Preprocessing data

Neural Networks for Machine Learning Applications

Spring 2023

Sakari Lukkarinen

Helsinki Metropolia University of Applied Sciences

Contents

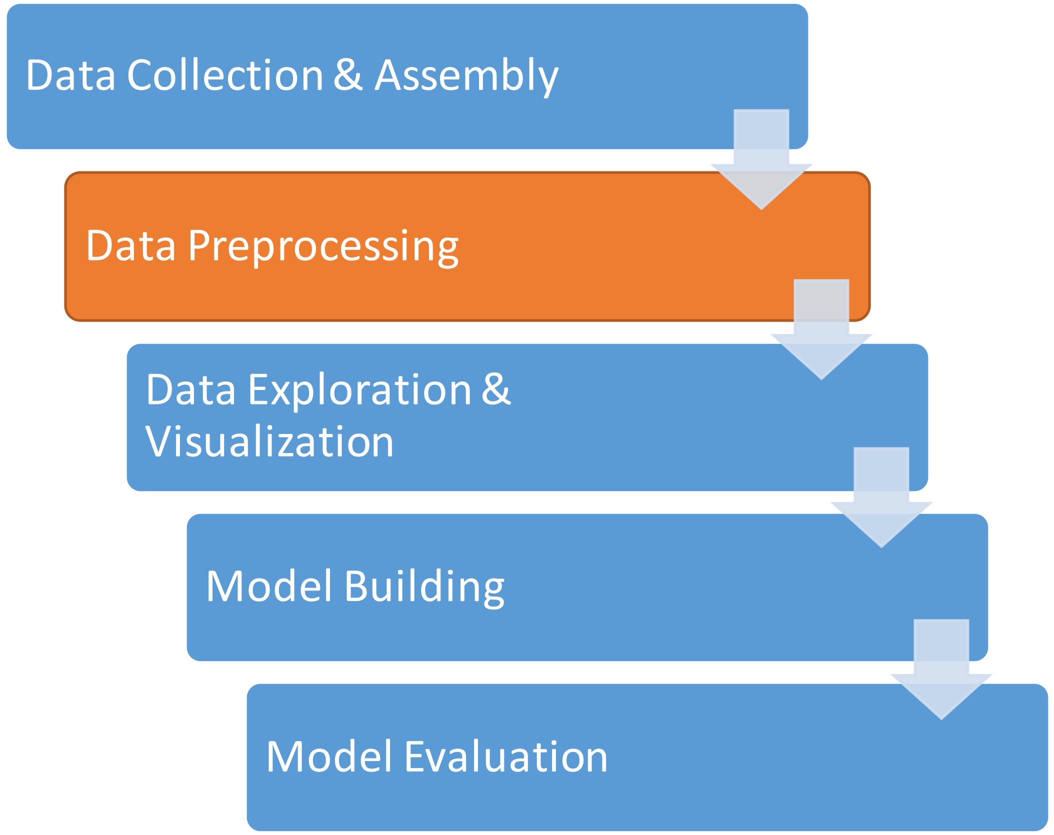
- Data pipeline & preprocessing data
- Rescaling, standardization and normalization
- Encoding categorical data
- Discretization
- Imputation of missing values

Preprocessing data

The **sklearn.preprocessing** package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

In general, learning algorithms benefit from *standardization* of the data set.

If some *outliers* are present in the set, *robust scalers* or transformers are more appropriate.



https://scikit-learn.org/stable/modules/preprocessing.html

Rescaling data

Easy way to rescale the data is to add/subtract a constant and then divide by another constant

$$x_{sc} = \frac{x - x_0}{k}$$

For example, if you subtract the minimum and divide by the span (max – min), the data is normalized between 0 and 1:

```
x = np.random.randint(100, 200, 10)
print(x)
x_sc = (x - 100)/100
print(x_sc)
```

```
[101 185 138 157 100 163 126 170 191 123]
[0.01 0.85 0.38 0.57 0.   0.63 0.26 0.7  0.91 0.23]
```

Standardization

Standardization of datasets is a common requirement for many machine learning estimators.

The estimators might behave badly if the individual features have totally different scales and distributions.

In practice, we often ignore the shape of the distribution and just transform the data to the center by removing the *mean* value of each feature, and then scale it by dividing by their *standard deviation*.

Preprocessing.scale does this.

scikit-learn > preprocessing > standardization

Mathematically

$$z = \frac{x-\overline{\mu}}{\sigma}$$

where

μ is the mean value of x

σ is the standard deviation of x

```
data_sc = preprocessing.scale(data)

m = data_sc.mean(axis=0)
for i in m:
    print(f'{i:.2f}', end = " ")
print()

s = data_sc.std(axis = 0)
for i in s:
    print(f'{i:.2f}', end = " ")

```

```
-0.00 -0.00 -0.00 0.00 -0.00 -0.00 0.00 0.00 -0.00
1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1
```

StandardScaler

The preprocessing module further provides a utility class **StandardScaler** that computes the mean and standard deviation on a training set to be able to later reapply the same transformation on the testing set.

```
scaler = preprocessing.StandardScaler().fit(data)
for i in scaler.mean_:
    print(f'{i:.1f}', end = " ")
print()
for i in scaler.scale_:
    print(f'{i:.1f}', end = " ")

data_sc = scaler.transform(data)
data_sc
```

```
54.5 0.7 3.2 131.7 247.4 0.1 1.0 149.6 0.3 1.1 1.6 0.7 4.7
9.0 0.5 1.0 17.7 51.9 0.4 1.0 22.9 0.5 1.2 0.6 0.9 1.9

array([[ 0.93618065,  0.69109474, -2.24062879, ...,  2.26414539,
        -0.72197605,  0.65587737],
       [ 1.3789285 ,  0.69109474,  0.87388018, ...,  0.6437811 ,
         2.47842525, -0.89422007],
```

scikit-learn > preprocessing > standardization

Normalization (=scaling features to a range)

An alternative standardization is **scaling features to lie between a given minimum and maximum value**, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size.

This can be achieved using **MinMaxScaler** or **MaxAbsScaler**, respectively.

[scikit-learn > preprocessing > standardization]()

```
minmax_scaler = preprocessing.MinMaxScaler()
data_sc = minmax_scaler.fit_transform(data)
for i in minmax_scaler.min_:
    print(f'{i:.1f}', end = " ")
print()
for i in minmax_scaler.scale_:
    print(f'{i:.1f}', end = " ")

df = pd.DataFrame(data_sc)
df.describe()
```

```
-0.6 0.0 -0.3 -0.9 -0.3 0.0 0.0 -0.5 0.0 0.0 -0.5 0.0 -0.8
0.0 1.0 0.3 0.0 0.0 1.0 0.5 0.0 1.0 0.2 0.5 0.3 0.2
```

	0	1	2	3	4	5
count	297.000000	297.000000	297.000000	297.000000	297.000000	297.000000
mean	0.532127	0.676768	0.719416	0.355600	0.277055	0.144781
std	0.188536	0.468500	0.321620	0.167574	0.118716	0.352474
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.395833	0.000000	0.666667	0.245283	0.194064	0.000000
50%	0.562500	1.000000	0.666667	0.339623	0.267123	0.000000
75%	0.666667	1.000000	1.000000	0.433962	0.342466	0.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Scaling data with outliers

If your data contains many outliers, scaling using the mean and variance of the data is likely to not work very well.

In these cases, you can use **robust_scale** and **RobustScaler** as drop-in replacements instead.

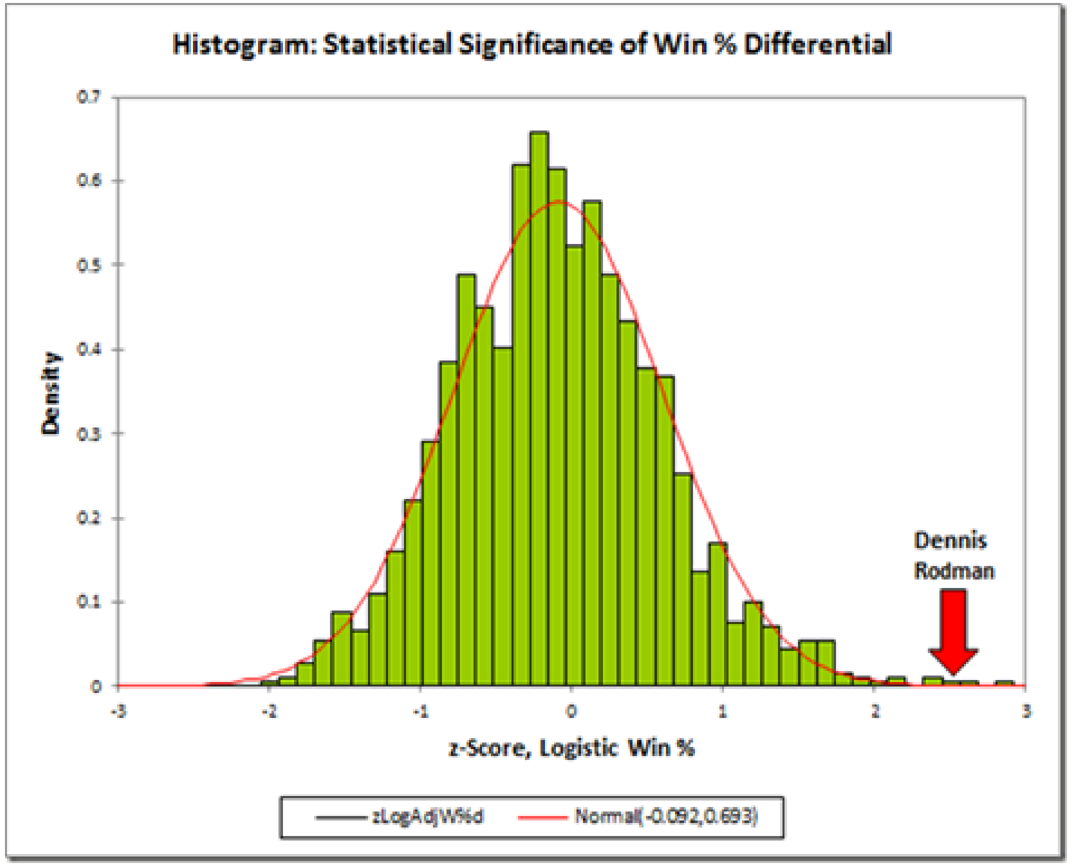
They use more robust estimates for the center and range of your data.

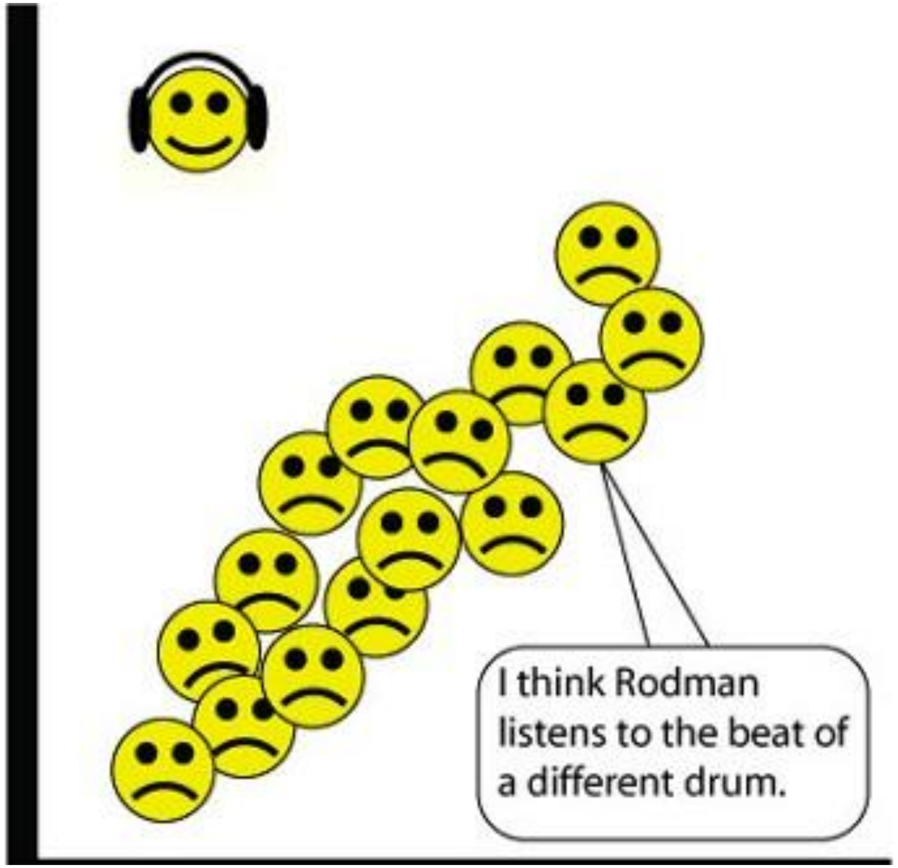
See: sklearn.preprocessing.RobustScaler

scikit-learn > preprocessing > standardization

What is an outlier?

Graphical interpretation of outliers





[The Case for Dennis Rodman: Guide » Skeptical Sports Analysis]

Who is (Dennis) Rodman?

Dennis Rodman – The Outlier

Photo: Four Dennis Rodman Sneakers We Want Back

Career highlights and awards

- 5× NBA champion (1989, 1990, 1996–1998)
- 2× NBA All-Star (1990, 1992)
- 2× All-NBA Third Team (1992, 1995)
- 2× NBA Defensive Player of the Year (1990, 1991)
- 7× NBA All-Defensive First Team (1989–1993, 1995, 1996)
- NBA All-Defensive Second Team (1994)
- 7× NBA rebounding champion (1992–1998)
- IBM Award (1992)
- NBA 75th Anniversary Team
- No. 10 retired by Detroit Pistons
- 3× NAIA All-American (1984–1986)

https://en.wikipedia.org/wiki/Dennis_Rodman

Summary of rescaling, standardization and normalization of the data

Rescaling means add or subtract any constant and multiply or divide by another constant

$$x_{sc} = \frac{x - c}{k}$$

Normalizing (often) refers to rescaling to make all elements lie between 0 and 1

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Standardizing means subtracting the mean and dividing by standard deviation, so that z resembles normal distribution

$$\mathrm{z} = \frac{x - \bar{x}}{\sigma}$$

Further discussion on the importance of centering and scaling data is available on [Should I normalize/standardize/rescale the data.]

Other transformations

Encoding categorical data

Often features are not given as continuous values but categorical.

For example a person could have features ["male", "female"] or [“healthy", “mild CVD", “moderate CVD“, “severe CVD”]

To convert categorical features to integer codes, we can use the ***OrdinalEncoder***.

```
enc = preprocessing.OrdinalEncoder()
x = ["male", "female", "female", "male", "other"]
data1 = pd.DataFrame({'sex': x})
enc.fit(data1)
enc.transform(data1)
```

```
array([[1.],
       [0.],
       [0.],
       [1.],
       [2.]])
```

[scikit-learn.org > preprocessing > encoding-categorical-features](scikit-learn.org)

One-hot-encoding

Another possibility to convert categorical features to features that can be used with machine learning estimators is to use a *one-of-K*, also known as *one-hot* or *dummy encoding*.

This type of encoding can be obtained with the **OneHotEncoder**, which transforms each categorical feature with n_categories possible values into n_categories binary features, with one of them 1, and all others 0.

```
enc = preprocessing.OneHotEncoder()
x = ["male", "female", "female", "male", "other"]
data1 = pd.DataFrame({'sex': x})
enc.fit(data1)
enc.transform(data1).toarray()
```

```
array([[0., 1., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Discretization

Discretization (otherwise known as *quantization* or *binning*) provides a way to partition continuous features into discrete values.

Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.

scikit-learn > preprocessing > discretization

Example of discretization

```
# Discretizing age into 5 categories
est = preprocessing.KBinsDiscretizer(n_bins = 5, encode = 'ordinal')
age = data[0].values.reshape(-1, 1)
est.fit(age)
print('Parameters = ', est.get_params())
est.transform(age)

```

```
Parameters =  {'encode': 'ordinal', 'n_bins': 5, 'strategy': 'quantile'}

array([[4.],
       [4.],
       [4.],
       [0.],
       [0.],
       [2.],
       [3.],
```

Imputation of missing values

scikit-learn > impute

Where the missing values come from?

For various reasons, many real-world datasets contain *missing values*, often encoded as blanks, NaNs or other placeholders. Such datasets however are incompatible with neural networks which assume that all values in an array are numerical, and that all have and hold meaning.

A basic strategy to use incomplete datasets is to *discard entire rows and/or columns* containing missing values. However, this comes at the price of losing data which may be valuable (even though incomplete).

A better strategy is to *impute the missing values*, i.e., to infer them from the known part of the data.

Ways to compensate (impute) missing values

1. Do nothing (if the model and algorithm allows it !!)
2. Discard the rows (basic strategy)
3. Use most frequent (mode) or zero/constant (0) value
4. Use mean/median values (mean, media)
5. Use k-nearest neighbour (KNN)
6. Use multivariate imputation by chained equation (MICE)
7. Use deep learning (datawig)

Will Badr (Jan 5, 2019). 6 different ways to compensate for missing values in a dataset. Towards Data Science.

Pandas solutions for missing values

DataFrame object have several methods

- Do nothing (don’t work with tensorflow)
- `dropna()` – drop the rows with missing values
- `fillna(0)` – Fill with zero or constant
- `fillna(method=’pad’)` – Fill gaps forward and backward
- `fillna(df.median())` – Fill with median (or mean value)
- `df.interpolate()` – Interpolate

1. Do nothing

Leave the data as it is. Missing values are marked with NaN.

```
filename = r'https://archive.ics.uci.edu/ml/machine-learning-databases/heart-diseas
df = pd.read_csv(filename,
                  index_col = None,
                 header = None,
                 na_values = '?')
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.0	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.0	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.0	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.0	3.0	1
302	38.0	1.0	3.0	138.0	175.0	0.0	0.0	173.0	0.0	0.0	1.0	NaN	3.0	0

Do nothing don’t work with tensorflow ?!

Problem:

Fitting the model with missing data values cause the loss and metrics become nans (not-a-numbers)

```
EPOCHS = 5

history = model.fit(data, labels, epochs = EPOCHS, validation_split = 0.2, verbose = 2)
```

```
Train on 242 samples, validate on 61 samples
Epoch 1/5
242/242 - 0s - loss: nan - mae: nan - mse: nan - val_loss: nan - val_mae: nan - val_mse: nan
Epoch 2/5
242/242 - 0s - loss: nan - mae: nan - mse: nan - val_loss: nan - val_mae: nan - val_mse: nan
Epoch 3/5
242/242 - 0s - loss: nan - mae: nan - mse: nan - val_loss: nan - val_mae: nan - val_mse: nan
Epoch 4/5
242/242 - 0s - loss: nan - mae: nan - mse: nan - val_loss: nan - val_mae: nan - val_mse: nan
Epoch 5/5
242/242 - 0s - loss: nan - mae: nan - mse: nan - val_loss: nan - val_mae: nan - val_mse: nan
```

2. Discard the rows (basic strategy)

Pandas has a method for that - [df.dropna()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html)

```
filename = r'https://archive.ics.uci.edu/ml/machine-lear
df = pd.read_csv(filename,
                 index_col = None,
                header = None,
                na_values = '?')
df = df.dropna()
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
297	57.0	0.0	4.0	140.0	241.0	0.0	0.0	123.0	1.0	0.2	2.0	0.0	7.0	1
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.0	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.0	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.0	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.0	3.0	1

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

3. Use zero or constant values

Pandas has a method fillna() for that

```
filename = r'https://archive.ics.uci.edu/ml/machine-learning-databases/heart-di
df = pd.read_csv(filename,
                 index_col = None,
                 header = None,
                 na_values = '?')
df = df.fillna(0)
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.0	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.0	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.0	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.0	3.0	1
302	38.0	1.0	3.0	138.0	175.0	0.0	0.0	173.0	0.0	0.0	1.0	0.0	3.0	0

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#filling-missing-values-fillna

4. Fill missing values with mean or median

You can fill missing values with mean or median of the column:

```
df = df.fillna(df.mean())
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.000000	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.000000	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.000000	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.000000	3.0	1
302	38.0	1.0	3.0	138.0	175.0	0.0	0.0	173.0	0.0	0.0	1.0	0.672241	3.0	0

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#filling-with-a-pandasobject

5. Fill gaps forward and backward

method = ’pad’ propagates proper values forward and backward

```
df = df.fillna(method='pad')
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.0	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.0	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.0	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.0	3.0	1
302	38.0	1.0	3.0	138.0	175.0	0.0	0.0	173.0	0.0	0.0	1.0	1.0	3.0	0

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#filling-missing-values-fillna

6. Interpolate the missing values

DataFrame objects have interpolate() that, by default, performs a linear interpolation at missing data points:

```
df = df.interpolate()
df.tail()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
298	45.0	1.0	1.0	110.0	264.0	0.0	0.0	132.0	0.0	1.2	2.0	0.0	7.0	1
299	68.0	1.0	4.0	144.0	193.0	1.0	0.0	141.0	0.0	3.4	2.0	2.0	7.0	2
300	57.0	1.0	4.0	130.0	131.0	0.0	0.0	115.0	1.0	1.2	2.0	1.0	7.0	3
301	57.0	0.0	2.0	130.0	236.0	0.0	2.0	174.0	0.0	0.0	2.0	1.0	3.0	1
302	38.0	1.0	3.0	138.0	175.0	0.0	0.0	173.0	0.0	0.0	1.0	1.0	3.0	0

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#interpolation

Summary

The **sklearn.preprocessing** package provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the neural networks.

The **Pandas** package offers several ways to handle the missing data: drop the whole rows away (basic strategy) or fill the missing data with zeros, constants, mean, median, or interpolated values calculated for each column separately.