1. 데이터 탐색 및 확보

Kaggle Automobile Loan Default Set을 연구데이터로 선정 (38개 독립변수로 구성된 202,765건의 DataSet)

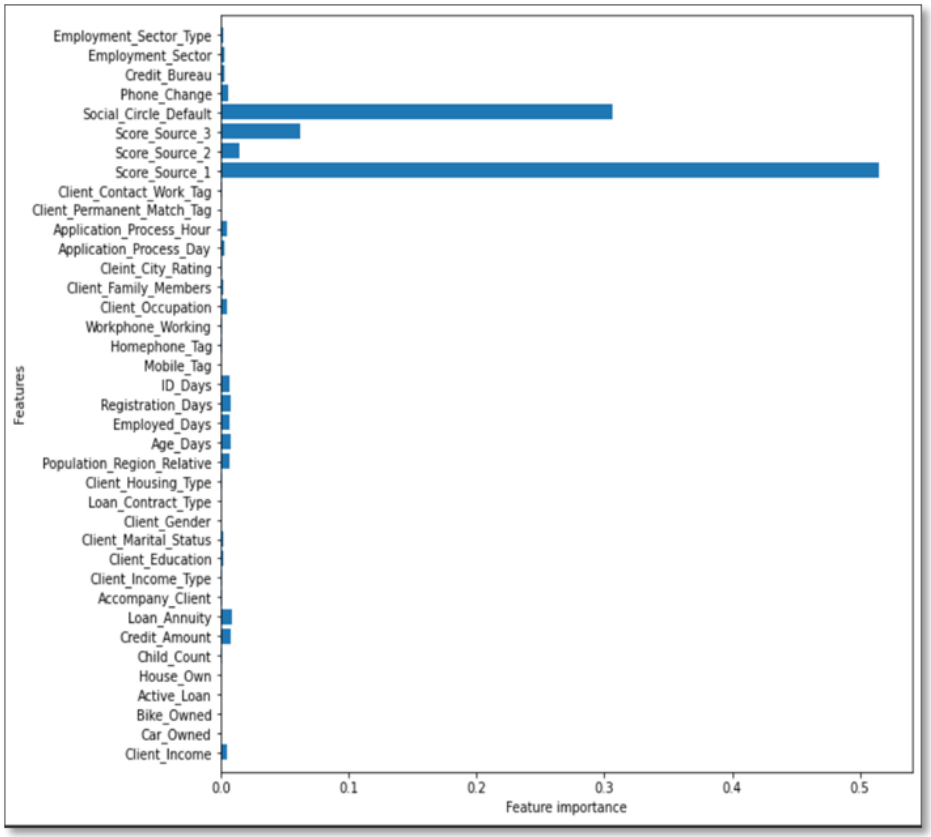
2. 변수 선정

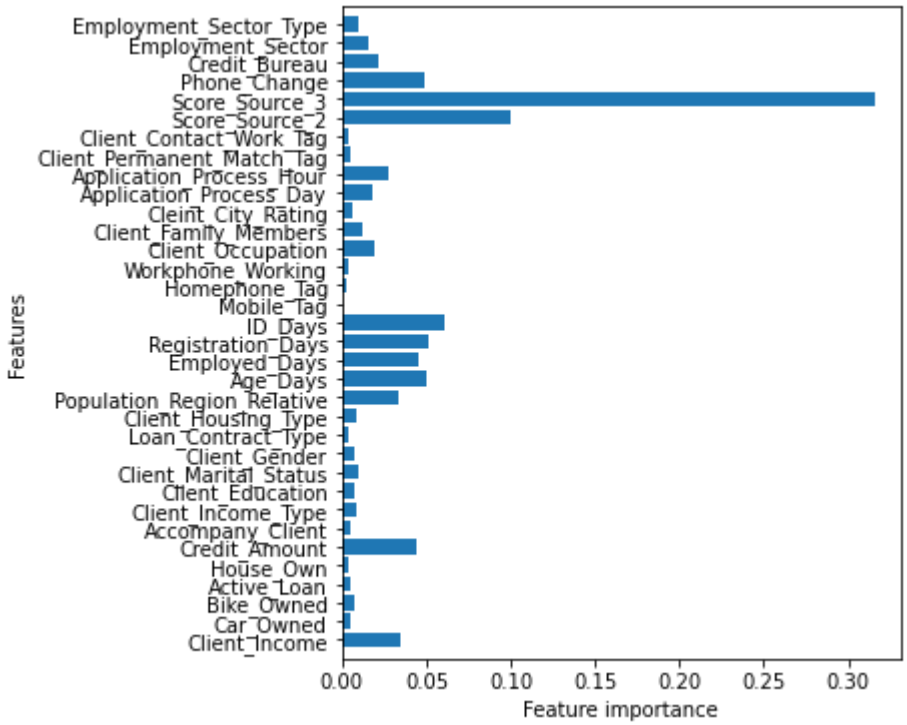
(EDA => 변수간 상관관계 분석, 유효변수 추론/선정, Feature Importance 검토 등)

사람에 의해 해석이 가능한 Decision Tree Model의 특성을 활용하여 Feature Importance 분석 후 영향도 상위권 변수(엔트로피가 낮은 Feature)를 도출함.

Feature Importance는 높지만, 실 영업환경에서 확보불가한 데이터(친구/가족 Default 및 타기관 신용점수 일부와, 상관관계가 높은 변수값)를 유효하지 않은 변수로 판단하고 학습 대상에서 제외시킴

```
X = df.drop(['ID','Default','Social_Circle_Default','Score_Source_1','Child_Count','Loan_Annuity'], axis=1)
```



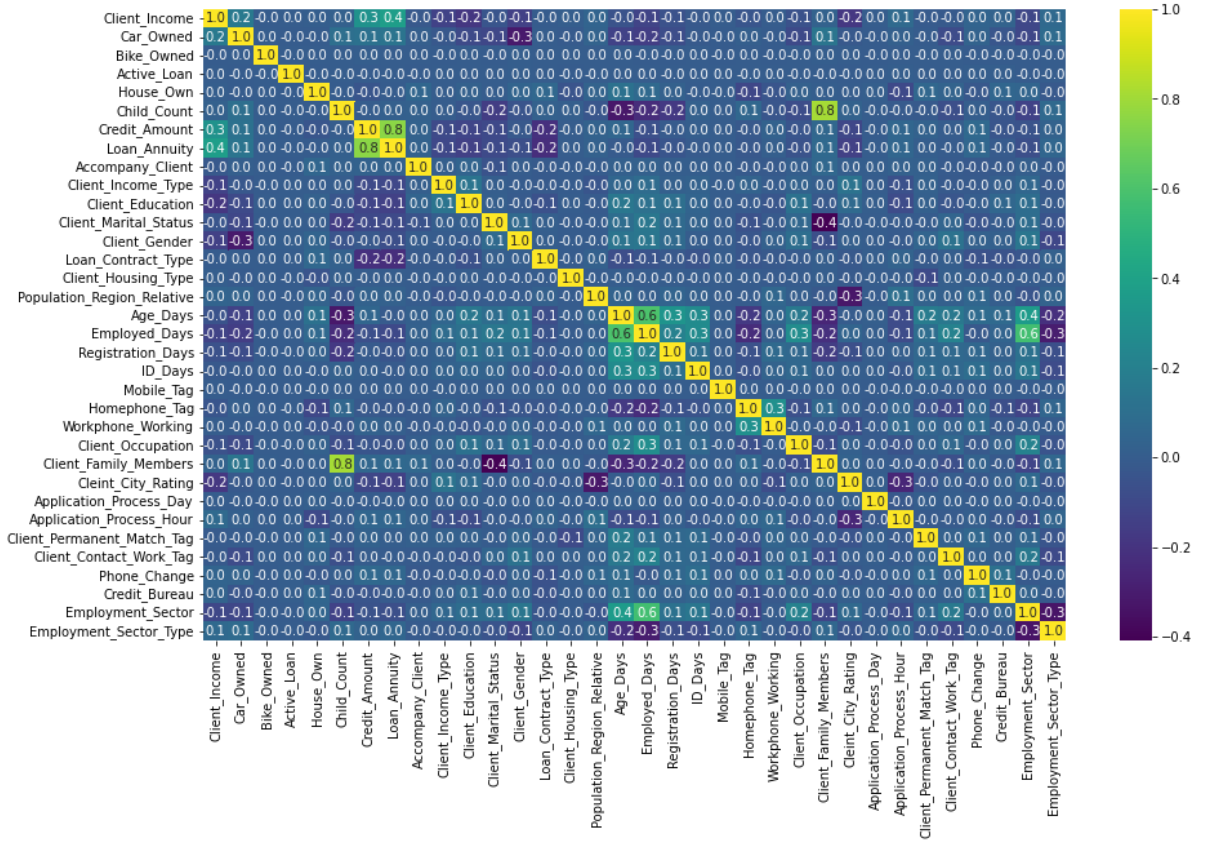


feature를 최대한 다 사용한다고 해도 한 가지 고려해야 할 사항이 있다. 바로 다중공선성이다. 다중공선성이란 feature들 사이에 강한 상관관계가 존재하는 것으로, 만약 데이터셋에

다중공선성이 있다면 제대로 된 모델을 만들 수 없다. feature들 사이에 상관관계가 존재하지 않도록 주의해야 한다.

☞ Child_count와 Client_Family_members, 그리고 Credit_Amount와 Loan_Annuity간의 Correlation이 0.8로 높은 편이기 때문에, Child_count와 Loan_Annuity는 제거하여 진행해보기로 함.

```
#feature별 상관관계 시각화
df_corr = X.corr()
plt.figure(figsize = (15,9))
sns.heatmap(df_corr, cmap='viridis', annot=True, fmt='.1f')
plt.show()
```



3. 데이터 전처리

결측데이터 처리, 이상치 제거, Categorial Data를 Numerical Data로 변환, OverSampling, 정규화 및 표준화 등

Variable	Description	Preprocessing
ID	고객 대출신청서 ID	drop
Client_Income	소득	convert to float
Car_Owned	차 소유 여부	consider NoN as 0
Bike_Owned	바이크 소유 여부	consider NoN as 0
Active_Loan	대출 신청 당시 대출 유무	consider NoN as 0
House_Own	집 소유 여부	consider NoN as 0
Child_Count	자녀 수	consider NoN as 0
Credit_Amount	대출 금액	remove rows
Loan_Annuity	대출 연금	mean()
Accompany_Client	동행인	Label Encoder
Client_Income_Type	소득형태	Label Encoder
Client_Education	교육수준	Label Encoder
Client_Marital_Status	결혼여부	Label Encoder
Client_Gender	성별	Label Encoder
Loan_Contract_Type	대출유형	Label Encoder
Client_Housing_Type	주택현황	Label Encoder
Population_Region_Relative	거주지역 인구	convert to float
Age_Days	나이	median()
Employed_Days	소득발생일수	median()
Registration_Days	거주변동일수	median()
ID_Days	신분증갱신일수	median()
Own_House_Age	거주지 준공연수	High NoN (remove)
Mobile_Tag	핸드폰번호 제공여부	1 : Yes / 0 : No
Homephone_Tag	집전화번호 제공여부	1 : Yes / 0 : No
Workphone_Working	직장번호 연결 여부	1 : Yes / 0 : No
Client_Occupation	직업 형태	Label Encoder
Client_Family_Members	가족 수	consider NoN as 0
Cleint_City_Rating	도시 등급	median()
Application_Process_Day	대출 신청 요일	median()
Application_Process_Hour	대출 신청 시간	median()
Client_Permanent_Match_Tag	거주지와 본적 불일치 여부	Label Encoder
Client_Contact_Work_Tag	근무지와 본적 불일치 여부	Label Encoder
Type_Organization	근무회사 형태	Label Encoder
Score_Source_1	타기관 점수1	median()
Score_Source_2	타기관 점수2	median()
Score_Source_3	타기관 점수3	median()
Social_Circle_Default	최근 60일 친구/가족 Default 수	mean()
Phone_Change	대출신청 전 핸드폰 변경 횟수	median()
Credit_Bureau	최근 1년 신용평가조회 수	median()
Default	Default 여부	drop

전체 훈련데이터 중 5.23%(9,798건)만 Default 데이터. (이진분류 Problem에서 대표적으로 발생하는 데이터 불균형 존재함)

Default	
0	187030
1	9798

다수 클래스(Majority) 대비 소수 클래스(Minority)의 비율을 10%로 설정함.

☞ 이진분류 문제에서 소수클래스 비율을 너무(50%수준으로) 높이면, Accuracy, Precision, Recall, F1 Score 모든 수치가 비정상적으로 높아짐 (*수행결과 99%에 달하는 학습결과 나옴)

소수클래스 비율을 30%로만 늘려도, baseline모델에서 f1_score가 87.5점이 나옴

따라서, **Default 데이터 비율(sampling_strategy)을 10%수준으로 오버샘플링 진행했고, 오버샘플링기법으로는 SMOTE(K-NN기반 OverSampling)를 사용했었으나, 너무 낮은 Precison과 Recall, F1-Score가 나와서 RandomOverSampling하는 방식으로 변경해서**

시도해 봄. 전체적인 모델의 성능이 올라감 (오버샘플링 기법과 소수 클래스 비율에 따라서 상당히 다른 결과를 보여줌)

```
# oversample dataset
oversample = RandomOverSampler(sampling_strategy=0.1, random_state=0)
#oversample = SMOTE(sampling_strategy=0.1, random_state=0)
#oversample = ADASYN(sampling_strategy=0.1, random_state=0
```

```
oversampling 전 학습 data 수 : 196828
oversampling 전 default data 수 : 9798 ( 4.98 %)
oversampling 후 학습 data 수 : 205733
oversampling 후 default data 수 : 18703 ( 9.09 %)
```

데이터 스케일링 : StandardScaler() 사용하여 표준화(**Standardization**) 작업 수행

```
1 scaler = StandardScaler() #사이킷런의 import 모듈 (평균0 분산1로 스케일링)
2 X_over = scaler.fit_transform(X_over)
3 print(X_over)
```

```
[[-0.86625175 -0.70563737 -0.69485482 ... -0.44983638  0.82819995
  -0.639201  ]
 [ 0.30175137  1.43623591 -0.69485482 ... -0.44983638 -0.75773356
  -0.639201  ]
 [ 0.10708419 -0.70563737 -0.69485482 ... -1.02706335  0.82819995
  -0.639201  ]
 ...
 [-0.64428573 -0.70563737  1.4611301  ... -0.58158155  0.0352332
  -0.639201  ]
 [ 0.04272491  1.43623591 -0.69485482 ...  0.9166631  -0.44054686
  -0.639201  ]
 [ 0.13833777 -0.70563737 -0.69485482 ... -0.33597485  1.30398
  -0.639201  ]]
```

학습,검증,테스트 dataset 분리 작업 (train_test_split 사용)

```
X_train, X_valid, y_train, y_valid = train_test_split(X_over, y_over, test_size=0.2,
stratify=y_over, shuffle=True, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.3,
stratify=y_train, shuffle=True, random_state=42)
```

4. 모델링

Baseline 선정, 각종 모형적용, 앙상블 적용, Grid Search, 모형성능 평가지표 향상

4-1 Baseline (NN-Model)

본 수업에서 가장 먼저 실습했던 2 Hidden Layer (2계층)으로 구성된 Neural Network (Sequential) Model을 Baseline으로 설정.
Overfitting문제를 해결하기 위해 사이킷런 Dropout함수 적용하여 일부데이터 무작위로 입력값을 0으로 처리 수행함. (*학습시에만 적용하며, 검증/테스트 단계에서는 사용하면 안됨)

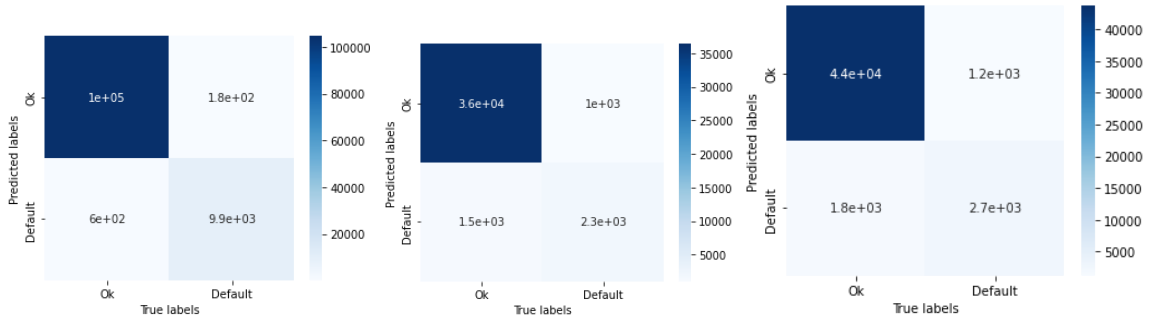
```
# compose the NN model
# https://wikidocs.net/32105

# compose the NN model
model = Sequential()
model.add(Dense(256, activation='relu')) #hidden layer 1
model.add(Dense(256, activation='relu')) #hidden layer 2
model.add(Dropout(0.3))

model.add(Dense(2, activation='softmax')) #output layer
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['AUC'])
model.fit(X_train, y_train, batch_size = 256,  epochs = 80, verbose = True,
validation_data=(X_test, y_test), shuffle=True)
# Evaluate the model accuracy on the validation set.
score = model.evaluate(X_valid, y_valid, verbose=0)

# predict for train and validation (will be used later for confusion matrix)
y_train_pred = model.predict(X_train)
```

```
y_test_pred = model.predict(X_test)
y_valid_pred = model.predict(X_valid)
```

Model Shape

- 활성화함수 : Relu(은닉층), Softmax(출력층)
- 손실함수 : Binary Cross Entropy
- 배치Size : 256, Epochs : 80, Optimizer : ADAM

Baseline 모델 모형성능 평가



(훈련 > 검증 > 테스트 Confusion Matrix)

데이터 불균형(imbalanced lable)이 존재하는 이진분류 문제에서 Accuracy만으로 모델의 성능을 평가할 수 없으므로, Precision, Recall, F1 Score 모형성능 평가지표를 사용하기로 함. 또한, 모델에 따라 ROC Curve 또는 AUC(ROC Curve를 환산한 값)을 추가 평가지표로 채택하고자 함. (Default 고객을 판별하는 Problem은 Recall 값을 높이는 분류문제에 해당함)

```
# accuracy 만으로 모델의 성능을 판단하는 것은 적합하지 않다.
#imbalanced한 label값 분포에서는 Precision, Recall, F1 Score 등의 평가지표를 고려해야 한다.

# print accuracy and f1 score for training data
print(accuracy_score(y_train_for_cmatrix, y_train_preds_for_cmatrix),
f1_score(y_train_for_cmatrix, y_train_preds_for_cmatrix))
# print accuracy and f1 score for validation data
```

```
print(accuracy_score(y_valid_for_cmatrix, y_valid_preds_for_cmatrix),
f1_score(y_valid_for_cmatrix, y_valid_preds_for_cmatrix))
# print accuracy and f1 score for test data
print(accuracy_score(y_test_for_cmatrix, y_test_preds_for_cmatrix),
f1_score(y_test_for_cmatrix, y_test_preds_for_cmatrix))

0.9932297543616005 0.9620068192888458
0.9390721073225266 0.6427248111728658
0.9403556383668179 0.648945047085469

print(classification_report(y_test_for_cmatrix, y_test_preds_for_cmatrix,
target_names=['OK','Default']))
```

	precision	recall	f1-score	support
OK	0.96	0.97	0.97	44887
Default	0.70	0.61	0.65	4489

Precision과 Recall 평가지표의 Trade-Off

실제 값이 Positive(1, Default)일 때, 예측한 값이 Positive 일 경우가 매우 중요한 문제이기 때문에 메인 성능지표를 Recall(재현율)로 하고, Precision과 Recall의 중간값인 f1-score을 보조 지표로 사용하기로 한다. (예: 암환자를 암에 걸리지 않았다고 판단할 위험) f1-score가 높을수록 보다 객관적인 지표로고 판단할 수 있다.

☞ Precision과 Recall은 상호 보완적인 평가 지표이기 때문에 어느 한 쪽을 강제로 높이면 다른 하나의 수치는 떨어지기 쉬움. Precision/Recall Trade-off라고 하며, 임계치(Threshold)를 조정을 통하여, Precision과 Recall 값은 조작이 가능함.

아래는 정밀도와 재현율에 대한 개념 및 trade-off에 대한 내용을 갈무리함.

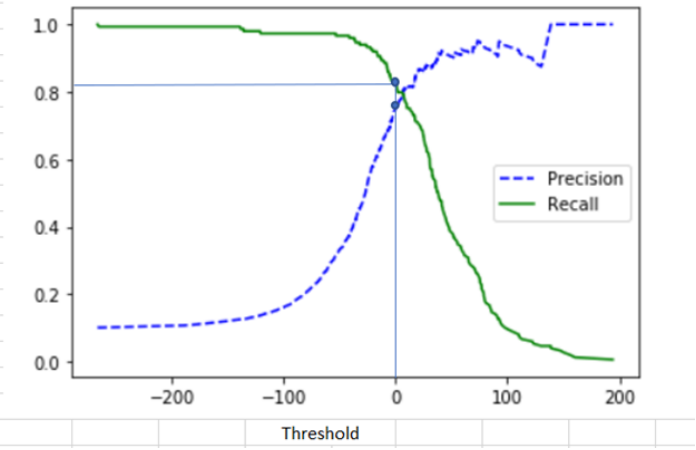
정밀도(Precision), 재현율(Recall)

Precision과 **Recall**은 각각 다음 공식으로 계산된다.

$$\text{Precision} = \frac{\text{TP}}{\text{FP} + \text{TP}} = \frac{\text{예측과 실제 값이 Positive로 일치하는 것들}}{\text{Positive로 예측한 것들}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{FN} + \text{TP}} = \frac{\text{예측과 실제 값이 Positive로 일치하는 것들}}{\text{실제 값이 Positive인 것들}}$$

Precision과 Recall 모두 TP를 높이는 데 동일하게 초점을 맞추지만, **Precision은 FP(실제는 0인데 예측은 1)을 낮추는 데에 초점을, Recall은 FN(실제는 1인데 예측 0)을 낮추는 데에 초점을 맞춘다.**



4-2 Decision Tree (Model #1)

Model Shape

- Decision Tree Classifier : Gini Index (default)
- random_state=0
- min_samples_leaf=2
- min_samples_split=3
- max_features=31
- max_depth=None
-

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=0, min_samples_leaf=2,
min_samples_split=3, max_features=31, max_depth=None)

tree.fit(X_train, y_train)
print("훈련 세트 정확도: {:.3f}".format(tree.score(X_train, y_train)))
```

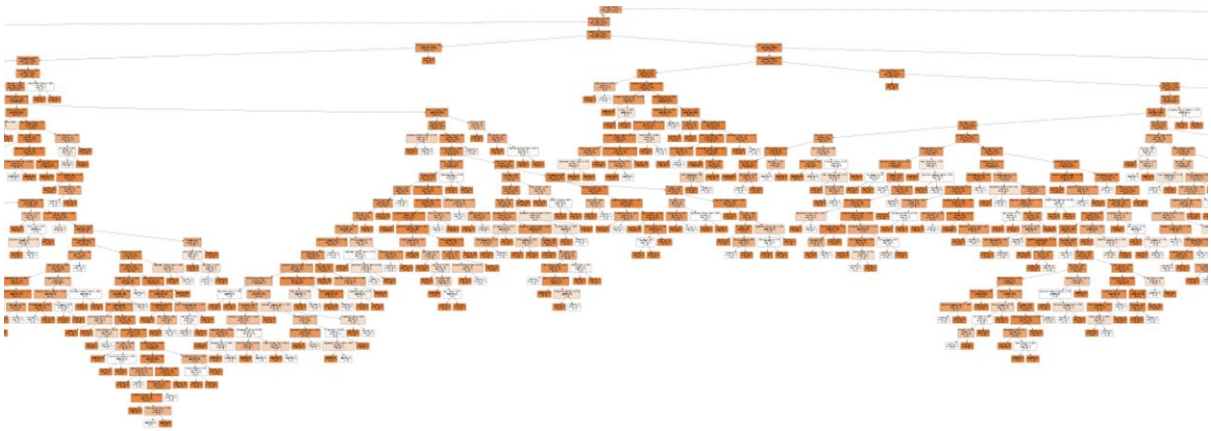
```
print("테스트 세트 정확도: {:.3f}".format(tree.score(X_test, y_test)))
print("검증 세트 정확도: {:.3f}".format(tree.score(X_valid, y_valid)))
```

```
훈련 세트 정확도: 0.976
테스트 세트 정확도: 0.913
검증 세트 정확도: 0.915
```

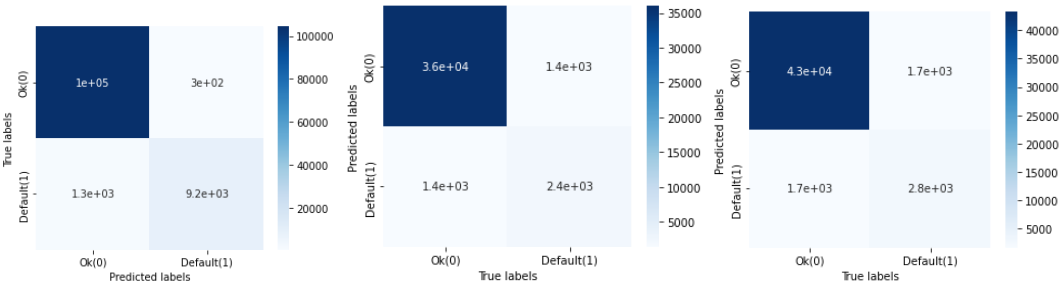
ROC AUC값은 0.8131

```
from sklearn.metrics import roc_curve, roc_auc_score

roc_score = roc_auc_score(y_test, y_test_pred)
print('ROC AUC 값 : {:.4f}'.format(roc_score))

ROC AUC 값 : 0.8131
```

Decision Tree Split을 도식화한 결과



왼쪽부터 차례대로, 훈련 > 검증 > 테스트 결과를 Confusion Matrix로 분석



f1-score가 0.62 이고, Decision Tree의 여러가지 하이퍼파라미터를 조정하여도 더이상 좋은 성능을 기대할 수 없었음. (다만, 독립변수의 선정이나 전처리과정에서 오버샘플링하는 방식에 의해서 값이 크게 움직일 수는 있음. 프로젝트 중간발표의 경우처럼)

```
# print accuracy and f1 score for training data
print(accuracy_score(y_train_for_cmatrix, y_train_preds_for_cmatrix),
f1_score(y_train_for_cmatrix, y_train_preds_for_cmatrix))
# print accuracy and f1 score for test data
print(accuracy_score(y_test_for_cmatrix, y_test_preds_for_cmatrix),
f1_score(y_test_for_cmatrix, y_test_preds_for_cmatrix))
# print accuracy and f1 score for validation data
print(accuracy_score(y_valid_for_cmatrix, y_valid_preds_for_cmatrix),
f1_score(y_valid_for_cmatrix, y_valid_preds_for_cmatrix))

0.9860949570349796 0.9196750902527077
0.931302657161374 0.6204967554262697
0.9319999027875665 0.627132196162047
```

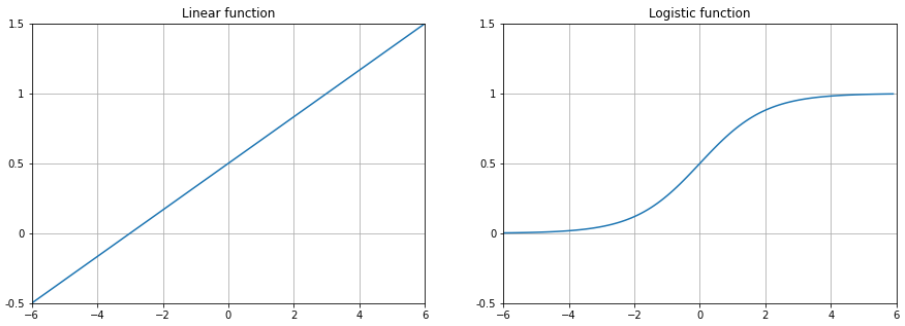
```
import warnings
warnings.filterwarnings(action='ignore')
print(classification_report(y_test, y_test_pred, target_names=['OK','Default']
```

	precision	recall	f1-score	support
OK	0.97	0.94	0.96	44887
Default	0.62	0.62	0.62	4489

4-3 Logistic Regression (Model #1)

Logistic은 성능이 너무 안나와서 발표자료에서는 제외해도 좋을 것 같습니다.

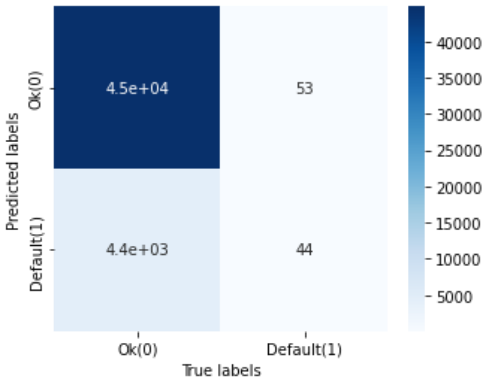
지금부터는 다양한 모델에 하이퍼파라미터를 조정해가며 단순히 (즉, 변수 변경/전처리 등의 조정없이) 우리의 연구과제에 가장 근접한 모델이 어떤건지? 그리고 모델별로 어떤 특성이 있고 어떠한 성능을 보여주는지 1차적(개략적)으로 수행해보고, 그 결과를 검토한 후 방향성을 수립하고자 함.



- Parameter C를 조정해 과대적합 혹은 과소적합 문제를 해결
- C(cost function) 값이 크면(높은 코스트) -> 훈련을 더 복잡하게 -> 약한 규제
- C 값이 작으면 (낮은 코스트) -> 훈련을 덜 복잡하게 -> 강한 규제

```
# 학습/예측/평가
lr_clf = LogisticRegression()
lr_clf.fit(X_train, y_train[:,1])
lr_pred = lr_clf.predict(X_test)
get_clf_eval(y_test[:,1],lr_pred)
print(classification_report(y_test[:,1], lr_pred, target_names=['OK','Default']))
```

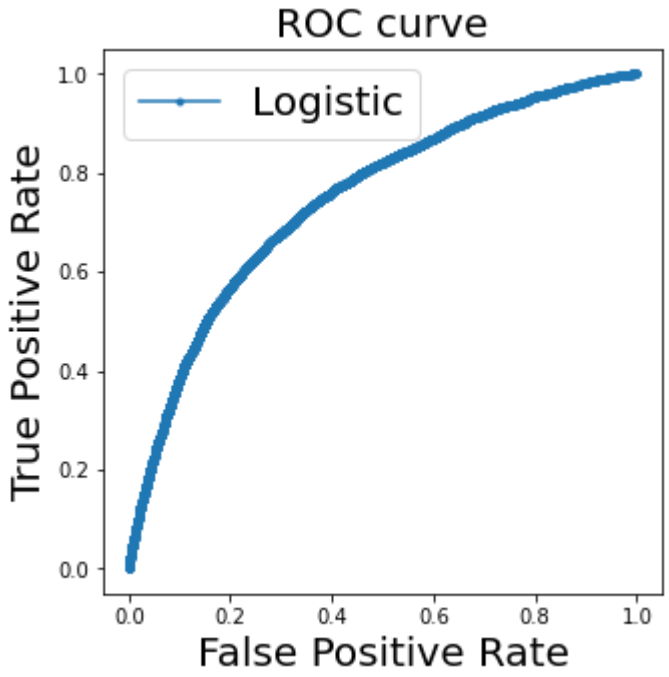
```
정확도: 0.9089, 정밀도: 0.4536, 재현율: 0.0098
```



	precision	recall	f1-score	support
OK	0.91	1.00	0.95	44887
Default	0.45	0.01	0.02	4489

positive와 negative의 분류기준(threshold)을 0.5가 아닌 0.1~1사이의 값으로 조정해보면서, 성능확인했으나 눈에 띄는 개선없음. precision/recall 지표간 trade-off 는 발생하지만 f1-score를 보면 성능이 매우 낮음. logistic regression은 최악의 f1-score와 recall을 보여줬음

```
probs = lr_clf.predict_proba(X_test)[:,1]
model_fpr, model_tpr, threshold1 = roc_curve(y_test[:,1], probs)
plt.plot(model_fpr, model_tpr, marker = '.', label = "Logistic")

# threshold 값 변경하기 - Binarizer
from sklearn.preprocessing import Binarizer

binarizer = Binarizer(threshold=0.5).fit(probs.reshape(-1,1))
pred_bin = binarizer.transform(probs.reshape(-1,1))

binarizer.threshold, pred_bin

print(classification_report(y_test[:,1], pred_bin, target_names=['OK','Default']))
```



4.4 KNN Model

```
kn_clf = KNeighborsClassifier(n_neighbors=20)
```

정확도: 0.9089, 정밀도: 0.4500, 재현율: 0.0080 (if K=20)

정확도: 0.9000, 정밀도: 0.4392, 재현율: 0.3622 (if K=3)

	precision	recall	f1-score	support
OK	0.94	0.95	0.95	44887
Default	0.44	0.36	0.40	4489

정확도: 0.9134, 정밀도: 0.6272, 재현율: 0.1170 (if K=10)

정확도: 0.9217, 정밀도: 0.5599, 재현율: 0.6469 (if K=1)

	precision	recall	f1-score	support
OK	0.96	0.95	0.96	44887
Default	0.56	0.65	0.60	4489

정확도: 0.9171, 정밀도: 0.5820, 재현율: 0.3132 (if K=2)

	precision	recall	f1-score	support
OK	0.93	0.98	0.96	44887
Default	0.58	0.31	0.41	4489

K값을 20으로 했을 때, 정밀도가 69.9%수준이었고, K를 1으로 설정하면 정밀도 43.8%이지만, 재현율이 35%로 높아졌음(과적합을 의도했는데 오히려 좋아???)

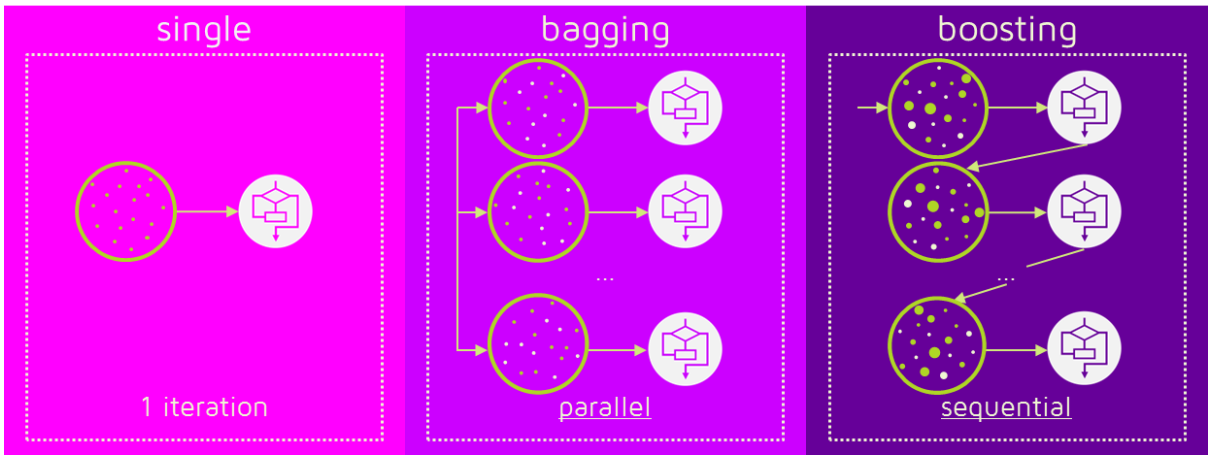
```
oversample = RandomOverSampler(sampling_strategy=0.1, random_state=0)
```

위에 한차례 언급했듯이, Oversampling 기법을 SMOTE에서 RandomOverSampler로 변경하고 KNN모델의 K를 1로 조정했을 때, 보다 향상된 예측결과를 보여주었음. (*훈련데이터 Set의 형태에 따라 차이가 있음) 이때부터 oversampling 및 전처리방식에 따라 그 결과가 많이 바뀐다고 합리적 의심을 하기 시작함.

4.5 XGBoost Model(앙상블 모델)

※ 모델 앙상블(Model Ensemble)이란?

각 기법에 맞게 여러 개의 의사결정 나무를 만들고, 이들을 조화롭게 학습시켜 모델의 예측력을 높이는 과정. 앙상블 모델의 핵심은 약한 분류 모델들을 결합하여 강한 분류 모델을 만들어 예측력을 높이는 것. 앙상블 기법은 대표적으로 배깅(Bagging)과 부스팅(Boosting)이 있음



배깅은 학습 데이터를 매번 다르게 해주는 병렬 학습 기법이고, 부스팅은 모델 자체를 변화시키는 기법입니다. 배깅은 병렬로 학습하는 기법이고, 부스팅은 순차적으로 학습하는 기법입니다. 배깅은 분산의 차이를 감소시키기 위해 사용되고, 부스팅은 편차를 감소시키기 위해 사용됩니다. 부스팅은 배깅에 비해 오류가 적지만 과적합이 일어날 가능성이 상대적으로 높습니다. 모델의 성능을 높이고 싶다면 부스팅을 사용하고, 과적합을 잡고 싶다면 배깅을 사용합니다.

부스팅 (Boosting)

- 부스팅(Boosting)은 각 이터레이션에서 맞추지 못한 데이터에 **가중치를 부여**하여 다음 학습 모델에 반영하여 예측력을 높입니다.
    - 잘못 예측된 데이터에 높은 가중치를 부여하여 오차를 보완합니다.
- **부스팅은 예측 모델의 편차가 클 때 편차를 감소시키기 위해 사용됩니다.**
- 대표적으로 **XGBoost**, Gradient Boost, AdaBoost, CatBoost 등이 있습니다
- 부스팅은 Decision Tree를 발전시킨 방법으로 머신러닝 사용할 때 성능이 좋다.

```
from xgboost import XGBClassifier

# 모델 선언 예시
xgb_clf = XGBClassifier(n_estimators=500, learning_rate=0.2, max_depth=12,
random_state = 32)
# n_estimators : 학습 모델의 수, 많아질수록 성능 향상의 가능성이 있으나, 속도가 느려짐
# learning_rate : 학습률, 너무 크면 gradient 발산의 가능성이 있으며, 너무 작으면 학습이 느림
# max_depth : 최대 탐색 깊이, 너무 크면 과적합의 가능성, 너무 작으면 학습 성능 저하
# min_samples_split : 분할 종료 최소 샘플 수, 큰 수면 과적합을 막지만 학습 성능 저하 가능성
# min_samples_leaf : leaf node가 되기 위한 최소 샘플 수, min_samples_split과 비슷한 용도
```

max_depth = 5 일 때, 정확도: 0.9526, 정밀도: 0.9235, 재현율: 0.5219

max_depth = 8 일 때, 정확도: 0.9606, 정밀도: 0.9218, 재현율: 0.6193

max_depth = 10 일 때, 정확도: 0.9719, 정밀도: 0.9396, 재현율: 0.7380

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.94	0.74	0.83	4489

max_depth = 10 & learning_rate=0.1 일 때

정확도: 0.9718, 정밀도: 0.9459, 재현율: 0.7322

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.95	0.73	0.83	4489

max_depth = 12 일 때, 정확도: 0.9640, 정밀도: 0.9302, 재현율: 0.653

max_depth = 15 일 때, 정확도: 0.9718, 정밀도: 0.9459, 재현율: 0.7322

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.95	0.73	0.83	4489

max_depth = 20 일 때, 정확도: 0.9710, 정밀도: 0.9381, 재현율: 0.7289

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.94	0.73	0.82	4489

max_depth = 30 일 때 정확도: 0.9708, 정밀도: 0.9394, 재현율: 0.7256

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.94	0.73	0.82	4489

4.6 LightGBM Model(앙상블 모델)

LightGBM란 무엇일까요?

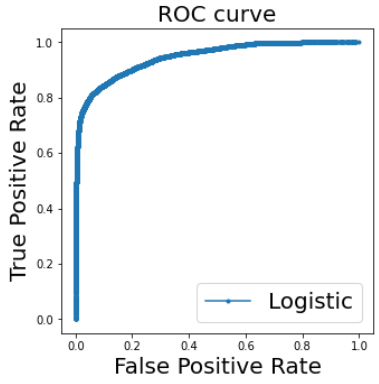
- LightGBM은 분류, 회귀 문제에 모두 사용할 수 있는 강력한 모델입니다.
- 각 이터레이션에서 맞추지 못한 데이터에 **가중치를 부여**하여 모델을 학습시키는 **부스팅(Boosting) 계열의 트리 모델**입니다.
- 강력한 **병렬 처리 성능**과 **자동 가지치기 알고리즘**이 적용되어 Gradient Boosting Model 대비 빠른 속도를 갖습니다.
- **과적합 규제 기능(Regularization)의** 이점이 있습니다.
- 또한 **자체 교차 검증 알고리즘**과 **결측치 처리 기능**을 가지고 있습니다.

- **리프 중심 트리 분할 방식**으로 **비대칭적인 트리**를 형성하여 모델을 학습하고, **예측 오류 손실을 최소화**합니다.
- Early Stopping 기능이 있습니다.
- 성능이 좋은 XGBoost와 성능은 비슷하지만 **속도가 훨씬 빠릅니다.**

```
from lightgbm import LGBMClassifier

# LGBMClassifier 모델 선언 후 Fitting
lgb_clf = LGBMClassifier(n_estimators=500, learning_rate=0.2, max_depth=15,
random_state = 32)
# n_estimators (int) : 내부에서 생성할 결정 트리의 개수
# max_depth (int) : 생성할 결정 트리의 높이
# learning_rate (float) : 훈련량, 학습 시 모델을 얼마나 업데이트할지 결정하는 값
# colsample_bytree (float) : 열 샘플링에 사용하는 비율
# subsample (float) : 행 샘플링에 사용하는 비율
# reg_alpha (float) : L1 정규화 계수
# reg_lambda (float) : L2 정규화 계수
# boosting_type (str) : 부스팅 방법 (gbdt / rf / dart / goss)
# random_state (int) : 내부적으로 사용되는 난수값
# n_jobs (int) : 병렬처리에 사용할 CPU 수
```

max_depth = 15일 때, 정확도: 0.9589, 정밀도: 0.9067, 재현율: 0.6104

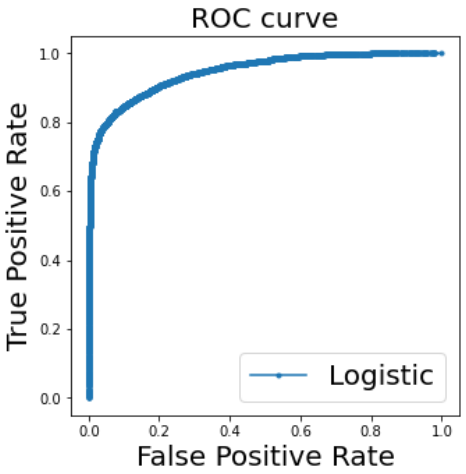
	precision	recall	f1-score	support
OK	0.96	0.99	0.98	44887
Default	0.91	0.61	0.73	4489



max_depth = 30일때 정확도: 0.9595, 정밀도: 0.9106, 재현율: 0.6153

	precision	recall	f1-score	support

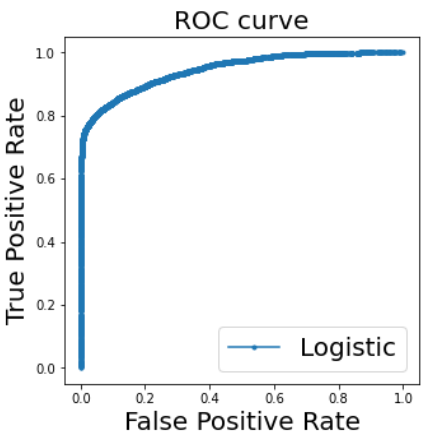
OK	0.96	0.99	0.98	44887
Default	0.91	0.62	0.73	4489




```
lgb_clf = LGBMClassifier(n_estimators=500, learning_rate=0.45, max_depth=20,
random_state = 32, objective='binary', boosting_type='gbdt', num_leaves=200,
metric='mae', sub_feature=0.5, min_data=500, reg_alpha=0.3,
num_boost_round=200,min_data_in_leaf=300,max_cat_group=1000)
```

정확도: 0.9679, 정밀도: 0.8957, 재현율: 0.7325

	precision	recall	f1-score	support
OK	0.97	0.99	0.98	44887
Default	0.90	0.73	0.81	4489



ROC AUC 값 : 0.8619

HyperParameter 조정을 통해 더 좋은 성능을 찾아볼 수 있을 듯함.

4.7 RandomForest Model(앙상블 모델)

랜덤 포레스트 (Random Forest)란 무엇일까요?

- 랜덤 포레스트는 분류, 회귀 문제에 모두 사용할 수 있는 강력한 모델입니다.
- 이전 포스팅에서 다뤘던 의사결정 나무를 여러개 사용하여 문제를 해결하는 방식으로 작동합니다.
- 랜덤 포레스트는 의사결정 나무가 여러개 모인 숲입니다. (의사결정 나무의 앙상블 모델)
- **Decision Tree는 나무 하나로 예측을 했다면 Random Forest는 여러 나무를 통해 종합된 결과를 바탕으로 예측합니다.**
- **과적합이 될 가능성이 높은 의사결정 나무의 단점을 보완**한 모델입니다.
- 학습 데이터의 부분집합을 랜덤하게 만들어 의사결정 나무를 구축하는 배깅(Bagging) 계열의 트리 모델입니다

```
from sklearn.ensemble import RandomForestClassifier

# RandomForestClassifier 모델 선언 후 Fitting
rf_clf = RandomForestClassifier(n_estimators=500, max_depth=30, random_state = 32)

# n_estimators (int) : 내부에서 생성할 결정 트리의 개수
# crierion (str) : 정보량 계산 시 사용할 수식 (분류 모델 : gini / entropy, 회귀 모델 : mse / rmse)
# max_depth (int) : 생성할 트리의 높이
# min_samples_split (int) : 분기를 수행하는 최소한의 데이터 수
# max_leaf_nodes (int) : 리프 노드에서 가지고 있을 수 있는 최대 데이터 수
# random_state (int) : 내부적으로 사용되는 난수값
# n_jobs (int) : 병렬처리에 사용할 CPU 수
# class_weight (dict) : 학습 시 클래스의 비율에 맞춰 손실값에 가중치를 부여 (분류 모델에서만 쓰임)

rf_clf.fit(X_train, y_train)

# Fitting된 모델로 x_valid를 통해 예측을 진행
rf_pred = rf_clf.predict(X_test)
```

	precision	recall	f1-score	support
OK	0.97	1.00	0.98	44887
Default	0.95	0.68	0.79	4489

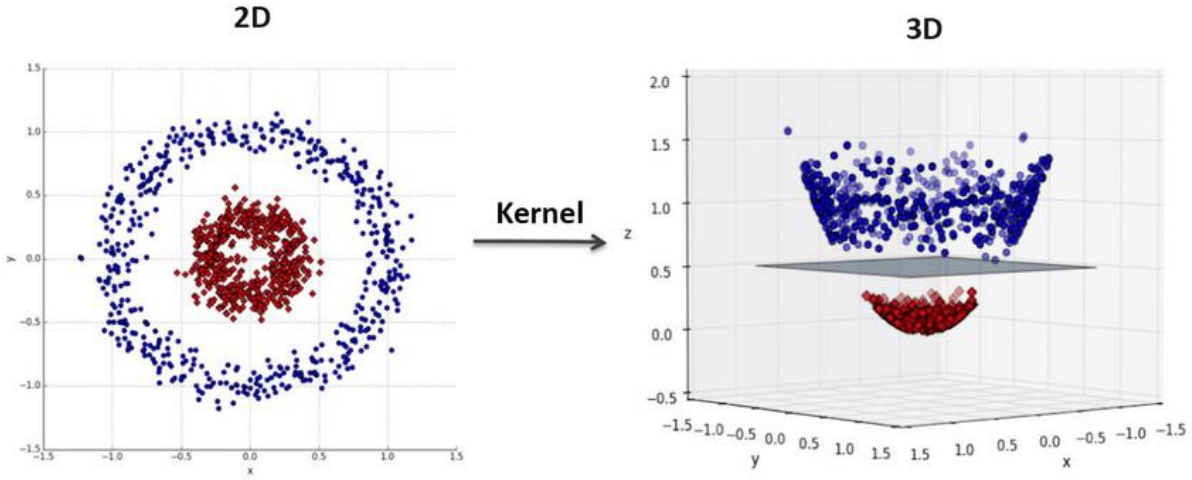
```
정확도: 0.9672, 정밀도: 0.9487, 재현율: 0.6759
```

```
roc_score = roc_auc_score(y_test[:,1].reshape(-1,1), rf_pred[:,1])
print('ROC AUC 값 : {:.4f}'.format(roc_score))
y_pred_proba = rf_clf.predict_proba(X_test) # 0 ~ 1 사이의 확률값으로 예측

ROC AUC 값 : 0.8361
```

4.8 SVM Model(Support Vector Machine)

분류와 회귀에 모두 사용할 수 있는 강력한 모델이지만, 학습하는데 많은 시간이(poly 커널의 경우, 80분 이상) 소요됨. 여러가지 파라미터 변경해가며 성능 확인해보고 싶었으나, 시간 제약상 2가지 case만 시도함.

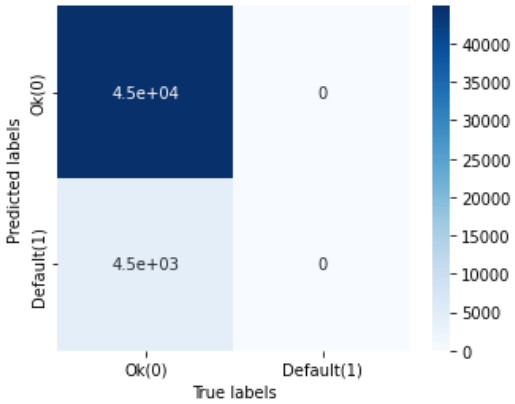
SVM모델에서 C(Cost, 비용) 파라미터 0.8와 Kernel을 Linear로 적용했을 때, Precision/Recall값 모두 0으로 나왔음. (* C : 경계를 조금씩 넘어서는 데이터를 어느정도까지 허용할지에 따라, Soft Margin과 Hard Margin을 만들어 낼 수 있다.)
커널을 Linear로 지정했으나, 우리의 문제가 선형으로 분류가 불가한 문제이기 때문에 이러한 데이터 나왔을 것으로 추정함. 저차원일 때에는 선형분리가 불가능하기 때문에, Kernel Trick으로 선형 분리가 불가능한 저차원(2D) 데이터를 고차원(3D 또는 무한대 차원?)으로 보내 선형 분리를 시도해보고자 함.




```
from sklearn import svm

svm_clf = svm.SVC(C = 0.8, kernel='linear')
# Linear : 선형 함수
# Poly : 다항식 함수
# RBF : 방사 기저 함수 (가우시안)
# Hyper-Tangent : 쌍곡선 탄젠트 함수
# ------------------------------------
# C (float) : 얼마나 모델에 규제를 넣을지 결정하는 값 (값이 작을수록 모델에 규제가 높아진다. Hard or Soft 결정 파라미터)
# degree (int) : Poly kernel 사용 시 차수를 결정하는 값
# kernel (str) : kernel trick에 사용할 kernel 종류
# random_state (int) : 내부적으로 사용되는 난수 값
# class_weight (dict) : 학습 시 클래스의 비율에 맞춰 손실 값에 가중치 부여 (분류 모델에서만 쓰인다.)
# gamma (float) : 모델이 생성하는 경계가 복잡해지는 정도 (값이 커질수록 데이터 포인터가 영향력을 행사하는 거리가 짧아져서 경계가 복잡해진다.)

svm_clf.fit(X_train, y_train[:,1])
svm_pred = svm_clf.predict(X_test)
```

사이킷런 SVM 주요 파라미터

파라미터	default	설명
C	1.0	오류를 얼마나 허용할 것인지 (규제항) 클수록 하드마진, 작을수록 소프트마진에 가까움
kernel	'rbf' (가우시안 커널)	'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'
degree	3	다항식 커널의 차수 결정
gamma	'scale'	결정경계를 얼마나 유연하게 그릴지 결정 클수록 오버피팅 발생 가능성 높아짐
coef0	0.0	다항식 커널에 있는 상수항 r



정확도: 0.9091, 정밀도: 0.0000, 재현율: 0.0000

	precision	recall	f1-score	support
OK	0.91	1.00	0.95	44887
Default	0.00	0.00	0.00	4489

Kernel을 Poly로 적용하여 2시간 넘게 학습하였으나, 동일한 결과(=precision과 recall 모두 0)가 나옴 Cost(C) 값을 조절 또는 Kernel을 변경하여 test해보려 했으나 너무 오랜 시간이 소요되어 일단 추가적인 튜닝작업은 중단하였음

```
svm_clf = svm.SVC(kernel='poly')
# linear : 선형 함수
# poly : 다항식 함수
# RBF : 방사 기저 함수
# Hyper-Tangent : 쌍곡선 탄젠트 함수
svm_clf.fit(X_train, y_train[:,1])
svm_pred = svm_clf.predict(X_test)
```

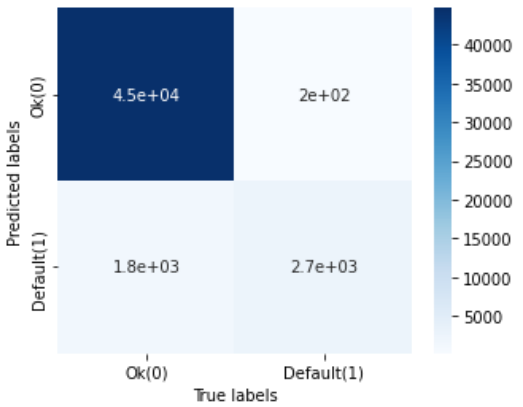
4.9 AdaBoost Model

AdaBoost는 매 스텝마다 이전 학습데이터에서 오차가 크거나 오분류된 데이터를 가중치를 크게 한다. 반대로 오차가 작거나 정분류된 데이터의 가중치를 낮게 한다.이러한 가중치에 비례하여 새로운 학습데이터를 복원 추출하여 새로운 학습데이터를 만들고 모형을 적합(fitting)한다. 즉 AdaBoost는 이전 모형이 제대로 예측하지 못한 데이터를 더 집중하여 이들을 더 잘 예측하는 모형을 만들게 된다.(약한 모형으로부터 시작하여 이전 모형의 약점을 보완한 새로운 모형들의 선형결합을 통해 강한 모형을 얻는다는 점에서 AdaBoost는 부스팅 알고르즘인 것이다.)

n_estimators를 300에서 200으로 줄이고, max_depth를 2에서 10으로 키운 후에 학습을 진행시켜보니 precision 0.93, 재현율 0.61 F1스코어 0.74를 기록하였다.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

adab_clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=None),
                              n_estimators=500,
                              random_state=10,
                              learning_rate=0.1)

adab_clf.fit(X_train, y_train[:,1])
adab_pred = adab_clf.predict(X_test)
```

```
정확도: 0.9605, 정밀도: 0.9392, 재현율: 0.6053
              precision    recall  f1-score   support

          OK       0.96      1.00      0.98     44887
```



Default	0.94	0.61	0.74	4489

일부 하이퍼파라미터 변경하면서 확인한 성능결과는 아래와 같습니다.

if n_estimators= 200, max_depth = 10, learning rate=0.3 then

precision 0.93, recall 0.61, f1-score 0.74

if n_estimators= 200, max_depth = 15, learning rate=0.3 then

precision 0.93, recall 0.61, f1-score 0.74

if n_estimators= 300, max_depth = 25, learning rate=0.3 then

precision 0.93, recall 0.60, f1-score 0.73

if n_estimators= 300, max_depth = 25, learning rate=0.1 then

precision 0.94, recall 0.61, f1-score 0.74

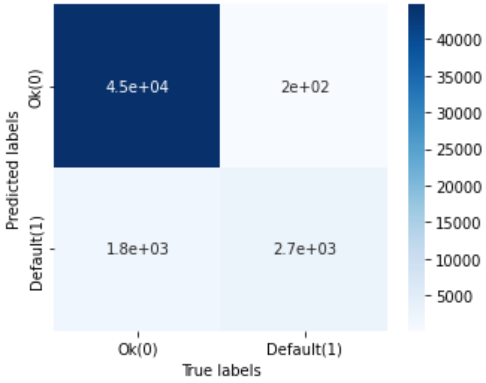
if n_estimators= 500, max_depth = None, learning rate=0.1 then

precision 0.93, recall 0.60, f1-score 0.73

4.10 GBM(Gradient Boost Machine) Model

GBM이 오차를 학습하기 위해 사용하는 학습기는 DecisionTreeRegressor입니다. Decision TreeRegressor의 불순도 조건은 'mse', 'mae' 등입니다. 따라서 그레이디언트 부스팅의 criterion 매개변수도 DecisionTreeRegressor의 불순도 조건을 따라서 'mse', 'mae', 그리고 제롬 프리드먼이 제안한 MSE 버전인 'friedman_mse'(기본값)를 사용합니다.

```
#AdaBoost와 비슷하나 가중치 업데이트를 경사하강법(Gradient Descent) 를 이용한다는 점이 큰 차이점이다.
from sklearn.ensemble import GradientBoostingClassifier
#grdb_clf = GradientBoostingClassifier()
grdb_clf = GradientBoostingClassifier(n_estimators=100, criterion='friedman_mse',
max_depth=5, max_leaf_nodes=None, min_samples_split=2, min_samples_leaf=1,
max_features=None)

grdb_clf.fit(X_train, y_train[:,1])
grdb_pred = grdb_clf.predict(X_test)
```



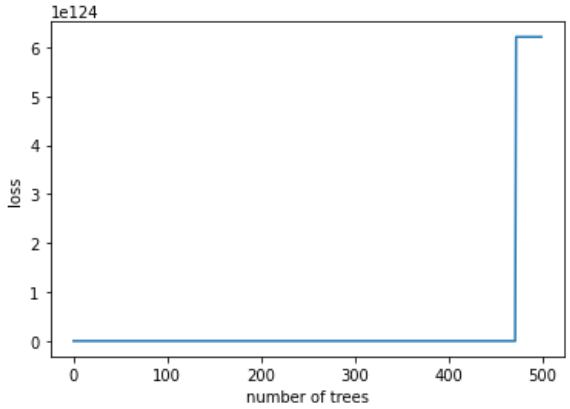
	precision	recall	f1-score	support
OK	0.96	1.00	0.98	44887
Default	0.93	0.60	0.73	4489

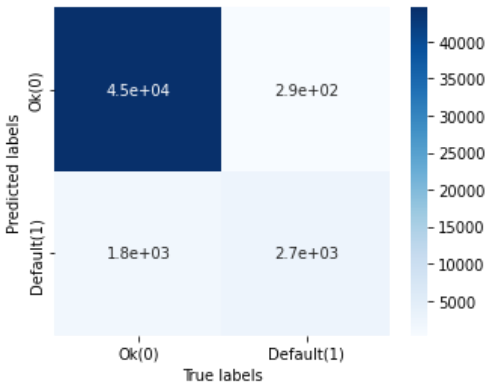
```
gbm_clf = GradientBoostingClassifier(learning_rate=0.005, n_estimators=500,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5)
# subsample 매개변수를 기본값 1.0보다 작은 값으로 지정하면 훈련 데이터셋에서
subsample 매개변수에 지정된 비율만큼 랜덤하게 샘플링하여 트리를 훈련합니다.
# 이를 확률적 그레이디언트 부스팅이라고 부릅니다. 이는 랜덤 포레스트나 에이다부스트의
부트스트랩 샘플링과 비슷하게 과대적합을 줄이는 효과를 냅니다.
# 또한, 남은 샘플을 사용하여 OOB 점수를 계산할 수 있습니다. subsample 매개변수가
1.0보다 작을 때 그레이디언트 부스팅 객체의 oob_improvement_ 속성에 이전 트리의 OOB
손실 값에서 현재 트리의 OOB 손실을 뺀 값이 기록되어 있습니다.
# 이 값에 음수를 취해서 누적하면 트리가 추가되면서 과대적합되는 지점을 찾을 수 있습니다.
```

tree가 늘어나면서 overfitting 되는지 확인하기 위해 누적 gradient boosting 객체의 oob_improvement_ (= 이전 tree의 손실값 - 현재 tree의 손실값)을 누적합 활용해 봄.
학습속도는 0.3으로, max tree(estimators)는 500으로 설정하여 학습진행 결과 확인

> ※ OOB(out of bag) 평가 : Bagging을 사용하면 어떤 데이터는 여러 번 사용되고, 어떤 것은 전혀 선택되지 않을 수 있습니다. 앞서 사용한 BaggingClassifier는 평균적으로 각 에측기에 Training dataset의 63% 정도만 사용하는데, 이때 사용되지 않은 나머지 data를 oob(out of bag) 데이터라고 합니다. 이렇게 남겨진 oob data들은 별도의 Validation set 없이 각 예측기를 평가하는데 사용됩니다. 앙상블 자체의 평가는 각 예측기의 oob 평가를 평균하여 얻습니다. sklearn에서는 oob_score=True로 지정하면, 자동으로 oob 평가를 수행합니다

```
gbm_clf = GradientBoostingClassifier(learning_rate=0.3, n_estimators=500,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5) # 50% 비율만큼 subsample 하여 overfitting 줄이는 효과

gbm_clf.fit(X_train, y_train[:,1])
grdb_pred = gbm_clf.predict(X_test)
```





정확도: 0.9579, 정밀도: 0.9024, 재현율: 0.6017

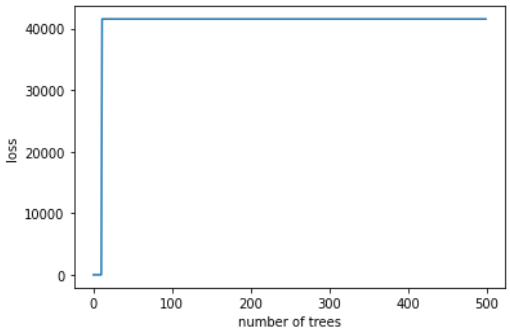
	precision	recall	f1-score	support
OK	0.96	0.99	0.98	44887
Default	0.90	0.60	0.72	4489

learning rate를 0.2로 줄였을 때, 정확도가 3%증가하여 f1-score 소폭 증가
다만 oob improvement 누적값을 확인해보면 여전히 손실값의 크기가 줄지 않는 문제 확인

```
gbm_clf = GradientBoostingClassifier(learning_rate=0.2, n_estimators=500,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5) # 50% 비율만큼 subsample 하여 overfitting 줄이는 효과
```

정확도: 0.9598, 정밀도: 0.9327, 재현율: 0.6017

	precision	recall	f1-score	support
OK	0.96	1.00	0.98	44887
Default	0.93	0.60	0.73	4489

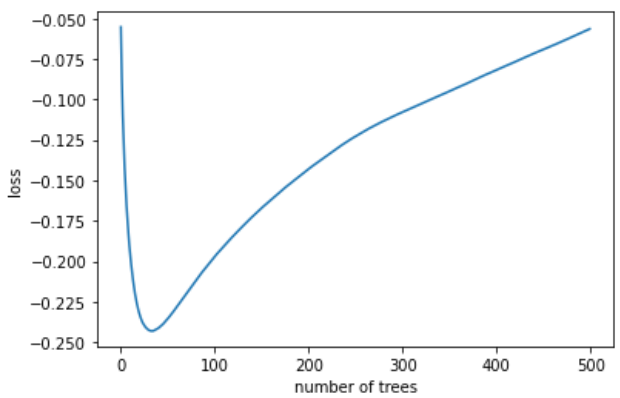


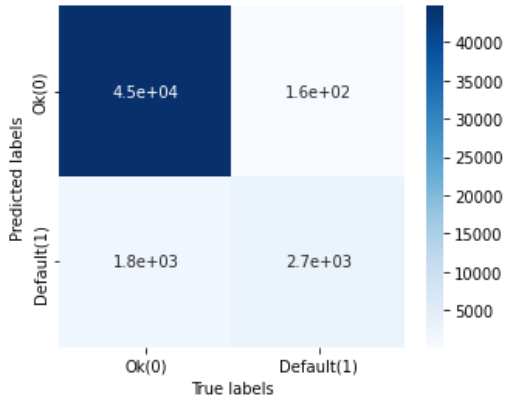
learning rate를 0.08로 줄여서 보다 촘촘히 학습 진행함.

학습 Tree(estimator)가 약 30개를 초과하는 경우 overfitting 발생하는 것 확인할 수 있음.

gradient boosting 객체의 oob_improvement_ (이전 tree의 손실값 - 현재 tree의 손실값)에서 현재 tree의 손실값이 증가하는 경우 oob_improvement객체의 값은 작아짐. 즉, `oob_loss = np.cumsum(-gbm_clf.oob_improvement_)` 와 같이 tree가 증가하면서 현재 tree의 손실값이 증가하는 경우, oob_improvent는 값이 작아진다. 여기에 (gbm_clf앞)에 -값을 취해줌으로써 아래와 같이 과적합되는 시점부터는 그래프가 우상향도록 표현함.

```
gbm_clf = GradientBoostingClassifier(learning_rate=0.08, n_estimators=500,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5) # 50% 비율만큼 subsample 하여 overfitting 줄이는 효과
```



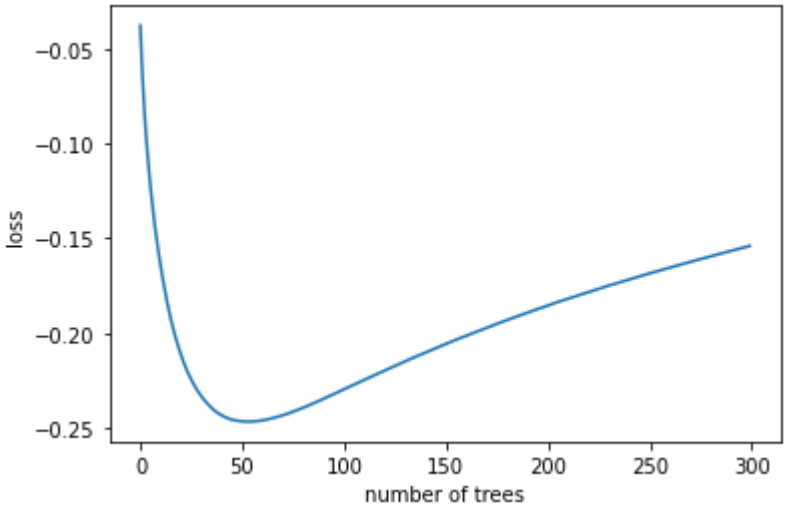


	precision	recall	f1-score	support
OK	0.96	1.00	0.98	44887

Default	0.94	0.60	0.73	4489

learning rate를 0.05, estimator 300에서 학습진행시, tree의 갯수가 50개를 넘어서면서 과대적합이 발생하고 있음을 확인.

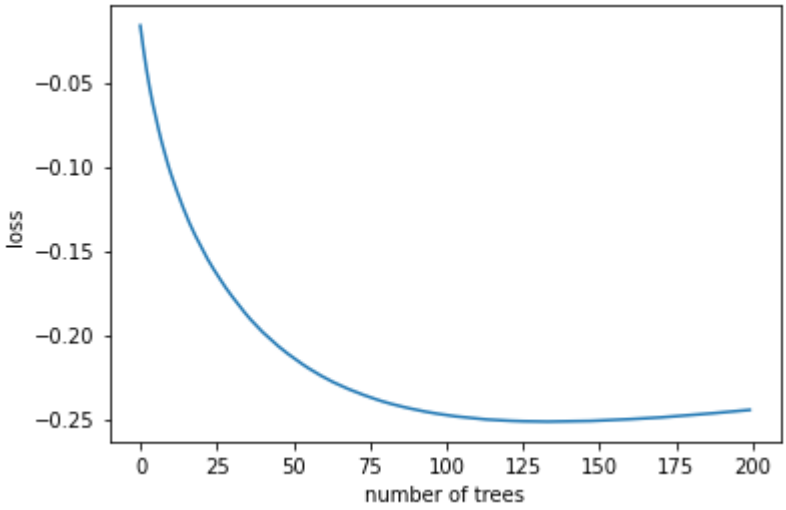
```
gbm_clf = GradientBoostingClassifier(learning_rate=0.05, n_estimators=300,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5) # 50%
비율만큼 subsample 하여 overfitting 줄이는 효과
```



	precision	recall	f1-score	support
OK	0.96	1.00	0.98	44887
Default	0.95	0.59	0.73	4489

learning rate를 0.02로 줄이고, estimator도 200으로 변경하여 학습한 결과 tree 150개를 초과하는 시점부터 loss가 증가하고 있음을 확인.

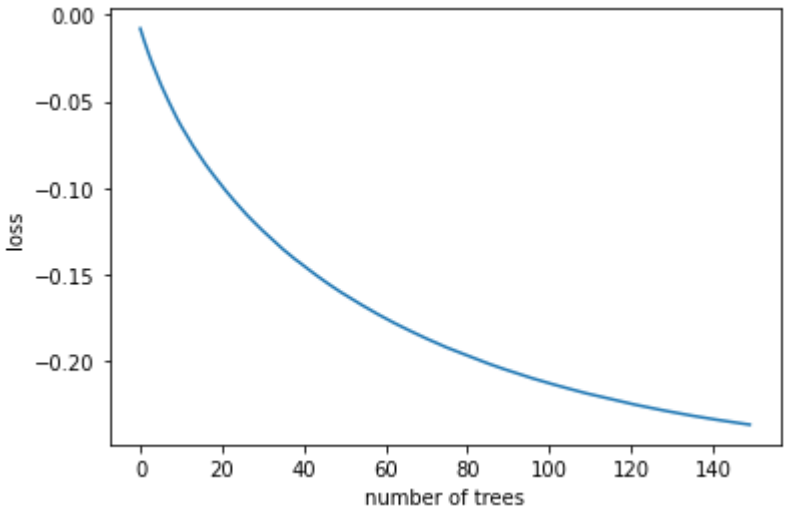
```
gbm_clf = GradientBoostingClassifier(learning_rate=0.02, n_estimators=200,
criterion='friedman_mse', max_depth=None, max_features=None, subsample=0.5)
```



	precision	recall	f1-score	support
OK	0.96	1.00	0.98	44887
Default	0.95	0.60	0.74	4489

learning rate를 0.01로 줄이고, estimator도 150으로 변경하여 학습한 결과

학습률이 낮아서인지, OOB평가를 통한 손실값(y축)이 -0.25에 다다르지 못하고(과적합이 발생하는 시점을 확인하지 못하고) 학습이 종료된 것을 확인할 수 있다. 손실값을 최소로 하지 못했기 때문인지 f1-score역시 크게 떨어진 결과를 확인 할 수 있었다.



	precision	recall	f1-score	support
OK	0.95	1.00	0.97	44887
Default	0.96	0.46	0.63	4489