기계학습의 원리와 응용 기말과제

2022021585 이재윤

Department of Computer Science and Engineering, Korea University

IDS508: 기계학습의원리와응용

1. SVM으로 multi-class 분류 문제를 해결하기

이미 구현된 머신러닝 라이브러리를 사용하지 않기 위해 SVM 클래스를 따로 정의하였다. 구현한 클래스의 멤버 변수와 함수는 다음과 같다.

```python
import numpy as np

class SVM:

    def __init__(self, learning_rate=1e-3, lambda_parameter=1e-2, n_iters=1000):
        self.lr = learning_rate
        self.lp = lambda_parameter
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def _init_weights_bias(self, X):
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        self.b = 0

    def _get_cls_map(self, y):
        return np.where(y <= 0, -1, 1)

    def _satisfy_constraint(self, x, index):
        linear_model = np.dot(x, self.w) + self.b
        return self.cls_map[index] * linear_model >= 1

    def _get_gradients(self, constrain, x, index):
        if constrain:
            dw = self.lp * self.w
            db = 0
            return dw, db
        dw = self.lp * self.w - np.dot(self.cls_map[index], x)
        db = - self.cls_map[index]
        return dw, db

```

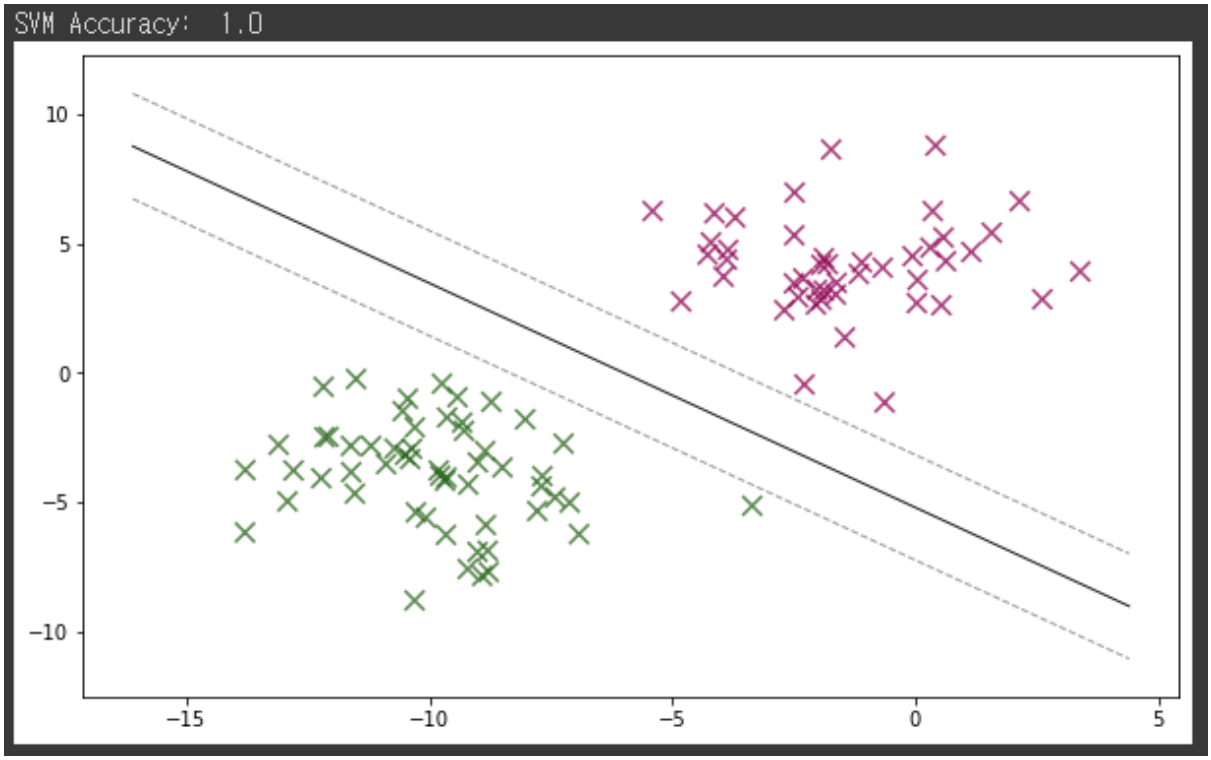
```
    def _update_weights_bias(self, dw, db):
        self.w -= self.lr * dw
        self.b -= self.lr * db

    def fit(self, X, y):
        self._init_weights_bias(X)
        self.cls_map = self._get_cls_map(y)
        for _ in range(self.n_iters):
            for index, x in enumerate(X):
                constrain = self._satisfy_constraint(x, index)
                dw, db = self._get_gradients(constrain, x, index)
                self._update_weights_bias(dw, db)

    def predict(self, X):
        estimate = np.dot(X, self.w) + self.b
        prediction = np.sign(estimate)
        return np.where(prediction == -1, 0, 1)
```

우선 hyperparameter로 learning rate와 lambda 값, 그리고 iteration 횟수를 설정하였다. Hyperplane에 의해 쉽게 데이터를 구별할 수 있도록 class의 label을 (0, 1)에서 (−1, 1)로 옮겨주었으며 gradient descent를 적용할 수 있도록 작성하였다. Prediction 과정에서는 (−1, 1)로 옮겨주었던 class label을 다시 (0, 1)로 옮겨 올바르게 분류될 수 있도록 만들었다. 성능 평가를 위해 SVM으로 분류하기 위한 현실의 데이터셋을 찾아봤으나 명확하게 구분되는 데이터를 좀처럼 찾아보기 어려워서 직접 명확히 구분되도록 임의의 두 군집으로 나타나는 데이터를 생성해주었다. 이 과정에서 데이터의 생성과 분할을 위해 scikit−learn 라이브러리가 사용되었다.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = datasets.make_blobs(
    n_samples=500, n_features=2, centers=2, cluster_std=2, random_state=1
)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)


clf = SVM(n_iters=1000)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)

```

```
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true==y_pred) / len(y_true)
    return accuracy

print("SVM Accuracy: ", accuracy(y_test, predictions))

def get_hyperplane(x, w, b, offset):
    return (-w[0] * x - b + offset) / w[1]

fig, ax = plt.subplots(1, 1, figsize=(10,6))

plt.scatter(X_test[:, 0], X_test[:, 1], marker="x", c=y_test, s=100, alpha=0.75)

x0_1 = np.amin(X_train[:, 0])
x0_2 = np.amax(X_train[:, 0])

x1_1 = get_hyperplane(x0_1, clf.w, clf.b, 0)
x1_2 = get_hyperplane(x0_2, clf.w, clf.b, 0)

x1_1_m = get_hyperplane(x0_1, clf.w, clf.b, -1)
x1_2_m = get_hyperplane(x0_2, clf.w, clf.b, -1)

x1_1_p = get_hyperplane(x0_1, clf.w, clf.b, 1)
x1_2_p = get_hyperplane(x0_2, clf.w, clf.b, 1)

ax.plot([x0_1, x0_2], [x1_1, x1_2], "-", c='k', lw=1, alpha=0.9)
ax.plot([x0_1, x0_2], [x1_1_m, x1_2_m], "--", c='grey', lw=1, alpha=0.8)
ax.plot([x0_1, x0_2], [x1_1_p, x1_2_p], "--", c='grey', lw=1, alpha=0.8)

x1_min = np.amin(X[:, 1])
x1_max = np.amax(X[:, 1])
ax.set_ylim([x1_min - 3, x1_max + 3])

plt.show()
```

500개의 샘플 데이터 중 400개를 훈련에 사용하고 100개를 테스트 데이터셋으로 사용하였으며 모델의 iteration 횟수는 1000번으로 설정하였다. 테스트 데이터는 x표시로 시각화 하였으며 구해진 hyperplane은 검은색 실선으로, 위아래로 오프셋을 적용한 hyperplane은 회색 점선으로 시각화 하였다. 그 결과는 다음과 같다.



2. Ensemble 방식으로 Random Forest를 구현하기

우선 강의자료에서 사용된 배드민턴 데이터를 csv파일 형태로 정리한 후 학습에 사용하였다.

```
[ ]  1 import numpy as np
     2 import pandas as pd
     3 from google.colab import files
```

```
     1 badminton = files.upload()
```

```
파일 선택  선택된 파일 없음        Upload widget is on
Saving 배드민턴.csv to 배드민턴.csv
```

```
[ ]  1 df = pd.read_csv("배드민턴.csv", encoding='cp949')
     2 print(df)
```

```
   날씨  바람  온도  습도 배드민턴
0  맑음  강함  낮음  보통  아니오
1  흐림  강함  높음  보통  아니오
2  맑음  약함  높음  보통    네
3   비   약함  보통  높음    네
4  맑음  약함  보통  낮음    네
5  흐림  약함  낮음  높음  아니오
6  흐림  약함  보통  높음  아니오
7  맑음  강함  높음  낮음  아니오
```

하지만 이대로 사용할 경우 데이터의 내용이 문자로 이루어져있어 학습이 어려우므로 문자로 된 데이터를 전부 숫자로 변환해주었다.

```
mapping_dict_weather = {'맑음' : 1,
                        '흐림' : 2,
                        '비' : 3}
mapping_dict_wind = {'약함' : 1,
                     '강함' : 2}
mapping_dict_temperature = {'낮음' : 1,
                            '보통' : 2,
                            '높음' : 3}
mapping_dict_humidity = {'낮음' : 1,
                         '보통' : 2,
                         '높음' : 3}
mapping_dict_badminton = {'아니오' : 0,
                          '네' : 1}
```

```
df['날씨'] = df['날씨'].apply(lambda x : mapping_dict_weather[x])
df['바람'] = df['바람'].apply(lambda x : mapping_dict_wind[x])
df['온도'] = df['온도'].apply(lambda x : mapping_dict_temperature[x])
df['습도'] = df['습도'].apply(lambda x : mapping_dict_humidity[x])
df['배드민턴'] = df['배드민턴'].apply(lambda x : mapping_dict_badminton[x])
```

변환시켜준 후의 데이터셋은 다음과 같다.

```
df
```

	날씨	바람	온도	습도	배드민턴
0	1	2	1	2	0
1	2	2	3	2	0
2	1	1	3	2	1
3	3	1	2	3	1
4	1	1	2	1	1
5	2	1	1	3	0
6	2	1	2	3	0
7	1	2	3	1	0

해당 데이터셋을 학습하기 위해 X로 날씨, 바람, 온도, 습도 값을 사용하고 Y로 배드민턴 값을 사용하여 numpy 배열 형태로 변환시켜주었다.

```
[ ]   1 X = df[["날씨", "바람", "온도", "습도"]]
      2 X = X.to_numpy()
      3 y = df["배드민턴"]
      4 y = y.to_numpy()
      5
      6 print(X, y)
```

```
[[1 2 1 2]
 [2 2 3 2]
 [1 1 3 2]
 [3 1 2 3]
 [1 1 2 1]
 [2 1 1 3]
 [2 1 2 3]
 [1 2 3 1]] [0 0 1 1 1 0 0 0]
```

이제 전처리가 완료된 해당 데이터를 training data와 test data로 나누어 Random Forest 방식으로 학습해주었다.

```
[ ]   1 from sklearn.model_selection import train_test_split
      2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state=42)
```

```
[ ]   1 from sklearn.ensemble import RandomForestClassifier
      2 from sklearn.metrics import accuracy_score
      3
      4 for i in range(1,10):
      5     for j in range(1,10):
      6         clf = RandomForestClassifier(n_estimators=i, max_depth=j, random_state=42)
      7         clf.fit(X_train, y_train)
      8
      9         predict = clf.predict(X)
     10         print("n_estimators: ", i, "max_depth: ", j)
     11         print(accuracy_score(y,predict))
     12
     13     print("========================================")
     14
     15 clf = RandomForestClassifier(n_estimators=3, max_depth=3, random_state=42)
     16 clf.fit(X_train, y_train)
     17
     18 predict = clf.predict(X_test)
     19 print(accuracy_score(y_test,predict))
```

이 과정에서 적절한 estimator 값과 max depth 값을 찾기 위해 각각의 값을 1에서 9까지 바꿔가면서 정확도를 확인해보았다.

```
n_estimators:  1 max_depth:  1
0.75
n_estimators:  1 max_depth:  2
0.75
n_estimators:  1 max_depth:  3
0.75
n_estimators:  1 max_depth:  4
0.75
n_estimators:  1 max_depth:  5
0.75
n_estimators:  1 max_depth:  6
0.75
n_estimators:  1 max_depth:  7
0.75
n_estimators:  1 max_depth:  8
0.75
n_estimators:  1 max_depth:  9
0.75
====================================
n_estimators:  9 max_depth:  1
1.0
n_estimators:  9 max_depth:  2
1.0
n_estimators:  9 max_depth:  3
1.0
n_estimators:  9 max_depth:  4
1.0
n_estimators:  9 max_depth:  5
1.0
n_estimators:  9 max_depth:  6
1.0
n_estimators:  9 max_depth:  7
1.0
n_estimators:  9 max_depth:  8
1.0
n_estimators:  9 max_depth:  9
1.0
====================================
1.0
```

결과값의 출력이 많아 중간의 결과값은 생략하였다. 데이터셋의 크기가 작아서 정확도를 측정하는 것이 오차가 클 수 있으나 해당 출력값을 통해 대체로 n_estimators 값이 3 이상이면 1.0에 가까운 정확도를 보이는 것을 확인할 수 있었다.

동일한 데이터셋에 대해 Decision Tree 방식을 적용한 경우도 알아보았다.

```python
from sklearn.tree import DecisionTreeClassifier

for i in range(1,10):
    DTclf = DecisionTreeClassifier(min_samples_split=2, max_depth=i, random_state=42)
    DTclf.fit(X_train, y_train)

    predict = DTclf.predict(X)
    print("max_depth: ", i)
    print(accuracy_score(y,predict))

    print("========================================")

DTclf = DecisionTreeClassifier(min_samples_split=2, max_depth=3, random_state=42)
DTclf = DTclf.fit(X_train, y_train)

predict = DTclf.predict(X_test)
print(accuracy_score(y_test,predict))
```

Decision Tree 방식에서는 max_depth 값만 변화시켜가며 정확도를 확인해보았다. 그 결과는 다음과 같다.

```
max_depth:  1
0.75
========================================
max_depth:  2
0.875
========================================
max_depth:  3
1.0
========================================
max_depth:  4
1.0
========================================
max_depth:  5
1.0
========================================
max_depth:  6
1.0
========================================
max_depth:  7
1.0
========================================
max_depth:  8
1.0
========================================
max_depth:  9
1.0
========================================
1.0
```

Decision Tree 모델의 경우에도 max_depth 값이 3이상이면 충분히 좋은 정확도를 보이는 것을 확인할 수 있었다.

```
[ ]  1 clf = RandomForestClassifier()
     2 clf.fit(X_train, y_train)
     3
     4 predict = clf.predict(X_test)
     5 print("Random Forest: ", accuracy_score(y_test,predict))
     6
     7 DTclf = DecisionTreeClassifier()
     8 DTclf = DTclf.fit(X_train, y_train)
     9
    10 predict = DTclf.predict(X_test)
    11 print("Decision Tree: ", accuracy_score(y_test,predict))

    Random Forest:  1.0
    Decision Tree:  1.0
```

Hyperparameter를 바꾸지 않고 기본값으로 설정하여 학습한 경우에도 두 모델 모두 1.0의 정확도를 보이는 것을 확인할 수 있었다.

3. Ensemble 방식으로 SVM과 ANN을 활용하여 Adaboost를 구현하기

Adaboost를 구현하기 위해 사용한 데이터셋으로는 고객들이 특정 상품을 구매했는지 안했는지를 분류하는 데이터셋을 선정하였다. 해당 데이터셋은 고객의 ID와 성별, 나이, 예상 수입, 그리고 상품 구매 여부로 이루어져있다. 자세한 내용과 코드는 다음과 같다.

```
1 import numpy as np
2 import pandas as pd
3 from google.colab import files
4
5 uploaded = files.upload()
```

```
파일 선택  선택된 파일 없음        Upload widget is only availabl
Saving social.csv to social.csv
```

```
[ ]  1 df = pd.read_csv('social.csv')
     2 df
```

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
...	...	...	...	...	...
395	15691863	Female	46	41000	1
396	15706071	Male	51	23000	1
397	15654296	Female	50	20000	1
398	15755018	Male	36	33000	0
399	15594041	Female	49	36000	1

400 rows × 5 columns

여기서 고객의 ID는 상품 구매 여부와 연관이 없으므로 삭제하고, 성별은 문자로 이루어져 있으므로 숫자로 바꾸어 학습에 사용하였다.

```
1 mapping_dict = {'Male' : 1,
2                 'Female' : 2}
3
4 df['Gender'] = df['Gender'].apply(lambda x : mapping_dict[x])
5 df
```

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	1	19	19000	0
1	15810944	1	35	20000	0
2	15668575	2	26	43000	0
3	15603246	2	27	57000	0
4	15804002	1	19	76000	0
...	...	...	...	...	...
395	15691863	2	46	41000	1
396	15706071	1	51	23000	1
397	15654296	2	50	20000	1
398	15755018	1	36	33000	0
399	15594041	2	49	36000	1

400 rows × 5 columns

```
[ ] 1 X = df[['Gender', 'Age', 'EstimatedSalary']]
    2 y = df['Purchased']
```

SVM과 Adaboost는 scikit-learn 라이브러리를 활용하여 구현하였다.

```
[ ] 1 from sklearn.model_selection import train_test_split
    2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
[ ] 1 from sklearn.ensemble import AdaBoostClassifier
    2 from sklearn.svm import SVC
    3 from sklearn.metrics import accuracy_score
    4
    5 SVM = SVC()
    6 SVMclf = AdaBoostClassifier(base_estimator = SVM, algorithm="SAMME")
    7 SVMclf.fit(X_train, y_train)
    8 SVMpred = SVMclf.predict(X_test)
    9
   10 print("SVM adaboost accuracy: ", accuracy_score(y_test, SVMpred))
```

```
SVM adaboost accuracy:  0.675
```

그 결과 SVM을 활용한 Adaboost 모델은 0.675의 정확도를 보여주었다. ANN을 활용한 Adaboost도 scikit-learn 라이브러리의 인공신경망에 해당하는 MLPClassifier를 활용하여 구현하고 싶었으나 MLPClassifier는 base_estimator 값으로 설정이 불가능하여 SVM과 동일한 방식으로는 구현할 수 없었다. 따라서 base_estimator 값으로 적용 가능한 인공신경망을 찾아본 결과 hep_ml 라이브러리가 이를 지원한다는 사실을 알아냈다. 해당 라이브러리를 설치하고 ANN 모델을 생성하여 Adaboost에 적용한 결과 다음과 같은 결과값을 얻을 수 있었다.

```
[ ]  1 pip install hep_ml
```

```
[ ]  1 from hep_ml.nnet import MLPClassifier
     2
     3 ANN = MLPClassifier()
     4 ANNclf = AdaBoostClassifier(base_estimator = ANN)
     5 ANNclf.fit(X_train, y_train)
     6 ANNpred = ANNclf.predict(X_test)
     7
     8 print("ANN adaboost accuracy: ", accuracy_score(y_test, ANNpred))

    ANN adaboost accuracy:  0.875
```

ANN을 활용한 Adaboost 모델은 0.875의 정확도를 보여 SVM을 활용한 Adaboost 모델보다 상대적으로 더 우수한 정확도를 보였다.

4. 분류 문제를 정의하고 PCA와 LDA 방식을 적용하여 분류한 결과를 비교하기

차원 축소를 통해 데이터를 구분하기 위해 펭귄 종류를 분류하는 데이터셋을 선정하였다. 해당 데이터셋은 펭귄의 종류, 부리의 길이와 깊이, 지느러미의 길이와 몸무게로 이루어져 있으며 펭귄의 특성을 통해 해당 펭귄의 종류를 알아내는 분류 문제를 PCA와 LDA를 통해 해결할 것이다. 우선 데이터의 전처리 과정에 대한 코드는 다음과 같다.

```
1 import numpy as np
2 import pandas as pd
3 from google.colab import files
4
5 uploaded = files.upload()
```

파일 선택 선택된 파일 없음 Upload widget is only available when the

```
Saving penguins.csv to penguins.csv
```

```
1 penguins = pd.read_csv('penguins.csv')
2 penguins
```

	Species	BeakLength	BeakDepth	FlipperLength	BodyMass
0	Adelie	39.1	18.7	181	3750
1	Adelie	39.5	17.4	186	3800
2	Adelie	40.3	18.0	195	3250
3	Adelie	36.7	19.3	193	3450
4	Adelie	39.3	20.6	190	3650
...	...	...	...	...	...
337	Gentoo	47.2	13.7	214	4925
338	Gentoo	46.8	14.3	215	4850
339	Gentoo	50.4	15.7	222	5750
340	Gentoo	45.2	14.8	212	5200
341	Gentoo	49.9	16.1	213	5400

342 rows × 5 columns

```
[ ] 1 X = penguins[['BeakLength', 'BeakDepth', 'FlipperLength', 'BodyMass']]
    2 y = penguins['Species']
```

```
[ ] 1 from sklearn.model_selection import train_test_split
    2 X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
[ ] 1 from sklearn.preprocessing import StandardScaler
    2 std_scale = StandardScaler()
    3 std_scale.fit(X_train)
    4 X_tn_std = std_scale.transform(X_train)
    5 X_te_std = std_scale.transform(X_test)
```

데이터에 PCA를 적용하여 4개의 특성(Beak Length, Beak Depth, Flipper Length, Body Mass)으로 이루어진 4차원 데이터를 2차원으로 축소하였다. 그 과정에 대한 코드는 다음과 같다.

```
1 from sklearn.decomposition import PCA
2 pca = PCA(n_components=2)
3 pca.fit(X_tn_std)
4 X_tn_pca = pca.transform(X_tn_std)
5 X_te_pca = pca.transform(X_te_std)
```

```
1 print(X_tn_std.shape)
2 print(X_tn_pca.shape)
```

```
(256, 4)
(256, 2)
```

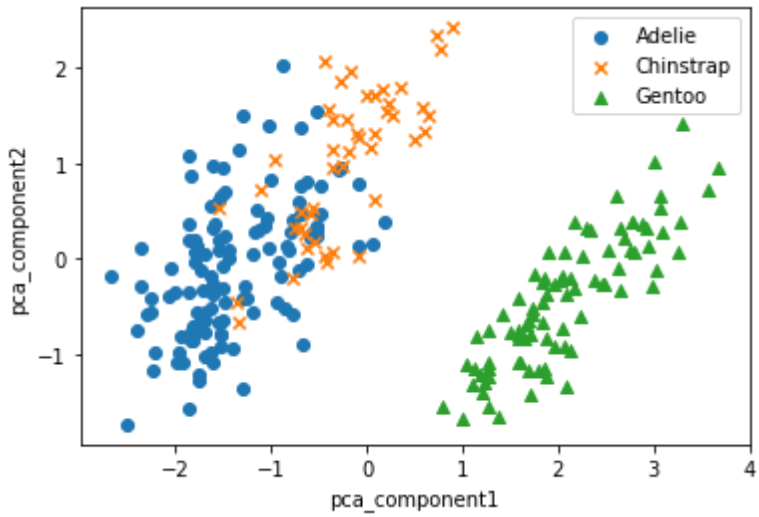
```
1 pca_columns = ['pca_comp1', 'pca_comp2']
2 X_tn_pca_df = pd.DataFrame(X_tn_pca,
3                            columns=pca_columns)
4 y_tn = y_train.to_numpy()
5 X_tn_pca_df['target'] = y_tn
6 X_tn_pca_df
```

	pca_comp1	pca_comp2	target
0	-2.059789	-0.395383	Adelie
1	-0.769367	-0.205328	Chinstrap
2	1.762191	-0.160041	Gentoo
3	-1.816328	-0.712135	Adelie
4	-1.538791	-0.674351	Adelie
...	...	...	...
251	2.472782	-0.256444	Gentoo
252	-0.773698	-0.574726	Adelie
253	-1.727952	-0.120788	Adelie
254	-0.748975	0.317885	Chinstrap
255	-2.012118	-0.342015	Adelie

256 rows × 3 columns

PCA 결과를 도표로 나타낸 결과 다음과 같은 결과를 얻을 수 있었다.

```
import matplotlib.pyplot as plt

pca_df = X_tn_pca_df
df_0 = pca_df[pca_df['target']=='Adelie']
df_1 = pca_df[pca_df['target']=='Chinstrap']
df_2 = pca_df[pca_df['target']=='Gentoo']

X_11 = df_0['pca_comp1']
X_12 = df_1['pca_comp1']
X_13 = df_2['pca_comp1']

X_21 = df_0['pca_comp2']
X_22 = df_1['pca_comp2']
X_23 = df_2['pca_comp2']

plt.scatter(X_11, X_21,
            marker='o',
            label='Adelie')
plt.scatter(X_12, X_22,
            marker='x',
            label='Chinstrap')
plt.scatter(X_13, X_23,
            marker='^',
            label='Gentoo')
plt.xlabel('pca_component1')
plt.ylabel('pca_component2')
plt.legend()
plt.show()
```



Adelie, Chinstrap, Gentoo 각기 다른 종류의 펭귄이 잘 분류된 것을 확인할 수 있다.

PCA를 통해 축소된 데이터를 Random Forest 분류 모델에 넣어 학습한 결과 다음과 같은 결과를 보였다.

```
[ ]  1 from sklearn.ensemble import RandomForestClassifier
     2
     3 clf_rf_pca = RandomForestClassifier(max_depth=2,
     4                                     random_state=0)
     5 clf_rf_pca.fit(X_tn_pca, y_tn)
     6
     7 pred_rf_pca = clf_rf_pca.predict(X_te_pca)
```

```
[ ]  1 from sklearn.metrics import accuracy_score
     2
     3 y_te = y_test.to_numpy()
     4 accuracy_pca = accuracy_score(y_te, pred_rf_pca)
     5 print(accuracy_pca)
```

```
0.8372093023255814
```

그 결과 약 0.8372의 정확도를 보이는 것을 확인할 수 있었다.

PCA와 동일한 방식으로 LDA를 통한 차원 축소도 실행해보았다.

```
[ ]  1 from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
     2
     3 y_tn = y_train.to_numpy()
     4 lda = LinearDiscriminantAnalysis()
     5 lda.fit(X_tn_std, y_tn)
     6 X_tn_lda = lda.transform(X_tn_std)
     7 X_te_lda = lda.transform(X_te_std)
```

```
[ ]  1 print(X_tn_std.shape)
     2 print(X_tn_lda.shape)
```

```
(256, 4)
(256, 2)
```

LDA 역시 4차원의 데이터를 2차원으로 축소하였다.

```
[ ]  1 lda_columns = ['lda_comp1', 'lda_comp2']
     2 X_tn_lda_df = pd.DataFrame(X_tn_lda,
     3                            columns=lda_columns)
     4 X_tn_lda_df['target'] = y_tn
     5 X_tn_lda_df
```

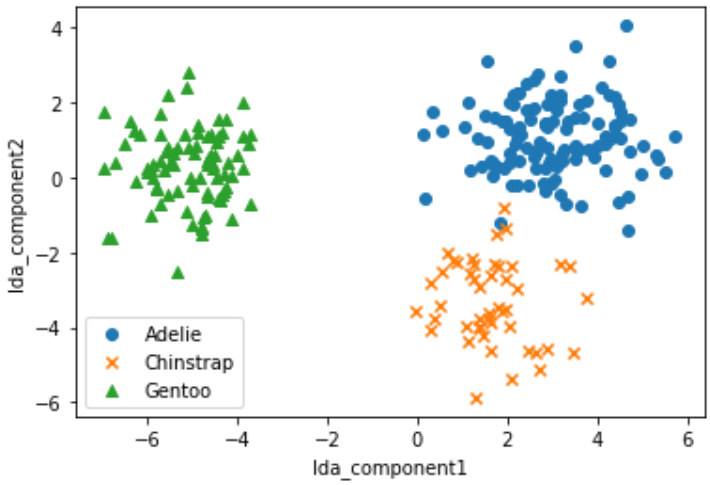
	lda_comp1	lda_comp2	target
0	4.250629	3.102804	Adelie
1	1.155025	-2.550611	Chinstrap
2	-4.008128	0.626154	Gentoo
3	3.185424	2.233571	Adelie
4	2.671447	2.767240	Adelie
...	...	...	...
251	-5.725227	1.134088	Gentoo
252	1.046404	1.343899	Adelie
253	3.759206	2.042073	Adelie
254	1.805610	-2.350223	Chinstrap
255	4.016153	1.475418	Adelie

256 rows × 3 columns

LDA 결과를 도표로 나타낸 결과 다음과 같은 결과를 얻을 수 있었다.

```
lda_df = X_tn_lda_df
df_0 = lda_df[lda_df['target']=='Adelie']
df_1 = lda_df[lda_df['target']=='Chinstrap']
df_2 = lda_df[lda_df['target']=='Gentoo']

X_11 = df_0['lda_comp1']
X_12 = df_1['lda_comp1']
X_13 = df_2['lda_comp1']

X_21 = df_0['lda_comp2']
X_22 = df_1['lda_comp2']
X_23 = df_2['lda_comp2']

plt.scatter(X_11, X_21,
            marker='o',
            label='Adelie')
plt.scatter(X_12, X_22,
            marker='x',
            label='Chinstrap')
plt.scatter(X_13, X_23,
            marker='^',
            label='Gentoo')
```

```
plt.xlabel('lda_component1')
plt.ylabel('lda_component2')
plt.legend()
plt.show()
```



LDA의 경우 역시 Adelie, Chinstrap, Gentoo 각기 다른 종류의 펭귄이 잘 분류된 것을 확인할 수 있다.

PCA와 마찬가지로 LDA를 통해 축소된 데이터를 Random Forest 분류 모델에 넣어 학습한 결과 다음과 같은 결과를 보였다.

```
[ ]  clf_rf_lda = RandomForestClassifier(max_depth=2,
                                         random_state=0)
     clf_rf_lda.fit(X_tn_lda, y_tn)

     pred_rf_lda = clf_rf_lda.predict(X_te_lda)
```

```
[ ]  accuracy_lda = accuracy_score(y_te, pred_rf_lda)
     print(accuracy_lda)
```

```
0.9534883720930233
```

그 결과 약 0.9535의 정확도를 보여 PCA로 학습한 경우보다 더 높은 정확도를 보이는 것을 확인할 수 있었다.