2022 빅콘테스트
퓨처스부문

KUBIG A 팀

민윤기 MYK0115@NAVER.COM
김진서 JINSEO1018@KOREA.AC.KR
천원준 CJSDNJSWNS9@KOREA.AC.KR

목차

분석 배경 및 목적

- 핀다 어플은 대출을 필요로 하는 고객을 대상으로 통합 대출 관리 서비스를 제공.

- 본 연구의 목적은 핀다 어플을 사용하는 고객의 정보
  - 개인 신용 정보 및 접속 로그 데이터를 바탕으로 특정 고객의 대출 신청 여부를 예측하는 모델 개발

Raw Data Examination

#1 필요한 라이브러리 호출

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from collections import Counter
from IPython.core.display import display, HTML
sns.set_style('darkgrid')
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.metrics import f1_score
```

Raw Data Examination

#2 데이터 확인 및 분리

총 3개의 데이터 셋 (log_data.csv, loan_result.csv, user_spec.csv)으로 구분

```
log_data = pd.read_csv(r"C:₩Users₩82107₩Desktop₩Uni₩2022-2R₩KUBIG project₩log_data.csv")
loan_result = pd.read_csv(r"C:₩Users₩82107₩Desktop₩Uni₩2022-2R₩KUBIG project₩loan_result.csv")
user_spec = pd.read_csv(r"C:₩Users₩82107₩Desktop₩Uni₩2022-2R₩KUBIG project₩user_spec.csv")
```

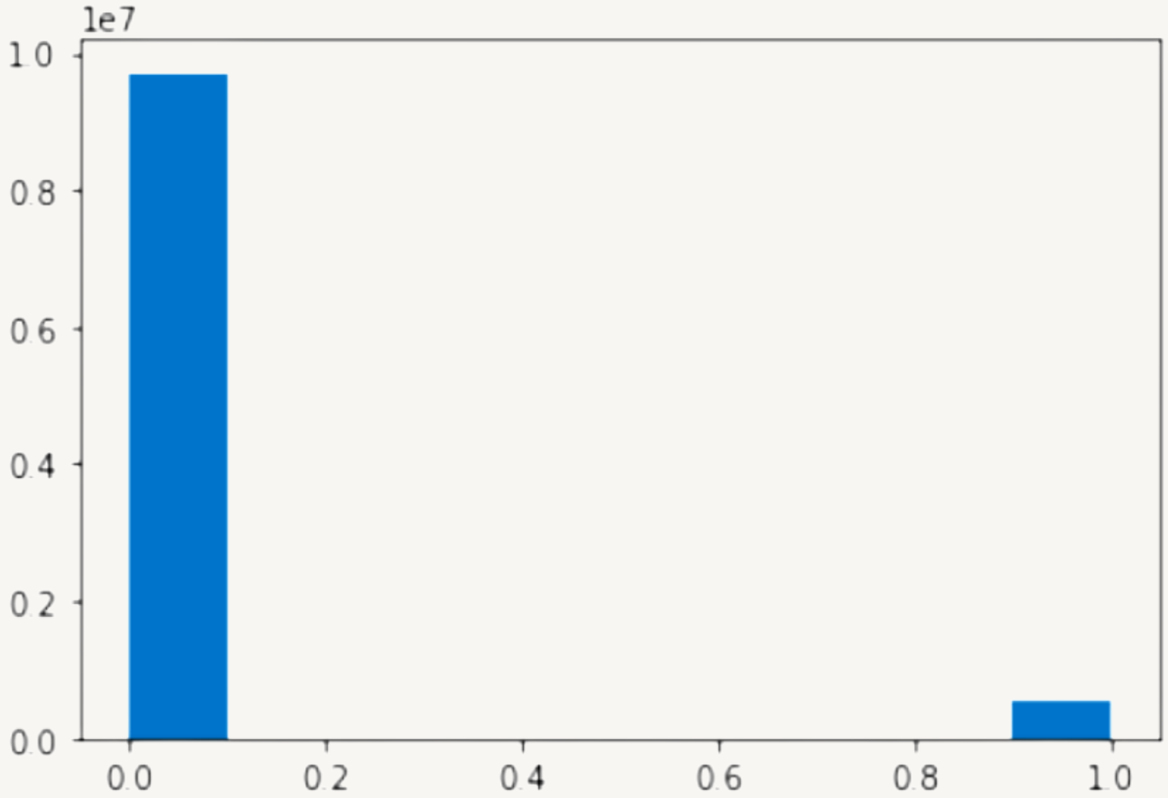
Loan_result, User_spec 데이터 셋 병합
Log_data는 크기가 매우 커서 이후에 다른 방법으로 병합

```
#loan_result를 기준으로 user_spec과 병합
test_loan = pd.merge(test_loan, test_user, on='application_id', how='left')
train_loan = pd.merge(train_loan, train_user, on='application_id', how='left')

test_loan = test_loan.drop_duplicates(['application_id','product_id'])

train_loan = train_loan.drop_duplicates(['application_id','product_id'])
```

Raw Data Examination

#2 데이터 확인 및 분리

Loan_result 파일에서 타겟 변수 확인
데이터 불균형이 심하기 때문에 이후 SMOTE 기법 활용



*SMOTE: 임의의 소수 클래스로부터 인근 소수의 클래스 사이에 새로운 데이터를 생성하는 오버샘플링 기법

Raw Data Examination

#2 데이터 확인 및 분리

Train/Test set 분리

```
#대출신청 데이터가 NaN이 아니면 train,
#이에 해당하는 대출신청서(application_id), 사용자(user_id)가 있는 세트를 추출
train_loan = loan_result[loan_result['is_applied'].notnull()]
train_user = user_spec.loc[user_spec['application_id'].isin(train_loan['application_id'])]
train_log = log_data.loc[log_data['user_id'].isin(train_user['user_id'])]

#대출신청 데이터가 NaN이면 test,
#이에 해당하는 대출신청서(application_id), 사용자(user_id)가 있는 세트를 추출
test_loan = loan_result[loan_result['is_applied'].isna()]
test_user = user_spec.loc[user_spec['application_id'].isin(test_loan['application_id'])]
test_log = log_data.loc[log_data['user_id'].isin(test_user['user_id'])]
```

Data Preprocessing / Data Preparation

#1 결측치/이상치 파악 및 처리

Train set의 결측치 제거: 결측치가 존재하는 row (loan_rate, loan_limit)를 제거한 후 확인해 본 결과, 데이터셋의 크기 변화가 크지 않으므로 그대로 진행

```
train_loan1 = train_loan.dropna(subset=na_train)
train_loan1 = train_loan1.dropna(subset=['loan_rate', 'loan_limit'])
test_loan = test_loan.dropna(subset=['loan_rate', 'loan_limit'])
print(train_loan.shape, train_loan1.shape, test_loan.shape)
```

Data Preprocessing / Data Preparation

#1 결측치/이상치 파악 및 처리

- Impute (NaN, Null 등을 어떤 값으로 채워주는 기능) 메소드 사용하여 결측치 대치
- 수치형의 경우에는 Iterative imputer 사용

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

```
# Imputer 적용이 필요한 column 추출
imp_need_col_test=test_loan1.loc[:,["birth_year", "gender", "credit_score", "company_enter_month", "existing_loan_cnt", "existing_loan_amt"]]
imp_need_col_train=train_loan2.loc[:,["birth_year", "gender", "credit_score", "company_enter_month", "existing_loan_cnt", "existing_loan_amt"]]
imp_need_df = pd.concat([imp_need_col_train, imp_need_col_train])
```

```
imputer = IterativeImputer(max_iter = 10, random_state=27)
imputed = imputer.fit(imp_need_col_train)
imputedtrain = imputer.transform(imp_need_col_train)
imputedTest = imputed.transform(imp_need_col_test)

imputed_train = pd.DataFrame(imputedtrain, columns=imp_need_col_train.columns)
imputed_test = pd.DataFrame(imputedTest, columns=imp_need_col_test.columns)
```

Data Preprocessing / Data Preparation

#1 결측치/이상치 파악 및 처리

- 범주형 결측치의 경우에는 트레인 셋의 mode로 대치

```
missing_cat = list(imp_need_col_train_cat.columns)
for i in missing_cat:
    imp_need_col_train_cat[i] = imp_need_col_train_cat[i].fillna(imp_need_col_train_cat[i].mode()[0])
    imp_need_col_test_cat[i] = imp_need_col_test_cat[i].fillna(imp_need_col_train_cat[i].mode()[0])
```

Data Preprocessing / Data Preparation

#1 결측치/이상치 파악 및 처리

크기가 큰 데이터 셋에서는 결측치 비율이 높은 column은 제거 하는 것이 효율적.
그러나 **데이터 손실 방지**를 위하여 **box-cox 변환** 활용,
train에서 매개변수 산출해 test에도 똑같이 적용

```
transform_column=['loan_limit','yearly_income','desired_amount','existing_loan_cnt','existing_loan_amt']
```

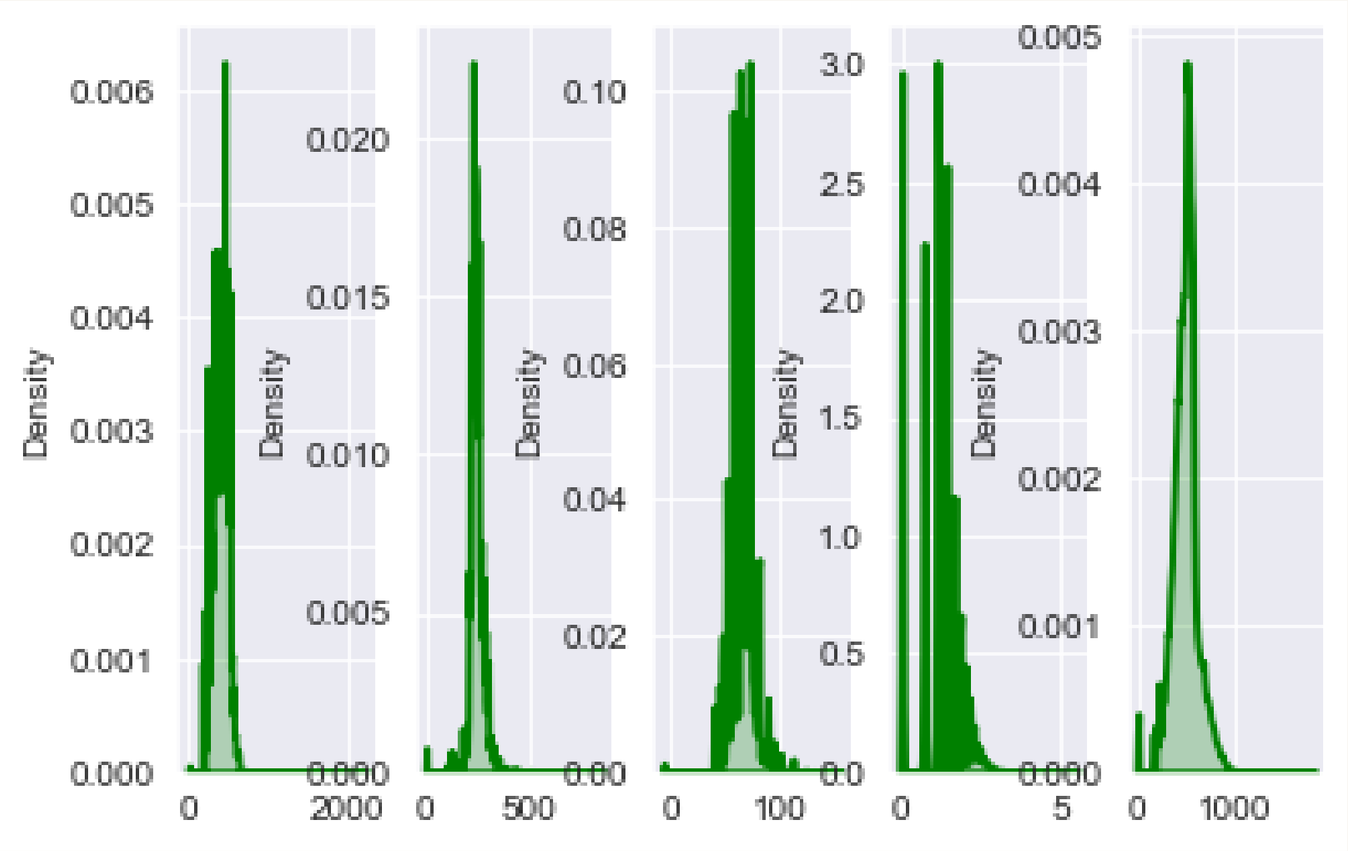
```
train2['loan_limit'] = train2['loan_limit'].clip(lower = 0.00001)
train2['yearly_income'] = train2['yearly_income'].clip(lower = 0.00001)
train2['desired_amount'] = train2['desired_amount'].clip(lower = 0.00001)
train2['existing_loan_cnt'] = train2['existing_loan_cnt'].clip(lower = 0.00001)
train2['existing_loan_amt'] = train2['existing_loan_amt'].clip(lower = 0.00001)
```

```
test2['loan_limit'] = test2['loan_limit'].clip(lower = 0.00001)
test2['yearly_income'] = test2['yearly_income'].clip(lower = 0.00001)
test2['desired_amount'] = test2['desired_amount'].clip(lower = 0.00001)
test2['existing_loan_cnt'] = test2['existing_loan_cnt'].clip(lower = 0.00001)
test2['existing_loan_amt'] = test2['existing_loan_amt'].clip(lower = 0.00001)
```

Data Preprocessing / Data Preparation

#1 결측치/이상치 파악 및 처리

데이터 손실 방지를 위하여 box-cox 변환 활용



*(좌)변환 후 분포 그래프
**(우) box-cox 변환 매개변수

```
print(f"Lambda value used for Transformation: {fitted_lambda_1_train}")
print(f"Lambda value used for Transformation: {fitted_lambda_2_train}")
print(f"Lambda value used for Transformation: {fitted_lambda_3_train}")
print(f"Lambda value used for Transformation: {fitted_lambda_4_train}")
print(f"Lambda value used for Transformation: {fitted_lambda_5_train}")
```

```
Lambda value used for Transformation: 0.27900191848685163
Lambda value used for Transformation: 0.23015083451814475
Lambda value used for Transformation: 0.133600581698523
Lambda value used for Transformation: 0.018613033494703998
Lambda value used for Transformation: 0.27287566906353156
```

Data Preprocessing / Data Preparation

#2 변수 처리 (더미 변수화)

'개인 회생' 변수에 '무응답' 추가 및 더미 변수화

```
#personal_rehabilitation 시리즈는 yes(1) or no(0)이기 때문에 범주형으로 변환, 결측치는 무응답(none)으로 대치
train_loan2 = train_loan1.replace({'personal_rehabilitation_yn': 0}, {'personal_rehabilitation_yn': 'no'})
train_loan2 = train_loan2.replace({'personal_rehabilitation_yn': 1}, {'personal_rehabilitation_yn': 'yes'})
train_loan2 = train_loan2.replace({'personal_rehabilitation_yn': np.nan}, {'personal_rehabilitation_yn': 'none'})
train_loan2 = train_loan2.replace({'personal_rehabilitation_complete_yn': 0}, {'personal_rehabilitation_complete_yn': 'no'})
train_loan2 = train_loan2.replace({'personal_rehabilitation_complete_yn': 1}, {'personal_rehabilitation_complete_yn': 'yes'})
train_loan2 = train_loan2.replace({'personal_rehabilitation_complete_yn': np.nan}, {'personal_rehabilitation_complete_yn': 'none'})
```

Label-Encoder 활용하여 분석에 사용할 수 있게 범주형으로 형태 변환

```
from sklearn.preprocessing import LabelEncoder
encoder=LabelEncoder()
label=list(train_loan2.select_dtypes(include = 'object').columns)
oneset=[train_loan2, test_loan1]

for data in oneset:
  for i in label:
    data[i] = encoder.fit_transform(np.array(data[i]))

test_loan1[label].head()
```

Data Preprocessing / Data Preparation

#3 로그 데이터 (Log_data.csv) 병합

- 로그 데이터에 존재하는 수많은 row를 그대로 병합하여 반영하는 것은 시간적으로나 여러모로 비효율적
- 이에 'event' 컬럼에서 '고객이 특정 행동을 하였는가'에 초점 맞추어 **파생변수** 생성
- 우선적으로 'EndLoanApply'는 자신의 대출한도를 조회하기 시작하여 완료까지 한 행동이므로 대출신청여부에 가장 유의미하게 작용할 것으로 추정
- 그리고 이 행동을 기준으로 타겟 및 행동 간 상관관계 분석한 결과, 'UseLoanManage'를 또 하나의 기준으로 적절하다고 판단함
- 따라서 사용자가 이 두 가지 행동을 했는지 여부에 따라 기존의 데이터셋과 결합

Data Preprocessing / Data Preparation

#3 로그 데이터 (Log_data.csv) 병합

```
Use = ['EndLoanApply', 'UseLoanManage']
Not_Use = list(set(log_data['event'].unique()) - set(Use))
train_loan6 = train_loan2.drop(Not_Use, axis=1)
train_loan6.info()

for i in Use:
  print(i)
  test_user = test_loan1[['user_id']]
  log_limit = log_data[log_data['event']==i]
  log_limit = log_limit[['user_id']]
  log_limit[i] = 1
  log_limit = log_limit.drop_duplicates(ignore_index = True)
  log_limit = log_limit.drop(log_limit[~log_limit['user_id'].isin(train_user['user_id'])].index)
  print(log_limit.shape)
  test_user = test_user.merge(log_limit, on='user_id', how='left')
  test_user = test_user.fillna(0)
  test_loan1[i] = test_user[i]
```

Data Preprocessing / Data Preparation

#4 SMOTE 적용

- 타겟 변수의 **불균형**을 처리하기 위해 **SMOTE** 기법을 사용
- SMOTE는 합성데이터를 생성하는 오버샘플링 기법으로 **가장 많이 사용**되고 있는 모델
- 다수 클래스를 샘플링하고 기존 소수 샘플을 보간하여 새로운 소수 인스턴스를 합성

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=1)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

print('SMOTE 적용  학습용 피처/레이블 데이터 세트: ', X_train_smote.shape, y_train_smote.shape)
print('SMOTE 적용 후 레이블 값 분포: ₩n', pd.Series(y_train_smote).value_counts())
```

```
SMOTE 적용  학습용 피처/레이블 데이터 세트:  (19432836, 24) (19432836,)
SMOTE 적용 후 레이블 값 분포:
 1.0    9716418
0.0    9716418
Name: is_applied, dtype: int64
```

Data Preprocessing / Data Preparation

#5 최종 Train/Test set 저장

```
train_loan6.to_csv(address+"train.csv")
test_loan1.to_csv(address+"test.csv")
```

Modeling

예측 모델을 기반으로 오차행렬과 정확도 및 F1 Score를 산출하여 각각의 모델 평가

Soft Voting을 활용하여 앙상블 모델 구축

K-Fold Stacking 기법을 사용하여 모델 검증

Modeling

#1 개별 모델링

- Logistic Regression : 선형 회귀와 달리 **범주형 데이터**도 예측할 수 있는 가장 대표적인 분류 모델
- Random Forest : **신용평가**에서 성능이 좋은 편 + **클래스 불균형** 문제에도 잘 대응하는 것으로 알려짐, 또한 **큰 데이터셋**에서도 잘 작동 → **메인 모델**로 사용 검토
- Decision Tree : 특정 기준에 따라 데이터를 구분하는 모델. **고차원 대형 데이터셋**에 강하다는 장점, Random Forest의 기반
- Naive Bayes : 빠르고 정확한 모델이나 모든 Feature가 독립이어야 한다는 한계
- LightGBM : 균형 트리 분할 기반 모델로 과적합에 강하지만
  균형 잡힌 트리를 만들기 위한 소요시간 큼

Modeling

#1 개별 모델링

Random Forest

```
# 랜덤 포레스트 임포트하여 학습 및 예측
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 10)
rf.fit(X_train_smote, y_train_smote)
rf_predict=rf.predict(X_test)
```

```
# 오차행렬 및 정확도, F1-Score 출력
rf_conf_matrix = confusion_matrix(y_test, rf_predict)
rf_acc_score = accuracy_score(y_test, rf_predict)
rf_f1_score = f1_score(y_test, rf_predict)
print(rf_conf_matrix)
print("정확도: ", rf_acc_score*100, "₩nF1 score:", rf_f1_score)
```

```
[[955609  15246]
 [ 44074  11461]]
정확도:  94.22052046493047
F1 score: 0.278714039055470 43
```

Modeling

#2 Soft Voting 앙상블

<Soft Voting A안>

Random Forest, Decision Tree, Logistic Regression 세 모델을 합쳐서 앙상블 시도

Random Forest로 안정적인 성능 확보, 보조적으로 Decision Tree를 통해 고차원 데이터에 대한 적응성을 올리고, Logistic Regression을 통해 overfitting을 방지

Tree 기반 모델이고 하이퍼파라미터를 넓게 설정하여 실행 시간과 메모리 사용량이 다소 크지만 성능이 많이 향상된 결과를 얻을 수 있음

Modeling

#2 Soft Voting 앙상블

Soft Voting A안

```
estimator1 = RandomForestClassifier(n_estimators = 50)
estimator2 = LogisticRegression()
estimator3 = DecisionTreeClassifier(max_depth = 15)
```

```
sv = VotingClassifier(estimators=[('RF', estimator1), ('LR', estimator2), ('DT', estimator3)], voting='soft')

sv.fit(X_train_smote, y_train_smote)
sv_predict = sv.predict(X_test)
```

```
sv_conf_matrix = confusion_matrix(y_test, sv_predict)
sv_acc_score = accuracy_score(y_test, sv_predict)
sv_f1_score = f1_score(y_test, sv_predict)
print(sv_conf_matrix)
print("정확도: ", sv_acc_score*100, "₩nF1 score:", sv_f1_score)
```

```
[[930529  40326]
 [ 36382  19153]]
정확도:  92.52642757626244
F1 score: 0.333055106334880 95
```

Modeling

#2 Soft Voting 앙상블

<Soft Voting B안>

Soft Voting A안에서 BernoulliNB 모델을 추가로 앙상블

BernoulliNB: 값이 0 또는 1인 이진형 자료에 사용하는 Naive Bayes 모델

4가지 모델을 앙상블 하는 대신 하이퍼 파라미터를 A안보다 축소시켜서 적용,
A안에 비해 실행 시간과 메모리 사용량은 적지만 성능이 다소 떨어지는 결과

Modeling

#2 Soft Voting 앙상블

Soft Voting B안

```
estimator4 = RandomForestClassifier(n_estimators = 5)
estimator2 = LogisticRegression()
estimator6 = DecisionTreeClassifier(max_depth = 10)
estimator7 = BernoulliNB()
```

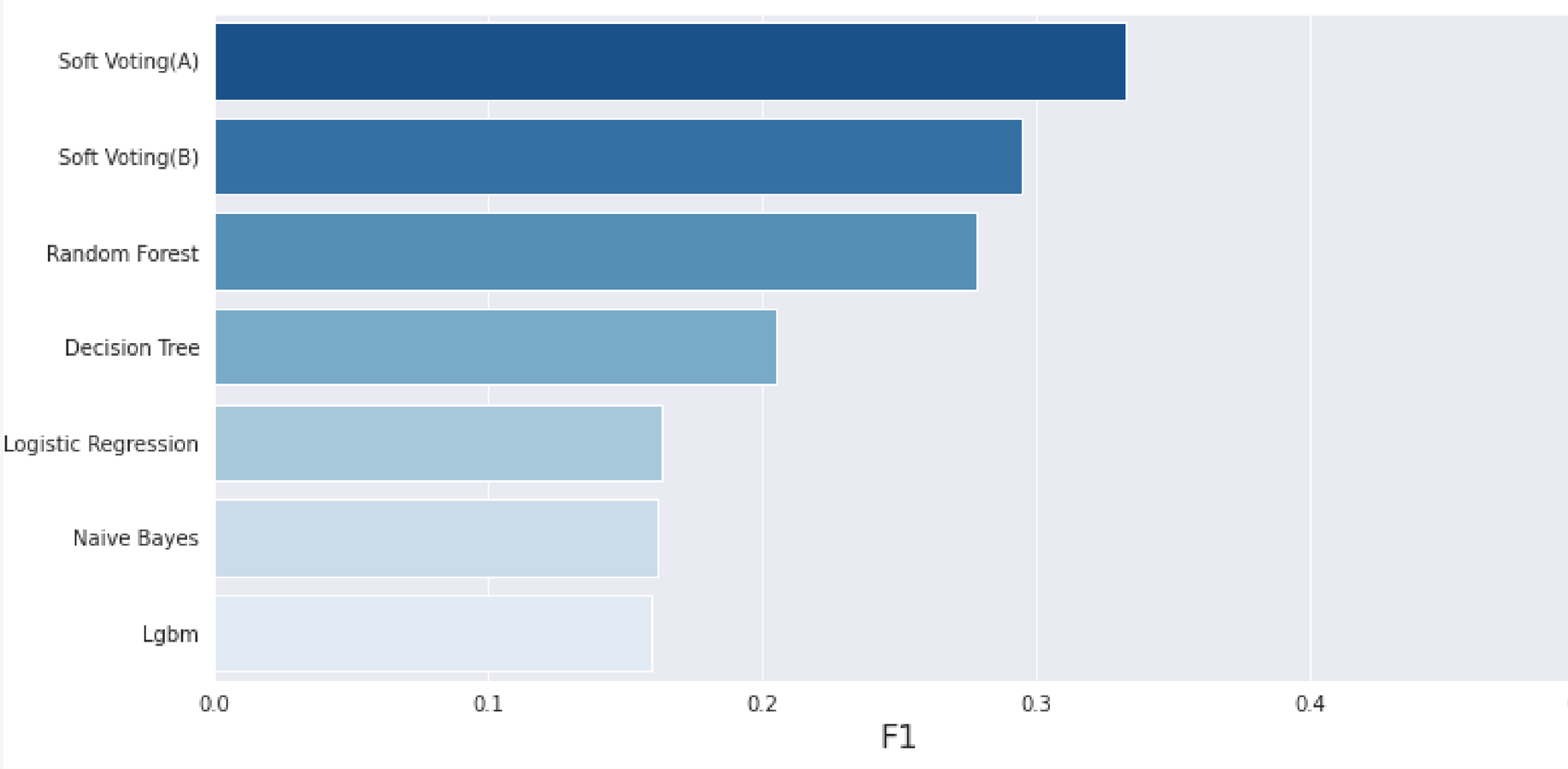
```
sv1_conf_matrix = confusion_matrix(y_test, sv1_predict)
sv1_acc_score = accuracy_score(y_test, sv1_predict)
sv1_f1_score = f1_score(y_test, sv1_predict)
print(sv1_conf_matrix)
print("정확도: ", sv1_acc_score*100, "₩nF1 score:", sv1_f1_score)
```

```
[[905556  65299]
 [ 34663  20872]]
정확도:  90.26081703835774
F1 score: 0.29458173965816553
```

Modeling

#2 Soft Voting 앙상블

단일&앙상블 모델 F1 Score 비교: 전반적으로 낮은 점수는 타겟의 **불균형** 때문으로 추정



Modeling

#3 K-Fold 적용

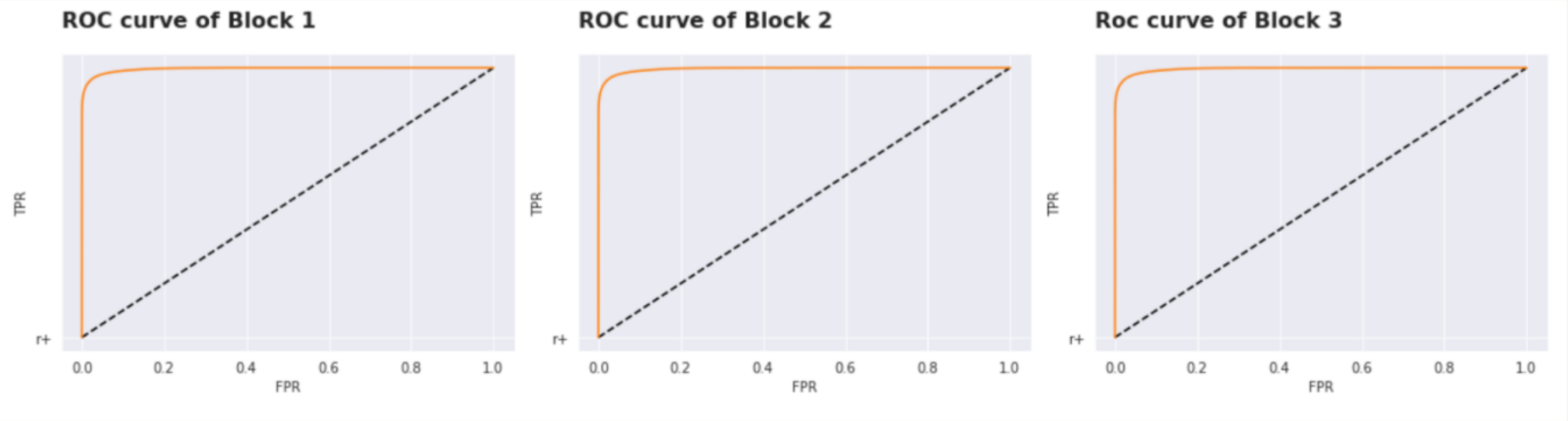
- 3 분할 방식으로 K-Fold stacking을 적용
- Stacking은 훈련이 된 여러 모델들을 잘 취합하여 최선의 결과를 기대하고 시행하는 방법으로, 데이터를 여러 갈래로(k fold)나눈 후 특정 set에 대해서 예측을 시행 즉, 나머지 set은 학습에 사용되지 않아 예측(검증)용 데이터로 사용 가능
- 이때 모델은 Soft Voting(A)에서 파라미터를 일부 축소하여 사용
- (랜덤 포레스트 estimators=15, 결정트리 max_depth=10)

```
kf = StratifiedKFold(n_splits = 3, shuffle = True, random_state = 50)

train_fold_predict = np.zeros((X_train_smote.shape[0], 1))
test_predict = np.zeros((test_loan4.shape[0], 3))
```

Modeling

#3 K-Fold 적용

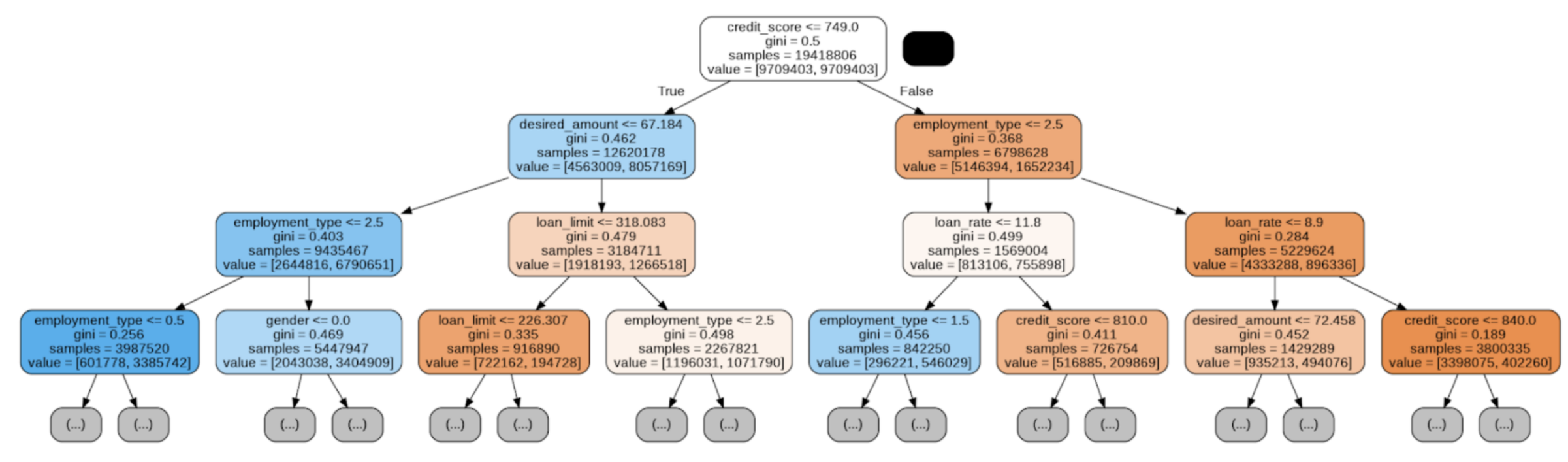
각 K-Fold 분할의 ROC Curve 확인,
SMOTE로 타겟 불균형을 해소한 데이터의 예측이므로 이전 모델 평가보다 성능 액면가 높음



시각화

#1 결정 트리 일부 시각화

- 고차원&대형 데이터 특성 상 설정한 깊이 최대치인 10까지 모두 사용
- 상위 4단계를 확인한 결과 credit_score, desired_amount, employment_type 변수와 loan 관련 변수가 가장 결정적으로 작용하고 있는 것으로 파악됨

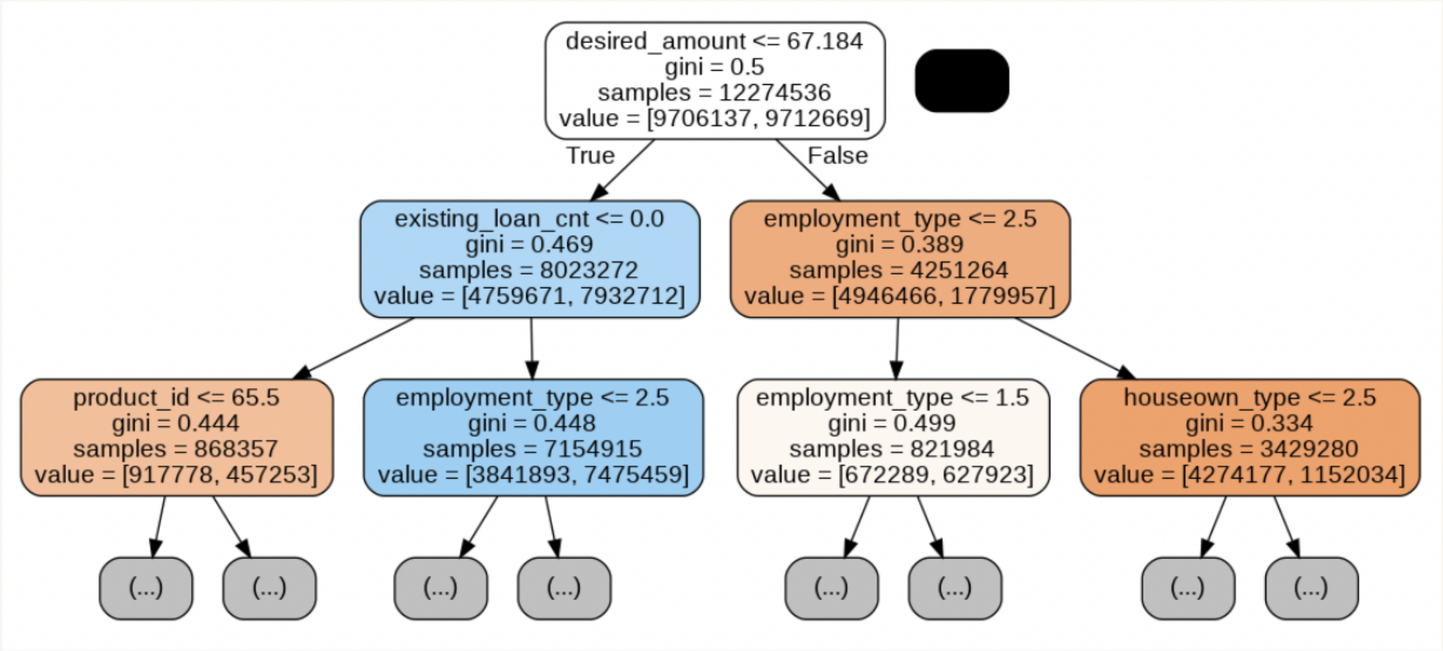


시각화

#2 랜덤 포레스트 일부 시각화

- 결정 트리와 같은 이유로 각 estimator의 상위 3단계만 확인
- 최상위 조건은 estimator 간 동일하나 세부 변수에서 차이 발생, desired_amount, employment_type 그리고 existing_loan 관련 변수가 가장 결정적으로 작용

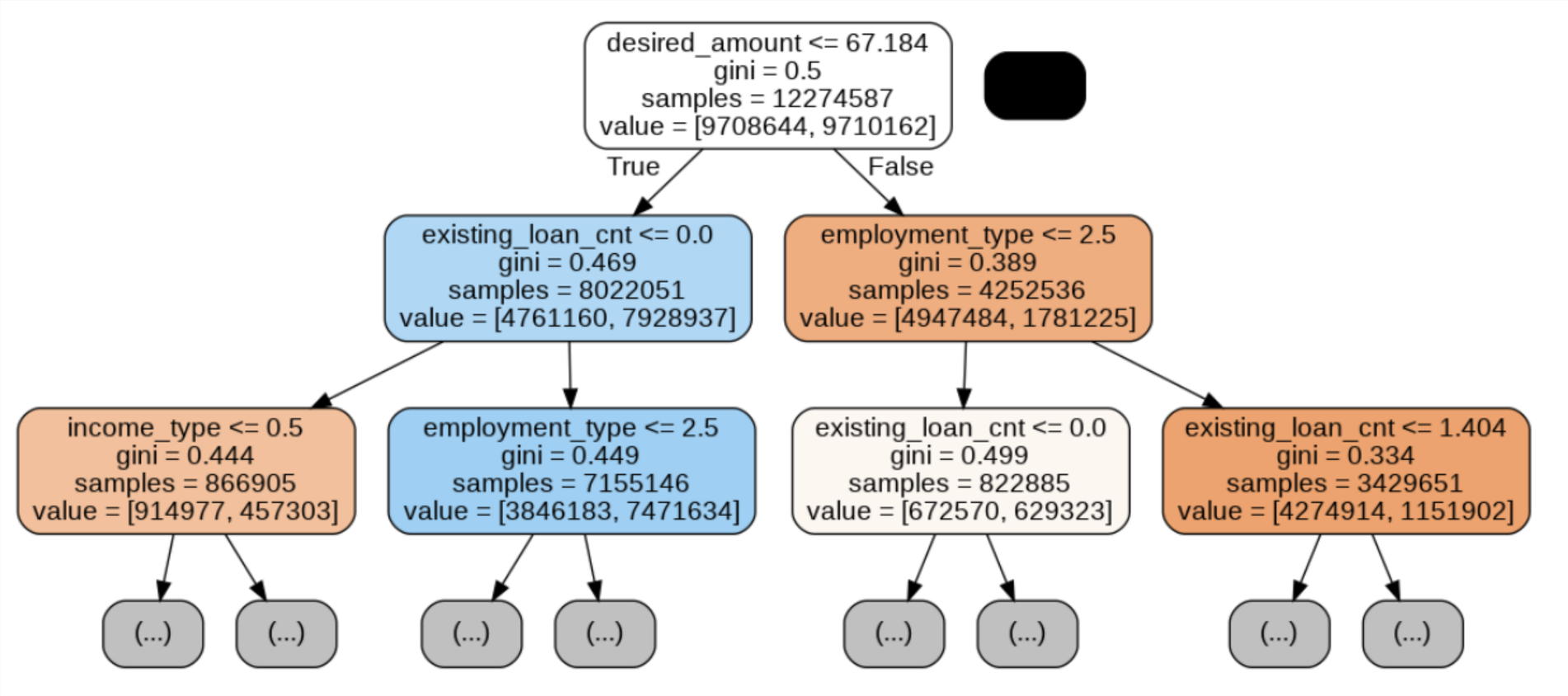
첫 번째 Estimator



시각화

#2 랜덤 포레스트 시각화

두 번째 Estimator

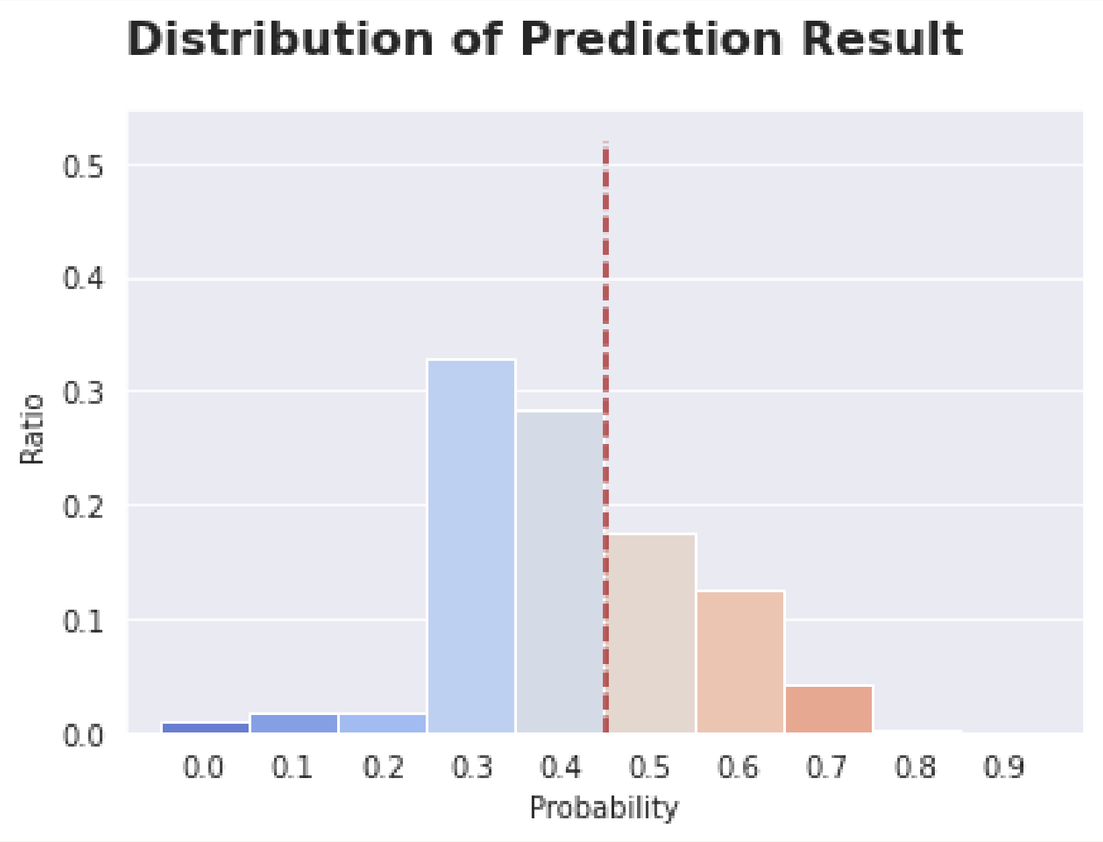


- 3단계 트리에서부터 첫 번째 Estimator와 차이 발생

시각화

#3 예측 결과 확률 분포 그래프

테스트 데이터에 대한 (타겟이 1일) 확률 예측값 분포가 어느 정도 중심에 모여있는 분포 경향 확인, 최종 모델이 데이터를 상대적으로 **신중하게 분류**하는 것으로 추측



분석 의의 및 한계점

- 본 연구에서는 특정 고객이 어플을 얼마나, 어떤 방식으로 사용하는지를 바탕으로 **대출 신청 여부**를 예측하는 모델을 구현
- Finda는 해당 예측 결과를 토대로 다양한 고객 집단에게 각각 **적합한 대출 상품**을 제공할 수 있을 것이며, 회사의 **인력을 절감**하는 효과 역시 거둬들일 것으로 기대
- 분석 내적으로, 통계적 원리 기반의 **효율적인 데이터 전처리**, 분석의 특성 가장 부합하는 모델 생성 및 앙상블을 통한 **성능 최적화**에서 의의
- 하지만 기존 데이터 셋이 매우 큰 관계로, 데이터 하나하나를 주의 깊게 살피기보다는 전체 데이터 셋을 **축소하는 방향**으로만 분석이 진행되었고, 특히 고객의 **로그 데이터**를 좀 더 유의미하게 활용하지 못한 한계 존재