기계학습론 IC-PBL 최종 보고서

반도체 데이터 성능 예측

조 이름: 6조

이름: 박현일 이정인 최영훈 최현지

0. 요약

반도체 공정에서 측정된 데이터 분석을 통해 특정 표본이 불량인지 아닌지 식별하는 모델을 만드는 것이 목표다. 불균형 클래스 분포를 가진 데이터의 특성을 보완하기 위해 Oversampling 및 Undersampling을 진행하였고 여러 가지 학습 방법을 통해 모델을 구축한 뒤, 모델 별 ROC 점수를 비교해보았다.

1. 문제 정의

반도체 공정 내에서 측정된 여러 값을 분석하여 어떤 표본이 불량인지 아닌지 식별하는 모델을 만들어 불량 반도체를 식별한다. 반도체 제조사가 불량 제품을 직접 확인하지 않고도 어떤 제품이 불량인지 알 수 있게 하여 쉽게 불량 제품과 정상 제품을 분류하는 것을 목적으로 한다.

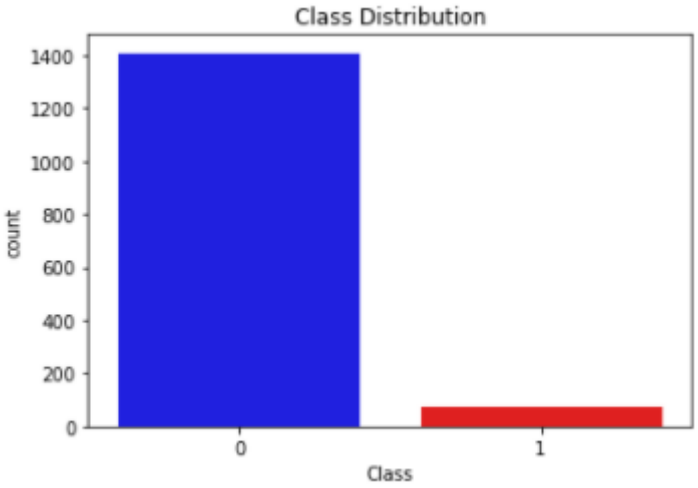
2. 데이터 분석

2.1. 데이터 설명

데이터 세트에는 반도체 공정에서 수집된 측정 데이터가 포함되어 있다. 해당 데이터에는 1,763건의 표본이 포함되어 있고 이 중 143건의 불량 반도체 데이터가 포함되어 있다. 데이터는 세 개의 수치형 feature와 1,554개의 범주형 feature로 이루어져 있다. 또한, 클래스는 불량 반도체의 클래스 값을 1, 그렇지 않을 땐 0의 값을 갖는다.

앞서 말했듯이 총 1,763건의 데이터 중 143건의 표본만이 불량 반도체의 데이터이다. 즉, 데이터 세트가 매우 심각한 불균형 클래스 분포를 보임을 알 수 있다.

2.1.1. 클래스 분포



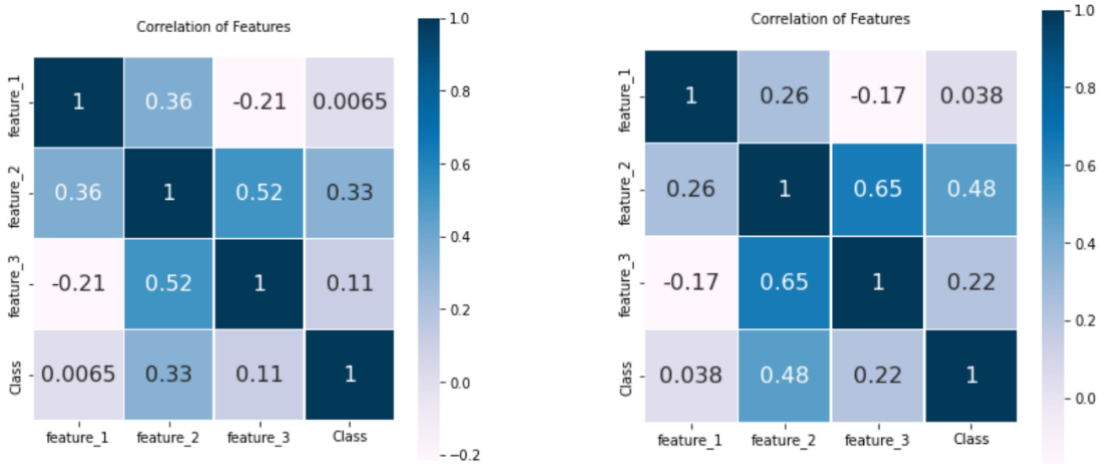
위 그림에서도 확인할 수 있는 것처럼 매우 불균형한 데이터로 정상 반도체의 비율은 91.89%, 불량 반도체의 비율은 8.11%이다. 만약 극심한 불균형 클래스 분포를 가진 해당 데이터 세트를 모델 학습에 그대로 사용할 경우, 낮은 성능을 보여줄 것이다. 또한, 모델이 대부분 표본을 정상 제품으로 추정하기 때문에 학습 데이터에 대한 과적합이 발생할 확률이 높을 것이다. 반도체 공정에서 측정된 데이터를 학습하는 목적이 불량 반도체를 식별하는 것이기 때문에 정상 제품을 예측하는 것도 중요하지만 불량 제품이 생기는 것의 패턴을 잘 파악 및 예측하는 모델을 만드는 것도 중요하다.

2.2. 분석 및 시각화

2.2.1. 시각화

2.2.1.1. Feature 간 상관관계

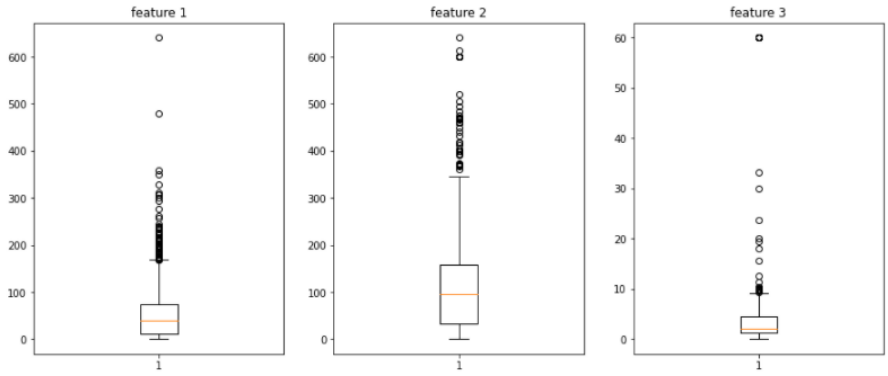
각 feature 간 상관 계수의 분석은 데이터를 이해하는 데 있어 중요하다. 특정한 표본이 정상인지 불량인지 판단하는데 가장 큰 영향을 미치는 feature가 어떤 것인지 확인하기 위해서는 필수적이다. 이를 위해 오버 샘플링을 진행한 서브 데이터 세트로 상관 계수를 분석하였다. 서브 데이터 세트로 상관관계를 시각화한 이유는 아래의 왼쪽과 오른쪽 그림을 비교해보면 알 수 있듯이 기존 데이터는 불균형한 클래스 분포를 하고 있으므로 올바른 상관관계를 파악하기 어렵기 때문이다.



상관관계 분석 결과를 참고한 후 해당 feature 들을 클래스별로 분포를 시각화해보면 위 그림과 같이 명확하게 알 수 있다.

2.2.1.2. 수치형 데이터의 Box plot

수치형 데이터의 분포를 box plot을 통해 확인해보면 아래 그림과 같다. 표본 대부분은 비슷한 값을 가지지만 일부 표본은 이를 벗어나 있음을 알 수 있고 이에 대한 전처리가 필요해 보인다.



2.2.2. 전처리

2.2.2.1. 학습 데이터 및 검증 데이터 분리

Train set, Test set 분리

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=777, stratify=Y)
X_train.reset_index(drop=True, inplace=True)
X_test.reset_index(drop=True, inplace=True)
Y_train.reset_index(drop=True, inplace=True)
Y_test.reset_index(drop=True, inplace=True)
```

```
# 분리 확인
print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)
```

```
(1410, 1558) (353, 1558) (1410,) (353,)
```

전체 데이터를 학습 데이터 및 검증 데이터로 분리하였다. 전체 데이터의 20%만큼 검증 데이터로 할당하였고 random state를 고정하여 추후 다시 모델을 학습하더라도 같은 데이터를 기준으로 하여 정확하게 성능을 평가할 수 있도록 하였다.

2.2.2.2. 중복 제거

중복 레코드 제거

```
temp = pd.concat([X_train, Y_train], axis=1)
```

```
temp.drop_duplicates(keep='first', inplace=True)
```

```
Y_train = temp['Class']
X_train = temp.drop(['Class'], axis=1)
```

해당 코드를 통해 모든 feature에 대해 같은 값을 중복된 데이터를 제거하여 과적합을 방지하였다.

2.2.2.3. 단일 값을 갖는 feature 제거

단일 클래스를 가진 피쳐 제거

```
for i in X_train.columns:
    if len(X_train[i].unique()) == 1:
        X_train.drop(i, axis=1, inplace=True)
        X_test.drop(i, axis=1, inplace=True)
```

```
# 감소한 피쳐 수 확인
print(X_train.shape, X_test.shape)
```

```
(1378, 1515) (353, 1515)
```

모든 샘플에 대해 같은 값을 갖는 feature는 결과에 영향을 끼치지 않는다고 볼 수 있으므로 이를 제거하여 모델을 학습하기 위한 데이터의 차원을 감소시켰다. 이를 통해 고차원의 데이터 세트에서의 학습 데이터가 많음에도 낮은 성능이 나오는 소위 차원의 저주 현상을 어느 정도 완화할 수 있었다.

2.2.2.4. 이상치 제거

Outlier 처리 #1 iqr 기준 삭제

```
toScale = ["feature_1", "feature_2" , "feature_3"]
```

```
trimmer = OutlierTrimmer(capping_method='iqr', tail='both', fold=3, variables=toScale)
X_train = trimmer.fit_transform(X_train)
Y_train = Y_train[X_train.index]
```

상위 25%와 하위 25% 사이의 차이의 1.0, 1.25, 1.5, 3배 밖의 이상치를 제거함으로써 모델에서 학습할 데이터의 전처리를 시행하였다.

2.2.2.5. Feature 간 상관 계수가 높은 feature 제거

학습 데이터의 차원을 줄이기 위해 아래 그림과 같은 코드를 통해 feature 사이의 상관 계수와 feature와 클래스(Y) 사이의 상관 계수를 파악한 다음 0.7 이상의 높은 상관관계를 갖는 두 feature 중 클래스와의 상관관계가 낮은 하나의 feature를 제거하는 과정을 거쳤다.

Feature간 상관계수 계산 후 제거

```
corr = []

for i in X_train.columns:
    corr.append(abs(np.corrcoef(X_train[i], Y_train)[0][1]))
corrX_Y = pd.DataFrame(columns=['features', 'corr'])
corrX_Y['features'] = X_train.columns
corrX_Y['corr'] = corr

corr_mat = abs(X_train.corr())

overCorr = pd.DataFrame(columns=['INDEX', 'COLUMN', 'CORR'])
INDEX, COLUMN, CORR = [], [], []
for i in range(len(corr_mat.index)):
    for j in range(i+1, len(corr_mat.columns)):
        if corr_mat.iloc[i, j] > 0.7:
            #print(corr_mat.index[i], corr_mat.columns[j], corr_mat.iloc[i, j])
            INDEX.append(corr_mat.index[i])
            COLUMN.append(corr_mat.columns[j])
            CORR.append(corr_mat.iloc[i, j])
overCorr['INDEX'] = INDEX
overCorr['COLUMN'] = COLUMN
overCorr['CORR'] = CORR

for i in overCorr.index:
    try:
        if corrX_Y[corrX_Y['features'] == overCorr.loc[i]['INDEX']]['corr'].values[0] <= \
        corrX_Y[corrX_Y['features'] == overCorr.loc[i]['COLUMN']]['corr'].values[0]:
            X_train.drop([overCorr.loc[i]['INDEX']], axis=1, inplace=True)
            X_test.drop([overCorr.loc[i]['INDEX']], axis=1, inplace=True)
        else:
            X_train.drop([overCorr.loc[i]['COLUMN']], axis=1, inplace=True)
            X_test.drop([overCorr.loc[i]['COLUMN']], axis=1, inplace=True)
    except:
        continue
```

2.2.2.6. Scaler 적용

Feature Scaling #1 StandardScaler

```
scaler = StandardScaler()
for feature in toScale:
    X_train.loc[:, feature] = scaler.fit_transform(X_train[feature].to_numpy().reshape(-1, 1))
    X_test.loc[:, feature] = scaler.transform(X_test[feature].to_numpy().reshape(-1, 1))
```

Feature Scaling #2 MinMaxScaler

```
mmScaler = MinMaxScaler()
for feature in toScale:
    X_train.loc[:, feature] = mmScaler.fit_transform(X_train[feature].to_numpy().reshape(-1, 1))
    X_test.loc[:, feature] = mmScaler.transform(X_test[feature].to_numpy().reshape(-1, 1))
```

Feature Scaling #3 RobustScaler

```
robustScaler = RobustScaler()
for feature in toScale:
    X_train.loc[:, feature] = robustScaler.fit_transform(X_train[feature].to_numpy().reshape(-1, 1))
    X_test.loc[:, feature] = robustScaler.transform(X_test[feature].to_numpy().reshape(-1, 1))
```

Feature Scaling #4 MaxAbsScaler

```
maxabsScaler = MaxAbsScaler()
for feature in toScale:
    X_train.loc[:, feature] = maxabsScaler.fit_transform(X_train[feature].to_numpy().reshape(-1, 1))
    X_test.loc[:, feature] = maxabsScaler.transform(X_test[feature].to_numpy().reshape(-1, 1))
```

Feature의 scale을 통일함으로써 변수의 값이 갖는 범위 및 단위가 달라서 발생할 수 있는 bias 같은 문제를 예방할 수 있다. 위에 이미지로 제시된 4가지 StandardScaler, MinMaxScaler, RobustScaler, MaxAbsScaler 등을 각각 적용하여 어떤 Scaler가 더 높은 성능을 가지는지 확인하였다.

2.2.2.7. 불균형한 클래스 분포 해결

2.2.2.7.1. Random Undersampling

기존 데이터에는 1,763건의 반도체 공정 데이터 중 143개의 불량 제품 표본이 있다. 따라서 정상 제품 표본 중 143개의 데이터를 추출하여 데이터를 합해 새로운 데이터를 만드는 것을 Undersampling이라고 한다. 이는 더 균형 있는 데이터 세트를 얻기 위해 다수 클래스의 일부 표본을 제거하고 모델이 과대 적합 되지 않도록 하는 기법이다. 하지만 해당 데이터를 Undersampling 하게 되면 286개의 데이터를 갖게 되어 모델을 학습하는데 너무 적은 데이터만 남기 때문에 제외하였다.

* 절차

1. 클래스 분포의 불균형을 확인
2. 다수 클래스에서 소수 클래스 수만큼 데이터 세트에서 추출하여 1:1의 분포를 갖도록 만들기

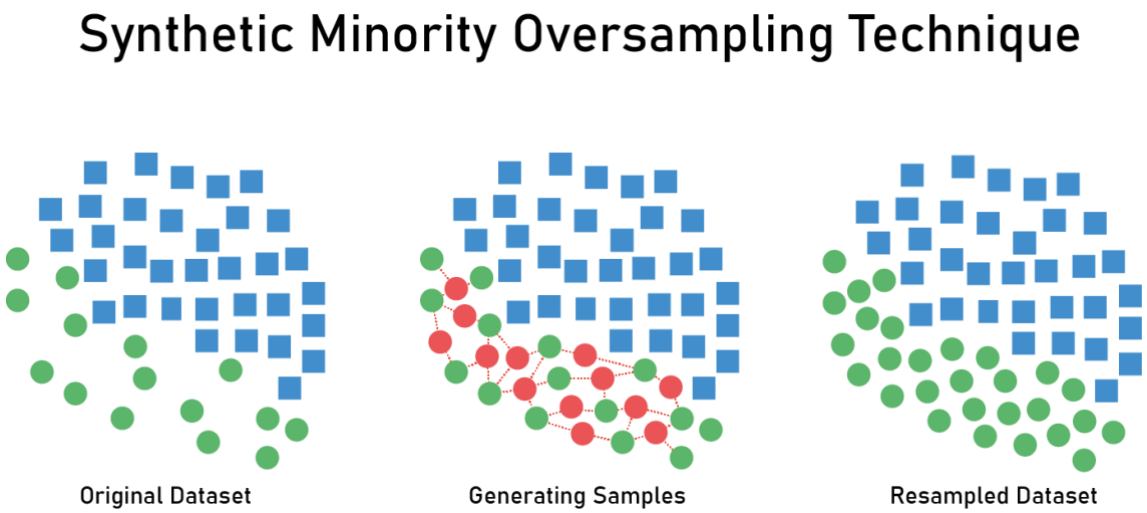
* 주의점
Undersampling은 기존의 데이터를 삭제하는 방법이기 때문에 모델의 학습 과정에서 기존 데이터를 온전히 반영하지 못한다는 문제가 있다. 왜냐하면, 1,763개 중 1,477개의 데이터를 제거하고 143개의 데이터만 가져오기 때문이다.

2.2.2.7.2. Oversampling

Undersampling과 반대로 클래스 불균형 분포를 해결하기 위해 Oversampling 방법을 이용할 수 있다. 이는 소수 클래스의 데이터를 다수 클래스의 수만큼 증가시키는 기법이다. 이는 기존 데이터의 손실이 없어 장점을 갖고 있지만, 소수 클래스에 대해 과대 적합 될 가능성이 크다.

- SMOTE

SMOTE는 소수 클래스를 합성하여 이에 해당하는 표본을 복제하는 것이 아니라 새로운 레코드를 생성하는 것을 의미한다. 아래 그림과 같이 먼저 소수 클래스의 데이터의 표본을 취하고 해당 표본의 K개의 가까운 표본을 찾는다. 또한, 해당 표본과 K개의 이웃 표본과의 차이를 구하고 이 차이에 0과 1 사이의 임의의 값을 곱하여 기존 표본에 더하는 방식으로 합성하고 이를 훈련 데이터에 추가한다. 결과적으로 SMOTE는 기존의 표본을 주변의 이웃 표본과 합쳐 새로운 표본을 추가하여 이를 추가하는 방식으로 동작한다.
또한 SMOTE 기법을 기반으로 좀 더 발전된 SMOTENC, Borderline SMOTE, ADASYN 등의 방법도 적용하여 테스트하였다.



2.2.2.7.3. Class weight 적용

Undersampling 또는 Oversampling을 적용하지 않고 모델 학습 시 손실 함수에 클래스 비율에 따라 소수 클래스에 더 많은 가중치를 주어 불균형한 클래스 분포를 해소하는 방법이다. 이를 활용하면 소수 클래스가 전체 Loss에 동일하게 기여하도록 할 수 있다. 또한, Undersampling과 Oversampling과 달리 기존 데이터 레코드에 변형을 일으키지 않기 때문에 상대적으로 과소 적합과 과대 적합에 자유로운 장점이 있다.

클래스 가중치 적용

```
weight_0 = (1 / Y_train.value_counts()[0]) * (Y_train.value_counts().sum() / 2.0)
weight_1 = (1 / Y_train.value_counts()[1]) * (Y_train.value_counts().sum() / 2.0)
class_weight = {0 : weight_0, 1 : weight_1}
```

3. 모델

3.1. 모델 입력 값

학습 데이터는 반도체 공정에서 측정된 3개의 수치형 데이터, 1,554개의 범주형 데이터를 독립 변수로 갖고 있다. 전처리 과정을 거쳐 학습 데이터를 축소하고 이들을 Input data로 사용했다.

3.2. 모델이 예측하는 값

반도체 불량 예측 모델은 해당 표본이 불량인지 아닌지 판단하는 모델이다. 종속 변수로는 불량인 경우로는 1, 아닌 경우에는 0의 값을 갖는 “Class” feature를 예측 값으로 사용하였다.

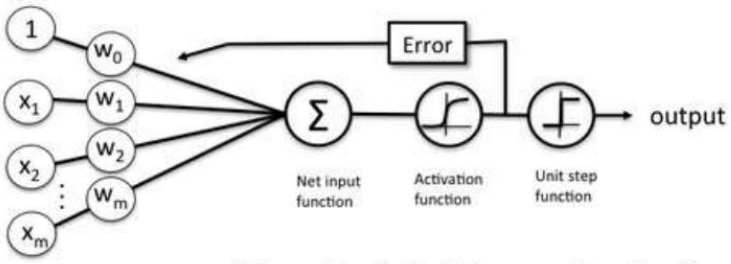
3.3. 사용한 모델 및 하이퍼파라미터

모든 모델에 대해 10부터 100까지 10 단위로 random state를 지정하여 다양한 데이터에 대한 성능을 ROC-AUC 점수로 평가하였다. 또한 모델의 학습에 필요한 하이퍼파라미터의 튜닝 및 다양한 전처리 환경 하에서의 성능을 제시할 것이다.

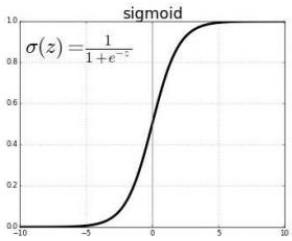
3.3.1. Keras Logistic Regression

Keras Logistic regression

```
AUCScores_log = np.zeros(10)
model_log = [0 for _ in range(10)]
for i in range(10, 101, 10):
    tf.random.set_seed(i)
    model_log[int(i/10 - 1)] = models.Sequential(name=f"Keras_log_Random_state_{i}")
    model_log[int(i/10 - 1)].add(layers.Dense(1, activation='sigmoid', input_shape=(X_train.shape[1],),
                                 kernel_regularizer=regularizers.l2(0.0001)))
    model_log[int(i/10 - 1)].compile(optimizer='adam', loss='binary_crossentropy', \
                                     metrics=['binary_accuracy'])

    kfold = StratifiedKFold(n_splits=5)
    modelHistory = ModelHistory()
    cb = [callbacks.EarlyStopping(monitor='val_loss', patience=10, min_delta=0.01), \
          modelHistory]
    with tqdm(total=5, ascii=True) as pbar:
        for train_index, test_index in kfold.split(X_train, Y_train):
            X_train_log, X_val_log = X_train.loc[train_index], X_train.loc[test_index]
            Y_train_log, Y_val_log = Y_train.loc[train_index], Y_train.loc[test_index]
            model_log[int(i/10 - 1)].fit(X_train_log, Y_train_log, \
                                         validation_data=(X_val_log, Y_val_log), \
                                         epochs=50, use_multiprocessing=True, workers=-1, verbose=0, \
                                         callbacks=cb, class_weight=class_weight)
            pbar.update(1)

    model_log_pred = model_log[int(i/10 - 1)].predict(X_test)
    AUCScores_log[int(i/10 - 1)] = roc_auc_score(Y_test, model_log_pred)
    #print_clf_eval(Y_test, model_log_pred)

print(AUCScores_log.mean().round(3))
```



Schematic of a logistic regression classifier.



Keras 라이브러리에서 지원하는 Logistic Regression 모델을 사용하여 학습을 진행하였다. Logistic Regression 모델은 데이터가 어떤 클래스에 속하는지에 대한 확률을 0과 1 사이의 값으로 예측하는 모델이다. K-fold 검정을 활용하여 전체 데이터를 5 등분하고 Early stopping의 기준을 정하여 학습의 속도와 성능을

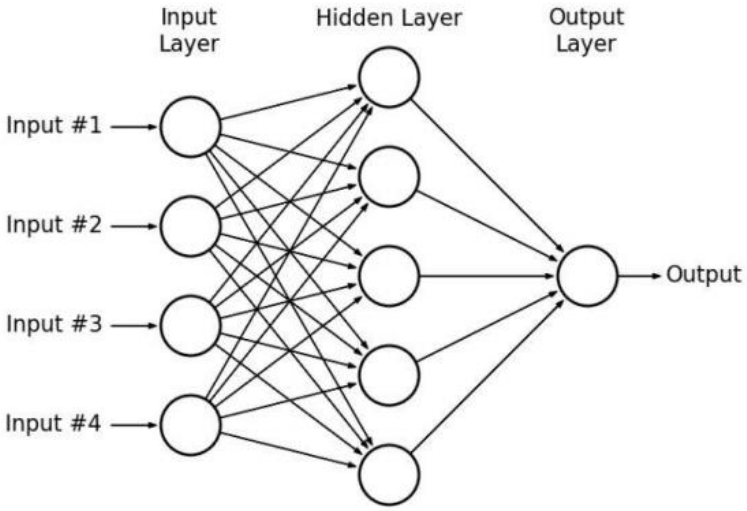
향상하였다. 또한, 과적합 방지를 위해 모델의 손실 함수에 Regularization을 적용하였다. Regularization을 적용한 손실 함수 식은 아래와 같다. 해당 모델의 성능의 평가한 결과 0.85~0.93의 값을 보여주었다.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^i \log(\hat{y}^i) - (1-y^i)\log(1-\hat{y}^i)] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

3.3.2. Keras MLP

Keras MLP

```
AUCScores_mlp = np.zeros(10)
model_mlp = [0 for _ in range(10)]
for i in range(10, 101, 10):
    tf.random.set_seed(i)
    model_mlp[int(i/10 - 1)] = models.Sequential(name=f"Keras_mlp_Random_state_{i}")
    model_mlp[int(i/10 - 1)].add(layers.Dense(100, activation='relu', input_shape=(X_train.shape[1],), \
                                kernel_regularizer=regularizers.l2(0.0001)))
    #model_mlp[int(i/10 - 1)].add(layers.Dropout(0.5))
    model_mlp[int(i/10 - 1)].add(layers.Dense(1, activation='sigmoid'))
    model_mlp[int(i/10 - 1)].compile(optimizer='adam', loss='binary_crossentropy', \
                                     metrics=['binary_accuracy'])

    kfold = StratifiedKFold(n_splits=5)
    modelHistory = ModelHistory()
    cb = [callbacks.EarlyStopping(monitor='val_loss', patience=10, min_delta=0.01), \
          modelHistory]
    with tqdm(total=5, ascii=True) as pbar:
        for train_index, test_index in kfold.split(X_train, Y_train):
            X_train_mlp, X_val_mlp = X_train.loc[train_index], X_train.loc[test_index]
            Y_train_mlp, Y_val_mlp = Y_train.loc[train_index], Y_train.loc[test_index]
            model_mlp[int(i/10 - 1)].fit(X_train_mlp, Y_train_mlp, \
                                         validation_data=(X_val_mlp, Y_val_mlp), \
                                         epochs=50, use_multiprocessing=True, workers=-1, verbose=0, \
                                         callbacks=cb, class_weight=class_weight)
            pbar.update(1)

    model_mlp_pred = model_mlp[int(i/10 - 1)].predict(X_test)
    AUCScores_mlp[int(i/10 - 1)] = roc_auc_score(Y_test, model_mlp_pred)
    #print_clf_eval(Y_test, model_mlp_pred)

print(AUCScores_mlp.mean().round(3))
```



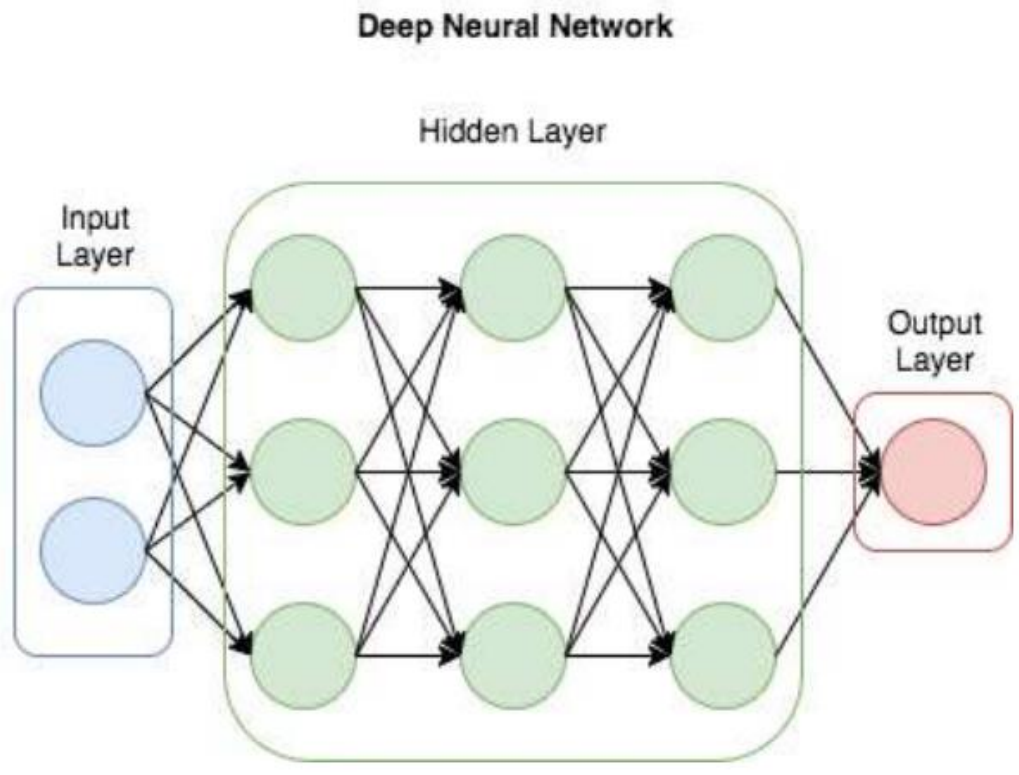
Keras 라이브러리에서 지원하는 MLP 모델을 사용하여 학습을 진행하였다. 해당 모델은 인공 신경망의 일종으로 뉴런이 연결된 형태를 모방한 모델이다. 하나의 뉴런을 모델로 구현한 퍼셉트론과 이를 여러 층으로 연결한 것이 Logistic Regression과 차이점이다. Keras MLP모델도 위 Logistic Regression 모델과 마찬가지로 Regularization과 Early stopping을 적용하고 K-fold 검정을 통해 전체 데이터를 5 등분하고 성능을 측정한 결과 0.77~0.91 사이의 값을 보였다.

3.3.3. Keras DNN

Keras 라이브러리에서 Sequential 함수를 사용해 hidden layer를 포함한 모델이라는 점은 MLP 모델과 비슷하나 한 층의 hidden layer를 가진 MLP와 달리 DNN은 2개 이상의 hidden layer를 갖고 있다는 차이점이 있다. 위 Keras 모델들과 동일한 방법을 적용하여 학습을 진행했고 성능을 측정한 결과 0.84~0.9 사이의 값을 가졌다.

Keras DNN

```
AUCScores_dnn = np.zeros(10)
model_dnn = [0 for _ in range(10)]
for i in range(10, 101, 10):
    tf.random.set_seed(i)
    model_dnn[int(i/10 - 1)] = models.Sequential()
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu', input_shape=(X_train.shape[1],)))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(100, activation='relu'))
    model_dnn[int(i/10 - 1)].add(layers.Dense(1, activation='sigmoid'))
    model_dnn[int(i/10 - 1)].compile(optimizer='adam', loss='binary_crossentropy', \
                                     metrics=['binary_accuracy'])

    kfold = StratifiedKFold(n_splits=5)
    modelHistory = ModelHistory()
    cb = [callbacks.EarlyStopping(monitor='val_loss', patience=10, min_delta=0.01), \
          modelHistory]
    with tqdm(total=5, ascii=True) as pbar:
        for train_index, test_index in kfold.split(X_train, Y_train):
            X_train_dnn, X_val_dnn = X_train.loc[train_index], X_train.loc[test_index]
            Y_train_dnn, Y_val_dnn = Y_train.loc[train_index], Y_train.loc[test_index]
            model_dnn[int(i/10 - 1)].fit(X_train_dnn, Y_train_dnn, \
                                         validation_data=(X_val_dnn, Y_val_dnn), \
                                         epochs=50, use_multiprocessing=True, workers=-1, verbose=0, \
                                         callbacks=cb, class_weight=class_weight)
            pbar.update(1)

    model_dnn_pred = model_dnn[int(i/10 - 1)].predict(X_test)
    AUCScores_dnn[int(i/10 - 1)] = roc_auc_score(Y_test, model_dnn_pred)

print(AUCScores_dnn.mean().round(3))
```



3.3.4. Scikit Learn Logistic Regression

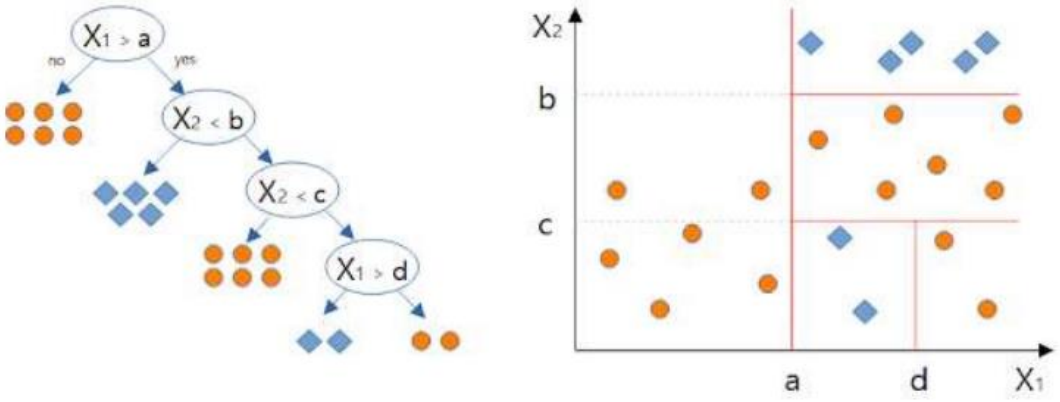
Scikit Learn 내 Logistic regression 모델을 사용하여 모델을 구축하였다. 모델의 원리는 3.3.1. 에 제시된 Keras 라이브러리의 Logistic regression 모델과 비슷하다. 해당 모델은 0.63~0.84 정도의 값을 가졌고 Keras 라이브러리에 있는 모델 대비 낮은 성능을 보였다.

Sklearn Logistic regression

```
AUCScores_sk_log = np.zeros(10)
sk_log = [0 for _ in range(10)]
for i in range(10, 101, 10):
    sk_log[int(i / 10 - 1)] = LogisticRegression(random_state=i, solver='liblinear', \
                                                 class_weight=class_weight)
    sk_log[int(i / 10 - 1)].fit(X_train, Y_train)
    sk_log_pred = sk_log[int(i / 10 - 1)].predict(X_test)
    AUCScores_sk_log[int(i/10 - 1)] = roc_auc_score(Y_test, sk_log_pred)
    #print_clf_eval(Y_test, sk_log_pred)

print(AUCScores_sk_log.mean().round(3))
```

3.3.5. Scikit Learn Decision Tree

Scikit Learn 내 Decision Tree 모델을 사용하여 모델을 구축하였다. 해당 모델은 학습 데이터를 분석하여 이들 사이에 존재하는 패턴을 예측 가능한 규칙들의 조합으로 나타낸 모델로 0.58~0.68 사이의 값을 가졌고 타 모델 대비 좋은 성능을 보이지 못했다.

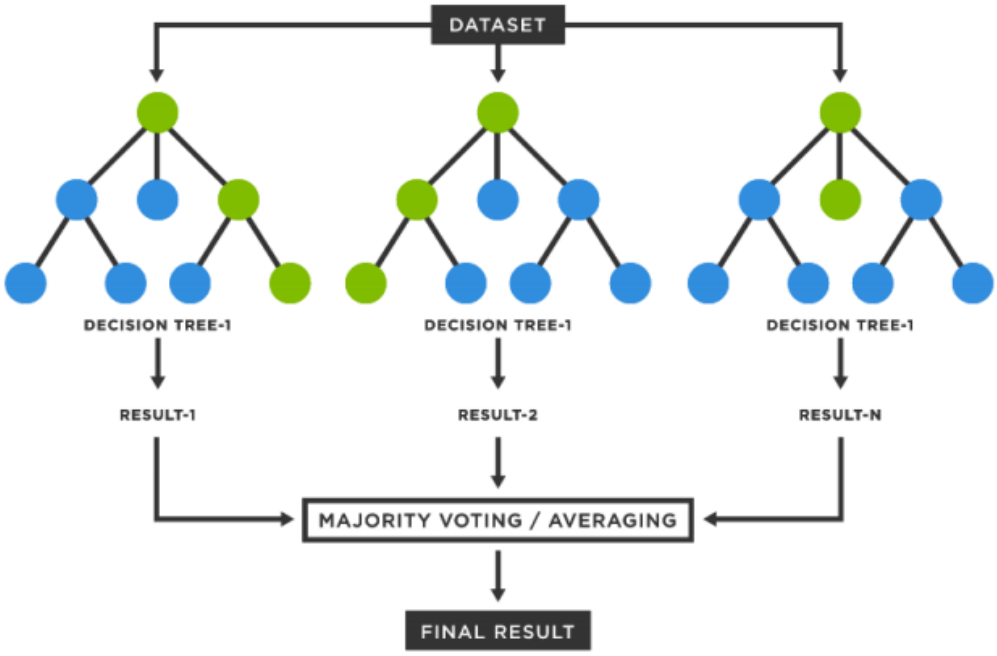


Sklearn Decision Tree

```
AUCScores_sk_tree = np.zeros(10)
sk_tree = [0 for _ in range(10)]
for i in range(10, 101, 10):
    sk_tree[int(i/10 - 1)] = DecisionTreeClassifier(random_state=i, max_depth=30, \
                                                    class_weight=class_weight)
    sk_tree[int(i/10 - 1)].fit(X_train, Y_train)
    sk_tree_pred = sk_tree[int(i/10 - 1)].predict(X_test)
    AUCScores_sk_tree[int(i/10 - 1)] = roc_auc_score(Y_test, sk_tree_pred)
    #print_clf_eval(Y_test, sk_tree_pred)

print(AUCScores_sk_tree.mean().round(3))
```

3.3.6. Scikit Learn Random Forest

Scikit Learn 내 Random Forest 모델을 사용하여 모델을 구축하였다. 해당 모델은 3.3.5. 에 적힌 Decision tree를 Bagging 기법을 통해 앙상블을 진행하여 과적합을 줄이고 정확도를 높이는 모델이다. 해당 모델은 0.63~0.72 사이의 값을 가졌고 Decision tree가 좋은 성능을 갖지 못했기 때문에 이를 앙상블한 Random forest 모델은 Decision tree 대비 약간의 성능 향상이 있었지만 다른 모델 대비 현저하게 낮은 성능을 보였다.



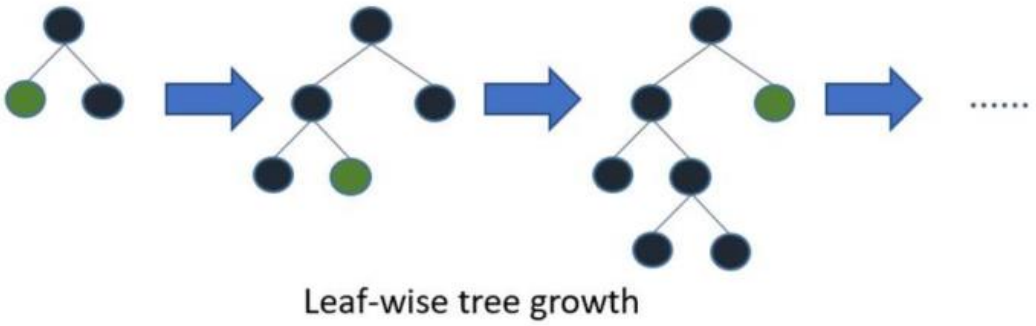
Sklearn RandomForest

```
AUCScores_sk_rf = np.zeros(10)
sk_rf = [0 for _ in range(10)]
for i in range(10, 101, 10):
    sk_rf[int(i/10 - 1)] = RandomForestClassifier(random_state=i, n_jobs=-1, n_estimators=10, \
                                                  class_weight=class_weight)
    sk_rf[int(i/10 - 1)].fit(X_train, Y_train)
    sk_rf_pred = sk_rf[int(i/10 -1 )].predict(X_test)
    AUCScores_sk_rf[int(i/10 - 1)] = roc_auc_score(Y_test, sk_rf_pred)
    #print_clf_eval(Y_test, sk_rf_pred)

print(AUCScores_sk_rf.mean().round(3))
```

3.3.7. LightGBM

LightGBM은 Random forest와 같이 Decision tree를 앙상블 하는 기법의 하나로 각 tree의 오차에 가중치를 두어 오차가 작은 tree의 가중치를 크게 하여 성능을 개선하는 모델이다. 해당 모델은 0.62~0.79 사이의 값을 가졌고 Decision tree가 좋은 성능을 갖지 못했기 때문에 이를 앙상블한 LightGBM 모델은 Decision tree 대비 약간의 성능 향상이 있었고 오차가 적은 tree에 가중치를 두는 모델 특성상 Random forest와 비교해서도 좀 더 높은 성능을 보였지만 다른 모델 대비 현저

하게 낮은 성능을 보였다.

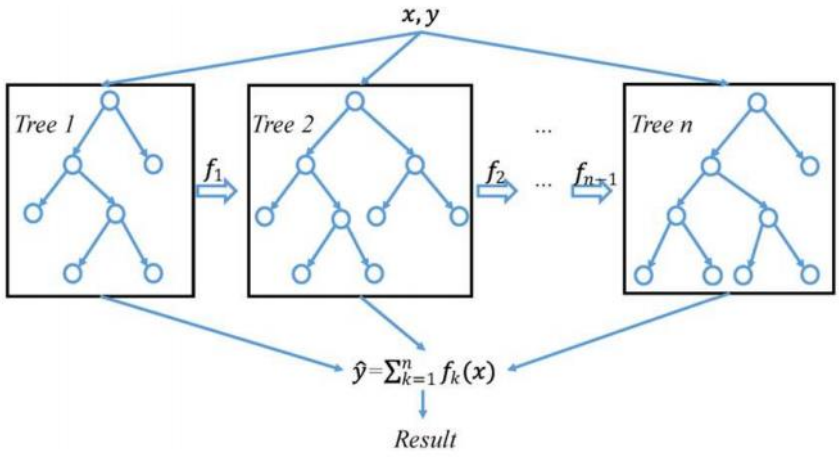


LightGBM

```
AUCScores_lgbm = np.zeros(10)
lgbm = [0 for _ in range(10)]
for i in range(10, 101, 10):
    lgbm[int(i/10 - 1)] = LGBMClassifier(n_estimators=300, num_leaves=100, n_jobs=-1, \
                          boost_from_average=False, random_state=i, class_weight=class_weight)
    lgbm[int(i/10 - 1)].fit(X_train, Y_train)
    lgbm_pred = lgbm[int(i/10 - 1)].predict(X_test)
    AUCScores_lgbm[int(i/10 - 1)] = roc_auc_score(Y_test, lgbm_pred)
    #print_clf_eval(Y_test, lgbm_pred)

print(AUCScores_lgbm.mean().round(3))
```

3.3.8. XGBoost

XGBoost는 Random forest와 같이 Decision tree를 앙상블 하는 기법의 하나로 각 tree를 boosting으로 앙상블하고 이를 병렬 학습이 가능하도록 만든 모델이다. 해당 모델은 0.5 정도의 값을 가졌고 Decision tree가 좋은 성능을 갖지 못했기 때문에 이를 앙상블한 XGBoost 모델은 Decision tree 대비 더 나은 성능을 얻을 것으로 예상되었으나 반대의 성능으로 측정되었다.



XGBoost

```
AUCScores_xgb = np.zeros(10)
xgb = [0 for _ in range(10)]
for i in range(10, 101, 10):
    xgb[int(i/10 - 1)] = XGBClassifier(random_state=i, eta=0.2, max_depth=0, min_child_weight=1, \
                                       n_estimators=500, objective='binary:logistic', \
                                       scale_pos_weight=Y_train.value_counts()[1]/Y_train.value_counts()
    xgb[int(i/10 - 1)].fit(X_train, Y_train)
    xgb_pred = xgb[int(i/10 -1)].predict(X_test)
    AUCScores_xgb[int(i/10 - 1)] = roc_auc_score(Y_test, xgb_pred)
    #print_clf_eval(Y_test, x_pred)

print(AUCScores_xgb.mean().round(3))
```

4. 실험 결과

4.1. 성능 지표

ROC-AUR 함수를 이용하여 각 모델의 성능을 판단했다. ROC 곡선은 모든 임계값에 대해 분류 모델의 성능을 보여주는 그래프로 AUC는 해당 곡선의 아래 영역의 비율을 의미한다. 해당 데이터가 불균형한 데이터 분포를 하고 있을 때 Accuracy의 단점을 보완하여 모델을 평가하기에 적합하기 때문이다.

4.2. 학습 환경

- 학습된 OS: MacOS 및 Windows 10 (Python 3.8.2)
- 이상치 제거: IRQ 기준 제거(1.0, 1.25, 1.5, 1.75, 2.0, 3.0) 및 Winsorizer 적용
- 상관 관계 분석을 통한 feature 제거: 상관 관계 값이 0.7 ~ 0.9 기준 적용
- Feature Scaler: StandardScaler, MinMaxScaler, RobustScaler, MaxAbsScaler
- Oversampling: SMOTE, Random oversampling, Borderline SMOTE, ADASYNC
- Class weight: Oversampling을 진행하지 않고 클래스 별 가중치를 부여하는 기법 사용
- 학습 모델: Logistic Regression, MLP, DNN(이상 Keras), Logistic Regression, Decision Tree, Random Forest(이상 Scikit learn), LightGBM, XGBoost

4.3. 학습 결과

이상치 제거(IRQ 3 기준) 진행 및 Oversampling 대신 Class별로 가중치를 부여하고 MaxAbsScaler를 사용한 Keras Logistic 모델이 다른 모델에 비해 상대적으로 높은 성능을 보여주었고 최종적으로 0.93202427의 결과를 보여주었다. 또한 같은 전처리 과정 하에서 MLP 및 DNN도 0.92 정도의 성능을 보여 상대적으로 높은 성능을 보였다. 하지만 Tree 모델인 Decision Tree 모델과 이를 앙상블한 Random Forest, LightGBM, XGBoost 모델은 항상 0.8~0.9 정도의 낮은 점수를 보여주었다.

5. 결과

불균형한 데이터 세트를 학습하기 위해서는 Oversampling 및 Undersampling을 통해 각 클래스의 분포를 맞추는 것을 정석으로 생각했다. 하지만 해당 기법을 통한 서브 데이터 세트를 학습하는 것보다 클래스 별 가중치를 주어 학습하는 것이 좀 더 나은 성능을 보여주었다.

또한 수치형 데이터를 학습에 사용할 때는 각 feature 별로 Scaler를 통해 단위 및 범위를 맞춰야 높은 성능을 보이는데 해당 데이터 세트에는 Scaler를 적용한

것보다 원본 데이터를 사용한 것이 좀 더 나은 성능을 보여주었다.

최적의 조합을 가진 모델의 성능을 측정해본 결과 0.93202427의 ROC 점수를 구할 수 있었다. Threshold가 0.5 일 때, 3 건의 정상 제품을 잘못 분류하였고 23개의 불량 제품 중 12개의 불량 제품을 잡아낼 수 있었다.