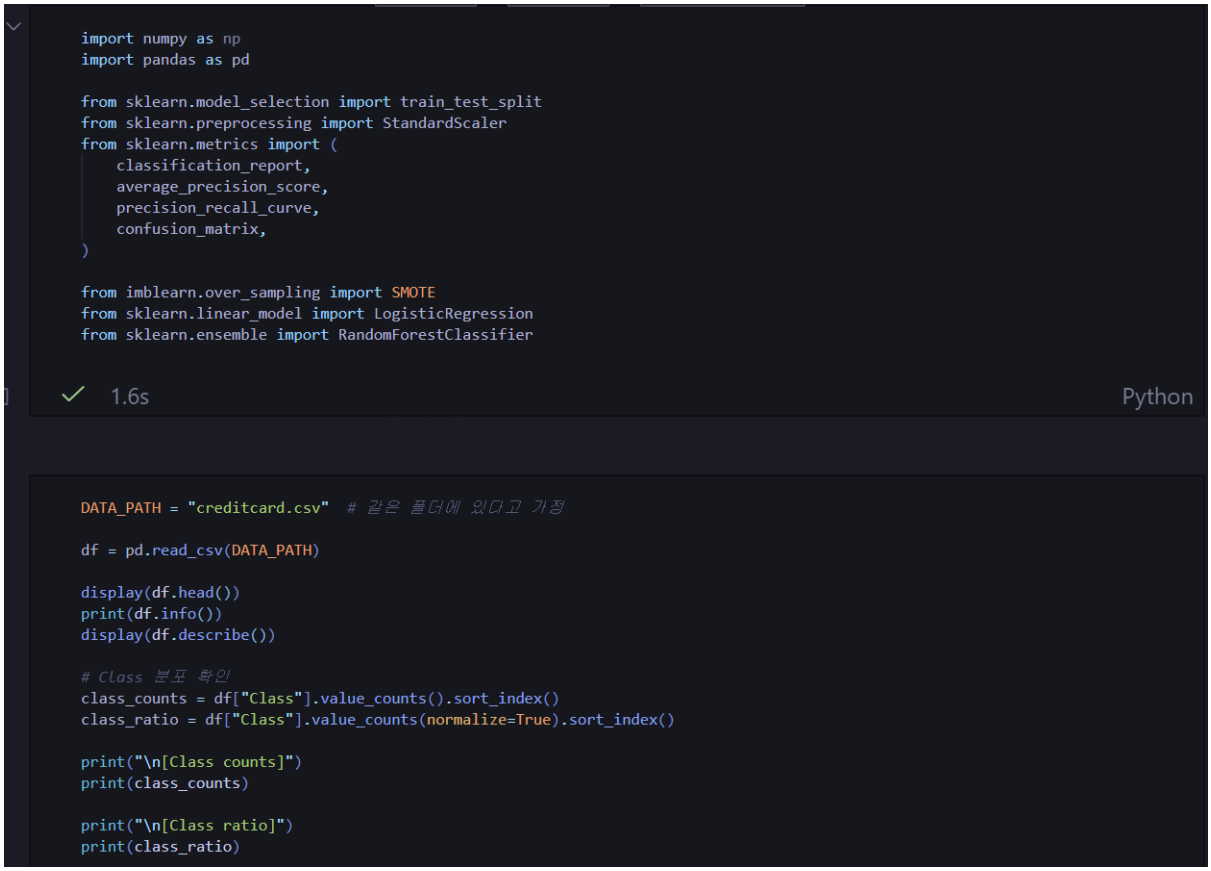
산업공학과 22학번 문형서 과제 보고서

00 데이터 로드 및 기본 탐색

creditcard.csv를 불러오고 데이터 구조를 확인하시오.

(예: head(), info(), describe(), Class 비율 출력)

정상 거래와 사기 거래 건수를 확인하시오.

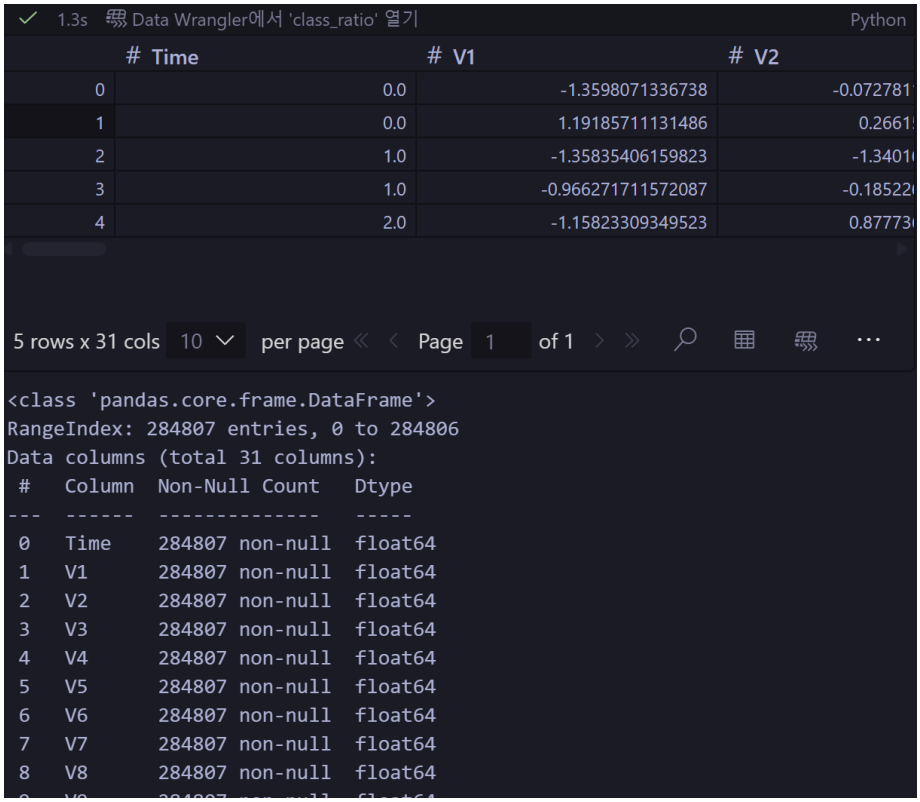


```python
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    classification_report,
    average_precision_score,
    precision_recall_curve,
    confusion_matrix,
)

from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

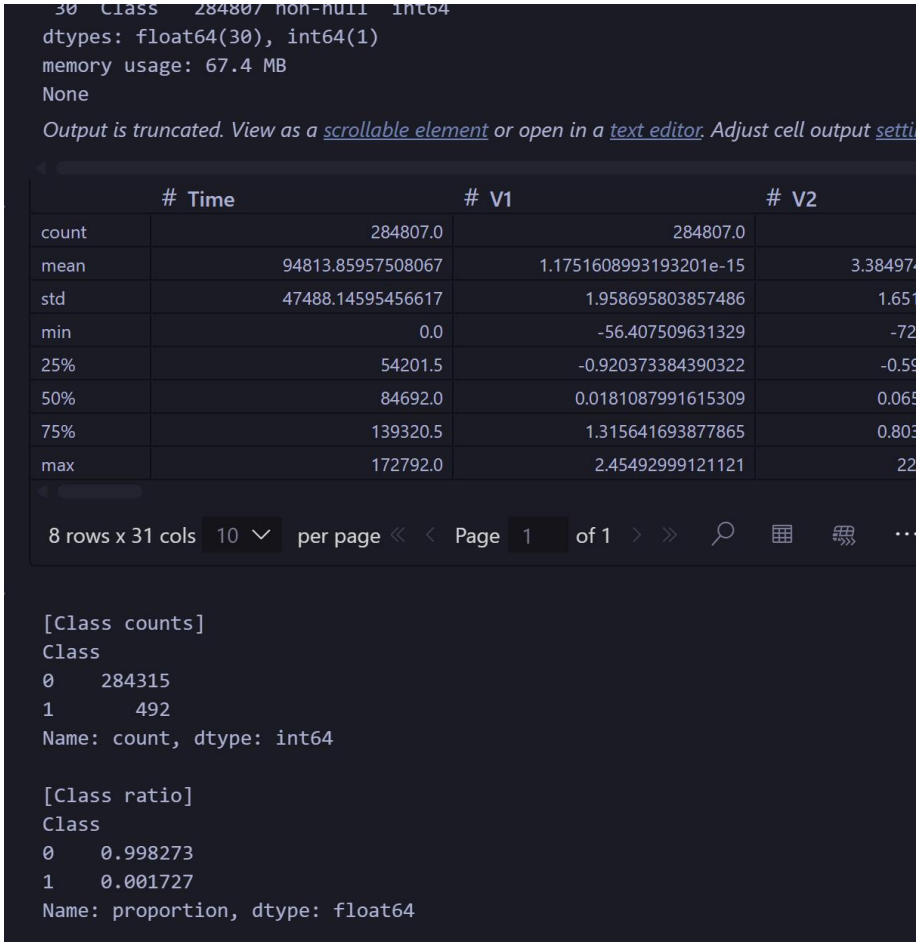
✓ 1.6s Python

```python
DATA_PATH = "creditcard.csv"  # 같은 폴더에 있다고 가정

df = pd.read_csv(DATA_PATH)

display(df.head())
print(df.info())
display(df.describe())

# Class 분포 확인
class_counts = df["Class"].value_counts().sort_index()
class_ratio = df["Class"].value_counts(normalize=True).sort_index()

print("\n[Class counts]")
print(class_counts)

print("\n[Class ratio]")
print(class_ratio)
```

info 및 클래스 수 및 비율프린트로 하여금 클래스 비율 및 전반적인 구조를 인지할 수 있음 (출력 결과 아래 첨부)


1.3s Data Wrangler에서 'class_ratio' 열기 Python
Time # V1 # V2
0 0.0 -1.3598071336738 -0.072781
1 0.0 1.19185711131486 0.2661
2 1.0 -1.35835406159823 -1.3401
3 1.0 -0.966271711572087 -0.18522
4 2.0 -1.15823309349523 0.87773
5 rows x 31 cols 10 per page Page 1 of 1
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
Column Non-Null Count Dtype
0 Time 284807 non-null float64
1 V1 284807 non-null float64
2 V2 284807 non-null float64
3 V3 284807 non-null float64
4 V4 284807 non-null float64
5 V5 284807 non-null float64
6 V6 284807 non-null float64
7 V7 284807 non-null float64
8 V8 284807 non-null float64

(중간 일부 생략)


dtypes: float64(30), int64(1)
memory usage: 67.4 MB
None
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output
Time # V1 # V2
count 284807.0 284807.0
mean 94813.85957508067 1.1751608993193201e-15 3.38497
std 47488.14595456617 1.958695803857486 1.65
min 0.0 -56.407509631329 -72
25% 54201.5 -0.920373384390322 -0.5
50% 84692.0 0.0181087991615309 0.06
75% 139320.5 1.315641693877865 0.80
max 172792.0 2.45492999121121 22
8 rows x 31 cols 10 per page Page 1 of 1
[Class counts]
Class
0 284315
1 492
Name: count, dtype: int64
[Class ratio]
Class
0 0.998273
1 0.001727
Name: proportion, dtype: float64

01 샘플링

사기 거래(Class=1)는 전부 유지하고, 정상 거래(Class=0)는 10,000건만 무작위 샘플링하시오.(sampling 진행시, random_state는 42로 설정)

두 데이터셋을 합쳐 새로운 분석용 데이터프레임으로 만드시오.

샘플링 후 Class 비율을 다시 출력하시오.

샘플링 후 및 데이터 프레임 구조로 만듦.

```python
RANDOM_STATE = 42
N_NORMAL = 10_000

df_fraud = df[df["Class"] == 1].copy()
df_normal = df[df["Class"] == 0].sample(n=N_NORMAL, random_state=RANDOM_STATE).copy()

df_sampled = pd.concat([df_normal, df_fraud], axis=0).sample(frac=1.0, random_state=RANDOM_STATE).reset_index(drop=True)

print("[After sampling] shape:", df_sampled.shape)
print("\n[Class counts]")
print(df_sampled["Class"].value_counts().sort_index())
print("\n[Class ratio]")
print(df_sampled["Class"].value_counts(normalize=True).sort_index())
```

✓ 0.0s Python

```
[After sampling] shape: (10492, 31)

[Class counts]
Class
0    10000
1      492
Name: count, dtype: int64

[Class ratio]
Class
0    0.953107
1    0.046893
Name: proportion, dtype: float64
```

02 데이터 전처리

Amount 변수만 표준화(StandardScaler) 하여 새로운 변수 Amount_Scaled로 대체하시오. Amount 원본 변수는 제거하시오.

그리고 X, y로 데이터프레임을 분리하시오.

```python
scaler = StandardScaler()
df_sampled["Amount_Scaled"] = scaler.fit_transform(df_sampled[["Amount"]])

# Amount 원본 제거
df_sampled = df_sampled.drop(columns=["Amount"])

# X, y 분리
X = df_sampled.drop(columns=["Class"])
y = df_sampled["Class"].astype(int)

print("X shape:", X.shape)
print("y distribution:\n", y.value_counts().sort_index())
```

✓ 0.0s

```
X shape: (10492, 30)
y distribution:
 Class
0    10000
1      492
Name: count, dtype: int64
```

03 학습 데이터와 테스트 데이터 분할

train_test_split을 사용해 학습셋:테스트셋 비율을 8:2로 나누고,

stratify=y 옵션으로 클래스 비율 유지, 분할된 데이터의 Class 비율을 출력하시오.

(random_state는 42로 설정)

```python
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=RANDOM_STATE
)

print("[Train] y distribution:")
print(y_train.value_counts(normalize=True).sort_index())

print("\n[Test] y distribution:")
print(y_test.value_counts(normalize=True).sort_index())
```

✓ 0.0s

```
[Train] y distribution:
Class
0    0.953056
1    0.046944
Name: proportion, dtype: float64

[Test] y distribution:
Class
0    0.953311
1    0.046689
Name: proportion, dtype: float64
```

04 SMOTE 적용

학습 데이터(X_train)에 SMOTE를 적용하여 소수 클래스(사기 거래)를 오버샘플링하시오. (왜 SMOTE를 적용해야하는지까지 서술하시오.)

SMOTE 적용 전후의 사기 거래 건수를 출력하시오.

```
print("[Before SMOTE] fraud count in y_train:", int((y_train == 1).sum()))
print("[Before SMOTE] normal count in y_train:", int((y_train == 0).sum()))

smote = SMOTE(random_state=RANDOM_STATE)
X_train_sm, y_train_sm = smote.fit_resample(X_train, y_train)

print("\n[After SMOTE] fraud count in y_train_sm:", int((y_train_sm == 1).sum()))
print("[After SMOTE] normal count in y_train_sm:", int((y_train_sm == 0).sum()))
```

✓ 2.3s

```
[Before SMOTE] fraud count in y_train: 394
[Before SMOTE] normal count in y_train: 7999

[After SMOTE] fraud count in y_train_sm: 7999
[After SMOTE] normal count in y_train_sm: 7999
```

클래스 불균형에서 모델이 다수 클래스 위주로 학습되어 소수 클래스에 대한 재현율(Recall)이 낮아질 수 있습니다.

SMOTE는 소수 클래스 주변의 이웃(KNN)을 기반으로 새로운 합성 샘플을 만들어 학습 데이터에서 소수 클래스 비중을 늘려, 모델이 소수 데이터의 패턴을 더 잘 학습하도록 돕습니다.

따라서 데이터 불균형 상황을 해결할 수 있습니다.
(실제로 위에서 클래스 불균형 상황이 어느정도 해결되었음을 인지할 수 있습니다.)

05 모델 학습

적합한 ML 모델을 선정하여 모델을 학습시키고,

테스트셋에서 예측값(predict)과 예측 확률(predict_proba)을 출력하시오.

classification_report로 Precision, Recall, F1-score를 확인하시오.

그리고 average_precision_score로 PR-AUC를 계산하여 출력하시오.

```python
def evaluate_with_threshold(y_true, y_proba, threshold: float):
    y_pred = (y_proba >= threshold).astype(int)
    report = classification_report(y_true, y_pred, digits=4)
    cm = confusion_matrix(y_true, y_pred)
    pr_auc = average_precision_score(y_true, y_proba)
    return report, cm, pr_auc

models = {
    "LogReg": LogisticRegression(
        max_iter=2000,
        n_jobs=None,
        random_state=RANDOM_STATE
    ),
    "RandomForest": RandomForestClassifier(
        n_estimators=300,
        random_state=RANDOM_STATE,
        n_jobs=-1,
        class_weight=None  # SMOTE를 이미 쓰므로 우선 None으로 둠
    ),
}

results = {}

for name, model in models.items():
    model.fit(X_train_sm, y_train_sm)

    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    print(f"\n===== {name} (default threshold=0.5) =====")
    print("[predict] head:", y_pred[:10])
    print("[predict_proba] head:", y_proba[:10])

    print("\n[classification_report]")
    print(classification_report(y_test, y_pred, digits=4))

    pr_auc = average_precision_score(y_test, y_proba)
    print("[PR-AUC]", pr_auc)

    results[name] = {"model": model, "y_proba": y_proba, "pr_auc": pr_auc}
```

✓ 4.1s

일단 ML에서 주로 쓰이는 2개의 모델을 정의하여 더 적합한 (성능이 우수한) 모델을 선택하도록 하겠다.

(만일 클래스 불균형이 있다면 가중치를 주어 더 신경써서 소수 데이터 셋을 다룰 수 있긴 하나 이미 위에서 클래스 불균형에 대해서 조치를 취하였기에 해당 과정에서는 클래스 불균형을 더 이상 고려하지 않음)

05회차 - basic_ML.ipynb

```
===== LogReg (default threshold=0.5) =====
[predict] head: [0 0 0 0 0 0 1 0 0 0]
[predict_proba] head: [1.41254269e-04 2.31221185e-04 7.69087384e-02 4.48421258e-02
 1.37827512e-02 1.91148496e-02 1.00000000e+00 1.62572699e-03
 8.58260258e-04 5.76142693e-03]

[classification_report]
              precision    recall  f1-score   support

           0     0.9935    0.9895    0.9915      2001
           1     0.8019    0.8673    0.8333        98

    accuracy                         0.9838      2099
   macro avg     0.8977    0.9284    0.9124      2099
weighted avg     0.9845    0.9838    0.9841      2099

[PR-AUC] 0.9162936016457237

===== RandomForest (default threshold=0.5) =====
[predict] head: [0 0 0 0 0 0 1 0 0 0]
[predict_proba] head: [0.         0.03       0.00333333 0.         0.08666667 0.00333333
 1.         0.02333333 0.00666667 0.01      ]

[classification_report]
              precision    recall  f1-score   support

           0     0.9930    0.9970    0.9950      2001
           1     0.9333    0.8571    0.8936        98

    accuracy                         0.9905      2099
   macro avg     0.9632    0.9271    0.9443      2099
weighted avg     0.9902    0.9905    0.9903      2099

[PR-AUC] 0.9218712809466375
```

7. 최종 성능 평가
   모델 선정, 하이퍼파라미터 튜닝과 Threshold 조정 등을 통해 최종 모델이
   목표 Recall ≥ 0.80, F1 ≥ 0.88, PR-AUC ≥ 0.90 을 달성하였는지
   여부를 작성하시오.(Class 0, 1 둘 다!)
   달성하지 못했다면 추가로 어떤 방법을 시도할지 간략히 제안하시오.

위 기본 상황에서 로지스틱 회귀 모델은 위에서 언급한 조건을 모두 만족시키진 못했다.
이는 위 사진의 F-1 점수를 보면 파악할 수 있다.

다만 랜덤 포레스트는 위에서 언급한 조건을 모두 만족하는 형상을 볼 수 있다.
(이는 위 사진에서 인지할 수 있다.)

다만 이대로 끝내기엔 아쉬우니 조정을 가해보았다.

```python
def find_best_threshold(y_true, y_proba, min_recall=0.80):
    precisions, recalls, thresholds = precision_recall_curve(y_true, y_proba)
    # precision_recall_curve는 thresholds 길이가 (len(precisions)-1) 이므로 정렬 맞춰줌
    # thresholds[i]는 precisions[i+1], recalls[i+1]에 대응
    best = None

    for i, thr in enumerate(thresholds):
        p = precisions[i+1]
        r = recalls[i+1]
        if r >= min_recall:
            if (best is None) or (2*p*r/(p+r) > best["f1"]):
                f1 = 2*p*r/(p+r) if (p+r) > 0 else 0.0
                best = {"threshold": float(thr), "precision": float(p), "recall": float(r), "f1": float(f1)}
    return best

for name in results:
    y_proba = results[name]["y_proba"]
    best = find_best_threshold(y_test.values, y_proba, min_recall=0.80)

    print(f"\n===== {name} threshold search (constraint: Recall>=0.80) =====")
    if best is None:
        print("No threshold satisfies Recall>=0.80 on this test set.")
        continue

    print("Best threshold:", best["threshold"])
    print("Precision:", best["precision"])
    print("Recall:", best["recall"])
    print("F1:", best["f1"])

    report, cm, pr_auc = evaluate_with_threshold(y_test.values, y_proba, best["threshold"])
    print("\n[classification_report @ best threshold]")
    print(report)
    print("[confusion_matrix]\n", cm)
    print("[PR-AUC]", pr_auc)

    results[name]["best_threshold"] = best
```

05회차 - basic_ML.ipynb

```
===== LogReg threshold search (constraint: Recall>=0.80) =====
Best threshold: 0.9650157732192337
Precision: 0.9882352941176471
Recall: 0.8571428571428571
F1: 0.9180327868852458

[classification_report @ best threshold]
              precision    recall  f1-score   support

           0     0.9930    0.9990    0.9960      2001
           1     0.9767    0.8571    0.9130        98

    accuracy                         0.9924      2099
   macro avg     0.9849    0.9281    0.9545      2099
weighted avg     0.9923    0.9924    0.9921      2099

[confusion_matrix]
 [[1999    2]
 [  14   84]]
[PR-AUC] 0.9162936016457237

===== RandomForest threshold search (constraint: Recall>=0.80) =====
Best threshold: 0.68
Precision: 1.0
Recall: 0.8469387755102041
F1: 0.9171270718232044

[classification_report @ best threshold]
              precision    recall  f1-score   support

           0     0.9926    0.9995    0.9960      2001
           1     0.9881    0.8469    0.9121        98

    accuracy                         0.9924      2099
   macro avg     0.9903    0.9232    0.9541      2099
weighted avg     0.9923    0.9924    0.9921      2099

[confusion_matrix]
 [[2000    1]
 [  15   83]]
[PR-AUC] 0.9218712809466375
```

본질적으로 클래스 불균형 문제가 존재했기에 이슈가 있을 수 있는 수치는 Recall이라고 생각했다. 이에 따라 Recall이 0.8보다 크게 하여 임계값을 서치해본 결과 위와 같은 결과가 나왔다.

threshold=0.965에서 (Class 1) Precision/Recall/F1=0.9767/0.8571/0.9130(PR-AUC=0.9163)로 기본(0.5) 대비 F1이 크게 개선되었다.

이에 따라 앞서 문제에서 언급한 요구사항을 충족시키는 상황으로 만들었다.

RandomForest는 threshold=0.68에서 (Class 1) 0.9881/0.8469/0.9121(PR-AUC=0.9219)로 F1이 상승했으며, 결과적으로 두 모델 모두 목표(Recall≥0.80, F1≥0.88, PR-AUC≥0.90)를 만족했다.

감사합니다!