파이썬 머신러닝 노트정리 - 박선우

데이터 전처리

1. LabelEncoding: from sklearn.preprocessing import LabelEncoder

from sklearn.preprocessing import LabelEncoder

items = ['TV', '냉장고', '전자렌지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

encoder = LabelEncoder()

encoder.fit(items)

labels = encoder.transform(items)

print('인코딩 변환값:', labels)

#인코딩 변환값: [0 1 4 5 3 3 2 2]

print('디코딩 원본 값:', encoder.inverse\_transform([4,5,2,0,1,1,3,3]))

# 원본값이 무엇인지 확인이 가능하다

디코딩 원본 값: ['전자렌지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']

* 활용 방법:

#scikit learn은 먼저 숫자값으로변환을 못해서 LabelEncoder로 변환후,

# 2차원 데이터로 변환하고 원-핫 인코딩을 적용한다

from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()

encoder.fit(items)

labels = encoder.transform(items)

labels = labels.reshape(-1,1)

from sklearn.preprocessing import OneHotEncoder

oh\_encoder = OneHotEncoder()

oh\_encoder.fit(labels)

oh\_labels = oh\_encoder.transform(labels)

print(oh\_labels.toarray())

print(oh\_labels.shape)

[[1. 0. 0. 0. 0. 0.]

[0. 1. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0.]

[0. 0. 0. 0. 0. 1.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 1. 0. 0. 0.]

[0. 0. 1. 0. 0. 0.]]

(8, 6)

Pandas에서 One-hot Encoding 하는법:

#pandas 에서는 더 쉽게 one-hot encoding 을 할수 있다 먼저 DataFrame을 만든다음에 그 DataFrame을

# pd.get\_dummies(df)를 하면 one-hot encoding이 된다

import pandas as pd

df = pd.DataFrame({'items': ['TV', '냉장고', '전자렌지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']})

df = pd.get\_dummies(df)

df.head()

Feature들을 다듬는 방법들:

from sklearn.preprocessing import LabelEncoder

# Null 처리 함수

def fillna(df):

df['Age'].fillna(df['Age'].mean(),inplace=True)

df['Cabin'].fillna('N',inplace=True)

df['Embarked'].fillna('N',inplace=True)

df['Fare'].fillna(0,inplace=True)

return df

# 머신러닝 알고리즘에 불필요한 속성 제거

def drop\_features(df):

df.drop(['PassengerId','Name','Ticket'],axis=1,inplace=True)

return df

# 레이블 인코딩 수행.

def format\_features(df):

df['Cabin'] = df['Cabin'].str[:1]

features = ['Cabin','Sex','Embarked']

for feature in features:

le = LabelEncoder()

le = le.fit(df[feature])

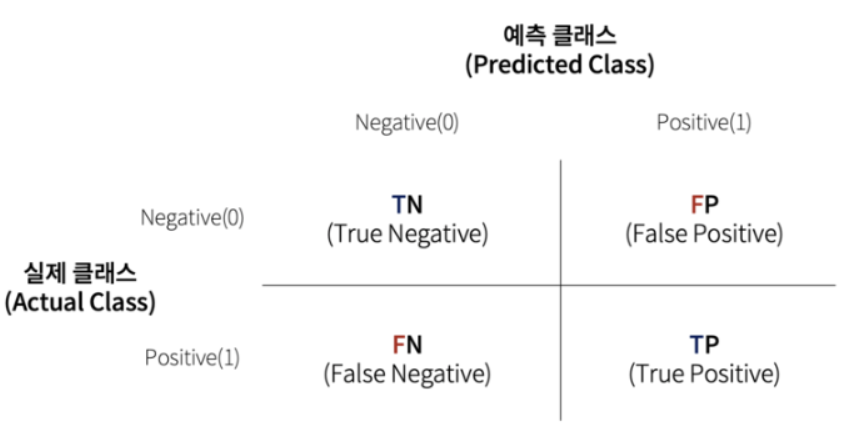
df[feature] = le.transform(df[feature])

return df

**Feature Scaling**

* 표준화: 데이터의 피처 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것을 의미한다
* 정규화: 서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 개념이다
* StandardScaler: 평균이 0이고 분산이 1인 정규 분포 형태로 변환
* MinMaxScaler: 데이터 값을 0과 1사이의 범위의 값으로 변환한다

분류 성능 평가 지표 정리

오차행렬(Confusion Matrix)

-> 이진 분류의 예측 오류가 얼마인지, 어떤 유형의 예측오류가 발생하는지 파악가능, => 정확도, 정밀도, 재현율 값 확인

정확도 = 예측 결과와 실제 값이 동일한 건수 / 전체 데이터수 = (TN+TP)/(TN+FP+FN+TP)

정밀도 = TP / (FP+TP) -> 예측을 Positive로 한 대상중에 예측과 실제 값이 Positive로 일치한 데이터의 비율

재현율 = TP / (FN+TP) 실제값이 Positive인 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻한다

1. 재현율이 상대적으로 더 중요한 지표인 경우, 실제 Positive 양성인 데이터 예측을 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우: > 암진단, 금융사기
2. 정밀도: 상대적으로 더 중요한 지표인 경우는 실제 Negative 음성인 데이터 예측을 Positive 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우: >스팸 메일 => 분류 하려는 업무의 임곗값을 조정해야 한다. (Trade-OFF)

임곗값이 - Classifier가 판단할때 기준. 분류 결정 임곗값이 낮아질수록 Positive로 예측할 확률이 높아져 재현율이 증가한다. -> 재현율의 FN이 떨어진다. 정밀도의 FP증가

from sklearn.metrics import accuracy\_score, precision\_score , recall\_score , confusion\_matrix, f1\_score

def get\_clf\_eval(y\_test , pred):

confusion = confusion\_matrix( Y\_test, pred)

accuracy = accuracy\_score(Y\_test , pred)

precision = precision\_score(Y\_test , pred)

recall = recall\_score(Y\_test , pred)

# F1 스코어 추가

f1 = f1\_score(Y\_test,pred)

print('오차 행렬')

print(confusion)

# f1 score print 추가

print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1:{3:.4f}'.format(accuracy, precision, recall, f1))

F1 Score: 정밀도 와 재현율을 결합한 지표이다. 정밀도와 재현율이 어느 한쪽으로 치우치지 않는 수치를 나타날때 상대적으로 높은 값을 가집니다.

F1 스코어의 공식은 2 \* precision \* recall / (precision + recall)

ROC곡선은 FPR(False Positive Rate)이 변할떄 TPR(True Positive Rate)이 어떻게 변하는지를 나타내는 곡선입니다. FPR을 X축으로 TPR을 Y축으로 잡으면 FPR의 변화에 따른 TPR의 변화가 곡선 형태로 나타난다

AUC(Area Under Curve) 값은 ROC 곡선 밑의 면적을 구한 것으로서 일반적으로 1에 가까울 수록 좋은 수치이다

* TPR은 TP / (FP + TP) 재현율(민감도),
* FPR은 Negative를 잘못 예측한 비율 = FP / (FP + TN)

***분류 알고리즘***

* 베이즈(Bayes) 통계와 생성 모델에 기반한 Naive Bayes
* 독립 변수와 종속변수의 선형 관계성에 기반한 Logistic Regression
* 데이터 균일도에 따른 규칙 기반의 결정 트리 Deicision Tree
* 개별 클래스간의 최대 분류 마진을 효과적으로 찾아주는 Support Vector Machine
* 근접 거리를 기준으로 하는 최소 근접 Nearest Neighbor
* 심층 연결 기반의 신경망 Neural Network
* 서로 다른 머신러닝 알고리즘을 결합한 Ensemble

정보 균일도 측정방법

* 정보이득: 엔트로피라는 개념을 기반. 데이터 집합의 혼잡도 의미. 서로 다른 값이 섞여 있으면 엔트로피가 높고 같은 값이 섞여 있으면 엔트로피가 낮다. 정보 이득 지수는 1에서 엔트로피 지수를 뺀 값이다.
* 지니 계수: 원래 경제학에서 불평등 지수를 나타낼 때 사용하는 계수 . 0이 평등, 1로 갈수록 불평등. 지니 계수가 낮을 수록 데이터 균일도 높은것으로 해석되어 낮은 속성 기준으로 분할한다.

앙상블(Ensemble Learning): 여러개의 분류기를 생성하고 그 예측을 결합함으로써 보다 정확한 최종 예측을 도출하는 기법.

* Voting
* Bagging : RandomForest
* Boosting : AdaBoosting, GradientBoositng, XGBoost, LightBGM

1. Voting: 서로 다른 알고리즘을 가진 분류기가 결합
2. Bagging: 각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행해 Voting 수행

Hardvoting vs SoftVoting

Hardvoting: 다수결

SoftVoting: classifier들의 class확률을 평균하여 결정

* predict\_proba메소드를 이용하여 class 별 확률 결정

# 개별 모델은 로지스틱 회귀와 KNN 임.

lr\_clf = LogisticRegression()

knn\_clf = KNeighborsClassifier(n\_neighbors=8)

# 개별 모델을 소프트 보팅 기반의 앙상블 모델로 구현한 분류기

vo\_clf = VotingClassifier( estimators=[('LR',lr\_clf),('KNN',knn\_clf)] , voting='soft' )

X\_train, X\_test, y\_train, y\_test = train\_test\_split(cancer.data, cancer.target,

test\_size=0.2 , random\_state= 156)

# VotingClassifier 학습/예측/평가.

vo\_clf.fit(X\_train , y\_train)

pred = vo\_clf.predict(X\_test)

print('Voting 분류기 정확도: {0:.4f}'.format(accuracy\_score(y\_test , pred)))

# 개별 모델의 학습/예측/평가.

classifiers = [lr\_clf, knn\_clf]

for classifier in classifiers:

classifier.fit(X\_train , y\_train)

pred = classifier.predict(X\_test)

class\_name= classifier.\_\_class\_\_.\_\_name\_\_

print('{0} 정확도: {1:.4f}'.format(class\_name, accuracy\_score(y\_test , pred)))

Voting 분류기 정확도: 0.9474

LogisticRegression 정확도: 0.9386

KNeighborsClassifier 정확도: 0.9386

#Bagging: RandomForestClassifier - Fast and High accuracy

#전체 데이터에서 일부가 중첩되게 샘플링된 데이터 세트. -> Bootstrapping 분할 방식

#n\_estimators: 결정 트리의 개수.

# max\_features: 결정 트리에 사용된 max\_features와 같다.

# max\_depth, min\_samples\_leaf-> 똑같이 적용

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import GridSearchCV

rf\_clf = RandomForestClassifier(n\_jobs = -1)

params = {'max\_depth':[4,6,8,10,12],

'min\_samples\_leaf':[4,8,10,16,32],

'n\_estimators':[100, 200]}

gs\_clf = GridSearchCV(rf\_clf, param\_grid=params, cv=2)

gs\_clf.fit(X\_train, y\_train)

preds = gs\_clf.predict(X\_test)

score = accuracy\_score(preds, y\_test)

print("score: ", score)

gs\_clf.best\_params\_

{'max\_depth': 6, 'min\_samples\_leaf': 8, 'n\_estimators': 100}

gs\_clf.best\_score\_

0.9560727258675323

gs\_clf.best\_index\_

12

Boosting: weaker leaner들을 순차적으로 학습예측하면서 잘못 예측한 데이터에 가중치 부여를 통해 오류를 개선해 나가면서 학습하는 방식이다

* GBM 가중치 업데이트를 경사하강법을 이용하는 것이 큰 차이이다.
* loss: 비용함수 지정
* learning\_rate = 학습률
* n\_estimators = weak leaner 개수

from xgboost import XGBClassifier

# scikit-learn wrapper 에서는

# learning\_rate, n\_estimators, min\_child\_weight, max\_depth, subsample

# reg\_lambda: l2규제 적용값, reg\_alpha: l1규제 적용값, colsample\_bytree: max\_features,

# scale\_pos\_weight: 특정 값으로 치우친 비대창한 클래스로 구성된 데이터 세트의 균형을 유지하기 위한 파라미터

# gamma; 트리의 리프 노드를 추가적으로 나눌지를 결정할 최소 손실 감소 값

# hyperparameter가 있다

from xgboost import XGBClassifier

from sklearn.model\_selection import GridSearchCV

evals = [(X\_test, y\_test)]

params = {'n\_estimators':[100,200,300,400],

'learning\_rate':[0.1,0.001,0.0001],

'max\_depth':[4,6,8,10,12],

'colsample\_bytree':[0.5,0.75]

}

xgb\_wrapper = XGBClassifier()

gr\_xgb = GridSearchCV(xgb\_wrapper, param\_grid=params)

gr\_xgb.fit(X\_train , y\_train, early\_stopping\_rounds=400,eval\_set=evals, eval\_metric="logloss", verbose=True)

w\_preds = gr\_xgb.predict(X\_test)

score = accuracy\_score(preds, y\_test)

print("score: ", score)

XGBoost 조기중단 기능(EArly Stopping)

* early\_stopping\_rounds: 더이상 비용 평가 지표가 감소하지 않는 최대 반복횟수
* eval\_metric: 반복 수행시 사용하는 비용 평가 지표
* eval\_set: 평가를 수행하는 별도의 검증 데이터 세트, 일반적으로 검증 데이터 세트에서 반복적으로 비용 감소 성능 평가

LightGBM: 더 빠르고, 더 작은 메모리 사용량, 카테고리형 피처의 자동 변환과 최적 분할.

* 균형 트리 분할 보다는 리프 중심 트리 분할 => 더 빠르게

1. Log변환: 왜곡된 분포도를 가진 데이터 세트를 비교적 정규분포에 가깝게 변환해주는 Feature Engineering
2. IQR(Inter Quantile Range), Outlier Removal

-> 박스 플롯, -> 1/4 분위수 ~ 3/4분위수 IQR 최소값: 1/4분위수에서 1.5*IQR을 뺀 지점 최댓값: 3/4 분위수에서 1.5*IQR값을 더한 지점 이상치는 최댓값, 최소값 /이상, 이하이다

1. UnderSampling, OverSampling; UnderSampling: 많은 레이블을 가진 데이터 세트를 적은 레이블을 가진 데이터 세트 수준으로 감소 샘플링 OverSampling: 적은 레이블을 가진 데이터 세트를 많은 레이블을 가진 데이터 세트 수준으로 증식
2. SMOTE(Synthetic Minority Over-Sampling Technique): K-최근접 이웃으로 데이터 신규 증식, Over sampling 한다.

amount\_n = np.log1p(df\_copy['Amount\_Scaled'])

card\_df.insert(0, 'Amount\_Scaled', amount\_n)

1. log를 적용한 이유는 데이터 분포의 왜도가 너무 커서 적용한 것입니다. StandardScaler(평균이 0, 표준편차가 1 로 변환)을 적용한 이유는 Recency, Frequency, Monetary 가 서로 다른 단위 값이라 이에 대한 공통 Scale을 적용하기 위한 것입니다. cust\_df[['Recency', 'Frequency', 'Monetary']].describe()를 보면 std 값이 각 컬럼별로 큰 차이가 있습니다.

보통 Kmeans는 단위가 다른 여러개의 feature들이 있을 경우 보통은 StandardScaler를 적용하는데, 이는 Kmeans가 일반적으로 거리기반(Euclidean distance) 기반으로 둥글게(등방성, isotropic)하게 데이터를 clustering 하는데 있어서 더 좋은 성능을 나타내기 때문입니다(그렇다고 무조건 좋아지지는 않습니다)

2.np.log1p와 standardscaler 이 두개를 같이 사용할 때 크게 주의해야할 부분은 없습니다.

그리고 원래값을 알기 위해서는 스케일링 했던 반대 순서대로 하면 됩니다.

StandardScaler.fit\_transform() 후 log1p하여 변환되었다면, 다시 이 값을 expm1 변환 후 StandardScaler.inverse\_transform( ) 하면 됩니다.

def get\_outlier(df=None, column=None, weight=1.5):

fraud = df[df['Class']==1][column]

quantile\_25 = np.percentile(fraud.values, 25)

quantile\_75 = np.percentile(fraud.values, 75)

iqr = quantile\_75 - quantile\_25

iqr\_weight = iqr\*weight

lowest\_val = quantile\_25 - iqr\_weight

highest\_val = quantile\_75+iqr\_weight

outlier\_index = fraud[(fraud<lowest\_val) | (fraud>highest\_val)].index

return outlier\_index

outlier\_index = get\_outlier(df=card\_df, column='V14', weight=1.5)

print(outlier\_index)

from imblearn.over\_sampling import SMOTE

smote = SMOTE(random\_state=0)

X\_train\_over, y\_train\_over = smote.fit\_sample(X\_train, y\_train)

**Basic** Stacking Model: 기반 모델들이 예측한 값들을 Stacking 형태로 만들어서 메타 모델이 이를 학습후 예측한다

#STacking Model knn -> kneighbor, rf - randomforest, dt- decisiontree, ada - adaboost

pred = np.array([knn\_pred, rf\_pred, dt\_pred, ada\_pred])

print(pred.shape)

# transpose를 이용해 행과 열의 위치 교환. 컬럼 레벨로 각 알고리즘의 예측 결과를 피처로 만듦.

pred = np.transpose(pred)

print(pred.shape)

**Feature Selection:** 모델을 구성하는 주요 피처들을 선택

* 불 필요한 다수의 피처들로 인해 모델 성능을 떨어뜨릴 가능성 제거
* 설명 가능한 모델이 될 수 있도록 피처들을 선별
* 피처 값의 분포, Null, 피처간 높은 상관도, 결정값과 의 독립성등을 고려
* Feature importance 기반

1. RFE(Recursive Feature Elimination): 모델 최초 학습후 Feature 중요도 선정, feature 중요도가 낮은 속성들을 차례대로 제거해가며 반복적으로 학습 수행, 최적 Feature 추출

-> 데이터가 작을때만 사용, 너무 오래걸림

1. SelectFromModel: 모델 최초 학습 후 선정된 Feature 중요도에 따라 평균/중앙값의 특정 비율이상인 Feature 들을 선택

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from sklearn.model\_selection import StratifiedKFold

from sklearn.feature\_selection import RFECV, RFE

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=1000, n\_features=25, n\_informative=3, n\_redundant=2, n\_repeated=0, n\_classes=8, n\_clusters\_per\_class=1, random\_state=0)

svc=SVC(kernel='linear')

rfecv=RFECV(estimator=svc, step=1, cv=StratifiedKFold(2), scoring='accuracy', verbose=2)

rfecv.fit(X, y)

print(rfecv.n\_features\_)

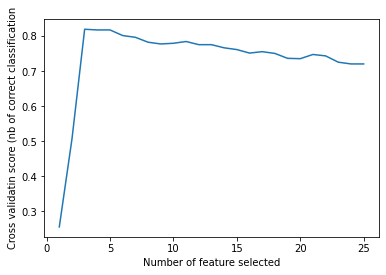
#Fitting estimator with 25 features.

#Fitting estimator with 24 features.

#Fitting estimator with 23 features.

#Fitting estimator with 22 features.

#Fitting estimator with 21 features.



#SelectFromModel

from sklearn.datasets import load\_diabetes

diabetes = load\_diabetes()

X, y = diabetes.data, diabetes.target

import matplotlib.pyplot as plt

import numpy as np

from sklearn.linear\_model import LassoCV

lasso = LassoCV().fit(X,y)

importance = np.abs(lasso.coef\_)

feature\_names = np.array(diabetes.feature\_names)

plt.bar(height = importance, x=feature\_names)

plt.title('Feature importance via coefficients')

plt.show()

from sklearn.feature\_selection import SelectFromModel

from time import time

threshold = np.sort(importance)[-3] + 0.01

print('threshold: ', threshold)

sfm = SelectFromModel(lasso, threshold =threshold).fit(X, y)

print('feature selected by selectfromModel: ', f"{feature\_names[sfm.get\_support()]}")

threshold: 521.748542606749

feature selected by selectfromModel: ['s1' 's5']

Permutation importance 개요

* 특정 피처들의 값을 완전히 변조했을 때 모델 성능이 얼마나 저하되는지를 기준으로 해당 피처의 중요도를 산정.
* 반복적으로 변조한뒤 피처의 중요도를 평균적으로 산정

from sklearn.inspection import permutation\_importance

r = permutation\_importance(model, X\_val, y\_val,n\_repeats=30,random\_state=0)

for i in r.importances\_mean.argsort()[::-1]:

if r.importances\_mean[i] - 2 \* r.importances\_std[i] > 0:

print(f"{diabetes.feature\_names[i]:<8}"f"{r.importances\_mean[i]:.3f}"f" +/- {r.importances\_std[i]:.3f}")

s5 0.204 +/- 0.050

bmi 0.176 +/- 0.048

bp 0.088 +/- 0.033

sex 0.056 +/- 0.023

Feature importance는

* 최적 tree 구조를 만들기 위한 피처들의 impurity가 중요 기준. 결정값과 관련이 없어도 높아질수 있다.
* 학습 데이터를 기반으로 생성되기 때문에 테스트 데이터에서는 다를수가 있다.
* number형의 높은 cardinality feature에 biased 되어있다.

회귀: 데이터 값이 평균과 같은 일정한 값으로 돌아가려는 경향을 이용한 통계학 기법

* 여러개의 독립변수와 한 개의 종속 변수간의 상관관계를 모델링하는 기법을 통칭한다.

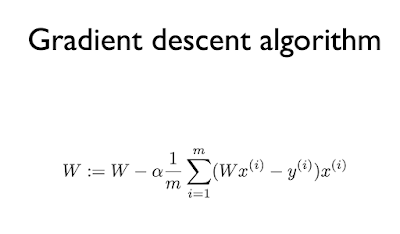
머신러닝 회귀 예측의 핵심은 주어진 피처와 결정 값 데이터 기반에서 학습을 통해 최적의 회귀계수를 찾아내는 것이다.

* 독립변수 개수 1개 : 선형 회귀
* 독립변수 개수 여러개: 다중 회귀, 비선형 회귀

선형 회귀 종류:

* 일반 선형 회귀: 규제를 적용하지 않은 모델
* Ridge: L2규제
* Lasso: L1규제
* ElasticNet: L2, L1 규제 함께 결합
* LogisticRegression: 사실은 분류에 속함

RSS: 오류 값의 제곱을 구해서 더하는 방식. 미분등의 계산을 편리하게 하기 위해서 RSS 방식으로 오류합을 구합니다. => 비용함수 -> 미분값이 계속 감소하는 방향으로 순차적으로 w를 업데이트 한다. 더이상 기울기가 감소하지 않는 지점을 비용함수가 최소인 지점으로 간주하고 그때의 w를 반환한다



#Gradient descent

def get\_weight\_updates(w1, w0, X, y, learning\_rate=0.01):

N = len(y)

w1\_update = np.zeros\_like(w1)

w0\_update = np.zeros\_like(w0)

y\_pred = np.dot(X, w1.T) + w0

diff = y-y\_pred

w0\_factors = np.ones((N,1))

w1\_update = -(2/N)\*learning\_rate\*(np.dot(X.T, diff))

w0\_update = -(2/N)\*learning\_rate\*(np.dot(w0\_factors.T, diff))

return w1\_update, w0\_update

def stochastic\_gradient\_descent\_steps(X, y, batch\_size=10, iters=1000):

w0 = np.zeros((1,1))

w1 = np.zeros((1,1))

prev\_cost = 100000

iter\_index = 0

for ind in range(iters):

np.random.seed(ind)

stochastic\_random\_index = np.random.permutation(X.shape[0])

sample\_X = X[stochastic\_random\_index[0:batch\_size]]

sample\_y = y[stochastic\_random\_index[0:batch\_size]]

w1\_update, w0\_update = get\_weight\_updates(w1, w0, sample\_X, sample\_y, learning\_rate=0.01)

w1 = w1-w1\_update

w0 = w0 - w0\_update

return w1, w0

**회귀 평가 지표**

* MAE(Mean Absolute Error(MAE)): 실제값과 예측값의 차이를 절대값으로 변환해 평균한거
* MSE(Mean Squared Error(MSE)): 실제값과 예측값의 차이를 제곱해 평균한 것
* MSLE: MSE에 로그 적용, 결정값이 클수록 오류값도 커지기 때문에 일부 큰 오류값들로 인해 전체 오류 값이 커지는 것을 막아줌
* RMSE: MSE에 루트를 씌운것
* RMSLE: RMSE에 로그 적용
* R^2 = 분산 기반으로 예측 성능 평가, 1에 가까울수록 정확도가 높다

평가 API

* MAE: metrics\_mean\_absolute\_error
* scoring 파라미터 값: neg\_mean\_absolute\_error: 음수값을 반환하는 이유는 scoring함수가 score가 클수록 좋은 평가 결과로 자동 평가한다. 고로 음수를 만들어 작은 오류값이 더 큰 숫자로 인식하게 만든다

from sklearn.model\_selection import cross\_val\_score

y\_target = bostonDF['PRICE']

X\_data = bostonDF.drop(['PRICE'],axis=1,inplace=False)

lr = LinearRegression()

# cross\_val\_score( )로 5 Fold 셋으로 MSE 를 구한 뒤 이를 기반으로 다시 RMSE 구함.

neg\_mse\_scores = cross\_val\_score(lr, X\_data, y\_target, scoring="neg\_mean\_squared\_error", cv = 5)

rmse\_scores = np.sqrt(-1 \* neg\_mse\_scores)

avg\_rmse = np.mean(rmse\_scores)

# cross\_val\_score(scoring="neg\_mean\_squared\_error")로 반환된 값은 모두 음수

print(' 5 folds 의 개별 Negative MSE scores: ', np.round(neg\_mse\_scores, 2))

print(' 5 folds 의 개별 RMSE scores : ', np.round(rmse\_scores, 2))

print(' 5 folds 의 평균 RMSE : {0:.3f} '.format(avg\_rmse))

5 folds 의 개별 Negative MSE scores: [-12.46 -26.05 -33.07 -80.76 -33.31]

5 folds 의 개별 RMSE scores : [3.53 5.1 5.75 8.99 5.77]

5 folds 의 평균 RMSE : 5.829

다항 회귀는 선형회귀, 회귀를 나누는 기준은 회귀 계수가 선형인지 비선형인지에 따라 나누어진다.

다항회귀: PolynomialFeatures로 원본 단항피처들을 다항 피처들로 변환한 데이터 세트에 LinearRegression객체를 적용하여 다항회귀 기능을 제공 => 이를 편하게 만들어주는 클래스는 Pipeline

규제 선형 회귀: 최적 모델을 위한 Cost 함수 구성요소 = 학습 데이터 잔차 오류 최소화 + 회귀계수 크기 제어 alpha값을 조절한다

1. alpha=0, 비용함수는 Min(RSS(W))
2. alpha=무한대, alpha\*회귀계수가 무한대 -> 비용함수는 W를 0에 가깝게 최소화

규제: alpha값으로 페널티를 부여해 회귀 계수 값의 크기를 감소시켜 과적합 개선

1. L2규제(Ridge):alpha\*W^2와 같이 W의 제곱에 대해 페널티를 부여
2. L1규제(Lasso): alpha\*W, W의 절댓값에 대해 페널티 부여 (영향력이 크지 않은 회귀 계수는 0)
3. ElasticNet: L2 + L1규제를 함께 결합한 모델

선형 회귀 모델을 위한 데이터 변환: 정규분포 형태를 매우 선호한다.

* 타깃값 변환: 반드시 정규분포, 로그 변환!
* 피처값 변환:

1. StandardScaler, MinMaxScaler(예측 성능 향상 어려움)
2. 다항 특성을 적용하여 변환(과적합이슈)
3. log변환

* 데이터 인코딩은 원-핫 인코딩 적용

로지스틱 회귀: 선형 회귀 방식을 분류에 적용한 알고리즘이다. 시그모이드 함수 최적선을 찾고 이 시그모이드 함수의 반환값을 확률로 간주해 확률에 따라 분류를 결정한다. (0과 1)

회귀트리: 분류뿐만 아니라 회귀도 가능하다

* Cart 회귀 트리는 분류와 유사하게 분할을 하며, 분할 기준은 RSS가 최소가 될 수 있는 기준을 학습, 예측한다

1. RSS를 최소화 하는 규칙 기준에 따라 분할
2. 최종 분할된 영역에 있는 데이터들의 평균값들로 학습/예측

DecisionTreeRegressor/GradientboostingRegressor 등

from datetime import date, datetime, time

#datetime object로 바꾸고 쉽게 year, month, day, hour등을 뽑아낸다

bike\_df['datetime'] = bike\_df.datetime.apply(pd.to\_datetime)

bike\_df['year'] = bike\_df.datetime.apply(lambda x:x.year)

bike\_df['month'] = bike\_df.datetime.apply(lambda x:x.month)

bike\_df['day'] = bike\_df.datetime.apply(lambda x:x.day)

bike\_df['hour'] = bike\_df.datetime.apply(lambda x:x.hour)

def rmsle(y, pred):

log\_y = np.log1p(y)

log\_pred = np.log1p(pred)

squared\_error = (log\_y - log\_pred)\*\*2

rmsle = np.sqrt(np.mean(squared\_error))

return rmsle

def rmse(y,pred):

return np.sqrt(mean\_squared\_error(y,pred))

def evaluate\_regr(y,pred):

rmsle\_val = rmsle(y,pred)

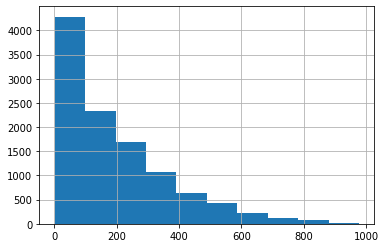
rmse\_val = rmse(y,pred)

mae\_val = mean\_absolute\_error(y,pred)

print('RMSLE: {0:.3f}, RMSE: {1:.3F}, MAE: {2:.3F}'.format(rmsle\_val, rmse\_val, mae\_val))

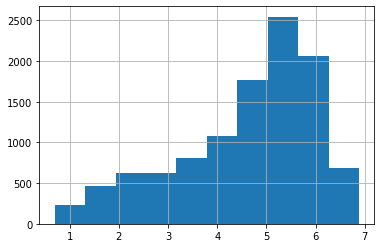
#1 회귀에서 제일 중요한것은 target값이 정규분포인가를 확인해야한다

y\_target.hist()



y\_log\_transform = np.log1p(y\_target)

y\_log\_transform.hist()



y\_target\_log = np.log1p(y\_target)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X\_data, y\_target\_log, test\_size=0.3)

lr=LinearRegression()

lr.fit(x\_train, y\_train)

pred = lr.predict(x\_test)

#log1p해주고 나서 다시 test할때는 expm1으로 돌아와준다

y\_test\_exp = np.expm1(y\_test)

pred\_exp = np.expm1(pred)

evaluate\_regr(y\_test\_exp, pred\_exp)

from scipy.stats import skew

#왜곡도

features\_index = house\_df.dtypes[house\_df.dtypes != 'object'].index

skew\_features = house\_df[features\_index].apply(lambda x:skew(x))

skew\_features\_top = skew\_features[skew\_features >1]

print(skew\_features\_top.sort\_values(ascending=False))

MiscVal 24.451640

PoolArea 14.813135

LotArea 12.195142

3SsnPorch 10.293752

LowQualFinSF 9.002080

KitchenAbvGr 4.483784

BsmtFinSF2 4.250888

ScreenPorch 4.117977

BsmtHalfBath 4.099186

EnclosedPorch 3.086696

MasVnrArea 2.673661

LotFrontage 2.382499

# 차원축소 PCA

차원이 커질수록 데이터 포인트들간 거리가 크게 늘어나고 데이터가 희소화된다

* 상관관계가 높아 다중 공선성 문제로 예측 성능 저하 가능성  
  차원축소의 장점 - 학슴 시간 절약, 성능 향상, 시각적으로 쉽게 데이터 패턴 인지

1. 피처 선택: 불필요한 피처는 아예 제거, 데이터의 특징이 잘 나타내는 주요 피처만 선택
2. 피처 추출: 기존 피처를 저차원의 중요피처로 압축해서 추출. 새로운 피처로 추출

잠재적인 Latent요소 추출

* 추천엔진, 이미지 분류 및 변환, 문서 토픽 모델링

PCA: 고차원의 원본 데이터를 저차원의 부분 공간으로 투영하여 데이터를 축소(10 차원-> 2차원)

* 원본 데이터 변동성이 가장 큰 방향으로 순차적으로 축들을 생성하고 이렇게 생성된 축으로 데이터를 투영하는 방식
* 가장 큰 데이터 변동성을 기반으로 첫 번째 벡터 축을 생성하고 두번째 축은 첫번째 축을 제외하고 그다음으로 큰 축을 설정하는데 이는 첫번째 축에 직각이 되는 벡터축. 또 세번재는 두번째의 직각.

PCA변환

1. 원본 데이터의 공분산 행렬 추출
2. 공분산 행렬을 고유벡터와 고유값 분해
3. 원본 데이터를 고유 벡터로 선형 변환
4. PCA 변환 값 도출

공분산 행렬: 여러변수와 관련된 공분산을 포함하는 정방형 행렬이며 대칭행렬이다.

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

pca.fit(iris\_scaled)

iris\_pca = pca.transform(iris\_scaled)

|  | **pca\_component\_1** | **pca\_component\_2** | **target** |
| --- | --- | --- | --- |
| **0** | -2.264703 | 0.480027 | 0 |
| **1** | -2.080961 | -0.674134 | 0 |
| **2** | -2.364229 | -0.341908 |  |

LDA: 선형 판별 분석법

* 지도학습의 분류에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하며서 차원을 축소. 입력 데이터의 결정값 클래스를 최대한으로 분리할수 있는 축을 찾는다

특이값 분해 - SVD(Singular Value Decomposition)

* 정방행렬 뿐만아니라 행과 열의 크기가 다른 m\*n행렬도 분해 가능
* 모든 특이 벡터는 서로 직교하는 성질을 가진다.
* 시그마는 대각행렬이며, 행렬의 대각에 위치한 값만 0이 아니고 나머지 위치의 값은 모두 0이다.

A = U Σ V T

(1)

여기서 네 행렬( A , U , Σ , V )의 크기(혹은 차원)와 성질은 다음과 같다.

A : m × n rectangular matrix U : m × m orthogonal matrix Σ : m × n diagonal matrix V : n × n orthogonal matrix

orthogonal matrix는 다음의 성질을 만족하는 행렬이다.

U 가 orthogonal matrix라고 한다면,

U U T = U T U = I

(2)

이에 따라, U − 1 = U T 라는 사실도 부가적으로 확인된다.

또, diagonal matrix는 다음과 같은 성질을 만족하는 행렬이다.

Σ 가 diagonal matrix라고 한다면 Σ 의 대각성분을 제외한 나머지 원소의 값은

모두 0이다.

SVD는 차원 축소를 위한 행렬분해를 통해 Latent Factor를 찾을수 있는데 이렇게 찾아진 Latent Factor는 많은 분야에 활용. 행렬 분해된뒤 다시 분해된 행렬을 이용하여 원복된 데이터 셋은 잡음이 제거된 형태로 재구성될 수 있다.

from sklearn.preprocessing import StandardScaler

# iris 데이터를 StandardScaler로 변환

scaler = StandardScaler()

iris\_scaled = scaler.fit\_transform(iris\_ftrs)

# 스케일링된 데이터를 기반으로 TruncatedSVD 변환 수행

tsvd = TruncatedSVD(n\_components=2)

tsvd.fit(iris\_scaled)

iris\_tsvd = tsvd.transform(iris\_scaled)

# 스케일링된 데이터를 기반으로 PCA 변환 수행

pca = PCA(n\_components=2)

pca.fit(iris\_scaled)

iris\_pca = pca.transform(iris\_scaled)

# TruncatedSVD 변환 데이터를 왼쪽에, PCA변환 데이터를 오른쪽에 표현

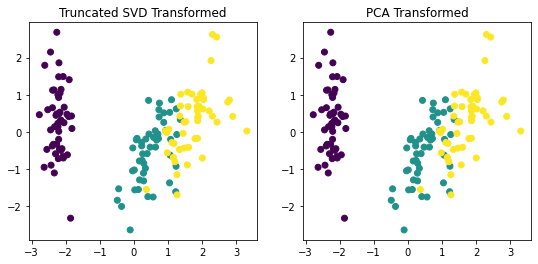
fig, (ax1, ax2) = plt.subplots(figsize=(9,4), ncols=2)

ax1.scatter(x=iris\_tsvd[:,0], y= iris\_tsvd[:,1], c= iris.target)

ax2.scatter(x=iris\_pca[:,0], y= iris\_pca[:,1], c= iris.target)

ax1.set\_title('Truncated SVD Transformed')

ax2.set\_title('PCA Transformed')



from sklearn.decomposition import NMF

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

%matplotlib inline

iris = load\_iris()

iris\_ftrs = iris.data

nmf = NMF(n\_components=2)

nmf.fit(iris\_ftrs)

iris\_nmf = nmf.transform(iris\_ftrs)

plt.scatter(x=iris\_nmf[:,0], y= iris\_nmf[:,1], c= iris.target)

plt.xlabel('NMF Component 1')

plt.ylabel('NMF Component 2')

군집화: 데이터 포인트들을 별개의 군집으로 그룹화 하는것. 서로 다른 군집들이 상이성을 가지도록 분류한다

* 고객 맞춤 세분화, Image검출, 이상 검출등에 활용된다

알고리즘

* K-Means
* Mean Shift
* Gaussian Mixture Model: 어떤 정규분포
* DBSCAN: 데이터의 밀도에 따라 군집화

1. K-Means Clustering: 군집 중심점 기반 클러스터링

* 속성의 개수가 매우 많을 경우 군집화 정확도가 떨어져 PCA로 차원 축소를 적용해야할수도 있다 + 이상치 데이터에 취약
* n\_clusters: 군집화 개수
* init: 군집 중심점의 좌표를 설정할 방식, 보통은 임의로 중심을 설정하지 않고 일반적으로 k-means++ 방식으로 최초 설정
* max\_iter: 최대 반복 횟수.

실루엣 분석: 각 군집간의 거리가 얼마나 효율적으로 분리돼 있는지를 나타낸다.

* 실루엣 계수를 기반으로 한다: 해당 데이터가 같은 군집 내의 데이터와 얼마나 가깝게 군집화돼있고 다른 군집 데이터와 얼마나 멀리 분리되어 있는지

sklearn.metrics.silhouette\_samples(X, labels, metric='euclidean', \*\*kwds)

* 0~1사이의 값을 가지며, 1에 가까울수록 좋다
* 하지만 실루엣 계수의 평균값과 더불어 개별 군집의 평균값의 편차가 크지 않아야 한다

Mean Shift: KDE(Kernel Density Estimation)를 이요하여 데이터 포인트들이 데이터 분포가 높은곳으로 이동하면서 군집화를 수행

* Mean shift 클러스터링은 데이터 포인트의 밀집된 영역을 찾기 위해 시도하는 슬라이딩 윈도우 기반 알고리즘입니다. 이는 중심점에 대한 후보를 슬라이딩 윈도우 내의 포인트의 평균으로 업데이트하여 작동하는 각 그룹 / 클래스의 중심점을 찾는 것이 목표라는 것을 의미하는 centroid 기반 알고리즘입니다. 이러한 후보 윈도우는 후 처리 단계에서 필터링되어 거의 중복을 제거하여 최종 세트의 중심점과 해당 그룹을 형성합니다.

KDE: 커널함수를 통해 어떤 변수의 확률밀도 함수를 추정하는 방식. 관측된 데이터 각각에 커널함수를 적용한 값을 모두 더한뒤 데이터 건수로 나누어서 확률 밀도 함수를 추정

**확률 밀도 추정 방법**

* 모수적 (PArametric) 추정: 데이터가 특정 데이터 분포를 따른다는 가정하에 데이터 분포를 찾는 방법 - Gaussian Mixture
* 비모수적(Non-Parametric) 추정: 데이터가 특정 분포를 따르지 않는다는 가정 하에서 밀도를 추정. 관측된 데이터만으로 확률 밀도를 찾는 방법 - KDE

Bandwidth에 따른 KDE변화 : Mean Shift 는 Bandwidth가 클수록 적은 수의 클러스터링 중심점을, Bandwidth가 작을수록 많은 수의 클러스터링 중심점을 가지게 된다.

import pandas as pd

clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])

clusterDF['target'] = y

# estimate\_bandwidth()로 최적의 bandwidth 계산

best\_bandwidth = estimate\_bandwidth(X, quantile=0.2)

meanshift= MeanShift(bandwidth=1)

cluster\_labels = meanshift.fit\_predict(X)

print('cluster labels 유형:',np.unique(cluster\_labels))

GMM - 거리기반 K-Means의 문제점: 중심점 기반으로 거리적으로 퍼져있지 않은 데이터 분포를 가지지 않는 데이터 세트에 대해서는 효율적인 군집화가 어렵다

* 데이터가 여러개의 다른 가우시안 분포를 가지는 모델로 가정하고 군집화 수행한다. 어떤 정규분포에 속하는지.

GMM의 모수 추정: 개별 정규 분포들의 평균과 분산, 그리고 데이터가 특정 정규 분포에 해당될 확률을 추정

1. Expectation: 개별 데이터 각각에 대해서 특정 정규분포에 소속될 확률을 구하고 가장 높은 확률을 가진 정규분포에 소속
2. Maximization: 데이터들이 Expectation으로 특정 정규분포로 소속되면 다시 해당 정규분포의 평균과 분산을 구함.

* 더이상 변경되지 않으면 최종 군집화 결정. 아니면 반복

from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n\_components=3).fit(iris.data)

gmm\_cluster\_labels = gmm.predict(iris.data)

iris\_data['gmm\_cluster'] = gmm\_cluster\_labels

iris\_result = iris\_data.groupby(['target'])['gmm\_cluster'].value\_counts()

iris\_result

target gmm\_cluster

0 2 50

1 0 45

1 5

2 1 50

Name: gmm\_cluster, dtype: int64

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=3).fit(iris.data)

kmeans\_cluster\_labels = kmeans.predict(iris.data)

iris\_data['kmeans\_cluster'] = kmeans\_cluster\_labels

iris\_result = iris\_data.groupby(['target'])['kmeans\_cluster'].value\_counts()

iris\_result

target kmeans\_cluster

0 0 50

1 1 48

2 2

2 2 36

1 14

DBSCAN: 특정 공간 내에 데이터 밀도 차이를 기반 알고리즘으로 하고 있어서 복잡한 기하학적 분포도를 가진 데이터 세트에 대해서도 군집화를 잘 수행한다

* 사용자가 군집 개수를 지정할 수 없다
* 밀도가 자주 변하거나 아예 변하지 않으면 군집화 성능이 떨어진다
* 피처의 개수가 많으면 군집화 성능이 떨어진다ᴬ

구성요소

1. 입실론(epsilon)으로 표기하는 주변 영역: 개별 데이터를 중심으로 입실론 반경을 가지는 원형의 영역
2. 최소 데이터 개수(min points): 개별 데이터의 입실론 주변 영역에 포함되는 타 데이터의 개수
3. 핵심(Core) 포인트, 이웃 포인트, 경계(Border) 포인트, 잡음 포인트

* eps: 입실론 주변 영역의 반경
* min\_samples: 핵심 포인트가 되기 위해 입실론 주변 영역 내에 포함되어야 할 데이터의 최소 개수

방법: 점을 중심으로 epsilon 반경내에 minPts 이상수의 점이 있으면 그 점을 중심으로 군집이 되고 그 점을 core point라고 한다. Core point 가 서로 다른 core point의 군집의 일부가 되면 그 군집을 서로 연결되어 있다고 하고 하나의 군집으로 연결을 한다.

군집에는 속하지만, 스스로 core point가 안되는 점을 border point라고 하고, 주로 클러스터의 외곽을 이루는 점이 된다.

그리고 어느 클러스터에도 속하지 않는 점은 Noise point가 된다.

from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.6, min\_samples=6, metric='euclidean')

dbscan\_labels = dbscan.fit\_predict(iris.data)

iris\_data['dbscan\_cluster'] = dbscan\_labels

iris\_result = iris\_data.groupby(['target'])['dbscan\_cluster'].value\_counts()

print(iris\_result)

target dbscan\_cluster

0 0 49

-1 1

1 1 46

-1 4

2 1 44

-1 6

Name: dbscan\_cluster, dtype: int64

### Log 변환을 통해 데이터 변환

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score, silhouette\_samples

# Recency, Frequecny, Monetary 컬럼에 np.log1p() 로 Log Transformation

cust\_df['Recency\_log'] = np.log1p(cust\_df['Recency'])

cust\_df['Frequency\_log'] = np.log1p(cust\_df['Frequency'])

cust\_df['Monetary\_log'] = np.log1p(cust\_df['Monetary'])

# Log Transformation 데이터에 StandardScaler 적용

X\_features = cust\_df[['Recency\_log','Frequency\_log','Monetary\_log']].values

X\_features\_scaled = StandardScaler().fit\_transform(X\_features)

kmeans = KMeans(n\_clusters=3, random\_state=0)

labels = kmeans.fit\_predict(X\_features\_scaled)

cust\_df['cluster\_label'] = labels

print('실루엣 스코어는 : {0:.3f}'.format(silhouette\_score(X\_features\_scaled,labels)))

텍스트 분석

1. Text Classification
2. Sentiment Analysis
3. Summarization
4. 텍스트 군집화와 유사도 측정

Process: Text 문서 -> Feature Vectorization -> 학습

패키지

* NLTK
* Gensim
* SpaCy

텍스트 전처리

* Cleansing: 불필요한 문자 기호등을 사전에 제거
* Tokenization: 문장 토큰화, 단어 토큰화, n-gram
* 필터링/스톱워드 제거/철자수정: 불필요한 단어나 분석에 큰의미가 없는 단어 그리고 잘못된 철자 수정
* Stemming/Lemmatization: 어근 추출, 의미론적 기반에서 단어 원형 찾아줌

n-gram: 연속된 n개의 단어를 하나의 토큰화 단위로 분리. n개 단어 크기 윈도우를 만들어 문장의 처음부터 오른쪽으로 움직이면서 토큰화를 수행

텍스트의 피처 벡터화 유형

1. Bag of Words(BOW): Document Term Matrix: 개별 문서를 단어들의 횟수나 정규화 변환된 횟수로 표현

* 문맥이나 순서를 무시하고 일괄적으로 단어에 대해 빈도 값을 부여해 피처값 추출
* 단점: 문맥 의미 반영 문제, 희소 행렬 문제

BOW 피처 벡터화 유형

* 단순 카운트 기반의 벡터화: Count값이 높을수록 중요한 단어
* TF-IDF 벡터화: 언어의 특성상 문장에서 자주 사용될수 밖에 없는 단어까지 높은 값을 부여하게 된다. 모든 문서에서 전반적으로 자주 나타나는 단어에 대해서는 페널티를 준다. => TF(Term Frequency): 문서에서 해당단어가 얼마나 나왔는지 => DF(Document Frequency): 해당 단어가 몇개의 문서에서 나타났는지 => IDF(Inverse Document Frequency): DF의 역수로서 전체 문서수/DF

1. Word Embedding(Word2Vec): 개별 단어를 문맥을 가지는 N차원 공간에 벡터로 표현

#CountVectorizer

# max\_df: 전체 문서에 걸쳐서 너무 높은 빈도수를 가지는 단어 피처를 제외하기 위한 파라미터

# 1. max\_df =100 이면 전체 문서에 걸쳐 100개 이하로 나타나는 단어만 피처로 추출

# 2. max\_df = 0.95는 0~95%까지의 단어만 피처로 추출

# min\_df: 너무 낮은 빈도수를 가지는 단어피처를 제외

# 1. min\_df = 2: 전체 문서에 걸쳐서 2번 이하

# 2. min\_df = 0.02 하위 2%이하 빈도수 단어는 추출 X

# max\_features: 피처로 추출하는 피처의 개수를 제한하며 정수로 값을 지정

# max\_features = 2000, 가장 높은 빈도를 가지는 단어 순으로 정렬해 2000개까지만 피처로 추출

# stop\_wrods = 'english'로 하면 영어의 스톱 워드로 지정된 단어는 추출 제외

# ngram\_range: (1,1)로 지정하면 토큰화된 단어를 1개씩 추출

# ngram\_range= (1,2)이면 단어를 1개씩 순서대로 2개씩 묶어서 피처로 추출

from sklearn.feature\_extraction.text import CountVectorizer

cnt\_vect = CountVectorizer(ngram\_range=(1,2))

cnt\_vect.fit(text)

ftr\_vect = cnt\_vect.transform(text)

print(type(ftr\_vect), ftr\_vect.shape)

print(cnt\_vect.vocabulary\_)

희소 행렬의 저장 변환 형식: COO형식: Coordinate(좌표) 방식을 의미하며 0이 아닌 데이터만 별도의 배열에 저장하고 그 데이터를 가리키는 행과 열의 위치를 별도의 배열에 저장 CSR 형식: COO형식이 위치 배열값을 중복적으로 가지는 문제를 해결한 방식. 일반적으로 CSR>COO Scipy의 coo\_matrix(), csr\_matrix()

from sklearn.feature\_extraction.text import CountVectorizer

cnt\_vect = CountVectorizer()

cnt\_vect.fit(X\_train)

#train으로 fit을 한다음에 train data와 test data의 feature 개수를 맞춰줘야지 predict가 가능해진다

X\_train\_cnt\_vect = cnt\_vect.transform(X\_train)

X\_test\_cnt\_vect = cnt\_vect.transform(X\_test)

print(X\_train\_cnt\_vect.shape, X\_test\_cnt\_vect.shape)

from sklearn.feature\_extraction.text import TfidfVectorizer

tf\_vect = TfidfVectorizer(stop\_words='english', ngram\_range=(1,2))

tf\_vect.fit(X\_train)

X\_train\_tf\_vect = tf\_vect.transform(X\_train)

X\_test\_tf\_vect = tf\_vect.transform(X\_test)

from sklearn.linear\_model import LogisticRegression

lr = LogisticRegression()

lr.fit(X\_train\_tf\_vect, y\_train)

pred = lr.predict(X\_test\_tf\_vect)

print(accuracy\_score(pred, y\_test))

from sklearn.pipeline import Pipeline

pipeline = Pipeline([

('tfidf\_vect', TfidfVectorizer(stop\_words='english')),

('lr\_clf', LogisticRegression())

])

# Pipeline에 기술된 각각의 객체 변수에 언더바(\_)2개를 연달아 붙여 GridSearchCV에 사용될

# 파라미터/하이퍼 파라미터 이름과 값을 설정. .

params = { 'tfidf\_vect\_\_ngram\_range': [(1,1), (1,2), (1,3)],

'tfidf\_vect\_\_max\_df': [100, 300, 700],

'lr\_clf\_\_C': [1,5,10]

}

# GridSearchCV의 생성자에 Estimator가 아닌 Pipeline 객체 입력

grid\_cv\_pipe = GridSearchCV(pipeline, param\_grid=params, cv=3 , scoring='accuracy',verbose=1)

grid\_cv\_pipe.fit(X\_train , y\_train)

print(grid\_cv\_pipe.best\_params\_ , grid\_cv\_pipe.best\_score\_)

pred = grid\_cv\_pipe.predict(X\_test)

print('Pipeline을 통한 Logistic Regression 의 예측 정확도는 {0:.3f}'.format(accuracy\_score(y\_test ,pred)))

감성분석: 문서의 주관적인 감성/의견/감정/기분등을 파악하기 위한 방법, 소셜미디어, 여론조사 등에서 활용.

* 지도 학습 기반의 분석
* 감성 어휘 사전을 이용한 분석: 감성 수치 계산

감성 어휘 사전 기반의 감성 분석

* SentiWordNet: 감성지수

1. 문서 -> 문장 분해
2. 문장 -> 단어 토큰화 품사 태깅
3. 단어 기반으로 synset 객체와 senti\_synset 객체 생성
4. Senti\_synset 에서 긍정 감성/부정 감성 지수 구하고 이를 모두 합산해 특정 임계치 값 이상일때 긍정 감성으로, 그렇지 않을 때는 부정 감성으로 결정

* VADER  
  소셜 미디어의 감성 분석 용도로 만들어진 Lexicon SentimentIntensityAnalyzer 클래스를 이용해 감성 분석 제공 polarity\_scores() 메서드 호출 감성 점수 구한다 compound score 를 기반으로 결정한다
* Pattern

import nltk

nltk.download('vader\_lexicon')

from nltk.sentiment.vader import SentimentIntensityAnalyzer

senti\_analyzer = SentimentIntensityAnalyzer()

senti\_scores = senti\_analyzer.polarity\_scores(review\_df['review'][0])

print(senti\_scores)

def vader\_polarity(review,threshold=0.1):

analyzer = SentimentIntensityAnalyzer()

scores = analyzer.polarity\_scores(review)

# compound 값에 기반하여 threshold 입력값보다 크면 1, 그렇지 않으면 0을 반환

agg\_score = scores['compound']

final\_sentiment = 1 if agg\_score >= threshold else 0

return final\_sentiment

토픽 모델링: 문서들에 잠재되어 있는 공통된 토픽들을 추출해 내는 기법을 의미한다. 공통된 유사성을 도출한다는 측면에서 문서 군집화/유사도와 비슷한 기법 일수도 있지만 토픽 모델링은 문서들이 가지는 주요 토픽의 분포도와 개별 토픽이 어던 의미인지를 제공.

알고리즘

* LSA, NMF: 행렬분해 기반 토픽 모델링
* pLSA, LDA 확률 기반의 토픽 모델링

LDA: 관찰된 문서내의 단어들을 이용하여 베이즈 추론을 통해 잠재된 문서내 토픽분포와 토픽별 단어 분포를 추론. LDA베이즈 추론의 사전 확률분포로 사용되는 것이 디리클레 분포이다.

lda = LatentDirichletAllocation(n\_components=8)

lda.fit(feat\_vect)

def display\_topics(model, feature\_names, no\_top\_words):

for topic\_index, topic in enumerate(model.components\_):

print('Topic #', topic\_index)

topic\_word\_indexes = topic.argsort()[::-1]

top\_indexes = topic\_word\_indexes[:no\_top\_words]

feature\_concat = ' '.join([feature\_names[i] + '\*' + str(round(topic[i],1)) for i in top\_indexes])

print(feature\_concat)

#print(topic)

feature\_names = count\_vect.get\_feature\_names()

display\_topics(lda, feature\_names, 15)

문서 군집화: 비슷한 텍스트 구성의 문서를 군집화 하는것입니다.

from nltk.stem import WordNetLemmatizer

#먼저 훈련 말뭉치의 각 단어를 해당 어간으로 수정 한 다음 새 말뭉치에 대해서만 훈련하여

#품사 태그 지정 프로젝트에 NLTK WordNet Lemmatizer를 사용하고 있습니다.

#그러나 lemmatizer가 예상대로 작동하지 않는다는 것을 알았습니다.

#예를 들어 loves 라는 단어는 올바른 love 로 표현되지만 loving 이라는 단어는 이후에도

#loves 로 남아 있습니다. lemmatization. 여기서 loving 은 "I 'm loving it"문장에서와 같습니다.

import nltk

import string

remove\_punct\_dict = dict((ord(punct), None) for punct in string.punctuation)

lemmar = WordNetLemmatizer()

def LemTokens(tokens):

return [lemmar.lemmatize(token) for token in tokens]

def LemNormalize(text):

return LemTokens(nltk.word\_tokenize(text.translate(remove\_punct\_dict)))

문사 유사도 측정 지표

1. Cosine Similarity

* 두 벡터 사이의 Cosine값을 구한다.

1. Jaccard Similarity
2. Manhattan Distance
3. Euclidean Distance

from sklearn.metrics.pairwise import cosine\_similarity

similarity = cosine\_similarity(feature\_vect\_simple, feature\_vect\_simple)

from sklearn.preprocessing import LabelBinarizer

#brand\_name, item\_condition

lb\_brand\_name = LabelBinarizer(sparse\_output=True)

X\_brand = lb\_brand\_name.fit\_transform(mecari\_df['brand\_name'])

from scipy.sparse import hstack

import gc

#hstack은 NumPy ndarray 형식의 배열을 옆으로 결합하는 방법

sparse\_matrix\_list = (X\_name, X\_descp, X\_brand, X\_item\_cond\_id, X\_shipping, X\_cat\_dae, X\_cat\_jung, X\_cat\_so)

X\_features\_sparse = hstack(sparse\_matrix\_list).tocsr()

print(type(X\_features\_sparse), X\_features\_sparse.shape)

추천 시스템 방식:

1. 콘텐츠 기반 필터링

* 콘텐츠에 기반한 속성.
* 영화 구성 콘텐츠 텍스트 -> 피처 벡터화 -> 코사인 유사도 -> 콘텐츠 별로 가중 평점 계산 -> 유사도 및 평점에 따른 영화 추천

1. 협업 필터링

movies\_df = movies[['id','title', 'genres', 'vote\_average', 'vote\_count',

'popularity', 'keywords', 'overview']]

pd.set\_option('max\_colwidth', 100)

movies\_df[['genres','keywords']][:1]

from ast import literal\_eval

movies\_df['genres'] = movies\_df['genres'].apply(literal\_eval)

movies\_df['keywords'] = movies\_df['keywords'].apply(literal\_eval)

movies\_df['genres'] = movies\_df['genres'].apply(lambda x : [ y['name'] for y in x])

movies\_df['keywords'] = movies\_df['keywords'].apply(lambda x : [ y['name'] for y in x])

movies\_df[['genres', 'keywords']][:1]

from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer

movies\_df['genres\_literal'] = movies\_df['genres'].apply(lambda x: ' '.join(x))

print('Before: ' , movies\_df['genres\_literal'].shape)

count\_vect = CountVectorizer(min\_df=0, ngram\_range=(1,2))

genre\_mat = count\_vect.fit\_transform(movies\_df['genres\_literal'])

print('After: ',genre\_mat.shape)

Before: (4803,)

After: (4803, 276)

from sklearn.metrics.pairwise import cosine\_similarity

genre\_sim = cosine\_similarity(genre\_mat, genre\_mat)

print(genre\_sim.shape)

print(genre\_sim[:2])

(4803, 4803)

[[1. 0.59628479 0.4472136 ... 0. 0. 0. ]

[0.59628479 1. 0.4 ... 0. 0. 0. ]]

genre\_sim\_sorted\_ind = genre\_sim.argsort()[:, ::-1]

print(genre\_sim\_sorted\_ind[:1])

title\_movie = movies\_df[movies\_df['title']=='The Godfather']

title\_movie.index.values

array([3337])

def find\_sim\_movie(df, sorted\_ind, title\_name, top\_n = 10):

title\_movie = df[df['title'] == title\_name]

title\_index = title\_movie.index.values

similar\_indexes = sorted\_ind[title\_index, :(top\_n)]

print(similar\_indexes)

similar\_indexes = similar\_indexes.reshape(-1)

return df.iloc[similar\_indexes]

percentile = 0.6

m = movies\_df['vote\_count'].quantile(percentile)

C = movies\_df['vote\_average'].mean()

def weighted\_vote\_average(record):

v = record['vote\_count']

R = record['vote\_average']

return ( (v/(v+m)) \* R ) + ( (m/(m+v)) \* C )

movies\_df['weighted\_vote'] = movies\_df.apply(weighted\_vote\_average, axis=1)

movies\_df[['title','vote\_average','weighted\_vote','vote\_count']].sort\_values('weighted\_vote',

ascending=False)[:10]

| **title** | **vote\_average** | **weighted\_vote** | **vote\_count** |  |
| --- | --- | --- | --- | --- |
| **1881** | The Shawshank Redemption | 8.5 | 8.396052 | 8205 |
| **3337** | The Godfather | 8.4 | 8.263591 | 5893 |
| **662** | Fight Club | 8.3 | 8.216455 | 9413 |
| **3232** | Pulp Fiction | 8.3 | 8.207102 | 8428 |

def find\_sim\_movie(df, sorted\_ind, title\_name, top\_n=10):

title\_movie = df[df['title'] == title\_name]

title\_index = title\_movie.index.values

# top\_n의 2배에 해당하는 쟝르 유사성이 높은 index 추출

similar\_indexes = sorted\_ind[title\_index, :(top\_n\*2)]

similar\_indexes = similar\_indexes.reshape(-1)

# 기준 영화 index는 제외

similar\_indexes = similar\_indexes[similar\_indexes != title\_index]

# top\_n의 2배에 해당하는 후보군에서 weighted\_vote 높은 순으로 top\_n 만큼 추출

return df.iloc[similar\_indexes].sort\_values('weighted\_vote', ascending=False)[:top\_n]

similar\_movies = find\_sim\_movie(movies\_df, genre\_sim\_sorted\_ind, 'The Godfather',10)

similar\_movies[['title', 'vote\_average', 'weighted\_vote']]

협업 필터링

* 사용자 기반: 당신과 비슷한 고객이 다음 상품도 구매했습니다
* 아이템기반: 특정 상품과 유사한 좋은 평가를 받은 다른 비슷한 상품을 추천

ratings = ratings[['userId', 'movieId', 'rating']]

ratings\_matrix = ratings.pivot\_table('rating', index='userId', columns='movieId')

ratings\_matrix.head(3)

아이템 기반 협업 필터링 구현 순서

1. 사용자-아이템 행렬 데이터를 아이템-사용자 행렬 데이터로 변환
2. 아이템간의 코사인 유사도로 아이템 유사도 산출
3. 사용자가 관람 하지 않은 아이템중에서 아이템간 유사도를 반영한 예측점수 계산
4. 예측점수가 가장 높은 순으로 아이템 추천

rating\_movies = pd.merge(ratings, movies, on='movieId')

ratings\_matrix = rating\_movies.pivot\_table('rating', index='userId', columns='title')

ratings\_matrix = ratings\_matrix.fillna(0)

ratings\_matrix.head(3)

ratings\_matrix\_T = ratings\_matrix.transpose()

from sklearn.metrics.pairwise import cosine\_similarity

item\_sim = cosine\_similarity(ratings\_matrix\_T, ratings\_matrix\_T)

item\_sim\_df = pd.DataFrame(data=item\_sim, index=ratings\_matrix.columns, columns=ratings\_matrix.columns)

def predict\_rating(ratings\_arr, item\_sim\_arr):

ratings\_pred = ratings\_arr.dot(item\_sim\_arr)/np.array([np.abs(item\_sim\_arr).sum(axis=1)])

return ratings\_pred

ratings\_pred = predict\_rating(ratings\_matrix.values, item\_sim\_df.values)

ratings\_pred\_matrix = pd.DataFrame(data=ratings\_pred, index = ratings\_matrix.index, columns = ratings\_matrix.columns)

**잠재 요인 협업 필터링**

* 사용자-아이템 평점 행렬 속에 숨어 있는 잠재 요인을 추출해 추천 예측을 할 수 있게 하는 기법. 분해하는 과정에서 잠재요인을 추출, 평점 행렬 재 구성하면서 추천을 구현한다.

Surprise는 무비렌즈 데이터 세트와 같이 userid, itemid, rating 컬럼들이 사용자(userid)를 기준으로 한 로우 레벨의 평점 데이터로 구성된 데이터 세트만 입력 가능

* Dataset.laod\_builtin(name='ml-100k'): 무비렌즈 아카이브 FTP 서버에서 무지렌즈 데아터를 내려받습니다.
* Reader: Raw데이터 소스에서 Dataset로 로딩 규칙을 지정하기 위해 사용됩니다.

from surprise.dataset import DatasetAutoFolds

from surprise.dataset import Reader

from surprise import SVD

reader = Reader(line\_format='user item rating timestamp', sep=',',

rating\_scale=(0.5, 5))

# DatasetAutoFolds 클래스를 사용해서 개별적으로 생성

# index와 header가 없는 상태로 재생성했던 ratings\_surprise.csv파일에 기반

data\_folds = DatasetAutoFolds(ratings\_file='/content/drive/MyDrive/data/ml-latest-small/ml-latest-small/ratings\_noh.csv',

reader=reader)

# 위에서 개별적으로 생성한 csv파일을 학습데이터로 생성

trainset = data\_folds.build\_full\_trainset()

algo = SVD(n\_factors=50, n\_epochs=20, random\_state=42)

algo.fit(trainset)

# 영화에 대한 정보 데이터 로딩

movies = pd.read\_csv('/content/drive/MyDrive/data/ml-latest-small/ml-latest-small/movies.csv')

ratings = pd.read\_csv('/content/drive/MyDrive/data/ml-latest-small/ml-latest-small/ratings.csv')

# 특정 사용자 9번의 movieId를 추출해서 특정 영화에 대한 평점 있는지 확인

movieIds = ratings[ratings['userId']==9]['movieId']

if movieIds[movieIds==42].count() == 0:

print('user id=9인 사람은 movie id=42에 대한 평점이 없음')

# 영화에 대한 정보 데이터에서 movieId가 42인 영화가 무엇인지 출력

print(movies[movies['movieId']==42])

def get\_unseen\_surprise(ratings, movies, userId):

# 특정 유저가 본 movie id들을 리스트로 할당

seen\_movies = ratings[ratings['userId']==userId]['movieId'].tolist()

# 모든 영화들의 movie id들 리스트로 할당

total\_movies = movies['movieId'].tolist()

# 모든 영화들의 movie id들 중 특정 유저가 본 movie id를 제외한 나머지 추출

unseen\_movies = [movie for movie in total\_movies if movie not in seen\_movies]

print(f'특정 {userId}번 유저가 본 영화 수: {len(seen\_movies)}\n추천한 영화 개수: {len(unseen\_movies)}\n전체 영화수: {len(total\_movies)}')

return unseen\_movies

def recomm\_movie\_by\_surprise(algo, userId, unseen\_movies, top\_n=10):

# 알고리즘 객체의 predict()를 이용해 특정 userId의 평점이 없는 영화들에 대해 평점 예측

predictions = [algo.predict(str(userId), str(movieId)) for movieId in unseen\_movies]

# predictions는 Prediction()으로 하나의 객체로 되어있기 때문에 예측평점(est값)을 기준으로 정렬해야함

# est값을 반환하는 함수부터 정의. 이것을 이용해 리스트를 정렬하는 sort()인자의 key값에 넣어주자!

def sortkey\_est(pred):

return pred.est

# sortkey\_est함수로 리스트를 정렬하는 sort함수의 key인자에 넣어주자

# 리스트 sort는 디폴트값이 inplace=True인 것처럼 정렬되어 나온다. reverse=True가 내림차순

predictions.sort(key=sortkey\_est, reverse=True)

# 상위 n개의 예측값들만 할당

top\_predictions = predictions[:top\_n]

# top\_predictions에서 movie id, rating, movie title 각 뽑아내기

top\_movie\_ids = [int(pred.iid) for pred in top\_predictions]

top\_movie\_ratings = [pred.est for pred in top\_predictions]

top\_movie\_titles = movies[movies.movieId.isin(top\_movie\_ids)]['title']

# 위 3가지를 튜플로 담기

# zip함수를 사용해서 각 자료구조(여기선 리스트)의 똑같은 위치에있는 값들을 mapping

# zip함수는 참고로 여러개의 문자열의 똑같은 위치들끼리 mapping도 가능!

top\_movie\_preds = [(ids, rating, title) for ids, rating, title in zip(top\_movie\_ids, top\_movie\_ratings, top\_movie\_titles)]

return top\_movie\_preds

### 위에서 정의한 함수를 사용해 특정 유저의 추천 영화들 출력해보기

unseen\_lst = get\_unseen\_surprise(ratings, movies, 9)

top\_movies\_preds = recomm\_movie\_by\_surprise(algo, 9, unseen\_lst,

top\_n=10)

print()

print('#'\*8,'Top-10 추천영화 리스트','#'\*8)

# top\_movies\_preds가 여러가지의 튜플을 담고 있는 리스트이기 때문에 반복문 수행

for top\_movie in top\_movies\_preds:

print('\* 추천 영화 이름: ', top\_movie[2])

print('\* 해당 영화의 예측평점: ', top\_movie[1])

print()