{k=1}^K \frac{n_k}{N} \max_m (P_{km}) = \frac{1}{N} \sum_{k=1}^K \max_m |\omega_k \cap c_m|$$

Purity 在 $[0,1]$ 区间内，越接近 1，表示聚类结果越好。

- 归一化互信息（Normalized Mutual Information，NMI）

$$H(\Omega) = - \sum_{k=1}^K \frac{\omega_k}{N} \log\left(\frac{\omega_k}{N}\right)$$

$$H(C) = - \sum_{m=1}^M \frac{c_m}{N} \log\left(\frac{c_m}{N}\right)$$

$$I(\Omega, C) = \sum_{k=1}^K \sum_{m=1}^M \frac{|\omega_k \cap c_m|}{N} \log \left(\frac{N|\omega_k \cap c_m|}{|\omega_k| \cdot |c_m|} \right)$$

$$\text{NMI} = \frac{2 \cdot I(\Omega, C)}{H(\Omega) + H(C)}$$

NMI 在 [0,1] 区间内，越接近 1，表示聚类结果越好。

整个数据集会有 $\binom{N}{2} = \frac{N(N-1)}{2}$ 个样本对，考虑如下 4 种情况。

- TP：同一类的样本被分到同一个簇。
- TN：不同类的样本被分到不同簇。
- FP：不同类的样本被分到同一个簇。
- FN：同一类的样本被分到不同簇。

则有

$$\begin{aligned} \text{TP} + \text{FP} &= \sum_{m=1}^M \binom{|c_m|}{2} \\ \text{TP} + \text{FN} &= \sum_{k=1}^K \binom{|\omega_k|}{2} \\ \text{TP} &= \sum_{k=1}^K \sum_{m=1}^M \binom{|\omega_k \cap c_m|}{2} \\ \text{TP} + \text{TN} + \text{FP} + \text{FN} &= \binom{N}{2} \end{aligned}$$

可以计算得到

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

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

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

- 兰德系数 (Rand Index, RI)

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

RI 在[0,1]区间内，越接近 1，表示聚类结果越好，意味着聚类结果与标签类别情况越吻合。

- 调整兰德系数（Adjusted Rand Index，ARI）

$$\text{Index} = TP$$

$$\text{ExpectedIndex} = \frac{(TP + FP)(TP + FN)}{TP + TN + FP + FN}$$

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

$$ARI = \frac{\text{Index} - \text{ExpectedIndex}}{\text{MaxIndex} - \text{ExpectedIndex}}$$

ARI 在[-1,1]区间内，越接近 1，表示聚类结果越好；在聚类结果为随机产生的时候，指标会接近零。

17.2 K-Means聚类

K 均值（K-Means）聚类，用户定义所要的类的个数 K ，算法按照距离将数据自动聚合为 K 个类，使类内的对象的距离近，而不同类内的对象的距离远，每个类以该类所有数据的“均值”作为其中心点。

17.2.1 算法简介

算法描述如下。

- ①在给定的 n 个对象中，随机选取 K 个对象作为每个类的中心（初始均值），或者通过其他方式指定 K 个中心。
- ②对于全部 n 个对象，分别计算与 K 个中心的距离，将对象指派到最接近的类。
- ③更新每个类的新均值，得出 K 个新的中心。
- ④根据 K 个新的中心，重复第②步和第③步，直至满足收敛准则。

下面通过一个例子使读者更直观地理解上述算法。我们生成均匀分布的二维数据，每个维度的取值范围都是[0, 100]，生成 100 亿个这样的数据点。聚类的个数设为 9，我们可以想象到

最佳的聚类状态：将整个区域按“井”字等分成 9 份。

下面通过 K-Means 聚类算法实际计算一下，这里假设最坏的初始点情况为 9 个点重合且位于区域的边缘，如图 17-2 的左上图所示。随后展现了第 5 次迭代和第 10 次迭代后的情况，可以发现，重合的初始中心点被展开了；第 30 次迭代时，中心点已在整个区域散开了；第 50 次迭代时，这些中心点按 3×3 的位置排列；第 100 次迭代时，中心点已经排列得非常整齐了。

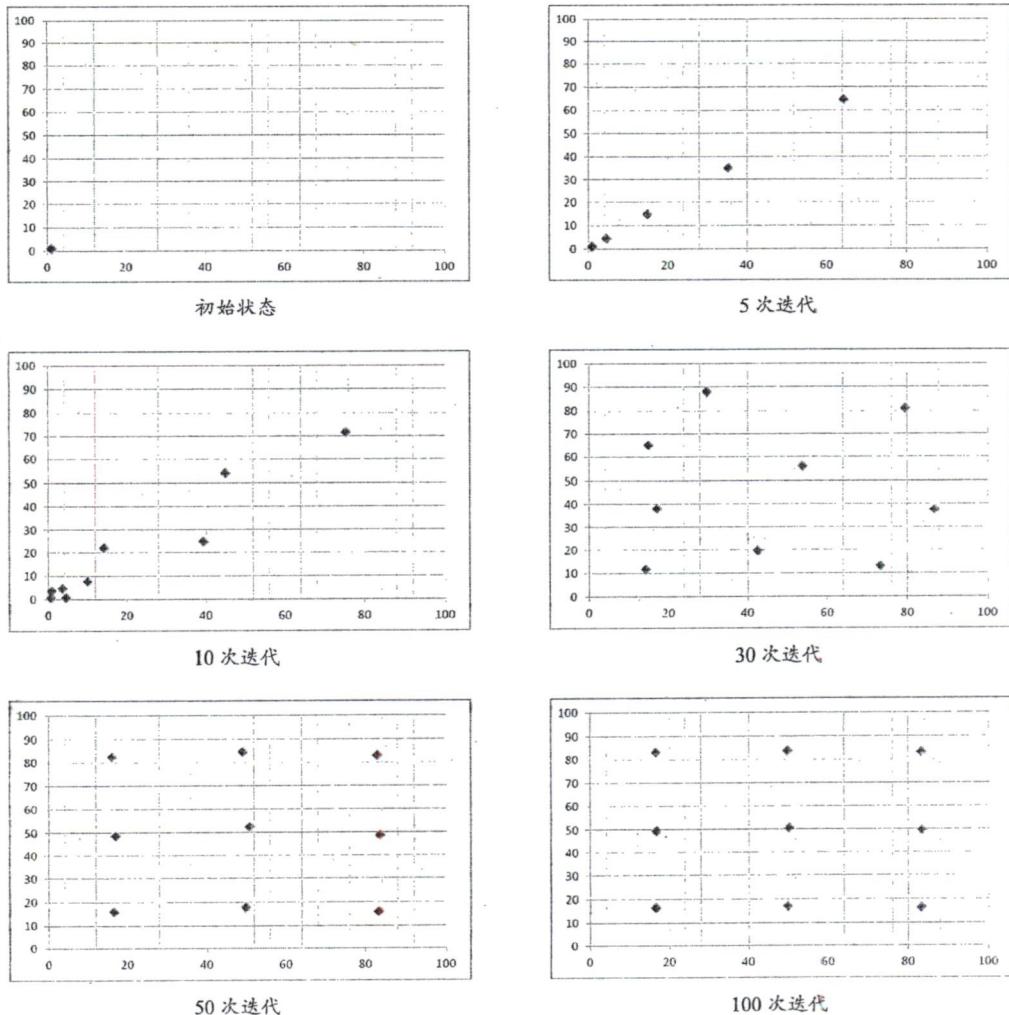


图 17-2

17.2.2 K-Means实例

我们尝试使用 K-Means 算法对数据集 Iris 进行自动聚类。由于聚类组件支持的数据为向量格式，我们需要将原始数据集 Iris 中各特征列合并为向量。使用向量装配组件 VectorAssemblerBatchOp，将 Iris 数据集的各特征列作为输入列，输出为一个向量列，并把转换结果保存为 AK 格式文件。具体代码如下。

```
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)
    )
    .link(
        new AkSinkBatchOp().setFilePath(DATA_DIR + VECTOR_FILE)
    );
};
```

使用 AkSourceBatchOp 读取前面产生的向量数据文件，并输出 5 条数据，具体代码如下。

```
AkSourceBatchOp source = new AkSourceBatchOp().setFilePath(DATA_DIR + VECTOR_FILE);
source.lazyPrint(5);
```

输出数据如下，第 2 列就是向量序列化字符串的数据，各个数值间使用空格分隔。

category	vec
Iris-versicolor	5.0 2.0 3.5 1.0
Iris-versicolor	5.9 3.0 4.2 1.5
Iris-versicolor	6.0 2.2 4.0 1.0
Iris-versicolor	6.1 2.9 4.7 1.4
Iris-versicolor	5.6 2.9 3.6 1.3

下面使用 K-Means 组件进行聚类实验，需要用到 K-Means 训练组件 KMeansTrainBatchOp，设置聚类个数为 2，并设置向量列名称；还用到了 K-Means 预测组件 KMeansPredictBatchOp，需要设置聚类预测结果数据列的名称。数据源连接训练组件，训练组件的输出为 K-Means 模型；再用预测组件接收来自 K-Means 模型和数据源的连接，预测组件的输出即为聚类结果。具体代码如下。

```
KMeansTrainBatchOp kmeans_model = new KMeansTrainBatchOp()
    .setK(2)
    .setVectorCol(VECTOR_COL_NAME);
```

```
KMeansPredictBatchOp kmeans_pred = new KMeansPredictBatchOp()
.setPredictionCol(PREDICTION_COL_NAME);

source.link(kmeans_model);
kmeans_pred.linkFrom(kmeans_model, source);

kmeans_model.lazyPrintModelInfo();

kmeans_pred.lazyPrint(5);
```

模型信息显示如下。

```
===== KMeansInfo =====
KMeans clustering with 2 clusters. Clustering on 150 samples of 4 dimension based on EUCLIDEAN.
===== ClusterCenters =====
[0: [6.301, 2.8866, 4.9588, 1.6959],
 1: [5.0057, 3.3604, 1.5623, 0.2887]]
```

可以看到聚类个数、样本总数及样本中向量的维度，以及采用的是何种距离。下面列出了各个聚类簇中心的向量。

聚类输出结果如下。

```
category|vec|cluster_id
-----|---|-----
Iris-versicolor|5.0 2.0 3.5 1.0|1
Iris-versicolor|5.9 3.0 4.2 1.5|1
Iris-versicolor|6.0 2.2 4.0 1.0|1
Iris-versicolor|6.1 2.9 4.7 1.4|1
Iris-versicolor|5.6 2.9 3.6 1.3|1
```

cluster_id 就是聚类结果预测列，为从 0 开始的聚类索引值。评估聚类结果的好坏不像评估分类问题那样，直接与标签值对比是否一致即可让我们对它有一个大致的印象。在 17.1 节中有详细介绍，其中一部分指标（譬如 VRC、DB、SilhouetteCoefficient 等）是通过样本的向量值和聚类情况计算出来的，还有的指标（譬如 ARI、NMI、Purity 等）是通过样本的标签值与聚类情况计算出来的。当然，如果样本没有标签值，就无法计算这些指标，它们也不会在评估结果中显示。

下面我们使用聚类评估组件 EvalClusterBatchOp 对前面的 K-Means 聚类结果进行评估。

```
kmeans_pred
.link(
    new EvalClusterBatchOp()
    .setVectorCol(VECTOR_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .lazyPrintMetrics("KMeans EUCLIDEAN")
);
```

输出评估结果如下。

```
KMeans EUCLIDEAN
Metrics:
k:2
VRC:513.3038 DB:0.4048 SilhouetteCoefficient:0.8502
ARI:0.5399 NMI:0.6565 Purity:0.6667

| Label\Cluster | 1 | 0 |
|-----|---|---|
| Iris-virginica | 0 | 50 |
| Iris-versicolor | 3 | 47 |
| Iris-setosa | 50 | 0 |
```

其中显示了一些常用的指标和一个交叉表，对样本的标签值和聚类情况进行了细分。可以看出，标签为 Iris-virginica 的样本都在第 0 个聚类簇；标签为 Iris-setosa 的样本都在第 1 个聚类簇；标签为 Iris-versicolor 的样本在 2 个聚类簇中都有分布，主要分布在第 0 个聚类簇。

下面我们将着重分析 Iris-versicolor 标签的样本，看看其属于不同聚类簇的样本的特点。使用 SQL OrderBy 对样本数据进行排序，先是按聚类结果排序，然后按标签值排序；将输出范围设定为 200，因为 Iris 的总样本数为 150，因此超出 Iris 的总样本数会全部排序输出；将排序是否为升序的参数设为 false，表示使用降序。具体代码如下。

```
kmeans_pred
.orderBy(PREDICTION_COL_NAME + ", " + LABEL_COL_NAME, 200, false)
.lazyPrint(-1, "all data");
```

运行结果如下，这里省略了前端的样本，主要展示标签为 Iris-versicolor，且在第 0 个聚类簇的 3 个样本。

```
...
Iris-versicolor|5.7 2.8 4.1 1.3|0
Iris-versicolor|6.0 2.7 5.1 1.6|0
Iris-versicolor|6.1 3.0 4.6 1.4|0
Iris-versicolor|5.8 2.7 4.1 1.0|0
Iris-versicolor|6.6 3.0 4.4 1.4|0
Iris-versicolor|5.5 2.3 4.0 1.3|0
Iris-versicolor|6.2 2.9 4.3 1.3|0
Iris-versicolor|5.5 2.4 3.7 1.0|0
Iris-versicolor|5.5 2.5 4.0 1.3|0
Iris-versicolor|5.1 2.5 3.0 1.1|1
Iris-versicolor|4.9 2.4 3.3 1.0|1
Iris-versicolor|5.0 2.3 3.3 1.0|1
Iris-setosa|4.9 3.0 1.4 0.2|1
Iris-setosa|4.9 3.1 1.5 0.1|1
Iris-setosa|5.1 3.5 1.4 0.3|1
```

```
Iris-setosa|5.0 3.0 1.6 0.2|1
Iris-setosa|5.5 4.2 1.4 0.2|1
Iris-setosa|4.5 2.3 1.3 0.3|1
Iris-setosa|5.0 3.3 1.4 0.2|1
Iris-setosa|4.6 3.1 1.5 0.2|1
...
...
```

可以看到，标签为 Iris-versicolor 且属于第 1 个聚类簇的 3 个样本，与同为 Iris-versicolor 标签的其他样本相比，在“尺寸”上相对较小，在各项指标上较同类小，这样从“距离”的角度看，它和 Iris setosa 类的花更近些。对比下面标签为 Iris-setosa 的数据，会发现其向量值的最后一项都小于 0.3，但在计算欧氏距离时，我们关注的是两个样本间相对的数值。我们是否能换一个度量方式进行聚类呢？

我们可以选择 K-Means 算法使用余弦度量。前面讲了如何使用 K-Means 的训练和预测组件，这里我们使用 Pipeline 的方式，直接使用 PipelineStage K-Means，可以通过 fit 和 transform 方法简化训练和预测流程，使用 Lazy 方式输出模型信息，并连接聚类评估组件。具体代码如下。

```
new KMeans()
.setK(2)
.setDistanceType(DistanceType.COSINE)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.enableLazyPrintModelInfo()
.fit(source)
.transform(source)
.link(
    new EvalClusterBatchOp()
        .setVectorCol(VECTOR_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .setLabelCol(LABEL_COL_NAME)
        .lazyPrintMetrics("KMeans COSINE")
);

```

输出模型信息如下，聚类个数为 2，使用 COSINE（余弦）度量。

```
KMeansInfo
KMeans clustering with 2 clusters. Clustering on 150 samples of 4 dimension based on COSINE.
ClusterCenters
{0: [0.8027, 0.5467, 0.2352, 0.0389],
 1: [0.7282, 0.3349, 0.5662, 0.1924]}
```

评估结果如下。

```
Metrics:
k:2
VRC:501.9249 DB:0.3836 SilhouetteCoefficient:0.8462
ARI:0.5681 NMI:0.7337 Purity:0.6667
```

Label\Cluster	1	0
Iris-virginica	50	0
Iris-versicolor	50	0
Iris-setosa	0	50

可以看到，新的聚类方式可以将 Iris-Setosa 标签的样本与其他两种标签的样本自动聚合为两类，也就是完整地分隔开了。

17.3 高斯混合模型

高斯混合模型（Gaussian Mixed Model, GMM）用多维高斯概率模型来描述每个类，将聚类问题看作多个高斯分布函数的线性组合。

17.3.1 算法介绍

高斯混合模型用多维高斯概率模型来描述每个类，通过计算概率的方式来确定数据点属于哪个类。通过期望最大化（Expectation Maximization, EM）方法进行迭代，求得模型参数的估计值。下面首先介绍多维高斯分布，然后介绍如何使用 EM 方法求解。

所用的概率模型为多维高斯分布，将每条记录的属性集作为一个向量，每个特征作为向量的一个分量，记作 $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$ ，各特征列的均值向量为 $\mathbf{u} = \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_m \end{pmatrix}$ ，协方差矩阵为 $\mathbf{C} = \begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mm} \end{pmatrix}$ ，则多维高斯分布的概率密度函数定义为

$$N(\mathbf{x}; \mathbf{u}, \mathbf{C}) = \frac{1}{(2\pi)^{m/2} |\mathbf{C}|^{1/2}} \cdot \exp \left\{ -\frac{1}{2} (\mathbf{x} - \mathbf{u})^\top \mathbf{C}^{-1} (\mathbf{x} - \mathbf{u}) \right\}$$

观察这个公式，我们可以很容易地理解该方法和 K-Means 算法的关联。从协方差矩阵 \mathbf{C} 入手，假设各个特征之间是独立的、各个特征的方差值相等。

由第一个假设可以知道，对于两个不同的特征 i, j ，有 $c_{ij} = 0$ ，所以协方差矩阵 \mathbf{C} 只有对角线上的元素不为 0。再由第二个假设，设相等的方差值为 σ^2 ，则

$$\mathbf{C} = \begin{pmatrix} \sigma^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma^2 \end{pmatrix}$$

其逆矩阵为

$$\mathbf{C}^{-1} = \begin{pmatrix} \frac{1}{\sigma^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma^2} \end{pmatrix}$$

m 维矩阵 \mathbf{C} 的行列式 $|\mathbf{C}|$ 的值为

$$|\mathbf{C}| = (\sigma^2)^m$$

则相应的概率密度函数为

$$\begin{aligned} N(\mathbf{x}; \mathbf{u}, \mathbf{C}) &= \frac{1}{(2\pi)^{m/2} |\mathbf{C}|^{1/2}} \cdot \exp \left\{ -\frac{1}{2} (\mathbf{x} - \mathbf{u})^\top \mathbf{C}^{-1} (\mathbf{x} - \mathbf{u}) \right\} \\ &= \frac{1}{(2\pi)^{m/2} (\sigma^2)^{m/2}} \cdot \exp \left\{ -\frac{1}{2} (\mathbf{x} - \mathbf{u})^\top \begin{pmatrix} \frac{1}{\sigma^2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma^2} \end{pmatrix} (\mathbf{x} - \mathbf{u}) \right\} \\ &= \frac{1}{(2\pi\sigma^2)^{m/2}} \cdot \exp \left\{ -\frac{1}{2\sigma^2} \sum_{i=1}^m (x_i - \mu_i)^2 \right\} \end{aligned}$$

$\sum_{i=1}^m (x_i - \mu_i)^2$ 即为 K-Means 算法所使用的向量 \mathbf{x} 和 \mathbf{u} 的欧氏距离的平方，而由上面的公式可知，该平方与概率密度的值呈负指数关系，即 \mathbf{x} 与中心 \mathbf{u} 的欧氏距离越小，所对应的概率密度的值越大。

下面详细讲解一下 GMM 算法的过程，我们可以了解到 K-Means 和 GMM 的更多相似之处。对于 GMM 聚类算法，假设在 K 个聚类的情况下，每个数据 \mathbf{x} 都满足如下的概率分布。

$$p(\mathbf{x}) = \sum_{k=1}^K \pi_k N(\mathbf{x}; \mathbf{u}_k, \mathbf{C}_k)$$

对数似然函数为

$$\begin{aligned} \log \left(\prod_{i=1}^N p(\mathbf{x}_i) \right) &= \log \left(\prod_{i=1}^N \sum_{k=1}^K \pi_k N(\mathbf{x}_i; \mathbf{u}_k, \mathbf{C}_k) \right) \\ &= \sum_{i=1}^N \log \left(\sum_{k=1}^K \pi_k N(\mathbf{x}_i; \mathbf{u}_k, \mathbf{C}_k) \right) \end{aligned}$$

对数据进行训练的目的就是得到使对数似然函数达到最大值的参数 π_k, \mathbf{u}_k, C_k 。
算法过程如下。

(1) 用随机函数初始化 K 个高斯分布的参数，同时保证

$$\sum_{k=1}^K \pi_k = 1$$

(2) (E 步骤) 对每个训练数据 x ，计算 K 个高斯函数 $\pi_k \cdot N(x; \mathbf{u}_k, C_k)$ 中概率的大小，把概率最大的作为其分类。

(3) (M 步骤) 用最大似然估计，在 E 步骤计算的分类的基础上，估计相应的参数值，具体公式如下。

$$\begin{aligned}\mathbf{u}_k &= \frac{1}{N_k} \sum_{x_k} x_k \\ C_k &= \frac{1}{N_k} \sum_{x_k} (x_k - \mathbf{u}_k)^T (x_k - \mathbf{u}_k) \\ \pi_k &= \frac{N_k}{N}\end{aligned}$$

(4) 判断是否收敛，若收敛，则返回当前参数；否则返回第 2 步继续计算。

在 M 步骤中，新中心点 \mathbf{u}_k 的计算公式与 K-Means 相同，但在 E 步骤判断分类时，还要考虑每个分类占总体的比例 π_k ，不像 K-Means 只考虑距离。

17.3.2 GMM 实例

下面我们利用 GMM 算法对 17.2 节中的数据进行聚类。使用组件设置向量列名称、设置聚类结果列名称，定义聚类个数为 2，并选择使用 Lazy 方式输出模型信息。具体代码如下。

```
new GaussianMixture()
.setK(2)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.enableLazyPrintModelInfo()
.fit(source)
.transform(source)
.link(
    new EvalClusterBatchOp()
        .setVectorCol(VECTOR_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME))
```

```

.setLabelCol(LABEL_COL_NAME)
.lazyPrintMetrics("GaussianMixture 2")
);

```

显示模型信息如下，包括各聚类中心点的向量及各聚类的协方差矩阵。

```

----- GMMInfo -----
GMM clustering with 2 clusters.
----- ClusterCenters -----
{0: [6.262, 2.872, 4.906, 1.676],
 1: [5.006, 3.418, 1.464, 0.244]}
----- CovarianceMatrix of each clusters -----
{0: mat[4,4]:
 [[0.435, 0.1209, 0.4489, 0.1655],
 [0.1209, 0.1096, 0.1414, 0.0792],
 [0.4489, 0.1414, 0.6749, 0.2859],
 [0.1655, 0.0792, 0.2859, 0.1786]],
 1: mat[4,4]:
 [[0.1218, 0.0983, 0.0158, 0.0103],
 [0.0983, 0.1423, 0.0114, 0.0112],
 [0.0158, 0.0114, 0.0295, 0.0056],
 [0.0103, 0.0112, 0.0056, 0.0113]]}

```

各原始数据的聚类评估如下。

```

----- Metrics: -----
k:2
VRC:501.9249 DB:0.3836 SilhouetteCoefficient:0.8462
ARI:0.5681 NMI:0.7337 Purity:0.6667

```

Label\Cluster	1	0
Iris-virginica	0	50
Iris-versicolor	0	50
Iris-setosa	50	0

显然，Iris setosa 亚属的样本与其他两个亚属的样本被自动聚合为两类，即被完整地分隔开了。其结果要优于使用欧氏距离的 K-Means 算法，与使用夹角余弦作为距离的 K-Means 算法的聚类结果相同。

17.4 二分K-Means聚类

二分 K 均值 (Bisecting K-Means) 聚类算法是 K-Means 聚类算法的一个变体，采用“自顶向下”方法的层次聚类。首先将全部数据点作为一个簇，使用 K-Means 算法将其分为两个簇，

然后在所有簇中选择聚类指标误差平方和最大的簇，继续使用 K-Means 算法将其分为两个簇，持续进行下去，每次一分二的操作都会增加一个簇，直到达到用户的聚类数目为止。

二分 K-Means 聚类组件的使用方式和参数设置与 K-Means 聚类组件完全相同。如下代码所示，使用组件 BisectingKMeans，设置聚类个数为 3。

```
new BisectingKMeans()
    .setK(3)
    .setVectorCol(VECTOR_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .enableLazyPrintModelInfo("BiSecting KMeans EUCLIDEAN")
    .fit(source)
    .transform(source)
    .link(
        new EvalClusterBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .lazyPrintMetrics("Bisecting KMeans EUCLIDEAN")
    );
BatchOperator.execute();
```

整理运行结果，并与 K-Means 二分类的结果进行对比，如表 17-1 所示。

表 17-1 聚类中心对比

Bisecting K-Means 3 个聚类簇	K-Means 2 个聚类簇
聚类簇中心： {0: [5.0057, 3.3604, 1.5623, 0.2887], 1: [5.9475, 2.7661, 4.4542, 1.4542], 2: [6.85, 3.0737, 5.7421, 2.0711]}	聚类簇中心： {0: [6.301, 2.8866, 4.9588, 1.6959], 1: [5.0057, 3.3604, 1.5623, 0.2887]}
VRC:520.6334 DB:0.6755 SilhouetteCoefficient:0.7232 ARI:0.6809 NMI:0.6914 Purity:0.8733 Label\Cluster 2 1 0 ----- --- --- --- Iris-virginica 36 14 0 Iris-versicolor 2 45 3 Iris-setosa 0 0 50	VRC:513.3038 DB:0.4048 SilhouetteCoefficient:0.8502 ARI:0.5399 NMI:0.6565 Purity:0.6667 Label\Cluster 1 0 ----- --- --- Iris-virginica 0 50 Iris-versicolor 3 47 Iris-setosa 50 0

可以得到如下信息。

- 3 个聚类簇的第 0 簇中心为[5.0057, 3.3604, 1.5623, 0.2887]，与 2 个聚类簇的第 1 簇中心的位置相同。

- 左边的第 0 簇与右边的第 1 簇都是由全部标签为 Iris-setosa 的样本和 3 个 Iris-versicolor 标签的样本构成的。

- 左边的第 1、2 簇是将右边的第 0 簇进行一次二分聚类产生的。

K-Means 算法在使用 COSINE 度量时，实现了完美的二分聚类。这里我们使用二分 K-Means 聚类算法，同样也选择 COSINE 度量，看看分类效果如何。相比前面的示例，只需设置距离类型 DistanceType=COSINE。具体代码如下。

```

new BisectingKMeans()
.setDistanceType(DistanceType.COSINE)
.setK(3)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
.enableLazyPrintModelInfo("BiSecting KMeans COSINE")
.fit(source)
.transform(source)
.link(
    new EvalClusterBatchOp()
        .setDistanceType("COSINE")
        .setVectorCol(VECTOR_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .setLabelCol(LABEL_COL_NAME)
        .lazyPrintMetrics("Bisecting KMeans COSINE")
);
BatchOperator.execute();

```

整理运行结果，并与 K-Means 二分类的结果进行对比，如表 17-2 所示。

表 17-2 COSINE 度量的聚类中心对比

Bisecting K-Means 3 个聚类簇	K-Means 2 个聚类簇
聚类簇中心： {0: [0.8027, 0.5467, 0.2352, 0.0389], 1: [0.7536, 0.3495, 0.532, 0.1641], 2: [0.7058, 0.3222, 0.5931, 0.2152]}	聚类簇中心： {0: [0.8027, 0.5467, 0.2352, 0.0389], 1: [0.7282, 0.3349, 0.5662, 0.1924]}
VRC:21836.8509 DB:0.3085 SilhouetteCoefficient:0.8488 ARI:0.9039 NMI:0.8997 Purity:0.9667 Label\Cluster 2 1 0 ----- --- --- --- Iris-virginica 50 0 0 Iris-versicolor 5 45 0 Iris-setosa 0 0 50	VRC:501.9249 DB:0.3836 SilhouetteCoefficient:0.8462 ARI:0.5681 NMI:0.7337 Purity:0.6667 Label\Cluster 1 0 ----- --- --- Iris-virginica 50 0 Iris-versicolor 50 0 Iris-setosa 0 50

可以得到如下信息。

- 3 个聚类簇的第 0 簇中心为 [0.8027, 0.5467, 0.2352, 0.0389]，与 2 个聚类簇的第 0 簇中心的位置相同。
- 3 个聚类簇的各项聚类指标都优于 2 个聚类簇。DB 指标越小越好，VRC、Silhouette-Coefficient、ARI、NMI、Purity 都是越大越好。
- 左右两边的第 0 簇都是完全由标签为 Iris-setosa 的样本构成的，左边的第 1、2 簇是将右边的第 1 簇进行一次二分聚类产生的。

有了 17.4 节的基础，下面我们来挑战难度更高的聚类数为 3 的情形。原始的样本数据本来属于 3 个分类，从散点图可以看出，Iris virginica 和 Iris versicolor 分类的样本点没有清晰的分界线。如何能较好地聚成 3 簇，还尽量与其原来的分类属性一致呢？

我们先尝试 17.4 节的最佳参数组合，在余弦距离下，选择 2 个特征列与选择全部特征列的实验组件，对于选择两个特征 petal_width 和 petal_length 的情况，散点图中左下角聚集的点被分到 3 个聚类中。

下面尝试在欧氏距离下，分别选择 2 个特征列与选择全部特征列的实验组件，从散点图上不容易看出区别，通过对比原始分类类别与聚类结果的统计数据可知，选择 2 个特征列的情况会更好一些。

综合起来进行对比，我们虽然没有找到一种参数设置，能够使原始的分类类别与聚类结果一致，但是可以使它们之间的差异变小。

17.5 基于经纬度的聚类

我们将使用美国 50 个州首府的经纬度数据，通过 Alink 经纬度聚类组件 GeoKMeans 进行聚类试验。

各州首府的经纬度数据集显示如下。

State	Region	Division	longitude	latitude
Alabama	South	East South Central	-86.7509	32.5901
Alaska	West	Pacific	-127.2500	49.2500
Arizona	West	Mountain	-111.6250	34.2192
Arkansas	South	West South Central	-92.2992	34.7336
California	West	Pacific	-119.7730	36.5341

第 1 列是州名，第 2 列和第 3 列分别是对 50 个州的两种划分方式，Region 划分为 4 个部分，Division 划分为 9 个部分，第 4 列为州首府所在的经度值，第 5 列为州首府所在的纬度值。

为了进一步了解 Region 与 Division 的关系，我们可以使用 SQL groupBy 操作。

```
source
    .groupBy("Region, Division", "Region, Division, COUNT(*) AS numStates")
    .orderBy("Region, Division", 100)
    .lazyPrint(-1);
```

运行结果如下。

Region	Division	numStates
North Central	East North Central	5
North Central	West North Central	7
Northeast	Middle Atlantic	3
Northeast	New England	6
South	East South Central	4
South	South Atlantic	8
South	West South Central	4
West	Mountain	8
West	Pacific	5

Division 的 9 个部分是在 Region 的 4 个部分基础上进行的，除了 Region 的 South 被分为 3 个部分，其他的 Region 都被分为了 2 个部分。

随后，使用经纬度聚类组件 GeoKMeans 分别针对 2 个和 4 个聚类的情况进行实验。具体代码如下。

```
for (int nClusters : new int[] {2, 4}) {
    BatchOperator <?> pred = new GeoKMeans()
        .setLongitudeCol("longitude")
        .setLatitudeCol("latitude")
        .setPredictionCol(PREDICTION_COL_NAME)
        .setK(nClusters)
        .fit(source)
        .transform(source);

    pred.link(
        new EvalClusterBatchOp()
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol("Region")
            .lazyPrintMetrics(nClusters + " with Region")
    );
    pred.link(
```

```

new EvalClusterBatchOp()
    .setPredictionCol(PREDICTION_COL_NAME)
    .setLabelCol("Division")
    .lazyPrintMetrics(nClusters + " with Division")
);
BatchOperator.execute();
}

```

经纬度聚类组件 GeoKMeans 的输入数据列参数的设置上与聚类组件不同，不是通过一个统一的 Vector 输入，而是需要指定具体的经度列 LongitudeCol 和纬度列 LatitudeCol。在评估时，只需输入聚类结果列 PredictionCol 和标签列 LabelCol，这里分别针对 Region 和 Division 进行了评估。

2 个聚类的评估结果对比如表 17-3 所示。

表 17-3 2 个聚类的评估结果对比

2 with Region			2 with Division		
ARI:0.2698	NMI:0.3849	Purity:0.54	ARI:0.1428	NMI:0.3647	Purity:0.32
Label\Cluster	1	0	Label\Cluster	1	0
----- --- ---	----- --- ---	----- --- ---	----- --- ---	----- --- ---	----- --- ---
West	0	13	West South Central	2	2
South	14	2	West North Central	3	4
Northeast	9	0	South Atlantic	8	0
North Central	8	4	Pacific	0	5
			New England	6	0
			Mountain	0	8
			Middle Atlantic	3	0
			East South Central	4	0
			East North Central	5	0

在左边 Region 的 4 个区域中，West 都属于聚类簇 0，Northeast 都属于聚类簇 1；South 和 North Central 大部分属于聚类簇 1，并分别有 2 个和 4 个州属于聚类簇 0。对应右边 Division 的情况，West South Central 和 West North Central 中分别有 2 个和 4 个州属于聚类簇 0。

4 个聚类的评估结果对比如表 17-4 所示。

表 17-4 4 个聚类的评估结果对比

4 with Region				4 with Division					
ARI:0.5149	NMI:0.6297	Purity:0.78		ARI:0.3221	NMI:0.5862	Purity:0.46			
Label\Cluster	3	2	1	0	Label\Cluster	3	2	1	0
----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----		
West	0	1	12	0	West South Central	4	0	0	0
South	11	0	0	5	West North Central	1	6	0	0
Northeast	0	0	0	9	South Atlantic	3	0	0	5
North Central	3	7	0	2	Pacific	0	0	5	0
					New England	0	0	0	6
					Mountain	0	1	7	0
					Middle Atlantic	0	0	0	3
					East South Central	4	0	0	0
					East North Central	2	1	0	2

在左边 Region 的 4 个区域中，Northeast 都属于聚类簇 0，而聚类簇 0 也是以 Northeast 为主的；聚类簇 1 都是以 West 的各州构成的；聚类簇 3 以 South 为主，聚类簇 2 以 North Central 为主。可以看到，聚类数为 4 的评估指标 ARI 都要优于聚类数为 2 的 ARI。

批式与流式聚类

第 17 章侧重于介绍聚类算法，使用的是小数据集。本章将使用手写数字数据集，模型训练时间较长，各个算法在计算时间上会有明显的差异，输入的向量是稠密格式还是稀疏格式对计算量也有显著的影响。本章还会介绍与流式数据相关的聚类功能，一个是通过批式训练的聚类模型预测流式数据，另一个是直接针对流式数据进行聚类模型训练和预测。

18.1 稠密向量与稀疏向量

第 13 章介绍了如何从原始数据中解析稠密向量数据和稀疏向量数据，并保存为 AK 格式数据文件。我们分别选取其稠密向量训练数据和稀疏向量训练数据作为聚类实验的稠密向量数据源与稀疏向量数据源，并定义一个停表对象，用来对各次试验进行计时。具体代码如下。

```
AkSourceBatchOp dense_source = new AkSourceBatchOp().setFilePath(DATA_DIR + DENSE_TRAIN_FILE);
AkSourceBatchOp sparse_source = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);
Stopwatch sw = new Stopwatch();
```

由于各个试验流程相似，而且都要分别使用稠密向量和稀疏向量进行试验，使用 Pipeline 会使代码更简洁。定义 Pipeline 和相关的标识字符串作为一个二元组，以及使用默认距离的 KMeans、余弦距离的 KMeans 和 BisectingKMeans。具体代码如下。

```
ArrayList <Tuple2 <String, Pipeline>> pipelineList = new ArrayList <>();
pipelineList.add(new Tuple2 <>("KMeans EUCLIDEAN",
    new Pipeline()
        .add(
            new KMeans()
```

```

.setK(10)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
)
));
pipelineList.add(new Tuple2 <>("KMeans COSINE",
new Pipeline()
.add(
new KMeans()
.setDistanceType(DistanceType.COSINE)
.setK(10)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
)
));
pipelineList.add(new Tuple2 <>("BisectingKMeans",
new Pipeline()
.add(
new BisectingKMeans()
.setK(10)
.setVectorCol(VECTOR_COL_NAME)
.setPredictionCol(PREDICTION_COL_NAME)
)
));

```

整体的代码如下。

```

For (Tuple2 <String, Pipeline> pipelineTuple2 : pipelineList) {
    sw.reset();
    sw.start();
    pipelineTuple2.f1
        .fit(dense_source)
        .transform(dense_source)
        .link(
            new EvalClusterBatchOp()
                .setVectorCol(VECTOR_COL_NAME)
                .setPredictionCol(PREDICTION_COL_NAME)
                .setLabelCol(LABEL_COL_NAME)
                .lazyPrintMetrics(pipelineTuple2.f0 + " DENSE")
        );
    BatchOperator.execute();
    sw.stop();
    System.out.println(sw.getElapsedTimeString());
}

sw.reset();
sw.start();
pipelineTuple2.f1
    .fit(sparse_source)
    .transform(sparse_source)
    .link(
        new EvalClusterBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
}

```

```

.setLabelCol(LABEL_COL_NAME)
.lazyPrintMetrics(pipelineTuple2.f0 + " SPARSE")
);
BatchOperator.execute();
sw.stop();
System.out.println(sw.getElapsed TimeSpan());
}
}

```

针对每个 Pipeline 及其标识字符串构成的二元组 pipelineTuple2，选择其 Pipeline 分量 pipelineTuple2.f1，先对稠密向量数据源进行训练、预测及评估，然后再对稀疏向量数据源进行训练、预测及评估。在输出评估结果信息时，加上标识字符串 pipelineTuple2.f0，便于我们查看结果。

运行结果汇总如表 18-1 所示。

表 18-1 稠密向量与稀疏向量聚类对比

距离	稠密向量	稀疏向量
KMeans (EUCLIDEAN)	VRC:2278.5669 DB:2.8836 SilhouetteCoefficient:0.0931 ARI:0.4048 NMI:0.5184 Purity:0.618	VRC:2236.2382 DB:2.911 SilhouetteCoefficient:0.1182 ARI:0.4063 NMI:0.5045 Purity:0.6121
	34 seconds 763.0 milliseconds	15 seconds 658.0 milliseconds
KMeans (COSINE)	VRC:2221.7892 DB:2.8565 SilhouetteCoefficient:0.0763 ARI:0.4064 NMI:0.5395 Purity:0.613	VRC:2248.4341 DB:2.8205 SilhouetteCoefficient:0.0815 ARI:0.3962 NMI:0.5188 Purity:0.6075
	24 seconds 567.0 milliseconds	12 seconds 75.0 milliseconds
BisectingKMeans	VRC:1984.6809 DB:3.2848 SilhouetteCoefficient:0.0836 ARI:0.226 NMI:0.3646 Purity:0.4626	VRC:1898.6387 DB:3.562 SilhouetteCoefficient:0.0736 ARI:0.2546 NMI:0.372 Purity:0.4601
	47 seconds 263.0 milliseconds	25 seconds 227.0 milliseconds

从表中能明显看到，使用稠密向量的计算时间是使用稀疏向量的 2 倍。

18.2 使用聚类模型预测流式数据

本节的内容与 18.3 节是有关联的。本节演示如何使用已有的聚类模型预测流式的数据，在

整个预测过程中，聚类模型是不变的；18.3 节会将已有的聚类模型作为初始模型，随着流式数据不断调整聚类模型，并基于新的聚类模型对数据进行聚类预测。

首先获取数据源，使用批式 AK 数据源组件 AkSourceBatchOp 和流式 AK 数据源组件 AkSourceStreamOp，分别以批和流的方式从 AK 格式数据文件中获取稀疏向量格式的数据源。

```
AkSourceBatchOp batch_source = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);
AkSourceStreamOp stream_source = new AkSourceStreamOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);
```

随后，我们构建一个初始模型。从批式数据源中采样 100 条样本，训练 K-Means 模型，并将模型保存到文件 INIT_MODEL_FILE 中。具体代码如下。

```
batch_source
.sampleWithSize(100)
.link(
    new KMeansTrainBatchOp()
        .setVectorCol(VECTOR_COL_NAME)
        .setK(10)
)
.link(
    new AkSinkBatchOp()
        .setFilePath(DATA_DIR + INIT_MODEL_FILE)
);
```

基于这个聚类模型，我们可以对批式数据进行聚类预测。新建一个批式数据源读取聚类模型，然后使用 K-Means 批式预测组件，接入聚类模型和待预测的批式数据，将预测结果与聚类评估组件连接。相关代码如下。

```
AkSourceBatchOp init_model = new AkSourceBatchOp().setFilePath(DATA_DIR + INIT_MODEL_FILE);

new KMeansPredictBatchOp()
.setPredictionCol(PREDICTION_COL_NAME)
.linkFrom(init_model, batch_source)
.link(
    new EvalClusterBatchOp()
        .setVectorCol(VECTOR_COL_NAME)
        .setPredictionCol(PREDICTION_COL_NAME)
        .setLabelCol(LABEL_COL_NAME)
        .lazyPrintMetrics("Batch Prediction")
);
```

评估结果如下。

```
----- Metrics: -----
k:10
VRC:1743.4614 DB:3.3561 SilhouetteCoefficient:0.0821
ARI:0.3064 NMI:0.407 Purity:0.5202
```

Label\Cluster	9	8	7	6	5	4	3	2	1	0
9	1294	1581	74	576	1	1689	322	187	7	218
8	34	64	378	1665	686	361	40	1749	48	826
7	4698	378	0	89	9	460	67	222	11	331
6	1	2	218	122	1	1138	3659	45	21	711
5	13	47	1480	1521	17	986	42	1094	7	214
4	86	1080	2	350	0	2931	1231	3	7	152
3	73	15	2346	2351	594	92	15	141	96	408
2	145	30	96	126	596	554	369	180	2730	1132
1	5	0	10	17	3	77	4	8	1	6617
0	17	222	402	600	28	354	403	3614	268	15

下面使用流式预测组件 KMeansPredictStreamOp 对流式数据进行预测，并把结果保存到数据文件中，便于后面使用批式聚类评估组件进行详细评估。将批式聚类模型数据源 init_model 作为 KMeansPredictStreamOp 构造函数的输入，预测结果写到 AK 格式数据文件中，如下代码所示。

```
stream_source
    .link(
        new KMeansPredictStreamOp(init_model)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
    .link(
        new AkSinkStreamOp()
            .setFilePath(DATA_DIR + TEMP_STREAM_FILE)
            .setOverwriteSink(true)
    );
StreamOperator.execute();
```

接下来，使用批式数据源打开刚才流式预测保存的数据文件，然后使用批式聚类评估组件进行详细评估，具体代码如下。

```
new AkSourceBatchOp()
    .setFilePath(DATA_DIR + TEMP_STREAM_FILE)
    .link(
        new EvalClusterBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .lazyPrintMetrics("Stream Prediction")
    );
```

评估结果如下。

```
----- Metrics: -----
k:10
VRC:1743.4614 DB:3.3561 SilhouetteCoefficient:0.0821
ARI:0.3064 NMI:0.407 Purity:0.5202
```

Label\Cluster	9	8	7	6	5	4	3	2	1	0
9	1294	1581	74	576	1	1689	322	187	7	218
8	34	64	378	1665	686	361	40	1749	48	826
7	4698	378	0	89	9	460	67	222	11	331
6	1	2	218	122	1	1138	3659	45	21	711
5	13	47	1480	1521	17	986	42	1094	7	214
4	86	1080	2	350	0	2931	1231	3	7	152
3	73	15	2346	2351	594	92	15	141	96	408
2	145	30	96	126	596	554	369	180	2730	1132
1	5	0	10	17	3	77	4	8	1	6617
0	17	222	402	600	28	354	403	3614	268	15

与批式聚类预测结果的评估结果完全一致。

18.3 流式聚类

Alink 的流式聚类组件需要输入初始聚类模型，这样可以保证最初的流式数据也可以有比较好的预测结果。注意，流式聚类个数已经由初始模型决定了。流式聚类以一个时间窗口为周期，每个周期产生一个新的聚类模型，并用此新模型预测后续数据。随着新数据的不断流入，我们用来训练聚类模型的数据也有一个逐步退出的过程。这里使用半衰期（HalfLife）的概念，含义是经过几个周期后，数据量减半。

先获取流式数据源 stream_source 和初始模型 init_model，然后由初始模型 init_model 构建流式聚类组件 StreamingKMeansStreamOp，并将流式数据源 stream_source 连接到流式聚类组件。流式聚类组件需要设置时间窗口间隔参数 TimeInterval=1，设置半衰期参数 HalfLife=1，随后使用 SQL Select 方法选择数据，以便于查看。对于预测结果数据，一方面，我们使用采样方式输出部分结果，便于看到计算的过程和结果；另一方面，将预测结果导出到文件中，便于后续分析。具体代码如下。

```

AkSourceStreamOp stream_source
= new AkSourceStreamOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);

AkSourceBatchOp init_model
= new AkSourceBatchOp().setFilePath(DATA_DIR + INIT_MODEL_FILE);

StreamOperator<?> stream_pred = stream_source
.link(
  new StreamingKMeansStreamOp(init_model)
    .setTimeInterval(1L)
    .setHalfLife(1)
  )

```

```

        .setPredictionCol(PREDICTION_COL_NAME)
    )
.select(PREDICTION_COL_NAME + ", " + LABEL_COL_NAME +", " + VECTOR_COL_NAME);

stream_pred.sample(0.001).print();

stream_pred
.link(
    new AkSinkStreamOp()
    .setFilePath(DATA_DIR + TEMP_STREAM_FILE)
    .setOverwriteSink(true)
);
StreamOperator.execute();

```

在计算过程中不断输出聚类预测数据，整体的预测结果也被保存到了文件中。我们再以批式数据的方式打开此文件，就可以使用上批式聚类评估组件了，具体代码如下。

```

new AkSourceBatchOp()
.setFilePath(DATA_DIR + TEMP_STREAM_FILE)
.link(
    new EvalClusterBatchOp()
    .setVectorCol(VECTOR_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .setLabelCol(LABEL_COL_NAME)
    .lazyPrintMetrics("StreamingKMeans")
);
BatchOperator.execute();

```

输出的结果如下。

Metrics:										
k:10										
VRC:1970.4495	DB:3.1637	SilhouetteCoefficient:0.1009								
ARI:0.3549	NMI:0.4549	Purity:0.5651								
Label\Cluster	9	8	7	6	5	4	3	2	1	0
9	1624	1869	36	312	13	1746	87	56	9	197
8	87	93	314	1263	1440	705	40	983	59	867
7	4738	573	1	44	13	408	17	45	33	393
6	1	14	227	52	5	884	4078	70	35	552
5	63	87	1334	1602	21	1160	50	771	7	326
4	221	2015	1	110	1	2869	478	3	8	136
3	92	20	2373	2213	767	98	29	58	114	367
2	113	27	84	97	868	379	272	87	3135	896
1	8	0	12	18	13	50	5	1	8	6627
0	4	57	288	546	33	212	290	4419	66	8

对比前面直接使用初始模型预测的聚类评估结果，可以发现，各项指标都得到了提升，这也说明了流式聚类算法的有效性。



主成分分析

主成分分析 (Principal Component Analysis, PCA) 是一种多元统计分析方法，将多个变量通过线性变换组合为少数几个新的变量，使它们尽可能多地保留原始变量的信息，且彼此互不相关。

对于一个变量，不同个体间的差异越大，变量包含的信息量越大，我们将首选“差异大”的那些新变量作为主成分。借助主成分分析得到的新变量要能概括诸多信息的主要方面，我们也希望新变量之间能够互相独立，每个新变量能独立代表某一方面的性质。

设 m 个变量分别为 $X^{(1)}, \dots, X^{(m)}$ ，由 n 个样本形成的数据为

$$\mathbf{X} = \begin{pmatrix} X_1^{(1)} & \cdots & X_1^{(m)} \\ \vdots & \ddots & \vdots \\ X_n^{(1)} & \cdots & X_n^{(m)} \end{pmatrix}$$

设单位正交矩阵为

$$\mathbf{A} = \begin{pmatrix} A_1^{(1)} & \cdots & A_1^{(m)} \\ \vdots & \ddots & \vdots \\ A_m^{(1)} & \cdots & A_m^{(m)} \end{pmatrix}$$

并设

$$\mathbf{A}^{(i)} = \begin{pmatrix} A_1^{(i)} \\ \vdots \\ A_m^{(i)} \end{pmatrix}$$

则经过线性变换得到新变量 $Z^{(1)}, \dots, Z^{(m)}$ ，并满足

$$\begin{pmatrix} Z_1^{(1)} & \cdots & Z_1^{(m)} \\ \vdots & \ddots & \vdots \\ Z_n^{(1)} & \cdots & Z_n^{(m)} \end{pmatrix} = \begin{pmatrix} X_1^{(1)} & \cdots & X_1^{(m)} \\ \vdots & \ddots & \vdots \\ X_n^{(1)} & \cdots & X_n^{(m)} \end{pmatrix} \begin{pmatrix} A_1^{(1)} & \cdots & A_1^{(m)} \\ \vdots & \ddots & \vdots \\ A_m^{(1)} & \cdots & A_m^{(m)} \end{pmatrix}$$

则 $Z^{(i)}$ 与 $Z^{(j)}$ 的协方差为

$$\begin{aligned} \text{cov}(Z^{(i)}, Z^{(j)}) &= \frac{1}{n-1} (Z^{(i)} - \bar{Z}^{(i)})(Z^{(j)} - \bar{Z}^{(j)}) \\ &= \frac{1}{n-1} (XA^{(i)} - \bar{XA}^{(i)})^T (XA^{(j)} - \bar{XA}^{(j)}) \\ &= \frac{1}{n-1} A^{(i)T} (X - \bar{X})^T (X - \bar{X}) A^{(j)} \\ &= A^{(i)T} \left[\frac{1}{n-1} (X - \bar{X})^T (X - \bar{X}) \right] A^{(j)} \end{aligned}$$

即

$$\text{cov}(Z^{(i)}, Z^{(j)}) = A^{(i)T} \text{cov}(X) A^{(j)}$$

当 $i = j$ 时， $Z^{(i)}$ 的方差为

$$\text{var}(Z^{(i)}) = A^{(i)T} \text{cov}(X) A^{(i)}$$

另外，由于 $\text{cov}(X)$ 是对称的非负定矩阵，对其进行特征值分解，可得

$$\text{cov}(X)V = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_m \end{pmatrix} V$$

其中， $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_m$, $V^T V = I$ 。

令 $A^{(i)} = V^{(i)}$ ，则

$$\text{var}(Z^{(i)}) = \lambda_i$$

方差满足如下大小关系

$$\text{var}(Z^{(1)}) \geq \text{var}(Z^{(2)}) \geq \dots \geq \text{var}(Z^{(m)})$$

方差的和为

$$\sum_{i=1}^m \text{var}(Z^{(i)}) = \sum_{i=1}^m \lambda_i$$

且协方差满足

$$\text{cov}(Z^{(i)}, Z^{(j)}) = 0$$

综合前面的分析过程， $X^{(1)}, \dots, X^{(m)}$ 通过单位正交的矩阵 A 在高维空间进行了旋转变换，得到 $Z^{(1)}, \dots, Z^{(m)}$ ，满足 $\text{var}(Z^{(1)}) \geq \text{var}(Z^{(2)}) \geq \dots \geq \text{var}(Z^{(m)})$ 。显然，索引号较小的 $Z^{(i)}$ 更重要，可以看作数据的“主成分”。

$\lambda_1, \lambda_2, \dots, \lambda_m$ 是由矩阵进行特征值分解得到的，由矩阵的迹（trace）的定义和性质，可知

$$\sum_{i=1}^m \lambda_i = \text{trace}(\text{cov}(X)) = \sum_{i=1}^m \text{var}(X^{(i)})$$

$\sum_{i=1}^m \lambda_i$ 可以看作矩阵的总能量， λ_i 可以看作每个特征向量所对应的能量，分解前后的总能量相等。从 λ_1 开始，依次加入 $\lambda_2, \dots, \lambda_m$ ，当加入到第 p 个时，若这 p 个成分的能量和 $\sum_{i=1}^p \lambda_i$ 与总能量 $\sum_{i=1}^m \lambda_i$ 的比值大于 w （默认值为 0.9），即

$$\frac{\sum_{i=1}^p \lambda_i}{\sum_{i=1}^m \lambda_i} > w$$

我们就可以确定主成分的个数为 p 。

在实际使用中，有时会先将各个变量进行标准化，此时的协方差矩阵就相当于原始数据的相关系数矩阵。所以 Alink 的主成分分析组件提供了两种计算选择，参数 CalculationType 可以设置为相关系数矩阵（CORR）或者协方差矩阵（COV），默认为相关系数矩阵，即对标准化后的数据计算其主成分。

19.1 主成分的含义

本节的案例是调查美国 50 个州的 7 种犯罪率。数据是美国 50 个州每 10 万人中 7 种犯罪的概率数据。这 7 种犯罪分别是 murder（杀人罪）、rape（强奸罪）、robbery（抢劫罪）、assault（斗殴罪）、burglary（夜盗罪）、larceny（偷盗罪）和 auto（汽车犯罪）。下面我们来做主成

分分析，直接从这 7 个变量出发来评价各州的治安和犯罪情况是很难的，而使用主成分分析则可以把这些变量概括为 2 个或 3 个综合变量（即主成分），以便帮助我们更简便地分析这些数据。

数据只有 50 条，将其写在代码中，定义为数组 CRIME_ROWS_DATA，数据源组件 MemSourceBatchOp 会在构造函数中使用该数组，如下代码所示。

```
private static final Row[] CRIME_ROWS_DATA = new Row[] {
    Row.of("ALABAMA", 14.2, 25.2, 96.8, 278.3, 1135.5, 1881.9, 280.7),
    Row.of("ALASKA", 10.8, 51.6, 96.8, 284.0, 1331.7, 3369.8, 753.3),
    Row.of("ARIZONA", 9.5, 34.2, 138.2, 312.3, 2346.1, 4467.4, 439.5),
    ...
    Row.of("WEST VIRGINIA", 6.0, 13.2, 42.2, 90.9, 597.4, 1341.7, 163.3),
    Row.of("WISCONSIN", 2.8, 12.9, 52.2, 63.7, 846.9, 2614.2, 220.7),
    Row.of("WYOMING", 5.4, 21.9, 39.7, 173.9, 811.6, 2772.2, 282.0)
};

static final String[] CRIME_COL_NAMES =
    new String[] {"state", "murder", "rape", "robbery", "assault", "burglary", "larceny", "auto"};
...
MemSourceBatchOp source = new MemSourceBatchOp(CRIME_ROWS_DATA, CRIME_COL_NAMES);

source.lazyPrint(10, "Origin data");
```

输出部分原始数据如表 19-1 所示。

表 19-1 原始数据

state	murder	rape	robbery	assault	burglary	larceny	auto
ALABAMA	14.2000	25.2000	96.8000	278.3000	1135.5000	1881.9000	280.7000
ALASKA	10.8000	51.6000	96.8000	284.0000	1331.7000	3369.8000	753.3000
ARIZONA	9.5000	34.2000	138.2000	312.3000	2346.1000	4467.4000	439.5000
ARKANSAS	8.8000	27.6000	83.2000	203.4000	972.6000	1862.1000	183.4000
CALIFORNIA	11.5000	49.4000	287.0000	358.0000	2139.4000	3499.8000	663.5000
COLORADO	6.3000	42.0000	170.7000	292.9000	1935.2000	3903.2000	477.1000
CONNECTICUT	4.2000	16.8000	129.5000	131.8000	1346.0000	2620.7000	593.2000
DELAWARE	6.0000	24.9000	157.0000	194.2000	1682.6000	3678.4000	467.0000
FLORIDA	10.2000	39.6000	187.9000	449.1000	1859.9000	3840.5000	351.4000
GEORGIA	11.7000	31.1000	140.5000	256.5000	1351.1000	2170.2000	297.9000

主成分分析组件为 PCA，设置主成分的个数为 $K=4$ ；设置参与计算的数据列为除了州名称

的其他列；主成分分量的输出形式为向量，设置输出列的名称为 VECTOR_COL_NAME；设置使用 Lazy 的方式输出模型信息。为了方便后面的分析，使用格式转换组件将向量列转化为 4 个数据列，名称分别为“pc_1、pc_2、pc_3、pc_4”，数据类型都为 double 类型。计算主成分的代码如下。

```
BatchOperator <?> pca_result = new PCA()
    .setK(4)
    .setSelectedCols("murder", "rape", "robbery", "assault", "burglary", "larceny", "auto")
    .setPredictionCol(VECTOR_COL_NAME)
    .enableLazyPrintModelInfo()
    .fit(source)
    .transform(source)
    .link(
        new VectorToColumnsBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setSchemaStr("pc_1 double, pc_2 double, pc_3 double, pc_4 double")
            .setReservedCols("state")
    )
    .lazyPrint(10, "state with principle components");
```

我们可以得到如下模型信息。

PCA

CalculationType: CORR

Number of Principal Component: 4

EigenValues:

Prin	Eigenvalue	Proportion	Cumulative
Prin1	4.115	0.5879	0.5879
Prin2	1.2387	0.177	0.7648
Prin3	0.7258	0.1037	0.8685
Prin4	0.3164	0.0452	0.9137

EigenVectors:

colName	Prin1	Prin2	Prin3	Prin4
murder	-0.3003	-0.6292	-0.1782	0.2321
rape	-0.4318	-0.1694	0.2442	-0.0622
robbery	-0.3969	0.0422	-0.4959	0.558
assault	-0.3967	-0.3435	0.0695	-0.6298
burglary	-0.4402	0.2033	0.2099	0.0576
larceny	-0.3574	0.4023	0.5392	0.2349
auto	-0.2952	0.5024	-0.5684	-0.4192

由 EigenValues 可知各主成分的贡献率。在结果表中，第一列为主成分所对应的奇异值大小，第二列为各主成分的贡献率。可以看出，第一个主成分的贡献率非常突出，达到了 58.8%；最

后一列为累计贡献率，可以看到前两个主成分的累计贡献率达 76.5%，前三个主成分的累计贡献率达 86.8%，前四个主成分的累计贡献率达 91.4%。在分析数据时，我们可以根据实际需要考虑选取几个主成分。

由 EigenVectors 可知，主成分的计算表达式如下。

```
Prin1 = -0.3003 * murder - 0.4318 * rape - 0.3969 * robbery - 0.3967 * assault - 0.4402 * burglary - 0.3574
* larceny - 0.2952 * auto
Prin2 = -0.6292 * murder - 0.1694 * rape + 0.0422 * robbery - 0.3435 * assault + 0.2033 * burglary + 0.4023
* larceny + 0.5024 * auto
```

注意：这里的 murder、rape 等变量是原变量标准化后的变量，即原变量减去均值，再除以标准差得到的变量。第一主成分 Prin1 对所有的变量都有近似相等的权重，可认为第一主成分是对所有犯罪率的总度量取相反数；第二主成分 Prin2 可以看作抢、盗罪（robbery、burglary、larceny 和 auto 权重为正）与杀、淫罪（murder、rape 和 assault 权重为负）的对比，Prin2 值较小的州的暴力犯罪比重较大。

输出的部分 PCA 结果如表 19-2 所示。

表 19-2 主成分分析结果

state	prin1	prin2	prin3	prin4
ALABAMA	0.0499	-2.0961	-0.5016	-0.2510
ALASKA	-2.4215	0.1665	0.0697	-1.1605
ARIZONA	-3.0141	0.8449	1.7520	0.1162
ARKANSAS	1.0544	-1.3454	0.0183	-0.0215
CALIFORNIA	-4.2838	0.1432	-0.2762	-0.0251
COLORADO	-2.5093	0.9166	1.1516	-0.1126
CONNECTICUT	0.5413	1.5012	-0.7839	-0.0862
DELAWARE	-0.9646	1.2967	0.5259	0.4173
FLORIDA	-3.1118	-0.6039	1.2154	-0.4951
GEORGIA	-0.4904	-1.3808	-0.2446	0.0625

除了关注每个州的主成分分量，我们也希望了解每个分量下最大值和最小值对应的是哪个州？下面分别对 Prin1 和 Prin2 的数值进行排序，具体代码如下。需要注意的是；orderBy 函数的第二参数为输出排序结果的前多少项，当数据量很大时，可以根据这个参数大幅减少计算量。

避免全量排序；`orderBy` 函数的第三参数为是否升序，这里设置为 `false`，则会按降序排序，输出前 100 个数据，因为我们整体的数据量为 50，所以会排序输出全部数据。

```
pca_result
  .select("state, prin1")
  .orderBy("prin1", 100, false)
  .lazyPrint(-1, "Order by prin1");

pca_result
  .select("state, prin2")
  .orderBy("prin2", 100, false)
  .lazyPrint(-1, "Order by prin2");
```

运行结果汇总在表 19-3 中，左边为按第一主成分值排序的结果，右边为按第二主成分值排序的结果。这里只列出了排名在前 3 名和后 3 名的各州。

表 19-3 排名在前 3 名和后 3 名的各州

Order by prin1	Order by prin2
state prin1	state prin2
----- -----	----- -----
NORTH DAKOTA 3. 9641	MASSACHUSETTS 2. 6311
SOUTH DAKOTA 3. 1720	RHODE ISLAND 2. 1466
WEST VIRGINIA 3. 1477	HAWAII 1. 8239
...
NEW YORK -3. 4525	ALABAMA -2. 0961
CALIFORNIA -4. 2838	SOUTH CAROLINA -2. 1621
NEVADA -5. 2670	MISSISSIPPI -2. 5467

第一主成分 `Prin1` 被看作所有犯罪率的总度量的相反数，`Prin1` 值较大的州的犯罪率较低，其中，`NORTH DAKOTA` 州的犯罪率最低（`Prin1 = 3.9641`）；`Prin1` 值较小的州的犯罪率较高，`NEVADA` 州的犯罪率最高（`Prin1 = -5.2670`）。第二主成分 `Prin2` 值较小的州的暴力犯罪性质比重较大，`MISSISSIPPI` 州的暴力犯罪性质比重最大（`Prin2 = -2.5467`）；`Prin2` 值较大的州的暴力犯罪性质比重较小，`MASSACHUSETTS` 州的暴力犯罪性质比重最小（`Prin2 = 2.6311`）。

19.2 两种计算方式

Alink 的主成分分析组件提供了两种计算方式，参数 `CalculationType` 可以设置为相关系数

矩阵或者协方差矩阵，默認為相关系数矩阵，即计算标准化后的数据的主成分。

构建实验流程如下。

```

Pipeline std_pca = new Pipeline()
    .add(
        new StandardScaler()
            .setSelectedCols("murder", "rape", "robbery", "assault", "burglary", "larceny", "auto")
    )
    .add(
        new PCA()
            .setCalculationType(CalculationType.COV)
            .setK(4)
            .setSelectedCols("murder", "rape", "robbery", "assault", "burglary", "larceny", "auto")
            .setPredictionCol(VECTOR_COL_NAME)
            .enableLazyPrintModelInfo()
    );
}

std_pca
    .fit(source)
    .transform(source)
    .link(
        new VectorToColumnsBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setSchemaStr("prin1 double, prin2 double, prin3 double, prin4 double")
            .setReservedCols("state")
    )
    .lazyPrint(10, "state with principle components");
BatchOperator.execute();

```

在 Pipeline 中先使用标准化变换组件 StandardScaler 对各数据列进行标准化，然后调用主成分组件 PCA。注意：这里特别设置了参数 CalculationType=COV。

首先看 PCA 阶段输出的模型信息，对比 19.1 节的模型信息，除了“CalculationType: COV”与 19.1 节模型信息中的“CalculationType: CORR”不同，其他数据值都一样。

PCA

CalculationType: COV

Number of Principal Component: 4

EigenValues:

Prin	Eigenvalue	Proportion	Cumulative
Prin1	4.115	0.5879	0.5879
Prin2	1.2387	0.177	0.7648
Prin3	0.7258	0.1037	0.8685
Prin4	0.3164	0.0452	0.9137

EigenVectors:

colName	Prin1	Prin2	Prin3	Prin4
murder	-0.3003	-0.6292	-0.1782	0.2321
rape	-0.4318	-0.1694	0.2442	-0.0622
robbery	-0.3969	0.0422	-0.4959	0.558
assault	-0.3967	-0.3435	0.0695	-0.6298
burglary	-0.4402	0.2033	0.2099	0.0576
larceny	-0.3574	0.4023	0.5392	0.2349
auto	-0.2952	0.5024	-0.5684	-0.4192

接下来看 PCA 输出的部分计算结果，如表 19-4 所示，也与 19.1 节 PCA 输出的结果一样。

表 19-4 新的 PCA 输出结果

state	prin1	prin2	prin3	prin4
ALABAMA	0.0499	-2.0961	-0.5016	-0.2510
ALASKA	-2.4215	0.1665	0.0697	-1.1605
ARIZONA	-3.0141	0.8449	1.7520	0.1162
ARKANSAS	1.0544	-1.3454	0.0183	-0.0215
CALIFORNIA	-4.2838	0.1432	-0.2762	-0.0251
COLORADO	-2.5093	0.9166	1.1516	-0.1126
CONNECTICUT	0.5413	1.5012	-0.7839	-0.0862
DELAWARE	-0.9646	1.2967	0.5259	0.4173
FLORIDA	-3.1118	-0.6039	1.2154	-0.4951
GEORGIA	-0.4904	-1.3808	-0.2446	0.0625

综合上面的PCA模型信息和输出结果，证实了两种计算方式间的差距只在于数据标准化变换。选择相关系数矩阵（CORR）的计算方式，通过标准化变换，统一了各数据列的尺度；而选择协方差矩阵（COV）的计算方式，输出的主成分实质上是原数据的降维数据，是原始数据在高维空间进行了旋转变换后¹提取了几个重要维度而形成的数据。

19.3 在聚类方面的应用

本节及 19.4 节将使用手写数字数据集来演示如何通过 PCA 获取降维数据，以便更高效地

¹ 注意：变换保持了各点间的距离。

处理一些聚类和分类问题。

我们将只使用原始数据 5% 的维度，即 $784 \times 5\% \approx 39$ （维）。使用稀疏格式的训练数据作为数据源，训练出来的 PCA 模型会被多次用到，并会将模型保存到 AK 格式文件（本地路径为 DATA_DIR + PCA_MODEL_FILE）中。具体代码如下。

```
AkSourceBatchOp source = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);

source
    .link(
        new PcaTrainBatchOp()
            .setK(39)
            .setCalculationType(CalculationType.COV)
            .setVectorCol(VECTOR_COL_NAME)
            .lazyPrintModelInfo()
    )
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + PCA_MODEL_FILE)
            .setOverwriteSink(true)
    );
BatchOperator.execute();
```

运行结束后，生成了 PCA 模型文件，同时输出了如下模型信息，这里略去了后面的 EigenVectors 部分。

PCA			
CalculationType: COV			
Number of Principal Component: 39			
EigenValues:			
	Prin	Eigenvalue	Proportion
			Cumulative
----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----	----- ----- ----- -----
Prin1 332724.6674 0.097 0.097			
Prin2 243283.9391 0.071 0.168			
Prin3 211507.3671 0.0617 0.2297			
Prin4 184776.3859 0.0539 0.2836			
Prin5 166926.8313 0.0487 0.3323			
Prin6 147844.9617 0.0431 0.3754			
Prin7 112178.2027 0.0327 0.4081			
Prin8 98874.4296 0.0288 0.437			
Prin9 94696.2491 0.0276 0.4646			
Prin10 80809.8245 0.0236 0.4881			
... 			
Prin30 23686.123 0.0069 0.7305			
Prin31 22562.7619 0.0066 0.7371			
Prin32 22221.7664 0.0065 0.7436			
Prin33 20660.6718 0.006 0.7496			
Prin34 20110.9854 0.0059 0.7555			

Prin35	19543. 2009	0. 0057	0. 7612
Prin36	18638. 2921	0. 0054	0. 7666
Prin37	17340. 9003	0. 0051	0. 7717
Prin38	16726. 2448	0. 0049	0. 7766
Prin39	16505. 8174	0. 0048	0. 7814

可以看到，前 39 个主成分占了总能量的 78%。

载入 PCA 模型，并对原始数据进行 PCA 变换，结果为 pca_result。

```
BatchOperator <?> pca_result = new PcaPredictBatchOp()
    .setVectorCol(VECTOR_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME)
    .linkFrom(
        new AkSourceBatchOp().setFilePath(DATA_DIR + PCA_MODEL_FILE),
        source
    );
```

接下来，针对原始数据及 PCA 结果数据 pca_result 分别进行 K-Means 聚类操作，并对比聚类评估结果，具体代码如下。

```
KMeans kmeans = new KMeans()
    .setK(10)
    .setVectorCol(VECTOR_COL_NAME)
    .setPredictionCol(PREDICTION_COL_NAME);

sw.reset();
sw.start();
kmeans
    .fit(source)
    .transform(source)
    .link(
        new EvalClusterBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .lazyPrintMetrics("KMeans")
    );
BatchOperator.execute();
sw.stop();
System.out.println(sw.getElapsedTimeSpan());

sw.reset();
sw.start();
kmeans
```

```

    .fit(pca_result)
    .transform(pca_result)
    .link(
        new EvalClusterBatchOp()
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .lazyPrintMetrics("KMeans + PCA")
    );
    BatchOperator.execute();
    sw.stop();
    System.out.println(sw.getElapsed TimeSpan());
}

```

运行结果如下。

KMeans

Metrics:

k:10
 VRC:2293.0127 DB:2.8759 SilhouetteCoefficient:0.0925
 ARI:0.3661 NMI:0.4848 Purity:0.5818

Label\Cluster	9	8	7	6	5	4	3	2	1	0
9	9	1478	145	32	9	1713	40	2385	87	51
8	49	296	314	28	53	157	3171	171	1319	293
7	29	2431	223	13	3	701	3	2674	4	184
6	132	3	312	68	4866	210	159	0	32	136
5	17	612	224	64	132	248	1778	216	1739	391
4	25	1559	119	7	115	2349	10	1515	0	143
3	221	41	425	32	62	100	1263	121	3817	49
2	4208	68	357	58	181	204	144	39	371	328
1	9	5	3697	0	8	4	8	10	5	2996
0	47	61	6	4701	217	53	568	12	248	10

17 seconds 7.0 milliseconds.

KMeans + PCA

Metrics:

k:10
 VRC:3205.3665 DB:2.4339 SilhouetteCoefficient:0.146
 ARI:0.3992 NMI:0.5131 Purity:0.6146

Label\Cluster	9	8	7	6	5	4	3	2	1	0
9	86	52	100	2743	223	175	2498	28	35	9
8	1265	3202	363	312	340	21	214	55	29	50
7	5	7	306	320	313	4129	1127	40	15	3
6	31	92	165	12	247	0	135	571	60	4605
5	1762	1648	625	628	177	15	330	13	66	157

```

| 4|   0|   7| 152|2713| 127|   5|2660|   6|   6| 106|
| 3|3835|1275|  83|   31| 442|   40| 163| 161|   31|   70|
| 2| 356| 181| 347|   28| 338|   96| 148|4260|   64| 140|
| 1|   5|   6|3023|   5|3672|   5|   6| 10|   0|   10|
| 0| 243| 496|  26|   29|   4|   43|   53|4749| 276|

```

9 seconds 627.0 milliseconds.

二者在计算时间上有明显差异，提取的主成分在向量维度上要远小于原始向量，所以在计算量上有优势；计算时间相差一倍，这是因为我们选择的原始数据为稀疏向量格式，其计算量也远小于同等维度的稠密向量。从聚类指标上看，基于主成分聚类并不等价于基于原始数据的聚类，而是各有千秋。

19.4 在分类方面的应用

提取特征向量的主成分可以显著减少向量长度，从而大幅减少计算量，缩短分类模型的训练时间。

我们以手写数字分类为例，看看不同的训练数据格式（784 维稠密向量、784 维稀疏向量、39 维主成分向量）对模型训练时间的影响。首先获取稠密向量和稀疏向量的数据源，并定义一个停表对象，用来对各次试验进行计时。具体代码如下。

```

AkSourceBatchOp dense_train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + DENSE_TRAIN_FILE);
AkSourceBatchOp dense_test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + DEMSE_TEST_FILE);
AkSourceBatchOp sparse_train_data = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TRAIN_FILE);
AkSourceBatchOp sparse_test_data = new AkSourceBatchOp().setFilePath(DATA_DIR + SPARSE_TEST_FILE);

Stopwatch sw = new Stopwatch();

```

设置 K 最近邻分类器 KnnClassifier，并分别对稠密向量数据和稀疏向量数据进行训练、预测，并评估分类效果。左边使用稠密向量，右边使用稀疏向量，具体代码如下。

<pre> new KnnClassifier() .setK(3) .setVectorCol(VECTOR_COL_NAME) .setLabelCol(LABEL_COL_NAME) .setPredictionCol(PREDICTION_COL_NAME) .fit(dense_train_data) .transform(dense_test_data) .link(new EvalMultiClassBatchOp() .setLabelCol(LABEL_COL_NAME) .setPredictionCol(PREDICTION_COL_NAME) .lazyPrintMetrics("KnnClassifier Dense")); </pre>	<pre> new KnnClassifier() .setK(3) .setVectorCol(VECTOR_COL_NAME) .setLabelCol(LABEL_COL_NAME) .setPredictionCol(PREDICTION_COL_NAME) .fit(sparse_train_data) .transform(sparse_test_data) .link(new EvalMultiClassBatchOp() .setLabelCol(LABEL_COL_NAME) .setPredictionCol(PREDICTION_COL_NAME) .lazyPrintMetrics("KnnClassifier Sparse")); </pre>
--	---

左边使用稠密向量，右边使用稀疏向量，分类效果对比显示如下。

KnnClassifier Dense

Metrics:

Accuracy:0.9719 Macro F1:0.9718

Micro F1:0.9719 Kappa:0.9688

Pred\Real	9	8	7	...	2	1	0
9	971	4	12	...	1	0	0
8	4	924	0	...	2	0	0
7	8	4	991	...	13	0	1
...
2	1	3	4	...	994	2	1
1	4	0	19	...	9	1133	1
0	4	7	0	...	10	0	974

5 minutes 17 seconds 179.0 milliseconds.

KnnClassifier Sparse

Metrics:

Accuracy:0.9719 Macro F1:0.9718

Micro F1:0.9719 Kappa:0.9688

Pred\Real	9	8	7	...	2	1	0
9	971	4	12	...	1	0	0
8	4	924	0	...	2	0	0
7	8	4	991	...	13	0	1
...
2	1	3	4	...	994	2	1
1	4	0	19	...	9	1133	1
0	4	7	0	...	10	0	974

1 minutes 1 seconds 528.0 milliseconds.

可见，稠密向量数据和稀疏向量数据在评估指标上是完全一致的，但计算时间的比例大约是 5:1。

接下来，我们将主成分分析和 K 最近邻分类器组合为 Pipeline，再分别对稠密向量数据和稀疏向量数据进行训练、预测，并评估分类效果，左边使用稠密向量，右边使用稀疏向量，具体代码如下。

```
new Pipeline()
    .add(
        new PCA()
            .setK(39)
            .setCalculationType(CalculationType.COV)
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(VECTOR_COL_NAME)
    )
    .add(
        new KnnClassifier()
            .setK(3)
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
    .fit(dense_train_data)
    .transform(dense_test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("Knn with PCA Dense")
    );
};
```

```
new Pipeline()
    .add(
        new PCA()
            .setK(39)
            .setCalculationType(CalculationType.COV)
            .setVectorCol(VECTOR_COL_NAME)
            .setPredictionCol(VECTOR_COL_NAME)
    )
    .add(
        new KnnClassifier()
            .setK(3)
            .setVectorCol(VECTOR_COL_NAME)
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
    )
    .fit(sparse_train_data)
    .transform(sparse_test_data)
    .link(
        new EvalMultiClassBatchOp()
            .setLabelCol(LABEL_COL_NAME)
            .setPredictionCol(PREDICTION_COL_NAME)
            .lazyPrintMetrics("Knn with PCA Sparse")
    );
};
```

注意，在 PCA 中设置参数 CalculationType=COV。