第 3 章

面向 IoT 的机器学习

机器学习深刻地改变了生产制造商利用 IoT 所能做的事情。如今，众多的行业都提出了特定的 IoT 需求。例如，**医疗物联网**（**Internet of Medical Things，IoMT**）拥有可以居家佩戴的门诊心脏监护仪等设备，这些设备往往需要通过网络发送大量数据，或者在边缘端需要超大算力来处理与心脏有关的事件。另一个例子是**农业物联网**（**Agricultural IoT，AIoT**），其设备通常部署在没有 Wi-Fi 或移动网络的地方。指令或模型被推送至这些半连接的设备（semiconnected device）上。许多类似的设备都需要在边缘端直接进行决策。当使用 LoRAWAN 或 TV 等技术最终建立连接后，空余空间（white space）模型将会下载至这些设备端。

本章讨论使用机器学习模型（如 Logistic 回归和决策树）解决常见的 IoT 问题，如医疗诊断分类、危险驾驶行为检测以及化学数据分类等，研究应用于受限设备（constrained device）的技术，以及如何使用无监督学习理解如原型机等仅拥有少量数据的设备。

本章将涵盖以下实用案例：

- 采用异常检测分析化学传感器；
- IoMT 中的 Logistic 回归；
- 使用决策树对化学传感器进行分类；
- 使用 XGBoost 进行简单的预测性维护；
- 危险驾驶行为检测；
- 在受限设备端进行人脸检测。

3.1 采用异常检测分析化学传感器

精确的预测模型需要数量庞大且曾发生过故障的现场设备，以便有足够的故障数据用于预测。对于一些做工精良的工业设备来说，发生所需规模的故障可能需要耗费数年的时间。异常检测可以用来识别那些表现异常的设备，还可以用来筛查成千上万相似的信息，并精准地从中找出异常信息。

机器学习中的异常检测可以分为**无监督的**(**unsupervised**)、**有监督的**(**supervised**)和**半监督的**(**semisupervised**)。通常首先使用一个无监督的机器学习算法,将数据聚类分为行为模式或群体,这类满足特定条件的数据集合称为桶(bucket)。当需要对机器进行检查时,一些 bucket 用于识别行为,而另一些则用于识别设备出现的故障。设备可以表现出不同的行为模式,包括静止状态(resting state)、使用状态(in-use state)、冷状态(cold state)或其他需要进一步研究的状态。

本实用案例中,假定对使用的数据集里的数据所知甚少。异常检测作为发现问题过程中的一部分,经常与原型一起使用。

3.1.1 预备工作

异常检测是最容易实现的机器学习模型之一。本实用案例使用的数据来自检测酒类的化学传感器。作为预备工作,需要导入以下数据库:NumPy、Sklearn 和 Matplotlib。

3.1.2 操作步骤

要完成本实用案例,需要遵循以下步骤。

(1) 导入所需的库:

```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

(2) 上传数据文件到 DataFrame 中:

```
df = spark.read.format("csv" \
    .option("inferSchema", True) \
    .option("header", True) \
    .option("sep", "\t") \
    .load("/FileStore/tables/HT_Sensor_metadata.dat")
```

(3) 通过数据集查看数据的分组是否与聚类的数量相关:

```
pdf = df.toPandas()

y_pred = KMeans(n_clusters = 3,
                random_state = 2).fit_predict(pdf[['dt','t0']])

plt.scatter(pdf['t0'],pdf['dt'], c = y_pred)
display(plt.show())
```

输出如图 3-1 所示,给出了三组不同的数据,紧凑的聚类表示数据具有明确的边界。如果将聚类的数目调整为 10,将能够更好地分离不同的组。这些聚类片段可以帮助我们识别不同的数据段,也有助于确定传感器的最佳位置以及在机器学习模型中执行特征工程。

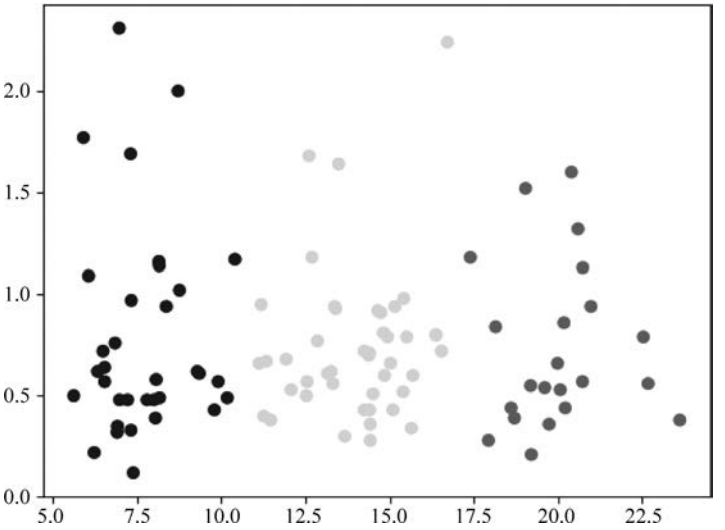

图 3-1　案例输出结果

3.1.3　工作机理

本实用案例使用 NumPy 处理数据，使用 Sklearn 作为机器学习算法，并使用 Matplotlib 查看结果。将制表符分割的文件导入 Spark 数据框架中，将数据转换为 Pandas DataFrame。然后在 3 个聚类(cluster)上运行 K-means 算法，这样就可以输出图表了。

K-means 是一种将数据分组并形成聚类的算法。它是一种流行的用于无标签数据的聚类算法。K-means 首先随机初始化聚类质心(cluster centroid)，本例中有 3 个聚类质心。然后，将聚类质心分配给相近的数据点。接下来，修正每个聚类质心到各自聚类的中间位置。重复上述步骤，直到对数据点进行了适当的划分。

3.1.4　补充说明

注意图 3-1 中的离群值(outlier)。对原型来说，这些值是非常值得关注的。离群值代表的可能是机器内部的功率波动、传感器放置不当或其他问题。下面的代码示例给出了对数据的简单的标准差计算，结果显示有 2 个值落在距离均值 3 个标准差之外：

```
from numpy import mean
from numpy import std

data_mean, data_std = mean(pdf['dt']), std(pdf['dt'])

cut_off = data_std * 3
lower, upper = data_mean - cut_off, data_mean + cut_off
```

```
outliers = [x for x in pdf['dt'] if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
print(outliers)
```

3.2 IoMT 中的 Logistic 回归

本实用案例讨论如何使用 Logistic 回归对乳房 X 光检查数据进行分类。近几年，IoMT 发展迅速，开发的许多设备可以供患者出院回家后佩戴，为居家医疗监测提供了一种解决方案，而在医院配套使用的设备则为医生提供了医学检测之外的补充反馈。许多情况下，机器学习算法能够发现医生可能忽略的疾病和问题，或给他们提供额外的建议。本实用案例将使用乳腺癌数据集判断一份乳房 X 光检查记录是恶性还是良性。

3.2.1 预备工作

所需的数据集和 Databricks notebooks 可以在 GitHub 资源库(repository)中找到。该数据集比较冗杂，有些甚至是高度相关的坏数据列。换句话说，有些传感器是重复的，存在未使用的列和无关的数据。为了便于阅读，GitHub 资源库中有两个 notebook：第一个执行所有的数据操作，并把数据放到一个数据表中；第二个进行机器学习。下面重点介绍关于数据操作的 notebook。本实用案例的最后将讨论另外两个 notebook 展示 MLflow 的例子。

本实用案例中还需要 MLflow 工作区，进入 Databricks 并创建工作区，以便在工作区记录结果。

3.2.2 操作步骤

本实用案例的操作步骤如下。

(1) 导入所需的库：

```
import pandas as pd

from sklearn import neighbors, metrics
from sklearn.metrics import roc_auc_score, classification_report,\
precision_recall_fscore_support,confusion_matrix,precision_score, \
roc_curve,precision_recall_fscore_support as score
from sklearn.model_selection import train_test_split

import statsmodels.api as sm
import statsmodels.formula.api as smf
```

（2）导入数据：

```
df = spark.sql("select * from BreastCancer")
pdf = df.toPandas()
```

（3）分割数据：

```
X = pdf
y = pdf['diagnosis']
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size = 0.3, random_state = 40)
```

（4）创建 formula：

```
cols = pdf.columns.drop('diagnosis')
formula = 'diagnosis ~ ' + ' + '.join(cols)
```

（5）训练模型：

```
model = smf.glm(formula = formula, data = X_train,
                family = sm.families.Binomial())
logistic_fit = model.fit()
```

（6）测试模型：

```
predictions = logistic_fit.predict(X_test)
predictions_nominal = [ "M" if x < 0.5 else "B" for x in \
                        predictions]
```

（7）评估模型：

```
print(classification_report(y_test, predictions_nominal, digits = 3))
```

输出给出了恶性肿瘤(M)和良性肿瘤(B)的 precision、recall 和 f1-score 的值，如图 3-2 所示。

（8）评估误差矩阵(confusion matrix)：

```
cfm = confusion_matrix(y_test, predictions_nominal)
precision,recall,fscore,support = score(y_test, predictions_nominal,
                                        average = 'macro')
print('Confusion Matrix: \n', cfm, '\n')
```

输出结果如图 3-3 所示。

	precision	recall	f1-score
B	0.957	0.957	0.957
M	0.907	0.907	0.907

图 3-2　评估模型输出

```
Confusion Matrix:
 [[112   5]
 [  5  49]]
```

图 3-3　评估误差矩阵

结果显示，在测试集的 171 条记录中，112 条为真阴性，49 条为真阳性，即在 171 条记录中，能够正确识别 161 条记录。其中 10 个预测是错误的：5 个误测为假阴性，5 个误测为假阳性。

3.2.3 工作机理

本实用案例使用了 Logistic 回归。Logistic 回归是一种既可用于传统统计又可用于机器学习的技术。由于其简洁但功能强大，许多数据科学家将 Logistic 回归作为首选模型，并将它作为基准(benchmark)。Logistic 回归是一个二元分类器，这意味着它的输出可以为 true 或 false。本例采用 Logistic 回归将肿瘤分类为良性或恶性。

首先，导入 Koalas 进行数据处理，导入 Sklearn 用于模型和分析。从数据表中导入数据，并将其放入一个 Pandas DataFrame 中。然后把数据分成测试集和训练集，并创建一个公式(formula)描述模型中使用的数据列。这样，就给出了该模型的 formula、训练数据集以及将要使用的算法。最终输出可以用于评估新数据模型。创建一个名为 predictions_nominal 的 DataFrame，可以用它与测试结果数据集进行比较。分类报告给出了 precision、recall 和 f1-score 的值。

(1) **precision**：预测的正向结果与预期的正向结果的数量之比。

(2) **recall**：预测的正向结果与总数之比。

(3) **f1-score**：precision 和 recall 的混合结果。

查看模型结果，并计算准确率，主要考查的因素如下。

(1) **True Negatives**：被模型预测为负向结果的负向测试集样本。

(2) **False Positives**：被模型预测为正向结果的负向测试集样本。

(3) **False Negatives**：被模型预测为负向结果的正向测试集样本。

(4) **True Positives**：被模型预测为正向结果的正向测试集样本。

3.2.4 补充说明

将结果记录在 MLflow 中，以便与其他算法进行比较。此外还将保存其他参数(如使用的主要 formula 和预测的项目等)：

```
import pickle
import mlflow

with mlflow.start_run():
    mlflow.set_experiment("/Shared/experiments/BreastCancer")
    mlflow.log_param("formula", formula)
    mlflow.log_param("family", "binomial")
    mlflow.log_metric("precision", precision)
    mlflow.log_metric("recall", recall)
```

```
    mlflow.log_metric("fscore", fscore)
    filename = 'finalized_model.sav'
    pickle.dump(model, open(filename, 'wb'))
    mlflow.log_artifact(filename)
```

3.3　使用决策树对化学传感器进行分类

本实用案例将使用**金属氧化物**(**Metal-Oxide**，**MOx**)传感器的化学数据确定空气中是否含有酒精。这种传感器通常用于确定空气中是否含有食物或化学微粒,化学传感器可以检测到对人体有毒的气体或仓库中的食物泄漏等。

3.3.1　操作步骤

本实用案例的操作步骤如下。

(1) 导入库：

```
import pandas as pd
import numpy as np

from sklearn import neighbors, metrics
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
```

(2) 导入数据：

```
df = spark.sql("select * from ChemicalSensor")
pdf = df.toPandas()
```

(3) 对数值进行编码：

```
label_encoder = LabelEncoder()
integer_encoded = \
    label_encoder.fit_transform(pdf['classification'])
onehot_encoder = OneHotEncoder(sparse = False)

integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
```

(4) 测试/训练分割的数据：

```
X = pdf[feature_cols]
y = onehot_encoded
```

```
X_train, X_test, y_train, y_test = \
train_test_split(X, y, test_size = 0.2, random_state = 5)
```

(5) 训练和预测：

```
clf = DecisionTreeClassifier()
clf = clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

(6) 评估准确率：

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("AUC:",roc_auc_score(y_test, y_pred))
```

3.3.2 工作机理

首先要导入本项目所需的库，将数据从 Spark 数据表中导入 Pandas DataFrame 中。One-hot 编码可以将分类值(本例中为 Wine 和 No Wine)转换为适用于机器学习算法的编码值。在步骤(4)中，将特征列和 One-hot 编码列进行分割，将它们分割成测试集和训练集。在步骤(5)中，创建了一个决策树分类器，使用 X_train 和 y_train 数据训练模型，然后使用 X_test 数据创建 y_prediction 数据集。换句话说，最后将根据数据集对 X_test 数据集的预测得到一组称为 y_pred 的预测。在步骤(6)中，评估模型的准确率和曲线下面积(**Area Under the Curve，AUC**)。

当数据较为复杂时可以使用决策树分类器来解决。因此，可以遵循 Yes/No 的逻辑规则来决定是否使用决策树，如图 3-4 所示。

机器学习算法可以训练决策树模型使用数字型数据(numeric data)，如图 3-5 所示。

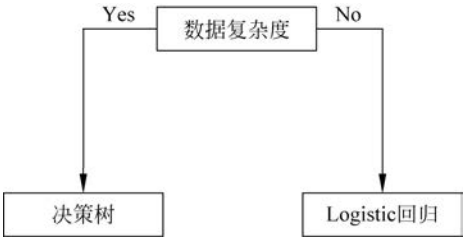

图 3-4 决策树

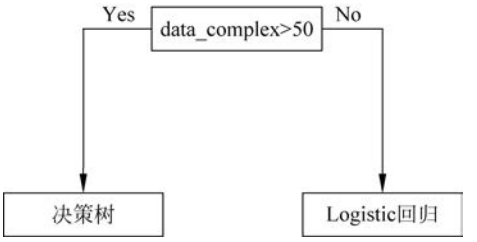

图 3-5 训练决策树模型使用数字型数据

在现有可用数据的情形下，机器学习算法可以训练模型准确地挑选出最佳的路径。

3.3.3 补充说明

Sklearn 决策树分类器有两个可以调整的超参数：**criterion** 和 **max depth**，通过调整超

参数可以观察准确率是否得到提高。超参数 criterion 可以是基尼系数(gini)或熵(entropy),这两个准则都可以评估子节点的不纯度(impurity)。另一个超参数 max depth 可能会导致过拟合(overfitting)或欠拟合(underfitting)。

欠拟合与过拟合

◇ 欠拟合的模型是不精确的,也不能很好地代表它们所训练的数据。

◇ 过拟合的模型无法对所训练的数据进行泛化,只对训练集中相同的数据进行拟合,而忽略了训练集中相似的数据。

3.4　使用 XGBoost 进行简单的预测性维护

每台设备都有使用寿命也需要经常性地进行维护。预测性维护是 IoT 中最常用的机器学习算法之一。第 4 章将深入探讨预测性维护,研究序列数据(sequential data)以及这些数据如何随季节性变化。本实用案例将从简单的分类角度探讨预测性维护。

本实用案例使用 NASA Turbofan engine degradation simulation 数据集,主要涉及 3 种分类:第一类是发动机不需要维护,一般用绿色表示;第二类是发动机在后续的 14 个维护周期内需要维护,一般用黄色表示;第三类是发动机需要维护,一般用红色表示。算法方面将使用 **extreme gradient boosting**,简称 **XGBoost**,XGBoost 近年来颇受欢迎,多次赢得了 Kaggle 竞赛。

3.4.1　预备工作

首先要准备 NASA Turbofan engine degradation simulation 数据集,这些数据和 Spark notebook 可以在本书配套的代码资源包或 NASA 网站上找到。另外,需要在 Databricks 中安装 XGBoost 库。

3.4.2　操作步骤

本实用案例的操作步骤如下。

(1) 导入所需的库:

```
import pandas as pd
import numpy as np
from pyspark.sql.types import *
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score
import pickle import mlflow
```

(2) 导入数据：

```
file_location = "/FileStore/tables/train_FD001.txt"
file_type = "csv"

schema = StructType([
                     StructField("engine_id", IntegerType()),
                     StructField("cycle", IntegerType()),
                     StructField("setting1", DoubleType()),
                     StructField("setting2", DoubleType()),
                     StructField("setting3", DoubleType()),
                     StructField("s1", DoubleType()),
                     StructField("s2", DoubleType()),
                     StructField("s3", DoubleType()),
                     StructField("s4", DoubleType()),
                     StructField("s5", DoubleType()),
                     StructField("s6", DoubleType()),
                     StructField("s7", DoubleType()),
                     StructField("s8", DoubleType()),
                     StructField("s9", DoubleType()),
                     StructField("s10", DoubleType()),
                     StructField("s11", DoubleType()),
                     StructField("s12", DoubleType()),
                     StructField("s13", DoubleType()),
                     StructField("s14", DoubleType()),
                     StructField("s15", DoubleType()),
                     StructField("s16", DoubleType()),
                     StructField("s17", IntegerType()),
                     StructField("s18", IntegerType()),
                     StructField("s19", DoubleType()),
                     StructField("s20", DoubleType()),
                     StructField("s21", DoubleType())
                    ])
df = spark.read.option("delimiter"," ").csv(file_location,schema = schema,header = False)
```

(3) 创建数据的表视图(table view)：

```
df.createOrReplaceTempView("raw_engine")
```

(4) 转换数据：

```
%sql
drop table if exists engine;
create table engine as
(select e.*, CASE WHEN mc - e.cycle = 1 THEN 1 ELSE
```

```
CASE WHEN mc - e.cycle < 14 THEN 2 ELSE
0 END END as label
from raw_engine e
join (select max(cycle) mc, engine_id from raw_engine group by engine_id) m
on e.engine_id = m.engine_id)
```

(5) 测试、训练和分割数据：

```
new_input = spark.sql("select * from engine").toPandas()
training_df, test_df = train_test_split(new_input)
```

(6) 准备模型：

```
dtrain = xgb.DMatrix(training_df[['setting1','setting2','setting3',
's1', 's2', 's3',
's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12', 's13', 's14',
's15', 's16','s17', 's18', 's19', 's20', 's21']],
label = training_df["label"])
param = {'max_depth': 2, 'eta': 1, 'silent': 1, 'objective':
'multi:softmax'}
param['nthread'] = 4
param['eval_metric'] = 'auc'
param['num_class'] = 3
```

(7) 训练模型：

```
num_round = 10
bst = xgb.train(param, dtrain, num_round)
```

(8) 评估模型：

```
dtest = xgb.DMatrix(test_df[['setting1', 'setting2', 'setting3',
                             's1', 's2', 's3', 's4', 's5', 's6',
                             's7', 's8', 's9', 's10', 's11',
                             's12', 's13', 's14', 's15', 's16',
                             's17', 's18', 's19', 's20', 's21']])

ypred = bst.predict(dtest)
pre_score = precision_score(test_df["label"], ypred, average = 'micro')
print("xgb_pre_score:",pre_score)
```

(9) 存储结果：

```
with mlflow.start_run():
    mlflow.set_experiment("/Shared/experiments/\
    Predictive_Maintenance")
    mlflow.log_param("type", 'XGBoost')
```

```
mlflow.log_metric("precision_score", pre_score)
filename = 'bst.sav'
pickle.dump(bst, open(filename, 'wb'))
mlflow.log_artifact(filename)
```

3.4.3 工作机理

首先导入 Pandas、Pyspark 和 NumPy 用于数据处理，导入 XGBoost 作为算法，导入 Sklearn 用于对结果进行评估，最后导入 MLflow 和 pickle 用于保存这些结果。步骤(2)在 Spark 中指定了一种模式。Databricks 的推断模式特性常常会使模式出错，通常需要指定数据类型。在步骤(3)中，创建了一个数据的临时视图，以便使用 Databricks 中的 SQL 工具。在步骤(4)中，使用页面顶部神奇的%sql 标签，将语言改为 SQL。然后创建了一个取名为 engine 的表，该表包含引擎数据和一个新的列，如果引擎还剩余 14 个以上的周期，列的值取 0；如果只剩余 1 个周期，列的值取 1；如果剩余 14 个周期，列的值就取 2。然后切换回默认的 Python 语言，将数据分为测试集和训练集。在步骤(6)中，指定模型中的列以及超参数。模型训练完成后进行测试，并打印输出准确率。最后将结果存储在 MLflow 中。第 4 章将针对这个数据集进行其他实验，看看哪一个性能最好。

XGBoost 有大量的参数可以进行调整。这些参数可以是允许算法使用的线程数，也可以是有助于提高准确率或防止过拟合和欠拟合的参数。常见的可调整参数如下。

(1) learning_rate：是算法更新其节点的步长大小。它有助于防止过拟合，但也会对完成训练所需的时间产生负面影响。

(2) max_depth：参数太大可能会出现过拟合，太小则可能出现欠拟合。

(3) predictor：是一个指示程序在 CPU 或 GPU 上进行计算的标志。在 GPU 上计算会显著增加运行的时间，但并不是所有的计算机都有 GPU。

XGBoost 中还有更多的参数可以进行调整。XGBoost 决策树的内部采用的是弱学习器(weak learner)或浅层树(shalow tree)，使用维度评分系统可以将它们组合成强学习器(strong learner)。这就像第一位医生给出了一个糟糕的诊断结果，但是我们还有第二位、第三位医生的诊断。第一位医生的诊断可能是错误的，但不太可能三位医生都是错的。

3.5 危险驾驶行为检测

机器学习中的计算机视觉技术有助于分辨道路上是否发生了交通事故或存在不安全的因素，并可以与智能销售助手等复杂系统结合起来使用。计算机视觉在 IoT 领域开辟了许多的可能性。从成本角度来看，计算机视觉也是最具挑战性的。接下来的实用案例将讨论使用计算机视觉的方式：接收从 IoT 设备生成的大量图像，并使用高性能分布式 Databricks 格式对其进行预测和分析。在下面的实用案例中，我们采用的是一种低计算量的算法，只需

进行少量计算，就可以在边缘设备端执行机器学习。

3.5.1　预备工作

首先需要准备 Databricks，本实用案例将从 Azure Blob Storage 中提取图像。

3.5.2　操作步骤

本实用案例的操作步骤如下。

(1) 导入库并进行配置：

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from sparkdl import DeepImageFeaturizer
from pyspark.ml.evaluation import \
MulticlassClassificationEvaluator
from pyspark.sql.functions import lit
import pickle
import mlflow

storage_account_name = "Your Storage Account Name"
storage_account_access_key = "Your Key"
```

(2) 读取数据：

```
safe_images = "wasbs://unsafedrivers@" + storage_account_name + \
              ".blob.core.windows.net/safe/"
safe_df = spark.read.format('image').load(safe_images)\
              .withColumn("label", lit(0))

unsafe_images = "wasbs://unsafedrivers@" + storage_account_name + \
              ".blob.core.windows.net/unsafe/"
unsafe_df = spark.read.format('image').load(unsafe_images)\
              .withColumn("label", lit(1))
```

(3) 查询数据：

```
display(unsafe_df)
```

(4) 创建测试集和训练集：

```
unsafe_train, unsafe_test = unsafe_df.randomSplit([0.6, 0.4])
safe_train, safe_test = safe_df.randomSplit([0.6, 0.4])
train_df = unsafe_train.unionAll(safe_train)
test_df = safe_test.unionAll(unsafe_test)
```

(5) 建立 pipeline:

```
featurizer = DeepImageFeaturizer(inputCol = "image",
                                 outputCol = "features",
                                 modelName = "ResNet50")
lr = LogisticRegression(maxIter = 20, regParam = 0.05,
                        elasticNetParam = 0.3, labelCol = "label")
p = pipeline(stages = [featurizer, lr])
```

(6) 训练模型:

```
p_model = p.fit(train_df)
```

(7) 评估模型:

```
predictions = p_model.transform(test_df)
predictions.select("filePath", "prediction").show(truncate = False)
df = p_model.transform(test_df)
predictionAndLabels = df.select("prediction", "label")
evaluator = \
MulticlassClassificationEvaluator(metricName = "accuracy")
print("Training set accuracy = " + \
       str(evaluator.evaluate(predictionAndLabels)))
```

(8) 记录结果:

```
with mlflow.start_run():
    mlflow.set_experiment("/Shared/experiments/Workplace Safety")
    mlflow.log_param("Model Name", "ResNet50")
    # Log a metric; metrics can be updated throughout the run
    precision, recall, fscore, support = score(y_test, y_pred, average = 'macro')
    mlflow.log_metric("Accuracy", \
    evaluator.evaluate(predictionAndLabels))
    filename = 'finalized_model.sav'
    pickle.dump(p_model, open(filename, 'wb'))
    # Log an artifact (output file)
    mlflow.log_artifact(filename)
```

3.5.3 工作机理

本实用案例使用 Azure Blob Storage,也可以使用其他存储系统(如 S3 或 HDFS)。用 Blob Storage 账户的密钥替换 storage_account_name 和 storage_account_access_key 字段,从存储账户中读取 safe 和 unsafe 的图像到 Spark Image DataFrame。将 safe 图像放在一个文件夹中,unsafe 图像放在另一个文件夹中。查询图像 DataFrame,看它是否已获取了图像。创建 safe 和 unsafe 的测试集和训练集,然后将数据集合并为一个训练集和一个测试

集。接下来创建一个机器学习 pipeline，使用 ResNet-50 算法作为特征器，使用 Logistic 回归作为分类器。然后将分类器放入 pipeline 并训练模型。通过 pipeline 运行训练 DataFrame，从而得到一个训练好的模型，之后再评估模型的准确率。最后将结果存储在 MLflow 中，以便与其他模型进行比较。

目前已经开发了许多的图像分类模型，如 ResNet-50 和 Inception v3 等。此案例使用了 ResNet-50（一种调谐卷积神经网络），这是一个功能强大的图像特征机器学习模型。在机器学习领域，有一个无免费午餐定理（no free lunch theorem），即没有哪个模型会优于其他所有模型。因此，使用者将需要对不同的算法进行测试，简单来说，可以通过修改参数达到这一目标。

除了模型，还可以利用 pipeline 声明算法实现流程中的不同步骤，并独立实现每个步骤。本例使用 ResNet-50 对图像进行特征化处理，ResNet-50 输出的是可以被分类器分类的特征向量。本案例中使用的是 Logistic 回归，也可以使用 XGBoost 或其他的神经网络。

3.5.4　补充说明

如果将算法模型由 ResNet-50 改为 Inception v3，则需要对 pipeline 进行修改：

```
featurizer = deepImageFeaturizer(inputCol = "image",outputCol = "features",
                                 modelName = "ResNet50")
```

使用 Inception v3，可以在图像集上测试不同模型的准确率：

```
featurizer = DeepImageFeaturizer(inputCol = "image",outputCol = "features",
                                 modelName = "InceptionV3")
```

使用一个模型阵列，并在 MLflow 中记录结果：

```
for m in ['InceptionV3', 'Xception','ResNet50', 'VGG19']:
    featurizer = DeepImageFeaturizer(inputCol = "image",
                                     outputCol = "features",
                                     modelName = m)
```

3.6　在受限设备端进行人脸检测

深度神经网络的性能往往优于其他分类技术。然而，IoT 设备中没有足够的 RAM、算力和存储量。在受限设备端，RAM 和存储往往是以 MB 为单位，而不是以 GB 为单位，这使得传统的分类器无法使用。云端的一些视频分类服务对每台设备的实时流媒体视频收费超过一万美元。OpenCV 的 Haar 分类器与卷积神经网络的基本原理相同，但其所需的算力和存储能力却很小。OpenCV 有多种语言版本，可以在一些受限的设备端运行。

本实用案例将设置一个 Haar Cascade 检测是否有人靠近摄像头，通常可用于 Kiosk 和其他交互式智能设备。Haar Cascade 运行速度很高，当发现有人靠近机器时，它可以通过云服务或不同的机载的机器学习模型发送图像。

3.6.1 预备工作

首先需要安装 OpenCV 框架：

```
pip install opencv - python
```

从 OpenCV 的 GitHub 页面下载模型或本书的提供资源包查找，对应的文件是 haarcascade_frontalface_default. xml。

导入 haarcascade_frontalface_default. xml 文件创建一个新文件夹，并为代码创建 Python 文件。最后，如果设备端没有摄像头，还需要安装一个摄像头。在下面的实用案例中，将使用 OpenCV 实现 Haar Cascade。

3.6.2 操作步骤

本实用案例的操作步骤如下。

(1) 导入库并进行设置：

```
import cv2
from time import sleep

debugging = True
classifier = \
cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
video = cv2.VideoCapture(0)
```

(2) 初始化摄像头：

```
while True:
    if not video.isOpened():
    print('Waiting for Camera.')
    sleep(5)
    pass
```

(3) 捕捉并转换图像：

```
ret, frame = video.read()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

(4) 对图像进行分类：

```
faces = classifier.detectMultiScale(gray,
```

```
                                    minNeighbors = 5,
                                    minSize = (100, 100)
                                    )
```

(5) 对图像进行调试：

```
if debugging:
    # Draw a rectangle around the faces
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

    cv2.imshow('Video', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

(6) 检测人脸：

```
if len(faces) > 0:
    # Your advanced code here
    pass
```

3.6.3　工作机理

首先，导入库并进行设置。下一步，导入 OpenCV 和 Python 库，另外导入 time，当摄像头还没有准备好时可以先进入等待。接下来设置一些调试标志，以便在调试时可以直观地测试输出。将 Haar Cascade XML 文件导入分类器中。最后打开连接到机器上的第一个摄像头。在步骤(2)中，等待摄像头准备好。在软件开发时这通常不是问题，因为系统已经识别了摄像头，然后通过设置使程序自动运行。重启系统后，在 1min 内摄像头可能不可用。之后便开始无限循环地处理摄像头图像，捕获图像并将其转换为黑白图像。

运行 detectMultiScale 分类器，检测不同尺寸的人脸。minNeighbors 参数规定了在检测人脸之前需要多少个协作邻居(collaborating neighbors)。minNeighbors 参数设置太小，可能会导致误判；设置太大，可能根本就检测不到人脸。同时，设置人脸需要的最小像素尺寸。为了确保摄像头精确地进行工作，加入了调试代码，将视频和边界框输出到相连的显示器上。对于测试来说，可以发现问题并进行调试。如果检测到了人脸，那么就可以执行相应的任务，例如机载情感分析，或者将其发送到外部服务，例如 Azure Face API，就可以通过人脸 ID 来进行识别了。

Haar Cascade 是一种高效的人脸检测分类器。它将图像的矩形部分与图像的另一部分进行比较，得到人脸的特征。本实用案例使用设备自带的摄像头，对其进行转换，然后使用 Haar Cascade 对其进行分类。