**National University of Computer and Emerging Sciences**



**Lab Manual 06**

Department of Computer Science

# Objectives

After performing this lab, students shall be able to understand:

Data preprocessing using pandas

* + Handle Missing values (dropna, fillna, replace, interpolation)
  + Handle duplicates
  + Handle outliers (using z-score, quantile functions)

***What are missing values in dataset?***

Missing data occurs when one or more values are absent in a dataset. This can happen for various reasons, such as measurement errors, data corruption, or simply incomplete data collection. Handling missing data is essential for accurate analysis and model building.

1. ***Identifying Missing Values***

Before you can start imputing/handling missing values, it’s important to identify where those missing values are in your dataset. Pandas provide several functions to detect missing values:

1. **isna() and isnull()** : These functions return a DataFrame of the same shape as the original, where each element is a Boolean value indicating whether it’s missing (True) or not (False).
2. **notna() and notnull() :** These functions return the opposite of isna() and isnull() , indicating non-missing values.
3. **Info():** This method provides a concise summary of the DataFrame, including the count of non-null values for each column. By comparing this count to the total number of rows, you can quickly identify columns with missing values.

***Handling Missing Values:***

1. **A Simple Option: Removing row or column with Missing Values**

One approach to handling missing data is, you can simply to remove the rows or columns that contain any missing values. You can use the **dropna()** function in Python to remove the rows or columns with missing data. By default, **dropna()** in Python pandas drops any row or column that contains at least one missing value.

If the missing values are limited to a few rows or columns and don’t represent a significant portion of the dataset, you might choose to remove those rows or columns.

Dropna()

Pandas DataFrame.dropna(~) method removes rows or columns with missing values.

**Parameters**

**1. axis | int or string | optional**

Whether or not to remove rows or columns with missing values:  
  
Axis | Description  
0 or "index" | Scans through each row, and if a missing value exists, drop the row.  
1 or "columns" | Scans through each column, and if a missing value exists, drop the column.  
  
By default, axis=0.

**2. how | string | optional**

The criteria by which to remove a row/column:  
  
How | Description  
"any" | If the row or column consists of at least one missing value, then remove it.  
"all" | If the row or column consists of all missing values, then remove it.  
  
By default, how="any".

**3. thresh | int | optional**

The number of non-NaN a row/column must at least contain to not be dropped. For instance, if thresh=2, then:  
  
a column with 1 non-missing value will be dropped.  
a column with 2 non-missing values will be kept.  
a column with 3 non-missing values will be kept.  
  
By default, no minimum is set.

**4. subset | array-like of strings | optional**

The columns to check for missing values when scans are performed row-wise (when axis=0). By default, all columns are considered.

**5. inplace | boolean | optional**

If True, then the method will directly modify the source DataFrame instead of creating a new DataFrame.  
  
If False, then a new DataFrame will be created and returned.  
  
By default, inplace=False.

**Return Value**

A DataFrame with rows or columns that contain missing values removed according to the provided parameters.

**Examples**

Consider the following DataFrame:

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[pd.np.NaN,2],"B":[3,4],"C":[5,6]})

df

Output:  
   A    B  C

0  NaN  3  5

1  2.0  4  6

**Removing rows with missing values**

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)()   # or axis=0 or axis="index"

Output:

   A    B  C

1  2.0  4  6

**Removing columns with missing values**

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(axis="columns")   # or axis=1

Output:

   B  C

0  3  5

1  4  6

Notice how column A was removed since it contained a missing value.

**Removing column with ALL missing values**

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[pd.np.NaN,2], "B":[3,4], "C":[pd.np.NaN,pd.np.NaN]})

df

Output:

   A    B  C

0  NaN  3  NaN

1  2.0  4  NaN

To remove columns whose values are all missing values, set how="all":

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(how="all" axis="columns")

output:

   A    B

0  NaN  3

1  2.0  4

**Setting a threshold**

Consider the following DataFrame:

import numpy as np

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":["a",np.nan,np.nan],"B":[3,4,np.nan]})

df

Output:

A B

0 a 3.0

1 NaN 4.0

2 NaN NaN

To remove columns with at least 2 non-NaN values, set thresh=2:

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(thresh=2, axis=1)

Output:

B

0 3.0

1 4.0

2 NaN

Notice how column A, which only had one non-missing value, was removed, while column B with 2 non-missing values was kept.

**Removing rows with missing values for certain columns only**

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[pd.np.NaN,2], "B":[3,4], "C":[pd.np.NaN,pd.np.NaN]})

df

Output:

   A    B  C

0  NaN  3  NaN

1  2.0  4  NaN

To remove rows where the value corresponding to column A is missing:

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(subset=["A"], axis="index")  
Output:

   A    B  C

1  2.0  4  NaN

Notice how only the first row (index=0) was removed despite the fact that both the two rows contained missing values. This is because, by specifying subset=["A"], the method only checks for missing values in column A.

**Removing columns with missing values for certain rows only**

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[pd.np.NaN,2], "B":[3,4], "C":[5,6]})

df

Output:

   A    B  C

0  NaN  3  5

1  2.0  4  6

To remove columns where the value corresponding to row index 1 is missing:

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(subset=[1], axis=1)   # or axis="columns"

Output:

   A    B

0  NaN  3

1  2.0  4

Notice how only column C was removed, despite the fact that column A also contained a missing value. This is because by specifying subset=[1], the method will only check for missing values at row index=1 (i.e. the second row). Since the value corresponding to column C in row index=1 was a missing value, the method removed column C.

**Removing rows/columns in-place**

To drop row(s) or column(s) in-place, we need to set inplace=True. This will directly modify the source DataFrame instead of creating and returning a new DataFrame.

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[pd.np.NaN,2], "B":[3,4], "C":[5,6]})

df

Output:

   A    B  C

0  NaN  3  5

1  2.0  4  6

We remove all rows containing missing value(s) with inplace=True:

df.[dropna](https://www.skytowner.com/explore/pandas_dataframe_dropna_method)(inplace=True)

df

Output:

   A    B  C

1  2.0  4  6

1. ***Imputation***

Imputation fills in the missing values with some number. For instance, we can fill in the mean value along each column. Different function can we used for this purpose.

Pandas DataFrame | fillna Method

Pandas' DataFrame.fillna(~) method fills NaN (missing values) with a specified value or with a filling rule.

**Parameters**

1. **value** | scalar or dict or Series or DataFrame | optional

The value to replace NaN. If a dict or Series is specified, then the key/index is the column label, and the value is the filler.

2. **method** | None or string | optional

The rule by which to fill NaN:

|  |  |
| --- | --- |
| **Value** | **Description** |
| "backfill" or "bfill" | Use the next non-NaN value to fill. |
| "pad" or "ffill" | Use the previous non-NaN value to fill. |
| None | Does not perform any filling. |

By default, method=None.

WARNING!

Only specify either value or method - not both.

3. **axis**| int or string | optional

Whether to fill each column or each row:

|  |  |
| --- | --- |
| **Axis** | **Description** |
| 0 or "index" | Fill column-wise |
| 1 or "columns" | Fill row-wise |

By default, axis=0. Note that this is only relevant if you specified method instead of value.

4. **inplace** | boolean | optional

* If True, then the method will directly modify the source DataFrame instead of creating a new DataFrame.
* If False, then a new DataFrame will be created and returned.

By default, inplace=False.

5. limit | None or int | optional

* If method is specified, then limit is the maximum number of consecutive NaN to fill. If there are more than limit number of consecutive NaN, then no fill will be performed.
* If method is not specified, then limit is the maximum number of fills to perform per row or column.

By default, limit=None.

**Return Value**

A DataFrame with NaN replaced by a filler.

**Examples**

Consider the following DataFrame:

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[None,5,6],"B":[7,None,8],"C":[9,None,None]})

df

Output:

   A    B    C

0  NaN  7.0  9.0

1  5.0  NaN  NaN

2  6.0  8.0  NaN

**Filling with a value**

To fill NaN with the value 10:

df.fillna(10)

   A     B     C

0  10.0  7.0   9.0

1  5.0   10.0  10.0

2  6.0   8.0   10.0

**Filling certain columns only**

To specify which columns to fill, provide a dict or Series like so:

df.fillna({"A":"\*","C":10})

A B C

0 \* 7.0 9.0

1 5 NaN 10.0

2 6 8.0 10.0

Notice how column B still has NaN here since the provided dict has no B key.

**Specifying a filling method**

Consider the same df as before:

df

   A    B    C

0  NaN  7.0  9.0

1  5.0  NaN  NaN

2  6.0  8.0  NaN

**Backward-fill**

To fill NaNs with the next non-NaN value column-wise:

df.fillna(method="bfill")   # or method="backfill"

   A    B    C

0  5.0  7.0  9.0

1  5.0  8.0  NaN

2  6.0  8.0  NaN

Notice how we still have some NaNs remaining. This is because there is no value that comes after the NaN, and so we don't have a filling value.

**Forward-fill**

To fill NaNs with the previous non-NaN value column-wise:

df.fillna(method="ffill")   # or method="pad"

   A    B    C

0  NaN  7.0  9.0

1  5.0  7.0  9.0

2  6.0  8.0  9.0

Again, we have a NaN at A[0] because there is no value that comes before the NaN, and so we don't have a filling value.

**Specifying the axis**

Just for your reference, here's the same df:

df

   A    B    C

0  NaN  7.0  9.0

1  5.0  NaN  NaN

2  6.0  8.0  NaN

By default, axis=0, which means that the filling method is applied column-wise:

df.fillna(method="ffill")      # axis=0

   A    B    C

0  NaN  7.0  9.0

1  5.0  7.0  9.0

2  6.0  8.0  9.0

We can perform the filling row-wise by setting axis=1:

df.fillna(method="ffill", axis=1)

   A    B    C

0  NaN  7.0  9.0

1  5.0  5.0  5.0

2  6.0  8.0  8.0

Note that the axis parameter is only relevant if you specify the method parameter instead of value.

**Specifying a limit**

**When parameter method is specified**

If method is specified, then limit is the maximum number of consecutive NaN to fill.

Just for your reference, here's the same df:

df

   A    B    C

0  NaN  7.0  9.0

1  5.0  NaN  NaN

2  6.0  8.0  NaN

To limit the number of consecutive NaNs to fill to 1:

df.fillna(method="ffill", limit=1)

   A    B    C

0  NaN  7.0  9.0

1  5.0  7.0  9.0

2  6.0  8.0  NaN

Notice how cell C[2] still has a NaN value. This is because we have 2 consecutive NaNs here, and since we specified limit=1, only the first one got filled.

**When parameter method is not specified**

When method is not specified, then the limit represents the maximum number of fills to perform per row or column. For instance, consider the following DataFrame:

df = pd.[DataFrame](https://www.skytowner.com/explore/pandas_dataframe_constructor)({"A":[None,5,None],"B":[7,None,8],"C":[9,None,None]})

df

A B C

0 NaN 7.0 9.0

1 5.0 NaN NaN

2 NaN 8.0 NaN

Performing a fill with limit=1 yields:

df.fillna(2, limit=1)

A B C

0 2.0 7.0 9.0

1 5.0 2.0 2.0

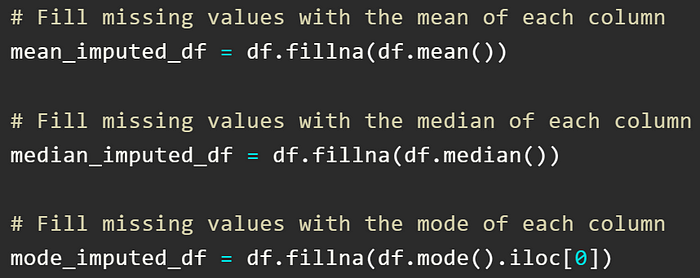
2 NaN 8.0 NaN

Notice how column A and C still have NaNs in them.

**Filling with Mean, Median, or Mode**

Imputing with summary statistics like mean, median, or mode is a common strategy. It’s important to consider the nature of the data and the presence of outliers.

* If data does not have outliers then replace missing values by mean because mean value will be effected by outliers
* If data has outliers then replace missing values by median because median will not be effected by outliers
* If data is categorical or text one can replace missing values by most frequent observation(mode)



Pandas DataFrame.replace() Method

**Description**

Pandas' DataFrame.replace(~) method replaces the specified values with another set of values.

Syntax:

*dataframe*.replace(*to\_replace*, *value*, inplace, limit, regex, method)

**Parameters**

**to\_replace** | string or regex or list or dict or Series or number or None  
The values that will be replaced.

**value** | number or dict or list or string or regex or None  
The value(s) that will replace to\_replace. By default, value=None.

**inplace** | boolean  
If True, then the method will directly modify the source DataFrame instead of creating a new DataFrame. If False, a new DataFrame will be created and returned. By default, inplace=False.

**limit** | int  
The maximum number of consecutive fills to perform. By default, limit=None.

**regex** | boolean or string  
If True, then to\_replace is interpreted as a regular expression. This requires to\_replace to be a string. By default, regex=False.

**method** | string or None  
The rule by which to replace to\_replace:  
 - "ffill" or "pad": Fills the value with the preceding row's value.  
 - "bfill": Fills the value with the next row's value.  
This parameter takes effect only when value=None. By default, method="pad".

**Return Value**

A DataFrame with the specified values replaced with your desired values.

**Examples**

**Replacing single value with a single value**

Consider the following DataFrame:  
Code:

df = pd.DataFrame({"A":[1,2], "B":[3,4]})

df

Output:

A  B

0  1  3

1  2  4

To replace all values of 1 with 5:

Code:  
df.replace(1, 5)

Output:

   A  B

0  5  3

1  2  4

**Replacing multiple values with a single value**

Consider the following DataFrame:

df = pd.DataFrame({"A":[1,2], "B":[3,4]})

df

Output:

   A  B

0  1  3

1  2  4

To replace all values of 1 and 2 with 5:  
df.replace([1,2], 5)

Output:

   A  B

0  5  3

1  5  4

**Replacing multiple values with corresponding values using an array**

Consider the following DataFrame:

df = pd.DataFrame({"A":[1,2], "B":[3,4]})

df

Output:

   A  B

0  1  3

1  2  4

To replace all values of 1 and 2 with 5 and 6, respectively:  
df.replace([1,2], [5,6])

Output:

   A  B

0  5  3

1  6  4

**Replacing multiple values using dict**

Consider the following DataFrame:

df = pd.DataFrame({"A":[1,2], "B":[3,4]})

df

Output:

   A  B

0  1  3

1  2  4

To replace all values of 1 and 3 with 5 and 6, respectively:  
df.replace({1:5, 3:6})

Output:

   A  B

0  5  6

1  2  4

**Replacing using regex**

Consider the following DataFrame:

df = pd.DataFrame({"A":["alex","bob"], "B":["cathy","doge"]})

df

To replace all values starting with 'a' with 'eric':  
df.replace("^a.\*", "eric", regex=True)

Output:

   A     B

0  eric  cathy

1  bob   doge

Notice how we had to enable regex by specifying regex=True.

**Replacing for certain columns only**

To Replacing single value with a single value

Consider the following DataFrame:

df = pd.DataFrame({"A":[1,2], "B":[1,2]})

df

Output:

   A  B

0  1  1

1  2  2

To replace all values of 1 with 3 for just column A. To do so, we must provide a dict, like follows:

df.replace({"A":1}, 3)

Output

   A  B

0  3  1

1  2  2

Notice how column B is unaffected despite containing a value of 1.

Replacing multiple values with corresponding values

Consider the following dataframe:

df = pd.DataFrame({"A":[1,2], "B":[3,4]})

df

OUTPUT:

   A  B

0  1  3

1  2  4

To replace values 1 and 2 with 5 and 6 respectively for just column A:

df.replace({"A":{1:5, 2:6}})

Output:

   A  B

0  5  3

1  6  4

**Replacing using fills**

When you don't explicitly provide the value parameter, the replace(~) function will automatically forward-fill the matched values, that is, replace the to\_replace with the preceding row's value.

Consider the following DataFrame:

df = pd.DataFrame({"A":["a","b","c"]})

df

Output:

   A

0  a

1  b

2  c

**Forward-fill**

To forward-fill all occurrences of "b":

df.replace("b", method="ffill")   # or simply leave out the method parameter.

Output:

   A

0  a

1  a

2  c

Notice how the value a, which is the value preceding the match (i.e. "b"), was used as the filler.

**Backward-fill**

To backward-fill all occurrences of "b":

df.replace("b", method="bfill")

Output:

   A

0  a

1  c

2  c

**Limit**

Consider the following DataFrame:

df = pd.DataFrame({"A":["a","b","b"]})

df

Output:

A

0 a

1 b

2 b

By default, limit=None, which means that there is no restriction on how many consecutive fills are allowed:

df.replace("b", method="ffill")

Output:

A

0 a

1 a

2 a

In contrast, setting limit=1 yields:

df.replace("b", method="ffill", limit=1)

Output:

A

0 a

1 a

2 b

Here, notice how b was filled only once. Also note that limit imposes a restriction on consecutive fills only.

**Replacing in-place**

To perform replacement in-place, we need to set inplace=True. This will directly perform the replace operation on the source DataFrame instead of creating a new one.

Consider the following DataFrame:

df = pd.DataFrame({"A":[1,2],"B":[3,4]})

Output:

   A  B

0  1  3

1  2  4

We replace all occurrences of 1 with 5 with inplace=True:

df.replace(1, 5, inplace=True)

Output:

   A  B

0  5  3

1  2  4

As shown in the output, the source DataFrame has been directly modified.

Pandas DataFrame | Interpolate

Pandas DataFrame.interpolate(~) method fills NaN using interpolated values.

**Parameters**

1. **method** | string | linear

The algorithm used for interpolation:

* "linear": simple linear interpolation.
* "time": interpolation using DatetimeIndex.
* "index" or "values": use the index to perform interpolation. See example below.
* "pad": use either the previous or next non-NaN value to fill. The direction can be set using limit\_direction.

In addition, you can also use the advanced interpolation methods like:

nearest, zero, slinear, quadratic, cubic, spline, barycentric, polynomial

Some of these methods require a argument to be passed, which you can do using \*\*kwargs like so:

df.interpolate(method="polynomial", order=5)

2. **axis** | int or string | optional

Whether to interpolate each row or column:

|  |  |
| --- | --- |
| **Axis** | **Description** |
| 0 or "index" | Interpolate each column |
| 1 or "columns" | Interpolate each row |

By default, axis=0.

3. **limit** | int | optional

The maximum number (inclusive) of consecutive NaN to fill. For instance, if limit=3, and there are 3 consecutive NaNs, then filling will be performed on the first two NaNs, and the third will be left as is.

4. **inplace** | boolean | optional

* If True, then the method will directly modify the source DataFrame instead of creating a new DataFrame.
* If False, then a new DataFrame will be created and returned.

By default, inplace=False.

5. **limit**\_direction | string | optional

The fill direction of NaN:

* "forward": use the previous non-NaN value to fill
* "backward": use the next non-NaN value to fill
* "both": use the next non-NaN value to fill if previous non-NaN value is unavailable, and vice versa.

This is only relevant if limit is specified. By default, limit\_direction="forward".

6. **limit\_area** | None or string | optional

The restriction imposed on filling:

* None: no restriction.
* "inside": only perform interpolation (i.e. when lower and upper bounds of the interval are defined)
* "outside": only perform extrapolation (i.e. when only one bound of the interval is defined)

By default, limit\_area=None.

7. **downcast** | "infer" or None | optional

Whether or not to downcast the resulting dtypes. By default, downcast=None.

8. **\*\*kwargs**

The keyword arguments to pass on to method.

**Return value**

A DataFrame with the NaN filled with interpolated values.

**Examples**

**Basic usage**

Consider the following DataFrame:

df = pd.DataFrame({"A":[3,np.nan,5,6],"B":[1,5,np.nan,9],"C":[1,5,np.nan,np.nan]})

df

A B C

0 3.0 1.0 1.0

1 NaN 5.0 5.0

2 5.0 NaN NaN

3 6.0 9.0 NaN

To fill NaN using linear interpolation:

df.interpolate() # method="linear"

A B C

0 3.0 1.0 1.0

1 4.0 5.0 5.0

2 5.0 7.0 5.0

3 6.0 9.0 5.0

Notice how the two NaN in column C were filled using forward-fill (default) instead since linear interpolation cannot be performed without an upper bound.

**Interpolating row-wise**

To interpolate row-wise, pass in axis=1 like so:

df.interpolate(axis=1)

A B C

0 3.0 1.0 1.0

1 NaN 5.0 5.0

2 5.0 5.0 5.0

3 6.0 9.0 9.0

**Interpolating using method=index**

Consider the following DataFrame

df = pd.DataFrame({"B":[5,np.nan,9]}, index=[5,10,30])

df

B

5 5.0

10 NaN

30 9.0

Performing simple linear interpolation yields:

df.interpolate() # method="linear"

B

5 5.0

10 7.0

30 9.0

Here, we get a 7 as the interpolated value because the difference between the lower and upper bound (4) is split up into 2 equally-distanced intervals.

In contrast, interpolating using method="index" instead gives:

df.interpolate(method="index")

B

5 5.0

10 5.8

30 9.0

Here, the difference between the lower and upper bound (4) is divided up not by the number of intervals there are, but by the difference of the index values (30-5=25). So, we end up with 5.8 because:

(4/25 \* 5) + 5 = 5.8

**Interpolation using method=time**

Consider the following DataFrame with a DatetimeIndex:

index\_date = pd.to\_datetime(["2020-12-01", "2020-12-02", "2020-12-15", "2020-12-31"])

df = pd.DataFrame({"A":[1,np.nan,np.nan,31]}, index=index\_date)

df

A

2020-12-01 1.0

2020-12-02 NaN

2020-12-15 NaN

2020-12-31 31.0

If we perform linear interpolation on df:

df.interpolate()

A

2020-12-01 1.0

2020-12-02 11.0

2020-12-15 21.0

2020-12-31 31.0

Here, the index is not taken into account - the lower bound is 1 and upper bound is 31, and the difference is evenly spaced out in 3 intervals.

To take into account the DatatimeIndex, pass in method="time":

df.interpolate(method="time")

A

2020-12-01 1.0

2020-12-02 2.0

2020-12-15 15.0

2020-12-31 31.0

Here, the bounds are still the same - lower bound is 1 and upper bound is 31. Instead of dividing the difference 30 by the number of intervals, we divide the difference by the length of time, which in this case is 30 days. This is why for instance, for day 15, we see an interpolated value for 15.

**Specifying limit direction**

Consider the following DataFrame:

df = pd.DataFrame({"A":[np.nan,np.nan,5], "B":[5,np.nan,9], "C":[5,np.nan,np.nan]})

df

A B C

0 NaN 5.0 5.0

1 NaN NaN NaN

2 5.0 9.0 NaN

By default, limit\_direction="forward", which means that we use the previous non-NaN value to fill NaN . To use the next non-NaN value to fill NaN, pass in limit\_direction="backward":

df.interpolate(limit\_direction="backward")

A B C

0 5.0 5.0 5.0

1 5.0 7.0 NaN

2 5.0 9.0 NaN

Notice how for both forward and backward, we may still end up with NaN values when there are no previous/next non-NaN values. We can prevent this by setting limit\_direction="both", which ensures that if the previous non-NaN value is unavailable, then the next non-value would be used, and vice versa:

df.interpolate(limit\_direction="both")

A B C

0 5.0 5.0 5.0

1 5.0 7.0 5.0

2 5.0 9.0 5.0

**Downcasting the resulting DataFrame**

By default, downcast=None, which means that even no casting will be performed if a column type can be casted to a more specific type.

For example, consider the following DataFrame:

df = pd.DataFrame({"A":[np.nan,5], "B":[5,np.nan]})

df

A B

0 NaN 5.0

1 5.0 NaN

Performing interpolation yields:

df.interpolate() # downcast=None

A B

0 NaN 5.0

1 5.0 5.0

Checking the column types of the resulting DataFrame:

df.interpolate().dtypes

A float64

B float64

dtype: object

In this scenario, it is possible to use a more specific type, namely int, as the column type of B. To perform this downcast, set downcast="infer":

df.interpolate(downcast="infer").dtypes

A float64

B int64

dtype: object

1. ***Handling Duplicate Values***

**drop\_duplicates()**:

drop\_duplicates() is used to remove duplicate rows from a dataset. Duplicates can skew analysis, so identifying and removing them is crucial.



You can also specify a subset of columns to check for duplicates using

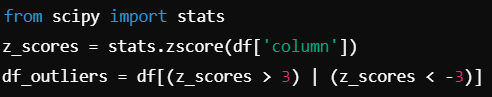


1. **Identifying and Handling Outliers**
2. **Z-Score Method**

**Z-Score**: The z-score is a statistical measure that describes a data point's relationship to the mean of the dataset. It indicates how many standard deviations a data point is from the mean. A high absolute z-score (typically greater than 3 or less than -3) indicates a potential outlier.

Formula:

where X is the data point, μ is the mean, and σ is the standard deviation.

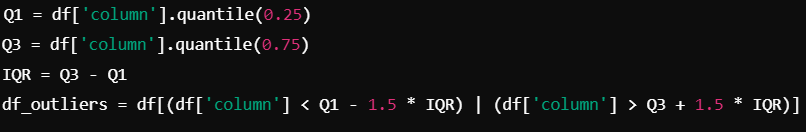


1. **Quantile Method (Interquartile Range - IQR)**

**Interquartile Range (IQR)**: This method defines outliers as values that fall below the 25th percentile (Q1) or above the 75th percentile (Q3) by more than 1.5 times the interquartile range.

Formula:

Outliers are defined as



|  |  |  |
| --- | --- | --- |
| Function | Purpose | Syntax |
| dropna() | Removes missing values | df.dropna(axis=0) |
| fillna() | Fills missing values with a constant or method | df.fillna(value) |
| interpolate() | Fills missing values via interpolation | df.interpolate(method='linear') |
| drop\_duplicates() | Removes duplicate rows | df.drop\_duplicates() |
| pd.to\_numeric() | Converts string to numeric | pd.to\_numeric(df['string\_col'], errors='coerce') |
| pd.to\_datetime() | Converts string to datetime | pd.to\_datetime(df['string\_date\_col']) |
| astype() | Converts a column to another datatype | df['float\_col'].astype(int) |
| pd.concat() | Concatenates DataFrames | pd.concat([df1, df2], axis=0) |
| pd.merge() | Merges DataFrames based on key | pd.merge(df1, df2, on='key') |
| join() | Joins DataFrames based on index | df1.join(df2, how='left') |
| stats.zscore() | Calculates z-score for outlier detection | stats.zscore(df['column']) |
| quantile() | Computes quantiles for IQR-based outlier detection | df['column'].quantile(0.25) |

TASK TO SUBMIT:

Read and understand all the functions above.

You have to submit a notebook where you run all the example given in this lab manual with one line of comment what in each code cell mentioning the what you understand about it.

**Bouns: You can apply all possible functions on iris\_dataset.**

1. Submission Instructions

Always read the submission instructions carefully.

* Rename your Jupyter notebook to your roll number and download the notebook as **.ipynb**

extension.

* To download the required file, go to **File->Download .ipynb**
* Only submit the **.ipynb** file. DO NOT **zip** or **rar** your submission file
* Submit this file on Google Classroom under the relevant assignment.
* Late submissions will not be accepted.