2019 STAT424

FINAL REPORT

Fraud Detection Using Python

2017150409 이소연

erinlee42@naver.com

1. Introduction

1.1. 레포트 목적 및 개요

본 레포트에서는 simulated sample of day-to-day money transactions including the occurrence of fraudulent transactions에 대한 classification을 진행하여, fraud detection에 통계적으로 유의미한 변수가 무엇인지 확인하고, 다양한 classification 모델 중 금융사기를 가장 잘 분류할 수 있는 모형을 찾을 것입니다.

우선, data set에 대한 Exploratory Data Analysis를 진행한 후, PCA를 통하여 선정된 유의미한 독립변수와 종속변수를 train data와 test data로 나눌 것입니다. 다음 Model Selection 단계에선, 데이터를 각 Logistic Regression, Random Forest, XGB, Decision Tree, LDA 모델에 적합시켜 각 모델의 분류성능(Classification accuracy rate)을 평가할 것입니다.

1.2. 데이터 소개

Data Source: https://www.kaggle.com/ntnu-testimon/paysim1

```
data.shape
```

```
(6362620, 11)
```

```
data.head(7)
```

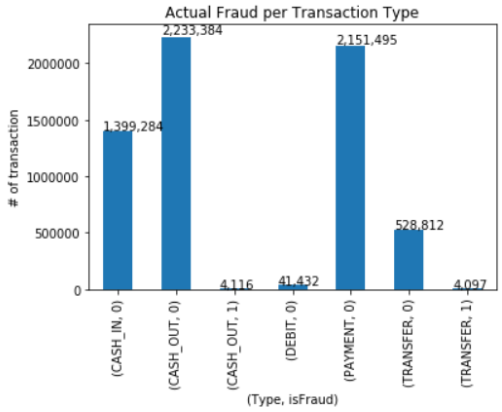
	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1	0
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0	0
5	1	PAYMENT	7817.71	C90045638	53860.0	46042.29	M573487274	0.0	0.0	0	0
6	1	PAYMENT	7107.77	C154988899	183195.0	176087.23	M408069119	0.0	0.0	0	0

step(*integer)* - transaction이 시행된 단위 시간(1 step = 1시간) max = 744.
type(*string/categorical*) - type of transaction (CASH-IN, CASH-OUT, DEBIT, PAYMENT)
nameOrig(*string*) - initiator of the transaction
nameDest(*string*) - recipient of the transaction
oldbalanceOrig*(float*) - initial balance before the transaction
newbalanceOrig(*float*) - new balance after the transaction
isFraud(*boolean/binary*) – fraud(1)/not fraud(0)를 구별해주는 binary indicator
isFlaggedFraud(*boolean/binary*) – 해당 데이터를 제공한 기관에서 시행한 자체적인 detection 결과(한번에 €200,000 이상의 transfer를 시도했을 경우 fraud(0)라고 판정)

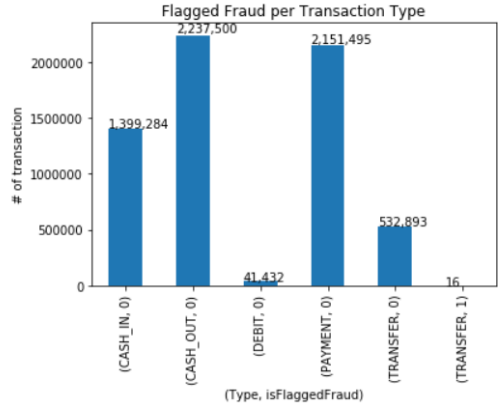
2. Explanatory Data Analysis

2.1. Removing Redundant Variables

```
ax = df.groupby(['type', 'isFraud']).size().plot(kind='bar')
ax.set_title("Actual Fraud per Transaction Type")
ax.set_xlabel("(Type, isFraud)")
ax.set_ylabel("# of transaction")
for p in ax.patches:
    ax.annotate(str(format(int(p.get_height()), ',d')), (p.get_x(), p.get_height()*1.01))
```




```
ax = df.groupby(['type', 'isFlaggedFraud']).size().plot(kind='bar')
ax.set_title("Flagged Fraud per Transaction Type")
ax.set_xlabel("(Type, isFlaggedFraud)")
ax.set_ylabel("# of transaction")
for p in ax.patches:
    ax.annotate(str(format(int(p.get_height()), ',d')), (p.get_x(), p.get_height()*1.01))
```



먼저, isFlaggedFraud변수와 isFraud변수를 각 transaction type에 대한 그래프를 그려서 비교해 보았습니다. 이를 통해 실제로 8,000건 이상의 금융 사기가 있었음에도, isFlaggedFraud는 오직 16건만을 감지할 수 있었음을 확인하였습니다. 특히, 실제로 4,000건 이상의 사기가 발생한 Cash_out type의 transaction에서 아무 금융사기를 발견하지 못한 것으로 보았을 때, 이 변수는 fraud detection에 불필요한 binary 변수라고 결론지었으며, 해당 변수를 제거하였습니다.

또한, 고객과 거래자의 식별ID인 namesOrig와 namesDest 또한 fraud detection과는 관계없는 string 변수라고 판단하여, 아래와 같이 세 변수를 제거하였습니다.

```
df.drop(['nameOrig','nameDest','isFlaggedFraud'], axis=1, inplace=True)
df = df.reset_index(drop=True)
```

	step	type	amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	isFraud
0	1	PAYMENT	9839.64	170136.0	160296.36	0.0	0.0	0

또한, 앞서 제시한 그래프에서 금융사기는 transaction type이 'cash_out' 혹은 'transfer' 일 때만 일어남을 확인하였습니다. 따라서 다른 transaction type을 가진 데이터를 제거하고, 두 변수를 카테고리화 하였습니다.

```
df_new = df[(df["type"] == "CASH_OUT") | (df["type"] == "TRANSFER")]
```

```
df_new["type"]=df_new["type"].astype('category')
```

2.2. PCA & Preprocessing

이제 앞서 언급된 type 변수를 dummy variable로 변형한 후, PCA를 통하여 각 남은 변수들이 종속변수 isFraud의 변동을 잘 설명할 수 있는지 알아보았습니다. 우선 데이터를 표준화시킨 다음, X, y를 지정하여 train과 test data로 나누어 준 후(preprocessing), 아래 과정을 시행하였습니다.

```
df_pca=pd.get_dummies(df_new, columns=["type"])
```

```
df_pca = df_pca[["step", "amount","oldBalanceOrig","newBalanceOrig","oldBalanceDest","newBalanceDest","type_CASH_OUT","type_TRANSFER","isFra
```

```
df_pca.head(1)
```

	step	amount	oldBalanceOrig	newBalanceOrig	oldBalanceDest	newBalanceDest	type_CASH_OUT	type_TRANSFER	isFraud
2	1	181.0	181.0	0.0	0.0	0.0	0	1	1

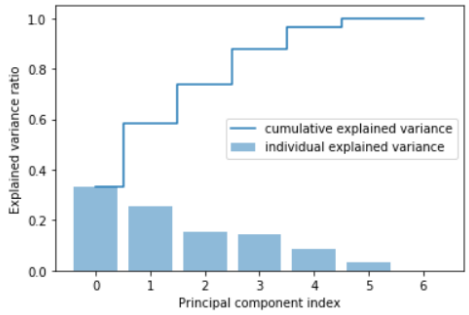
```
from sklearn.model_selection import train_test_split
X,y=df_pca.iloc[:,:7].values, df_pca.iloc[:,8].values
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3, random_state=42,stratify=y)
from sklearn.preprocessing import StandardScaler
std=StandardScaler()
X_train_std=std.fit_transform(X_train)
X_test_std=std.transform(X_test)
```

```
import numpy as np
scov=np.cov(X_train_std.T)
eigen_vals, eigen_vecs=np.linalg.eig(scov)
print('Eigenvalues \n%s' %eigen_vals)
```

```
Eigenvalues
[2.31030595 1.77298224 0.9964026  1.07582257 0.60998434 0.00778826
 0.22671765]
```

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)

import matplotlib.pyplot as plt
plt.bar(range(0,7), var_exp, alpha=0.5, align='center',label='individual explained variance')
plt.step(range(0,7), cum_var_exp, where='mid',label='cumulative explained variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.show()
```



Eigenvalue와 Cumulative Explained Variance를 살펴본 결과, 데이터의 약 80%를 설명할 수 있는 총 3개의 Principal component로 PCA를 시행하기로 결정하였습니다.

```
from sklearn.decomposition import PCA
lpca = PCA(n_components=3)
X_train_pca = lpca.fit_transform(X_train_std)
X_test_pca = lpca.transform(X_test_std)
```

```
print(lpca.components_)
print(lpca.explained_variance_ratio_)
```

```
[[ 0.05010558  0.4404905   0.03290576  0.0065947   0.59041178  0.63292286
  -0.2305512 ]
 [-0.00695437  0.07458838  0.70589716  0.70053911 -0.05680113 -0.04529507
  -0.00802158]
 [-0.08571875 -0.40176047  0.0048973   0.0907898   0.35813591  0.24168265
   0.79768554]]
[0.33004354 0.25328305 0.15368886]
```

그 결과, 각 factor score을 살펴보니 amount, newBalanceDest, CASH_OUT 변수들은 첫번째 PC에, oldBalanceOrig, newBalanceOrig, oldBalanceDest 변수들은 두번째 PC에, TRANSRER 변수는 세번째 PC에 비교적으로 큰 기여를 하고 있으므로, fraud detection을 하기 위한 최종 모델에 이 모든 독립변수를 포함시키는 것이 적절하다고 판단하였습니다. 이 때 첫번째 step 변수는 세 개의 PC에 대한 기여도가 모두 작지만, 아래 결과와 같이 4개의 PC를 사용할 경우 네 번째 PC에 약 99% 기여하므로 이 변수 또한 최종모델에 포함시키는 것이 적절하다고 판단하였습니다.

```
print(lpca.components_)
```

```
[[ 5.01055804e-02  4.40490500e-01  3.29057613e-02  6.59470324e-03
   5.90411782e-01  6.32922858e-01 -2.30551197e-01]
 [-6.95437452e-03  7.45883844e-02  7.05897165e-01  7.00539111e-01
  -5.68011253e-02 -4.52950673e-02 -8.02157718e-03]
 [-8.57187528e-02 -4.01760473e-01  4.89729611e-03  9.07898024e-02
   3.58135907e-01  2.41682652e-01  7.97685536e-01]
 [ 9.94713642e-01 -4.09723145e-02  1.38369966e-02  1.70337687e-04
  -2.05821326e-03 -1.55936009e-02  9.17995213e-02]]
```

3. Model Selection

3.1. PCA를 통해 차원 축소 후 로지스틱 회귀

앞서 PCA의 결과에 이어서, 로지스틱 회귀모형에 데이터를 적합시켜보았습니다.

```
from sklearn.linear_model import LogisticRegression
lr=LogisticRegression()
lr.fit(X_train_pca, y_train)
y_train_pre=lr.predict(X_train_pca)
y_test_pre=lr.predict(X_test_pca)
```

```
print("PCA-Logistic Regression Accuracy",metrics.accuracy_score(y_train, y_train_pre),metrics.accuracy_score(y_test, y_test_pre))
```

```
PCA-Logistic Regression Accuracy 0.9973129285726808 0.9973048513878211
```

```
print(metrics.accuracy_score(y_train, y_train_pre))
print(metrics.accuracy_score(y_test, y_test_pre))
print(metrics.confusion_matrix(y_test, y_test_pre))
```

```
0.9973129285726808
0.9973048513878211
[[828563     96]
 [  2144    320]]
```

그 결과, 약 99.73%의 test accuracy로 굉장히 높은 비율의 정확도를 얻었으며, confusion matrix를 살펴보니 test data 중 2144건의 사기 거래를 정상거래로 잘못 분류했음을 확인할 수 있었습니다.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=pipe_lr, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
import numpy as np
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99717422 0.99718454 0.99727735 0.99726704 0.99737017 0.99725157
 0.99720516 0.99719484 0.99729281 0.99727733]
CV accuracy: 0.997 +/- 0.000
```

이어 10-fold Cross Validation으로 해당 모델의 classification accuracy rate을 좀 더 자세히 살펴본 결과, 본 모델의 정확도는 0.997로 보는 것이 알맞다고 판단하였습니다.

3.2. Linear Discriminant Analysis

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
cld=LinearDiscriminantAnalysis(store_covariance=True)
cld.fit(X_train, y_train)
y_train_pred=cld.predict(X_train)
y_test_pred=cld.predict(X_test)
```

```
print("LDA Accuracy",metrics.accuracy_score(y_train, y_train_pred),metrics.accuracy_score(y_test, y_test_pred))
```

```
LDA Accuracy 0.9979337756266997 0.9979629970533844
```

```
print(metrics.confusion_matrix(y_test, y_test_pred))
```

```
[[828491    168]
 [  1525    939]]
```

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(),
                     LinearDiscriminantAnalysis(store_covariance=True))
```

```
scores = cross_val_score(estimator=pipe, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99782395 0.99792192 0.99808177 0.99794255 0.99796833 0.99788067
 0.99792192 0.99787034 0.99794254 0.99799409]
CV accuracy: 0.998 +/- 0.000
```

이번엔 주어진 데이터를 LDA모델에 적합해보았습니다. 그 결과, 약 99.8%의 test accuracy로 앞서 구한 PCA모델보다 조금 더 높은 비율의 정확도를 얻었으며, Confusion matrix를 살펴보니 test data 중 정상 거래를 fraud로 오분류한 경우의 수가 위 PCA-Logistic 모델보단 약 두배정도 많으나, 사기 거래를 정상거래로 잘못 분류한 경우의 수는 확연히 줄었음을 확인할 수 있었습니다.

3.2.1. LDA로 차원 축소 후 로지스틱 회귀

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda=LDA(n_components=3)
X_train_lda=lda.fit_transform(X_train_std,y_train)
X_test_lda=lda.transform(X_test_std)
lr=LogisticRegression()
lr.fit(X_train_lda, y_train)
y_train_pred=lr.predict(X_train_lda)
y_test_pred=lr.predict(X_test_lda)
```

```
print("LDA-Logistic Regression Accuracy",accuracy_score(y_train, y_train_pred),accuracy_score(y_test, y_test_pred))
```

```
LDA-Logistic Regression Accuracy 0.9979188216694186 0.9979461523745583
```

```
print(confusion_matrix(y_test, y_test_pred))
```

```
[[828465    194]
 [  1513    951]]
```

앞서 PCA를 통해 차원 축소 후 로지스틱 회귀를 한 것을 변형하여, 이번엔 LDA로 데이터 차원을 축소시킨 후 로지스틱 회귀모형을 적용해보았습니다. 그 결과, 앞서 시도한 두 모형보다 더 높은 정확도를 얻을 수 있었습니다. Confusion matrix를 통해 test data 중 총 1513건의 사기거래를 정상거래로 잘못 분류했음을 확인할 수 있었습니다.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=pipe_lr, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
import numpy as np
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99779816 0.99790645 0.9980663  0.99791676 0.99794255 0.9978652
 0.99789614 0.99788581 0.99793222 0.99797346]
CV accuracy: 0.998 +/- 0.000
```

이어서 10-fold Cross Validation으로 classification accuracy rate을 좀 더 자세히 살펴본 결과, 본 모델의 정확도를 0.998로, PCA-Logistic 모델보다 약 0.1% 향상된 결과를 나타낸다고 결론지었습니다.

3.3. Random Forest

```
parametersRF = {'n_estimators':15,'oob_score':True,'class_weight': "balanced",'n_jobs':-1,'random_state':42}
RF = RandomForestClassifier(**parametersRF)
RF.fit(X_train, y_train)
y_train_pred=RF.predict(X_train)
y_test_pred = RF.predict(X_test)
```

```
print("Random Forest Accuracy",accuracy_score(y_train, y_train_pred),accuracy_score(y_test, y_test_pred))
print(confusion_matrix(y_test, y_test_pred))
```

```
Random Forest Accuracy 0.9999623572799473 0.9992708660450981
[[828500     59]
 [   547   1917]]
```

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(),
                     RandomForestClassifier(**parametersRF))
```

```
scores = cross_val_score(estimator=pipe, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99929871 0.99925746 0.99927809 0.9992884  0.99934512 0.99926262
 0.99938637 0.99920073 0.99927808 0.99928839]
CV accuracy: 0.999 +/- 0.000
```

다음은 Random Forest모델을 이용한 classification 결과입니다. CV accuracy가 0.999로, 잘못 분류된 거래의 수가 다른 모델들에 비해 상당히 작으며, 앞서 LDA로 분류했을 때보다 0.1% 향상된 정확도를 확인할 수 있었습니다. 특히, 앞 모델보다 사기거래를 일반거래로 잘못 분류할 경우의 수가 절반 이상 줄어듦이 인상적이었습니다.

3.4. XGB

우선 GridSearch를 사용하여 XGB 모델에서 사용될 hyperparameter인 max_depth과 learning_rate의 optimal 값을 찾았습니다.

```
from sklearn.model_selection import GridSearchCV
param= {'max_depth': range (2, 5,1),'learning_rate': [0.1, 0.01]}
xgsearch=GridSearchCV(estimator = XGBClassifier(objective= 'binary:logistic', nthread=4, seed=42),param_grid=param,scoring = 'roc_auc', n_j
xgsearch.fit(X_train,y_train)
xgsearch.best_params_
```

```
{'learning_rate': 0.1, 'max_depth': 4}
```

이렇게 구한 parameter 값을 XGBClassifier 모델에 넣어 데이터를 적합시킨 결과, Random Forest 모형과 같은 99.9%의 아주 높은 정확도를 얻을 수 있었습니다. Confusion matrix를 확인해보니, 일반 거래를 사기 거래로 잘못 분류하는 경우의 수가 특히 아주 낮게 나옴을 확인할 수 있었습니다. 아래는 그 결과입니다.

```
xgb_model = XGBClassifier(max_depth=4,seed=17,learning_rate=0.1,random_state=42)
xgb_model.fit(X_train, y_train)
y_train_pred=xgb_model.predict(X_train)
y_test_pred = xgb_model.predict(X_test)
```

```
print("XGB Accuracy",accuracy_score(y_train, y_train_pred),accuracy_score(y_test, y_test_pred))
print(confusion_matrix(y_test, y_test_pred))
```

```
XGB Accuracy 0.9992254881435745 0.9992107064778619
[[828632     27]
 [   629   1835]]
```

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(),
                     XGBClassifier(max_depth=3,seed=17,learning_rate=0.1,random_state=42))
```

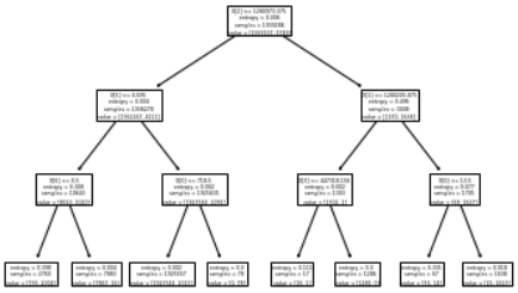
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=pipe, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99927293 0.9991698  0.99921621 0.99918011 0.99926262 0.9991698
 0.99928324 0.99912339 0.99913886 0.99923167]
CV accuracy: 0.999 +/- 0.000
```

3.5. Decision Tree

```
from sklearn import tree
dtc = tree.DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=1)
dtc.fit(X_train, y_train)
y_train_pred = dtc.predict(X_train)
y_test_pred = dtc.predict(X_test)
```

```
tree.plot_tree(dtc.fit(X_train,y_train))
```



```
print("Decision Tree Accuracy",accuracy_score(y_train, y_train_pred),accuracy_score(y_test, y_test_pred))
print(metrics.confusion_matrix(y_test, y_test_pred))
```

```
Decision Tree Accuracy 0.9985484348363264 0.998541732090196
[[828304    355]
 [   857   1607]]
```

```
from sklearn import tree
pipe = make_pipeline(StandardScaler(),
                     tree.DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=1))
```

```
scores = cross_val_score(estimator=pipe, X=X_train, y=y_train, cv=10)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [0.99858711 0.99858196 0.99859742 0.99862321 0.99859742 0.99846336
 0.99865415 0.99841178 0.99839116 0.99856647]
CV accuracy: 0.999 +/- 0.000
```

마지막으로, Decision Tree모델을 통해 데이터를 분류해보았습니다. 그 결과, 앞서 다룬 Random Forest모델이나 XGB모델과 비슷한 약 99.9%의 높은 정확도를 확인할 수 있었습니다.

4. Conclusion

이제 앞서 제시한 다양한 classification 모델 중 주어진 데이터에서 금융사기를 가장 잘 분류할 수 있는 모형을 찾기 위하여, 각 모델의 다양한 classification score를 비교하겠습니다.

아래는 각 모델의 classification report입니다.

PCA-Logistic

```
print(classification_report(y_test,y_test_pre))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.70      0.11      0.19      2464

    accuracy                           1.00    831123
   macro avg       0.85      0.56      0.60    831123
weighted avg       1.00      1.00      1.00    831123
```

LDA

```
print(classification_report(y_test,y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.85      0.38      0.53      2464

    accuracy                           1.00    831123
   macro avg       0.92      0.69      0.76    831123
weighted avg       1.00      1.00      1.00    831123
```

LDA-Logistic

```
print(classification_report(y_test,y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.83      0.39      0.53      2464

    accuracy                           1.00    831123
   macro avg       0.91      0.69      0.76    831123
weighted avg       1.00      1.00      1.00    831123
```

Random Forest

```
print(classification_report(y_test,y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.97      0.78      0.86      2464

    accuracy                           1.00    831123
   macro avg       0.98      0.89      0.93    831123
weighted avg       1.00      1.00      1.00    831123
```

XGB

```
print(classification_report(y_test,y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.99      0.74      0.85      2464

    accuracy                           1.00    831123
   macro avg       0.99      0.87      0.92    831123
weighted avg       1.00      1.00      1.00    831123
```

Decision Tree

```
print(classification_report(y_test,y_test_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    828659
           1       0.82      0.65      0.73      2464

    accuracy                           1.00    831123
   macro avg       0.91      0.83      0.86    831123
weighted avg       1.00      1.00      1.00    831123
```

앞서 Model Selection 과정에서 각 모델의 CV accuracy score를 비교해보면, 모든 모델이 99.8%에서 99.9%의 높은 정확도를 보였으므로, 이를 토대로 모형의 우열을 가리는 것은 힘들었습니다. 하지만, 모델의 최종목적이 fraud detection임을 고려했을 때, 사기거래를 일반거래로 잘못 분류하는 경우의 수가 가장 적은 모델이 선호될 것이며, 따라서 해당 오분류 수가 가장 작은 Random Forest모델과 XGB모델이 가장 적절하다고 판단하였습니다. 또한, 각 모델의 classification report 수치를 비교해보면, Random Forest모델과 XGB모델이 여섯 모델 중 가장 우수한 성능을 보임을 확인할 수 있었습니다. 그렇기에, 주어진 데이터에 대한 classification을 진행한 결과 fraud detection에 통계적으로 유의미한 변수는 'step', 'type', 'amount', 'oldbalanceOrig', 'newbalanceOrig', 'oldbalanceDest', 'oldbalanceDest'이며, 다양한 classification 모델 중 종속변수 'isFraud'를 가장 잘 분류할 수 있는 모형은 Random Forest와 XGB모델이라고 결론지었습니다.