Python, R패키지별 데이터 분석 기법 비교

목차

1. 개요

1.1 배경

최근 데이터 분석에 대한 필요성이 증가하며 그에 따른 데이터 분석 기법 또한 다양해지고 있다. 데이터 분석을 위한 이론을 실현하는 다양한 프로그램이 있는데, R과 Python이 대표적이라 할 수 있다. 두 프로그램 모두 데이터 분석을 위한 라이브러리가 다수 존재하며 그 성능과 옵션이 다양하여 같은 데이터를 같은 분석 방법으로 분석하더라도 결과가 다르게 나오는 경우가 많다. 이는 분석을 진행하는 사용자로 하여금 결과에 대한 혼란을 가져오게 되며, 데이터 분석의 신뢰도를 떨어트릴 수 있어 주의 깊게 살펴볼 필요가 있다.

이 프로젝트를 통해 데이터 분석 라이브러리 별 분석 결과를 비교하여 결과의 차이를 확인해보고 그 원인을 알아보고자 한다. 나아가, 함수 내에 존재하는 옵션을 활용하여 동일한 분석 결과를 추출해낼 수 있도록 구현해보고자 한다.

1.2 목표

첫 번째로, 동일한 데이터 셋에 대하여 다양한 분석 기법(선형 회귀분석, 시계열 분석, 로지스틱 회귀분석, 의사결정나무, 인공 신경망, SVM(Support Vector Machine), 앙상블(부스팅, 배깅), 랜덤 포레스트, xgboost)에 대한 R code, R의 caret 패키지, Python의

scikit-learn 패키지 별 분석 결과를 비교한다.

두 번째로, 분석 결과의 차이점을 확인해보고 패키지 별 분석 매커니즘을 상세히 탐구하여 차이가 발생하는 원인을 찾아낸다.

마지막으로, 분석 방법 별 분석 정확도가 가장 우수한 모델에 맞추어 다른 분석 패키지의 옵션을 조정하여 같거나 유사한 결과를 추출해낸다.

1.3 방법

다양한 분석 기법에 맞추어 데이터 셋을 준비한다. R과 Python 두 가지 프로그램을 사용하여 분석이 진행되므로 train/test 셋 분리 시 동일한 데이터가 추출되도록 SyncRNG를 사용하여 데이터를 분리한다. 이후 분석 모델을 생성하여 각 패키지 별 분석 정확도를 계산하여 가장 우수한 모델을 가린다. 분석 시 옵션은 기본 값으로 설정하며, 필수 입력 값은 동일하게 입력한다.

이후 패키지 별 분석 매커니즘을 확인하여 분석 결과의 차이를 유발하는 부분을 확인한다. 이후 옵션을 조정하여 분석 결과가 동일하게 추출되도록 조정을 거쳐 같은 결과를 도출해 낸다.

1.4 데이터

- **회귀 데이터 정의**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/cfe41b58-684e-4a56-b3b1-b7c5e7615a51/product.csv

264개 제품의 점수를 **“제품_친밀도, 제품_적절성, 제품_만족도”**의 세 항목으로 나누어 살펴볼 수 있는 데이터이다. 각 항목은 최소1에서 최대5까지의 점수 값을 포함하고 있다. 본 보고서에서는 **“제품_적절성”**이 **“제품_만족도”**에 미치는 영향을 분석해 보고자 한다. 따라서 **“제품_친밀도”**는 제외하고 **“제품_적절성”, ”제품_만족도”** 두 개 열을 이용하여 분석을 수행한다.

- **다중회귀 데이터 정의**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b594682e-fdaa-4825-9669-3a6180f73813/BostonHousing.csv

mlbench 패키지 내 BostonHousing 데이터 셋, 종속변수는 MEDV

변수명	설명
crim	자치시(town) 별 1인당 범죄율
zn	25,000 평방피트를 초과하는 거주지역의 비율
indus	비소매상업지역이 점유하고 있는 토지의 비율
chas	찰스강에 대한 더미변수 (강의 경계에 위치한 경우는 1, 아니면 0)
nox	10ppm 당 농축 일산화질소
rm	주택 1가구당 평균 방의 개수
age	1940년 이전에 건축된 소유주택의 비율
dis	5개의 보스턴 직업센터까지의 접근성 지수
tax	방사형 도로까지의 접근성 지수
ptratio	10,000달러당 재산세율
b	1000(Bk-0.63)^2, 여기서 Bk는 자치시별 흑인의 비율을 말함
lstat	모집단의 하위계층의 비율(%)
medv	본인 소유의 주택가격(중앙값) (단위: $1,000)

- **분류 데이터 정의**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ea59ad42-fd0f-4e92-b0c0-260c4d72fe31/diabetes.csv

이 데이터 셋의 출처는 National Institute of Diabetes and Digestive and Kidney Diseases로 다양한 측정값에 따른 당뇨병 유무를 예측하기 위한 데이터이다. 이 데이터는 피마 인디언 혈통의 21세 이상의 여성을 대상으로 한다. 종속 변수는 Outcome이다.

변수명	설명
Pregnancies	임신 횟수
Glucose	경구 포도당 부하 검사에서 2시간 동안의 혈장 포도당 농도
BloodPressure	이완기 혈압(mmHg)
SkinThickness	삼두근 피부 주름 두께(mm)
Insulin	2시간 혈청 인슐린(mu U/ml)
BMI	체질량지수(체중(kg)/(키(m))^2)
DiabetesPedigreeFunction	당뇨병 가족력
Age	나이
Outcome	클래스 변수(0 또는 1, 1은 양성 0은 음성)

1.5 공통 코드(baseline)

R

```
# R Baseline
library(SyncRNG)
library(caret)
library(ModelMetrics)
#데이터 불러오기&전처리
df<-read.csv('diabetes.csv')

#종속변수 factor형으로
df$Outcome<-as.factor(df$Outcome)

#데이터분할
v <- 1:nrow(df)
s <- SyncRNG(seed=42)
idx <- s$shuffle(v)[1:round(nrow(df)*0.7)]
idx[1:length(idx)]
train_df <- df[idx,]
test_df <- df[-idx,]

###########모델 train############
#빨간색부분만 모델마다 변경하면 됨
#R 모델
library(e1071)
model<-svm(Outcome~.,train_df)
pred<-predict(model,subset(test_df,select=-c(Outcome)))

#caret 기본 모델
caret_model <- train(Outcome ~ ., data = train_df,
                     method = "svmLinearWeights")
caret_pred<-predict(caret_model, newdata = test_df)

#caret 튜닝모델
caretGrid <-  expand.grid(cost= 2,
                          weight = 1)
caret_tune_model <- train(Outcome ~ ., data = train_df,
                        method = "svmLinearWeights",
                        tuneGrid= caretGrid)
caret_tune_pred<-predict(caret_tune_model, newdata = test_df)
################################

#평가
caret::confusionMatrix(pred,test_df$Outcome)$byClass #R 평가
caret::confusionMatrix(caret_pred,test_df$Outcome)$byClass #caret평가
caret::confusionMatrix(caret_tune_pred,test_df$Outcome)$byClass #caret tune평가
```

Python

```
# Python Baseline
import pandas as pd
from SyncRNG import SyncRNG
import os
os.getcwd()
os.chdir('E:/GoogleDrive/2022년 빅데이터&AI 강의/workplace/Python/work')

#데이터로드
raw_data = pd.read_csv('diabetes.csv')

# 데이터 셋  7:3 으로 분할
v=list(range(1,len(raw_data)+1))
s=SyncRNG(seed=42)
ord=s.shuffle(v)
idx=ord[:round(len(raw_data)*0.7)]

for i in range(0,len(idx)):
    idx[i]=idx[i]-1

# 학습데이터, 테스트데이터 생성
train=raw_data.loc[idx] # 70%
test=raw_data.drop(idx) # 30%

#전처리
y_train=pd.DataFrame(train.Outcome)
x_train=pd.DataFrame(train.drop('Outcome',axis=1))
y_test=pd.DataFrame(test.Outcome)
x_test=pd.DataFrame(test.drop('Outcome',axis=1))

########################모델 train###############
#빨간색부분만 모델마다 변경하면 됨
#기본코드
from sklearn import svm
model = svm.SVC()
model.fit(x_train, y_train)
pred = model.predict(x_test)
##################################################

#결과
output = pd.DataFrame({'idx': x_test.index, 'Outcome': pred})
output.head()

#평가
from sklearn.metrics import classification_report
from sklearn import metrics
print(classification_report(y_test, pred))
print(metrics.f1_score(y_test, pred))
```

2. 데이터 분석기법 비교

2.1 선형회귀

R

```
# R 기본 모델
p_lm<-predict(m_lm,test[,-1])
```

```
# R 기본 모델 평가
RMSE 0.5585512
```

Caret

```
# Caret 기본 모델
gbmFit1 <- train(제품_만족도~제품_적절성,
                 data = train,method = "lm")
```

```
# Caret 튜닝모델
nn_Grid <-  expand.grid(intercept=26)
gbmFit1 <- train(제품_만족도~제품_적절성,
                 data = train,method = "lm",
                 tuneGrid=nn_Grid)
```

```
# Caret 기본 모델 평가
RMSE  0.5585512
```

```
# Caret 튜닝 모델 평가
RMSE  0.5585512
```

Scikit-learn

```
# Scikit-learn 튜닝모델
line_fitter = LinearRegression()
line_fitter.fit(x_train, y_train)
```

```
# Scikit-learn 튜닝모델
line_fitter = LinearRegression(fit_intercept=26)
line_fitter.fit(x_train, y_train)
```

```
# Scikit-learn 기본 모델 평가
RMSE 0.5191416980994801
```

```
# Scikit-learn 튜닝 모델 평가
RMSE 0.5191416980994801
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
RMSE	0.55855	0.55855	0.55855 ➖	0.51914	0.51914 ➖

기본 모델에서는 Scikit-learn의 linear_model이 가장 성능이 좋았다.

caret, Scikit-learn 튜닝 결과 수치에는 변화가 보이지 않았다.

2.2 다중 회귀

R

```
# R 기본 모델
model <- lm(formula = medv ~ ., data = train)
p_lm<-predict(model,test)
```

```
# R 기본 모델 평가
RMSE  5.110768
```

Caret

```
# Caret 기본 모델
gbmFit1 <- train(medv~ ., data = train,method = "lm")
```

```
# Caret 튜닝 모델
nn_Grid <-  expand.grid(intercept=26)
gbmFit1 <- train(medv~ ., data = train,method = "lm",
                 tuneGrid=nn_Grid))
```

```
# Caret 기본 모델 평가
RMSE  5.110768
```

```
# Caret 튜닝 모델 평가
RMSE  5.110768
```

Scikit-learn

```
# Scikit-learn 기본 모델
line_fitter = LinearRegression()
line_fitter.fit(xtrain, ytrain)
```

```
# Scikit-learn 튜닝 모델
line_fitter = LinearRegression(fit_intercept=26)
line_fitter.fit(x_train, y_train)
```

```
# Scikit-learn 기본 모델 평가
RMSE  4.679735971158223
```

```
# Scikit-learn 튜닝 모델 평가
RMSE  4.679735971158223
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
RMSE	5.11076	5.11076	5.11076 ➖	4.67973	4.67973 ➖

기본 모델에서는 Scikit-learn의 linear_model이 가장 성능이 좋았다.

caret, Scikit-learn 튜닝 결과 수치에는 변화가 보이지 않았다.

2.3 로지스틱회귀

R

```
#로지스틱 모델
logit_model<-glm(Outcome ~., family=binomial(), data=train_df)
logit_pred <- predict(logit_model, newdata = test_df, type = 'response')
logit_pred <-ifelse(logit_pred>0.5,1,0)
```

```
 # R 기본 모델 평가
 Precision  0.8427673
 Recall     0.8535032
 F1         0.8481013
```

Caret

```
# Caret 기본 모델
# No tuning parameters for this model
caret_logit_model <- train(Outcome ~ ., data = train_df, method = 'bayesglm')
caretlogit_pred<-predict(caret_logit_model, newdata = test_df)
```

```
# Caret 기본 모델 평가
Precision  0.8427673
Recall     0.8535032
F1         0.8481013
```

Scikit-learn

```
# Scikit-learn 기본 모델
from sklearn.linear_model import LogisticRegression
lr_model = LogisticRegression()
lr_model.fit(x_train, y_train)
lr_y_model = lr_model.predict(x_test)
```

```
# Scikit-learn 튜닝 모델
from sklearn.linear_model import LogisticRegression
lr_model = LogisticRegression(solver="lbfgs",max_iter=200)
lr_model.fit(x_train, y_train)
lr_y_model = lr_model.predict(x_test)
```

```
# Scikit-learn 기본 모델 평가
precision  0.76
recall     0.76
f1-score   0.76
```

```
# Scikit-learn 튜닝 모델 평가
precision  0.76
recall     0.76
f1-score   0.76
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	**0.842**	**0.842**	➖	0.76	0.76 ➖
Recall	**0.853**	**0.853**	➖	0.76	0.76 ➖
F1	**0.848**	**0.848**	➖	0.76	0.76 ➖

기본 모델에서는 R, Caret의 glm, bayesglm 모델이 가장 성능이 좋았다.

caret, Scikit-learn 튜닝 결과 수치에는 변화가 보이지 않았다.

2.4 의사결정나무

R

```
# R 기본 모델
library(rpart)
tree_model<- rpart(Outcome~., data= train_df)
tree_pred<-predict(tree_model,subset(test_df,select=-c(Outcome)),type='class')
```

```
# R 기본 모델 평가
Precision  0.8376623
Recall     0.8216561
F1         0.8295820
```

Caret

```
# Caret 기본 모델
caret_tree_model = train(Outcome ~ .,data=train_df, method="rpart")
caret_tree_pred<-predict(caret_tree_model, newdata = test_df)
```

```
# Caret 튜닝 모델
caret_tune_tree_model <- train(Outcome ~ ., data = train_df,
                               method = "rpart",
                              trControl=trainControl(method='cv'))
caret_tune_tree_pred<-predict(caret_tune_tree_model, newdata = test_df)
```

```
# Caret 기본 모델 평가
Precision  0.8070175
Recall     0.8789809
F1         0.8414634
```

```
# Caret 튜닝 모델 평가
Precision  0.8070175
Recall     0.8789809
F1         0.8414634
```

Scikit-learn

```
# Scikit-learn 기본 모델
from sklearn.tree import DecisionTreeClassifier
tree_model = DecisionTreeClassifier()
tree_model.fit(x_train, y_train)
tree_pred = tree_model.predict(x_test)
```

```
# Scikit-learn 튜닝 모델
from sklearn.tree import DecisionTreeClassifier
tree_model = DecisionTreeClassifier(max_depth=3,min_samples_leaf=3,min_samples_split=2)
tree_model.fit(x_train, y_train)
tree_pred = tree_model.predict(x_test)
```

```
# Scikit-learn 기본 모델 평가
Precision  0.67
Recall     0.66
F1         0.66
```

```
# Scikit-learn 튜닝 모델 평가
Precision  0.74
Recall     0.72
F1         0.73
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	**0.837**	**0.807**	0.807 ➖	**0.67**	**0.74** 🔺
Recall	**0.821**	**0.878**	0.878 ➖	**0.66**	**0.72** 🔺
F1	**0.829**	**0.841**	0.841 ➖	**0.66**	**0.73** 🔺

기본 R모델에서의 Precision은 좋게 나왔지만Recall,F1-score는 Caret에 약간 못미치는 것을 확인 할 수 있다 .

Caret 기본과 튜닝의 성능 차이는 없었고 Scikit-learn은 튜닝 모델이 기본 모델보다 성능이 소폭 상승했음을 알 수 있다.

2.5 인공신경망

R

```
# R 기본 모델
nn_model <- neuralnet(Outcome ~.,
                      data=train,
                      hidden=c(2),
                      threshold=0.01,
```

```
                      stepmax = 1e+05,
                      rep = 1)
```

```
# R 기본 모델 평가
Precision  0.8397436
Recall     0.8343949
F1         0.8370607
```

Caret

```
# Caret 기본 모델
fitControl <- trainControl(method = "repeatedcv",
                           number = 5,
                           repeats = 1)

caret_nn_model<- train(Outcome ~ .,
                       data = train,
                       method = "nnet" ,
                       trControl = fitControl)
```

```
# Caret 튜닝 모델
fitControl <- trainControl(method = "repeatedcv",
                           number = 5,
                           repeats = 1)

nn_Grid <-  expand.grid(size = 15,
                        decay = 2)

caret_tune_nn_model<- train(Outcome ~ .,
                       data = train,
                       method = "nnet" ,
                       tuneGrid = nn_Grid,
                       trControl = fitControl)
```

```
# Caret 기본 모델 평가

Precision  0.8312500
Recall     0.8471338
F1         0.8391167
```

```
# Caret 튜닝 모델 평가

Precision  0.8081395
Recall     0.8853503
F1         0.8449848
```

Scikit-learn

```
# Scikit-learn 기본 모델
nn_model= MLPClassifier(random_state = 1)
nn_model.fit(x_train,y_train)
```

```
# Scikit-learn 튜닝 모델

nn_tune_model = MLPClassifier(solver="adam",
                    max_iter=5000,
                    activation = "relu",
                    hidden_layer_sizes = (12),
                    alpha = 0.05,
                    batch_size = 64,
                    learning_rate_init = 0.001,
                    random_state=1)
```

```
# Scikit-learn 기본 모델 평가

Precision   0.69
Recall      0.65
F1          0.66
```

```
# Scikit-learn 튜닝 모델 평가

Precision     0.70
Recall        0.69
F1            0.70
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	0.839	0.831	0.808	0.69	**0.70** ▲
Recall	0.834	0.847	**0.885** ▲	0.65	**0.69** ▲
F1	0.837	0.839	**0.844** ▲	0.66	**0.70** ▲

기본 모델에서는 Caret의 nn모델이 가장 성능이 좋았다.

nn모델과 비슷한 성능이 나오도록 튜닝한 결과 Scikit-learn 모델의 성능 향상이 있었다. 하지만 큰 향상이 보이진 않았다.

2.6 SVM

R

```
# R 기본 모델
library(e1071)
svm_model<-svm(Outcome~.,train)
svm_pred<-predict(svm_model,subset(test,select=-c(Outcome)))
```

```
# R 기본 모델 평가
Precision  0.8322981
Recall     0.8535032
F1         0.8427673
```

Caret

```
# Caret 기본 모델
caret_svm_model <- train(Outcome ~ ., data = train, method = "svmLinearWeights")
caretsvm_pred<-predict(caret_svm_model, newdata = test)
```

```
# Caret 튜닝모델
caretsvmGrid <-  expand.grid(cost= 5,
                             weight = 1)
caret_tune_svm_model <- train(Outcome ~ ., data = train_df,
                          method = "svmLinearWeights",
                          tuneGrid= caretsvmGrid
                         )
caret_tune_svm_pred<-predict(caret_tune_svm_model, newdata = test_df)
```

```
# Caret 기본 모델 평가
Precision  0.8260870
Recall     0.8471338
F1         0.8364780
```

```
# Caret 튜닝 모델 평가
Precision  0.8271605
Recall     0.8535032
F1         0.8401254
```

Scikit-learn

```
# Scikit-learn 기본 모델
from sklearn import svm
svm_model = svm.SVC()
svm_model.fit(x_train, y_train)
svm_y_pred = svm_model.predict(x_test)
```

```
# Scikit-learn 튜닝 모델
svm_tune_model = svm.SVC(kernel='rbf',cache_size=40,gamma=1/1000)
svm_tune_model.fit(x_train, y_train)
svm_tune_pred = svm_tune_model.predict(x_test)
```

```
# Scikit-learn 기본 모델 평가
precision  0.76
recall     0.72
f1-score   0.74
```

```
# Scikit-learn 튜닝 모델 평가
precision  0.77
recall     0.75
f1-score   0.76
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	**0.832**	0.826	**0.827** ▲	0.76	0.76
Recall	**0.853**	0.847	**0.853** ▲	0.72	**0.74** ▲
F1	**0.842**	0.836	**0.840** ▲	0.74	**0.75** ▲

기본 모델에서는 R(e1071) SVC모델이 가장 성능이 좋았다.

SVC모델과 비슷한 성능이 나오도록 튜닝한 결과 Caret, R, Scikit-learn 모델 모두 성능 향상이 있었다.

2.7 앙상블

R

```
# R 기본 모델
ada_model<-boosting(Outcome~.,train_df)
ada_pred<-predict(ada_model,subset(test_df,select=-c(Outcome)))
```

```
# R 튜닝 모델
R_tune_ada_model <- boosting(Outcome~.,train_df,mfinal = 10, coeflearn = "Zhu")
R_tune_ada_pred<-predict(R_tune_ada_model,subset(test_df,select=-c(Outcome)))
```

```
# R 기본 모델 평가
Precision  0.8289474
Recall     0.8025478
F1         0.8155340
```

```
# R 튜닝 모델 평가
Precision  0.8181818
Recall     0.8598726
F1         0.8385093
```

Caret

```
# Caret 기본 모델
caret_ada_model <- train(Outcome ~ ., data = train_df, method = 'adaboost')
caretada_pred<-predict(caret_ada_model, newdata = test_df)
```

```
# Caret 기본 모델 평가
Precision  0.8333333
Recall     0.8280255
F1         0.8306709
```

Scikit-learn

```
# Scikit-learn 기본 모델
from sklearn.ensemble import AdaBoostClassifier
adaboost_model = AdaBoostClassifier()
adaboost_model.fit(x_train, y_train)
adaboost_pred = adaboost_model.predict(x_test)
```

```
from sklearn.tree import DecisionTreeClassifier
base_model = DecisionTreeClassifier(max_depth = 5)
adaboost_tune_model =
                AdaBoostClassifier(base_estimator = base_model,
                                   n_estimators = 100,
                                   random_state = 10,
                                   learning_rate = 0.01)
```

```
#Scikit-learn기본모델 평가
precision 0.71
recall    0.71
f1-score  0.71
```

```
#Scikit-learn튜닝모델 평가
precision 0.75
recall    0.77
f1-score  0.76
```

결론

	R	R_tune	Caret	Scikit-learn	Scikit-learn_tune
Precision	0.829	0.818 ▲	**0.833**	0.71	0.75 ▲
Recall	0.803	0.860 ▲	**0.828**	0.71	0.77 ▲
F1	0.816	0.836 ▲	**0.830**	0.71	0.76 ▲

기본 모델에서는 Caret의 adaboost모델이 가장 성능이 좋았다.

Caret모델과 비슷한 성능이 나오도록 튜닝한 결과 R, Scikit-learn 모델 모두 성능 향상이 있었다.

2.8 랜덤포레스트

R

```
# R 기본 모델
library(randomForest)
random_model<-randomForest(Outcome~.,train_df)
# RandomForest 모델 예측
random_pred<-predict(random_model,subset(test_df,select=-c(Outcome)))
```

```
# R 기본 모델 평가
Precision   0.8375000
Recall      0.8535032
F1          0.8454259
```

Caret

```
# Caret 기본 모델
caret_random_model <- train(Outcome ~ ., data = train_df,
                        method = "rf")
caretrandom_pred<-predict(caret_random_model, newdata = test_df)
```

```
# Caret 튜닝모델
mtry <- sqrt(ncol(train_df))
caretrandomGrid <-  expand.grid(mtry = mtry)
caret_tune_random_model <- train(Outcome ~ ., data = train_df,
                             method = "rf",
                             ntree=100,
                             tuneGrid= caretrandomGrid)
caret_tune_random_pred<-predict(caret_tune_random_model, newdata = test_df)
```

```
# Caret 기본 모델 평가
Precision   0.8364780
Recall      0.8471338
F1          0.8417722
```

```
# Caret 튜닝 모델 평가
Precision   0.8417722
Recall      0.8471338
F1          0.8444444
```

Scikit-learn

```
# Scikit-learn 기본 모델
from sklearn.ensemble import RandomForestClassifier
random_model = RandomForestClassifier()
random_model.fit(x_train,y_train)
random_pred = random_model.predict(x_test)
```

```
#Scikit-learn 튜닝 모델
from sklearn.ensemble import RandomForestClassifier
random_model = RandomForestClassifier(n_estimators=100,max_features=4,max_depth=5)
random_model.fit(x_train,y_train)
random_pred = random_model.predict(x_test)
```

```
#Scikit-learn 기본 모델 평가
precision 0.74
recall    0.75
f1-score  0.75
```

```
#Scikit-learn 튜닝 모델 평가
precision 0.75
recall    0.76
f1-score  0.76
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	**0.837**	0.836	**0.841** ▲	0.74	**0.75** ▲
Recall	**0.853**	0.847	**0.847**	0.75	**0.76** ▲
F1	**0.845**	0.841	**0.844** ▲	0.75	**0.76** ▲

Caret 튜닝 모델 Precision이 가장 성능이 좋게 나왔고 Recall,F1-score은 R의 기본 모델이 성능이 가장 좋게 나왔다

Caret의 기본 모델에 비해서 튜닝 모델이 Recall은 동일하고 나머지 부분은 성능 향상이 있다.

Scikit-learn의 기본 모델에 비해 전체적으로 튜닝 모델이 성능이 더 좋음을 확인 할 수 있다.

2.9 xgboost

R

```
# R 기본 모델 생성
xgbmodel <- xgboost(data = dtrain,
                    max.depth=5,
                    nrounds=20)

# xgb모델 예측
XGB_pred <- predict(xgbmodel, dtest)
XGB_pred <- ifelse(XGB_pred >= 0.5, 1, 0)
```

```
#R 기본 모델 평가

Precision   0.8407643
Recall      0.8407643
F1          0.8407643
```

Caret

```
# caret 기본모델
caret_xgb_model<- train(Outcome ~ ., data = train,
                        method = "xgbTree")
caret_xgb_pred<-predict(caret_xgb_model, newdata = test)

# caret 튜닝모델
caretxgbGrid <-  expand.grid(nrounds = 200,
                             max_depth = 5,
                             eta = 0.05,
                             gamma = 0.01,
                             colsample_bytree = 0.75,
                             min_child_weight = 0,
                             subsample = 0.5)

caret_tune_xgb_model <- train(Outcome ~ .,
                              data = train,
                              method = "xgbTree",
                              tuneGrid= caretxgbGrid)

caret_tune_xgb_pred<-predict(caret_tune_xgb_model, newdata = test)
```

```
#caret 기본 모델 평가

Precision   0.8410596
Recall      0.8089172
F1          0.8246753
```

```
#caret tune 모델 평가

Precision   0.8506494
Recall      0.8343949
F1          0.8424437
```

Scikit-learn

```
# Scikit-learn 기본 모델
xgb_model = XGBClassifier()
xgb_model.fit(x_train, y_train)
```

```
# Scikit-learn 튜닝 모델
xgb_tune_model = xgb.XGBClassifier(booster = 'gbtree',
                                   learning_rate = 0.02,
                                   n_estimators=1000,
                                   min_child_weight=3 ,
                                   max_depth=8)
xgb_tune_model.fit(x_train, y_train,
                   early_stopping_rounds=30,
                   eval_set=[(x_test, y_test)],
                   verbose=False )
```

```
# Scikit-learn 튜닝 모델 평가

Precision
Recall
F1
```

```
# Scikit-learn기본모델 평가

precision 0.75
recall    0.76
f1-score  0.76
```

```
#Scikit-learn튜닝모델 평가

precision 0.77
recall    0.77
f1-score  0.77
```

결론

	R	Caret	Caret_tune	Scikit-learn	Scikit-learn_tune
Precision	**0.840**	0.841	**0.850** ▲	0.75	**0.77** ▲
Recall	**0.840**	0.808	**0.834** ▲	0.76	**0.77** ▲
F1	**0.840**	0.824	**0.842** ▲	0.76	**0.77** ▲

기본 모델에서는 R의 XGB모델이 가장 성능이 좋았다.

XGB모델과 비슷한 성능이 나오도록 튜닝한 결과 caret, sklearn 모두 성능 향상이 있었다.

2.10 최종 결론

<가장 좋은 모델 수>

	기본모델	튜닝모델
R	5	5
Caret	3	3
Scikit-learn	2	2

간단한 모델인 선형회귀,다중회귀 모델을 제외하고는 Python보다 R이 기본적으로 모델 성능이 좋았다. R내에서도 Caret라이브러리 함수 보다는 각 모델 라이브러리(기본 모델)를 사용하는것이 결과가 좋은 편이였다. 파라미터 튜닝후에도 R 기본 모델이 더 좋은 편으로 관찰됐다. Python보다 R 라이브러리 함수의 default 값이 결과가 잘 나오게 조정된 느낌이였고 R default값을 Python 모델에 적용해보았지만 약간의 성능 향상이 있을 뿐 R만큼 성능이 나오지는 못했다.

3. 패키지내 기능 설명

3.1 전처리

	Caret	Scikit-learn
표준화,정규화	preProcess 기능 센터링 및 스케일링	표준화 또는 평균 제거 및 분산 스케일링 비선형 변환 정규화
변수 인코딩	더미 변수 만들기	범주형 변수 인코딩 이산화
차원	선형 종속성 클래스 거리 계산 결측치 보정 예측 변수 변환	다항식 특징 생성
	상관된 예측 변수 식별 0 및 0에 가까운 분산 예측기	누락된 값의 대치 맞춤형 변환

3.1.1 caret

preProcess Function

`preProcess`는 중심화 및 스케일링을 포함하여 예측 변수에 대한 많은 작업에 사용할 수 있습니다. 매개변수를 추정하고 `predict`로 특정 데이터 세트에 적용합니다. `preProcess`는 실제로 데이터를 사전 처리하지 않습니다.

```
preProcess(data, method = c(" "))
```

`method`	처리 유형을 지정하는 파라미터 "BoxCox", "YeoJohnson", "expoTrans", "center", "scale", "range", "knnImpute", "bagImpute", "medianImpute", "pca", "ica", "spatialSign", "corr", "zv", "nzv", "conditionalX"

Centering and Scaling

`preProcess` 옵션 `"range"`은 데이터를 0과 1 사이의 간격으로 조정합니다.

```
preProcessData <- preProcess( data , method = c (" center ", " scale "))
dataTransformed <- predict ( preProcessData , data )
```

Zero and Near Zero-Variance Predictor

`nearZeroVar()`

고유한 값이 하나인 예측 변수(예: 분산이 0인 예측 변수)를 진단합니다.

```
nearZeroVar(
  x, # 데이터 프레임
  freqCut = 95/5, #가장 흔한 값과 두 번째로 흔한 값의 비율 컷오프
  uniqueCut = 10, #총 샘플 수에서 개별 값의 백분율에 대한 컷오프
  saveMetrics = FALSE,#F:0에 가장 가까운 예측 변수의 위치 반환
                      #T:예측자 정보가 있는 데이터 프레임 반환
  names = FALSE, #false:열 인덱스 반환 true:열 이름 반환
  foreach = FALSE, #foreach 패키지를 사용여부
  allowParallel = TRUE #병렬 처리 사용여부
)
```

Creating Dummy Variables(더미 변수 만들기)

`dummyVars()`

더미 변수의 전체 집합을 만들때 사용합니다

```
dummyVars(formula, ...)
```

data	관심 있는 예측 변수가 있는 데이터 프레임

Identifying Correlated Predictors(상관된 예측변수의 식별:중복 변수 제거)

`findCorrelation()`

-상관 관계가 있는 예측 변수에 대해서도 잘 작동하는 일부 모형이 있지만 다른 모델은

예측 변수간의 상관 관계 수준을 줄이는 것이 도움이 됨

```
findCorrelation(
  x,#데이터 프레임
  cutoff = 0.9,
  verbose = FALSE,
  names = FALSE,
  exact = ncol(x) < 100
)
```

cutoff	상관관계 컷오프에 대한 숫자 값
verbose	세부 정보를 가져오는 정수
names	열 이름을 가져오는 정수
exact	평균을 다시 평가함 각 단계에서 상관 관계를 모두 사용하는 동안 제거되었는지 여부에 관계 없이 적은 수의 예측 변수를 제거,차원이 클수록 느려짐

Linear Dependencies(선형 종속성)

`findLinearCombos()`

-행렬의 QR 분해를 사용하여 선형 결합의 집합을 열거한다.(존재하는 경우에)

```
findLinearCombos(x)
```

Class Distance Calculations(군집 거리 계산)

`classDist()`

-요빈 변수의 각 수준에 대해 군집의 중심과 공분산 행렬이 계산된다.

-새로운 표본에 대해 각 군집 중심까지의 마할라노비스 거리가 계산되고 이 값은 추가 예측 변수로 사용될 수 있다.

```
classDist(x, y, groups = 5, pca = FALSE, keep = NULL, ...)
```

groups	숫자 결과를 분할하기 위한 그룹 수에 대한 정수
pca	클래스별로 데이터를 분할하기 전에 데이터 세트에 주 성분 분석
keep	새로운 샘플을 예측하는 데 사용 해야하는 PCA구성 요소 수에 대한 정수
trans	각 클래스 거리에 적용할 수 있는 정수

Centering and Scaling(중심화와 척도화)

데이터의 절반을 예측 변수의 위치와 규모를 추정 할 때 사용된다

```
inTrain <- sample(seq(along = mdrrClass), length(mdrrClass)/2)

training <- filteredDescr[inTrain,]
test <- filteredDescr[-inTrain,]
trainMDRR <- mdrrClass[inTrain]
testMDRR <- mdrrClass[-inTrain]

preProcValues <- preProcess(training, method = c("center", "scale"))

trainTransformed <- predict(preProcValues, training)
testTransformed <- predict(preProcValues, test)
```

Imputation(전가)

-학습 집합의 정보만을 기반으로 데이터 집합을 대체하는 데 사용할 수 있다

-이를 수행하는 한 가지의 방법은 K-근접 이웃을 이용하는 것이다.임의의 하나의 표본에 대해 K개의 가장 가까운 이웃을 훈련용 자료에서 발견하고 이들 값을 이용하여 예측 변수의 결측값을 대치한다.

-이 접근법을 사용하면 method = 인자가 무엇이든 관계없이 자동적으로 preProcess()가 데이터의 중심화와 척도화를 수행하게 된다.

Transforming Predictors(예측 변수 변환)

-경우에 따라 주성분 분석(PCA)을 사용하여 데이터를 새변수가 서로 상관되지 않는 더 작은 부분 공간으로 변환해야 한다 클래스는 인수에 포함하여 변환을 적용 가능하다.

-이렇게 하면 예측 변수의 크기 조정도 강제로 수행된다.

3.1.2 Scikit-learn

Standardization, or mean removal and variance scaling

(평균제거하고 분산스케일링하여 표준화)

표준 정규 분포 데이터를 따르지 않는 데이터는 실제 모델에서 제대로 예측이 안될 수 있기 때문에 조정을 해줍니다.

```
#Z-score 표준화
scaler = preprocessing.StandardScaler().fit(data)
scaler.transform(data)
```

```
#min-max scale
min_max_scaler = preprocessing.MinMaxScaler()
min_max_scaler.fit_transform(data)
```

Non-linear transformation(비선형 변환)

Quantile 변환 및 Power 변환의 두 가지 유형의 변환을 사용할 수 있습니다. Quantile 및 Power 변환은 모두 기능의 단조로운 변환을 기반으로 하므로 각 기능을 따라 값의 순위를 유지합니다.

`QuantileTransformer` 는 0과 1 사이의 값을 사용하여 균일한 분포에 데이터를 매핑하는 비모수 변환을 제공합니다.

```
#Quantile 변환
q_trans = preprocessing.QuantileTransformer(random_state=0)
q_trans.fit_transform(data)
```

`PowerTransformer` 는 Yeo-Johnson 변환과 Box-Cox 변환을 제공합니다

```
#power 변환
pt = preprocessing.PowerTransformer(method='box-cox', standardize=False)
pt.fit_transform(data)
```

Nomarlize(정규화)

정규화는 **단위 표준을 갖도록 개별 샘플을 스케일링** 하는 프로세스입니다 . normalize 함수는 l1, l2, max규제를 사용하여 변환합니다.

`preprocessing.normalize(data, norm='l2')`

Encoding categorical features(범주형 변수 인코딩)

범주형 변수를 하나의 새로운 정수 변수(0에서 n_categories - 1)로 변환합니다.

```
#범주형 변수를 정수로 변환
enc = preprocessing.OrdinalEncoder()
enc.fit(data)
```

Discretization(이산화)

비닝이라고도 함. 연속형 변수를 이산 값으로 분할하는 방법을 제공합니다

```
# K-bin 이산화
est = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2], encode='ordinal').fit(data)
est.transform(data)
```

```
#변수 이진화
bin = preprocessing.Binarizer(threshold=1.1) #threshold 임계값 조정
bin.transform(X)
```

Imputation of missing values(결측치 대치)

`SimpleImputer` 는 누락된 값 `np.nan`이 포함된 열(축 0)의 평균값을 사용하여 로 인코딩된 누락된 값을 대체하는 방법 입니다.

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(data)
```

`KNNImputer` 는 k-최근접 이웃 접근 방식을 사용하여 누락된 값을 채우기 위한 대체를 제공합니다

```
from sklearn.impute import KNNImputer
nan = np.nan
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(data)
```

Generating polynomial features(다항식 특징 생성)

비선형 변수를 고려하여 모델에 차원을 추가합니다.

$$(X_1, X_2) -> (1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$$

```
#다항식 기능
fromsklearn.preprocessingimport PolynomialFeatures
poly = PolynomialFeatures(2) # 기존 변수에 2(제곱)
poly.fit_transform(data)
```

다항식은 다음 점으로 갈수록 점점 높은 차수가 필요한 반면,

`SplineTransformer` 은 고정된 낮은 차수를 유지하고 knots 수를 조정하는 경우 매우 유연하고 견고합니다.

```
#스플라인 변환
from sklearn.preprocessing import SplineTransformer
spline = SplineTransformer(degree=2, n_knots=3)
>>> spline.fit_transform(data)
```

Custom transformers(맞춤형 변환)

`FunctionTransformer` 은 기존 Python 함수를 변환기로 변환하려는 경우 사용합니다.

```
from sklearn.preprocessing import FunctionTransformer
transformer = FunctionTransformer(np.log1p, validate=True)
transformer.transform(X)
```

3.2 데이터 분할

	Caret	Scikit-learn
Based	`createDataPartition`	`train_test_split`
Predictors	`maxDissim`	—
Important Groups	`groupKFold`	[illegible]
Time Series	`createTimeSlices`	`TimeSeriesSplit`
Duplicate	`gcreateResample`	`RepeatedStratifiedKFold`

3.2.1 Caret

`createDataPartition`

1. sample 함수와 비슷함
2. caret에서 제공하는 방법 중 제일 간단한 방법
3. 중복을 허용하지 않고 자료의 일부분을 학습 자료로 나눌 수 있음

```
createDataPartition(
  y,
  times = 1,
  p = 0.5,
  list = TRUE,
  groups = min(5, length(y))
)
```

y	결과 벡터. createTimeSlices의 경우 시간순으로 정렬되어야 합니다.
times	생성할 파티션 수
p	교육에 사용되는 데이터의 비율
list	논리적 - 결과는 목록(`TRUE`) 또는 행 수가 `floor(p * length(y))` 과 `times` 같고 열이 있는 행렬 이어야 합니다 .
groups	숫자 `y` 의 경우 분위수의 나누기 수

- **Predictors**

`maxDissim`

1. 최대한 다양한 자료를 포함하도록 하는 방법
2. 시간이 오래 걸림
3. 자료가 적은 경우 유용함
4. 내가 가지고 있는 것과 가장 유사하지 않은 자료를 하나씩 뽑아가는 방법

```
maxDissim(
  a,
  b,
  n = 2,
  obj = minDiss,
  useNames = FALSE,
  randomFrac = 1,
  verbose = FALSE,
  ...
)
```

a	시작할 샘플의 매트릭스 또는 데이터 프레임
b	샘플링할 샘플의 행렬 또는 데이터 프레임
n	하위 샘플의 크기
obj	전반적인 비유사성을 측정하는 목적 함수
userNames	논리적: 함수가 행 이름을 반환해야 하는지(행 인덱스와 반대)
randomFrac	나머지 후보 값에서 하위 샘플링하는 데 사용할 수 있는 (0, 1]의 숫자
verbose	논리적; 각 단계를 print

- **ImportantGroups**

`groupKFold`

1. createDataPartition을 여러번 해주는 기능
2. 여러 번 시행함으로써 안정적인 결과를 얻을 수 있음

```
groupKFold(group, k = length(unique(group)))
```

group	길이가 전체 데이터 세트의 행 수와 일치하는 그룹의 벡터입니다.

- **TimeSeies(시계열)**

`createTimeSlices`

1. 시계열 데이터를 slicing 할때 사용
2. 기상예보를 예측하려고 할때를 사용
3. 순차적으로 데이터를 분할

```
createTimeSlices(y, initialWindow, horizon = 1, fixedWindow = TRUE, skip = 0)
```

initialWindow	각 트레이닝 세트 샘플의 초기 연속 값 수

- **Duplicate**

`createResample`

1. 중복을 허용하여 자료를 뽑음
2. 부트스트래핑 방법이라고 할 수 있음
3. 중복을 허용했기 때문에 training set에 자료를 많이 넣어도 test set에도 자료가 많이 존재함

```
createResample(y, times = 10, list = TRUE)
```

3.2.2 Scikit-learn

`train_test_split`

1. 배열 또는 행렬을 무작위 학습 및 테스트 부분 집합으로 분할
2. 입력 유효성 검사 및 응용 프로그램을 입력 데이터에 래핑하는 빠른 유틸리티 데이터를 분할

```
train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)
```

arrays	동일한 길이/모양[0]의 인덱서블 배열 시퀀스
test_size	float 또는 int, 기본값=없음
train_size	float 또는 int, 기본값=없음
random_state	int, RandomState 인스턴스 또는 없음, 기본값=없음
shuffle	bool, 기본값=True
stratify	array-like, 기본값=None

- **ImportantGroups**

`groupKFold`

1. 반복되는 층화 K-폴드 교차 검증기.
2. 층화된 K-Fold를 각각 다른 무작위화로 n번 반복

```
GroupKFold(n_splits=5)
```

n_splits	int, 기본값=5
max_train_size	정수, 기본값=없음
test_size	정수, 기본값=없음
gap	정수, 기본값=0

- **TimeSeries**

`TimeSeriesSplit`

1. 훈련용 데이터는 검증용보다 항상 앞선 시간으로 할당
2. 과거 시간을 훈련용으로 쓰는 것

```
TimeSeriesSplit(n_splits=5, *, max_train_size=None, test_size=None, gap=0)
```

- **Duplicate**

`RepeatedStratifiedKFold`

1. 래스들은 폴드를 한번만 나누는 것이 아니고 지정한 횟수만큼 반복해서 나눔
2. 다른 무작위 화로 Stratified K-Fold를 n 번 반복

```
RepeatedStratifiedKFold(*, n_splits=5, n_repeats=10, random_state=None)
```

n_splits	int, 기본값=5
n_repeats	int, 기본값=10
random_state	int, RandomState 인스턴스 또는 없음, 기본값=없음

3.3 평가

	Caret	Scikit-learn
accuracy	`confusionMatrix(pred,original)$overall[1]`	`metrics.accuracy_score`
f1	`confusionMatrix(pred,original)$byclass[7]`	`metrics.f1_score`
ROC	`twoClassSummary(pred,original)`	`metrics.roc_auc_score`
클러스터링	`Kmeans` , `trainControl()` , `train()`	`KMeans`
MAE	`postResample(pred, original)[2]`	`mean_absolute_error`
RMSE	`postResample(pred, original)[0]`	`metrics.mean_squared_error`
R2	`postResample(pred, original)[1]`	`metrics.r2_score`

3.3.1 caret

- **Accuracy**

`confusionMatrix(pred,original)`

분류 모델의 학습 성능 평가를 위한 행렬

전체 예측에서 옳은 예측의 비율

1에 가까울수록 좋다.

```
confusionMatrix(예측값 factor 벡터, 실제값 factor 벡터)
```

- **F1_score**

`confusionMatrix(pred,original)`

F1 = 2 * (precision * recall) / (precision + recall)

F1 점수는 정밀도의 평균으로 해석 될 수 있음

F1 점수가 1이 최고 값이고, 0에서 최고 낮은 점수

- **ROC_Curve**

`twoClassSummary(pred,original)`

ROC 곡선 아래의 면적, 민감도 및 특이도와 같은 2-집단(class) 문제에 대한 측도를 계산

왼쪽 위쪽으로 볼록하고 AUC값이 1에 가까울 수록 좋음

```
trainControl(method = "repeatedcv",
             repeats = 3,
             classProbs = TRUE,
             summaryFunction = twoClassSummary)
```

`classProbs = TRUE`	예측된 클래스 확률(자동으로 계산되지 않음)을 기반
`number`	훈련 데이터 fold 갯수
`summaryFunction`	관측값과 예측값을 취하여 성능 측도를 추정

```
train(Class ~ .,
    data = train,
    method = " ",
    tuneLength = 15,
    trControl = ctrl,
    metric = "ROC",
    preProc = c("center", "scale"))
```

`metric`	"accuracy", "Kappa", "RMSE", "Rsquared" 가 존재. 최종모형의 선택에 사용되는 목적함수를 지정
`tuneLength`	평가되는 수를 제어
`preProc`	변수변환, 필터링, 기타 다른 연산작업을 예측변수(predictors)에 적용 / 예측변수 정규화: "center", "scale", "range"

분류

- **K- Nearest Neighbor(KNN)**

가장 가까이에 있는 데이터 클래스에 속한다고 보는 방법

가까이 있는 데이터 K개

`trainControl()` : 데이터 훈련 과정의 파라미터 설정

```
trainControl(
          method = "repeatedcv"
          number = 10
          repeats = 5
)
```

`method`	cross - validation 반복
`number`	훈련 데이터 fold 갯수
`repeats`	cross-validation 반복 횟수

`train()` : 머신러닝 알고리즘 이용해 데이터학습을 통한 모델 생성

```
train(
         Class~.,
         data = train,
         method = "knn",
         trControl = trainControl(),
         preProcess = c("center","scale"),
         tuneGrid = expand.grid(k=1:10),
         metric = "Accuracy"
)
```

`Class~.`	타겟변수와 피쳐 .의 의미는 모든 피쳐를 다 넣는다는 의미
`method`	사용하고 싶은 머신러닝 방법
`trControl`	학습 방법
`preProcess`	표준화: 평균을 빼고 표준편차로 나눔
`tuneGrid`	튜닝 파라미터 값 목록(특정 값이 필요할 때 사용)

`metric`	모형 평가 방식

군집

- **Kmean**

주어진 군집 수 k에 대해서 군집내 거리제곱합의 합을 최소화 하는것이 목적

즉, 거리제곱합의 합이 군집화가 얼마나 잘 됐는지 알려주는 척도가 된다.

```
kmeans(x, centers, iter.max = 10, nstart = 1,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy",
                     "MacQueen"), trace=FALSE)
fitted(object, method = c("centers", "classes"), ...)
```

`x`	데이터로 구성된 숫자형 행렬 또는 강제 변환할 수 있는 개체 이러한 행렬 (예: 숫자형 벡터 또는 모두를 갖는 데이터 프레임 숫자 열).
`centers`	클러스터의 수
`iter.max`	그룹 중심점을 찾기 위한 최대 반복 횟수
`nstart`	초기에 그룹 중심점을 임의로 잡을 때 몇 개의 점을 이용할 것인가
`trace`	논리 또는 정수 숫자, 현재 기본 방법이 양수 (또는 참) 인 경우 알고리즘의 진행률에 대한 추적 정보
`algorithm`	사용할 알고리즘

- **MAE**

`postResample (pred, original)[2]`

회귀 모델이 잘 할습되었는지를 확인할 때 많이 사용되는 평가지표

실제 정답 값과 예측 값의 차이를 절대값으로 변환한 뒤 평균을 구하여 도출

```
postResample(pred, obs)
```

- **RMSE**

`postResample (pred, original)[0]`

MSE 값은 Error의 제곱의 평균으로 실제 Error보다 큰 Error들의 평균을 가지는 특성

MSE값에 √ root를 씌워 RMSE 값을 사용

- **R2**

`postResample (pred, original)[1]`

결정계수 방법은 Regression의 정합성을 판단시 자주 사용하는 방법

에러의 자체 크기 보단 실제 데이터의 분산과 Regression 에러간의 비율과 관련된 값

3.3.2 Scikit-learn

- **Accuracy**

`metrics.accuracy_score`

다중 레이블 분류에서 이 함수는 하위 집합 정확도를 계산

```
from sklearn.metrics import accuracy_score
accuracy_score(y_true, y_pred, normalize=False)
```

`sample_weight`	배열모양(n_samples), default=None
`normalize`	F:올바르게 분류된 샘플의 수를 반환 T:올바르게 분류된 샘플의 비율을 반환

- **F1_score**

`metrics.f1_score`

F1 = 2 * (precision * recall) / (precision + recall)

F1 점수는 정밀도의 평균으로 해석 될 수 있음

F1 점수가 1이 최고 값이고, 0에서 최고 낮은 점수

```
from sklearn.metrics import f1_score
f1_score(y_true, y_pred, average='macro')
```

`average`	'**micro', 'macro', 'samples', 'weighted', 'binary'**

- **ROC_Curve**

`metrics.roc_auc_score`

분류기(classifier)에 쓰이는 Metric

왼쪽 위쪽으로 볼록하고 AUC값이 1에 가까울 수록 좋음

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y, clf.predict_proba(X)[:, 1])
```

`average`	'**micro', 'macro', 'samples', 'weighted'**
`max_fpr`	float > 0 및 <= 1, [0,max_fpr]범위에 걸쳐 AUC계산
`multi_class`	**{'raise', 'ovr', 'ovo'}** 다중 클래스 대상에만 사용.

- **K-Means cluster(KMC)**

데이터 샘플을 지정한 k개의 클러스터로 분류하며 학습된 모델을 활용해 새로운 데이터 샘플에 대해 예측

지도학습에 비해 정확도가 떨어지지만 훈련 데이터의 정답 레이블을 따로 준비할 필요가 없다.

k개의 임의의 중심데이터를 설정하고 데이터가 들어올 때마다 가장 가까운 중심점의 클래스로 분류(군집화)해준다. 데이터가 쌓이면 군집이 생성되어 해당 군집의 '중심'을 찾아서 중심데이터 k를 갱신한다. 군집의 중심은 군집내 속한 데이터들의 위치(x와 y 데이터)를 모두 더하고, 전체 개수로 나눠준 평균 도출.

평균값을 클러스터의 중심으로 하는 과정을 반복하다가 데이터 샘플이 속한 클러스터가 더이상 바뀌지 않으면 학습을 종료한다.

```
from sklearn.cluster import KMeans
k_means = KMeans(n_clusters=3, random_state=0)
k_means.fit(train_input)
```

데이터가 들어올 때마다 가장 가까운 중심점의 클래스로 분류(군집화)해줍니다.

데이터가 쌓이면, 군집이 생성될텐데, 해당 군집의 '중심'을 찾아서 중심데이터 k를 갱신합니다.

군집의 중심을 찾는 방법은, 군집내 속한 데이터들의 위치(x와 y 데이터)를 모두 더하고, 전체 개수로 나눠준 평균을 구함

`n_clusters`	하이퍼 파라미터 k를 설정(몇개의 클러스터로 분류할 것인지 지정), KMeans 객체를 생성한다.

`k_means.fit`	생성한 객체에, 군집의 중심을 갱신시킬 훈련데이터를 입력
`random_state`	클러스터링 결과가 매번 동일하도록 고정

- **MAE**

`mean_absolute_error`

회귀 모델이 잘 할습되었는지를 확인할 때 많이 사용되는 평가지표

실제 정답 값과 예측 값의 차이를 절대값으로 변환한 뒤 평균을 구하여 도출

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_true, y_pred, multioutput='raw_values')
```

`multioutput`	{'raw_values', 'uniform_average'} 여러 출력 값의 집계를 정의합니다

- **MSE**

`metrics.mean_squared_error`

MSE 값은 Error의 제곱의 평균으로 실제 Error보다 큰 Error들의 평균을 가지는 특성

MSE값에 √ root를 씌워 RMSE 값을 사용

`sample_weight`	배열모양(n_samples), default=None
`mulitoutput`	{'raw_values', 'uniform_average'} 또는 배열모양(n_samples), default='uniform_average'
`squared`	bool, default=True T : MSE 값 반환 F : RMSE 값 반환

- **R2**

`metrics.r2_score`

결정계수 방법은 Regression의 정합성을 판단시 자주 사용하는 방법

에러의 자체 크기 보단 실제 데이터의 분산과 Regression 에러간의 비율과 관련된 값

`sample_weight`	배열모양(n_samples), default=None
`mulitoutput`	{'raw_values', 'uniform_average','variance_weighted'} 또는 배열모양(n_samples), default='uniform_average'
`force_finite`	bool, default=True

4. 소스코드

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/187f746d-5468-4e27-b697-9c2a45d1da20/%EC%86%8C%EC%8A%A4%EC%BD%94%EB%93%9C.zip