기계학습

Term Project

분류(Classification)
군집화(Clustering)

한준호
2018741035

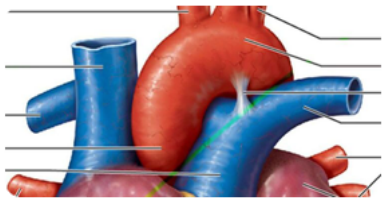
분류(Classification)

1) 데이터 선정

'heart.csv'

Heart Attack Analysis & Prediction Dataset

A dataset for heart attack classification



데이터셋 Link: https://www.kaggle.com/datasets/rashikrahmanpritom/heart-attack-analysis-prediction-dataset

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	output
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

```
df.shape
(303, 14)
```

Columns

- age : Age of the patient
- sex : Sex of the patient
- exng: exercise induced angina (1 = yes; 0 = no)
- caa: number of major vessels (0-3)
- cp : Chest Pain type chest pain type
  Value 1: typical angina / Value 2: atypical angina
  Value 3: non-anginal pain / Value 4: asymptomatic
- trtbps : resting blood pressure (in mm Hg)
- chol : cholestoral in mg/dl fetched via BMI sensor
- fbs : (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
- restecg : resting electrocardiographic results
  Value 0: normal

Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)

Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria

- thalachh : maximum heart rate achieved
- target : 0= less chance of heart attack 1= more chance of heart attack

데이터 선정 이유:

Dataset 은 kaggle 의 'Heart Attack' dataset 으로 선정하였다. dataset 은 age, sex, exng, caa, cp, trtbps, chol, fbs, restecg, thalachh 등 특징 값에 따른 심장 마비 가능성(target)에 대한 정보를 담고있다. 이 데이터를 사용하여 분석해보면 하나 또는 둘 이상의 정보(특성)에 대하여 심장 마비 가능성(target)에 대한 상관관계가 존재할 것이라고 생각하였고, 분석한 상관관계를 이해하고 이용하여 모델링하면 정확도가 높은 분류 모델을 구현할 수 있을 것이라고 생각하였다. 또 이 데이터의 target 값(output)은 0 또는 1 의 값만 가지기 때문에 이진 분류 모델을 모델링 하기에 적합하다고 생각하였다.

2) 데이터 전처리

- Exploratory Data Analysis

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       303 non-null    int64  
 1   sex       303 non-null    int64  
 2   cp        303 non-null    int64  
 3   trtbps    303 non-null    int64  
 4   chol      303 non-null    int64  
 5   fbs       303 non-null    int64  
 6   restecg   303 non-null    int64  
 7   thalachh  303 non-null    int64  
 8   exng      303 non-null    int64  
 9   oldpeak   303 non-null    float64
 10  slp       303 non-null    int64  
 11  caa       303 non-null    int64  
 12  thall     303 non-null    int64  
 13  output    303 non-null    int64  
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

'Heart Attack' dataset 은 303 개의 data 가 담겨있고 age, sex, cp, trtbps, chol, fbs, restecg, thalachh, exng, oldpeak, slp, caa, thall, output 으로 14 개의 column 으로 구성되어 있다. 사용하는 'Heart Attack' dataset 은 이미 문자열 정보에 대하여 value 값으로 Encoding 이 되어 있는 dataset 이기 때문에 모든 column 의 Dtype 이 사이킷런 알고리즘이 허용하는 'int' 나 'float' 인 것을 확인하였다. 그렇기 때문에 따로 Label Encoding 을 진행하지 않았다.

```
df.isnull().sum()
```

```
age         0
sex         0
cp          0
trtbps      0
chol        0
fbs         0
restecg     0
thalachh    0
exng        0
oldpeak     0
slp         0
caa         0
thall       0
output      0
dtype: int64
```

Missing data 가 없는 것을 확인하였다.

```
df.duplicated().sum()
```

```
1
```

중복 값을 가지는 행이 존재하는 것을 확인하였다.

```
df.drop_duplicates(inplace=True)
df.shape
```

```
(302, 14)
```

중복 값을 가지는 행을 drop 하였다. 중복됐던 행을 삭제함으로써, 원래 303 개의 data 가 담겨있는 dataset 에서 302 개의 data 가 담겨있는 dataset 으로 변경된 것을 확인하였다.

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
age	302.0	54.420530	9.047970	29.0	48.00	55.5	61.00	77.0
sex	302.0	0.682119	0.466426	0.0	0.00	1.0	1.00	1.0
cp	302.0	0.963576	1.032044	0.0	0.00	1.0	2.00	3.0
trtbps	302.0	131.602649	17.563394	94.0	120.00	130.0	140.00	200.0
chol	302.0	246.500000	51.753489	126.0	211.00	240.5	274.75	564.0
fbs	302.0	0.149007	0.356686	0.0	0.00	0.0	0.00	1.0
restecg	302.0	0.526490	0.526027	0.0	0.00	1.0	1.00	2.0
thalachh	302.0	149.569536	22.903527	71.0	133.25	152.5	166.00	202.0
exng	302.0	0.327815	0.470196	0.0	0.00	0.0	1.00	1.0
oldpeak	302.0	1.043046	1.161452	0.0	0.00	0.8	1.60	6.2
slp	302.0	1.397351	0.616274	0.0	1.00	1.0	2.00	2.0
caa	302.0	0.718543	1.006748	0.0	0.00	0.0	1.00	4.0
thall	302.0	2.314570	0.613026	0.0	2.00	2.0	3.00	3.0
output	302.0	0.543046	0.498970	0.0	0.00	1.0	1.00	1.0

data 들의 각종 통계량을 요약해서 보았다.

```
cat_cols = ['sex','exng','caa','cp','fbs','restecg','slp','thall']
con_cols = ["age","trtbps","chol","thalachh","oldpeak"]
target_col = ["output"]
print("The categorial cols are : ", cat_cols)
print("The continuous cols are : ", con_cols)
print("The target variable is :  ", target_col)
```

```
The categorial cols are :  ['sex', 'exng', 'caa', 'cp', 'fbs', 'restecg
', 'slp', 'thall']
The continuous cols are :  ['age', 'trtbps', 'chol', 'thalachh', 'oldpe
ak']
The target variable is :   ['output']
```

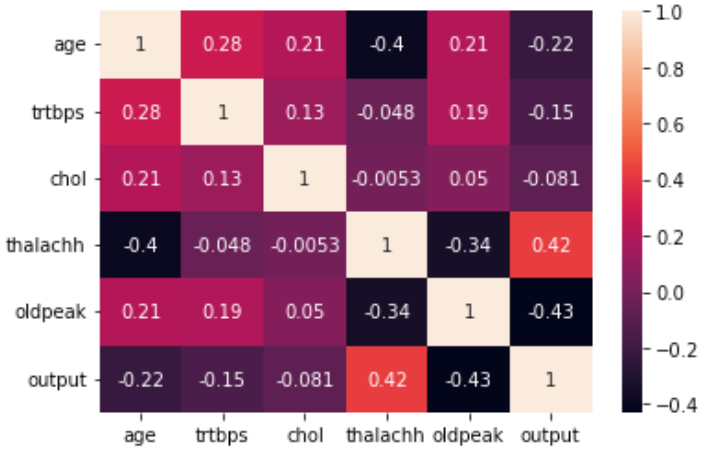
위에서 확인한 통계량을 토대로 카테고리형 정보를 가지는 열을 'cat_cols'로, 연속적인 정보를 가지는 열을 'con_cols'로, target 열을 'target_col'로 구분해주었다.

```
df1 = df.drop(cat_cols, axis=1)
df1.head()
```

	age	trtbps	chol	thalachh	oldpeak	output
0	63	145	233	150	2.3	1
1	37	130	250	187	3.5	1
2	41	130	204	172	1.4	1
3	56	120	236	178	0.8	1
4	57	120	354	163	0.6	1

```
sns.heatmap(df1.corr().transpose(), annot=True)
```

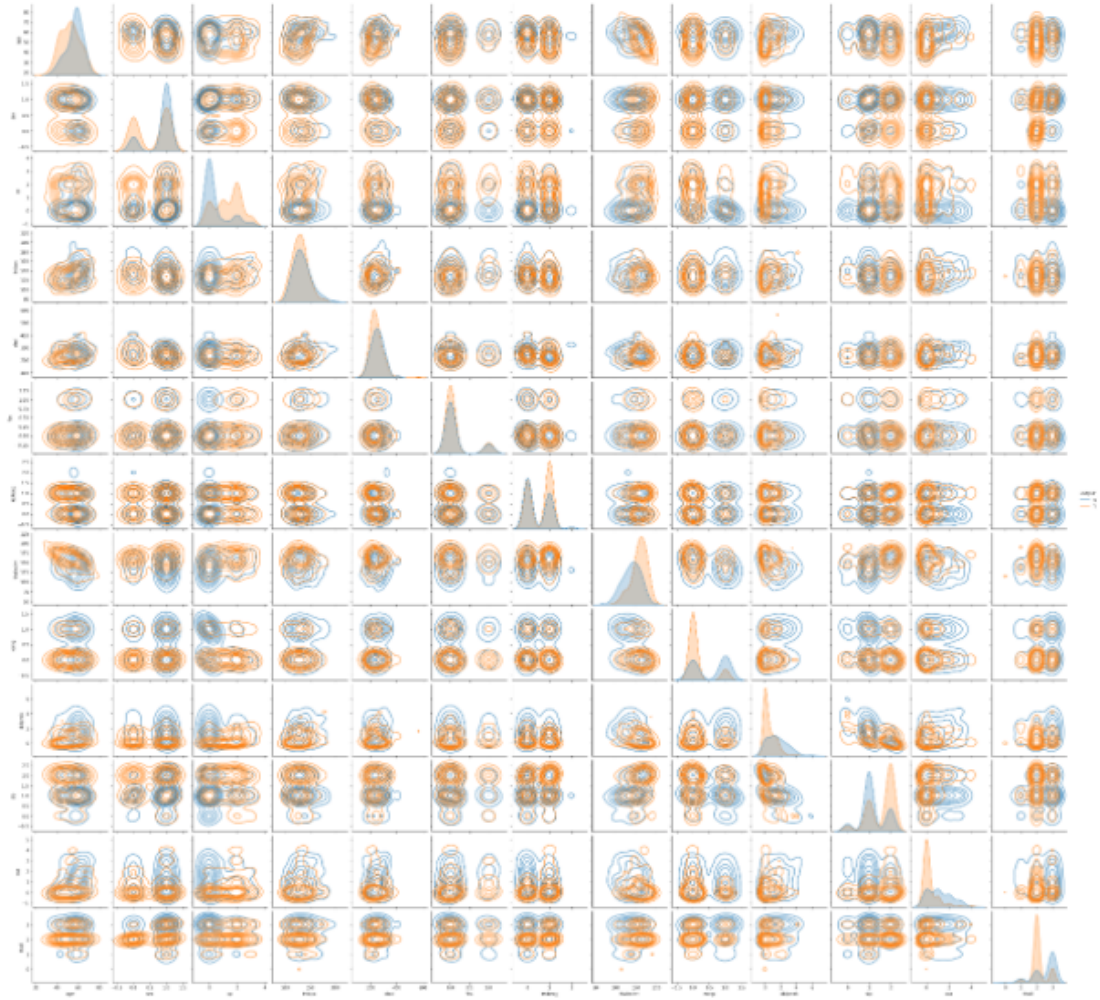
```
<AxesSubplot:>
```



연속적인 정보를 가지는 열들 간에 상관관계를 확인하기 위해 heatmap 을 통해 카테고리형 정보를 가지는 'cat_cols'을 제외한 열들의 Correlation Matrix 를 확인하였다. 이를 통해 각 변수들 간에 상관관계를 확인하였다. target 값인 'output' 열과는 특히 달성한 최대 심박수의 정보를 나타내는 'thalachh' 열이 유의미한 상관관계를 보였다.

```
sns.pairplot(df,kind="kde",hue="output")
```

```
<seaborn.axisgrid.PairGrid at 0x7fb290486400>
```



추가적으로, pairplot 을 이용해 그래프를 출력하여 데이터를 시각화 해보았다. hue 값을 추가해 target 값인 'output'을 기준으로 data 칼럼들의 모든 조합에 대해 상관관계를 확인할 수 있다. 이 때, plot 방식은 커널이라는 함수를 겹치는 방법으로 히스토그램보다 부드러운 형태의 분포 곡선을 보여주는 방법인 커널 밀도(kernel density)를 사용하였다.

```
X=df.drop(["output"],axis=1)
y=df["output"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2)
x_train = X_train
x_test = X_test
```

본격적으로 분류 모델을 모델링하기 전에 데이터를 학습 데이터와 test 데이터로 나눈다. 학습데이터를 전체 데이터의 80%, test 데이터를 20%로 설정하였다.

```
StandardScaler = StandardScaler()
X_train_scaled = StandardScaler.fit_transform(X_train)
X_test_scaled = StandardScaler.transform(X_test)
```

```
MinMaxScaler = MinMaxScaler(feature_range=(0, 1))
x_train_scaled = MinMaxScaler.fit_transform(x_train)
x_test_scaled = MinMaxScaler.transform(x_test)
```

데이터를 모델링하기 전에 데이터의 값이 너무 크거나 작은 경우에 모델 학습 과정에서 0 으로 수렴하거나 무한으로 발산하는 것을 방지하고, 다차원의 값들을 비교 분석하기 쉽게 만들고, 독립 변수의 공분산 행렬의 조건수를 감소시켜 최적화 과정에서의 안정성 및 수렴 속도를 향상 시키게 위해 두가지 스케일러를 사용하였다

feature 의 평균을 0, 분산을 1 로 변경하는 StandardScaler 와 모든 feature 값이 최소 0, 최대 1 로 있도록 정규화하여 데이터를 변환해주는 MinMaxScaler 를 사용하였다.

3) 분류 및 파라미터 최적화 + 결과 및 분석

〈 로지스틱 (Logistic) 회귀 〉

• StandardScaler

```
model = LogisticRegression()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
90.1639344262295
```

• MinMaxScaler

```
model = LogisticRegression()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
88.52459016393442
```

로지스틱 회귀의 목적은 종속 변수와 독립 변수간의 관계를 구체적인 함수로 나타내어 향후 예측 모델에 사용하는 것이다.

Logistic Regression 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 를 사용했을 때 정확도가 약 90.164 로 높은 정확도를 보여주었다.

〈 KNeighborsClassifier 〉

• StandardScaler

```
model = KNeighborsClassifier(n_neighbors=1)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
72.1311475409836
```

• MinMaxScaler

```
model = KNeighborsClassifier(n_neighbors=1)

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
75.40983606557377
```

K-최근접 이웃은 유사한 특성을 가진 데이터는 유사한 범주에 속하는 경향이 있다는 가정하에 사용된다.

KNeighborsClassifier 을 사용하여 데이터를 학습시키고, n_neighbors=1 로 설정하여 test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. MinMaxScaler 를 사용했을 때 정확도가 약 75.401 이었다. Logistic Regression 을 사용하여 학습했을 때보다 낮은 정확도가 확인할 수 있었다. 이는 n_neighbors 의 값이 너무 작았기 때문에 Overfitting(과적합) 문제가 발생했고 이로 인해 학습 모델의 성능이 낮아진 것이라고 생각됐다.

```
error_rate = []

for i in range(1, 40):

    model = KNeighborsClassifier(n_neighbors = i)
    model.fit(x_train, y_train)
    pred_i = model.predict(x_test)
    error_rate.append(np.mean(pred_i != y_test))

plt.figure(figsize =(10, 6))
plt.plot(range(1, 40), error_rate, color ='blue',
                 linestyle ='dashed', marker ='o',
         markerfacecolor ='red', markersize = 10)

plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

```
Text(0, 0.5, 'Error Rate')
```

Overfitting(과적합) 문제를 해결하고 학습 모델의 성능을 높이기 위해 KNeighborsClassifier 의 파라미터인 n_neighbors 값을 튜닝을 해주었다. 1~40 의 값을 가지는 n_neighbors(K) 값에 따라 오차율을 확인하였다. 이 때, 앞에서 StandardScaler 보다 MinMaxScaler 를 사용했을 때 높은 성능을 보여주었기 때문에 MinMaxScaler 를 사용하였다. n_neighbors(K)=5 일 때 가장 낮은 오차율을 보여주는 것을 확인하였다.

```
model = KNeighborsClassifier(n_neighbors=5)

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
80.32786885245902
```

MinMaxScaler 를 사용하고, n_neighbors=5 일 때 정확도가 약 80.328 인 것을 확인하였다. 파라미터 최적화를 진행하여 성능이 조금 향상되었지만 여전히 Logistic Regression 을 사용하여 학습했을 때보다 낮은 정확도였다.

〈 VotingClassifier 〉

• StandardScaler

```
lr = LogisticRegression(solver='liblinear')
knn = KNeighborsClassifier(n_neighbors=5)

model = VotingClassifier(estimators=[('LR', lr), ('KNN', knn)],
                         voting='soft')

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
90.1639344262295
```

• MinMaxScaler

```
lr = LogisticRegression(solver='liblinear')
knn = KNeighborsClassifier(n_neighbors=5)

model = VotingClassifier(estimators=[('LR', lr), ('KNN', knn)],
                         voting='soft')

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
86.88524590163934
```

보팅 분류기는 여러 개의 분류기를 생성하고 예측을 결합하여 보다 정확한 최종 예측을 도출하는 기법의 분류기이다.

사용한 여러 분류기로는 앞에서 사용한 Logistic Regression 와 KNeighborsClassifier 를 사용하였다. 이 때, KNeighborsClassifier 의 파라미터인 n_neighbors 값은 앞에 과정에서 파라미터 최적회를 통해 결정한 값으로 설정해주었다.

다수결 원칙을 사용하는 하드 보팅(Hard Voting)을 사용하지 않고, 각 분류기의 레이블 값 결정 확률을 모두 더하고 평균하여 가장 확률이 높은 레이블 값을 최종 결과값으로 사용하는 소프트 보팅(Soft Voting)을 사용하였다.

VotingClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 를 사용했을 때 정확도가 약 90.164 이었다. 이는 Logistic Regression 을 사용하여 데이터를 학습시켰을 때와 같은 정확도이다.

〈 DecisionTreeClassifier 〉

• StandardScaler

```
model = DecisionTreeClassifier(max_depth=1)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
67.21311475409836
```

• MinMaxScaler

```
model = DecisionTreeClassifier(max_depth=1)

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
67.21311475409836
```

결정 트리 분류기는 데이터의 규칙을 학습을 통해 자동으로 찾아내서 트리 기반의 분류 규칙을 만드는 것으로, 어떤 기준으로 규칙을 만드는지에 따라 성능이 달라진다.

DecisionTreeClassifier 을 사용하여 데이터를 학습시키고, max_depth =1 로 설정하여 test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 와 MinMaxScaler 를 사용했을 때 약 67.213 으로 같은 정확도였다. 트리의 최대 깊이를 설정하는 파라미터인 max_depth 값을 1 로 설정했기 때문에 다소 낮은 정확도를 보였다.

```
params = {
    'max_depth' : [1,2,3,4,5,6,7,8,9],
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.7967687074829932
{'max_depth': 3}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, StandardScaler 와 MinMaxScaler 를 사용했을 때 동일한 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. max_depth=3 일 때 약 0.797 로 가장 성능이 좋았다.

```
model = DecisionTreeClassifier(max_depth=3)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
78.68852459016394
```

max_depth=3 일 때 정확도가 약 78.689 인 것을 확인하였다. 파라미터 최적화를 진행하여 성능이 조금 향상되었지만 여전히 Logistic Regression 을 사용하여 학습했을 때보다 낮은 정확도였다.

〈 랜덤 포레스트 (Random Forest) Classifier 〉

• StandardScaler

```
model = RandomForestClassifier()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
83.60655737704919
```

• MinMaxScaler

```
model = RandomForestClassifier()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
80.32786885245902
```

랜덤포레스트는 배깅의 대표적인 알고리즘으로 여러 개의 결정 트리 분류기가 배깅 방식으로, 각자의 데이터를 샘플링하여 개별적으로 학습한 후 보팅을 통해 예측을 결정한다.

RandomForestClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 를 사용했을 때 정확도가 약 83.607 이었다.

```
params = {
    'max_depth' : [5, 10, 15, 20],
    'min_samples_leaf' : [4, 6, 8],
    'min_samples_split' : [4, 6, 8],
    'n_estimators' : [100, 200]
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.8465986394557824
{'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 6, 'n_esti
mators': 200}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, MinMaxScaler 사용했을 때보다 StandardScaler 를 사용했을 때 높은 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. max_depth=5, min_samples_leaf=4, min_samples_split=6, n_estimators=200 일 때 약 0.847 로 가장 성능이 좋았다.

```
model = RandomForestClassifier(max_depth=5, min_samples_leaf=4,
                               min_samples_split=6, n_estimators=200)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
85.24590163934425
```

max_depth=5, min_samples_leaf=4, min_samples_split=6, n_estimators=200 일 때 정확도가 약 85.246 인 것을 확인하였다. 파라미터 최적화를 진행하여 성능이 조금 향상되었지만 여전히 Logistic Regression 을 사용하여 학습했을 때보다 낮은 정확도였다.

〈 AdaBoostClassifier 〉

• StandardScaler

```
model = AdaBoostClassifier()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
80.32786885245902
```

• MinMaxScaler

```
model = AdaBoostClassifier()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
80.32786885245902
```

AdaBoost 분류기는 내부에 약한 성능의 분류기를 가지는 앙상블 모델로, 이전의 분류기가 잘못 분류한 샘플에 대해서 가중치를 높여서 다음 모델을 훈련시킨다. 반복마다 샘플에 대한 가중치를 수정하는 것이다.

AdaBoostClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 와 MinMaxScaler 를 사용했을 때 약 80.328 으로 같은 정확도였다.

```
params = {
    'n_estimators' : [10, 20, 30, 40, 50],
    'learning_rate': [0.05, 0.1, 0.2]
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.8506802721088436
{'learning_rate': 0.1, 'n_estimators': 20}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, StandardScaler 와 MinMaxScaler 를 사용했을 때 동일한 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. n_estimators=20, learning_rate=0.1 일 때 약 0.851 로 가장 성능이 좋았다.

```
model = AdaBoostClassifier(n_estimators=20, learning_rate=0.1)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
78.68852459016394
```

n_estimators=20, learning_rate=0.1 일 때 정확도가 약 78.689 인 것을 확인하였다. 파라미터 최적화를 진행하였지만 파라미터를 제한하기 전보다 오히려 조금 정확도가 낮아졌다. 아는 2 개의 파라미터만 최적화를 진행하였기 때문이었을 것이라고 예상된다.

〈 GradientBoostingClassifier 〉

• StandardScaler

```
model = GradientBoostingClassifier()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
78.68852459016394
```

• MinMaxScaler

```
model = GradientBoostingClassifier()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
78.68852459016394
```

GradientBoosting 분류기는 내부에 약한 성능의 분류기를 가지는 앙상블 모델로, 이전 분류기가 만든 잔여 오차에 대해 새로운 분류기를 만든다. AdaBoost 분류기와 유사하지만 가중치를 업데이트 할 때 경사 하강법(Gradient Descent)을 이용한다.

GradientBoostingClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 와 MinMaxScaler 를 사용했을 때 약 78.689 으로 같은 정확도였다.

```
params = {
    'max_depth' : [1,2,3,4,5,6,7,8,9],
    'n_estimators' : [100, 200]
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.8009353741496599
{'max_depth': 1, 'n_estimators': 100}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, StandardScaler 와 MinMaxScaler 를 사용했을 때 동일한 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. max_depth=1, n_estimators=100 일 때 약 0.800 로 가장 성능이 좋았다.

```
model = GradientBoostingClassifier(max_depth=1, n_estimators=100)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
88.52459016393442
```

max_depth=1, n_estimators=100 일 때 정확도가 약 88.525 인 것을 확인하였다.

〈 XGBClassifier 〉

• StandardScaler

```
model = XGBClassifier()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
83.60655737704919
```

• MinMaxScaler

```
model = XGBClassifier()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
83.60655737704919
```

XGBoost 는 Extreme Gradient Boosting 의 약자로, Boosting 기법을 이용하여 구현한 Gradient Boost 알고리즘을 병렬 학습이 지원되도록 구현한 라이브러리로 Overfitting 이 적은 방법이다.

XGBClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 와 MinMaxScaler 를 사용했을 때 약 83.607 으로 같은 정확도였다.

```
params = {
    'max_depth' : [3,5,7,9],
    'min_child_weight' : [1,3,5,7],
    'colsample_bytree' : [0.5, 0.75, 0.1],
    'n_estimators' : [100, 200]
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.8300170068027212
{'colsample_bytree': 0.1, 'max_depth': 3, 'min_child_weight': 5, 'n_est
imators': 100}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, StandardScaler 와 MinMaxScaler 를 사용했을 때 동일한 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. colsample_bytree=0.1, max_depth=3, min_child_weight=5, n_estimators=100 일 때 약 0.830 로 가장 성능이 좋았다.

```
model = XGBClassifier(colsample_bytree=0.1,
                      max_depth=3,
                      min_child_weight=5,
                      n_estimators=100)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
81.9672131147541
```

colsample_bytree=0.1, max_depth=3, min_child_weight=5, n_estimators=100 일 때 정확도가 약 81.967 인 것을 확인하였다. 파라미터 최적화를 진행하였지만 파라미터를 제한하기 전보다 오히려 조금 정확도가 낮아졌다.

〈 LGBMClassifier 〉

• StandardScaler

```
model = LGBMClassifier()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
85.24590163934425
```

• MinMaxScaler

```
model = LGBMClassifier()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
85.24590163934425
```

Light GBM 은 리프 중심 트리 분할 방식을 사용하는 Tree 기반 학습 알고리즘이다. Light GBM 은 Tree 가 수직적으로 확장되는 반면에 다른 Tree 알고리즘은 Tree 가 수평적으로 확장됩니다

LGBMClassifier 을 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 와 MinMaxScaler 를 사용했을 때 약 85.246 으로 같은 정확도였다.

```
params = {
    'n_estimators' : [100, 200, 300, 400, 500]
}
cv = GridSearchCV(model, param_grid=params, scoring='accuracy', cv=5)
cv.fit(X_train_scaled, y_train)
print(cv.best_score_)
print(cv.best_params_)
```

```
0.7923469387755102
{'n_estimators': 300}
```

최적의 하이퍼 파라미터 값을 찾기 위해 GridSearchCV 를 사용하였다. 이 때, StandardScaler 와 MinMaxScaler 를 사용했을 때 동일한 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. n_estimators=300 일 때 약 0.792 로 가장 성능이 좋았다.

```
model = LGBMClassifier(n_estimators=300)

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
81.9672131147541
```

n_estimators=300 일 때 정확도가 약 81.967 인 것을 확인하였다. 파라미터 최적화를 진행하였지만 파라미터를 제한하기 전보다 오히려 조금 정확도가 낮아졌다. 아는 1 개의 파라미터만 최적화를 진행하였기 때문이었을 것이라고 예상된다.

〈 SVM(Support Vector Machine) 〉

• StandardScaler

```
model = SVC()

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
90.1639344262295
```

• MinMaxScaler

```
model = SVC()

model.fit(x_train_scaled, y_train)
predict = model.predict(x_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
86.88524590163934
```

SVM(Support Vector Machine)은 두 카테고리 중 어느 하나에 속한 데이터의 집합이 주어졌을 때, SVM 알고리즘은 주어진 데이터 집합을 바탕으로 하여 새로운 데이터가 어느 카테고리에 속할지 판단하는 비확률적 이진 선형 분류 모델을 만든다. 만들어진 분류 모델은 데이터가 사상된 공간에서 경계로 표현되는데 SVM 알고리즘은 그 중 가장 큰 폭을 가진 경계를 찾는 알고리즘이다.

SVC 를 사용하여 데이터를 학습시키고, test 데이터를 인자로 받아 학습이 완료된 모델의 정확도를 출력했다. StandardScaler 를 사용했을 때 정확도가 약 90.164 이었다.

```
# kernel = "linear": 선형 SVM
model = SVC(kernel='linear')

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
86.88524590163934
```

```
# kernel = "poly": 다항 커널
model = SVC(kernel='poly')

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
88.52459016393442
```

```
#  kernel = "rbf" 또는 kernel = None: RBF 커널
model = SVC(kernel='rbf')

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
90.1639344262295
```

```
# kernel = "sigmoid": 시그모이드 커널
model = SVC(kernel='sigmoid')

model.fit(X_train_scaled, y_train)
predict = model.predict(X_test_scaled)
acc = accuracy_score(y_test, predict)

print(acc*100)
```

```
88.52459016393442
```

커널의 종류에 따라 성능을 비교해보았다. 이 때, MinMaxScaler 를 사용했을 때보다 StandardScaler 를 사용했을 때 더 좋은 성능을 보여주었기 때문에 StandardScaler 를 사용하였다. RBF 커널을 사용했을 때 정확도가 약 90.164 로 가장 좋았다.

• 결과 분석

학습 모델	구현한 최고 정확도	학습 모델	구현한 최고 정확도
Logistic Regression	약 90.164	KNeighborsClassifier	약 80.328
VotingClassifier	약 90.164	DecisionTreeClassifier	약 78.689
RandomForestClassifier	약 85.246	AdaBoostClassifier	약 80.328
GradientBoostingClassifier	약 88.525	XGBClassifier	약 83.607
LGBMClassifier	약 85.246	SVC	약 90.164

정확도를 기준으로 비교해봤을 때, Logistic Regression 알고리즘을 사용한 모델, VotingClassifier 를 사용한 모델, SVC 를 사용한 모델의 정확도가 약 90.164 로 가장 좋았다. VotingClassifier 를 사용한 모델의 분류기로 Logistic Regression 를 사용했기 때문에 선정한 'Heart Attack' dataset 에 가장 적합한 학습 모델은 Logistic Regression 알고리즘을 사용한 학습 모델과 RBF 커널을 사용한 SVC 학습 모델이라고 할 수 있다.

4) 고찰

여러가지 분류 모델을 모델링 해보면서 그 방법과 그 특성에 대해 공부할 수 있었다. 또한 원하는 목적에 맞게 dataset 을 선별하는 방법과 찾은 dataset 을 전처리하는 방법에 대해서도 처음부터 끝까지 직접 해보며 습득하였다. 이 과정 속에서 같은 분류 알고리즘을 사용한다고 하더라도 데이터의 상관 관계를 분석하고 이를 이용해 데이터의 중요도를 파악하여 데이터 전처리를 어떻게 해주는지에 따라 분류 성능의 큰 차이가 발생한다는 것을 알 수 있었고 그렇기 때문에 목적에 맞게 데이터를 전처리 해주는 것이 무엇보다 중요한 과정이라는 것을 알 수 있었다.

각 분류 모델마다 StandardScaler 와 MinMaxScaler 를 사용했을 때 각각의 정확도를 확인하였는데 두 스케일러에서 같은 정확도가 나오는 경우가 있었고 실행할 때마다 다른 정확도가 나오는 경우도 있었기 때문에 선택한 dataset 에 대한 스케일러에 따른 결과 비교는 크게 의미 있지 않았다. 그렇지만 스케일러를 사용하지 않은 것보다 사용했을 때 성능이 더욱 높았고, 학습 모델의 성능을 최대한 높이기 위해서는 선택한 dataset 에 따라 적정한 스케일러를 선택하고 전처리 하는 과정은 중요한 것 같다는 생각이 들었다.

또 직접 파라미터를 튜닝하거나 GridSearchCV 를 이용하여 파라미터 튜닝을 진행하였는데, 튜닝한 파라미터로 해당 학습 모델의 파라미터를 제한한 것보다 안 한 것이 성능이 더 좋은 경우가 있었는데 이는 사용한 학습 모델의 모든 파라미터를 튜닝하지 않았기 때문이었을 것이라고 생각이 들었다.

GridSearchCV 를 사용해서 파라미터를 튜닝을 진행할 때, 적절한 하이퍼 파라미터 설정이 어려웠다. 동시에 여러 파라미터를 튜닝 할 때는 파라미터들의 값들의 조합에 따라 결과가 달라지기 때문에 GridSearchCV 사용해 파라미터 튜닝을 진행할 파라미터를 결정하고 해당 파라미터 마다 범위를 설정해주는 과정에서 어려움을 느꼈다.

1) 데이터 선정

'penguins_size.csv'

데이터셋 Link: https://www.kaggle.com/parulpandey/palmer-archipelago-antarctica-penguin-data

```
15 penguins = pd.read_csv('penguins_size.csv')
16 penguins.head(5)
```

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

Colums
• species: penguin species (Chinstrap, Adélie, or Gentoo)
• culmen_length_mm: culmen length (mm)
• culmen_depth_mm: culmen depth (mm)
• flipper_length_mm: flipper length (mm)
• body_mass_g: body mass (g)
• island: island name (Dream, Torgersen, or Biscoe) in the Palmer Archipelago (Antarctica)
• sex: penguin sex

Kaggle 의 "Palmer Archipelago (Antarctica) penguin data" dataset 은 culmen 길이(culmen_length_mm), culmen 깊이(culmen_depth_mm), flipper 길이(flipper_length_mm), body_mass(body_mass_g), 남극의 섬 이름(island), 펭귄 성별(sex), 그리고 펭귄 종(species)이 있다. Species 는 Chinstrap, Adélie 또는 Gentoo 3 개의 class 로 구성되어 있다.

데이터 선정 이유:
Dataset 은 kaggle 의 “Palmer Archipelago (Antarctica) penguin data”로 선정하였다. 이 dataset 은 남극 펭귄의 신체적 특성이 종 마다 비슷한 값을 가지고 있기 때문에 그 특성을 기반으로 군집화 하기에 적절하다고 생각하였다. 또한 만약 target 값인 펭귄 종(species)의 종류가 2 종류로 구성되어 있었다면 군집화를 하는 것이나 위에서 진행한 이진 분류를 하는 것이나 크게 차이 없겠지만 선택한 dataset 은 target 값이 3 개로 구성되어 있기 때문에 이진 분류 모델과 다르게 군집화의 결과를 확인하기에 적절할 것 같다고 생각하였다.

2) 데이터 전처리

- Exploratory Data Analysis

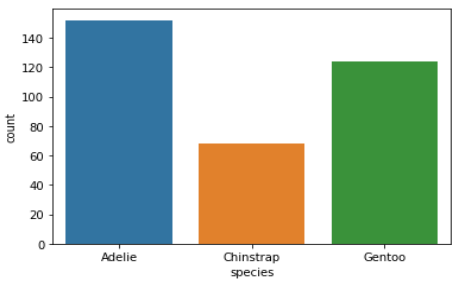
```
1  penguins.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   species            344 non-null    object
 1   island             344 non-null    object
 2   culmen_length_mm   342 non-null    float64
 3   culmen_depth_mm    342 non-null    float64
 4   flipper_length_mm  342 non-null    float64
 5   body_mass_g        342 non-null    float64
 6   sex                334 non-null    object
dtypes: float64(4), object(3)
memory usage: 18.9+ KB
```

위 dataset 에는 총 344 개의 펭귄 데이터가 들어있다.

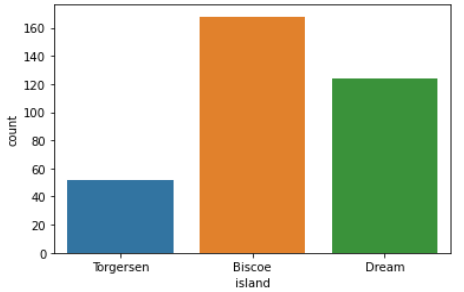
```
1  #exploring the data
2  sns.countplot(x = "species", data = penguins)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2aae16e7508>
```




```
1  sns.countplot(x = "island", data = penguins)
```

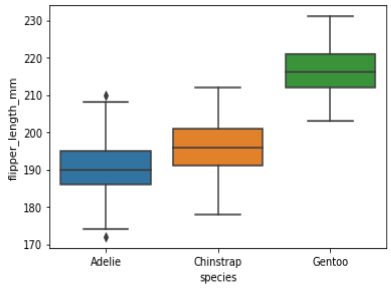
```
<matplotlib.axes._subplots.AxesSubplot at 0x2aae185f308>
```



펭귄의 수는 Adelie 종이 가장 많았으며 그 다음으로는 Gentoo, Chinstrap 순서로 많았다. 그리고 대부분의 펭귄은 Biscoe island 에서 왔고 Torgersen 에서 온 펭귄이 가장 적었다.

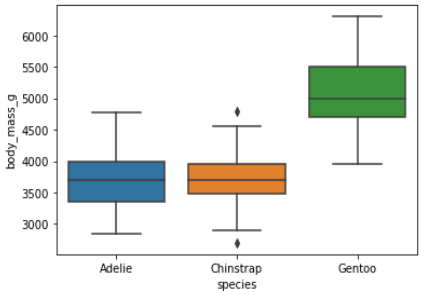
```
sns.boxplot(x = 'species', y = 'flipper_length_mm', data = penguins)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2aade0d7f88>
```




```
sns.boxplot(x = 'species', y = 'body_mass_g', data = penguins)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2aae152cb48>
```



Gentoo 종이 Flipper 의 길이가 가장 길고, 가장 무겁다.

missing data 확인

```
penguins.isna().sum()
```

```
species              0
island               0
culmen_length_mm     2
culmen_depth_mm      2
flipper_length_mm    2
body_mass_g          2
sex                 10
dtype: int64
```

군집화를 하기 전 missing data 가 있는지 확인한다. 확인해본 결과 culmen_length_mm, culmen_depth_mm, flipper_length_m, body_mass_g 에 missing data 가 2 개씩, sex 에 10 개가 있는 것을 확인할 수 있었다.

```
penguins["culmen_length_mm"] = penguins["culmen_length_mm"].fillna(value = penguins["culmen_length_mm"].mean())
penguins["culmen_depth_mm"] = penguins["culmen_depth_mm"].fillna(value = penguins["culmen_depth_mm"].mean())
penguins["flipper_length_mm"] = penguins["flipper_length_mm"].fillna(value = penguins["flipper_length_mm"].mean())
penguins["body_mass_g"] = penguins["body_mass_g"].fillna(value = penguins["body_mass_g"].mean())

penguins.isna().sum()
```

```
species              0
island               0
culmen_length_mm     0
culmen_depth_mm      0
flipper_length_mm    0
body_mass_g          0
sex                 10
dtype: int64
```

Missing data 에는 각 컬럼 값들의 평균 값을 채워주었다.

```
penguins['sex'] = penguins['sex'].fillna('FEMALE')
```

sex 컬럼에는 'FEMALE'으로 missing data 들을 채워주었다.

```
1 penguins.isna().sum()
```

```
species              0
island               0
culmen_length_mm     0
culmen_depth_mm      0
flipper_length_mm    0
body_mass_g          0
sex                  0
dtype: int64
```

누락된 데이터는 모두 채워주었다.

```
1 penguins['sex'].unique()
```

```
array(['MALE', 'FEMALE', '.'], dtype=object)
```

Gentoo	Biscoe	44.5	15.7	217	4875	.

```
1 penguins.drop(penguins[penguins['sex']=='.'].index, inplace=True)
```

sex 데이터 중에 '.'으로 되어있는 row 가 있어 그 row 는 삭제하였다.

Label Encoding

```
label_sex = LabelEncoder()
penguins["sexEnc"] = label_sex.fit_transform(penguins["sex"])
print("Encoded sex" + str(label_sex.classes_))
```

```
Encoded sex['FEMALE' 'MALE']
```

```
label_island = LabelEncoder()
penguins["islandEnc"] = label_island.fit_transform(penguins["island"])
print("Encoded island" + str(label_sex.classes_))
```

```
Encoded island['FEMALE' 'MALE']
```

```
label_species = LabelEncoder()
penguins["speciesEnc"] = label_species.fit_transform(penguins["species"])
print("Encoded species" + str(label_species.classes_))
```

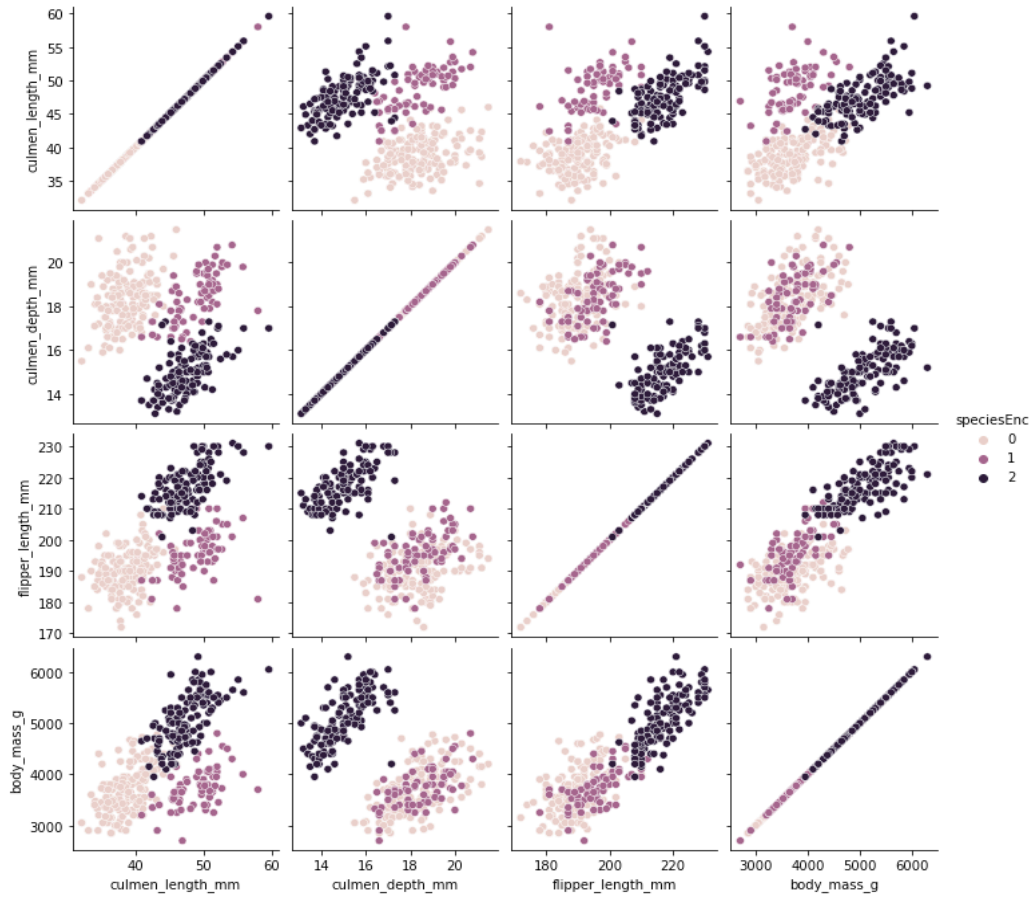
```
Encoded species['Adelie' 'Chinstrap' 'Gentoo']
```

sex 와 island, species 는 Label Encoding 과정을 통해 문자로 되어있는 data 를 라벨 숫자로 변환하였다.

```
1 penguins = penguins.drop(["sex", "island", "species"], axis=1)
```

인코딩 된 값을 사용할 것이기 때문에 기존에 문자로 되어있던 data 는 drop 시켰다.

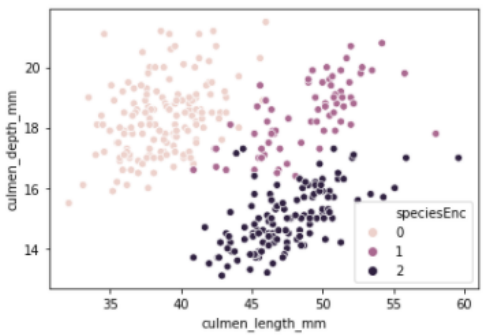
```
sns.pairplot(penguins.drop(['sexEnc', 'islandEnc'],axis=1), hue='speciesEnc', diag_kind=None)
plt.show()
```



Pairplot 을 통해 그래프를 출력하여 데이터를 시각화 해보았다. 이를 통해 data 컬럼들의 모든 조합에 대해 상관관계를 확인할 수 있었다. 위 그래프를 보면 culmen_length_mm, culmen_depth_mm 을 축으로 하였을 때 펭귄의 species 별로 값들이 뭉쳐 있어 좋은 군집화의 결과를 보기에 적당할 것이라고 생각하였다.

```
1 sns.scatterplot(x='culmen_length_mm', y='culmen_depth_mm', hue = 'speciesEnc', data = penguins)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x20eafc5a588>
```



culmen_length_mm, culmen_depth_mm 을 축으로 하였을 때 그래프 다시 확인해보았다.

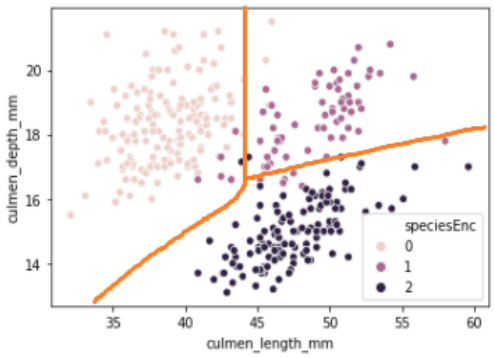
3) 군집화 및 파라미터 최적화 + 결과 및 분석

〈 KMeans 〉

```
penguins.head()
```

	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	islandEnc	speciesEnc
0	39.10000	18.70000	181.000000	3750.000000	2	0
1	39.50000	17.40000	186.000000	3800.000000	2	0
2	40.30000	18.00000	195.000000	3250.000000	2	0
3	43.92193	17.15117	200.915205	4201.754386	2	0
4	36.70000	19.30000	193.000000	3450.000000	2	0

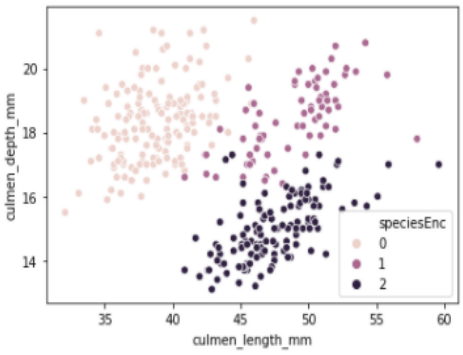
KMeans 알고리즘을 통해 군집화 하였다. 일반적인 군집화에서 많이 활용되고 데이터들이 같은 종끼리 뭉쳐 있어 데이터를 가장 가까운 중심점의 군집에 할당하여 각 군집에 속한 데이터들의 평균을 새로운 중심점으로 갱신하는 방법인 KMeans 알고리즘이 적절하다고 생각하였다.



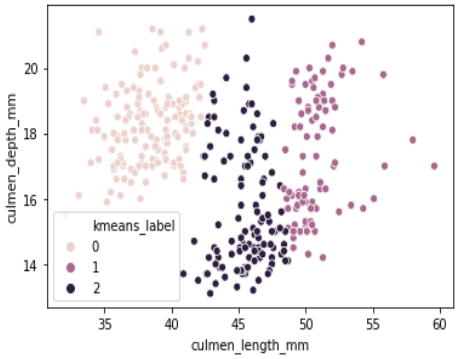
위 그림과 같이 세 군집으로 나누어 질 것으로 예상하였다.

```
kmeans = KMeans(n_clusters=3, max_iter=1000)
penguins['kmeans_label']=kmeans.fit_predict(penguins.drop(['speciesEnc', 'sexEnc', 'islandEnc',
                                                           'flipper_length_mm', 'body_mass_g'], axis=1))
```

```
1 sns.scatterplot(x='culmen_length_mm', y='culmen_depth_mm', hue = 'kmeans_label', data = penguins)
```



〈종 별로 색이 구분된 원본 데이터〉



〈KMeans 알고리즘으로 군집화한 데이터〉

위에서 culmen_length_mm, culmen_depth_mm 을 축으로 하여 그래프를 보았으므로 이 두 종류의 피처만으로 KMeans 알고리즘을 통해 군집화 해보았다.

비슷한 위치들끼리 묶여 군집화가 잘 된 듯 보이지만, 원본 데이터와 비교하였을 때 1, 2 번 군집에서 결과가 좋지 않았음을 확인할 수 있었다.

```
from sklearn.metrics import silhouette_samples, silhouette_score

score_samples = silhouette_samples(penguins,penguins['kmeans_label'])
print(score_samples.shape)

penguins['silhouette_coeff'] = score_samples
average_score = silhouette_score(penguins, penguins['kmeans_label'])
print(average_score)
```

```
(343,)
0.11572948172256721
```

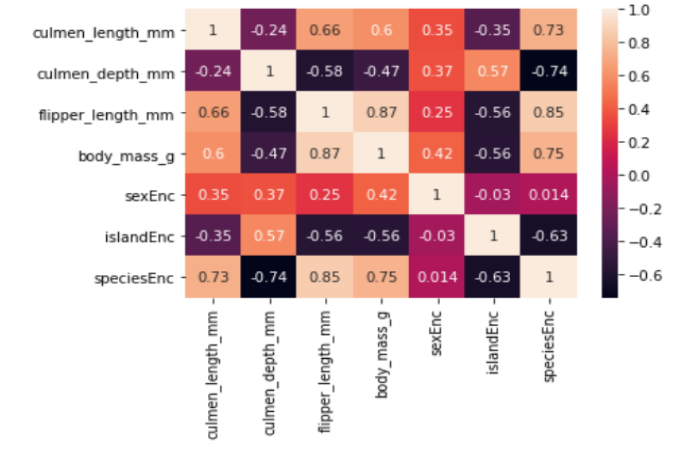
```
penguins.groupby('kmeans_label')['silhouette_coeff'].mean()
```

```
kmeans_label
0    0.422540
1   -0.012654
2   -0.216499
Name: silhouette_coeff, dtype: float64
```

위 군집화를 토대로 군집 평가를 진행해보았다. 실루엣 계수의 평균값이 0.116 정도로 낮은 값을 보였다. 군집 별 실루엣 계수 평균 값과 전체 평균 값의 차이도 많이 나는 것을 보니 좋지 않은 군집화가 이루어진 것을 알 수 있었다. 이는 위의 군집화 결과에서처럼 1 번과 2 번의 군집화가 제대로 이루어지지 않아서 그런 것이라고 예상되었다.

```
sns.heatmap(data = penguins.corr(), annot = True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x21bdb22b908>
```



heatmap 을 통해 각 변수 간에 상관관계를 표로 나타낸 것이다. Species 와 culmen_length_mm, culmen_depth_mm 의 상관계수가 높긴 하지만 다른 피처들의 상관계수 또한 높다. 이를 통해 위에서 1 과 2 종의 군집화가 제대로 이루어지지 않은 것은 상관관계가 높은 피처들이 더 있지만 drop 하고 두 개의 피처로만 군집화를 하여서 그런 것이라고 추측되었다.

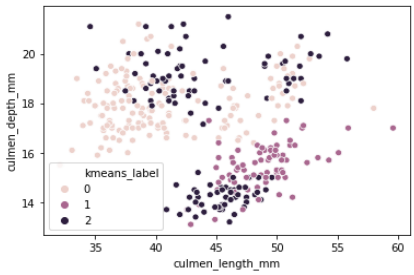
```
penguins = penguins.drop(["sexEnc"], axis=1)
```

heatmap 을 통한 각 변수 간에 상관관계 그래프를 보니 sex 컬럼은 0.014 로 영향을 크게 주지 않는다고 생각하여 drop 하였다.

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, max_iter=100)
penguins['kmeans_label'] = kmeans.fit_predict(penguins.drop(["speciesEnc"], axis=1))
```

```
sns.scatterplot(x='culmen_length_mm', y='culmen_depth_mm', hue = 'kmeans_label', data = penguins)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x21bdaf37e08>
```



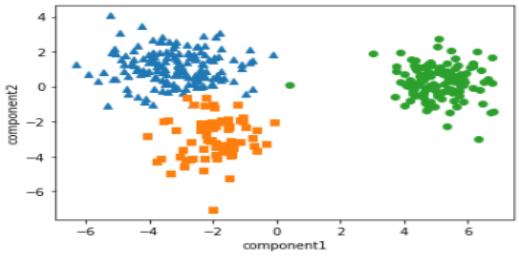
다른 피처들도 포함하여 군집화를 실행해보았다. 결과를 보니 눈으로 봐도 군집화가 잘 되지 않은 것을 확인할 수 있었다. 이는 차원의 저주(Curse of Dimensionality) 때문이라고 생각이 들었다. 차원의 저주란 데이터 학습을 위해 차원이 증가하면서 학습 데이터 수가 차원의 수 보다 적어져 성능이 저하되는 현상이다. 위의 경우에도 변수가 5 개로 늘어났기 때문에 차원이 5 개 이므로 문제가 발생한 것 같았다. 이는 차원축소를 통해 해결할 수 있을 것이라고 생각이 들었다.

LDA

```
import sys
from sklearn.preprocessing import StandardScaler

data_scaled = StandardScaler().fit_transform(penguins.iloc[:,:-1])

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=2)
data_lda = lda.fit(data_scaled, penguins.speciesEnc)
data_lda = lda.transform(data_scaled)
print(data_lda.shape)
```

```
(343, 2)
```

```
df_lda = pd.DataFrame(data_lda, columns=['component1','component2'])
df_lda['target']=penguins.speciesEnc

markers = ['^','s','o']

for i, marker in enumerate(markers):
    x_data = df_lda[df_lda['target'] == i]['component1']
    y_data = df_lda[df_lda['target'] == i]['component2']
    plt.scatter(x_data, y_data, marker=marker)

plt.xlabel('component1')
plt.ylabel('component2')
plt.show()
```



차원 축소 방법으로는 LDA 를 사용하였다. LDA 란 특징 공간 상에서 클래스 분리를 최대화하는 주축으로 사상시켜 선형 부공간으로 차원을 축소하는 방법이다. LDA 는 클래스 간 분산과 클래스 내부 분산의 비율을 최대화하는 방식으로 차원을 축소한다. 이 방법을 사용한 이유는 LDA 가 기본적으로 클래스를 가장 잘 구분할 수 있는 새로운 축을 찾아 주기 때문이다. 우리의 최종 목적은 species 에 맞게 군집화 하는 것이다. 이는 사실 분류와 비슷하다. LDA 는 지도학습이기 때문에 정답이 있을 때 사용 가능하다. 이 dataset 은 다양한 수치들을 통해 이미 species 가 구분되어 있는 data 이다. 따라서 LDA 를 사용하면 미리 dataset 의 species 값을 통해 sepcies 를 잘 구분할 수 있는 새로운 축을 찾을 수 있다고 생각하였다.

우선 각 Colums 마다 scale 의 범위가 다르기 때문에 같은 스케일로 변환해주어야 한다. 따라서 각 피처 데이터 값들을 동일한 스케일로 변환하고, 차원 축소를 진행하였다. 위의 LDA 결과를 보면 species 를 잘 구분할 수 있도록 각 species 가 서로 뭉쳐있고, 서로 다른 species 끼리는 떨어져 있는 것을 확인할 수 있다.

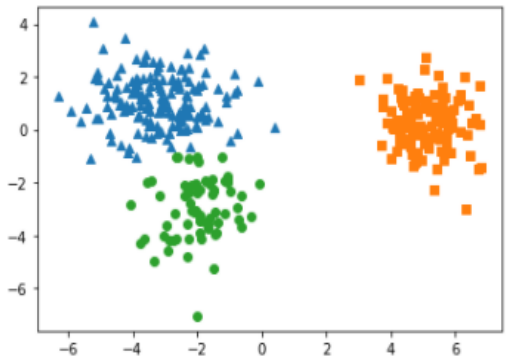
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
kmeans.fit_predict(df_lda.drop(['target'], axis=1))
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1])
```

```
df_lda['cluster'] = kmeans.labels_
df_lda['component1'] = data_lda[:,0]
df_lda['component2'] = data_lda[:,1]
```

```
marker0_index = df_lda[df_lda['cluster']==0].index
marker1_index = df_lda[df_lda['cluster']==1].index
marker2_index = df_lda[df_lda['cluster']==2].index
```

```
plt.scatter(x=df_lda.loc[marker0_index, 'component1'], y=df_lda.loc[marker0_index, 'component2'], marker='^')
plt.scatter(x=df_lda.loc[marker1_index, 'component1'], y=df_lda.loc[marker1_index, 'component2'], marker='s')
plt.scatter(x=df_lda.loc[marker2_index, 'component1'], y=df_lda.loc[marker2_index, 'component2'], marker='o')

plt.show()
```



축소된 차원을 이용하여 kmeans 알고리즘을 적용한 결과는 위와 같다. 결과를 보면 species 별로 잘 군집화 된 것을 확인할 수 있다.

```
from sklearn.metrics import silhouette_samples, silhouette_score

score_samples = silhouette_samples(df_lda,df_lda['cluster'])
print(score_samples.shape)

df_lda['silhouette_coeff'] = score_samples
average_score = silhouette_score(df_lda, df_lda['cluster'])
print(average_score)
```

```
(343,)
0.6889883219955076
```

```
df_lda.groupby('cluster')['silhouette_coeff'].mean()
```

```
cluster
0    0.615532
1    0.809981
2    0.640860
Name: silhouette_coeff, dtype: float64
```

군집평가를 해보면 위와 같다. 군집평가 결과, 실루엣 계수의 평균 값이 약 0.69 정도로 1 에 가깝고, 개별 군집의 실루엣 계수 평균 값이 전체 평균값과 비슷하기 때문에 군집화가 잘 된 것을 알 수 있다.

〈 MeanShift 〉

```
df_lda.head()
```

	component1	component2	target
0	-5.219932	1.175308	0.0
1	-2.931260	0.596634	0.0
2	-3.355162	-0.456898	0.0
3	-0.437249	-0.489702	0.0
4	-4.712161	1.140793	0.0

```
from sklearn.cluster import MeanShift
df_lda['target']=penguins.speciesEnc

x = data_lda
y = df_lda["target"]
```

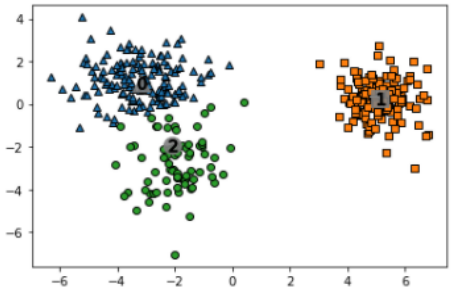
```
from sklearn.cluster import estimate_bandwidth

bandwidth = estimate_bandwidth(x)
print(bandwidth)
```

```
2.4743452035443623
```

```
meanshift = MeanShift(bandwidth=bandwidth)
cluster_label_mshift = meanshift.fit_predict(x)
print(np.unique(cluster_label_mshift))
```

```
[0 1 2]
```

```
df = pd.DataFrame(data=x, columns=['feature1', 'feature2'])
df['target'] = y

df['meanshift_label'] = cluster_label_mshift
centers = meanshift.cluster_centers_
unique_labels = np.unique(cluster_label_mshift)

markers = ['^','s','o']

for label in unique_labels:
    cluster_label_mshift = df[df['meanshift_label']==label]
    center = centers[label]
    plt.scatter(x=cluster_label_mshift['feature1'], y=cluster_label_mshift['feature2'], edgecolor='k', marker=markers[label])
    plt.scatter(x=center[0], y=center[1], s=200, color='gray', alpha=0.9, marker=markers[label])
    plt.scatter(x=center[0], y=center[1], s=100, color='k', edgecolor='k', marker='$%d$' % label)
plt.show()
```



Meanshift 로 군집화를 진행해보았다. Meanshift 는 소속된 데이터의 평균위치, 즉 밀도가 가장 높은 곳으로 중심 이동을 하는 방법이다. 최적의 대역폭을 계산하여 진행하였다. 결과는 kmeans 와 마찬가지로 군집화가 잘 된 것을 확인할 수 있었다.

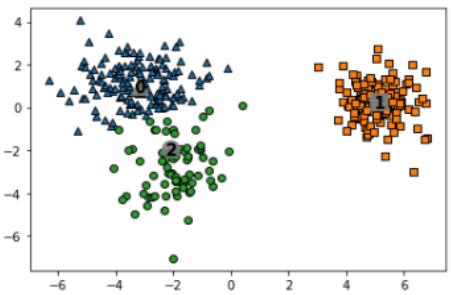
```
from sklearn.metrics import silhouette_samples, silhouette_score
df['target'] = df['target'].fillna(0.0)
score_samples = silhouette_samples(df,df['target'])
print(score_samples.shape)

df['silhouette_coeff'] = score_samples
average_score = silhouette_score(df, df['target'])
print(average_score)
```

```
(343,)
0.673784047008536
```

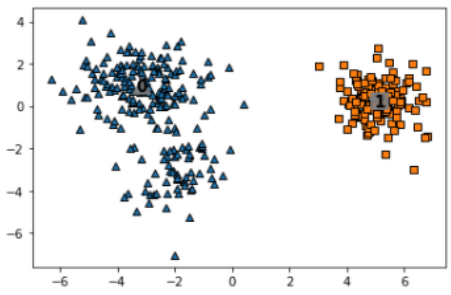
```
df.groupby('target')['silhouette_coeff'].mean()
```

```
target
0.0    0.592092
1.0    0.633369
2.0    0.800237
Name: silhouette_coeff, dtype: float64
```

군집평가를 해 본 결과, 실루엣 계수의 평균 값이 약 0.67 정도로 1 에 가깝고, 개별 군집의 실루엣 계수 평균 값이 전체 평균값과 비슷하기 때문에 군집화가 잘 된 것을 알 수 있다.



〈bandwidth = 약 2.47〉



〈bandwidth = 3.0〉

Meanshift 는 대역폭의 영향이 크다는 특징이 있기 때문에, 이를 확인하기 위해 대역폭을 수정하여 다시 진행해 보았다.

Bandwidth 를 키워 보았더니 군집이 2 가지로 줄어들었다. 이를 통해 Bandwidth 가 커질수록 군집의 개수가 감소하는 것을 확인할 수 있었다. 따라서 bandwidth 가 크면 Underfitting 의 가능성이 있고, bandwidth 가 작아질수록 군집 개수는 증가하여 overfitting 의 가능성이 생길 수 있다.

〈 GMM(Gaussian Mixture Model) 〉

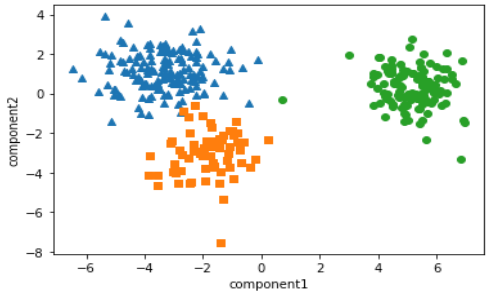
```
df_lda_g = df_lda.copy()
df_lda_g.head()
```

	component1	component2	target
0	-5.219932	1.175308	0.0
1	-2.931260	0.596634	0.0
2	-3.355162	-0.456898	0.0
3	-0.437249	-0.489702	0.0
4	-4.712161	1.140793	0.0

```
df_lda_g['target'] = penguins.speciesEnc

gmm = GaussianMixture(n_components=3)
df_lda_g['cluster'] = gmm.fit_predict(df_lda_g.drop(['target'], axis=1))
print(df_lda_g.groupby(['target'])['cluster'].value_counts())
```

```
target  cluster
0.0     0          151
        2            1
1.0     2           66
        0            2
2.0     1          121
        2            1
Name: cluster, dtype: int64
```

```
markers = ['^','s', 'o']

for i, marker in enumerate(markers):
    x_data = df_lda[df_lda_g['target'] == i]['component1']
    y_data = df_lda[df_lda_g['target'] == i]['component2']
    plt.scatter(x_data, y_data, marker=marker)

plt.xlabel('component1')
plt.ylabel('component2')
plt.show()
```



GMM 으로 군집화를 진행해보았다. GMM 은 이름 그대로 Gaussian 분포가 여러 개 혼합된 Clustering 알고리즘이다. 그렇기 때문에 데이터를 여러 개의 Gaussian 분포의 데이터 집합이 섞여서 생성된 데이터라고 가정한다. GMM Clustering 군집 개수(K)를 정하고 데이터 포인터들이 속하는 Cluster 를 확률적으로 할당하는 Soft Clustering 알고리즘이다.

결과는 Noise 가 많지 않은 것을 확인했고, 시각화 했을 때도 군집화가 잘 된 것을 확인할 수 있었다.

< DBSCAN >

```
def visualize_cluster_plot(clusterobj, dataframe, label_name, iscenter=True):
    if iscenter :
        centers = clusterobj.cluster_centers_

    unique_labels = np.unique(dataframe[label_name].values)
    markers=['o', 's', '^']
    isNoise=False

    for label in unique_labels:
        label_cluster = dataframe[dataframe[label_name]==label]
        if label == -1:
            cluster_legend = 'Noise'
            isNoise=True
        else :
            cluster_legend = 'Cluster '+str(label)

        plt.scatter(x=label_cluster['component1'], y=label_cluster['component2'], s=70,
                    edgecolor='k', marker=markers[label], label=cluster_legend)

        if iscenter:
            center_x_y = centers[label]
            plt.scatter(x=center_x_y[0], y=center_x_y[1], s=250, color='white',
                        alpha=0.9, edgecolor='k', marker=markers[label])
            plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k',
                        edgecolor='k', marker='$%d$' % label)
    if isNoise:
        legend_loc='upper center'
    else: legend_loc='upper right'

    plt.legend(loc=legend_loc)
    plt.show()

```

군집화된 그래프를 시각화하기 위해 함수를 추가하였다. 시각화 또한 다양한 방법으로 진행하고자 하였다.

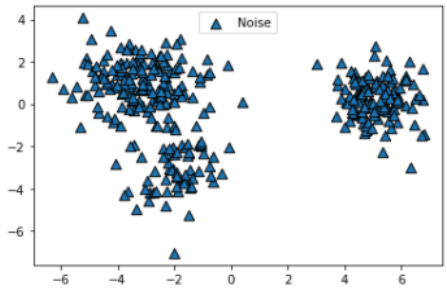
```
from sklearn.cluster import DBSCAN
import numpy as np

df_dbscan = pd.DataFrame(data=x, columns=['component1', 'component2'])
df_dbscan['target'] = penguins.speciesEnc

dbscan= DBSCAN(eps=0.5, min_samples=5)
dbscan_label = dbscan.fit_predict(data_lda)
df_dbscan['dbscan_label'] = dbscan_label

print(df_dbscan.groupby('target')['dbscan_label'].value_counts())
```

```
target  dbscan_label
0.0      0              123
        -1               23
         1                5
         3                1
1.0      2               49
        -1               13
         3                5
         0                1
2.0      4              107
        -1               15
Name: dbscan_label, dtype: int64
```

```
X = data_lda
Y = df_dbscan['target']
dbscan = DBSCAN(eps=0.7, min_samples=10) #eps와 min_samples를 변경해가면서 보자!
dbscan_label = dbscan.fit_predict(X)
df_lda['dbscan_label']=dbscan_label

df_lda['component1'] = data_lda[:,0]
df_lda['component2'] = data_lda[:,1]
visualize_cluster_plot(dbscan, df_lda, 'dbscan_label', iscenter=False)
```

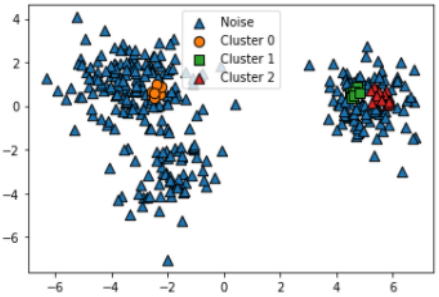
DBSCAN 을 진행해본 결과 eps 와 min_samples 파라미터의 세팅에 따라 결과가 많이 바뀌는 것을 확인할 수 있었다. eps 는 개별 데이터를 중심으로 입실론 반경을 의미하고, min_samples 는 최소 데이터 개수를 의미한다.

• eps:0.1, min_samples:100



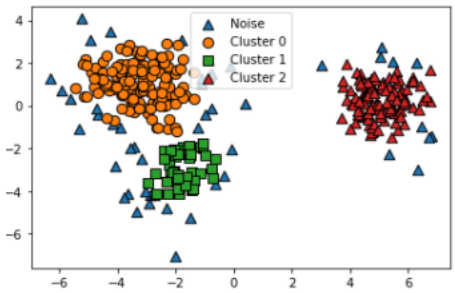
결과를 보면 모든 데이터가 Noise 로 인식된 것을 확인할 수 있다. 그래서 eps 를 0.3 으로 올려보았지만 결과는 똑같았다.

• eps:0.3, min_samples:10



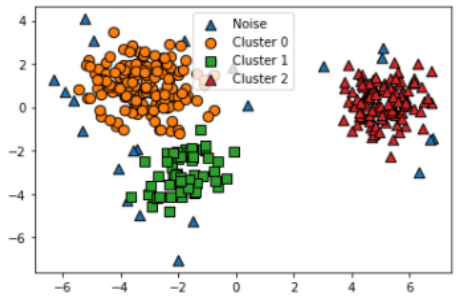
최소 데이터 개수인 min_samples 값을 낮추어 보았다. 결과를 보면 여전히 Noise 가 많지만 군집화가 조금 된 것을 확인할 수 있다. 입실론 반경이 너무 작아서 그런 것 같아 값을 더 키워보았다.

• eps:0.7, min_samples:10



결과를 보면 더 좋은 군집화가 이루어진 것을 확인할 수 있다.

• eps:0.865, min_samples:10



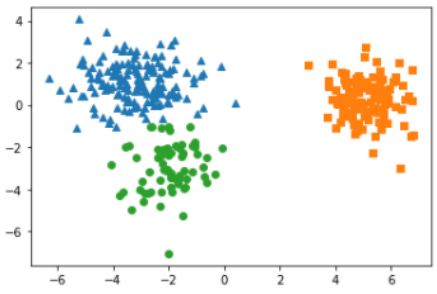
이와 같은 과정을 거쳐 적절한 파라미터를 선정한 결과 eps:0.865, min_samples=10 일 때 군집화가 잘 이루어 졌다.

```
matrix_new = pd.DataFrame({'labels': dbscan_label, 'species': y})
ct_new = pd.crosstab(matrix_new['labels'], matrix_new['species'])
print(ct_new)
```

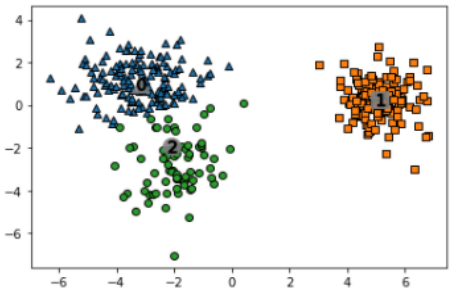
```
species  0.0  1.0  2.0
labels
-1         8    7    7
 0       144    6    0
 1         0   55    0
 2         0    0  115
```

실제 라벨의 값과 비교해본 결과 Noise 는 조금 있었고, 0 번 라벨(Adélie)은 조금의 오차가 있었지만, 1 번 라벨(Chinstrap)과 2 번 라벨(Gentoo)은 정확하게 군집화되었다.

• 결과 분석



〈KMeans〉



〈MeanShift〉

```
from sklearn.metrics import silhouette_samples, silhouette_score

score_samples = silhouette_samples(df_lda,df_lda['cluster'])
print(score_samples.shape)

df_lda['silhouette_coeff'] = score_samples
average_score = silhouette_score(df_lda, df_lda['cluster'])
print(average_score)
```

```
(343,)
0.688988319955076
```

```
df_lda.groupby('cluster')['silhouette_coeff'].mean()
```

```
cluster
0    0.615532
1    0.809981
2    0.640860
Name: silhouette_coeff, dtype: float64
```

〈KMeans〉

```
from sklearn.metrics import silhouette_samples, silhouette_score
df['target'] = df['target'].fillna(0.0)
score_samples = silhouette_samples(df,df['target'])
print(score_samples.shape)

df['silhouette_coeff'] = score_samples
average_score = silhouette_score(df, df['target'])
print(average_score)
```

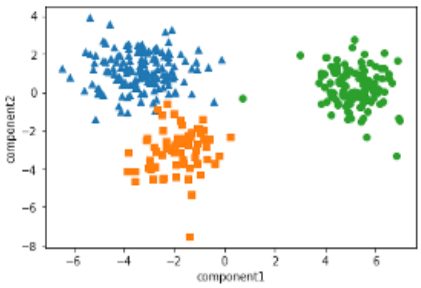
```
(343,)
0.673784047008536
```

```
df.groupby('target')['silhouette_coeff'].mean()
```

```
target
0.0    0.592092
1.0    0.633369
2.0    0.800237
Name: silhouette_coeff, dtype: float64
```

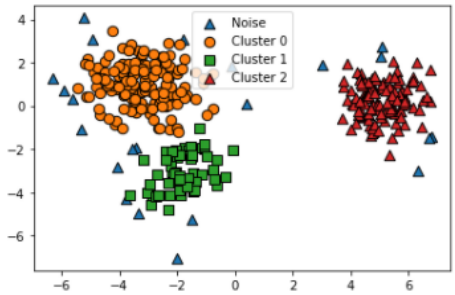
〈MeanShift〉

군집 평가를 보면 KMeans 알고리즘이 더 좋은 군집화를 보여주고 있음을 알 수 있다. 군집화 된 그래프를 비교해보면 서로 다른 라벨의 데이터가 가까이에 붙어있을 때 KMeans 알고리즘이 정확도가 더 높다. 따라서 이 때문에 더 좋은 군집 평가 수치가 나왔음을 확인할 수 있다. 그 이유는 각 군집에 속한 데이터들의 평균으로 중심점을 갱신하는 방법과, 밀도가 가장 높은 곳으로 중심을 이동하는 방법의 차이에 있을 것이다. kmeans 보다 성능은 조금 떨어졌지만 MeanShift 는 cluster 의 개수를 주어주지 않아도 군집화가 잘 이루어진다는 점에서 cluster 의 개수를 모를 때 사용하기 용이할 것 같다. 또한 MeanShift 는 bandwidth 가 큰 영향을 주어 값이 커질수록 군집의 개수가 감소하고, 작을수록 군집의 개수가 증가하는 것을 확인할 수 있었다.



〈GMM〉

GMM 은 데이터를 여러 개의 Gaussian 분포의 데이터 집합이 섞여서 생성된 데이터라고 가정하기 때문에 사용하는 데이터가 Gaussian 분포일수록 좋은 결과를 얻을 것으로 예상된다. . GMM Clustering 군집 개수(K)를 정하고 데이터 포인터들이 속하는 Cluster 를 확률적으로 할당하는 Soft Clustering 알고리즘이므로 군집 개수를 알 때 사용하기 용이할 것 같다.



〈DBSCAN〉

DBSCAN 은 eps 와 min_samples 파라미터의 세팅에 따라 결과가 많이 바뀌는 것을 확인할 수 있었다. Meanshift 와 마찬가지로 cluster 의 개수를 정해주지 않아도 되므로 cluster 의 개수를 모를 때 사용하기 좋을 것 같다. 또한 noise data 를 분류할 수 있기 때문에 outlier 에 의해 clustering 성능이 하락하는 현상을 완화할 수 있다. 하지만 데이터의 특성을 잘 모를 때에는 값을 바꿔가며 확인해야하기 때문에 적절한 hyper parameter 설정이 어려울 것이라고 생각된다.

4) 고찰

이번 텀프로젝트를 통해 여러가지 군집화 모델을 모델링 해보면서 그 방법과 그 특성에 대해 공부할 수 있었다. 여러가지 분류 모델을 모델링 해보면서 그 방법과 그 특성에 대해 공부할 수 있었다. 또한 원하는 목적에 맞게 dataset 을 선별하는 방법과 찾은 dataset 을 전처리하는 방법에 대해서도 처음부터 끝까지 직접 해보며 습득하였다 자신이 사용하려는 알고리즘과 dataset 에 따라 여러가지 시도와 실패를 통해 각 파라미터의 특징과 튜닝 방법을 익혔다.

적은 피처를 사용하여 군집화하면, 다른 라벨과 군집이 되는 안 좋은 군집화가 이루어졌다. 이는 피처가 너무 적어 그 두 개의 피처로의 overfitting 이 되어서 그렇다고 생각한다. 그리고 많은 피처를 차원 축소 없이 군집화 하여 2 차원으로 그래프를 그리면 성능이 매우 좋지 않았음을 확인할 수 있었다. 따라서 여러 개의 피처를 사용할 때에는 차원 축소가 필요하다는 것을 확인할 수 있었다.

차원 축소의 방법으로 LDA 를 사용했는데, LDA 는 지도학습으로, 정답인 data 가 있을 때에만 사용할 수 있었다. LDA 를 사용하여 차원축소를 하였더니 축소가 잘 된 것을 확인할 수 있었다. 이후 KMeans 를 통해 군집화를 하면 좋은 군집화가 이루어졌다.

MeanShift 는 cluster 의 개수를 주어주지 않아도 군집화가 잘 이루어진다는 점에서 cluster 의 개수를 모를 때 사용하기 용이할 것 같다. 또한 meanshift 는 bandwidth 가 큰 영향을 주어 값이 커질수록 군집의 개수가 감소하고, 작을수록 군집의 개수가 증가하는 것을 확인할 수 있었다.

GMM 은 데이터를 여러 개의 Gaussian 분포의 데이터 집합이 섞여서 생성된 데이터라고 가정하기 때문에 사용하는 데이터가 Gaussian 분포일수록 좋은 결과를 얻을 것으로 예상된다. . GMM Clustering 군집 개수(K)를 정하고 데이터 포인터들이 속하는 Cluster 를 확률적으로 할당하는 Soft Clustering 알고리즘이므로 군집 개수를 알 때 사용하기 용이할 것 같다.

DBSCAN 은 eps 와 min_samples 파라미터의 세팅에 따라 결과가 많이 바뀌는 것을 확인할 수 있었다. Meanshift 와 마찬가지로 cluster 의 개수를 정해주지 않아도 되므로 cluster 의 개수를 모를 때 사용하기 좋을 것 같다. 또한 noise data 를 분류할 수 있기 때문에 outlier 에 의해 clustering 성능이 하락하는 현상을 완화할 수 있다. 하지만 데이터의 특성을 잘 모를 때에는 값을 바꿔가며 확인해야하기 때문에 적절한 hyper parameter 설정이 어려울 것이라고 생각된다.

따라서 선택한 데이터의 특성에 따라 전처리를 진행하고, 전처리한 데이터에 맞는 적절한 군집화 방법을 선택하고 파라미터를 설정해주는 것이 중요하다는 것을 알 수 있었던 프로젝트였다.