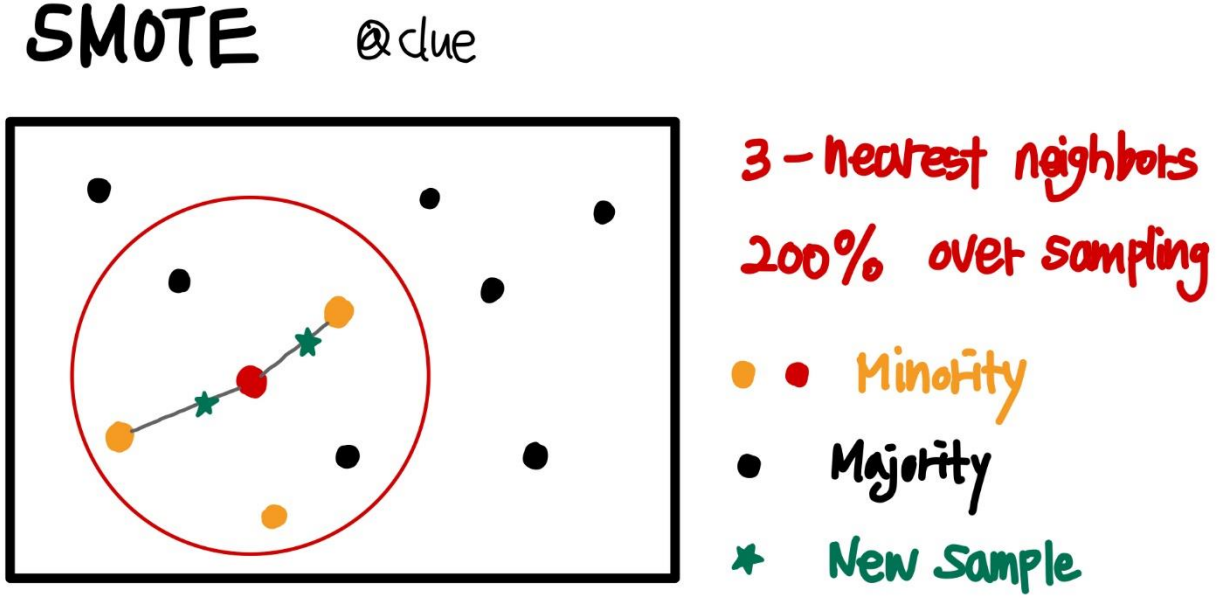
22510108

Data Mining-
데이터분석 최종 발표

Dataset Problem

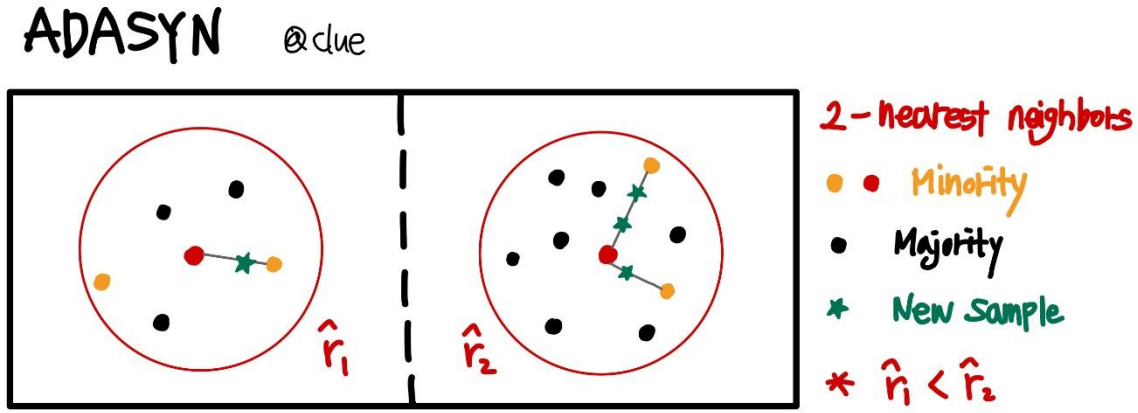
- 보건복지부와 한국보건산업진흥원이 개최한 “제8회 헬스케어 미래 포럼” 개최
- 데이터 빅뱅시대에 보건의료데이터와 인공지능이 나아갈 방향에 대한 주제 발표와 활용혁신 생태계 조성 방안에 대한 토론 진행
- 데이터 활용의 장애요인
  - ✓ 폐쇄·독점적인 활용 문화·행태
  - ✓ 학습에 사용할 데이터 부족,
  - ✓ 불신과 보상·거버넌스 미흡으로 인한 파이프라인 폐쇄
- 이와 마찬가지로 Pima-Indian-Diabetes 데이터셋의 크기가 (768,8)로 매우 작음
- 또한, Target의 Imbalance 문제도 존재
- Target의 불균형 문제 & 작은 크기의 데이터셋 문제를 해결하기 위해 Oversampling을 진행할 필요가 있다고 생각!

SMOTE(synthetic minority oversampling technique)



- minority class에서 synthetic 샘플을 생성하는 방법
- KNN을 이용한다는 점이 가장 큰 특징
- 작동원리
  1. KNN을 이용해 가까운 minority class 찾음
  2. 0과 1 사이의 랜덤값 선택
  3. 해당 값으로 내분하는 점을 새로이 생성
- 즉, minority class점을 선택해서 linear상의 랜덤한 초록색 별 위치를 새로운 샘플로 생성
- 단점 : Linear 상의 synthetic 샘플만 생성

ADASYN(Adaptive Synthetic Sampling Approach)



- minority class에서 synthetic 샘플을 생성하는 방법
- 각 관측치마다 생성하는 샘플의 수가 다르다.
- KNN 범위 내로 들어오는 majority class의 개수에 비례하도록 synthetic 샘플 수를 결정하고 생성
- 어려운 관측치에 집중하여 근방의 synthetic 샘플을 더 많이 생성하기 위해
- 즉, 한 minority class의 knn 내 majority 개수가 많다면 훈련 시 majority class와 비슷한 설명변수를 갖는 해당 minority class를 majority로 분류할 가능성이 높아질텐데, 더 많은 샘플을 생성함으로 해당 minority class가 무시되지 않도록 하는 것

Data Augmentation - Over Sampling

Python Code

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)
print(X_train.shape)

sm = SMOTE(random_state=0)
X_resampled2, y_resampled2 = sm.fit_resample(X_train,y_train)

print(X_resampled2.shape)

adasyn = ADASYN(random_state=0)
X_resampled3, y_resampled3 = adasyn.fit_resample(X_train,y_train)
print(X_resampled3.shape)
```

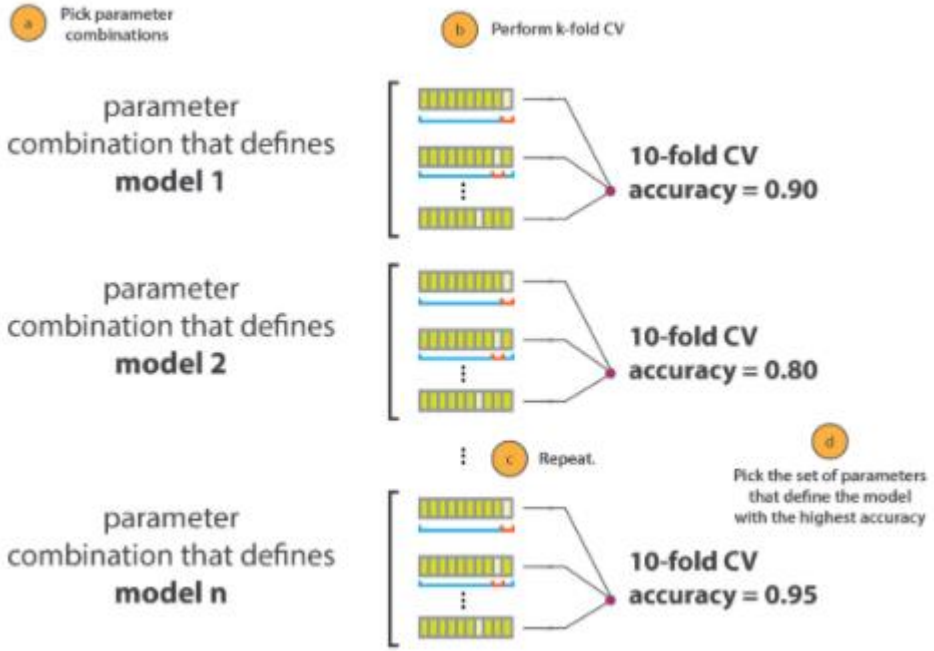
✓ 0.8s

```
(556, 8)
(748, 8)
(740, 8)
```

- 기존 데이터 셋 : (556,8)
- SMOTE 기법 적용 : (748, 8)
- ADASYN 기법 적용 : (740, 8)

Find Best Hyper parameters

GridSearchCV




```python
lr = LogisticRegression()
svc = SVC(probability=True)
dt = DecisionTreeClassifier()
rf = RandomForestClassifier()
xgb = XGBClassifier()

param_lr = {"penalty":["l1", "l2", "elasticnet", "none"]}
param_svc = {"kernel":["linear", "poly", "rbf", "sigmoid"]}
param_tree = {
    "max_depth" : [3, 4, 5, 6, 7],
    "min_samples_split" : [2, 3, 4],
    "min_samples_leaf" : [5,10, 15, 20,25]
}
param_xgb = { "n_estimators": range(25, 100, 25), "max_depth": [3, 4, 5, 6,7], "learning_rate": [0.0001, 0.001, 0.01, 0.1], "subsample": [0.7, 0.9]}



gs_lr = GridSearchCV(lr, param_grid=param_lr, cv=5, refit=True)
gs_svc = GridSearchCV(svc, param_grid=param_svc, cv=5, refit=True)
gs_dt = GridSearchCV(dt, param_grid=param_tree, cv=5, refit=True)
gs_rf = GridSearchCV(rf, param_grid=param_tree, cv=5, refit=True)
gs_xgb = GridSearchCV(xgb, param_grid=param_xgb, cv=5, refit=True)

gs_clfs = [gs_lr, gs_svc, gs_dt, gs_rf, gs_xgb]
fit_clasifiers(gs_clfs, X_train, y_train)
show_gridsearch_result(gs_clfs)
```


- 모델별로 최고의 성능을 만들어 줄 수 있는 hyper parameter값 탐색하는 방법
- 실험에 사용할 모델 선언(Logistic, SVM, Decision Tree, Random Forest, XGBoost)
- 각 모델별 하이퍼 파라미터 값을 설정
- 이를 앞서 생성한 각각의 데이터 셋에 적용

Find Best Hyper parameters

GridSearchCV

```
LogisticRegression()
0.8669240669240669
{'penalty': 'none'}
SVC(probability=True)
0.8849259974259974
{'kernel': 'rbf'}
DecisionTreeClassifier()
0.899259974259742
{'max_depth': 3, 'min_samples_leaf': 15, 'min_samples_split': 3}
RandomForestClassifier()
0.9083011583011583
{'max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 4}
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, gamma=None,
              gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, random_state=None,
              reg_alpha=None, reg_lambda=None, ...)
0.9047136422136421
{'learning_rate': 0.0001, 'max_depth': 4, 'n_estimators': 25, 'subsample': 0.9}
```

```
LogisticRegression()
0.858255033557047
{'penalty': 'l2'}
SVC(probability=True)
0.8930559284116331
{'kernel': 'rbf'}
DecisionTreeClassifier()
0.8716868008948546
{'max_depth': 6, 'min_samples_leaf': 10, 'min_samples_split': 4}
RandomForestClassifier()
0.8984161073825503
{'max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 3}
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, gamma=None,
              gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, random_state=None,
              reg_alpha=None, reg_lambda=None, ...)
0.9145055928411633
{'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 75, 'subsample': 0.7}
```

```
LogisticRegression()
0.8135362917096662
{'penalty': 'l2'}
SVC(probability=True)
0.8582129342965257
{'kernel': 'rbf'}
DecisionTreeClassifier()
0.8227124183006536
{'max_depth': 6, 'min_samples_leaf': 10, 'min_samples_split': 2}
RandomForestClassifier()
0.8673890608875128
{'max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 4}
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, gamma=None,
              gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, predictor=None, random_state=None,
              reg_alpha=None, reg_lambda=None, ...)
0.9067939456484349
{'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 75, 'subsample': 0.7}
```

- 기존 데이터 셋 : 로지스틱('penalty': 'none'), SVM('kernel': 'rbf'), 트리('max_depth': 3, 'min_samples_leaf': 15, 'min_samples_split': 3), 랜덤포레스트('max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 4), XGBoost('learning_rate': 0.0001, 'max_depth': 4, 'n_estimators': 25, 'subsample': 0.9)

- SMOTE 기법 적용 : 로지스틱('penalty': 'l2'), SVM('kernel': 'rbf'), 트리('max_depth': 6, 'min_samples_leaf': 10, 'min_samples_split': 2), 랜덤포레스트('max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 4), XGBoost('learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 75, 'subsample': 0.7)

- ADASYN 기법 적용 : 로지스틱('penalty': 'l2 '), SVM('kernel': 'rbf'), 트리('max_depth': 3, 'min_samples_leaf': 15, 'min_samples_split': 3), 랜덤포레스트('max_depth': 7, 'min_samples_leaf': 5, 'min_samples_split': 4), XGBoost('learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 75, 'subsample': 0.7)

Simulation Plan

- 각각의 모델과 데이터셋에 대하여 monte carlo simulation 진행
- 중심극한정리에 의해 충분한 설명력을 가질 수 있는 각 모델에 대해 100번의 에피소드 진행
- 해당 시뮬레이션을 통해 얻은 performance 값들의 평균값을 추출
- 해당 시뮬레이션 중 accuray와 F1을 활용해 가장 좋은 performance를 가지는 모델 저장
- 해당 모델을 시각화 & 파라미터 분석과 비교 진행!
- 모든 결과를 통합하여 가장 좋은 모델 선정 & 인사이트 도출

Logistic Classification

```
# 로지스틱 ADASYN
accuracy_list_adasyn = []
precision_list_adasyn = []
recall_list_adasyn  = []
f1_list_adasyn  = []
auc_list_adasyn  = []
performance = 0

for i in range(100):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)
    adasyn = ADASYN(random_state=0)
    X_resampled3, y_resampled3 = adasyn.fit_resample(X_train,y_train)

    # 로지스틱 회귀로 학습,예측 및 평가 수행.
    lr_clf3 = LogisticRegression(penalty='l2')#객체 생성
    lr_clf3.fit(X_resampled3 , y_resampled3) #학습
    pred3 = lr_clf3.predict(X_test) #예측
    # roc_auc_score 수정에 따른 추가
    pred_proba3 = lr_clf3.predict_proba(X_test)[:, 1] #예측 확률 array

    temp3 = get_clf_eval(y_test , pred3, pred_proba3)

    accuracy_list_adasyn.append(temp3[0])
    precision_list_adasyn.append(temp3[1])
    recall_list_adasyn.append(temp3[2])
    f1_list_adasyn.append(temp3[3])
    auc_list_adasyn.append(temp3[4])

    if performance < temp3[0] *temp3[3]:
        best_model_adasyn = lr_clf3
        performance = temp3[0] *temp3[3]

print('ADASYN')
print('평균 정확도: {0:.4f}, 평균 정밀도: {1:.4f}, 평균 재현율: {2:.4f}, 평균 F1: {3:.4f}, 평균 AUC:{4:.4f}'
      .format(np.mean(accuracy_list_adasyn ), np.mean(precision_list_adasyn ), np.mean(recall_list_adasyn ), np.mean(f1_list_adasyn ), np.mean(auc_list_adasyn)))
```

Soft Vector Machine

```
# SVC SMOTE
accuracy_list_smote = []
precision_list_smote  = []
recall_list_smote  = []
f1_list_smote  = []
auc_list_smote  = []
performance=0

for i in range(100):
    #학습 데이터 세트, 트레인 데이터 세트 분리
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)
    sm = SMOTE(random_state=0)
    X_resampled2, y_resampled2 = sm.fit_resample(X_train,y_train)

    # 로지스틱 회귀로 학습,예측 및 평가 수행.
    svc_clf2 = SVC(probability=True, kernel='rbf')  #객체 생성
    svc_clf2.fit(X_resampled2 , y_resampled2) #학습
    pred2 = svc_clf2.predict(X_test) #예측
    # roc_auc_score 수정에 따른 추가
    pred_proba2 = svc_clf2.predict_proba(X_test)[:, 1] #예측 확률 array

    temp2 = get_clf_eval(y_test , pred2, pred_proba2)

    accuracy_list_smote.append(temp2[0])
    precision_list_smote.append(temp2[1])
    recall_list_smote.append(temp2[2])
    f1_list_smote.append(temp2[3])
    auc_list_smote.append(temp2[4])

    if performance < temp2[0] *temp2[3]:
        best_model_smote = svc_clf2
        performance = temp2[0] *temp2[3]

print('SMOTE')
print('평균 정확도: {0:.4f}, 평균 정밀도: {1:.4f}, 평균 재현율: {2:.4f}, 평균 F1: {3:.4f}, 평균 AUC:{4:.4f}'
      .format(np.mean(accuracy_list_smote ), np.mean(precision_list_smote ), np.mean(recall_list_smote ), np.mean(f1_list_smote ), np.mean(auc_list_smote )))
```

Decision Tree

```
# DT pure
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
auc_list = []
performance=0

for i in range(100):
    #학습 데이터 세트, 트레인 데이터 세트 분리
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)

    # 로지스틱 회귀로 학습,예측 및 평가 수행.
    dt_clf = DecisionTreeClassifier(max_depth=4, min_samples_leaf=15, min_samples_split=2) #객체 생성
    dt_clf.fit(X_train , y_train) #학습
    pred = dt_clf.predict(X_test) #예측
    # roc_auc_score 수정에 따른 추가
    pred_proba = dt_clf.predict_proba(X_test)[:, 1] #예측 확률 array

    temp = get_clf_eval(y_test , pred, pred_proba)

    accuracy_list.append(temp[0])
    precision_list.append(temp[1])
    recall_list.append(temp[2])
    f1_list.append(temp[3])
    auc_list.append(temp[4])

    if performance < temp[0] *temp[3]:
        best_model = dt_clf

print('Pure')
print('평균 정확도: {0:.4f}, 평균 정밀도: {1:.4f}, 평균 재현율: {2:.4f}, 평균 F1: {3:.4f}, 평균 AUC:{4:.4f}'
      .format(np.mean(accuracy_list), np.mean(precision_list), np.mean(recall_list), np.mean(f1_list), np.mean(auc_list)))
```

Random Forest

```
# RF pure
accuracy_list = []
precision_list = []
recall_list = []
f1_list = []
auc_list = []


for i in range(100):
    #학습 데이터 세트, 트레인 데이터 세트 분리
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)

    # 로지스틱 회귀로 학습,예측 및 평가 수행.
    rf_clf = RandomForestClassifier(max_depth=7, min_samples_leaf=5, min_samples_split=4) #객체 생성
    rf_clf.fit(X_train , y_train) #학습
    pred = rf_clf.predict(X_test) #예측
    # roc_auc_score 수정에 따른 추가
    pred_proba = rf_clf.predict_proba(X_test)[:, 1] #예측 확률 array

    temp = get_clf_eval(y_test , pred, pred_proba)

    accuracy_list.append(temp[0])
    precision_list.append(temp[1])
    recall_list.append(temp[2])
    f1_list.append(temp[3])
    auc_list.append(temp[4])
print('Pure')
print('평균 정확도: {0:.4f}, 평균 정밀도: {1:.4f}, 평균 재현율: {2:.4f}, 평균 F1: {3:.4f}, 평균 AUC:{4:.4f}'
        .format(np.mean(accuracy_list), np.mean(precision_list), np.mean(recall_list), np.mean(f1_list), np.mean(auc_list)))
```

XGBoost

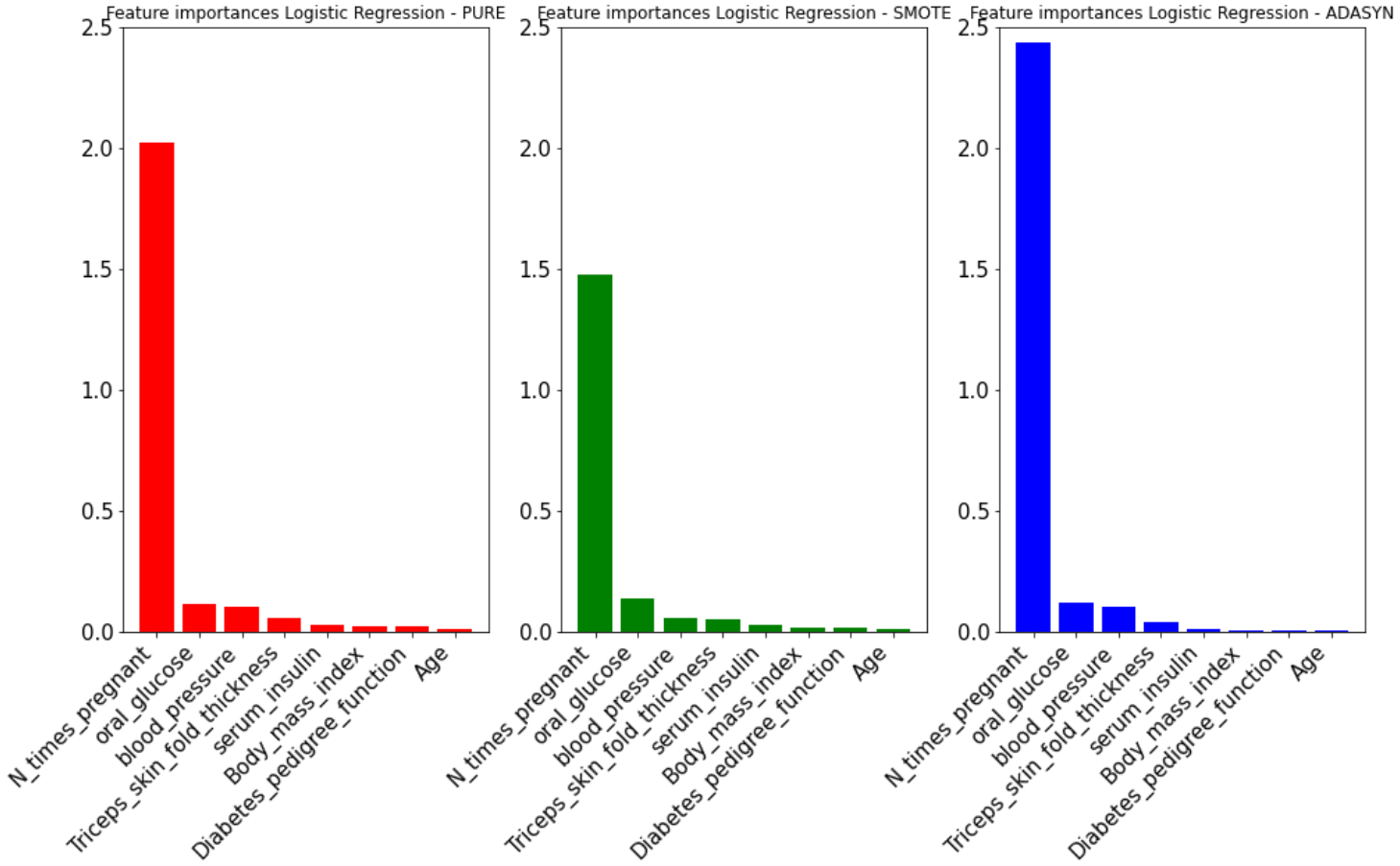
```
# RF ADASYN
accuracy_list_adasyn = []
precision_list_adasyn = []
recall_list_adasyn  = []
f1_list_adasyn  = []
auc_list_adasyn  = []
performance=0

for i in range(100):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, stratify=y)
    adasyn = ADASYN(random_state=0)
    X_resampled3, y_resampled3 = adasyn.fit_resample(X_train,y_train)

    # 로지스틱 회귀로 학습,예측 및 평가 수행.
    xgb_clf3 = XGBClassifier(learning_rate = 0.1, max_depth=7, n_estimators = 75, subsample = '0.7')
    xgb_clf3.fit(X_resampled3 , y_resampled3) #학습
    pred3 = xgb_clf3.predict(X_test) #예측
    # roc_auc_score 수정에 따른 추가
    pred_proba3 = xgb_clf3.predict_proba(X_test)[:, 1] #예측 확률 array

    temp3 = get_clf_eval(y_test , pred3, pred_proba3)

    accuracy_list_adasyn.append(temp3[0])
    precision_list_adasyn.append(temp3[1])
    recall_list_adasyn.append(temp3[2])
    f1_list_adasyn.append(temp3[3])
    auc_list_adasyn.append(temp3[4])

    if performance < temp3[0] *temp3[3]:
        best_model_adasyn = xgb_clf3

print('ADASYN')
print('평균 정확도: {0:.4f}, 평균 정밀도: {1:.4f}, 평균 재현율: {2:.4f}, 평균 F1: {3:.4f}, 평균 AUC:{4:.4f}'
      .format(np.mean(accuracy_list_adasyn ), np.mean(precision_list_adasyn ), np.mean(recall_list_adasyn ), np.mean(f1_list_adasyn ), np.mean(auc_list_adasyn)))
```

Model Performance Result

	평균 정확도	평균 정밀도	평균 재현율	평균 F1	평균 AUC
logistic Pure	0.848427	0.805423	0.716089	0.755749	0.814636
logistic SMOTE	0.831500	0.713629	0.819777	0.761742	0.828515
logistic ADASYN	0.789860	0.637830	0.846519	0.726296	0.804319

- 평균 정확도는 pure 모델이, F1 스코어는 SMOTE가 가장 좋게 나왔다.
- 단순한 모델임에도 정확도가 80% 이상으로 상당히 좋은 성능을 보이고 있다.
- 하지만 Pure와 SMOTE 모델들의 평균 1% 정도의 차이이기에 큰 performance의 차이가 있는 것은 아니다.
- 하지만 ADASYN 같은 경우 다른 모델에 비해 월등히 낮은 성능을 보인다
- 실험 시간은(모델 300개 학습) 총 Time: 7.1730sec이 걸렸다.

Logistic Classification



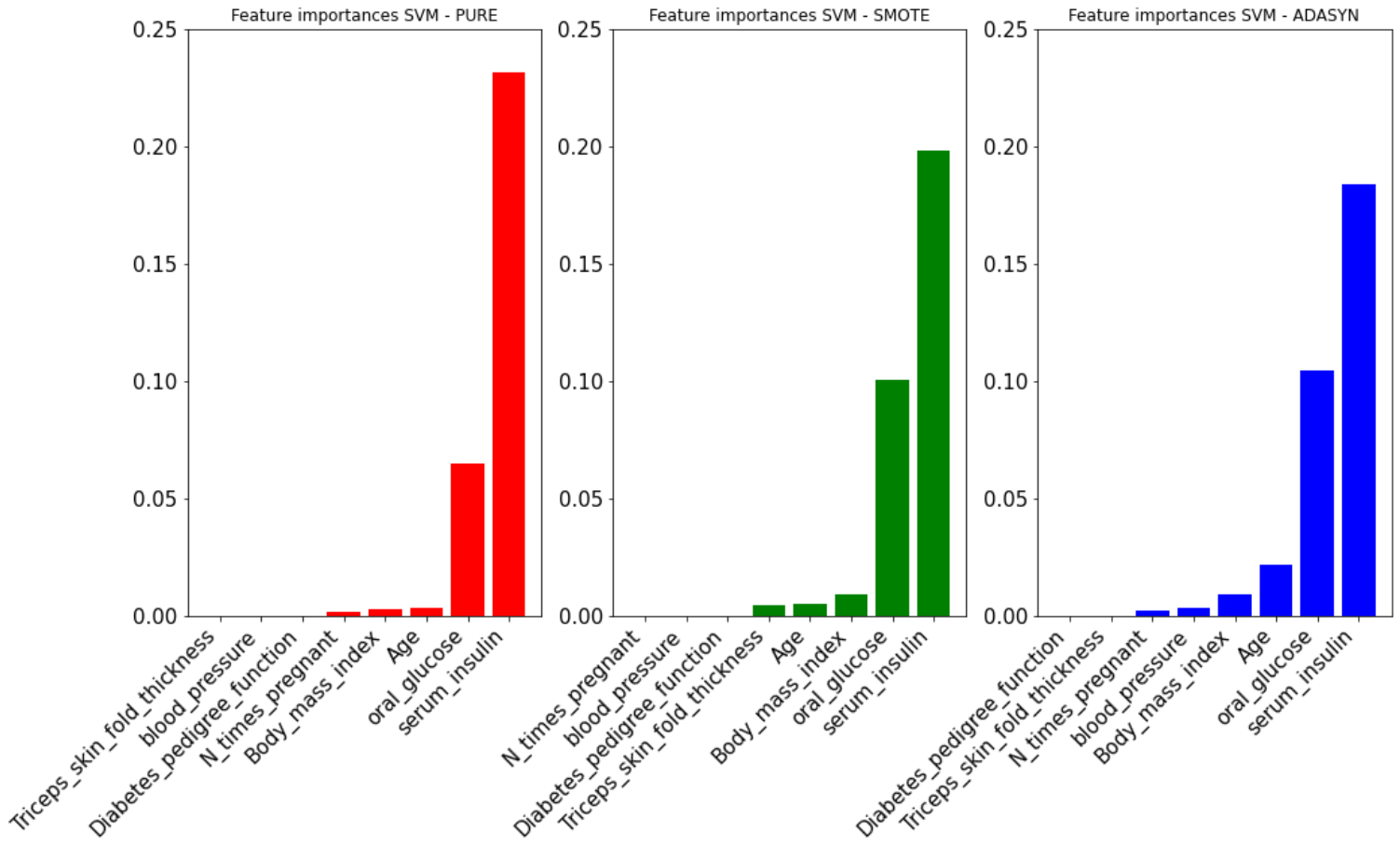
- 세 모델 상관 계수의 크기 순서는 같다 (‘N_times_pregnant’ => ‘oral_glucose’ => ‘blood_pressure’ ,,,)
- 다른 모델에 비해 ADASYN 모델이 ‘N_times_pregnant’ 영향력이 크다

Model Performance Result

	평균 정확도	평균 정밀도	평균 재현율	평균 F1	평균 AUC
logistic Pure	0.848427	0.805423	0.716089	0.755749	0.814636
logistic SMOTE	0.831500	0.713629	0.819777	0.761742	0.828515
logistic ADASYN	0.789860	0.637830	0.846519	0.726296	0.804319
SVM Pure	0.876998	0.933208	0.676308	0.781253	0.825758
SVM SMOTE	0.873431	0.827069	0.781080	0.801929	0.849852
SVM ADASYN	0.821071	0.675385	0.888034	0.765962	0.838176

- SVM은 전반적으로 봤을 때, SMOTE 기법이 가장 성능이 좋았다.
- Logistic 모델에 비해 성능이 약 4% 정도로 상당히 올랐다
- Logistic 모델과 마찬가지로 ADASYN 데이터 셋의 성능이 가장 안좋았다.
- 실험 시간은 총 16.1354sec으로 앞선 로지스틱 모델에 비해 2배 걸렸다.
- 하지만 복잡한 모델이지만 데이터셋이 작기에 학습 시간이 상당히 짧게 걸렸다고 평가 가능하다.

Soft Vector Machine



- 세 모델 상관 계수의 크기 순서는 앞선 2개 (인슐린, 구내혈당)는 같다
- 뒤에 오는 3-4번째는 다르다(pure & adasyn : 나이 -> BMI,
  smote : BMI->나이)
- 나머지 변수들은 거의 사용되지 않았다.

Model Performance Result

	평균 정확도	평균 정밀도	평균 재현율	평균 F1	평균 AUC
logistic Pure	0.848427	0.805423	0.716089	0.755749	0.814636
logistic SMOTE	0.831500	0.713629	0.819777	0.761742	0.828515
logistic ADASYN	0.789860	0.637830	0.846519	0.726296	0.804319
SVM Pure	0.876998	0.933208	0.676308	0.781253	0.825758
SVM SMOTE	0.873431	0.827069	0.781080	0.801929	0.849852
SVM ADASYN	0.821071	0.675385	0.888034	0.765962	0.838176
Decision Tree Pure	0.880283	0.918847	0.703922	0.793843	0.835250
Decision Tree SMOTE	0.851730	0.766774	0.795964	0.778532	0.837432
Decision Tree ADASYN	0.792002	0.660799	0.817173	0.724068	0.798427

- DT 는 PURE 기법이 가장 성능이 좋았다.
- 단순한 모델임에도 정확도가 88%로 매우 좋은 성능을 보이고 있다.
- ADASYN 같은 경우 다른 기법과 같이 다른 모델에 비해 월등히 낮은 성능을 보인다
- 실험 시간은(모델 300개 학습) 총 Time: 2.5983sec이 걸렸다.

SHAP(Shapely Additive exPlanations)

- Lundberg and Lee(2017). NeurIPS 논문에서 처음 제안

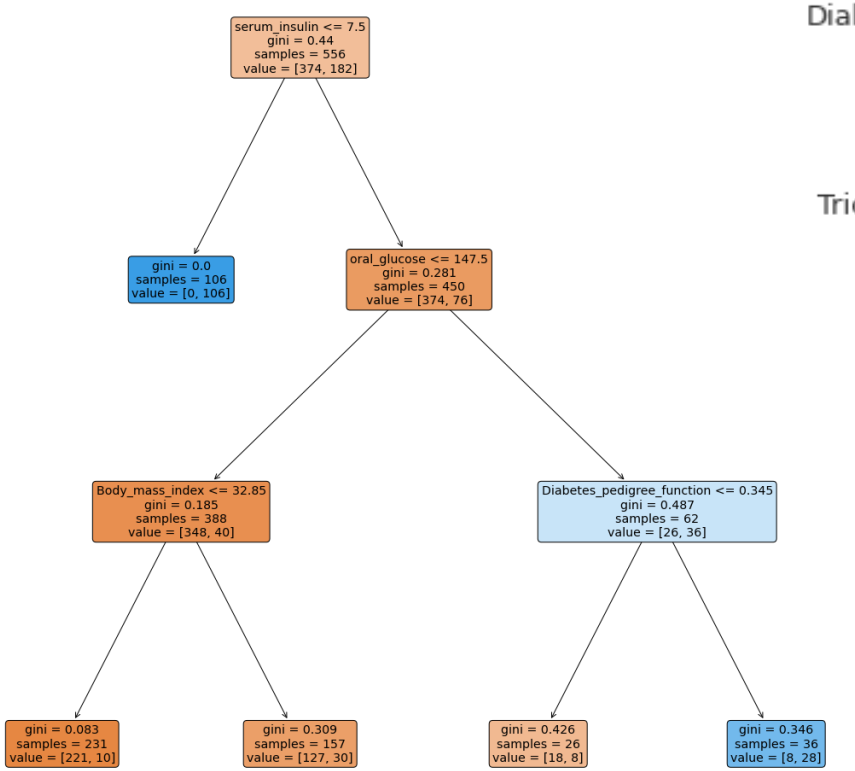
$$\phi_i = \sum_{S \subseteq F/\{i\}} \frac{|S|!\,(|F| - |S| - 1)!}{|F|!} \left(f(S \cup \{i\}) - f(S)\right)$$

- ϕ_i : 특정 변수의 Shapley value
- i : 관심 있는 변수 집합
- S : 관심 변수가 제외된 변수 부분집합
- F : 전체 변수의 부분 집합

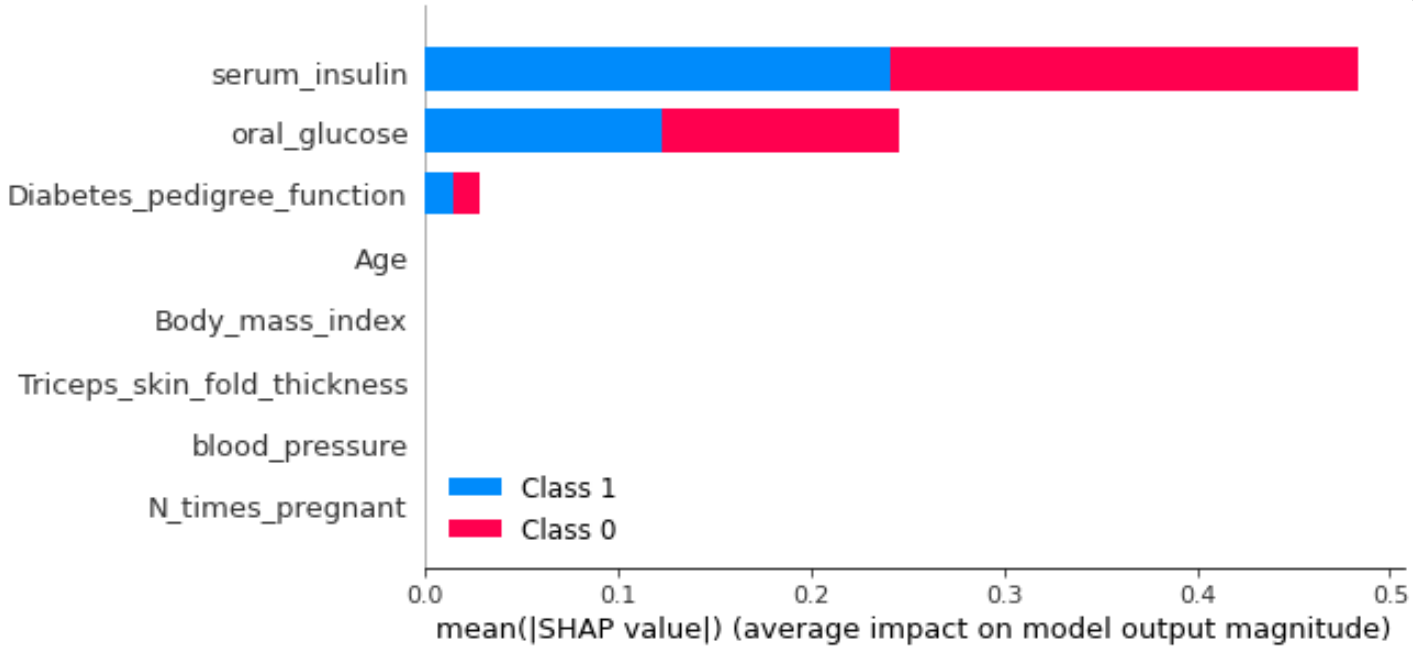
- Shapley Value를 사용하여 Additive Method를 만족시키는 설명 모델
- 전체 성과(판단)를 창출하는 데 각 참여자(피쳐)가 얼마나 공헌했는지 수치로 표현

Model Visualization

Decision Tree



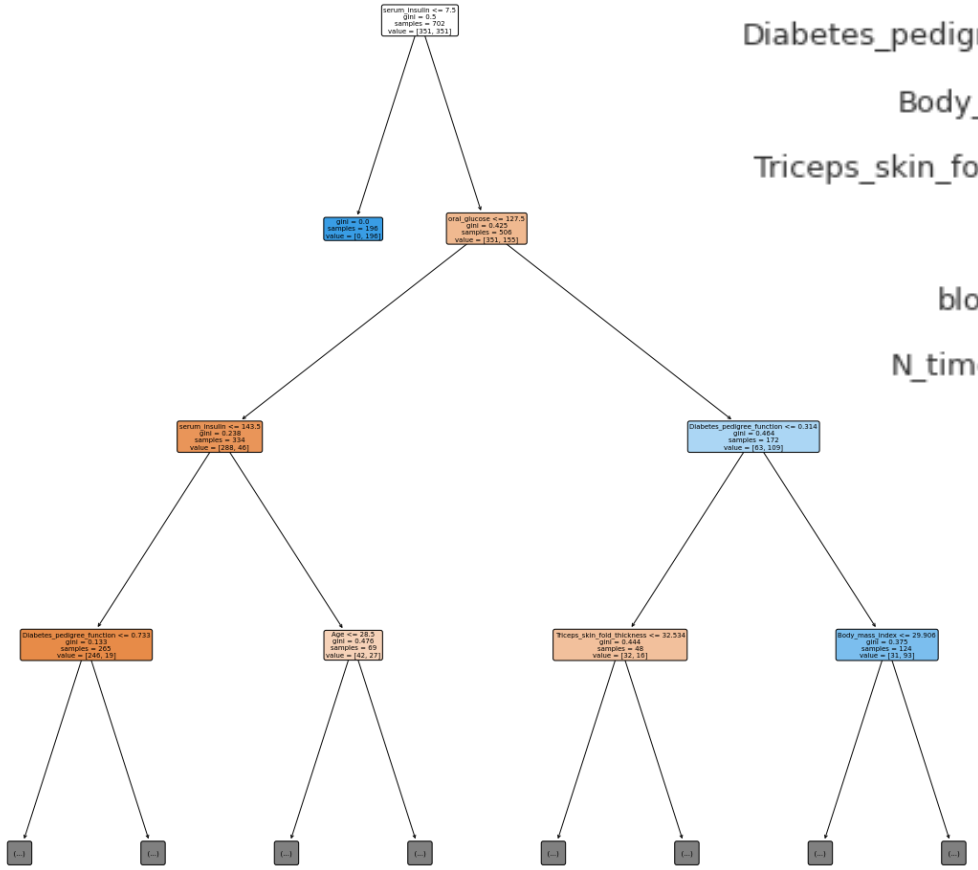
Pure tree



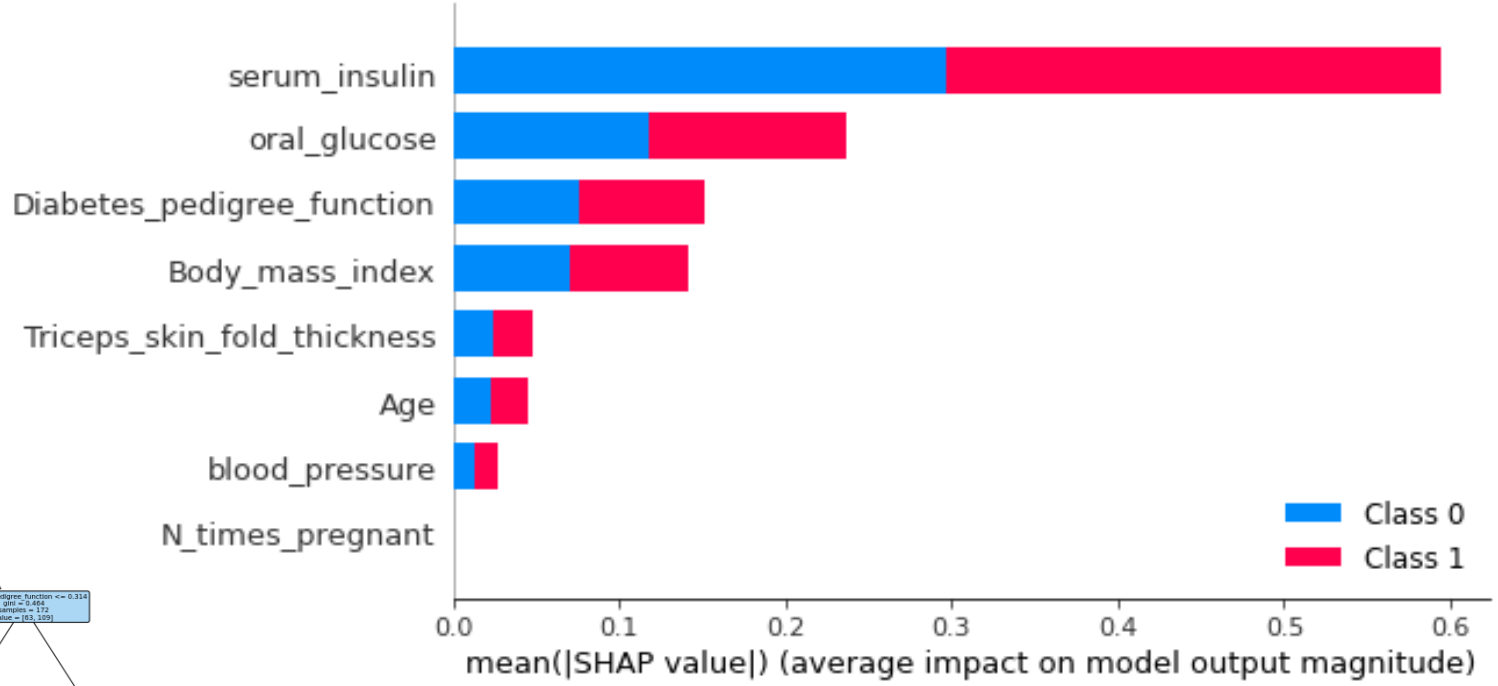
- Pure 데이터 셋에 대한 Tree는 총 3층이며 사용한 설명 변수는 인슐린, 구내혈당, 유전 인자, BMI 이다.
- SHAP에 따르면 루트 노드에 사용된 체내 인슐린 양이 당뇨에 큰 기여를 한다고 했으며 당뇨를 구분하는데 골고루 사용되었다.
- 그 다음으로 다음 층에 사용된 구내혈당이며 앞선 인슐린에 비해 절반 정도의 영향력을 가지고 있다고 판단된다.
- BMI를 제외한 설명 변수들은 중간 발표 때 조사한 체내 혈당과 관련된 요소들이다.

Model Visualization

Decision Tree



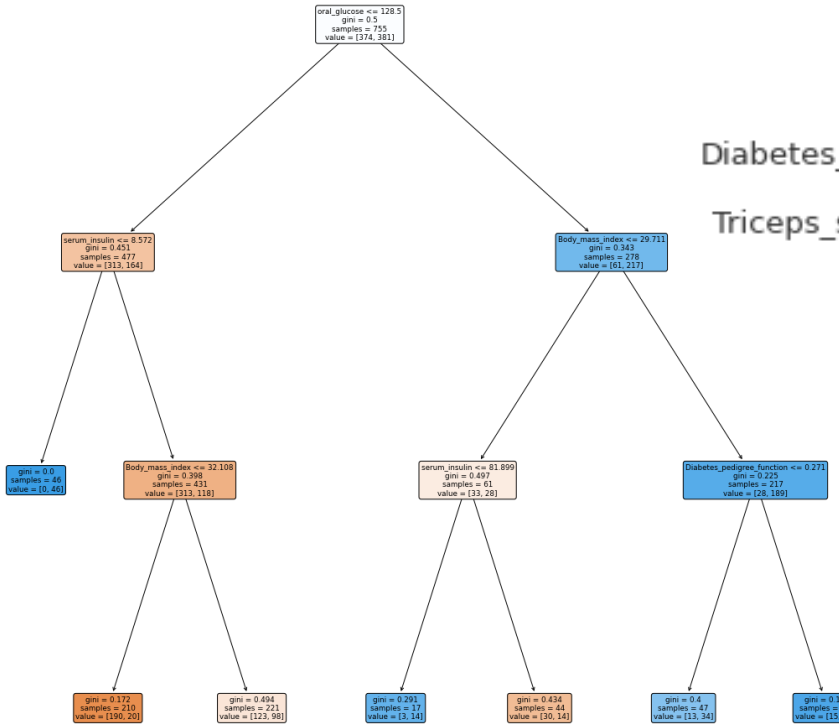
Smote tree



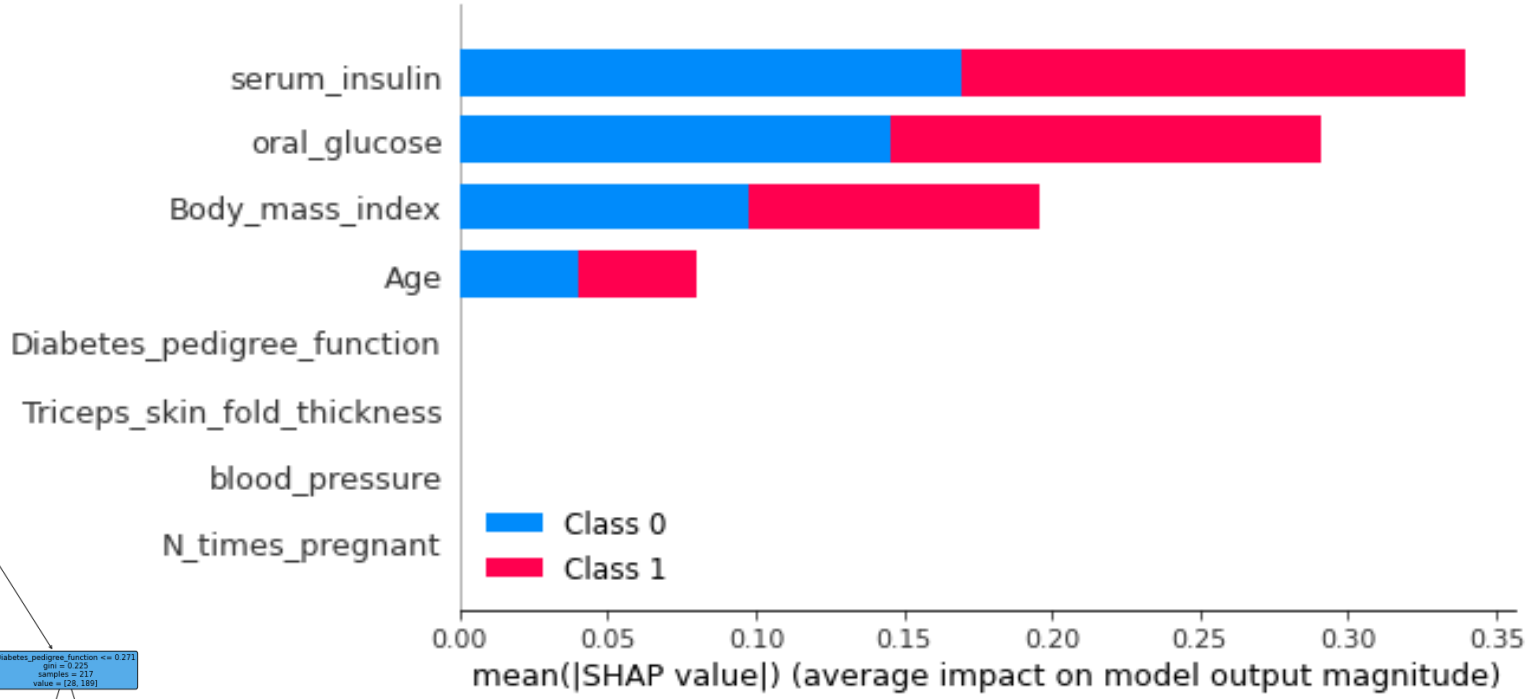
- SMOTE 데이터 셋에 대한 Tree는 앞선 나무에 비해 깊어 대부분의 변수가 사용되었다.
- 하지만 앞선 데이터셋 모델과 변수 중요도는 거의 비슷하다
  (인슐린 > 구내혈당 > 유전인자 > BMI)
- SMOTE 데이터셋 또한 인슐린 수치가 가장 중요하게 작용했으며 앞선 모델보다 더 강한 상관 관계를 보인다.
- 그 다음으로 구내 혈당 수치이며, 유전 인자와 BMI가 거의 비슷한 중요도를 보인다.

Model Visualization

Decision Tree



ADASYN tree



- ADASYN 데이터 셋에 대한 Tree는 pure 나무와 깊이가 같지만 조금 더 복잡하게 나무가 구성되었다.
- 앞선 모델과 다르게 루트 노드가 구내 혈당으로 시작된다.
- 그 결과 변수 중요도는 인슐린 > 구내 혈당 순이지만 둘이 비슷한 중요도로 주요하게 작용했다
- 그 다음으로는 다른 모델들과 다르게 BMI가 세번째로 주요하다고 나왔다
- 마지막으로는 나이 변수가 사용되었으며 다른 두 모델과 다르게 유전인자가 사용되지 않았다.

Model Performance Result

	평균 정확도	평균 정밀도	평균 재현율	평균 F1	평균 AUC
logistic Pure	0.848427	0.805423	0.716089	0.755749	0.814636
logistic SMOTE	0.831500	0.713629	0.819777	0.761742	0.828515
logistic ADASYN	0.789860	0.637830	0.846519	0.726296	0.804319
SVM Pure	0.876998	0.933208	0.676308	0.781253	0.825758
SVM SMOTE	0.873431	0.827069	0.781080	0.801929	0.849852
SVM ADASYN	0.821071	0.675385	0.888034	0.765962	0.838176
Decision Tree Pure	0.880283	0.918847	0.703922	0.793843	0.835250
Decision Tree SMOTE	0.851730	0.766774	0.795964	0.778532	0.837432
Decision Tree ADASYN	0.792002	0.660799	0.817173	0.724068	0.798427
Random Forest Pure	0.882715	0.915080	0.710438	0.797891	0.838722
Random Forest SMOTE	0.878600	0.891900	0.717400	0.795200	0.837400
Random Forest ADASYN	0.809710	0.678762	0.814340	0.738200	0.810890

- RF 또한 DT와 같게 Pure 데이터 셋에서 가장 좋은 성능을 보였다.
- 하지만 복잡한 모델임에도 불구하고 근간이 되는 DT와 성능 차이가 크게 보이지 않는다.
- 실험 시간 또한 Time: 13.4173sec 걸렸기에 간단하면서 좋은 DT가 훨씬 좋다고 판단된다.

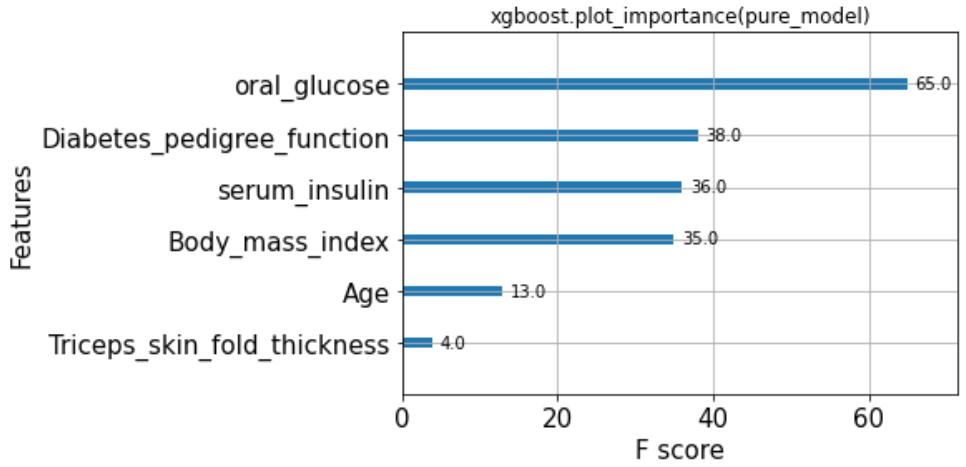
Model Performance Result

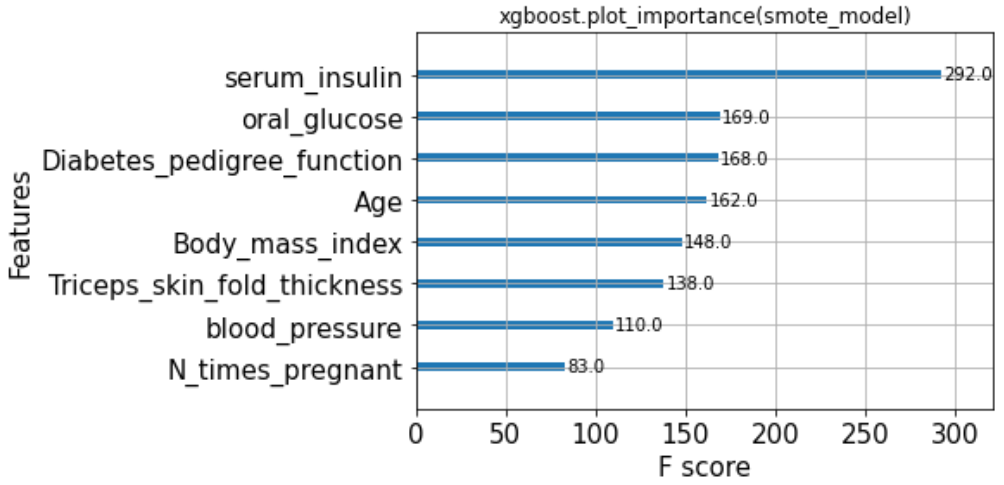
	평균 정확도	평균 정밀도	평균 재현율	평균 F1	평균 AUC
logistic Pure	0.848427	0.805423	0.716089	0.755749	0.814636
logistic SMOTE	0.831500	0.713629	0.819777	0.761742	0.828515
logistic ADASYN	0.789860	0.637830	0.846519	0.726296	0.804319
SVM Pure	0.876998	0.933208	0.676308	0.781253	0.825758
SVM SMOTE	0.873431	0.827069	0.781080	0.801929	0.849852
SVM ADASYN	0.821071	0.675385	0.888034	0.765962	0.838176
Decision Tree Pure	0.880283	0.918847	0.703922	0.793843	0.835250
Decision Tree SMOTE	0.851730	0.766774	0.795964	0.778532	0.837432
Decision Tree ADASYN	0.792002	0.660799	0.817173	0.724068	0.798427
Random Forest Pure	0.882715	0.915080	0.710438	0.797891	0.838722
Random Forest SMOTE	0.878600	0.891900	0.717400	0.795200	0.837400
Random Forest ADASYN	0.809710	0.678762	0.814340	0.738200	0.810890
XGBoost Pure	0.877213	0.907923	0.699134	0.787762	0.831741
XGBoost SMOTE	0.877882	0.827392	0.797189	0.810101	0.857190
XGBoost ADASYN	0.871502	0.793968	0.827377	0.808697	0.860235

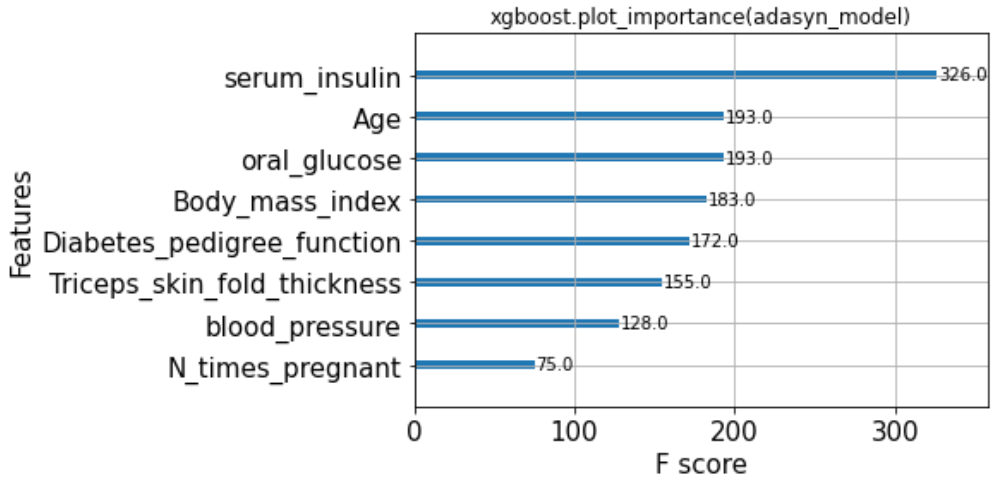
- Xgboost는 logistic과 SVM과 마찬가지로 SMOTE 기법에서 가장 좋은 성능을 보였다
- 다른 모델과 동일하게 ADASYN 데이터 셋에서 가장 안좋은 성능을 보이지만 성능 차이는 가장 적게 차이가 난다.
- 실험 시간은 Time: 14.4836sec이 걸렸다.
- 다른 모델들과 비교하여 성능면에서는 가장 훌륭한 성능을 보이고 있다.

Model Visualization

XGBoost



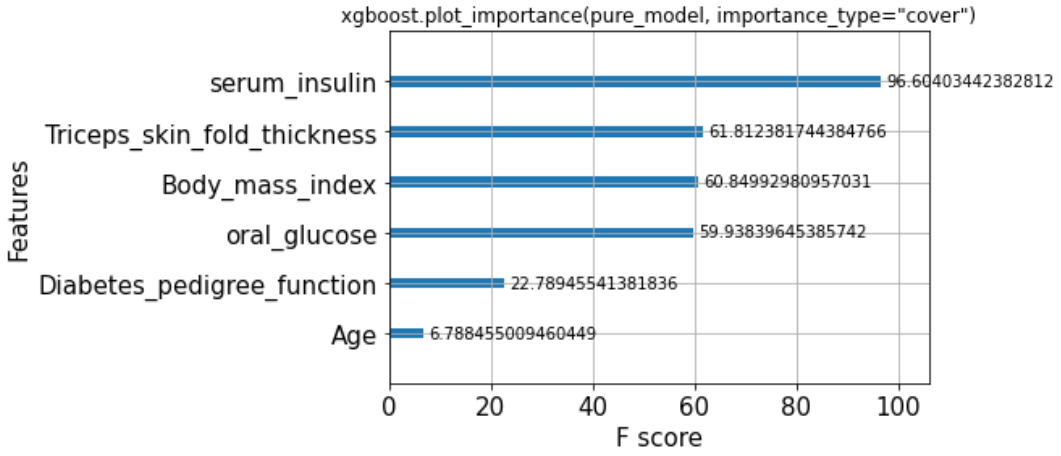


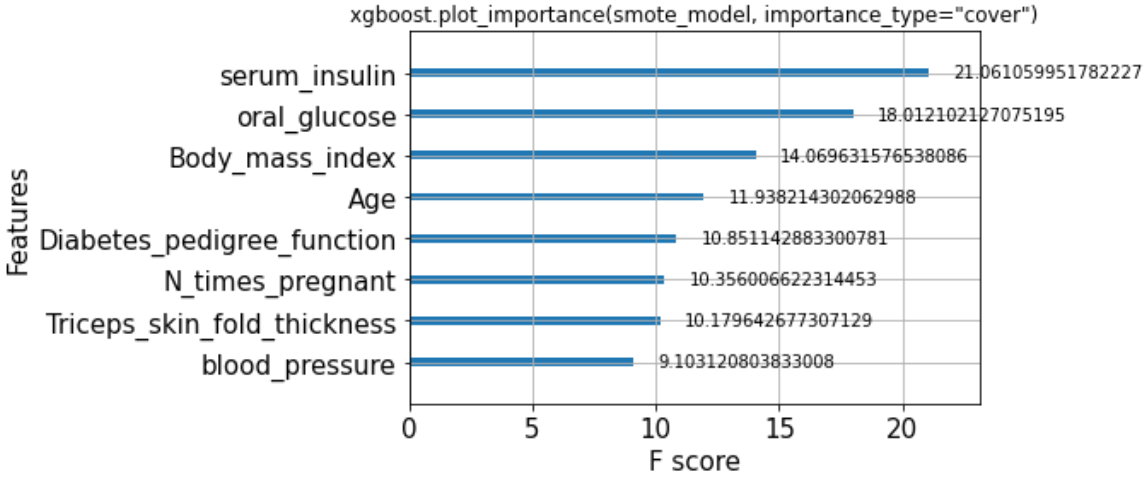


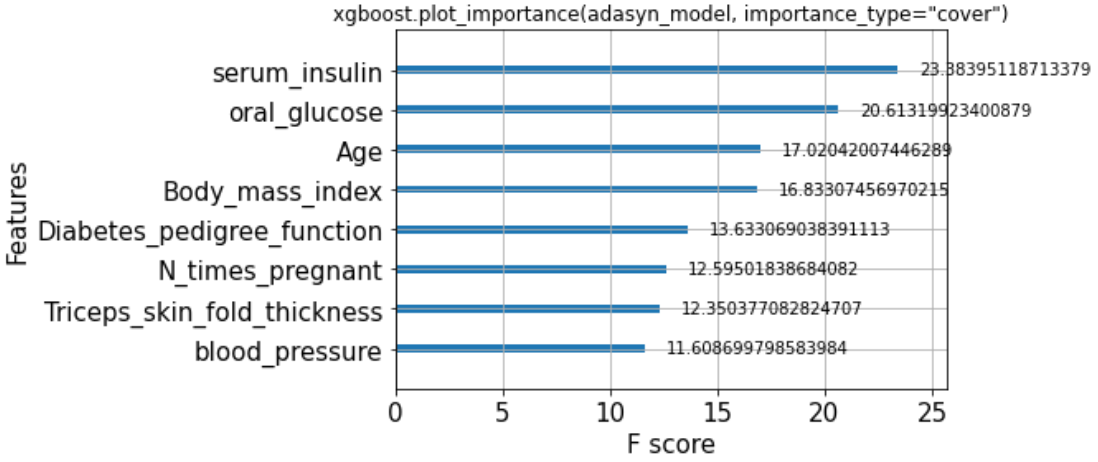
- 해당 feature가 노드 분기에 사용된 횟수를 나타낸 그래프
- Bucket이 많았던 smote & adasyn 모델이 사용 횟수 절대값은 크다
- Pure 모델은 혈당이 많이 쓰인 반면 다른 모델들은 인슐린 변수가 가장 많이 사용되었다.
- 그 뒤를 있는 변수들은 각 모델에 대해 다르나 혈압과 임신 횟수는 상대적으로 적게 사용된 모습을 알 수 있다.

Model Visualization

XGBoost



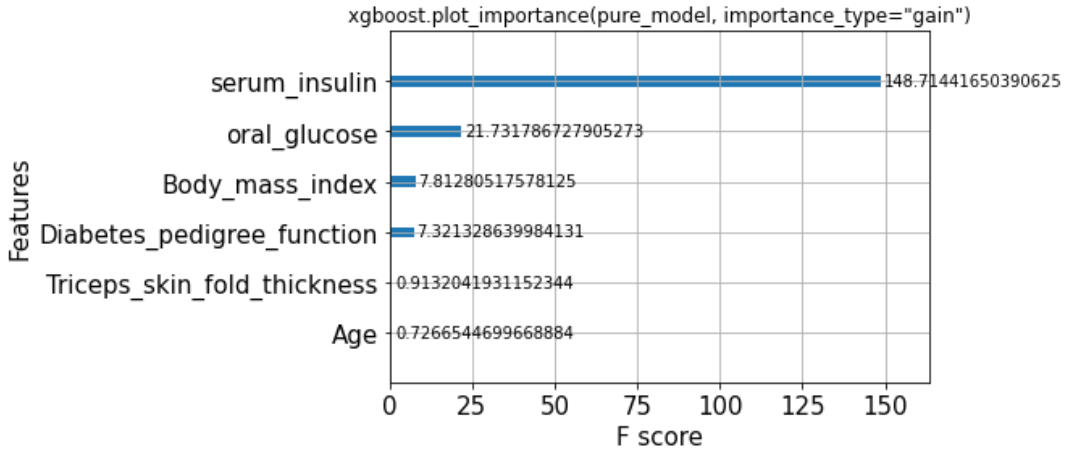


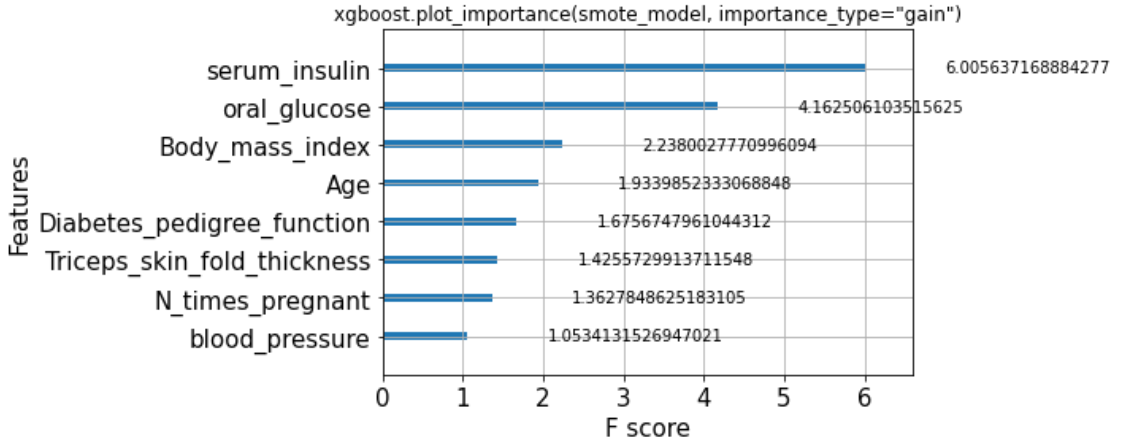


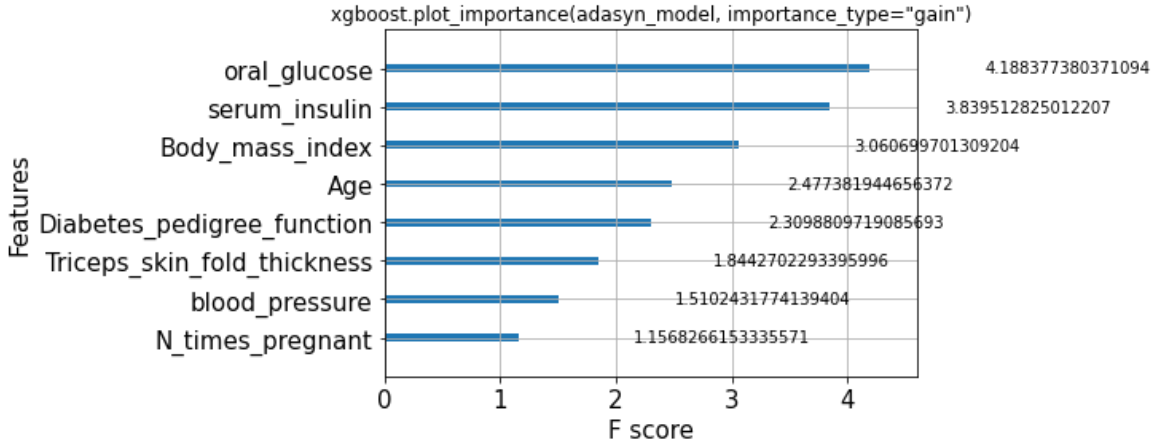
- 해당 feature와 관련된 샘플의 상대적인 개수
- 모든 모델의 모든 버켓에서 인슐린은 주요하게 작용했다
- 하지만 pure 모델에서 주요하게 사용된 피부 두께는 다른 모델에선 주요하게 작용하지 않았다
- 또한, Age 설명 변수 또한 pure 모델에서는 주요하게 사용되지 않았지만 다른 모델에서는 주요하게 사용되었다.
- 즉, smote 와 adasyn 모델의 설명 변수들은 비슷하게 작용했다는 것을 알 수 있다.

Model Visualization

XGBoost



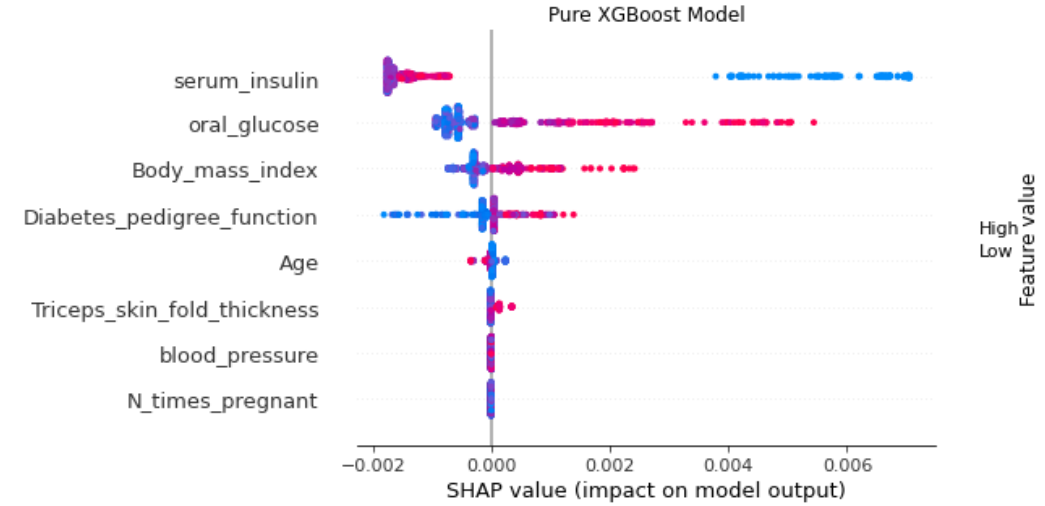


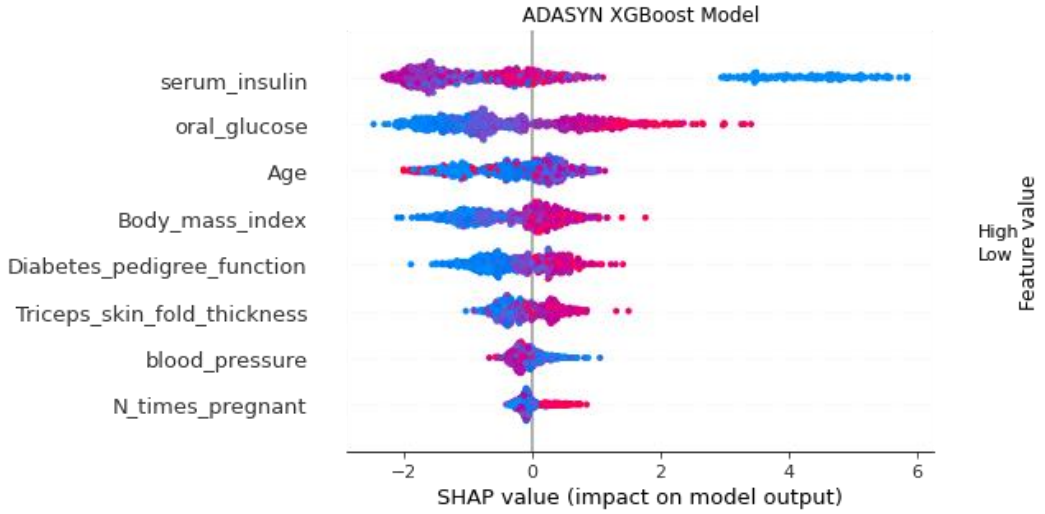


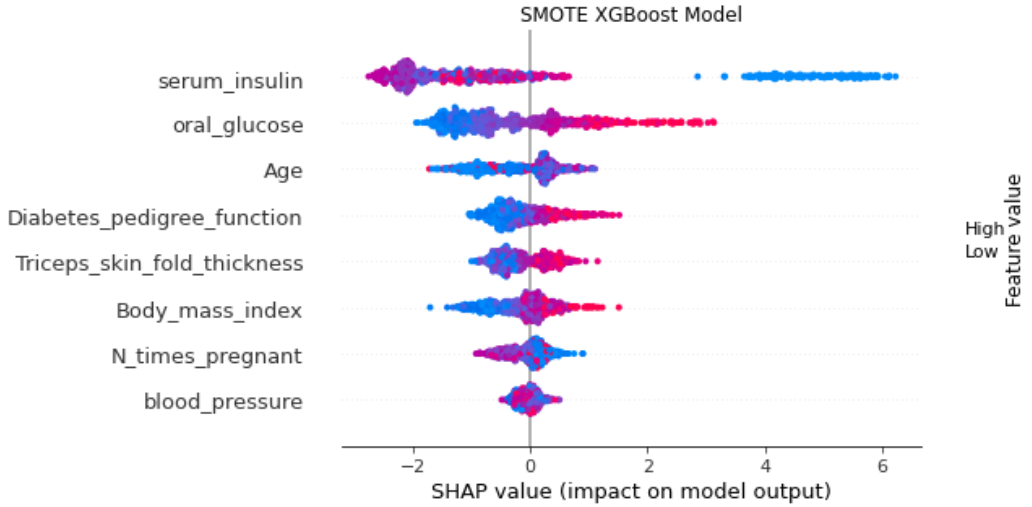
- 특정 feature로 분기되었을 때 얻는 성능 상의 이득 측정
- Pure 모델과 smote 모델에서 인슐린은 가장 주요하게 성능을 좌지우지 했으며 특히 pure 모델에서는 매우 상당하게 작용했다.
- SMOTE와 ADASYN 모델은 가장 주요하게 작용한 설명변수가 다르긴 하지만 앞선 cover 그래프처럼 상당히 비슷한 경향을 보이고 있다.

Model Visualization

XGBoost







- 인슐린 양이 적을수록 당뇨병일 확률이 높다 -> 인슐린이 모든 모델에서 가장 주요하게 기여
- 체내 혈당량이 높을수록 당뇨병일 확률이 높다 -> 인슐린이 제대로 작동을 하지 못하는 당뇨병은 구내 혈당이 높다!
- 나이가 어릴수록 당뇨병이 아닐 경향이 있지만 곳곳에 빨간 점이 있는 것을 봐서 이러한 요소는 1형 당뇨병으로 추측된다
- BMI, 피부 두께는 작을수록 당뇨병이 아니다 => 과체중-비만이 당뇨병에 크게 기여한다
- 혈압과 임신 횟수는 모델마다 다르므로 크게 당뇨병과 상관이 없다고 판단

Final Result

- 단순히 모델의 성능만 봤을 경우 XGBoost가 가장 좋은 성능을 보였다
- 하지만, 모델의 복잡도, 정보 획득 용이성을 모두 고려했을 때 성능이 조금 떨어지지만 간단한 Decision Tree가 가장 적합한 모델이라고 판단된다.
- SMOTE 알고리즘을 이용해 증식 시킨 데이터셋은 좋은 성능을 보였지만 ADASYN 알고리즘을 통해 증식 시킨 데이터셋은 모든 모델에서 가장 떨어지는 성능을 보였다
- 데이터셋이 단순하고 불균형도가 심하지 않았기에 오히려 복잡한 알고리즘은 좋지 않는 샘플을 생성했다는 것을 알 수 있다.
- 로지스틱 모델들을 제외한 거의 모든 모델들에서 인슐린과 구내 혈당량이 가장 주요하게 작용했다.
- 즉, 체내 혈당 수치와 직접적인 연관을 가진 요소(인슐린, 구내혈당)들이 간접적인 영향(나이, 임신 횟수 등)을 가진 요소들에 비해 당뇨병을 예측하는데 주요하게 작용했다.


Thank
you