The detection of credit card transaction fraud – Applying machine learning methods with Intel® oneAPI AI Analytics Toolkit

Team member:

21302010033 陈品轩
21302010055 向绪文
21302010008 李相昊

1.Abstract

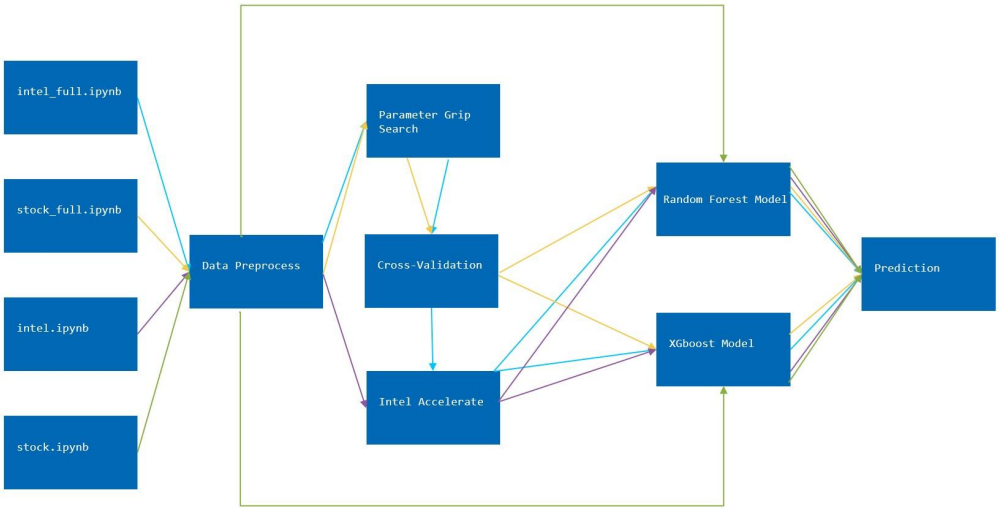
Amidst the Surge in digital transactions, credit cards have emerged as a revolutionary payment tool. However, the pervasive fraud against credit cards poses a formidable thread to the stability of financial systems. This constitutes a worrisome issue for banks, customers, and financial institutions in terms of significant financial losses, trust crisis, and reputation decrease. Considering the sharp rise in fraudulent credit card transactions, detecting fraudulent activities during transactions is crucial in aiding consumers and banks. This essay intends to introduce the Random Forest and XGBoost with several data-preprocessing methods as the main machine learning method, embedding the Intel® oneAPI AI Analytics Toolkit to train a series of machine learning models, anticipating fraud activities lurking in credit card trades. The results revealed that applying the Intel® oneAPI AI Analytics Toolkit, the machine learning process demonstrates a satisfied outcome with celerity and binary classification accuracy.

Keywords: Intel® oneAPI AI Analytics Toolkit, machine learning, credit cards fraud, random forest, XGBoost

2.Introduction

Machine learning is fundamental to process researches with massive data. Despite its comprehensive pervasion, a search of the literature revealed few studies applying the revolutionary creative Intel® oneAPI AI Analytics Toolkit to their main machine learning process.

In this essay, we attempt to derive a series of machine learning models by implementing Random Forest and XGBoost to provide the effective anticipation, with the focus on celerity and binary classification accuracy. In addition, the credit card fraud dataset acquired from the Intel® website was selected in the experiment as the raw data, and was preprocessed by our team. Furthermore, we applied Intel® oneAPI AI Analytics Toolkit, accelerating the Data-centric Workloads, furnishing the traditional machine learning methods with marvelous optimization.



Structure diagram

3.Methods

3.1 Dataset overview

The dataset comprises credit card transactions of European cardholders in September 2013.
It illustrates transactions occurring over 2 days, encompassing 284807 transactions, among which 492 are fraudulent. The dataset exhibits high imbalance, with the fraudulent positive class accounting for 0.172% of all transactions.
The dataset consists solely of numerical input variables derived from PCA. Regrettably, due to confidentiality concerns, V1 through V28 are kept unrevealed, while the Time feature denotes the seconds elapsed between each transaction and the first one in the dataset, and the Amount feature represents the transaction amount, serving as a cost-sensitive learning example. The Class feature is the response variable, taking a value of 1 in the presence of fraud and 0 otherwise.

Given the imbalance, out team employed the AUPRC to gauge accuracy.

3.2 Data preprocessing

3.2.1 Data cleansing

Data missing is an inevitable issue encountered throughout the data lifecycle, occurring during data collection, transmission, and processing stages. Dealing with missing data in datasets is a pivotal aspect of data preprocessing. There are various alternatives, including eliminating missing value, mean filling etc. In our case, considering the massive amount of data and the low percentages of missing data, we detected and eliminated the missing value. As shown in code segment 1.

```
df = df.dropna()  # 直接丢失含有缺失值的行记录
```

Code segment 1

3.2.2 Time processing

Considering the probability of daily cyclical nature of fraud, we disassembled Time feature into detailed attributes – days, hours, minutes, seconds. Implementing such methods enables out team to generate models with detailed subdivision of time instead of simply using the streaming time records. With this optimization, we can research on its characteristics more intuitively, and dig for more profound patterns. As shown in code segment 2.

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

Code segment 2
However, the training outcome have demonstrated suboptimal functions towards anticipation. Therefore, we have withdrawn this proposal with the Time feature.

3.2.3 Detection and Elimination of Noise Values

In this case, the detections and elimination of noise mainly focuses on Outlying Values. Many researchers have utilized threshold as 3 in similar studies. However, in our case, the fraud feature is extreme obvious (as indicated previously “Class = 1”), leaning to be outlying. Setting a low threshold would bring about the deletion of massive positive-class samples.

This conclusion is not only derived from a theoretical inference but also through the threshold detection and statistical analysis. Table 1 below has clearly demonstrated that with larger thresholds, there is an obvious reduction in the loss of positive-class data.

Table 1: Statistics pertaining to the 70% sample

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

Therefore, after several statistical attempts, our team set a greater threshold, as shown in code segment 3. After the detections of outlying values, we removed the rows of which.

```
# 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)
```

Code segment 3

3.2.4 Standardization

In the algorithms with gradient and matrix as their foundation, data standardization expedites the solution process. In distance-based models, data standardization enhances model accuracy by mitigating the impact of features with significantly larger value ranges on distance computations.

Considering the disparity between 'Amount' and Vx is substantial, our team also applied standardization to avoid its adverse impact on the training model. In detail, we opt to standardize 'Amount' to a normal distribution, storing it in the 'NormalAmount' column, which will be exclusively used thereafter. As shown in code segment 4.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
X_train = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test = pd.DataFrame(X_test_scaled, columns=X_test.columns)
```

Code segment 4

3.2.5 Attempts of Discretization of Continuous Variables (K-means)

K-means serves as a fundamental clustering algorithm in worldwide unsupervised machine learning. In our study, we implement K-means to identify inherent patterns and structures within the data. However, after several attempts, we determined not to implement K-means, considering the massive amounts of samples. The deficiency of clusters would generate large errors, while the augment of which would lead to low efficiency. Therefore, we have suspended this option. As shown in code segment 5.

```
# 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', 'Amount','days','hours','minutes','seconds']
for column in list:
    data = [[x] for x in df[column]]
    kmeans = KMeans(n_clusters=6, random_state=0).fit(data)
    df[column] = kmeans.labels_
```

Code segment 5

3.2.6 Handling Imbalanced Samples

The imbalance in sample distribution is mainly due to the difference in sample proportions between different categories. When the difference reaches a relatively high level, interventional disposal of unbalanced samples is required in machine learning.

In the case of credit cards fraud, the abnormal cases (fraud case) are quite rare. These data samples typically represent a relatively small fraction of the overall sample, which in our case representing 0.172%. Therefore, our team use the SMOTE algorithm to address imbalanced samples. It balances the number of samples between different classes by synthesizing minority class samples, enhancing the model's ability to recognize the minority class samples. Meanwhile, in order to alleviate the overfitting scenes, we set the sampling strategy to 0.5%, as the optimized argument. As shown in code segment 6.

We also attempted the use of penalty weight for positive and negative samples to address the issue of sample imbalance. In such cases, the sample will not be pre-processed. Contrarily, in the process of Random Forest, the argument "class_weight" will be set, resolving the problem succinctly. However, due to practical reasons, we eventually adopted the SMOTE algorithm.

```
    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')
    return X_train,y_train
```

Code segment 6

4. Training and Prediction

4.1 Random Forest

Regarding to the credit card fraud dataset, our team implemented Random Forest to generate machine learning models.

Random Forest is an ensemble learning method based on constructing multiple decision trees during the training phase. It operates by aggregating predictions from aa multitude of individual decision trees, each trained on different random subsets of the training data and features. Through a process of bagging and feature randomization, Random Forest mitigates overfitting and enhances robustness, making it resilient to noise and capable of handling high-dimensional datasets. In our case, we implement Random Forest as shown in Code segment 7. For efficiency reasons, we store the models in a .pkl file and directly call this file if it has already existed. Afterwards, we use the trained model for predictions.

```
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
```

Code segment 7

4.2 XGBoost

XGBoost is widely known as a prominent machine learning algorithm based on gradient boosting frameworks. It trains an ensemble of weak learners iteratively to optimize a given loss function, progressively refining model performance. An eminent advantage of XGBoost lies in its employ of regularization techniques and parallel processing to minimize prediction errors, as well as accommodating diverse data types and features during modeling process. In our case, we also implemented XGBoost as shown in code segment 8. Similarly, we adopted a .pkl file to store the trained models, optimizing the efficiency.

```
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
```

Code segment 8

4.3 Grid search optimization

Grid search optimization is a prevalent technique in machine learning. It involves systematically defining a range of hyperparameters for a model, constructing a “grid”. Subsequently, it iterates through this grid, evaluating the performance of the model for each parameter combination, and ultimately identifying the optimal configuration that maximizes model performance. In our case, we implemented grid search optimization on both Random Forests and XGBoost. We defined the grid parameter param_grid_rf, and generated the optimal parametric model, as shown in code segment 9.

```
# 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_
```

Code segment 9

5. Conclusions

Focusing on the dataset of credit card fraud, our team applied Intel® oneAPI AI Analytics Toolkit in order to acquire the revolutionary optimization to the machine learning process, thus ensuring our project with high celerity and binary classification accuracy, and finally generates a series of machine learning models, anticipating fraud activities lurking in credit card trades.

5.1 Intel® oneAPI AI Analytics Toolkit Optimizations

Applying the Intel® oneAPI AI Analytics Toolkit ensures our team with considerable convenience. The Intel Modin library accelerates the execution speed of Pandas API by enabling parallelization across multiple cores. While the original Pandas operates on a single thread, Modin repackages the APIs within Pandas to concurrently run across multiple cores using multithreading, thereby optimizing hardware utilization. In addition, the plugin library undergoes deep optimization targeting underlying hardware and heterogeneous platforms, significantly enhancing acceleration performance. Finally, the patch acceleration throughout machine learning also contributes to the overall optimization by substituting the Scikit-learn algorithms to the vectorized instruction version.

Intel® oneAPI AI Analytics Toolkit have greatly assisted our team with its concise and powerful optimization capabilities, as well as its straightforward usage. Its accelerating optimization is quite remarkable. Rough estimates from our team indicate a manifest acceleration in the two critical stages of our experiments. As shown in table 2 below.

	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	?

Table 2

According to table 2, we have adopted two algorithms: Random forests (RF) and XGBoost (XGboost). Subsequently, we implemented grid search optimization on them (Demonstrated in “Full” column). In addition, in order to estimate the optimization of Intel® oneAPI AI Analytics Toolkit (Intel), we established a control group to assess the effects of acceleration.

As demonstrated in the table, in terms of F1 Score, Intel® oneAPI AI Analytics Toolkit did not exhibit a notably significant impact. However, the assistance of grid search optimization led to a marked enhancement. We inferred that the substantial samples provided the models with an abundant data source, thus indicating a already-satisfied score, which is difficult to optimize.

In terms of efficiency (Time), we adopted modin to accelerate the whole process. When it comes to reading data, its significant effect reveals an approximate sixfold increase in performance. However, the preprocessing process doesn’t demonstrate such evident optimization. We speculate that this might be attributed to a divergence in this specific line of code “features = np.array(features)” within the modin library compared to the conventional Pandas library. This could lead to the additional time consumption during the conversion of np to arrays. Therefore, it cannot serve as the source for the evaluation of the optimization. Focusing on the training process,

the acceleration effect of Random Forest approached nearly 100-fold, while XGBoost achieved a nearly 7-fold improvement. In addition, when it comes to the prediction time, the optimization ensures a nearly 6-fold acceleration to Random Forests, while the optimization on XGBoost is not quite manifest.

In conclusion, the Intel® oneAPI AI Analytics Toolkit provide the whole machine learning process with a significant improvement on efficiency and convenience, especially under the circumstances where the grid search optimization is implemented.

5.2 Results regarding to credit card fraud

By implementing the Random Forest and XGBoost algorithm with several data-preprocessing methods as the main machine learning method, and embedding the Intel® oneAPI AI Analytics Toolkit to train a series of machine learning models, our team have successfully trained the model for anticipating fraud activities lurking in credit card trades. As the PRC curve demonstrates below in figure 1 and figure 2.

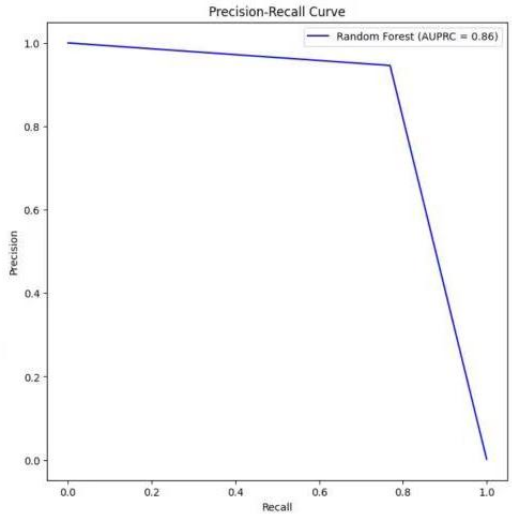


Figure 1

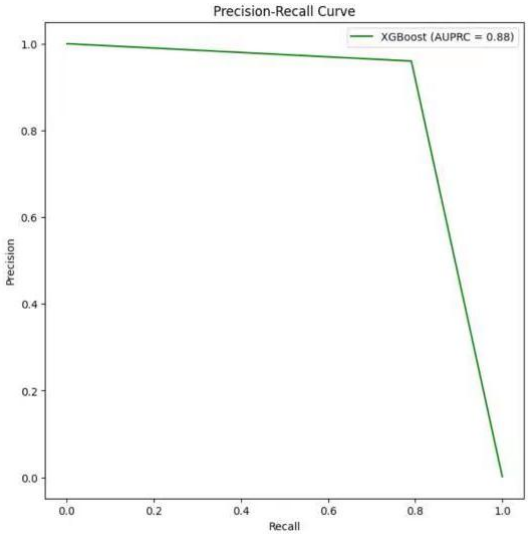


Figure 2

In conclusion, the models we trained demonstrate an AUPRC ranging between 0.86 to 0.88, indicating a strong capability in predicting fraudulent behavior. This proficiency allows for effective anticipation of fraudulent activities to a certain extent, enabling timely actions to mitigate losses.

Github:https://github.com/xs-keju/Credit_detection_accelerate_with_Intel_one_API.git