DataSet Load

데이터셋 로드

➢ 샘플 데이터 셋 로드

- sklearn.datasets 모듈 아래에 있는 함수들은 파이썬 딕셔너리와 유사한 Bunch 클래스 객체를 반환
- load_boston
- load_iris
- load_digits

```
from sklearn import datasets
#숫자 데이터셋을 적재
digits = datasets.load_digits()

features = digits.data # 특성 행렬을 만듭니다.
target = digits.target #타깃 벡터를 만듭니다.
features[0]
digits.keys()
digits['DESCR'][:70]

import numpy as np
#매개변수 return_X_y를 True로 설정하면 특성 X와 타깃 y 배열을 반환
#load_digits 함수는 필요한 숫자 개수를 지정할 수 있는 n_class 매개변수를 제공
x, y = datasets.load_digits(n_class=5, return_x_y=True)
np.unique(y)
```

데이터셋 로드

➢ 모의 데이터 생성

- make_regression() - 선형 회귀에 사용할 데이터 셋 생성 (선형으로 분산된 데이터를 생성, 가우스 노이즈의 표준 편차, 원하는 피쳐의 수 지정 가능
- make_classification() - 분류에 필요한 모의 데이터 셋 생성
- make_blobs() - 군집 알고리즘에 적용할 데이터셋 생성 (n개의 무작위 데이터 클러스터를 생성)
- make_circles() - 두개의 차원에 작은 원을 포함하는 큰 원이 포함된 임의의 데이터셋을 생성, SVM(Support Vector Machines)과 같은 알고리즘을 사용하여 분류를 수행할 때 유용

```
from sklearn.datasets import make_regression
#특성 행렬, 타깃 벡터, 정답계수를 생성
features, target, coefficients = make_regression(n_samples = 100,
n_features = 3, n_informative = 3, n_targets = 1, noise=0.0, coef=True, random_state = 1)
print('특성 행렬₩n', features[:3])
print('타깃 벡터₩n', target[:3])
features, target = make_classification(n_samples = 100,
n_features = 3, n_informative = 3, n_redundant = 1, n_classes = 2, weight=[.25, .75], random_state = 1)
print('특성 행렬₩n', features[:3])
print('타깃 벡터₩n', target[:3])
features, target  = make_blobs(n_samples = 100,
n_features = 2, centers =3, cluster_std = 0.5, shuffle=True, random_state = 1)
print('특성 행렬₩n', features[:3])
print('타깃 벡터₩n', target[:3])
```

데이터셋 로드

➢ 모의 데이터 생성

- make_regression은 실수 특성 행렬과 실수 타깃 벡터를 반환
- make_classification과 make_blobs는 실수 특성 행렬과 클래스의 소속을 나타내는 정수 타깃 벡터를 반환
- make_regression과 make_classification의 n_informative 는 타깃 벡터를 생성하는 데 사용할 특성 수를 결정
- n_features 는 전체 특성 수
- make_classification의 weight 매개변수를 사용해 불균형한 클래스를 가진 모의 데이터셋 생성
- make_blobs의 centers 매개변수는 생성될 클러스터의 수를 결정

```
from sklearn.datasets import make_regression

features, target  = make_blobs(n_samples = 100, n_features = 2, centers =3, cluster_std = 0.5,
shuffle=True, random_state = 1)
print('특성 행렬₩n', features[:3])
print('타깃 벡터₩n', target[:3])

import matplotlib.pyplot as plt
plt.scatter(features[:, 0], features[:, 1], c=target)
plt.show()
```

데이터셋 로드

➢ CSV 데이터 로드

- read_csv : 로컬 혹은 원격 CSV파일 적재 (seq, header, skiprows, nrows, ...)

```
import pandas as pd

url = 'https://tinyurl.com/simulated-data'
dataframe = pd.read_csv(url)
dataframe.head(2)

dataframe = pd.read_csv(url, skiprows=range(1, 11), nrows=1)
dataframe
```

데이터셋 로드

➢ Excel 데이터 로드

- read_excel : 엑셀 스프레드시트를 적재 (na_filter, skip_nrows, keep-default_na, na_values 등 매개변수 지원)
- sheet_name 매개변수는 시트 이름 문자열이나 시트의 위치를 나타내는 정수(0부터 시작되는 인덱스)를 모두 받을 수 있습니다.
- read_excel 함수를 사용하려면 xlrd 패키지를 설치해야 합니다.

```
import pandas as pd

url = 'https://tinyurl.com/simulated-excel'
dataframe = pd.read_excel(url, sheet_name=0, header=1)
dataframe.head(2)
```

데이터셋 로드

➢ JSON 데이터 로드

- orient 매개변수 - JSON 파일이 어떻게 구성되었는지 지정
- json_normalize() : 구조화가 덜 된 JSON 데이터를 판다스 데이터프레임으로 변환하는 도구
- split, records, index, columns, values 매개변수.

```
import pandas as pd

url = "https://tinyurl.com/simulated-json"
dataframe = pd.read_json(url, orient='columns')
dataframe.head(2)
```

데이터셋 로드

➢ 데이터베이스로부터 데이터 로드

- read_sql_query()를 사용하여 데이터베이스에 SQL 쿼리를 던져 데이터를 적재
- SQLite 데이터베이스 엔진으로 연결하기 위해 create_engine 함수를 사용합니다.

```
import pandas as pd
from sqlalchemy import create_engine

database_connection = create_engine('sqlite://sample.db')

dataframe = pd.read_sql_query(' select * from data', database_connection)

dataframe.head(2)

dataframe = pd.read_sql_table('data', database_connection)
dataframe.head(2)
```

데이터 랭글링(data wrangling)

데이터 랭글링(data wrangling)

➢ 데이터 랭글링(data wrangling)

- 광범위한 의미의 원본 데이터를 정제하고 사용 가능한 형태로 구성하기 위한 변환 과정
- 데이터 랭글링에 사용되는 가장 일반적인 데이터 구조 - 데이터프레임.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)
dataframe.head(5)
```

```
import pandas as pd

dataframe = pd.DataFrame()
dataframe['Name'] =['Jacky Jackson', 'Steven Stevenson']
dataframe['Age'] = [38, 25]
dataframe['Driver'] =[True, False]
Dataframe
new_person = pd.Series(['Molly Mooney', 40, True],
                       index = ['Name', 'Age','Driver' ])  #열 생성
dataframe.append(new_person, ignore_index = True)  #열 추가
dataframe
```

데이터 랭글링(data wrangling)

➢ 데이터프레임 생성

- 열 이름은 columns 매개변수에 지정
- 원본 리스트를 전달하여 데이터프레임 생성
- 열 이름과 데이터를 매핑한 딕셔너리를 사용해 데이터프레임 생성

```
import pandas as pd

data = [['Jacky Jackson', 38, True], ['Steven Stevenson', 25, False] ]
matrix = np.array(data)
pd.DataFrame(matrix, columns=['Name', 'Age', 'Driver'])
pd.DataFrame(data, columns=['Name', 'Age', 'Driver'])

data = {'Name' : ['Jacky Jackson' , 'Steven Stevenson'],
          'Age' : [38, 25],
         'Driver' : [True, False]}
pd.dataFrame(data)

data = [ {'Name': 'Jacky Jackson', 'Age' : 38, 'Driver' : True},
         {'Name': 'Steven Stevenson', 'Age' : 25, 'Driver' : False} ]
pd.DataFrame(data, index=['row1', 'row2'])
```

데이터 랭글링(data wrangling)

➢ 데이터프레임 구조 이해

- head() - 데이터의 처음 몇 개의 행을 확인
- tail() - 데이터의 마지막 몇 개의 행을 확인
- shape() - 데이터프레임의 행과 열의 수를 확인
- describe() - 수치형 열의 기본 통곗값을 확인
- loc(), iloc()를 사용하여 하나 이상의 행이나 값을 선택합니다.
- 콜론(:)을 사용하여 원하는 행의 슬라이스를 선택할 수 있습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)
dataframe.head(2)
dataframe.shape
dataframe.describe()
dataframe.iloc[0]
dataframe.iloc[1:4]
dataframe.iloc[:4]
```

데이터 랭글링(data wrangling)

- 데이터프레임 행 선택
  -

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Sex'] == 'female'].head(2)   #조건문으로 행 선택
dataframe[(dataframe['Sex'] == 'female']) & (dataframe['Age'] >=65)]  #여러 조건으로 행 선택 가능
```

데이터 랭글링(data wrangling)

➢ 값 치환

- replace() : 하나의 열 또는 전체 DataFrame객체에서 값을 찾아 바꿀 수 있습니다
  정규 표현식을 이용하여 값을 찾아 바꿀수 있습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)
dataframe['Sex'].replace("female", "Woman").head(2)
dataframe['Sex'].replace(["female", "male"], [ "Woman", "Man"] ).head(5)
dataframe.replace(r"1st", "First", regex=True).head(2)
dataframe.replace(["female", "male", "person").head(3)
dataframe.replace({"female" : 1, "male": 0 }).head(3)
```

데이터 랭글링(data wrangling)

➢ 열 이름 변경

- rename() - 여러 개의 열 이름을 변경할 경우 columns 매개변수에 딕셔너리를 전달
  index 매개변수를 사용하여 인덱스 변경

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.rename(columns={'PClass' : 'Passenger Class'}).head(2)
dataframe.rename(columns={'PClass' : 'Passenger Class', 'Sex' : 'Gender' }).head(2)

import collections
column_names = collections.defaultdict(str)
for name in dataframe.columns :                #키를 만듭니다.
    column_names[name]

column_names

dataframe.rename(index={0:-1}).head(2)
dataframe.rename(str.lower,   axis='columns').head(2)
```

데이터 랭글링(data wrangling)

➢ 최소값, 최대값, 합, 평균 계산

- std() : 표준편차
- kurt() : 첨도, 확률분포의 뾰족한 정도, 첨도가 3에 가까우면 정규분포와 비슷하며 3보다 작을 경우에는 정규분포보다 더 납작하고 3보다 크면 더 뾰족합니다.
- skew() : 확률분포의 비대칭도. 음수일 경우 정규분포보다 오른쪽으로 치우쳐 있고 양수일 경우 반대인 왼쪽으로 치우쳐 있습니다.
- sem() : 표준오차 , 샘플링된 표본의 평균에 대한 표준편차입니다
- mode() : 최빈값
- median() : 중간값
- corr() : 상관계수
- cov() : 공분산

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

print('최대값 : ' , dataframe['Age'].max())
print('최소값 : ' , dataframe['Age'].min())
print('평균 : ' , dataframe['Age'].mean())
print('합 : ' , dataframe['Age'].sum())
print('카운트 : ' , dataframe['Age'].count())
dataframe.corr()
dataframe.cov()
```

데이터 랭글링(data wrangling)

➢ 고유값 찾기
- unique와 value_unique는 범주형 열을 탐색하거나 조작할 때 유용
- value_counts() : 고유값과 등장 횟수를 출력
- nunique() : 고유값의 개수 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe['Sex'].unique()
dataframe['Sex'].value_counts()

dataframe['PClass'].value_counts()
dataframe['PClass'].nunique()
dataframe.nunique()
```

데이터 랭글링(data wrangling)

➢ 누락된 값 처리
- 판다스는 넘파이의 NaN("Not A Number")를 사용하여 누락된 값을 표시합니다.
- isnull() : 불리언 값 반환
- notnull() : 불리언 값 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Age'].isnull()].head(2)

import numpy as np
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)

#데이터를 로드하고 누락된 값을 설정
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

데이터 랭글링(data wrangling)

➢ 누락된 값 처리

- keep_default_na 매개변수를 False로 지정하면 판다스가 기본적으로 NaN으로 인식하는 문자열을 NaN으로 인식하지 않습니다.
- na_filter를 False로 설정하면 NaN 변환을 하지 않습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe = pd.read_csv(url, na_values=['female'], keep_default_na=False)
dataframe[12:14]

dataframe = pd.read_csv(url, na_filter=False)
dataframe[12:14]
```

데이터 랭글링(data wrangling)

➢ 열 삭제

- drop() : axis=1 매개변수 사용.
- 여러 열을 삭제해야 하는 경우 첫번째 매개변수로 열 이름을 리스트로 전달
- dataframe.columns에 열 인덱스를 지정하여 삭제할 수 있습니다
- inplace매개변수를 True로 설정하면 데이터프레임을 수정 가능한 객체처럼 다루므로 데이터 처리 파이프라인을 더욱 복잡하게 만듭니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.drop('Age', axis=1).head(2)
dataframe.drop(['Age', 'Sex'], axis=1).head(2)

dataframe.drop(dataframe.columns[1], aixs=1).head(2)

del dataframe['Age'] #판단스 내부의 호출 방식 때문에 권장하지 않음

dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

데이터 랭글링(data wrangling)

- 행 삭제
  - 불리언 조건을 사용하여 삭제하고 싶은 행을 제외한 새로운 데이터프레임을 만듭니다.
  - 조건을 사용하면 행 하나 또는 여러 개를 동시에 삭제할 수 있습니다.
  - drop_duplicates() 중복된 행 삭제
  - drop_duplicates는 기본적으로 모든 열이 완벽히 동일한 행만 삭제
  - 일부 열만 대상으로 중복된 행을 검사하여 삭제할 경우 subset 매개변수를 사용
  - keep매개변수는 중복된 행의 첫 행을 유지할지 마지막 행를 유지할지 선택할 수 있습니다.
  - duplicated() : 행이 중복되었는지 여부를 알려주는 불리언 시리즈를 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Sex'] != 'male').head(2)
dataframe[dataframe.index  != 0).head(2))

dataframe.drop_duplicates().head(2)
print("원본 데이터프레임 행의 수 :" , len(dataframe))
print("중복 삭제 후  행의 수 :" , len(dataframe.drop_duplicates()))
dataframe.drop_duplicates(subset=['Sex'])
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

데이터 랭글링(data wrangling)

- 값에 따라 행을 그룹핑
  - groupby() : 통계적 계산과 같이 각 그룹에 적용할 연산이 필요
  - resample() - 시간 간격에 따라 행을 그룹핑 , datetime 형태의 인덱스를 사용하므로 시간 간격(오프셋)을 넓혀서 행을 그룹핑

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.groupby('Survived')['Name'].count()
dataframe.groupby(['Sex', 'Survived'])['Age'].mean()
```

```
import numpy as np

time_index = pd.date_range('06/06/2017', periods=10000, freq='305')

dataframe = pd.DataFrame(index=time_index)
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)
dataframe.resample('W').sum()  #주 단위로 행을 그룹핑한 다음 합 계산
dataframe.resample('2W').mean()  #2주 단위로 그룹핑하고 평균을 계산
dataframe.resample('M').count()  #한달 간격으로 그룹핑하고 행을 카운트
```

데이터 랭글링(data wrangling)

- 열 원소 순회, 모든 열 원소에 함수 적용
  - 반복문 외에 리스트 컴프리헨션을 사용할 수 있습니다.
  - apply() : 열의 모든 원소에 내장 함수나 사용자 정의 함수를 적용합니다
  - map() : apply와 유사, 딕셔너리를 입력으로 넣을 수 있음

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

for name in dataframe['Name'][0:2]:
    print(name.upper())

[name.upper() for name in dataframe['Name'][0:2]]

def uppercase(x):
    return x.upper()

dataframe['Name'].apply(uppercase)[0:2]
dataframe['Survived'].map({1: 'Live', 0:'Dead'})[:5]
dataframe['Age'].apply(lambda x, age : x < age, age=30)[:5]
```

데이터 랭글링(data wrangling)

- 열 원소 순회, 모든 열 원소에 함수 적용
  - apply()와 applymap() : 데이터프레임 전체에 적용
  - apply() : 데이터프레임 열 전체에 적용
  - applymap() : 열의 각 원소에 적용

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.apply(lambda x: max(x)) #각 열에서 큰 값을 뽑음
def truncate_string(x):
    if type(x) == str:
        return x[:20]
    return x

dataframe.applymap(truncate_string)[:5] #문자열의 길이를 최대 20자로 줄입니다.

dataframe.groupby('Sex').apply(lambda x: x.count())
```

데이터 랭글링(data wrangling)

- 데이터 프레임 연결하기
  - concat (axis=0) : 행의 축에 따라 연결
  - append() : 데이터프레임에 새로운 행을 추가

```
import pandas as pd

data_a = {'id':['1', '2', '3'],
              'first' : ['Alex', 'Amy', 'Allen'],
              'last' : ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns =['id', 'first', 'last'])

data_b = {'id':['4', '5', '6'],
              'first' : ['Billy', 'Brian', 'Bran'],
              'last' : ['Bonder', 'Black', 'Balwner']}
dataframe_a = pd.DataFrame(data_a, columns =['id', 'first', 'last'])
pd.concat([dataframe_a, dataframe_b], axis=0)
pd.concat([dataframe_a, dataframe_b], axis=1)

row = pd.Series(10, 'Chris', 'Chillon' ], index=['id', 'first', 'last'])
dataframe_a.append(row, ignore_index=True)
```

데이터 랭글링(data wrangling)

➢ 데이터 프레임 병합

- merge() - 내부 조인 수행을 위해 on 매개변수에 병합 열을 지정
  동일한 매개변수로 왼쪽 조인과 오른쪽 조인을 지정할 수 있습니다.
- 각 데이터프레임에서 병합하기 위한 열 이름을 지정할 수 있습니다
- 각 데이터프레임의 인덱스를 기준으로 병합하려면 left_on과 right_on 매개변수를 right_index=True와 left_index=True로 바꿉니다.

```
import pandas as pd

employee_data = {'employee_id' : ['1', '2', '3', '4'],
                 'name' : ['Amy Jones', 'Allen Keys', 'Alice Bees', 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id', 'name'])

sales_data = {'employee_id' : ['3','4', '5','6'],
              , 'total_sales' : [23456, 2512, 2345, 1455] }
dataframe_sales = pd.DataFrame(sales_data, columns =['employee_id',  'total_sales'])
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
pd.merge(dataframe_employees, dataframe_sales, left_on='employee_id', right_on='employee_id', )
```

데이터 랭글링(data wrangling)

- 데이터 프레임 병합
  - how매개변수 inner : 두 데이터프레임에 모두 존재하는 행만 반환
  - how매개변수 outer : 두 데이터프레임의 모든 행이 반환, 행이 한쪽 데이터프레임에만 존재한다면 누락된 값은 NaN으로 채워짐
  - how매개변수 left : 왼쪽 데이터프레임의 모든 행이 반환, 오른쪽 데이터프레임은 왼쪽의 데이터프레임과 매칭되는 행만 반환
  - how매개변수 right : 오른쪽 데이터프레임의 모든 행이 반환, 왼쪽 데이터프레임은 오른쪽의 데이터프레임과 매칭되는 행만 반환

수치형 데이터 처리

데이터 스케일링이란 데이터 전처리 과정의 하나입니다.

데이터 스케일링을 해주는 이유는 데이터의 값이 너무 크거나 혹은 작은 경우에 모델 알고리즘 학습과정에서 0으로 수렴하거나 무한으로 발산해버릴 수 있기 때문입니다.

따라서, scaling은 데이터 전처리 과정에서 굉장히 중요한 과정입니다.

수치형 데이터 처리

➢ 특성 스케일 변환

- 수치형 특성이 두 값의 범위 안에 놓이도록 변환
- 대부분의 알고리즘은 모든 특성이 동일한 스케일을 가지고 있다고 가정합니다.
- 일반적으로 0~1이나 -1~1 사이입니다.
- MinMaxScaler - 특성의 최솟값과 최댓값을 사용하여 일정 범위 안으로 값을 조정합니다.
  모든 feature가 0과 1사이에 위치하게 만듭니다.
  데이터가 2차원 셋일 경우, 모든 데이터는 x축의 0과 1 사이에, y축의 0과 1사이에 위치하게 됩니다.

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- 사이킷런의 MinMaxScaler는 특성 스케일 fit()는 특성의 최솟값과 최댓값을 계산
- transform()는 특성의 스케일을 조정
- fit_transform()는 두 연산을 한번에 처리

```
import numpy as np
from sklearn import preprocessing

feature = np.array([[-500.5], [-100.1], [0], [100.1], [900.9]])
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

scaled_feature = minmax_scale.fit_transform(feature)
scaled_feature
```

수치형 데이터 처리

➢ 특성 표준화 변환

- 특성을 평균이 0이고 표준편차가 1이 되도록 변환 $x'_i = \frac{x_i - \bar{x}}{\sigma}$
- 특성을 표준 정규분포로 근사하는 스케일링 방식
- 변환된 특성은 원본 값이 특성 평균에서 몇 표준편차만큼 떨어져 있는지로 표현(통계학에서는 z-점수)
- 주성분 분석은 표준화가 적합하지만 신경망에는 최소-최대 스케일링을 권장합니다.
- StandardScaler - 각 feature의 평균을 0, 분산을 1로 변경합니다. 모든 특성들이 같은 스케일을 갖게 됩니다.

```
import numpy as np
from sklearn import preprocessing

x= np.array([[-1000.1], [-200.2], [500.5], [600.6], [9000.9]])
scaler = preprocessing.StandardScaler()
standardized = scaler.fit_tranform(x)
Standardized

print("평균:", round(standardized.mean()))
print("표준편차:", standardized.std())
```

수치형 데이터 처리

➢ 특성 스케일 변환

- 데이터에 이상치가 많다면 특성의 평균과 표준편차에 영향을 미치기 때문에 표준화에 부정적인 효과를 끼칩니다.
- 이상치가 많은 경우 중간값과 사분위 범위를 사용하여 특성의 스케일을 조정합니다.
- RobustScaler - 모든 특성들이 같은 크기를 갖는다는 점에서 StandardScaler와 비슷하지만, 평균과 분산 대신 median과 quartile을 사용합니다.
- RobustScaler는 이상치에 영향을 받지 않습니다.
- RobustScaler는 데이터에서 중간값을 빼고 IQR로 나눕니다.

```
robust_scaler = preprocessing.RobustScaler()
robust_scaler.fit_transform(x)
```

- QunatileTransformer : 훈련 데이터를 1,000개의 분위로 나누어 0~1 사이에 고르게 분포시킴으로써 이상치로 인한 영향을 줄입니다.

```
import numpy as np
from sklearn import preprocessing

x= np.array([[-1000.1], [-200.2], [500.5], [600.6], [9000.9]])
preprocessing.QuantileTransformer().fit_transform(x)
```

수치형 데이터 처리

➢ 정규화 변환

- Normalizer 클래스 - 단위 길이의 합이 1이 되도록 개별 샘플의 값을 변환
- 예) 각 단어나 n개의 단어 그룹이 특성인 텍스트 분류와 같이) 유사한 특성이 많을 때 종종 사용
- StandardScaler, RobustScaler, MinMaxScaler가 각 columns의 통계치를 이용한다면 Normalizer는 row마다 각각 정규화됩니다. Normalizer는 유클리드 거리가 1이 되도록 데이터를 조정합니다. (유클리드 거리는 두 점 사이의 거리를 계산할 때 쓰는 방법, L2 Distance)
- 행단위로 변환되므로 fit() 계산 작업 없이 transform() 사용

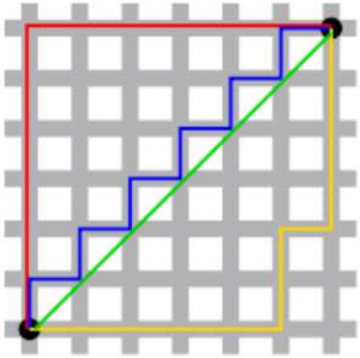
```
import numpy as np
from sklearn.preprocessing import Normalizer

features = np.array([[0.5, 0.5],  [1.1, 3.4],  [1.5, 20.2],  [1.63, 34.4],   [10.9, 3.3]])
normalizer = Normalizer(nor="12")
normalizer.transform(features)
fearues_l2_norm = Normalizer(norm="l2").tranform(features)
fearues_l2_norm
fearues_l1_norm = Normalizer(norm="l1").tranform(features)
fearues_l1_norm
print("첫 번째 샘플값의 합 : ", features_l1_norm[0, 0] + features_l1_norm[0, 1])
features / np.sum(np.abs(features), axis=1, keepdims=True)
features / np.sqrt(np.sum(np.square(features), axis=1, keepdims=True))
```

수치형 데이터 처리

➢ 정규화 변환

- Normalizer의 norm 옵션 L2는 유클리드로 기본값입니다. $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$
- 유클리드 거리는 두 점 사이의 거리를 계산할 때 쓰이는 방법으로 녹색과 같은 최단거리이다.
- Normalizer의 norm 옵션 L1은 맨하튼 거리계산(샘플 특성 값의 합을 1로 만듬) 방식입니다
- 빨간색, 노란색, 파란색이 모두 맨하탄 거리(맨하탄의 건물들을 가로지르지 않고 도로로 갔을 경우의 거리)이다.

$\|x\|_1 = \sum_{i=1}^{n} |x_i|$



- Normalizer의 norm 매개변수에 지정 옵션 max : 각 행의 최대값으로 행의 값을 나눔

```
import numpy as np
from sklearn.preprocessing import Normalizer

features = np.array([[0.5, 0.5],  [1.1, 3.4],  [1.5, 20.2],  [1.63, 34.4],   [10.9, 3.3]])
Normalizer(norm="max").transform(features)
```

수치형 데이터 처리

➢ 다항 특성과 교차항 특성 생성

- PolynomialFeatures 클래스는 교차항을 포함합니다.
- degree 매개변수가 다항식의 최대 차수를 결정
  예) degree=2는 2의 제곱까지 새로운 특성을 만듬 x_1, x_2, x_1^2, x_2^2
  degree=3은 2제곱과 3제곱까지 새로운 특성을 만듬 $x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3$
- interaction_only를 True로 지정하면 교차항 특성만 만들 수 있습니다.
- 특성과 타깃 사이에 비선형 관계가 있다는 가정을 추가할 때 다항 특성을 듭니다.
- 예) 주요 질병에 걸릴 확률에 나이가 미치는 영향은 일정한 상숫값이 아니고 나이가 증가함에 따라 같이 증가한다는 의심을 할 수 있음

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

features = np.array([[2, 3],      [2, 3],      [2, 3]])  # 특성 행렬을 만듭니다
# PolynomialFeatures 객체를 만듭니다.
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)
polynomial_interaction.fit_transform(features)      # 다항 특성을 만듭니다.

interaction = PolynomialFeatures(degree=2,  interaction_only=True, include_bias=False)
interaction.fit_transform(features)
```

수치형 데이터 처리

➢ 다항 특성과 교차항 특성 생성

- 특성 x에 변동 효과를 주입하기 위해서 고차항 특성을 만들 수 있습니다.
- include_bias 매개변수의 변수값은 True (변환된 특성에 상수항 1을 추가합니다.)

```
# 상수항 1을 추가합니다.
polynomial_bias = PolynomialFeatures(degree=2, include_bias=True).fit(features)
polynomial_bias.transform(features)
polynomial_bias.get_feature_names()
```

수치형 데이터 처리

➢ 특성 변환하기

- 하나 이상의 특성에 사용자 정의 변환을 적용
- FunctionTransformer 를 사용하여 일련의 특성에 어떤 함수를 적용할 수 있습니다.

```
import numpy as np
from sklearn.preprocessing import FunctionTransformer

features = np.array([[2, 3],    [2, 3],    [2, 3]])  # 특성 행렬 데이터 생성

def add_ten(x):    #  함수를 정의합니다.
    return x + 10

ten_transformer = FunctionTransformer(add_ten)  # 변환기 객체 생성
ten_transformer.transform(features)# 특성 행렬을 변환
```

- 판다스의 apply()를 사용하여 동일한 변환을 수행할 수 있다

```
import pandas as pd
df = pd.DataFrame(features, columns=['feature_1', 'feature_2'])
df.apply(add_ten)
```

수치형 데이터 처리

➢ 특성 변환하기

- FunctionTransformer의 validate 매개변수가 True이면 입력값이 2차원 배열인지 확인하고 아닐 경우 예외를 발생시킵니다
- validate가 False이면 일차원 배열에도 적용할 수 있습니다.

```
FunctionTranformer(add_ten, validate=False).transform(np.array([1, 2, 3]))
```

- ColumnTransformer : 특성 배열이나 데이터프레임의 열마다 다른 변환을 적용할 수 있습니다.

```
from sklearn.compose import ColumnTransformer

def add_hundred(x):       # 100을 더하는 함수 정의
    return x + 100

# (이름, 변환기, 열 리스트)로 구성된 튜플의 리스트를 ColumnTransformer에 전달합니다.
ct = ColumnTransformer(
    [("add_ten", FunctionTransformer(add_ten, validate=True), ['feature_1']),
     ("add_hundred", FunctionTransformer(add_hundred, validate=True), ['feature_2'])])
ct.fit_transform(df)
```

수치형 데이터 처리

➢ 이상치 검색

- 일반적인 방법은 데이터가 정규분포를 따른다고 가정하고 이런 가정을 기반으로 데이터를 둘러싼 타원을 그립니다. 타원 안의 샘플을 정상치(레이블 1)로 분류하고, 타원 밖의 샘플은 이상치(레이블 -1)로 분류합니다.
- 이상치의 비율은 contamination 매개변수로 지정 (contamination은 데이터가 얼마나 깨끗한지 추측)
- 데이터에 이상치가 적다면 contamination을 작게 지정, 데이터에 이상치가 많다면 contamination 값을 크게 설정할 수 있습니다.

```
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

features, _ = make_blobs(n_samples = 10,  n_features = 2,   centers = 1,  random_state = 1) # 모의 데이터 생성

features[0,0] = 10000  # 첫 번째 샘플을 극단적인 값으로 변경
features[0,1] = 10000

outlier_detector = EllipticEnvelope(contamination=.1)  # 이상치 감지 객체 생성
outlier_detector.fit(features)   # 감지 객체를 훈련시킴
outlier_detector.predict(features)  # 이상치를 예측
```

수치형 데이터 처리

➢ 이상치 검색

- 샘플을 전체적으로 보는 것보다 개별 특성에서 사분위범위(IQR)을 사용하여 극단적인 값을 구별할 수 있습니다.
- IQR은 데이터에 있는 1사분위와 3사분위 사이의 거리입니다.
- 보통 이상치는 1사분위보다 1.5 IQR 이상 작은 값이나 3사분위보다 1.5 IQR 큰 값으로 정의합니다.

```
# 하나의 특성을 만듭니다.
feature = features[:,0]

# 이상치의 인덱스를 반환하는 함수를 만듭니다.
def indicies_of_outliers(x):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# 함수를 실행합니다.
indicies_of_outliers(feature)
```

수치형 데이터 처리

➢ 이상치 처리

1. 삭제

```
import pandas as pd

houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

houses[houses['Bathrooms'] < 20]  # 샘플을 필터링
```

2. 이상치로 표시하고 이를 특성의 하나로 포함

```
import numpy as np

houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)  # 불리언 조건을 기반으로 특성을 생성
houses  # 데이터 확인
```

3. 이상치 처리 - 이상치의 영향이 줄어줄도록 특성을 변환

```
#로그 특성
houses['Log_Of_Square_Feet'] = [np.log(x) for x in houses["Square_Feet"]]
houses
```

수치형 데이터 처리

- **이상치 처리**
  - 이상치는 평균과 분산에 영향을 끼치기 때문에 이상치가 있다면 표준화가 적절하지 않습니다.
  - 이상치 처리 고려할 점

  1. 어떤 것을 이상치로 간주할 것인지?
  2. 이상치를 다루는 방법이 머신러닝의 목적에 맞아야 합니다.

수치형 데이터 처리

➢ 특성 이산화

- 이산화는 수치 특성을 범주형처럼 다루어야 할 때 유용한 전략입니다.
- 수치 특성을 개별적인 구간으로 나눌수 있습니다.
- Binarizer - 임계값에 따라 특성을 둘로 나누는 방법
- numpy.digitize() - 수치 특성을 여러 임계값에 따라 나누는 방법 (bins 매개변수의 입력값은 각 구간의 왼쪽 경계값입니다)

```
import numpy as np
from sklearn.preprocessing import Binarizer

age = np.array([[6], [12],[20],[36], [65]])
binarizer = Binarizer(18)
binarizer.fit_transform(age)

#bins 매개변수의 입력값은 각 구간의 왼쪽 경곗값입니다.
np.digitizer(age, bins=[20, 30, 64])
#right 매개변수를 True로 설정하면 이 동작을 바꿀 수 있습니다.
np.digitize(age, bins=[20, 30, 64], right=True)
np.digitize(age, bins=[18])
```

수치형 데이터 처리

➢ 특성 이산화

- KBinsDiscretizer - 연속적인 특성값을 여러 구간으로 나눔
- encode 매개변수의 기본값은 'onehot'으로 원-핫 인코딩된 희소 행렬을 반환합니다.
- onehot-dense 매개변수 값은 원-핫 인코딩된 밀집 배열을 반환합니다.
- 연속된 값을 이산화하여 원-핫 인코딩으로 만들면 범주형 특성으로 다루기 편리합니다.
- strategy 매개변수의 기본값은 'quantile'로 각 구간에 포함된 샘플 개수가 비슷하도록 만듭니다.
- strategy 매개변수의 기본값은 'uniform'은 구간의 폭이 동일하도록 만듭니다.
- 구간은 bin_edges_속성에서 확인할 수 있습니다.

```
from sklearn.preprocessing import KBinsDiscretizer

kb = KBinsDiscretizer(4, encode='ordinal', strategy='quantile') # 네 개의 구간으로 나눕니다
kb.fit_transform(age)

kb = KBinsDiscretizer(4, encode='onehot-dense', strategy='quantile')  # 원-핫 인코딩으로 반환
kb.fit_transform(age)

kb = KBinsDiscretizer(4, encode='onehot-dense', strategy='uniform') # 동일한 길이의 구간을 만듭니다
kb.fit_transform(age)
kb.bin_edges_
```

수치형 데이터 처리

➢ 군집으로 샘플을 그룹으로 묶기

- k개의 그룹이 있다는 것을 안다면 k-평균 군집(비지도 학습 알고리즘)을 사용하여 비슷한 샘플을 그룹으로 모을 수 있습니다.

```
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# 모의 특성 행렬 생성
features, _ = make_blobs(n_samples = 50,  n_features = 2,  centers = 3,  random_state = 1)
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

clusterer = KMeans(3, random_state=0)  # k-평균 군집 모델 생성
clusterer.fit(features)  # 모델 훈련

dataframe["group"] = clusterer.predict(features)  # 그룹 소속을 예측
dataframe.head(5)  # 처음 5 개의 샘플을 조회
```

수치형 데이터 처리

➢ 누락된 값을 가진 데이터 삭제

- 넘파이를 이용하여 누락된 값이 있는 샘플을 삭제

```
import numpy as np

features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]]) # 특성 행렬을 생성

# (~ 연산자를 사용하여) 누락된 값이 없는 샘플만 남깁니다.
features[~np.isnan(features).any(axis=1)]
```

- 판다스를 사용하여 누락된 값이 있는 샘플을 삭제

```
import pandas as pd

dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"]) # 데이터 로드
dataframe.dropna()   # 누락된 값이 있는 샘플을 제거
```

수치형 데이터 처리

➢ 누락된 값 채우기

- 데이터의 양이 작으면 k-최근접 이웃(KNN) 알고리즘을 사용해 누락된 값을 예측할 수 있습니다.
- KNN은 누락된 값에 가장 가까운 샘플을 구하기 위해 누락된 값과 모든 샘플 사이의 거리를 계산하므로 작은 데이터셋에서는 수용할 만하지만 데이터넷의 샘플이 수백만 개라면 문제가 됨

```
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

features, _ = make_blobs(n_samples = 1000, n_features = 2,  random_state = 1)  # 모의 특성 행렬 생성
scaler = StandardScaler()   # 특성을 표준화
standardized_features = scaler.fit_transform(features)

true_value = standardized_features[0,0]   # 첫 번째 샘플의 첫 번째 특성을 삭제
standardized_features[0,0] = np.nan

# 특성 행렬에 있는 누락된 값을 예측합니다.
features_knn_imputed = KNN(k=5, verbose=0).fit_transform(standardized_features)

print("실제 값:", true_value)   # 실제 값과 대체된 값을 비교
print("대체된 값:", features_knn_imputed[0,0])
```

수치형 데이터 처리

➢ 누락된 값 채우기

- 사이킷런의 Imputer 모듈을 사용하면 특성의 평균, 중간값, 최빈값으로 누락된 값을 채울 수 있습니다. (KNN 보다는 결과가 좋지 않습니다)
- 대용량 데이터셋에서는 누락된 값을 모두 평균값으로 채우는 것을 권장

```
from sklearn.preprocessing import Imputer

mean_imputer = Imputer(strategy="mean", axis=0) # Imputer 객체 생성
features_mean_imputed = mean_imputer.fit_transform(features) # 누락된 값을 채웁니다.

print("실제 값  True Value: ", true_value)
print("대체 값 Imputed Value :", features_mean_imputed[0, 0])
```

- SimpleImputer 는 strategy 매개변수 값 - mean, median, most_frequent, constant

```
from sklearn.impute import SimpleImputer

simple_imputer = SimpleImputer()
features_simple_imputed = simple_imputer.fit_transform(features)

print("실제 값 True Value:", true_value)  # 실제 값과 대체된 값을 비교
print("대체된 값 Imputed Value:", features_simple_imputed[0,0])
```