Handling Numerical Data

December 9, 2021

[1]:
```python
# Handling_Numerical_Data

# Rescaling a feature

import numpy as np
from sklearn import preprocessing

#Create a feature
feature = np.array([[1,1],[2,4],[34,8]])
print("Feature: ",feature)

#Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0,1))

#Scale feature
scaled_feature = minmax_scale.fit_transform(feature)

print(scaled_feature)

# MinMax Scaling new_x = (x-min(x))/(max(x)-min(x))
#Rescaling is a common preprocessing task in machine learning.
#use fit to calculate the minimum and maximum values of the feature, then use␣
 ↪trans form to rescale the feature.
#fit_transform does both at once
```

```
Feature:  [[ 1  1]
 [ 2  4]
 [34  8]]
[[0.         0.        ]
 [0.03030303 0.42857143]
 [1.         1.        ]]
```

[2]:
```python
# Standardizing a feature
# Transform a feature to have a mean of 0 and a standard deviation of 1.
np.random.seed(1)
#Create feature
x =np.array([[1,2],[23,4],[43,5]])
```

```
#Create a scale
scaler = preprocessing.StandardScaler()

#Scaling
standardized =scaler.fit_transform(x)

print("X: ",x)
print("Std_X: ",standardized)
print("mean = ",round(np.mean(standardized)))
print("std = ",np.std(standardized))

# new_x = (x-mean(x))/std(x)
#As a general rule,use standardization unless you have a specific reason to use␣
↪an alternative.
```

```
X:  [[ 1  2]
 [23  4]
 [43  5]]
Std_X:  [[-1.24371532 -1.33630621]
 [ 0.0388661   0.26726124]
 [ 1.20484922  1.06904497]]
mean =  0
std =  1.0
```

[3]:
```
# RobustScaler
'''If our data has significant outliers, it can negatively impact our␣
↪standardization by
affecting the feature's mean and variance. In this scenario, it is often␣
↪helpful to instead
rescale the feature using the median and quartile range.'''

robust_scaler = preprocessing.RobustScaler()

robust_scaler.fit_transform(x)
```

[3]:
```
array([[-1.04761905, -1.33333333],
       [ 0.        ,  0.        ],
       [ 0.95238095,  0.66666667]])
```

[4]:
```
# Normalizing Observations
#rescale the feature values of observations to have unit norm (a total length␣
↪of 1)
'''
    The L1 norm that is calculated as the sum of the absolute values of the␣
↪vector.
    The L2 norm that is calculated as the square root of the sum of the squared␣
↪vector values.
```

```
    The max norm that is calculated as the maximum vector values.
'''

features = np.array([[1.3,2.4],[4.4,2.0]])

normalizer = preprocessing.Normalizer(norm='l2')
print(features)
normalizer.transform(features)
#Normalizer provides three norm options with Euclidean norm (often called L2)
#"norm='l1' rescales an observation's values so they sum to 1, which can␣
↪sometimes be a desirable quality:"
#Manhattan norm (L1)

features_l1_norm = preprocessing.Normalizer(norm="l1").transform(features)
# Print sum
print("Sum of the first observation\'s values:",
features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

```
[[1.3 2.4]
 [4.4 2. ]]
Sum of the first observation's values: 1.0
```

[5]:
```
# Generating Polynomial and interaction features

feature = np.array([[1,2],[3,2],[3,6]])

polynomial_interaction = preprocessing.
↪PolynomialFeatures(degree=2,include_bias=False)

polynomial_interaction.fit_transform(features)

interaction = preprocessing.PolynomialFeatures(degree=2,interaction_only=True,␣
↪include_bias=False)
interaction.fit_transform(features)

'''Polynomial features are often created when we want to include the notion␣
↪that there
exists a nonlinear relationship between the features and the target.
The effects of each feature on the target
(sweetness) are dependent on each other. We can encode that relationship by␣
↪includ-
ing an interaction feature that is the product of the individual features.'''
```

[5]:
```
'Polynomial features are often created when we want to include the notion that
there\nexists a nonlinear relationship between the features and the target.\nThe
effects of each feature on the target\n(sweetness) are dependent on each other.
We can encode that relationship by includ-\ning an interaction feature that is
```

```
the product of the individual features.'
```

[6]:
```
# Transforming Features
from sklearn.preprocessing import FunctionTransformer

features = np.array([[1,2],[3,2],[3,6]])

def add_ten(x):
    return x+10

ten_transformer = FunctionTransformer(add_ten)
print(features)
ten_transformer.transform(features)

#This can be done in pandas
import pandas as pd
df = pd.DataFrame(features,columns=["feature_1","feature_2"])

df.apply(add_ten)
```

```
[[1 2]
 [3 2]
 [3 6]]
```

[6]:
```
   feature_1  feature_2
0         11         12
1         13         12
2         13         16
```

[7]:
```
# Detecting the Outliers
'''Common Method is to assume the data is normally distributed and based on␣
 ↪that assumption
"draw" an ellipse around the data, classifying any observation inside the␣
 ↪ellipse as an
inlier (labeled as 1 ) and any observation outside the ellipse as an outlier␣
 ↪(labeled as
-1 ):'''

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

#Simulate data
features,_ = make_blobs(n_samples=20,
                        n_features=2,
                        centers=1,
                        random_state=1)
```

```
#Replace the first observation values with extreme values
features[0,0]=10000
features[0,1]=10000

# Create detector
outlier_detector = EllipticEnvelope(contamination=.1)
# Fit detector
outlier_detector.fit(features)
# Predict outliers
outlier_detector.predict(features)

#If we expect our data to have few outliers, we can set contamination to␣
 ↪something small.
#Instead of looking at observations as a whole, we can instead look at␣
 ↪individual features and identify extreme values in those features using␣
 ↪interquartile range (IQR):

# Create a function to return index of outliers
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))

# Run function
indicies_of_outliers(feature)
```

```
[7]: (array([2]), array([1]))
```

```
[8]: # Handling Outliers
'''First, we should consider what
makes them an outlier. If we believe they are errors in the data such as from a␣
 ↪broken
sensor or a miscoded value, then we might drop the observation or replace␣
 ↪outlier
values with NaN. if we believe the outliers
are genuine extreme values (e.g., a house [mansion] with 200 bathrooms), then␣
 ↪mark-
ing them as outliers or transforming their values is more appropriate.'''

'''Second, how we handle outliers should be based on our goal for machine␣
 ↪learning.'''

houses = pd.DataFrame()
houses["Price"]=[534433, 392333, 293222, 4322032]
```

```
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Filter observations
print(houses[houses['Bathrooms'] < 20]) #We have removed the bathroom of 116

# Create feature based on boolean condition
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

print(houses)

# Log feature
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]
print(houses)

'''if you do have outliers standardization might not be appropri-
ate because the mean and variance might be highly influenced by the outliers.␣
↪In this
case, use a rescaling method more robust against outliers like RobustScaler .'''
```

```
     Price  Bathrooms  Square_Feet
0   534433        2.0         1500
1   392333        3.5         2500
2   293222        2.0         1500
     Price  Bathrooms  Square_Feet  Outlier
0   534433        2.0         1500        0
1   392333        3.5         2500        0
2   293222        2.0         1500        0
3  4322032      116.0        48000        1
     Price  Bathrooms  Square_Feet  Outlier  Log_Of_Square_Feet
0   534433        2.0         1500        0            7.313220
1   392333        3.5         2500        0            7.824046
2   293222        2.0         1500        0            7.313220
3  4322032      116.0        48000        1           10.778956
```

[8]:
```
'if you do have outliers standardization might not be appropri-\nate because the
mean and variance might be highly influenced by the outliers. In this\ncase, use
a rescaling method more robust against outliers like RobustScaler .'
```

[9]:
```
# Class intervals or Discretizating Features

import numpy as np
from sklearn.preprocessing import Binarizer

age = np.array([[44, 19, 26, 47, 11, 38, 53, 51, 67, 35]])
print(age)
binarizer = Binarizer(18)
```

```
binarizer.fit_transform(age)
```

```
[[44 19 26 47 11 38 53 51 67 35]]
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:70:
FutureWarning: Pass threshold=18 as keyword args. From version 1.0 (renaming of
0.25) passing these as positional arguments will result in an error
  warnings.warn(f"Pass {args_msg} as keyword args. From version "
```

```
[9]: array([[1, 1, 1, 1, 0, 1, 1, 1, 1, 1]])
```

```
[10]: #we can break up numerical features according to multiple thresholds:
#Bin feature
print(age)
np.digitize(age,bins=[20,30,68])

#bins parameter denote the left edge of each bin
#We can switch this behavior by setting the parameter right to True :

np.digitize(age, bins=[20,30,64], right=True)
```

```
[[44 19 26 47 11 38 53 51 67 35]]
```

```
[10]: array([[2, 0, 1, 2, 0, 2, 2, 2, 3, 2]])
```

```
[11]: # Grouping Observations Using Clustering
#You want to cluster observations so that similar observations are grouped␣
 ↪together.

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

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

#K-Means cluster
clusterer = KMeans(3,random_state=0)

#Fit clusterer
clusterer.fit(features)

#Predict values
dataframe['group'] =clusterer.predict(features)
```

```
dataframe.head(5)
```

[11]:
```
   feature_1  feature_2  group
0  -9.877554  -3.336145      0
1  -7.287210  -8.353986      2
2  -6.943061  -7.023744      2
3  -7.440167  -8.791959      2
4  -6.641388  -8.075888      2
```

[12]:
```
# Deleting Observations with Missing Values
import numpy as np

features = np.array([[1,2,3,np.nan],[3,4,5,6],[7,5,23,np.nan]])
print(features)
#Keep only observations that are not (denoted by ~)missing
print(features[~np.isnan(features).any(axis=1)])

# Pandas.dropna() can also be used
# Load data
dataframe = pd.DataFrame({"age":[1,2,3,4,np.nan],"name":["dfs",np.
 ↪nan,"fer","sdf","sfdg"]})
print(dataframe)
# Remove observations with missing values
dataframe.dropna()
```

```
[[ 1.   2.   3. nan]
 [ 3.   4.   5.   6.]
 [ 7.   5. 23. nan]]
[[3. 4. 5. 6.]]
   age  name
0  1.0   dfs
1  2.0   NaN
2  3.0   fer
3  4.0   sdf
4  NaN  sfdg
```

[12]:
```
   age name
0  1.0  dfs
2  3.0  fer
3  4.0  sdf
```

[13]:
```
'''There are three types of missing data:
Missing Completely At Random (MCAR)
The probability that a value is missing is independent of everything. For␣
 ↪example,
a survey respondent rolls a die before answering a question: if she rolls a␣
↪six, she
skips that question.
Missing At Random (MAR)
The probability that a value is missing is not completely random, but depends on
the information captured in other features. For example, a survey asks about␣
↪gen-
der identity and annual salary and women are more likely to skip the salary␣
↪ques-
tion; however, their nonresponse depends only on information we have captured
in our gender identity feature.
Missing Not At Random (MNAR)
The probability that a value is missing is not random and depends on informa-
tion not captured in our features. For example, a survey asks about gender iden-
tity and women are more likely to skip the salary question, and we do not have a
gender identity feature in our data.
It is sometimes acceptable to delete observations if they are MCAR or MAR. How-
ever, if the value is MNAR, the fact that a value is missing is itself␣
↪information. Delet-
ing MNAR observations can inject bias into our data because we are removing␣
↪obser-
vations produced by some unobserved systematic effect.'''
```

[13]:
```
'There are three types of missing data:\nMissing Completely At Random
(MCAR)\nThe probability that a value is missing is independent of everything.
For example,\na survey respondent rolls a die before answering a question: if
she rolls a six, she\nskips that question.\nMissing At Random (MAR)\nThe
probability that a value is missing is not completely random, but depends
on\nthe information captured in other features. For example, a survey asks about
gen-\nder identity and annual salary and women are more likely to skip the
salary ques-\ntion; however, their nonresponse depends only on information we
have captured\nin our gender identity feature.\nMissing Not At Random
(MNAR)\nThe probability that a value is missing is not random and depends on
informa-\ntion not captured in our features. For example, a survey asks about
gender iden-\ntity and women are more likely to skip the salary question, and we
do not have a\ngender identity feature in our data.\nIt is sometimes acceptable
to delete observations if they are MCAR or MAR. How-\never, if the value is
MNAR, the fact that a value is missing is itself information. Delet-\ning MNAR
observations can inject bias into our data because we are removing
obser-\nvations produced by some unobserved systematic effect.'
```

[14]:
```
# Imputing Missing values
#If you have a small amount of data, predict the missing values using k-nearest␣
↪neighbors (KNN):


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)
#Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

#Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0]=np.nan

# Predict the missing values in the feature matrix
features_knn_imputed = KNN(k=5,verbose=0).fit_transform(standardized_features)

# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])
```

```
True Value: 0.8730186113995938
Imputed Value: 1.0955332713113226
```

[15]:
```
"""Alternatively, we can use scikit-learn’s Imputer module to fill in missing␣
 ↪values with
the feature’s mean, median, or most frequent value. However, we will typically␣
 ↪get
worse results than KNN:"""

from sklearn.impute import SimpleImputer
#Create imputer
mean_imputer = SimpleImputer(strategy="mean",missing_values=np.nan)
#Impute values

features_mean_imputed = mean_imputer.fit_transform(features)

# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_mean_imputed[0,0])
```

```
True Value: 0.8730186113995938
Imputed Value: -3.058372724614996
```

[16]:
```
'''The downside to KNN is that in order to know which observations are the␣
 ↪closest to
```

```
the missing value, it needs to calculate the distance between the missing value␣
↪and
every single observation. This is reasonable in smaller datasets, but quickly␣
↪becomes
problematic if a dataset has millions of observations.
An alternative and more scalable strategy is to fill in all missing values with␣
↪some
average value. For example, in our solution we used scikit-learn to fill in␣
↪missing val-
ues with a feature’s mean value. The imputed value is often not as close to the␣
↪true
value as when we used KNN, but we can scale mean-filling to data containing mil-
lions of observations easily.
If we use imputation, it is a good idea to create a binary feature indicating␣
↪whether or
not the observation contains an imputed value.'''
```

[16]:
```
'The downside to KNN is that in order to know which observations are the closest
to\nthe missing value, it needs to calculate the distance between the missing
value and\nevery single observation. This is reasonable in smaller datasets, but
quickly becomes\nproblematic if a dataset has millions of observations.\nAn
alternative and more scalable strategy is to fill in all missing values with
some\naverage value. For example, in our solution we used scikit-learn to fill
in missing val-\nues with a feature’s mean value. The imputed value is often not
as close to the true\nvalue as when we used KNN, but we can scale mean-filling
to data containing mil-\nlions of observations easily.\nIf we use imputation, it
is a good idea to create a binary feature indicating whether or\nnot the
observation contains an imputed value.'
```