Data preprocessing with Python Pandas

Preprocessing is the process of doing a pre-analysis of data, in order to transform them into a standard and normalized format. Preprocessing involves the following aspects:

1. missing values
2. data formatting
3. data normalization
4. data standardization

1. Data Missing values

Step-1: Import data

Firstly, import data using the pandas library and convert them into a dataframe. Through the head(10) method we print only the first 10 rows of the dataset.

```
import pandas as pd
df = pd.read_csv(location of file)
df.head(10)
```

Step-2: Identify missing values

It is noted that the dataset contains missing values, we can use the function isna() , which returns if an cell of the dataset if NaN or not. Then we can count how many missing values there are for each column.

```
df.isna().sum()
```

Now we can count the percentage of missing values for each column, simply by dividing the previous result by the length of the dataset (len(df)) and multiplying per 100.

```
df.isna().sum()/len(df)*100
```

Step-3: Filling the missing values

It is necessary to fill in the missing data for the data set. The following methods are used to fill the missing cells.

- ✓ **drop missing values**
- ✓ **replace the missing value with a value**

Drop missing values:

Dropping missing values can be one of the following alternatives:

- remove rows having missing values
- remove the whole column containing missing values .

We can use the dropna() by specifying the axis to be considered.

If we set axis = 0 we drop the entire row,

```
df.dropna(axis=0)
```

if we set axis = 1 we drop the whole column.

```
df.dropna(axis=1)
```

However, removed values are not applied to the original dataframe, but only to the result. We can use the argument `inplace=True` in order to store changes in the original dataframe.

```
df.dropna(axis=1,inplace=True).
```

As an alternative, we can specify only the column on which the dropping operation must be applied. This can be achieved through the subset parameter, which permits to specify the subset of columns where to apply the dropping operation.

```
print(df['1_b'].dropna())
```

Now we can check whether there are still missing values for the column '1_b' .

```
df.isna().sum()/len(df)*100
```

Replace missing values:

A good strategy when dealing with missing values involves their replacement with another value. Usually, the following strategies are adopted:

- **Numerical values** replace the missing value with the average value of the column.
- **categorial values** replace the missing value with the most frequent value of the column .

In order to replace missing values, three functions can be used: `fillna() , replace() and interpolate().`

The fillna() function replaces all the NaN values with the value passed as argument. For example, for numerical values, all the NaN values in the numeric columns could be replaced with the average value.

Numeric columns:

Firstly, we select numeric columns. Then, we fill the NaN values of numeric columns with the average value, given by the df.mean( ) function.

```
df['1_b'] = df['1_b'].fillna(df['1_b'].mean())
```

Now, we can check whether the NaN values in numeric columns have been removed.

```
df.isna().sum()/len(df)*100
```

Categorial columns:

Now we can replace all the missing values with the most frequent value. We can use the mode() function to calculate the most frequent value. We use the fillna() function to replace missing values, but we could use also the replace(old_value, new_value) function.

```
df[boolean_columns].fillna(df.mode()[0])
```

Now our dataset does not contain any missing value.

```
df.isna().sum()/len(df)*100
```

Interpolation:

Another solution to replace missing values involves the usage of other functions, such as linear interpolation. In this case, we could replace a missing value over a column, with the interpolation between the previous and the next ones. This can be achieved through the use of the `interpolate()` function.

We select only numeric columns.

```
df.interpolate().to_string()
df.head(10)
```

Now we can apply the `interpolate ()` function to numeric columns, by setting also the limit direction to forward. This means that the linear interpolation is applied starting from the first row until the last one.

```
df[numeric_columns] = df[numeric_columns].interpolate(method
              ='linear', limit_direction ='forward')
```

For example, in line 6 the column '2_b' , which was NaN before the interpolation, now assumes the value 0.95, which is the interpolation between 0.90 (line 4) and 1.00 (line 6).

Code

```
import pandas as pd
print("------import the data from file marks sheet wise-----------------------")
df=pd.read_excel("C:\\Users\\admin\\Desktop\\marks.xlsx")
df=pd.read_excel('C:\\Users\\admin\\Desktop\\marks.xlsx',sheet_name='B_section')
df=pd.read_excel('C:\\Users\\admin\\Desktop\\marks.xlsx',sheet_name='C_section')
print(df)
print(df.head())
print(df.head(10))
print(df.tail(10))
print(df.to_string())
print ("--------------------read columns  headers of dataframe----------------")
print(df.columns.ravel())
print ("--------------------read only columns  of dataframe-------------------")
print(df['3_b'])
print(df['1_b'])
print("-------------------Identify missing values and percentage-------------")
print(df['1_b'].isna().sum())
print(df['1_b'].isna().sum()/len(df['1_b'])*100)
print("--------------------drop missing values-------------------------------")
print(df.dropna())
print(df.dropna(axis=0))
print(df.dropna(axis=1))
#print(df.dropna(axis=1),inplace=True)
print(df.isna().sum()/len(df)*100)
print("---------------------Replace missing values-------------------------")
print("-------------------------------- with mean()function----------------")
print(df['1_b'].fillna(df['1_b'].mean()))
print("----------------------------------with mode()function---------------")
print(df['2_a'].fillna(df['1_b'].mode()[0]))
print("----------------------------------interpolate()function-------------")
#print(df['2_a'].interpolate(method ='linear', limit_direction ='forward'))
print(df['2_a'].fillna(df['2_a'].interpolate()))
```

*****

2. Data formatting

Data formatting is the process of transforming data into a common format, which helps users to perform comparisons.

Import data:

Firstly, import data using the pandas library and convert them into a dataframe. Through the head (10) method we print only the first 10 rows of the dataset and drop all the missing values through the dropna()function.

```
import pandas as pd
df = pd.read_csv(' Location of the file')
df.head(5)
df.dropna(inplace=True)
```

Incorrect data types

First of all, we should make sure that every column is assigned to the correct data type. This can be checked through the property dtypes.

```
df.dtypes
```

In our case we can convert the column '1_a' to string by using the function astype() as follows:

```
df[‘1_a’] = df[‘1_a’].astype('string')
```

Make the data homogeneous:

This aspect involves categorical and numeric data.

- Categorical data should have all the same formatting style, such as lower case. In order to format all categorical data to lower case, we can use the following statement:

```
df['Tweet Content'] = df['Tweet Content'].str.lower()
```

Other techniques to make homogeneous categorical data involve the following aspects:

- ✓ remove white space everywhere:
- ✓ `df[’Tweet Content’] = df[’Tweet Content’].str.replace(‘ ‘, ‘’)`
- ✓ remove white space at the beginning of string:

```
df['Tweet Content'] =df['Tweet Content'].str.lstrip()
```

- ✓ remove white space at the end of string:

```
df['Tweet Content'] =df['Tweet Content']].str.rstrip()
```

- ✓ remove white space at both ends:
  `df['Tweet Content'] = df['Tweet Content'].str.strip() .`
- Numeric data should have for example the same number of digits after the point.

For example, if we want 2 decimal points, we can run the following command:

```
df['User Following'] = df['User Following'].round(2) .
```

Other techniques to make homogeneous numeric data include:

- Round up — Single DataFrame column —
  `df['User Following'] = df['User Following'].apply(np.ceil)`
- Round down — Single DataFrame column —

```
df['User Following'] = df['User Following'].apply(np.floor)
```

Code

```
import pandas as pd
print("Data Import")
df=pd.read_csv("G:\VVIT    (2018-19)\VVIT-2023-24\I Sem\ANN\studentsmarks.csv")
df=pd.read_excel('C:\\Users\\admin\\Desktop\\marks.xlsx',sheet_name='C_section')
df.dropna(inplace=True)
print(df)
print("Incorrect data types")
print(df["1_a"])
print(df["1_a"].dtypes)
print(df["1_a"].astype(int))
print(df["name"].astype('string'))
print("Make the data homogeneous")
print(df["name"].str.lower())
print("remove white space everywhere")
print(df["name"].str.replace(' ', ''))
print("remove white space at the beginning of string")
print(df["name"].str.lstrip())
print("remove white space at the end of string")
print(df["name"].str.rstrip())
print("remove white space at both ends")
print(df["name"].str.strip())
print("2 decimal points")
print(df["1_a"].round(2) )
print("Round up")
import numpy as np
print(df["1_a"].apply(np.ceil))
print("Round down")
print(df["1_a"].apply(np.floor) )
```

*****

3. Data-Normalisation

Data Normalisation involves adjusting values measured on different scales to a common scale. When dealing with dataframes, data normalization permits to adjust values referred to different columns to a common scale.

Normalization applies only to columns containing numeric values. Five methods of normalization exist:

1. single feature scaling
2. min max
3. z-score
4. log scaling
5. clipping

Step-1: Data Import

First of all, we need to import the Python pandas library and read the dataset through the read_csv() function. Then we can drop all the columns with NaN values. This is done through dropna() function.

```
import pandas as pd
df = pd.read_csv(' Location of the file')
df.dropna(axis=1,inplace=True)
df.head(10)
```

Single Feature Scaling

Single Feature Scaling converts every value of a column into a number between 0 and 1. The new value is calculated as the current value divided by the max value of the column.

```
df['A1'] = df['A1']/df['A1'].max()
```

Min Max

Similarly to Single Feature Scaling, Min Max converts every value of a column into a number between 0 and 1. The new value is calculated as the difference between the current value and the min value, divided by the range of the column values.

For example, we can apply the min max method to the column A1 .

```
df[‘A1'] = (df[‘A1'] - df[‘A1'].min())/
           (df[‘A1'].max() -df[‘A1'].min())
```

z-score

Z-Score converts every value of a column into a number around 0. The new value is calculated as the difference between the current value and the average value, divided by the standard deviation. The average value of a column can be obtained through the mean() function, while the standard deviation through the std() function.

For example, we can calculate the z-score of the column T .

```
            df['T']=df['T']-df['T'].mean())/df['T'].std()
```
Now we can calculate the minimum and maximum value obtained by the zscore transformation:
```
            df['T'].min()
```
and:
```
            df['T'].max()
```
Log Scaling

Log Scaling involves the conversion of a column to the logarithmic scale. If we want to use the natural logarithm, we can use the log() function of the numpy library.

For example, we can apply log scaling to the column 'T' . We must deal with log(0) because it does not exist. We use the lambda operator to select the single rows of the column.
```
import numpy as np
df['T'] = df['T'].apply(lambda x: np.log(x) if x != 0 else 0)
```

Clipping

Clipping involves the capping of all values below or above a certain value. Clipping is useful when a column contains some outliers. We can set a maximum vmax and a minimum value vmin and set all outliers greater than the maximum value to vmax and all the outliers lower than the minimum value to vmin .

For example, we can consider the column 'T' and we can set vmax = 10000 and vmin = 10 .
```
            vmax = 10000
            vmin = 10
            df['T'] =df['T'].apply(lambda x:
            vmax if x > vmax else vmin if x < vmin else x)
```
Summary:

- if you want an output between 0 and 1, you should use single feature scaling or min max
- if you want an output around 0, which includes also negative values, you should use z-score
- if your data contain many outliers, you can use clipping.

```
# Code
import pandas as pd
print("Data Import")
df=pd.read_csv("G:\VVIT(2018-19)\VVIT-2023-24\ISem\ANN\studentsmarks.csv")
a=pd.DataFrame(df)
print(df)
print(df.to_string())
print(df.head())
print(df.head(3))
print(df["T"])
print("Single Feature Scaling")
print(df['T']/df['T'].max())
print("Min Max")
print((df['T'] -df['T'].min())/(df['T'].max() -df['T'].min()))
print("z-score")
print((df['T']-df['T'].mean())/df['T'].std())
print("Log Scaling")
import numpy as np
print(df['T'].apply(lambda x: np.log(x) if x != 0 else 0))
print("Clipping")
vmax = 10000
vmin = 10
print (df['T'].apply(lambda x: vmax if x > vmax else vmin if x < vmin else x))
```

4. Data Standardization

Standardization is transforms data to have a mean of zero and a standard deviation of 1.

Standardization is done through a z-score transformation, where the new value is calculated as the difference between the current value and the average value, divided by the standard deviation.

The average value of a column can be obtained through the mean ( ) function, while the standard deviation through the std ( ) function.

For example, we can calculate the z-score of the column T.

```
df['T']=df['T']-df['T'].mean())/df['T'].std()
```

```
# Code
import pandas as pd
print("Data Import")
df=pd.read_csv("G:\VVIT(2018-19)\VVIT-2023-24\ISem\ANN\studentsmarks.csv")
a=pd.DataFrame(df)
print(df)
print(df.to_string())
print(df.head())
print(df.head(3))
print(df["T"])
print("z-score")
print((df['T']-df['T'].mean())/df['T'].std())
```

5. Data binning

Data binning (or bucketing) groups data in bins (or buckets), in the sense that it replaces values contained into a small interval with a single representative value for that interval. Sometimes binning improves accuracy in predictive models. Binning can be applied to convert numeric values to categorical or to sample numeric values.

- **convert numeric to categorical** includes binning by distance and binning by frequency.
- **reduce numeric values** includes quantisation (or sampling).