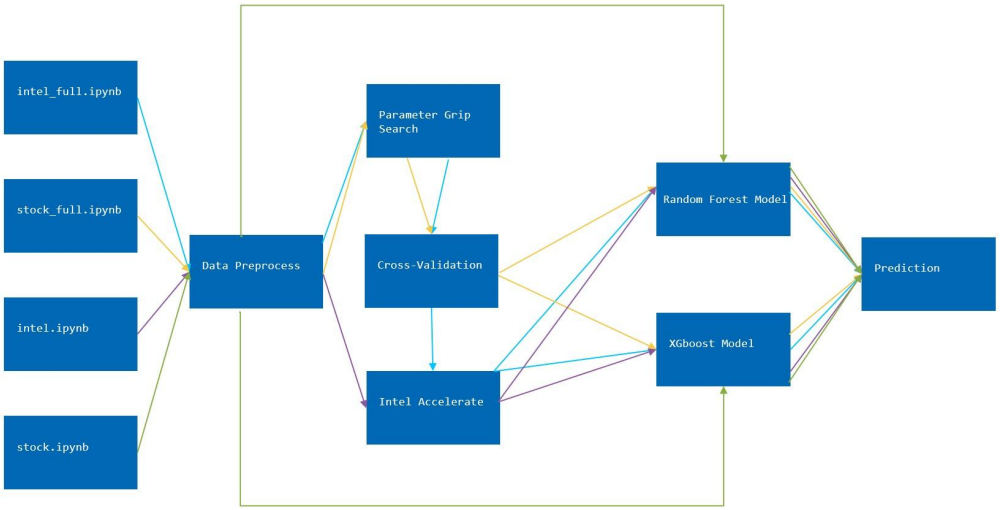
信用卡欺诈检测——利用配备 Intel® oneAPI AI Analytics Toolkit 的机器学习方法

（陈品轩 21302010033 向绪文 21302010055 李相昊 21302010008）

1. 概要

伴随着数字化交易的激增，信用卡已经成为一种广受欢迎的支付工具。然而，针对信用卡的普遍欺诈对金融体系的稳定性构成了巨大的影响。这对银行，客户和金融机构来说不容忽视，其可能导致重大的财务损失，信任危机和声誉下降等等问题。近年来信用卡交易欺诈数量的急剧增加，这使得预测交易过程中的欺诈活动越来越重要。我们团队针对这个问题，对数据集进行了预处理，使用了随机森林算法，XGBoost 算法，并通过网格搜索来进行优化。最重要的是，我们学会了利用 Intel® oneAPI AI Analytics Toolkit，并分别使用了 modin 与 patch 来对预处理以及各种机器学习算法进行加速，从结果上来看取得了不错的效果。

关键词：Intel® oneAPI AI Analytics Toolkit，机器学习，信用卡欺诈检测，随机森林，XGBoost



Structure

2. 数据处理

2.1 数据集简介

在本实验中采取的数据集包含欧洲持卡人在 2013 年 9 月通过信用卡进行的交易。该数据集显示了两天内发生的交易，其中 284807 笔交易中有 492 笔为欺诈交易。数据集高度不平衡，正类（欺诈）占所有交易的 0.172%。它仅包含 PCA 变换结果的数字输入变量，但由于保密问题，特征 V1 到 V28 无法体现有关数据的原始特征和更多背景信息。但其中的“时

间”特征包含了数据集中每个事务和第一个事务之间间隔的秒数。“金额”特征是交易的金额，该特征可以用于示例相关的成本敏感学习。“类别”特征是响应变量，如果存在欺诈，则为 1，否则为 0。考虑到类别不平衡率，我们将使用精确率-召回率曲线下面积 (AUPRC)来测量准确度。

2.2 数据预处理

2.2.1 数据清洗

在数据的生命周期中，会不可避免地发生数据丢失的问题。该问题客观存在于数据的采集，传输，处理等等阶段。处理数据集中的数据丢失问题是预处理中的关键问题。对于该问题，我们有不同的处理方式。其中，包括有删除缺失值，均值填充，回归、分类模型填充等等办法。在本次实验的数据预处理中，考虑到庞大的数据量以及低密度的数据缺失分布，我们采取了直接删除缺失值的方法。如下图。

```
print(data.isnull().sum()) # 缺失值检查
data = data.dropna()  # 直接丢失含有缺失值的行记录
```

2.2.2 Time 特征处理

我们小组一开始认为信用卡欺诈对于时间分布可能具有周期性。因此，我们尝试将时间特征分解为天，小时，分钟，秒，希望能通过该种方法，在数据集原本时间的基础上，以日常生活的逻辑细分化，从而更加深入的挖掘出潜藏的规律。但是在经过尝试后，我们发现训练与预测的结果并不好，甚至比原来有所降低。讨论后我们觉得 Time 这个特征对单个人是否欺诈这个行为应该并没有太大影响，于是最后选择舍弃了 Time 这个特征。

```
df["days"], df["hours"], df["minutes"], df["seconds"] = zip(*df["Time"].map(convert_seconds))   # 拆分Time特征
```

2.2.3 噪声检测与去除

在本次实验中，噪声的检测与去除主要体现在数据离群值的检测与去除。在一般实验中，离群值的阈值会被设置为 3，但是结合本次实验数据集的特点，在探索后我们发现 Class = 1，即欺诈行为的特征比较明显，有显著离群倾向。此时，较小的离群值阈值会使得大量正类样本被检测出离群并被去除。所以，在数次尝试后，我们在实验中设置了较高的离群值阈值。

```
# 提取特征数据，去除不需要的列
features = df.drop(["Time","Class"], axis=1)
# 计算每个特征的Z-score
z_scores = np.abs(stats.zscore(features))
# 设置离群值的阈值
threshold = 30
# 检测离群值
outliers = np.where(z_scores > threshold)
# 去除离群值所在的行
df = df.drop(outliers[0], axis=0)
```

我们小组通过对正负样本数量的跟踪，对离群值阈值的影响进行了以下探索：
以下是对 70%样本的统计

	Class=0	Class=1	Total
Original dataset	199008	356	199364
离群值阈值=3	172863	30	172893
离群值阈值=5	193307	49	193356
离群值阈值=10	197702	190	197892
离群值阈值=20	198816	296	199112
离群值阈值=30	198957	347	199304

通过比较正负样本减少的程度，可以看出阈值设的较大比较合理，不然会丢失较多正类数据。

2.2.4 标准化

在机器学习领域，尤其是以梯度和矩阵为核心的机器学习算法中，例如逻辑回归，神经网络等等，采取数据标准化可以加快求解速度。而在距离类模型，例如 KNN，K-Means 聚类中，采取数据标准化可以提升模型精度，避免取值范围过大的特征对距离计算造成不利影响。为了避免之后使用各类算法时对结果造成不良影响，我们也针对我们的数据集，采取了数据的标准化。通过对数据的检索，我们可以知道主要是 Amount 变量与 Vx 变量量级差距太大，于是我们选择将 Amount 变量标准化为符合 N(0,1)的正态分布，并存在 NormalAmount 列中，在之后的训练中只采用 NormalAmount 列。

```
# 标准化Amount属性，使量级相同
data['NormalAmount']=StandardScaler().fit_transform(np.array(data.Amount).reshape(-1,1))
```

2.2.5 有关连续变量离散化的尝试（K-means）

K-means 是无监督机器学习中的重要聚类算法。开始时，我们试图采用 K-means 算法来实现连续变量离散化。然而，考虑到本次实验数据集的特点，我们暂不使用 K-means。这是由于本次实验中样本数量过大，对于大量样本，过少的簇会导致较大的误差，而增加簇则又会导致效率低下。在经过尝试后我们发现对结果的损害比较显著，于是立刻停止了这种尝试。

```
# # 连续变量离散化
# list = ['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11', 'V12', 'V13',
#         'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25',
#     'V26', 'V27', 'V28', 'NormalAmount']
# for column in list:
#     df = [[x] for x in data[column]]
#     kmeans = KMeans(n_clusters=6, random_state=0).fit(df)
#     data[column] = kmeans.labels_
```

2.2.6 不平衡样本处理

样本分布的不平衡主要在于不同类别间的样本比例差异。如果不同类别之间的样本量差异过大，则应对其进行介入与处理。

在信用卡欺诈检测的案例中，正类情况（信用卡欺诈情况）很少。这些样本仅仅占据了整个数据集相当小的一部分，可以计算出，本实验中它们只占 0.172%。因此，我们小组利用了 SMOTE 算法，合成新的少数类样本，试图来解决该问题。该算法在原先的基础上，稍加平衡了各个分类间的样本数量，提高了机器学习模型识别少数类样本的能力。然而，一般情况下 SMOTE 生成会使得正负类样本数量相同，然而针对这个样本集会产生严重的过拟合问题，使得预测结果有极大偏差，因此我们没有选择生成太多少数类样本，仅将其比例控制在 0.5%，相对于原先有小部分的增加。

```
sm=SMOTE(sampling_strategy=1/200,random_state=42)
print('Original dataset shape %s' % Counter(y_train))
X_res, y_res = sm.fit_resample(X_train, y_train)
print('Resampled dataset shape %s' % Counter(y_res))
X_train = pd.DataFrame(X_res, columns=X_train.columns)
y_train = pd.Series(y_res, name='Class')
```

我们针对这类问题，也想过采取其他的解决方法。例如，通过正负样本的惩罚权重来解决样本的不均衡问题。这种方法的优势集中体现在，不需要对样本本身做额外处理，只需要在算法模型中设置 class_weigh 即可。但是由于一开始并没有确定好使用的算法是什么，可能算法并不支持这一策略，最终选择在一开始用 SMOTE 算法来解决不平衡问题。

3. 训练与预测

3.1 随机森林算法

随机森林算法是一种在训练阶段构建多个决策树的集成学习方法。它通过聚合来自大量单个决策树的预测来运行。每个决策树都在训练数据和特征的不同随机子集上进行训练。通过装袋法，以及特征的随机化过程，随机森林减轻了过拟合，并且增加了鲁棒性。使其能够适应噪声，处理高维数据集。在我们的实验中，首先采取了随机森林算法来训练模型，如下图所示。这里由于数据集比较庞大，训练比较耗时，我们将训练完的结果存在 pkl 文件下，方便之后直接使用该模型。在训练好后直接对测试集进行预测即可。

```
rf_model = RandomForestClassifier(random_state=42)
Executed at 2023.11.15 17:40:13 in 7ms
```

训练随机森林模型（若存在则直接读取）

```
start_time = time.time()
model_file = 'random_forest_model_intel.pkl'
if os.path.exists(model_file):
    with open(model_file, 'rb') as file:
        rf_model = pickle.load(file)
else:
    rf_model.fit(X_train, y_train)
    with open('random_forest_model_intel.pkl', 'wb') as file:
        pickle.dump(rf_model, file)
training_time_rf = time.time() - start_time
```

3.2 XGBoost 算法

XGBoost(Extreme Gradient Boosting)是一种高效的梯度提升决策树算法。他在原有的 GBDT 基础上进行改进，使得模型效果得到大大提升。作为一种前向加法模型，他的核心是采用集成思想——Boosting 思想，将多个弱学习器通过一定的方法整合为一个强学习器。XGBoost 是由多棵 CART 树组成，因此可以用于处理分类问题。在采取随机森林算法后，我们还希望选择 XGBoost 算法来重复这个实验，并能够将两种算法进行一定的比较。

```
xgb_model = XGBClassifier(random_state=42)
```
Executed at 2023.11.15 17:40:17 in 6ms

Add Code Cell | Add Markdown

训练xgboost模型（若存在则直接读取）

```
start_time = time.time()
model_file = 'xgboost_model_intel.pkl'
if os.path.exists(model_file):
    with open(model_file, 'rb') as file:
        xgb_model = pickle.load(file)
else:
    xgb_model.fit(X_train, y_train)
    with open('xgboost_model_intel.pkl', 'wb') as file:
        pickle.dump(xgb_model, file)
training_time_xgb = time.time() - start_time
```

3.3 网格搜索优化

网格搜索是一种穷举搜索方法，它通过遍历超参数的所有可能组合来寻找最优超参数。网格搜索首先为每个超参数设定一组候选值，然后生成这些候选值的笛卡尔积，形成超参数的组合网格。接着，网格搜索会对每个超参数组合进行模型训练和评估，从而找到性能最佳的超参数组合。然而网格搜索的缺点是计算量较大，当超参数的数量和候选值较多时，搜索空间会急剧增大，导致搜索效率低下，在我们优化后开始训练时，发现确实会耗费巨大的时间，这会在后面的分析中指出。这里针对随机森林算法和 XGBoost 算法，我们都进行了网格搜索优化，并希望通过与未优化的进行对比，来分析这种优化方法的好处。

```
# 使用StratifiedKFold进行分层交叉验证
cv_rf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# 使用GridSearchCV进行参数搜索和交叉验证
grid_search_rf = GridSearchCV(rf_model, param_grid_rf, scoring=f1_scorer, cv=cv_rf)
grid_search_rf.fit(X_train, y_train)
# 获取最佳参数的模型
best_rf_model = grid_search_rf.best_estimator_
```

4. Intel® oneAPI AI Analytics Toolkit 加速优化

应用 Intel® oneAPI AI Analytics Toolkit 给我们小组的机器学习过程带来了极大便利性。首先在预处理过程我们使用了 Intel 的 modin 库，modin 库主要通过支持多核的并行化来加速 Pandas API 的执行速度，即 modin 重新打包了 Pandas 中的 API，以便使用多线程在多个内核上并发运行，优化了硬件利用率。其内置的库还针对底层硬件和异构平台进行了深度优化，显著提高了加速性能。于是，针对这种数据量比较庞大的数据集，用 modin 库再合适

不过了。同时，为了使用更快的 numpy，我们改用 conda 来安装基于 Intel MKL 的版本（删掉了原来 pip 安装的版本）

另外，对于使用到的随机森林算法与 XGBoost 算法，可以用 Intel 的 patch 技术来加速，这是通过将 Scikit-learn 的库存算法替换为利用矢量指令的版本来实现。

数据表明，Intel® oneAPI AI Analytics Toolkit 的加速是十分显著的，这将在下面的实验结果中进行具体分析。

5. 实验结果

各项数据对比表（前缀 Full 代表有网格搜索优化，Intel 表示有使用 oneAPI，RF 指随机森林）：

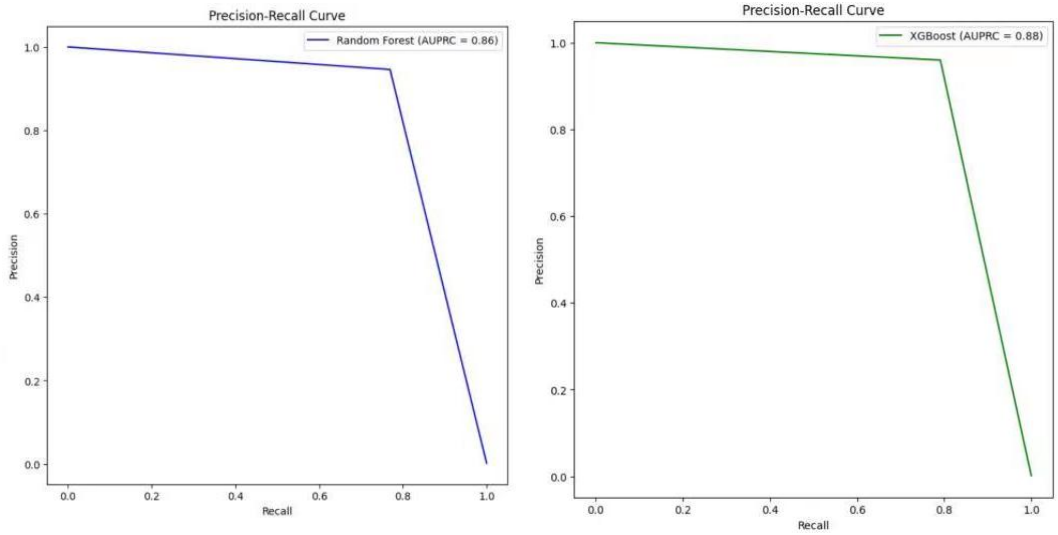
	Intel	Stock	Full_Intel	Full_Stock
Read Data Time	0.248s	1.447s	0.191s	1.341s
Pre_Processing Data Time	2.202s	0.737s	1.658s	0.781s
RF Training Time	2.787s	215.227s	647.079s	?
RF Predicting Time	0.072s	0.469s	0.105s	?
RF F1 Score	0.849	0.845	0.854	?
XGboost Training Time	9.631s	66.909s	949.776s	?
XGboost Predicting Time	0.174s	0.081s	0.093s	?
XGboost F1 Score	0.868	0.868	0.859	?

分析：

首先在 F1 Score 上，使用 Intel® oneAPI AI Analytics Toolkit 没有起到很显著的作用，每种算法前后几乎都是一样，在添加网格搜索优化后随机森林算法 F1 Score 会有一些提升，但也不显著，主要还是因为在大量样本训练下的模型，本身预测率就很高了，很难对其造成实质性的提升。

然后是 Intel® oneAPI AI Analytics Toolkit 在时间的加速上，在数据处理方面，首先读取数据的时候可以看到有明显的差别，大概提升了仅 6 倍。但是预处理过程没有显著提升，甚至不如原来的，主要是涉及到处理离群值，modin 中的 pandas 需要多一条 np 转换成数组的代码，即 features=np.array(features) 时耗费了比较多时间，所以并不能评判其加速效果的好坏。 在对模型的训练时间上，随机森林加速了近 80 倍，XGboost 加速了约 7 倍，对测试集的预测上，随机森林算法加速近 6 倍，但 XGboost 没有显著效果。但 Intel® oneAPI AI Analytics Toolkit 还是提供了很大的便利，尤其是为在两种算法里训练时实现网格搜索优化提供了可能，未用 Intel 进行加速的网格搜索优化程序在跑了 5h 后仍没有结果，不得已放弃了。

绘制出的 PRC 曲线（以 Intel 加速的未经网格搜索优化为例）：



总的来说，我们训练的模型 AUPRC 在 0.86--0.88 之间，预测欺诈行为的能力还是很不错的，在一定程度上能够有效地提前预测出欺诈行为，并及时执行相应措施来减少各个方面的损失。

GitHub 链接：https://github.com/xs-keju/Credit_detection_accelerate_with_Intel_one_API.git