

Chapter 21. CREDIT CARD FRAUD DETECTION



문제 - 데이터 소개

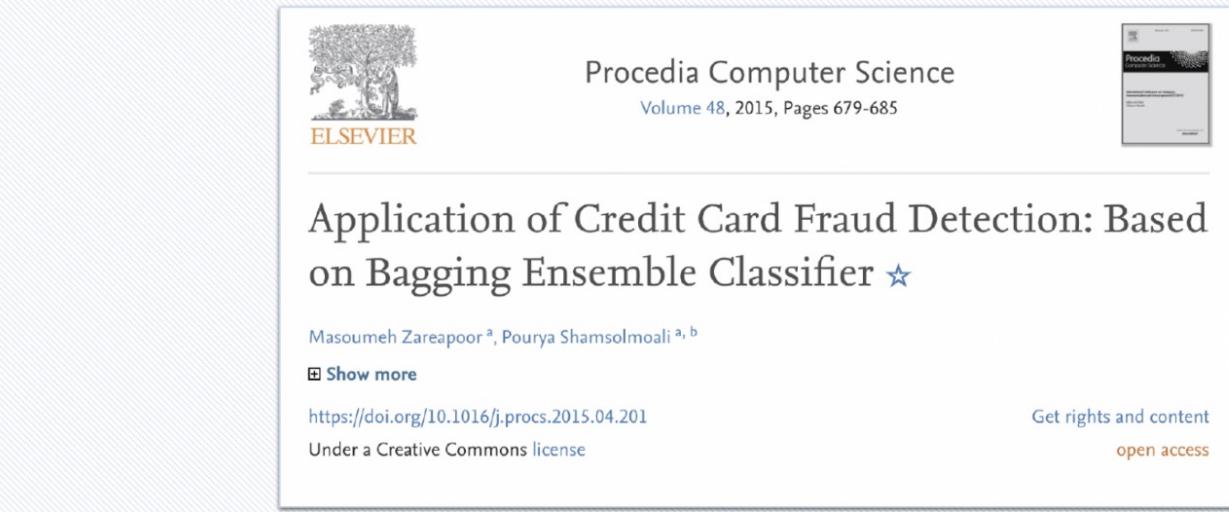
문제 - 데이터 소개

신용카드 부정 사용자 검출



문제 - 데이터 소개

신용카드 부정사용 검출에 관한 논문의 한 예

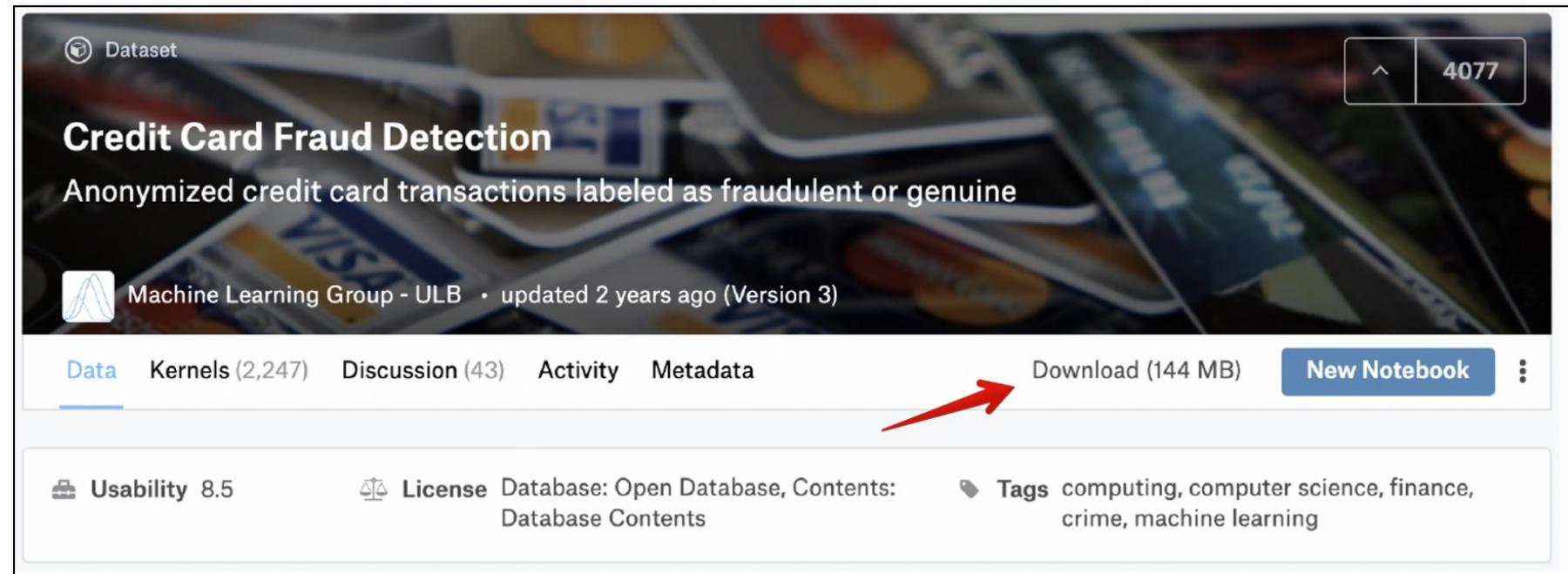


The screenshot shows a research paper titled "Application of Credit Card Fraud Detection: Based on Bagging Ensemble Classifier" published in Procedia Computer Science, Volume 48, 2015, pages 679-685. The paper is authored by Masoumeh Zareapoor and Pourya Shamsolmoali. It is available under a Creative Commons license and can be accessed via DOI: <https://doi.org/10.1016/j.procs.2015.04.201>. The journal logo features a tree and the Elsevier name.

- 신용카드와 같은 금융 데이터들은 구하기가 어려움
- 금융 데이터들의 데이터는 또한 다루기 쉽지 않음
- 그러나 지능화되어가는 현대 범죄에 맞춰 사전 이상 징후 검출 등 금융 기관이 많은 노력을 기울이고 있음
- 이 데이터 역시 센서를 이용한 사람의 행동 과정 유추처럼 머신러닝의 이용 분야 중에 하나임

문제 - 데이터 소개

데이터 다운로드



- <https://www.kaggle.com/MLG-ULB/CREDITCARDFRAUD>
- 데이터 받은 후 압축을 풀고 소스코드 폴더에 옮겨 두자

문제 - 데이터 소개

데이터 개요

-  신용카드 사기 검출 분류 실습용 데이터
-  데이터에 class라는 이름의 컬럼이 사기 유무를 의미
-  class 컬럼의 불균형이 극심해서 전체 데이터의 약 0.172%가 1 (사기 Fraud)을 가짐

문제 - 데이터 소개

데이터 특성

v13	v14	v15	v16	v17	v18	v19	v20	v21
-0.991390	-0.311169	1.468177	-0.470401	0.207971	0.025791	0.403993	0.251412	-0.018307
0.489095	-0.143772	0.635558	0.463917	-0.114805	-0.183361	-0.145783	-0.069083	-0.225775
0.717293	-0.165946	2.345865	-2.890083	1.109969	-0.121359	-2.261857	0.524980	0.247998
0.507757	-0.287924	-0.631418	-1.059647	-0.684093	1.965775	-1.232622	-0.208038	-0.108300
1.345852	-1.119670	0.175121	-0.451449	-0.237033	-0.038195	0.803487	0.408542	-0.009431

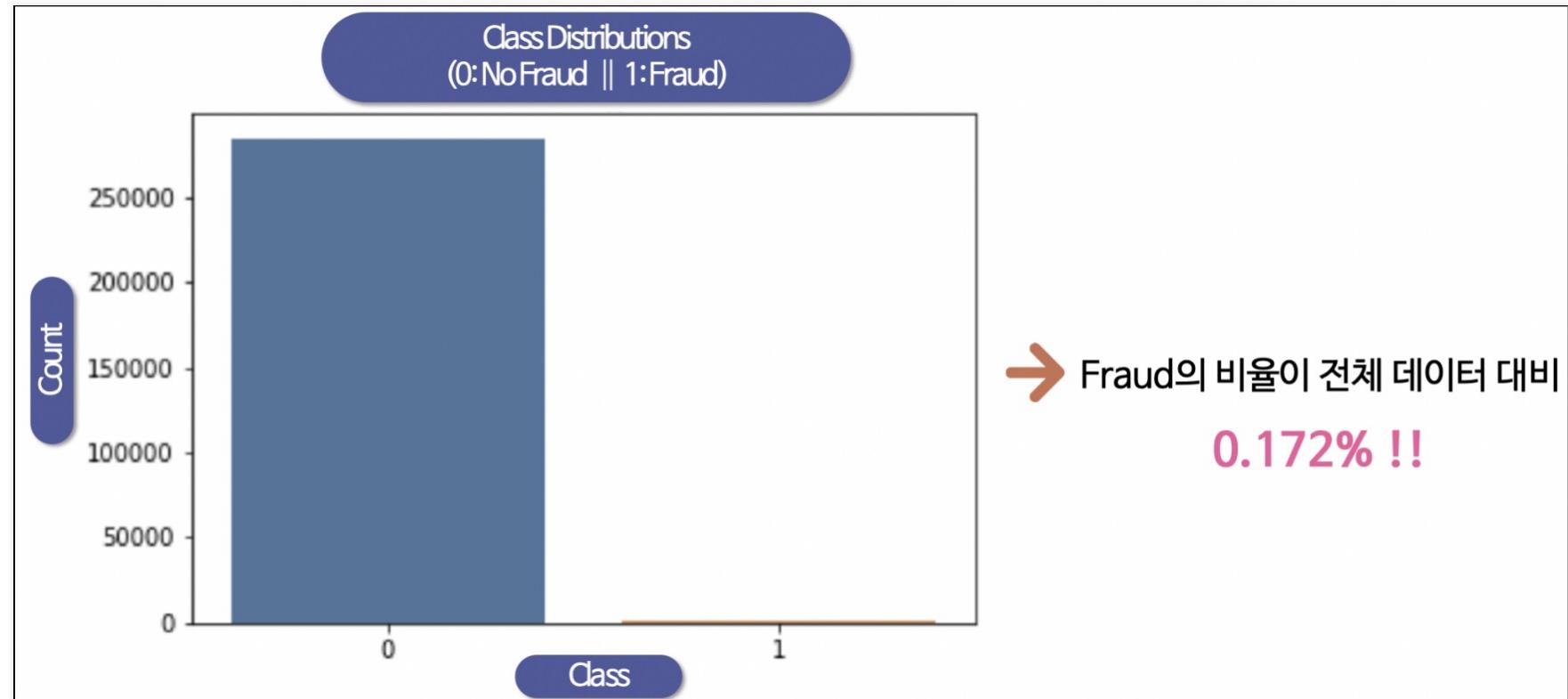
◆ 금융 데이터이고 기업의 기밀 보호를 위해 대다수 특성의 이름은 삭제되어 있음 ◆

→ Amount : 거래금액

→ Class : Fraud 여유 (1이면 Fraud)

문제 - 데이터 소개

데이터의 불균형이 극심함



데이터 읽고 관찰하기

데이터 읽고 관찰하기

데이터 읽기

```
import pandas as pd

data_path = './creditcard.csv'
raw_data = pd.read_csv(data_path)
raw_data.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V2'
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.00943

5 rows × 31 columns

데이터 읽고 관찰하기

특성

```
raw_data.columns
```

```
Index(['Time', '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',
       'Class'],
      dtype='object')
```

- 데이터의 특성은 여러 이유로 이름이 감춰져 있다.

데이터 읽고 관찰하기

데이터 라벨의 불균형이 정말 심하다

```
raw_data['Class'].value_counts()
```

```
0    284315  
1      492  
Name: Class, dtype: int64
```

```
frauds_rate = round(raw_data['Class'].value_counts()[1]/len(raw_data) * 100, 2)  
print('Frauds', frauds_rate, '% of the dataset')
```

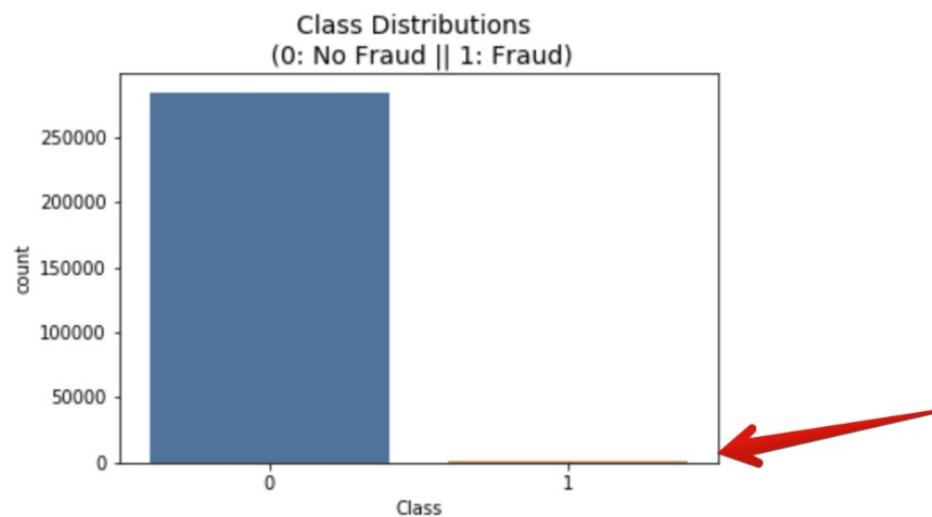
```
Frauds 0.17 % of the dataset
```

- 17%가 아니다, 0.17%이다.

데이터 읽고 관찰하기

그래프로 표현되기도 힘들다

```
import seaborn as sns  
import matplotlib.pyplot as plt  
  
sns.countplot('Class', data=raw_data)  
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=14)  
plt.show()
```



데이터 읽고 관찰하기

일단 X, y로 데이터 선정

```
| X = raw_data.iloc[:, 1:-1]  
y = raw_data.iloc[:, -1]
```

```
X.shape, y.shape
```

```
((284807, 29), (284807,))
```

데이터 읽고 관찰하기

데이터를 나누고

```
| from sklearn.model_selection import train_test_split  
  
X_train, X_test, y_train, y_test =  
    train_test_split(X, y, test_size=0.3,  
                    random_state=13, stratify=y)
```

데이터 읽고 관찰하기

나눈 데이터의 불균형 정도가 어떤지 확인해보자

```
| import numpy as np  
  
np.unique(y_train, return_counts=True)  
  
(array([0, 1]), array([199020,      344]))  
  
| tmp = np.unique(y_train, return_counts=True)[1]  
tmp[1]/len(y_train) * 100  
  
0.17254870488152324
```

데이터 읽고 관찰하기

```
| np.unique(y_test, return_counts=True)  
|  
|     (array([0, 1]), array([85295,    148]))  
  
| tmp = np.unique(y_test, return_counts=True)[1]  
| tmp[1]/len(y_test) * 100  
|  
| 0.17321489179921118
```

단순 무식한 첫 도전 - 1st Trial

단순 무식한 첫 도전
- 1st Trial

먼저 분류기의 성능을 return하는 함수 하나 작성

```
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score)

def get_clf_eval(y_test, pred):
    acc = accuracy_score(y_test, pred)
    pre = precision_score(y_test, pred)
    re = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)
    auc = roc_auc_score(y_test, pred)

    return acc, pre, re, f1, auc
```

- 이런 식으로 함수화 해두면 나중에 따로 사용하도록 해둘 수 있다.

단순 무식한 첫 도전
- 1st Trial

또 성능을 출력하는 함수 하나 작성

```
from sklearn.metrics import confusion_matrix

def print_clf_eval(y_test, pred):
    confusion = confusion_matrix(y_test, pred)
    acc, pre, re, f1, auc = get_clf_eval(y_test, pred)

    print('=> confusion matrix')
    print(confusion)
    print('=====')

    print('Accuracy: {:.4f}, Precision: {:.4f}'.format(acc, pre))
    print('Recall: {:.4f}, F1: {:.4f}, AUC:{:.4f}'.format(re, f1, auc))
```

단순 무식한 첫 도전
- 1st Trial

Logistic Regression

```
| from sklearn.linear_model import LogisticRegression  
  
lr_clf = LogisticRegression(random_state=13, solver='liblinear')  
lr_clf.fit(X_train, y_train)  
lr_pred = lr_clf.predict(X_test)  
  
print_clf_eval(y_test, lr_pred)  
  
=> confusion matrix  
[[85284    11]  
 [   60    88]]  
=====  
Accuracy: 0.9992, Precision: 0.8889  
Recall: 0.5946, F1: 0.7126, AUC:0.7972
```

단순 무식한 첫 도전
- 1st Trial

Decision Tree

```
| from sklearn.tree import DecisionTreeClassifier  
  
dt_clf = DecisionTreeClassifier(random_state=13, max_depth=4)  
dt_clf.fit(X_train, y_train)  
dt_pred = dt_clf.predict(X_test)  
  
print_clf_eval(y_test, dt_pred)  
  
=> confusion matrix  
[[85281    14]  
 [    42   106]]  
=====  
Accuracy: 0.9993, Precision: 0.8833  
Recall: 0.7162, F1: 0.7910, AUC:0.8580
```

단순 무식한 첫 도전
- 1st Trial

Random Forest

```
| from sklearn.ensemble import RandomForestClassifier  
  
rf_clf = RandomForestClassifier(random_state=13, n_jobs=-1, n_estimators=100)  
rf_clf.fit(X_train, y_train)  
rf_pred = rf_clf.predict(X_test)  
  
print_clf_eval(y_test, rf_pred)  
  
=> confusion matrix  
[[85290      5]  
 [    38     110]]  
=====  
Accuracy: 0.9995, Precision: 0.9565  
Recall: 0.7432, F1: 0.8365, AUC:0.8716
```

단순 무식한 첫 도전
- 1st Trial

LightGBM

```
from lightgbm import LGBMClassifier

lgbm_clf = LGBMClassifier(n_estimators=1000, num_leaves=64, n_jobs=-1,
                          boost_from_average=False)
lgbm_clf.fit(X_train, y_train)
lgbm_pred = lgbm_clf.predict(X_test)

print_clf_eval(y_test, lgbm_pred)

=> confusion matrix
[[85288    7]
 [   35   113]]
=====
Accuracy: 0.9995, Precision: 0.9417
Recall: 0.7635, F1: 0.8433, AUC:0.8817
```

단순 무식한 첫 도전
- 1st Trial

여기서 Recall과 Precision의 의미는

- 은행 입장에서는 Recall이 좋을 것이다.
- 사용자 입장에서는 Precision이 좋겠지.
- 왜?

한걸음 전진~

한걸음 전진~

모델과 데이터를 주면 성능을 출력하는 함수를 하나 만들자

```
| def get_result(model, X_train, y_train, X_test, y_test):
|     model.fit(X_train, y_train)
|     pred = model.predict(X_test)
|
|     return get_clf_eval(y_test, pred)
```

한걸음 전진~

다수의 모델의 성능을 정리해서 DataFrame으로 반환하는 함수 작성

```
def get_result_pd(models, model_names, X_train, y_train, X_test, y_test):
    col_names = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']
    tmp = []

    for model in models:
        tmp.append(get_result(model, X_train, y_train, X_test, y_test))

    return pd.DataFrame(tmp, columns=col_names, index=model_names)
```

한걸음 전진~

4개의 분류 모델을 한 번에 표로 정리해보자

```
import time

models = [lr_clf, dt_clf, rf_clf, lgbm_clf]
model_names = ['LinearReg.', 'DecisionTree', 'RandomForest','LightGBM']

start_time = time.time()
results = get_result_pd(models, model_names, X_train, y_train, X_test, y_test)

print('Fit time : ', time.time() - start_time)
results
```

한걸음 전진~

확실히 양상블 계열의 성능이 우수하다

Fit time : 29.66269588470459

	accuracy	precision	recall	f1	roc_auc
LinearReg.	0.999169	0.888889	0.594595	0.712551	0.797233
DecisionTree	0.999345	0.883333	0.716216	0.791045	0.858026
RandomForest	0.999497	0.956522	0.743243	0.836502	0.871592
LightGBM	0.999508	0.941667	0.763514	0.843284	0.881716

데이터를 정리해서 다시 도전하자 - 2nd Trial

데이터를 정리해서
다시 도전하자
- 2nd Trial

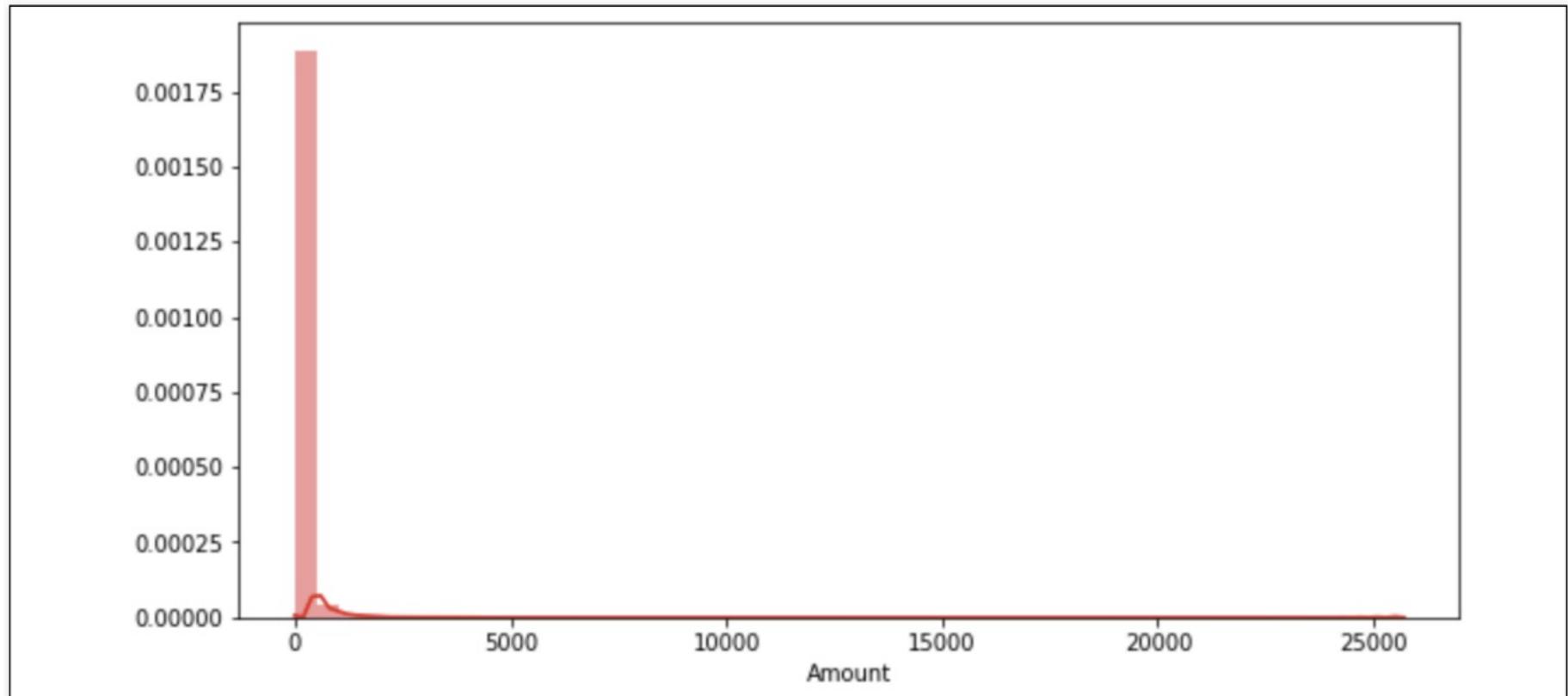
raw_data의 Amount 컬럼 확인

```
| plt.figure(figsize=(10, 5))  
| sns.distplot(raw_data['Amount'], color='r')  
  
plt.show()
```

- Amount는 신용카드 사용금액

데이터를 정리해서
다시 도전하자
- 2nd Trial

컬럼의 분포가 특정 대역이 아주 많다



데이터를 정리해서
다시 도전하자
- 2nd Trial

Amount 컬럼에 StandardScaler 적용

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
amount_n = scaler.fit_transform(raw_data['Amount'].values.reshape(-1, 1))

raw_data_copy = raw_data.iloc[:, 1:-2]
raw_data_copy['Amount_Scaled'] = amount_n
raw_data_copy.head()
```

10	...	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amount_Scaled
94	...	0.251412	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	0.244964
74	...	-0.069083	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	-0.342475
43	...	0.524980	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	1.160686
52	...	-0.208038	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	0.140534
4	...	0.408542	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	-0.073403

데이터를 정리해서
다시 도전하자
- 2nd Trial

데이터를 다시 나누고

```
| X_train, X_test, y_train, y_test =\  
|     train_test_split(raw_data_copy, y, test_size=0.3,  
|                         random_state=13, stratify=y)
```

데이터를 정리해서
다시 도전하자
- 2nd Trial

모델에 다시 평가를 해보면

```
models = [lr_clf, dt_clf, rf_clf, lgbm_clf]
model_names = ['LinearReg.', 'DecisionTree', 'RandomForest', 'LightGBM']

start_time = time.time()
results = get_result_pd(models, model_names, X_train, y_train, X_test, y_test)

print('Fit time : ', time.time() - start_time)
results
```

데이터를 정리해서
다시 도전하자
- 2nd Trial

결과

Fit time : 27.36718487739563

	accuracy	precision	recall	f1	roc_auc
LinearReg.	0.999169	0.888889	0.594595	0.712551	0.797233
DecisionTree	0.999345	0.883333	0.716216	0.791045	0.858026
RandomForest	0.999497	0.956522	0.743243	0.836502	0.871592
LightGBM	0.999520	0.942149	0.770270	0.847584	0.885094

데이터를 정리해서
다시 도전하자
- 2nd Trial

모델별 ROC 커브

```
from sklearn.metrics import roc_curve

def draw_roc_curve(models, model_names, X_test, y_test):
    plt.figure(figsize=(10,10))

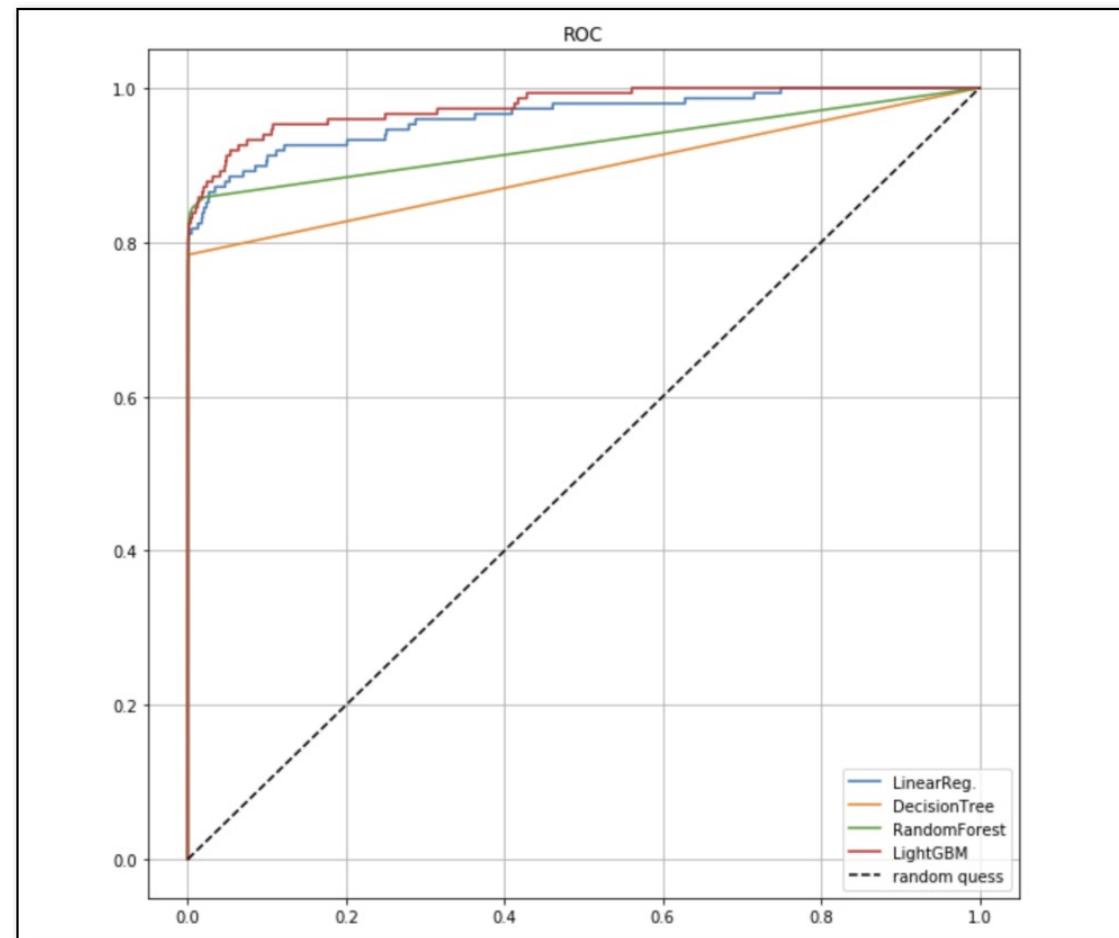
    for model in range(len(models)):
        pred = models[model].predict_proba(X_test)[:, 1]
        fpr, tpr, thresholds = roc_curve(y_test, pred)
        plt.plot(fpr, tpr, label=model_names[model])

    plt.plot([0,1], [0,1], 'k--', label='random quess')
    plt.title('ROC')
    plt.legend()
    plt.grid()
    plt.show()

draw_roc_curve(models, model_names, X_test, y_test)
```

데이터를 정리해서
다시 도전하자
- 2nd Trial

결과



데이터를 정리해서
다시 도전하자
- 2nd Trial

또 다른 시도 log scale

```
amount_log = np.log1p(raw_data['Amount'])

raw_data_copy['Amount_Scaled'] = amount_log
raw_data_copy.head()
```

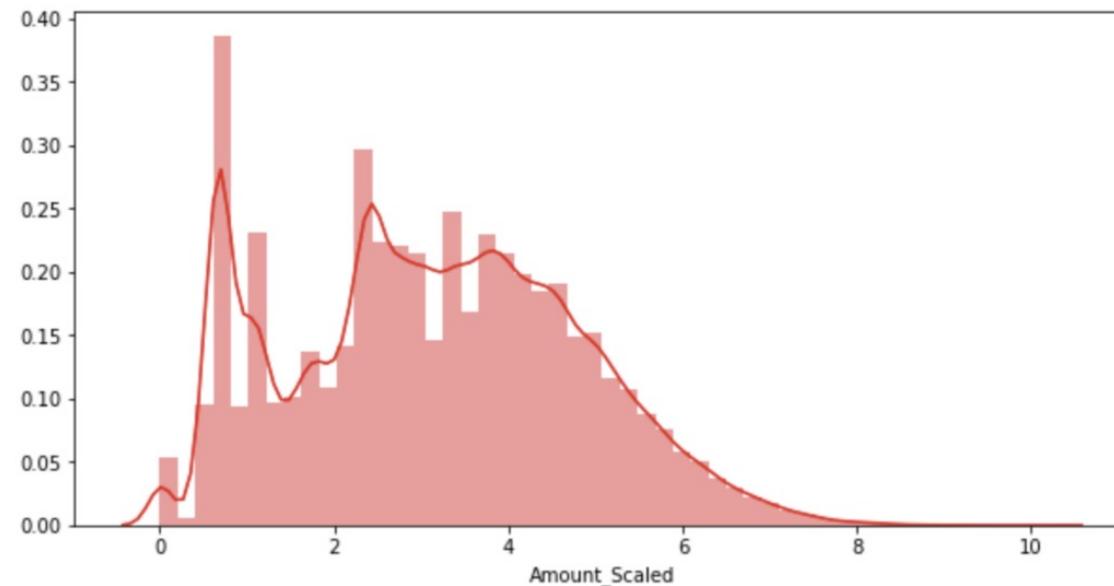
10	...	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amount_Scaled
94	...	0.251412	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	5.014760
74	...	-0.069083	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	1.305626
13	...	0.524980	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	5.939276
152	...	-0.208038	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	4.824306
14	...	0.408542	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	4.262539

데이터를 정리해서
다시 도전하자
- 2nd Trial

분포가 변화함

```
plt.figure(figsize=(10,5))
sns.distplot(raw_data_copy['Amount_Scaled'], color='r')

plt.show()
```



데이터를 정리해서
다시 도전하자
- 2nd Trial

다시 성능을 확인해보자

```
| X_train, X_test, y_train, y_test =\n|     train_test_split(raw_data_copy, y, test_size=0.3,\n|                         random_state=13, stratify=y)\n\nstart_time = time.time()\nresults = get_result_pd(models, model_names, X_train, y_train, X_test, y_test)\n\nprint('Fit time : ', time.time() - start_time)\nresults
```

데이터를 정리해서
다시 도전하자
- 2nd Trial

결과

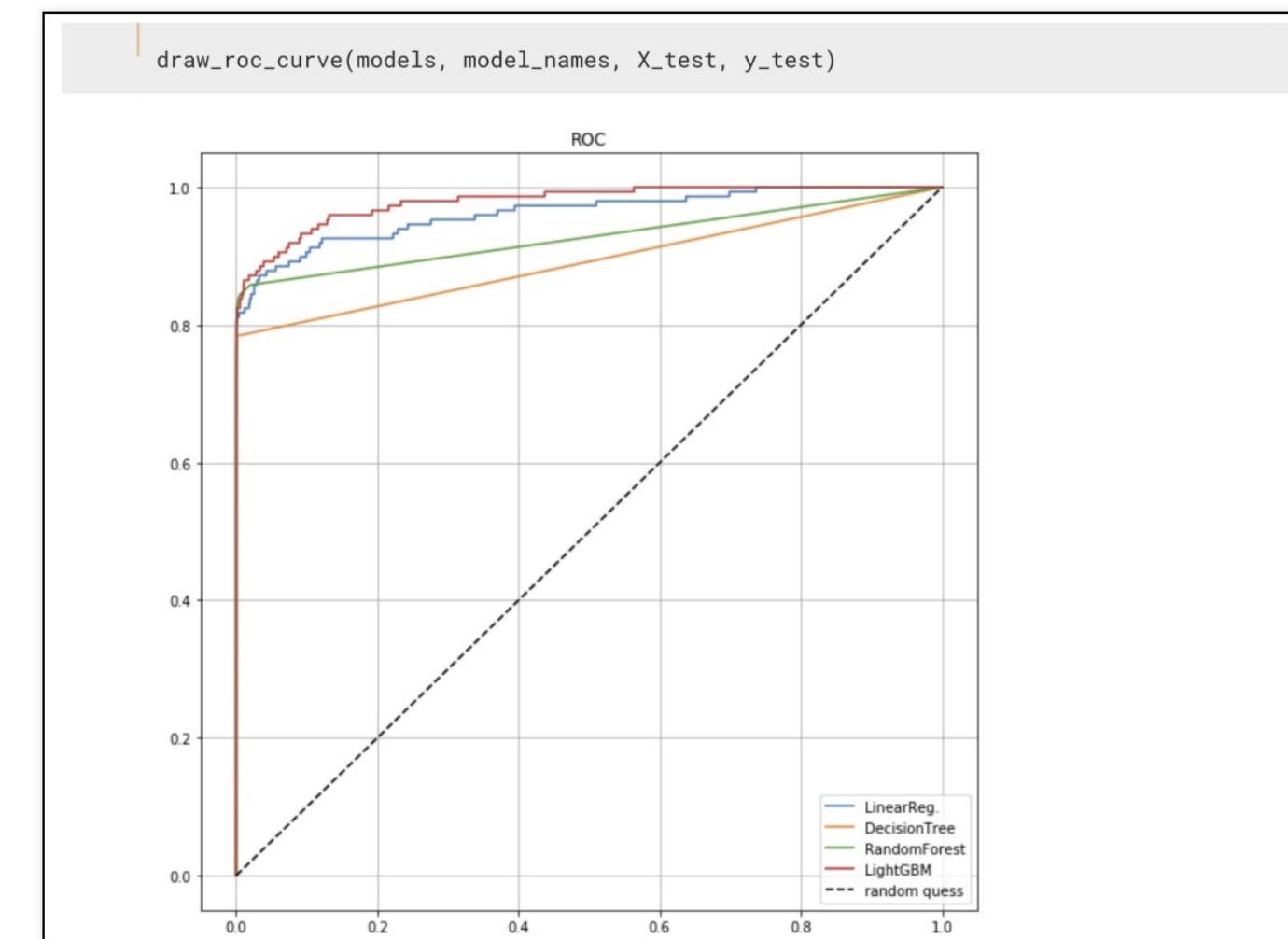
Fit time : 29.91777014732361

	accuracy	precision	recall	f1	roc_auc
LinearReg.	0.999157	0.887755	0.587838	0.707317	0.793854
DecisionTree	0.999345	0.883333	0.716216	0.791045	0.858026
RandomForest	0.999497	0.956522	0.743243	0.836502	0.871592
LightGBM	0.999508	0.941667	0.763514	0.843284	0.881716

- 미세한 변화가 보이지만 확실한 변화는 관찰되지 않는다.

데이터를 정리해서
다시 도전하자
- 2nd Trial

ROC 커브 결과



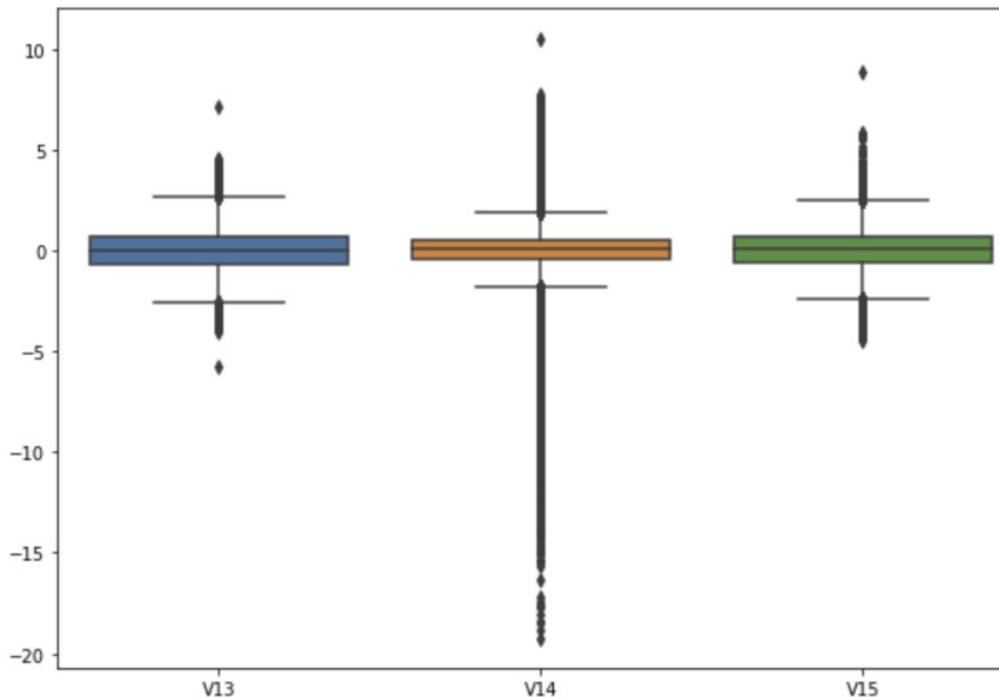
다시 데이터의 Outlier를 정리해보자

- 3rd Trial

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

특이 데이터

```
| import seaborn as sns  
  
plt.figure(figsize=(10, 7))  
sns.boxplot(data=raw_data[['V13', 'V14', 'V15']]);
```



다시 데이터의
Outlier를 정리해보자
- 3rd Trial

Outlier를 정리하기 위해 Outlier의 인덱스를 파악하는 코드

```
def get_outlier(df=None, column=None, weight=1.5):
    fraud = df[df['Class'] == 1][column]
    quantile_25 = np.percentile(fraud.values, 25)
    quantile_75 = np.percentile(fraud.values, 75)

    iqr = quantile_75 - quantile_25
    iqr_weight = iqr * weight
    lowest_val = quantile_25 - iqr_weight
    highest_val = quantile_75 + iqr_weight

    outlier_index = fraud[(fraud < lowest_val) | (fraud > highest_val)].index

    return outlier_index
```

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

Outlier 찾기

```
get_outlier(df=raw_data, column='V14', weight=1.5)
```

```
Int64Index([8296, 8615, 9035, 9252], dtype='int64')
```

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

Outlier 제거

```
raw_data_copy.shape
```

```
(284807, 29)
```

```
outlier_index = get_outlier(df=raw_data, column='V14', weight=1.5)
raw_data_copy.drop(outlier_index, axis=0, inplace=True)
raw_data_copy.shape
```

```
(284803, 29)
```

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

Outlier를 제거하고 데이터 나누기

```
X = raw_data_copy

raw_data.drop(outlier_index, axis=0, inplace=True)
y = raw_data.iloc[:, -1]

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.3,
                     random_state=13, stratify=y)
```

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

다시~

```
models = [lr_clf, dt_clf, rf_clf, lgbm_clf]
model_names = ['LinearReg.', 'DecisionTree', 'RandomForest', 'LightGBM']

start_time = time.time()
results = get_result_pd(models, model_names, X_train, y_train, X_test, y_test)

print('Fit time : ', time.time() - start_time)
results
```

다시 데이터의
Outlier를 정리해보자
- 3rd Trial

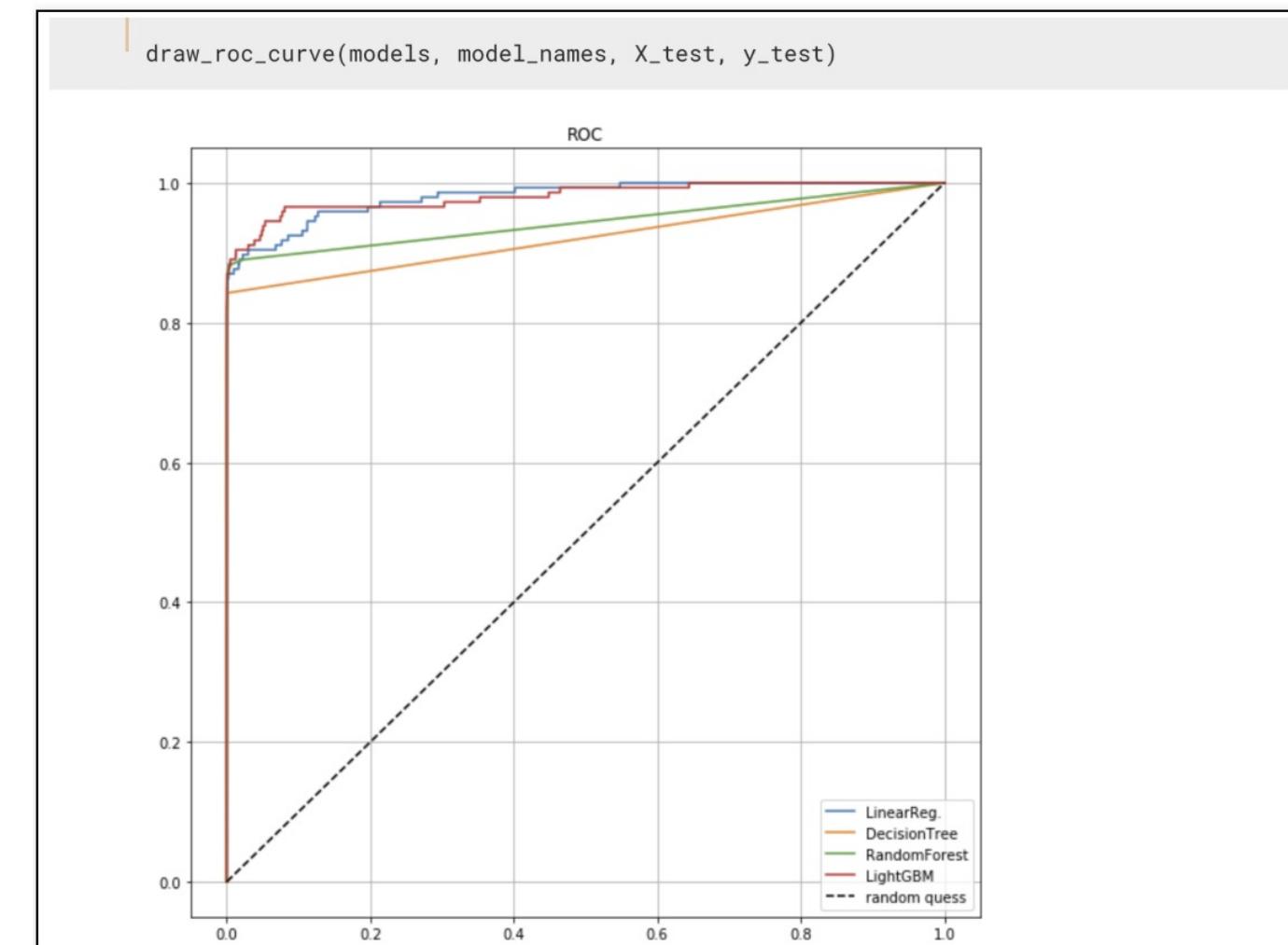
결과

Fit time : 29.531495809555054

	accuracy	precision	recall	f1	roc_auc
LinearReg.	0.999286	0.904762	0.650685	0.756972	0.825284
DecisionTree	0.999427	0.870229	0.780822	0.823105	0.890311
RandomForest	0.999497	0.918699	0.773973	0.840149	0.886928
LightGBM	0.999590	0.958678	0.794521	0.868914	0.897231

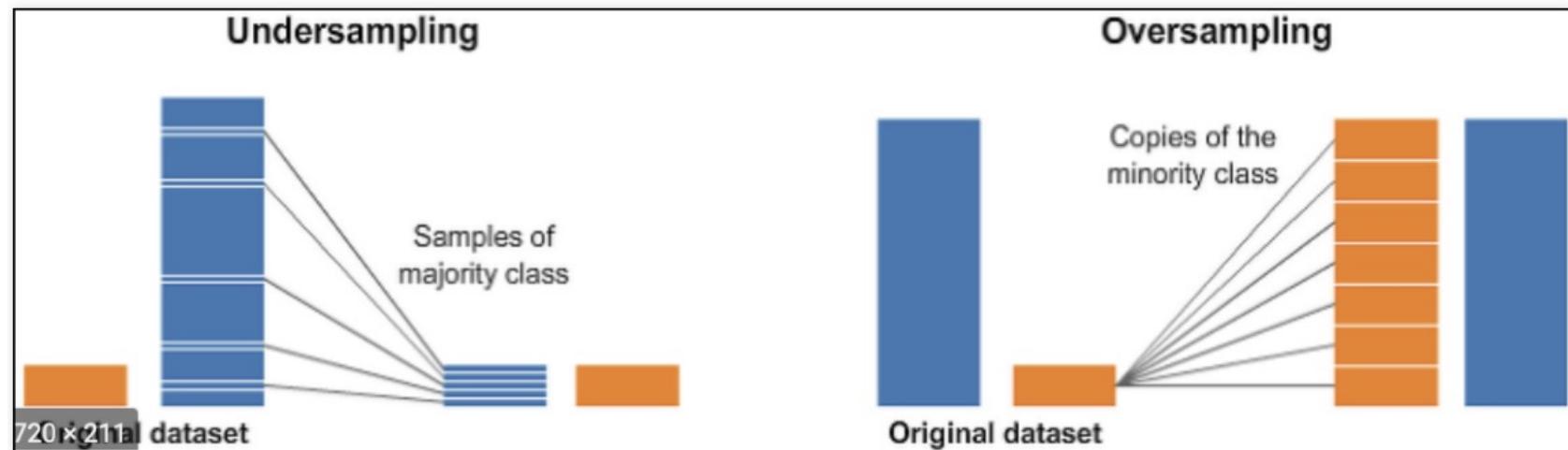
다시 데이터의
Outlier를 정리해보자
- 3rd Trial

ROC 커브



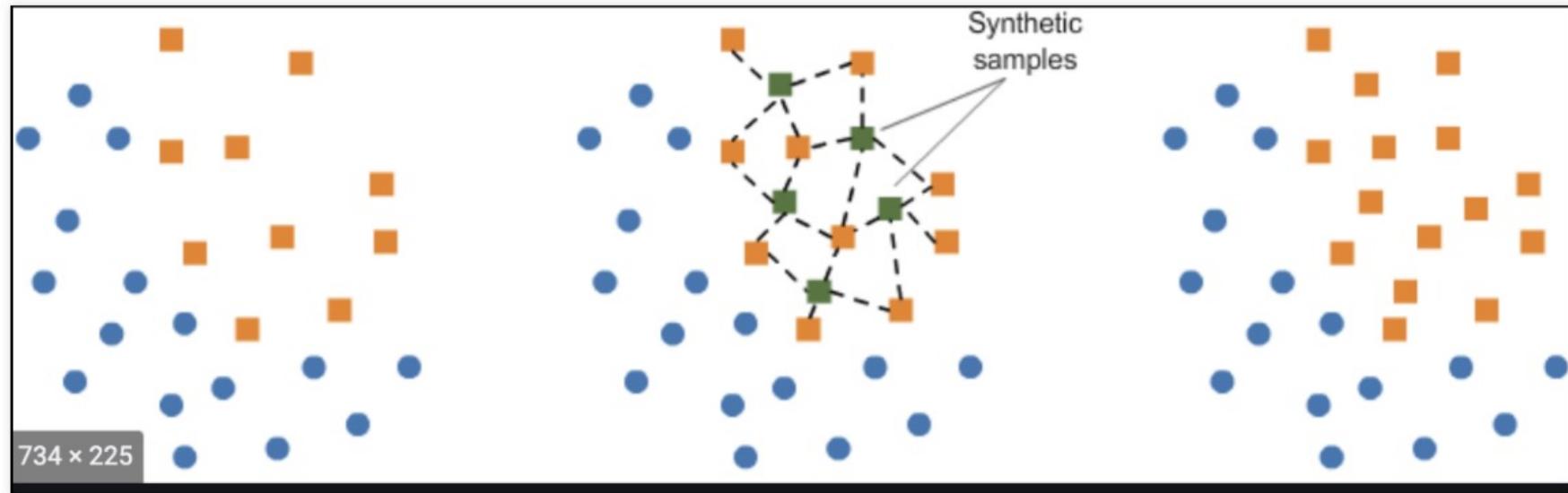
SMOTE Oversampling - 4th Trial

Undersampling vs Oversampling



- 데이터의 불균형이 극심할 때 불균형한 두 클래스의 분포를 강제로 맞춰보는 작업
- 언더샘플링 : 많은 수의 데이터를 적은 수의 데이터로 강제로 조정
- 오버샘플링 :
 - 원본데이터의 피처 값들을 아주 약간 변경하여 증식
 - 대표적으로 SMOTE(Synthetic Minority Over-sampling Technique) 방법이 있음
 - 적은 데이터 세트에 있는 개별 데이터를 k-최근접이웃 방법으로 찾아서 데이터의 분포 사이에 새로운 데이터를 만드는 방식
 - imbalanced-learn 이라는 Python pkg가 있음

SMOTE Oversampling
- 4th Trial



SMOTE Oversampling - 4th Trial

imbalanced-learn 설치

```
(fc) ~ ➔ pip install imbalanced-learn
Collecting imbalanced-learn
  Downloading imbalanced_learn-0.6.2-py3-none-any.whl (163 kB)
    ██████████ | 163 kB 226 kB/s
Requirement already satisfied: scikit-learn>=0.22 in ./opt/anaconda3/envs/fc/lib/python3.7/site-packages (from imbalanced-learn) (0.22.1)
Requirement already satisfied: numpy>=1.11 in ./opt/anaconda3/envs/fc/lib/python3.7/site-packages (from imbalanced-learn) (1.18.1)
Requirement already satisfied: scipy>=0.17 in ./opt/anaconda3/envs/fc/lib/python3.7/site-packages (from imbalanced-learn) (1.4.1)
Requirement already satisfied: joblib>=0.11 in ./opt/anaconda3/envs/fc/lib/python3.7/site-packages (from imbalanced-learn) (0.14.1)
Installing collected packages: imbalanced-learn
  • 언더샘플링 : 많은 수의 데이터를 적은 수의 데이터로 변환하는 기법
  • 오버샘플링 :
    ○ 원본데이터의 피처값들을 아주 약간
```

- pip install imbalanced-learn

SMOTE Oversampling - 4th Trial

SMOTE 적용

```
| from imblearn.over_sampling import SMOTE  
  
smote = SMOTE(random_state=13)  
X_train_over, y_train_over = smote.fit_sample(X_train, y_train)
```

SMOTE Oversampling - 4th Trial

데이터 증강 효과는

X_train.shape, y_train.shape

((199362, 29), (199362,))

X_train_over.shape, y_train_over.shape

((398040, 29), (398040,))

SMOTE Oversampling - 4th Trial

결과는?

```
| print(np.unique(y_train, return_counts=True))
| print(np.unique(y_train_over, return_counts=True))

(array([0, 1]), array([199020,      342]))
(array([0, 1]), array([199020, 199020]))
```

다시 학습을 돌려보자

```
| models = [lr_clf, dt_clf, rf_clf, lgbm_clf]
| model_names = ['LinearReg.', 'DecisionTree', 'RandomForest','LightGBM']
|
| start_time = time.time()
| results = get_result_pd(models, model_names,
|                         X_train_over, y_train_over, X_test, y_test)
|
| print('Fit time : ', time.time() - start_time)
| results
```

SMOTE Oversampling
- 4th Trial

조금 오래 걸린다

Fit time : 56.324262857437134

	accuracy	precision	recall	f1	roc_auc
LinearReg.	0.975609	0.059545	0.897260	0.111679	0.936502
DecisionTree	0.968984	0.046048	0.869863	0.087466	0.919509
RandomForest	0.999532	0.873239	0.849315	0.861111	0.924552
LightGBM	0.999427	0.834483	0.828767	0.831615	0.914243

- recall은 확실히 좋아진다.

SMOTE Oversampling
- 4th Trial

ROC 커브

```
draw_roc_curve(models, model_names, X_test, y_test)
```

