기계학습 7주차

ML 심화 : 와인 품질 예측

175530 박성혜

사용 데이터: 와인 품질 데이터

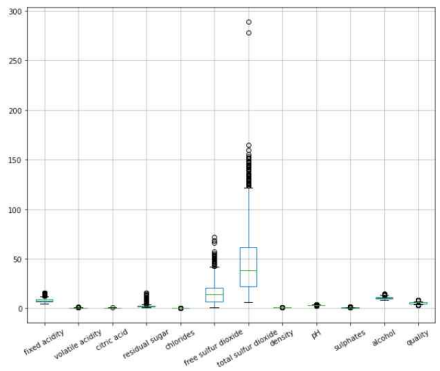
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

Wine.shape = (1599, 12) 데이터 샘플 1599개, 위의 특성 12개를 가진 데이터 셋이다.

```
np.unique(wine.quality)
```

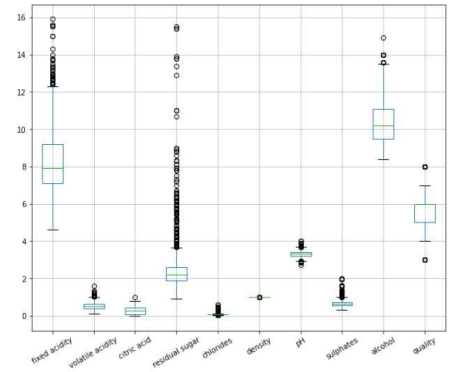
```
array([3, 4, 5, 6, 7, 8], dtype=int64)
```

quality가 와인의 품질을 나타내며 데이터에서는 3부터 8까지 수치가 나타난다.



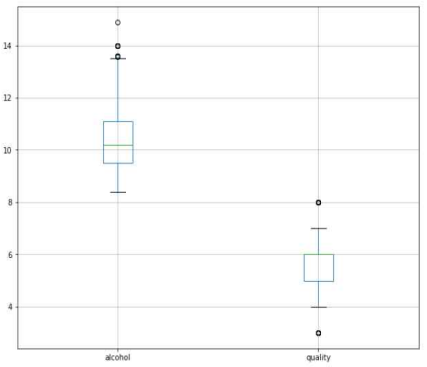
* 특성 전체를 박스플롯으로 출력한 결과

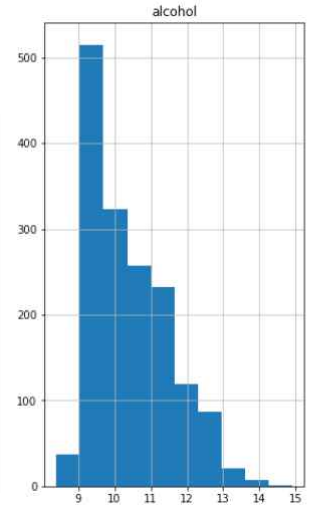
free sulfur dioxide, total sulfur dioxide 값이 가장 크게 나타나며 그 이상치 또한 크다. 이산화황의 기본 수치가 높은 것을 확인할 수 있다.

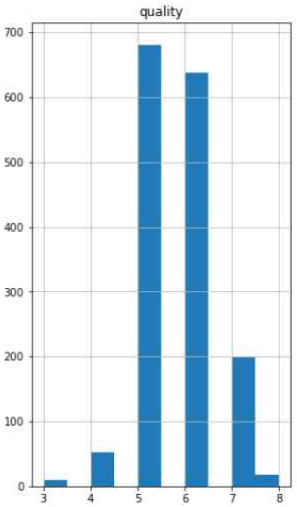


* 위의 이산화황을 제외한 특성을 다시 표현

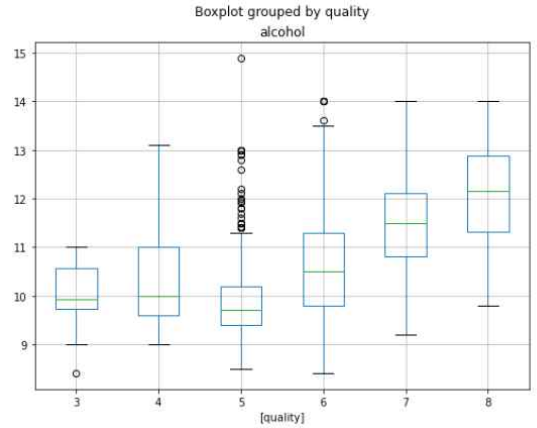
alcohol, fixed acidity, quality, ph, residual sugar ... 순으로 값의 크기가 나타났다.



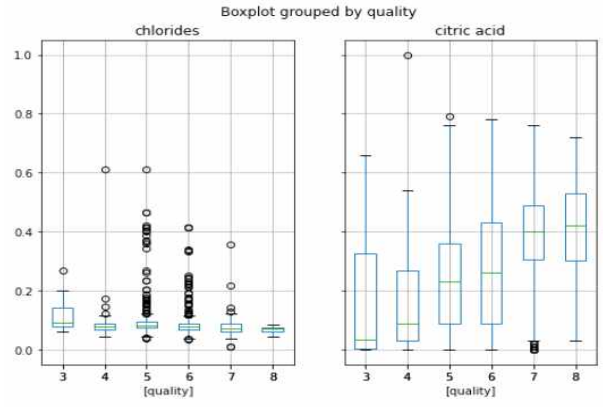




* 특성 alcohol과 quality 데이터
alcohol은 9~10 사이의 데이터가 가장 많으며 수치가 증가할수록 빈도수는 감소했다.
quality의 경우는 5와 6이 가장 많은 것을 확인할 수 있다.



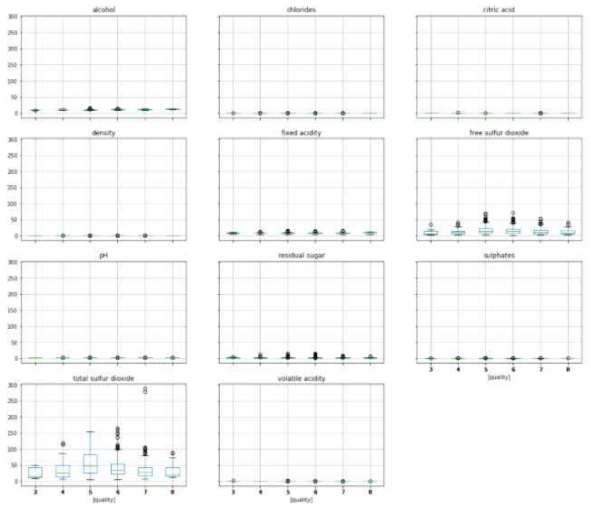
* alcohol을 quality 기준으로 표현
quality가 커질수록 alcohol 수치도 높아진다.

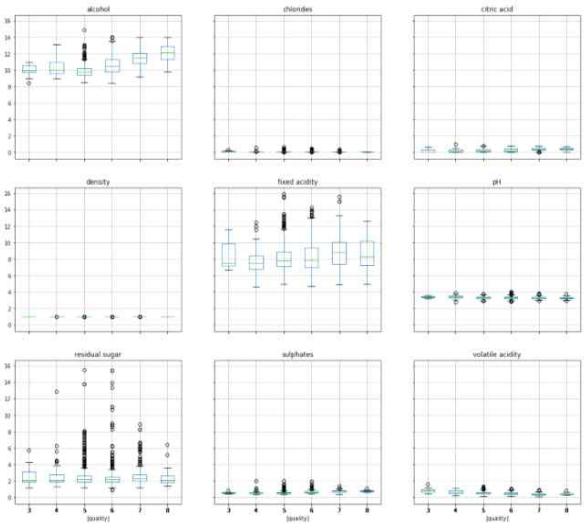


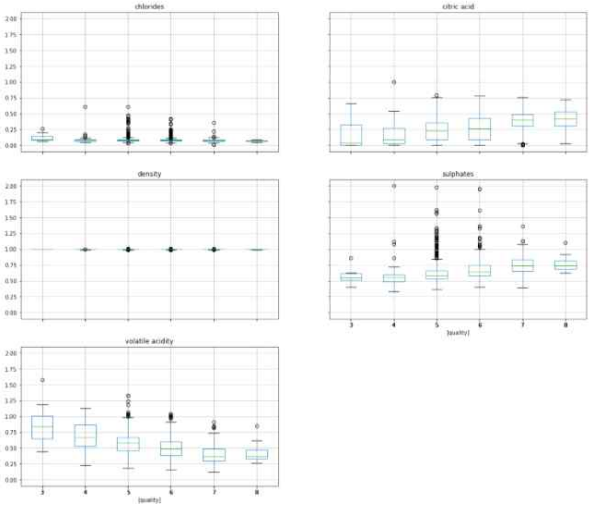
* 품질을 기준으로 chlorides와 citric acid

품질이 커질수록 시트르산도 커지고 염화물은 큰 차이가 없었다.

와인에서는 산도가 품질에 큰 영향을 미치며, 시트르산은 와인에 들어있는 산의 종류 중 하나다.







순서대로 품질 기준 전체 특성에 대한 box plot.
다음은 가장 큰 수치를 가졌던 이산화황 특성 2가지를 뺀 box plot.
마지막은 추가적으로 알코올, 결합 산도, 잔류 설탕을 뺀 5가지 특성을 표현한 것이다.

알코올, 결합 산도, 시트르산, 황산염은 품질이 커질수록 증가하며, 휘발성 산도는 감소한다.

free sulfur dioxide, total sulfur dioxide
alcohol, fixed acidity, residual sugar
critric acid, sulphates, volatile acidity
chlorides
density

```
np.min(wine.density)
```
```
0.99007
```
```
np.max(wine.density)
```
```
1.00369
```

순서로 품질에 따른 변화를 잘 관찰할 수 있으며 density는 데이터 스케일이 0.99~1.00.. 으로 매우 좁아서 품질에 따른 변화를 관찰하기 어렵다.

```
wine['quality'].value_counts()
```

```
5    681
6    638
7    199
4     53
8     18
3     10
```

품질 값에 따른 빈도수로 5~6이 가장 많고 7, 4, 8, 3순으로 나타났다.

quality	qual
5	0
5	0
5	0
6	0
5	0

```
wine['qual'].value_counts()
```

```
0    1382
1     217
Name: qual, dtype: int64
```

bins = (2.9, 6.5, 8.1)를 사용해서 6.5를 기준으로 quality를 bad(0), good(1)로 나눈다.
새로운 특성 qual 생성. 클래스0은 1382, 1은 217로 분류됐다.

기존 quality는 회귀용, qual은 분류용으로 이진 분류한다.

test_size = 0.2 / StandardScaler 적용.

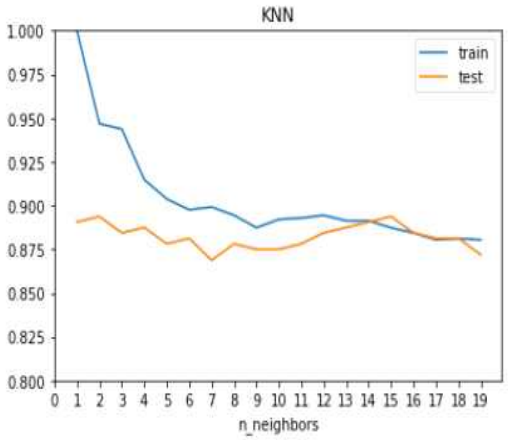
```
X_train.shape, y_train.shape
```

```
((1279, 11), (1279,))
```

```
X_test.shape, y_test.shape
```

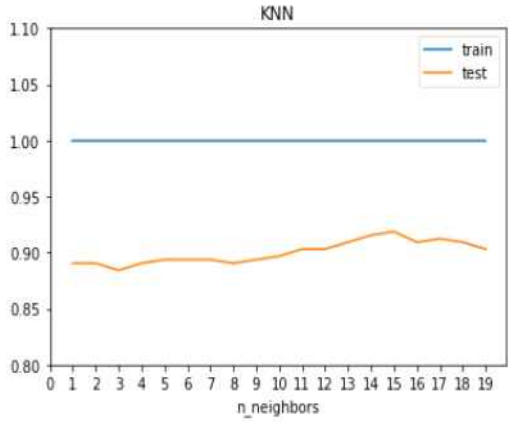
```
((320, 11), (320,))
```

훈련 데이터는 1279개. 테스트 데이터는 320



* KNN 최근접 이웃
이웃의 수에 따른 훈련, 테스트 데이터의 점수
1~9까지 이웃수를 사용하였다.

- 훈련 데이터는 1에서 과대적합 됐다가 이웃수가 증가하면서 완화됐다. 테스트 데이터는 비슷한 수준의 성능을 보인다.



weights='distance'로 가중치를 주어 위와 동일한 테스트를 진행했다.

- 1~20까지 n_neighbors에 대해 훈련 데이터가 모두 과대적합 됨을 확인할 수 있다. 테스트 데이터 정확도는 평균적으로 더 높아졌다. 클래스 데이터가 잘 모여 있어서 가까이 있는 영향을 받는데 그 정도가 큰 것으로 보인다.

```
rang =[0.01, 1, 100]

for r in rang:
    lr = LogisticRegression(C=r, max_iter=1000).fit(X_train,y_train)
    print("C=",r)
    print("Test =", lr.score(X_test, y_test))
    print("Train =",lr.score(X_train, y_train))
    print(confusion_matrix(y_test,lr.predict(X_test)))
```

```
C= 0.01
Test = 0.86875
Train = 0.8608287724784989
[[275   4]
 [ 38   3]]
C= 1
Test = 0.878125
Train = 0.8788115715402658
[[265  14]
 [ 25  16]]
C= 100
Test = 0.890625
Train = 0.8780297107114934
[[266  13]
 [ 22  19]]
```

* LogisticRegression

C값에 따른 결과

규제가 강화된 0.01의 경우 1번 클래스에 대해서 거의 제대로 분류하지 못했다. C가 커져서 규제가 약화될수록 1번 클래스에 대해 더 잘 분류가 됐다.

규제가 약화되면 패널티 영향력이 감소하고 다수의 포인터가 아닌 개개인의 포인터에 맞춰져서 성능이 개선된다. 0.01은 다수에(여기선 주로 0클래스)에 맞춰지기 때문에 1번에 대한 성능이 낮다.

```
classifier = LogisticRegression(max_iter=1000)
y_score = classifier.fit(X_train, y_train).decision_function(

#print(y_score)

print(classifier.score(X_test, y_test))
y_score = (y_score>0).astype(np.int)


print(confusion_matrix(y_test, classifier.predict(X_test)))
print(np.bincount(y_score))
```

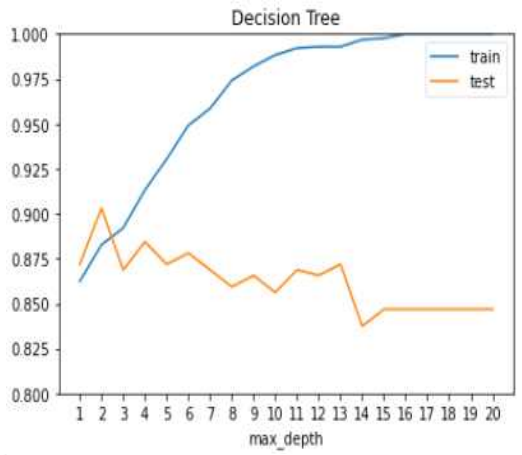
```
0.878125
[[266  13]
 [ 26  15]]
[292  28]
```

```
y_score = classifier.fit(X_train, y_train).predict_proba(X_test)
#predict_proba() 어떤 클래스라고 분류한 확률을 리턴해줌

#print(y_score)
y_label = []
for i in range(y_score.shape[0]):
    y_label.append((y_score[i][0] < y_score[i][1]).astype(np.int))

print(classifier.score(X_test, y_test))

print(confusion_matrix(y_test, classifier.predict(X_test)))
print(np.bincount(y_label))
```

```
0.878125
[[266  13]
 [ 26  15]]
[292  28]
```

decision_function과 predict_proba로 살펴본 결과.

0번 클래스를 1로 13개, 1번 클래스를 0으로 26개 잘 못 분류했다.

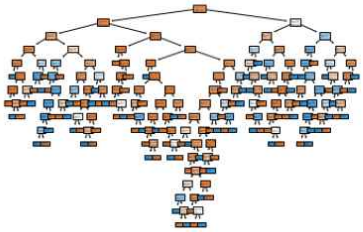


* DecisionTree(결정 트리)

max_depth에 따른 정확도를 보았다.

- 훈련 데이터는 깊어질수록 점수가 상승, 과대적합 된다. 테스트 데이터는 2에서 가장 좋은 성능을 보이며(이는 우연히 두 가지로 나눈 것이 잘 맞은 것으로 보임), 조금씩 변화의 폭이 있다가 16부터는 일정하게 유지된다.

```
dt = tree.DecisionTreeClassifier(random_state=0).fit(X_train, y_train)
plot_tree(dt, max_depth=16, filled=True)
plt.show()
```



깊이 16을 트리로 출력한 결과

plot_tree는 plot_tree내에서 max_depth를 조정하면, 다음 depth에서 생성될 수 있는 노드를 빈 박스로 보여주는데 왼쪽 트리를 보면 없다. 없는 것들은 전부 순수리프노드다.

```
dt = tree.DecisionTreeClassifier(max_depth = 15,random_state=0).fit(X_train, y_train)
print(dt.score(X_train, y_train))

dt = tree.DecisionTreeClassifier(max_depth = 16,random_state=0).fit(X_train, y_train)
print(dt.score(X_train, y_train))
```

```
0.9976544175136826
1.0
```

훈련 데이터 점수가 15에서는 0.99... 16에서 1로 고정된다. 이는 순수리프노드까지 훈련한 결과다.

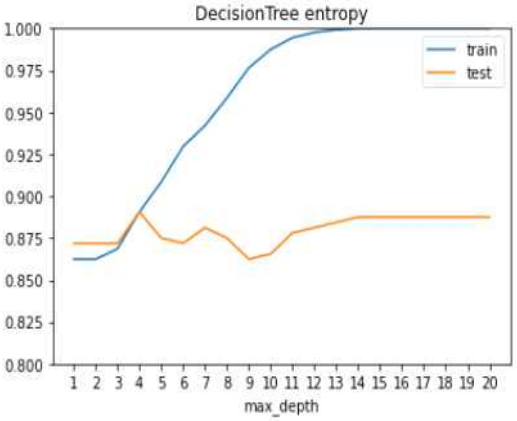
```
[199]: df = tree.DecisionTreeClassifier(max_depth=2,random_state=0).fit(X
       confusion_matrix( y_test, df.predict(X_test))
[199]: array([[271,   8],
              [ 23,  18]], dtype=int64)
[200]: df = tree.DecisionTreeClassifier(max_depth=16,random_state=0).fit(
       confusion_matrix( y_test, df.predict(X_test))
[200]: array([[246,  33],
              [ 16,  25]], dtype=int64)
```

혼돈 행렬 결과

결정 트리 역시, 1번 클래스(품질 6.5~8)을 잘 분류하지 못한다.

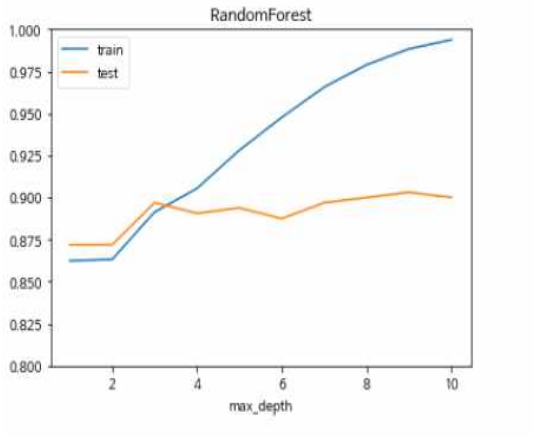
결정트리의 특성 중요도

- max_depth 2와 16에서의 결과로
  random_state=0에서 중요도가 가장
  높은 특성은 alcohol이다.



criterion='gini'에서 'entropy'로 손실함수를 바꾼 결과

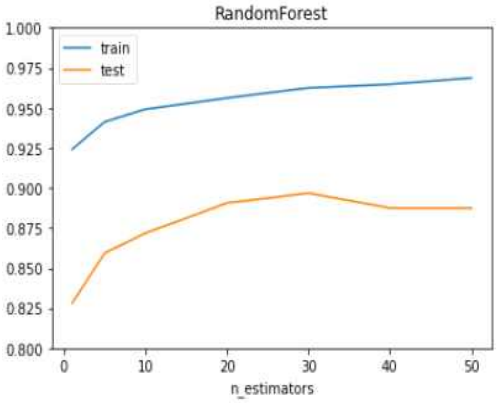
- gini보다 더 낮은 깊이에서 과대적합



* RandomForest

n_estimators에 따른 결과

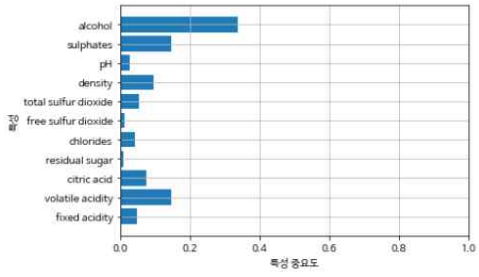
- estimator의 수가 늘어날수록 훈련, 테스트 세트 점수 모두 더 개선된다.



max_depth에 따른 결과

- 결정트리와 마찬가지로 깊이가 깊어질수록 훈련 데이터는 과대적합 된다. 테스트 데이터는 조금 증가하다가 비슷한 정확도를 가진다.

```
rf = RandomForestClassifier(max_depth=3, random_state=0).fit(X_train, y_train)
plot_feature_importances_wine(rf)
```



랜덤포레스트 분류기에서 특성 중요도. 결정트리와 마찬가지로 alcohol의 중요도가 가장 높다.

```
rfc = RandomForestClassifier(random_state=0).fit(X_train, y_train)

pred_rfc = rfc.predict(X_test)
confusion_matrix(y_test, pred_rfc)
# RandomForest에 대한 confusion_matrix
```

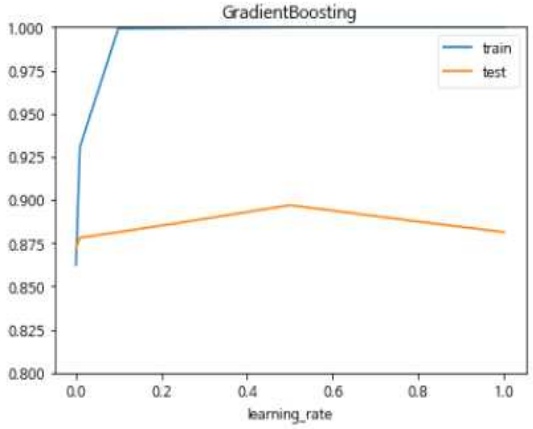
```
array([[265,  14],
       [ 17,  24]], dtype=int64)
```

랜덤포레스트 - 혼돈 행렬
확실히 1번 클래스에 대해서 잘 분류하지 못한다.

```
# 여러 성능 지표를 출력한다
print(classification_report(y_test, pred_rfc))
#밑에 숫자는 0을 기준으로 0.94 맞춤
#pos 긍정이 아닌 내가 찾고자 하는 답을 pos라고 봄(암환자)
```

	precision	recall	f1-score	support
0	0.94	0.95	0.94	279
1	0.62	0.59	0.60	41
accuracy			0.90	320
macro avg	0.78	0.77	0.77	320
weighted avg	0.90	0.90	0.90	320

랜덤포레스트 - report
precision과 recall 수치를 보아도 0번은 꽤 높지만 1번 클래스는 낮다.

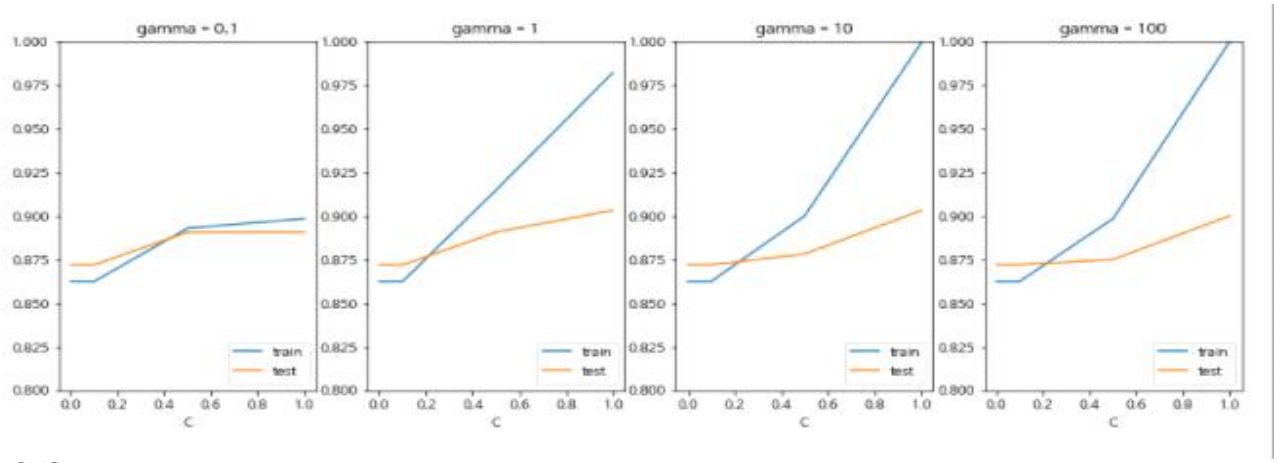


* GradientBoosting
learning_rate에 따른 결과
- 훈련 데이터는 급증하여 과대적합 되고, 테스트 데이터는 비슷한 정확도를 가진다.

```
from sklearn.naive_bayes import GaussianNB

nb = GaussianNB().fit(X_train, y_train)


print("train score " ,nb.score(X_train, y_train))
print("test score", nb.score(X_test, y_test))
```

```
train score  0.8467552775605942
test score 0.828125
```

* Gaussian Naive bayes
- 확실히 속도는 빠르지만 다른 모델들에 비해서 정확도가 조금 떨어지는 편이다.



* SVC

gamma와 C값에 따른 결과

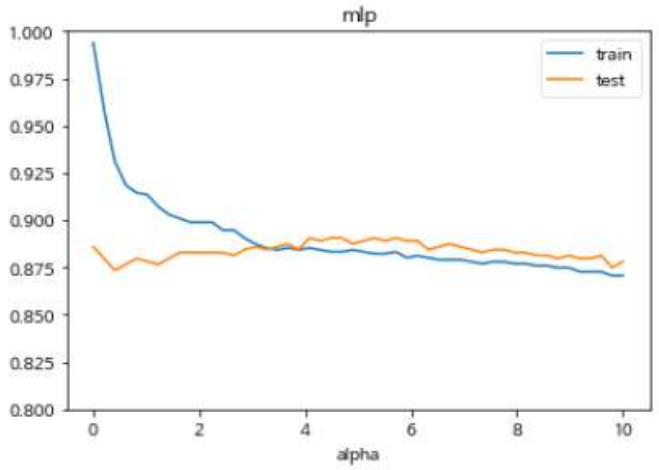
- gamma가 클수록 특정 포인트에 민감해지고 C는 커질수록 규제가 약화되고 작아질수록 강화된다.
- gamma 0.1, 1을 보면 1에서 0.1보다 빠르게 과대적합 되는 것을 볼 수 있다.
- C는 0~1로 증가하며 훈련, 테스트 모두 증가하고 이후에는 일정수준을 유지한다. 규제가 조금씩 풀리면서 과대적합 된다.

```
]: from sklearn.neural_network import MLPClassifier

mp = MLPClassifier(random_state=0, max_iter=500,hidden_layer_sizes=(100,))
print("train score " ,mp.score(X_train, y_train))
print("test score", mp.score(X_test, y_test))
```

```
train score  0.9491790461297889
test score 0.871875
```

* MLP

은둔 레이어를 층 없는 기본 100으로 적용



alpha에 따른 결과

- alpha가 커져서 규제가 강화될수록 훈련데이터의 과대적합이 풀린다.
- 테스트 데이터는 비교적 큰 변화가 없으나 4부터는 테스트 데이터 정확도가 훈련 데이터를 넘어선다. 너무 규제가 많이 들어간 상태.

Cross Validation(교차 검증)

1. KNN

```
# estimator = 모델, cv는 분할 테스트 숫자
knn = KNeighborsClassifier(weights='distance', n_neighbors=3).fit()

print(knn.score(X_test, y_test))
knn_eval = cross_val_score(knn, X = X_new, y = y, cv = 5)
print(knn_eval)  # 5번의 교차 검증 결과를 보여준다

knn_eval.mean()
```

```
0.884375
[0.859375  0.803125  0.8625    0.784375  0.86520376]

0.8349157523510972
```

2. DecisionTree

```
# estimator = 모델, cv는 분할 테스트 숫자
dt = tree.DecisionTreeClassifier(random_state=0).fit(X_train, y_train)

print(dt.score(X_test, y_test))
dt_eval = cross_val_score(dt, X = X_new, y = y, cv = 5)
print(dt_eval)  # 5번의 교차 검증 결과를 보여준다

dt_eval.mean()
```

```
0.846875
[0.853125  0.746875  0.875     0.759375  0.79310345]

0.8054956896551724
```

3. RandomForest

```
# estimator = 모델, cv는 분할 테스트 숫자
rfc = RandomForestClassifier(random_state=0, n_estimators=10).fit(X

print(rfc.score(X_test, y_test))
rfc_eval = cross_val_score(rfc, X = X_new, y = y, cv = 5)
print(rfc_eval)  # 5번의 교차 검증 결과를 보여준다

rfc_eval.mean()
```

```
0.884375
[0.878125  0.846875  0.896875  0.840625  0.88401254]

0.8693025078369907
```

4. GradientBoosting

```
# estimator = 모델, cv는 분할 테스트 숫자
gb = GradientBoostingClassifier(random_state=0, n_estimators=10).fit(

print(gb.score(X_test, y_test))
gb_eval = cross_val_score(gb, X = X_new, y = y, cv = 5)
print(gb_eval)  # 5번의 교차 검증 결과를 보여준다

gb_eval.mean()
```

```
0.896875
[0.875     0.875     0.88125   0.875     0.86833856]

0.8749177115987461
```

GradientBoosting > RandomForest > Knn > DecisionTree 순으로 성능이 좋다.

test size = 0.3

```
knn  0.8645833333333334
logistic  0.8854166666666666
decisiontree  0.8604166666666667
GradientBoosting  0.8875
randomforest  0.9041666666666667
gaussiannb  0.8291666666666667
svm  0.8854166666666666
mlp  0.8854166666666666
```

```
(1119, 11)
(480, 11)
```

모델 모두 default 인수를 사용하여 test size 0.3로 훈련:테스트 1119:480으로 나누어 테스트

랜덤포레스트가 가장 성능이 좋고 가우시안 모델이 비교적 성능이 조금 낮다.
svm, mlp, logistic, gradient boosting 이 비슷하고 knn과 decision tree가 비슷한 결과를 얻었다.

클래스의 기준 = 7

```
knn  0.9895833333333334
logistic  0.9875
decisiontree  0.9708333333333333
GradientBoosting  0.9833333333333333
randomforest  0.9916666666666667
gaussiannb  0.9625
svm  0.9895833333333334
mlp  0.9895833333333334
```

```
0    1581
1      18
```

0, 1 클래스 기준을 7로 바꾸어 테스트했다. 1번 클래스(7~8)은 18개밖에 없다.
대부분이 0이기 때문에 정확도가 0.1정도씩 크게 상승했다.
모든 모델이 과대적합 된 결과를 보인다.

클래스 기준 = 5

```
knn  0.7458333333333333
logistic  0.7416666666666667
decisiontree  0.7729166666666667
GradientBoosting  0.7875
randomforest  0.83125
gaussiannb  0.7229166666666667
svm  0.76875
mlp  0.7833333333333333
```

```
1    855
0    744
```

기준을 5로 바꾸어 실행했다. 클래스간의 샘플수가 거의 비슷해졌다.
테스트 결과도 처음 기준 6.5에 비해서 많이 떨어진 것을 확인할 수 있다.
위와 마찬가지로 랜덤포레스트가 가장 좋은 성능을, 가우시안이 가장 낮은 결과를 보여준다.

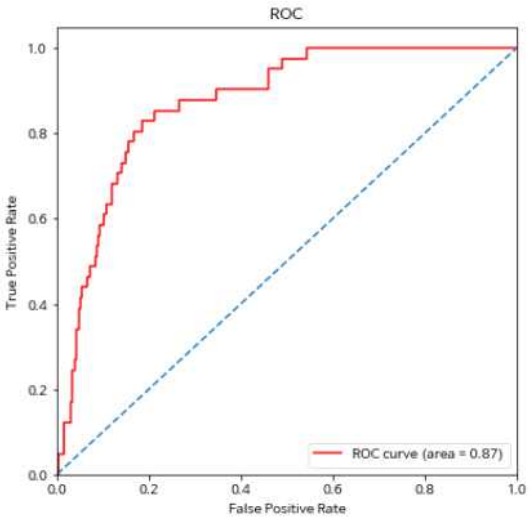
박스 플롯 결과, 품질에 따라 상승, 감소하는 정도가 잘 나타났던 특성들만 모아서 test_size=0.2, 분류 기준 = 6.5로 다시 실행
X_new = wine.drop(['quality', 'qual', 'residual sugar','density', 'free sulfur dioxide', 'total sulfur dioxide'], axis = 1)

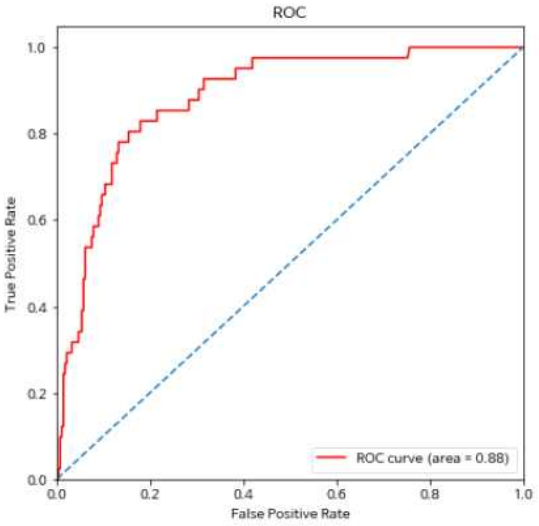
```
0    1382
1     217
Name: qual, dtype: int64
knn  0.875
logistic  0.878125
decisiontree  0.8625
GradientBoosting  0.875
randomforest  0.896875
gaussiannb  0.8375
svm  0.875
mlp  0.875
```

```
knn  0.878125
logistic  0.88125
decisiontree  0.846875
GradientBoosting  0.89375
randomforest  0.903125
gaussiannb  0.828125
svm  0.8875
mlp  0.8875
```

왼쪽이 특성을 덜어낸 것이고 오른쪽이 모든 특성을 포함한 결과다. 대체적으로 모든 특성을 포함한 결과가 더 정확도가 높게 나오지만 큰 차이는 없는 것으로 보인다. alcohol, citric acid 등이 미치는 영향이 큰 것을 알 수 있다.

ROC 성능 평가





Logistic Regression GradientBoosting

decision_function의 결과 값을 사용하기 때문에 가능한 로지스틱과 gradientboosting으로 평가 했다.
파란 대각선 = 랜덤 예측 모델로 FPR=TPR인 곡선이다. 확률이 5:5
TP/FP로 얼마나 양성으로 더 잘 예측했느냐를 보여준다. 0, 1일 때 가장 성능이 좋다.
두 그래프의 빨간 선의 형태가 조금 다른 것을 볼 수 있고, roc curve의 면적이 0.87, 0.88로 약간의 차이가 존재한다.

회귀

품질을 bad(0), good(1)로 나누지 말고 3~8까지를 그대로 이용하여 분석

선형 모델

```
lr = LinearRegression()
lr.fit(X_train_, y_train_)
linear_eval(lr)
```

```
MSE : 0.3795971085022988
RMSE : 0.6161145254758232
MAE : 0.48139673183329845
R2 : 0.3468858137200854
```

```
ridge = Ridge(alpha=0.001)
ridge.fit(X_train_, y_train_)
linear_eval(ridge)
```

```
MSE : 0.37684326150614733
RMSE : 0.613875607518451
MAE : 0.4814497006385282
R2 : 0.351623933425810 1
```

```
lasso = Lasso(alpha=0.001)
lasso.fit(X_train_, y_train_)
linear_eval(lasso)
```

```
MSE : 0.37418901920968467
RMSE : 0.6117099142646657
MAE : 0.48053221810481583
R2 : 0.35619067869023946
```

Support Vector Regressor(SVR)

```
svr = SVR(kernel='rbf',gamma='auto',C=1, epsilon=0.1)
svr.fit(X_train_, y_train_)
linear_eval(svr)
```

```
MSE : 0.3927420272166743
RMSE : 0.626691333286 71
MAE : 0.4732517815162476
R2 : 0.3242693798812514
```

RandomForestRegressor

```
from sklearn.ensemble import RandomForestRegressor
rfg = RandomForestRegressor(random_state=0).fit(X_train_,
linear_eval(rfg)
```

```
MSE : 0.28696031250000004
RMSE : 0.5356867671503562
MAE : 0.39028125
R2 : 0.5062716580415 35
```

KNeighborsRegressor

```
from sklearn.neighbors import KNeighborsRegressor
knn = KNeighborsRegressor().fit(X_train_, y_train_)
linear_eval(knn)
```

```
MSE : 0.49912499999999993
RMSE : 0.7064877918265821
MAE : 0.5418750000000001
R2 : 0.14123260971839524
```

GradientBoostingRegressor

```
from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor().fit(X_train_, y_train_)
linear_eval(gbr)
```

```
MSE : 0.3385241705516556
RMSE : 0.5818282998889411
MAE : 0.4569981533185475
R2 : 0.4175536819596489
```

테스트 사이즈 0.2로 실행

random forest가 가장 성능이 좋고 다음은 gradient boosting, ridge와 lasso가 비슷하며 선형 회귀, svr, knn 순이다.

test size = 0.3

Linear MSE : 0.42816456161020106 RMSE : 0.6543428471452875 MAE : 0.5073574335152197 R2 : 0.32660881529195107 Ridge MSE : 0.4275842741617416 RMSE : 0.6538992844175177 MAE : 0.5080541854142235 R2 : 0.3275214561021 1014 Lasso MSE : 0.42872091551 86099 RMSE : 0.6547678332955964 MAE : 0.5087762869243557 R2 : 0.3257338156981233 KNN MSE : 0.5343333333333332 RMSE : 0.7309810759064377 MAE : 0.56 R2 : 0.15963302752293584 RandomForest MSE : 0.3529972916666667 RMSE : 0.5941357518839164 MAE : 0.42922916666666666 R2 : 0.4448273263433813 SVR MSE : 0.5325079362271237 RMSE : 0.7297314137592842 MAE : 0.555468646235866 R2 : 0.16250390108447132 GradientBoosting MSE : 0.3962691437037235 RMSE : 0.6294991212890797 MAE : 0.48754215743519796 R2 : 0.37677198893254493	테스트 사이즈를 증가시켰다. 테스트 사이즈가 0.2일 때와 비교하여 조금씩 다 성능이 나빠졌다.	Linear MSE : 0.4021379989079101 RMSE : 0.6341435160181882 MAE : 0.4841043935017259 R2 : 0.3797315819355 1493 Ridge MSE : 0.4022518035833375 RMSE : 0.6342332406799075 MAE : 0.48418758385654154 R2 : 0.37955604655664643 Lasso MSE : 0.4045803701173514 RMSE : 0.6360663252502458 MAE : 0.48548933263620303 R2 : 0.37596440317966373 KNN MSE : 0.5921666666666667 RMSE : 0.7695236621876332 MAE : 0.5716666666666667 R2 : 0.08662627615062779 RandomForest MSE : 0.31794083333333334 RMSE : 0.5638624241189808 MAE : 0.403958333333333 36 R2 : 0.5095995447698746 SVR MSE : 0.5312922405000224 RMSE : 0.7288979630236474 MAE : 0.5391342336221768 R2 : 0.180520621 1802166 GradientBoosting MSE : 0.36378675175533 41 RMSE : 0.6031473715066112 MAE : 0.4636049995435734 R2 : 0.4388855725226 5126	stratify=y_로 클래스 비율을 일정하게 고정시켰다. knn을 제외한 모델의 성능이 좋아졌다. 5~6에서의 품질 빈도수가 높다. 따라서 희소한 클래스가 한 곳에 쏠릴 수 있기 때문에 골고루 나누어 더 성능이 개선됐다.

위 테스트에서는 랜덤 포레스트가 다른 모델에 비해 비교적 성능이 좋았다. 뒤에 모델들끼리 모아 비교할 때는 다 기본 값을 사용하여 모델을 생성했기 때문에 매개변수에 민감한 mlp, svm, gradientboosting의 경우는 그 영향이 있는 것 같다.

와인 데이터는 와인에 대한 여러 특성과 그 특성을 바탕으로 하는 품질로 이루어진 데이터세트다.
따라서 위와 비슷하게 여러 특성을 조합하여 결과를 내는, 예를 들어 음식의 맛, 양, 가격 등으로 매기는 만족도나 다른 식품의 제품 품질에 적용할 수 있다.
위에서는 품질의 값을 그대로 예측하는 회귀와 좋고 나쁜 것으로 나눈 이진 분류 두 문제로 나뉘었는데 마찬가지로 음식 만족도의 경우도 평점 회귀 예측과 분류 문제로 나눌 수 있다.
분류를 좋고 나쁨의 0, 1이 아닌 다중 클래스로 나누어 해당 음식점의 등급(예를 들어 A~E)을 정할 수도 있고, 어느 등급이 되기 위하여선 각 특성이 어느 정도인지 최소, 최대, 평균값을 제공할 수도 있을 것이다.
가전제품의 가격과 같은 분야에서도 활용 가능할 것으로 생각한다. 규격, 사양, 효율과 가격이 함께 있는 데이터 세트에서 회귀를 이용하여 제품 특성으로 현재 시장에서는 얼마정도로 평가되는지 예측하여 경쟁성을 높일 수도 있고, 기준점을 정하여 분류 문제로도 풀 수 있다.