Data Cleaning / Data Imputation

Data Cleaning

Data cleaning means fixing bad data in your data set.

Dirty data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

Empty Cells / Missing Values

- **Complete Case Analysis (CCA)**
- **Handling Missing Numerical Data**
  - Mean, Median, Mode imputation
  - End of Distribution Imputation
  - Arbitrary Value Imputation
- **Handling Missing Categorical Data**
  - Frequent Category Imputation
  - Missing Category Imputation

Empty Cells / Missing Values

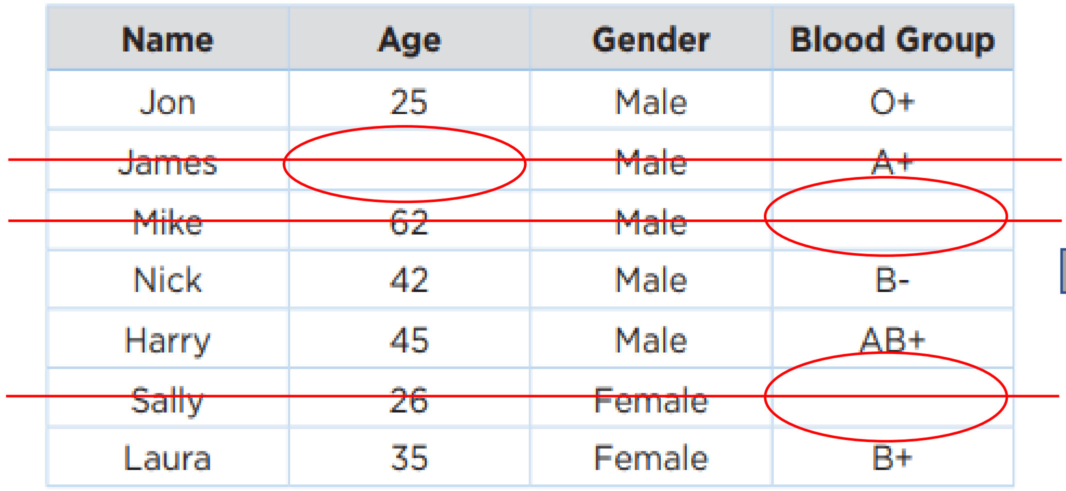
- **Complete Case Analysis (CCA)**
- **Handling Missing Numerical Data**
  - Mean, Median, Mode imputation
  - End of Distribution Imputation
  - Arbitrary Value Imputation
- **Handling Missing Categorical Data**
  - Frequent Category Imputation
  - Missing Category Imputation

1. Complete Case Analysis

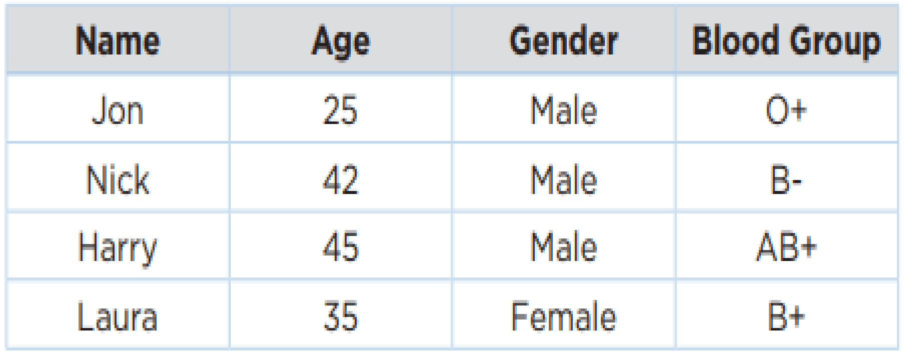
Complete case analysis (CCA)

- Also known as list-wise deletion,
- It is the most basic technique for handling missing data.
- In CCA, you simply remove all the rows or records where any column or field contains a missing value.
- Only those records are processed where an actual value is present for all the columns in the dataset.
- CCA can be applied to handle both numerical and categorical missing values.

Data Cleaning: Missing Values -> EAMPLE



Name	Age	Gender	Blood Group
Jon	25	Male	O+
~~James~~		~~Male~~	~~A+~~
~~Mike~~	~~62~~	~~Male~~	
Nick	42	Male	B-
Harry	45	Male	AB+
~~Sally~~	~~26~~	~~Female~~	
Laura	35	Female	B+



Name	Age	Gender	Blood Group
Jon	25	Male	O+
Nick	42	Male	B-
Harry	45	Male	AB+
Laura	35	Female	B+

Advantages

- The assumption behind the CCA is data is missing at **random.**
- CCA is extremely simple to apply, and **no statistical technique** is involved.
- Finally, the distribution of the variables is also preserved.

Disadvantages

- If a dataset contains a **large number of missing values, a large subset of data will be removed** by CCA.
- If the values are not missing randomly, CCA can create a **biased dataset.**
- Finally, statistical models trained on a dataset on which CCA is applied are not capable of handling missing values in production.

Data Cleaning: Empty Cells / Missing Values : Practice

Remove Rows

- One way to deal with empty cells is to **remove** rows that contain empty cells.
- This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.
- Example
  - Return a new Data Frame with no empty cells:

```
import pandas as pd
df = pd.read_csv('data.csv’)
new_df = df.dropna()
print(new_df.to_string())
```

- The **`dropna()`** method returns a *new* DataFrame, and will not change the original.

- If you want to change the original DataFrame, use the **`inplace = True`** argument:

Example
Remove all rows with NULL values:

```
import pandas as pd

df = pd.read_csv('data.csv')

df.dropna(inplace = True)

print(df.to_string())
```

- The **dropna(inplace = True)** will NOT return a new DataFrame, but it will remove all rows contains NULL values from the original DataFrame.

Replace Empty Values

- Another way of dealing with empty cells is to insert a ***new*** value instead.
- This way you do not have to delete entire rows just because of some empty cells.
- The **`fillna()`** method allows us to replace empty cells with a value:

Example

- Replace NULL values with the number 130:

```
import pandas as pd

df = pd.read_csv('data.csv’)

df.fillna(130, inplace = True)
```

Replace Only For a Specified Columns

- The example above replaces all empty cells in the whole Data Frame.
- To only replace empty values for **one column**, specify the ***column name*** for the DataFrame:
- Example

  Replace NULL values in the "Calories" columns with the number 130:

```
import pandas as pd
df = pd.read_csv('data.csv’)
df["Calories"].fillna(130, inplace = True)
```

Empty Cells / Missing Values

- **Complete Case Analysis (CCA)**
- **Handling Missing Numerical Data**
  - Mean, Median, Mode imputation
  - End of Distribution Imputation
  - Arbitrary Value Imputation
- **Handling Missing Categorical Data**
  - Frequent Category Imputation
  - Missing Category Imputation

DATA IMPUTATION:

Handling Missing Values using Statistical Techniques

How to check missing values in the dataset?

- The command to check the presence of missing values in the dataset.
- **Dataset.isnull().mean()**
- The above command shows the percentage (%) of missing values in each column in the dataset.
- Ex:

```
import numpy as np
import pandas as pd
booc= pd.read_csv("D:/IIIT-NR/Autumn-2021/Data Preprocessing/Demonstrations/Book.csv")
booc
booc.isnull().mean()
```

Imputation: Handling Missing Numerical Data

- One of the most commonly occurring data type is the numeric data, which consists of numbers.
- To handle missing numerical data, statistical techniques can be used.
- The use of statistical techniques or algorithms to replace missing values with statistically generated values is called **imputation**.

- **Separation of Numerical Data :** Separate numerical data from the dataset and fill the missing values.
  - Ex
    - **Booc_numeric = Book[[“Price”, “YoP”]]**
    - Booc_numeric

1. Mean, Median, or Mode Imputation

- A common way to replace empty cells, is to calculate the mean, median or mode value of the column.
- Pandas uses the **`mean()`** **`median()`** and **`mode()`** methods to calculate the respective values for a specified column:
- Example

  Calculate the MEAN, and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv’)
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)
```

Example

Calculate the MEDIAN, and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace = True)
```

Example

Calculate the MODE, and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mode()[0]
df["Calories"].fillna(x, inplace = True)
```

Contd…

- The mean and median imputation can affect the data distribution for the columns containing the missing values.
- Specifically, the **variance of the column is decreased by mean and median imputation** now since more values are added to the center of the distribution.

Advantages:

- Mean and median imputations are easy to implement and are a useful strategy to quickly obtain a large dataset.
- Furthermore, the mean and median imputations can be implemented during the production phase.

Disadvantages:

- The biggest disadvantage of mean and median imputation is that it affects the default data distribution and variance and covariance of the data.

Recommendations

- Mean and Median imputation could be used for missing numerical data in case the data is missing at random.
- If the data is normally distributed, mean imputation is better, or else median imputation is preferred in case of skewed distributions.

2. End of Distribution Imputation (1/4)

- For randomly missing data, the most commonly used techniques are end of distribution/ end of tail imputation.
- In the end of tail imputation, a value is chosen from the tail end of the data. This value signifies that the actual data for the record was missing. Hence, data that is not randomly missing can be taken to account while training statistical models on the data.

2. End of Distribution Imputation (2/4)

Normally Distributed Data:

- The end of distribution value can be calculated using mean and standard deviations of the attribute.
- **EoD_Imp_value = Attribute_Mean+3 *Attribute_standard deviation**

Skewed data distributions:

- The Inter Quartile Rule can be used to find the tail values.
- IQR = 75th Quantile – 25th Quantile
- **Upper IQR Limit = 75th Quantile + 1.5 * IQR**
- **Lower IQR Limit = 25th Quantile – 1.5 * IQR**

2. End of Distribution Imputation (3/4)

Check data is Normally Distributed or not

- To check the data is normally distributed use **hist()**.
  - td1.Age.hist()
  - td1.Age.hist(bins=50)
- **Compute end of distribution value**

```
eod_value = td1.Age.mean() + 3 * td1.Age. std()
print(eod_value)
```

- Now the missing values in the age column can be replaced by the computed end of tail value.

```
td1['age_eod'] = td1.Age.fillna(eod_value)
td1.head(20)
```

2. End of Distribution Imputation (4/4)

Advantages

- Can be applied to the dataset where values are not missing at random.
- Its simplicity to understand, ability to create big datasets in a short time, and applicability in the production environment.

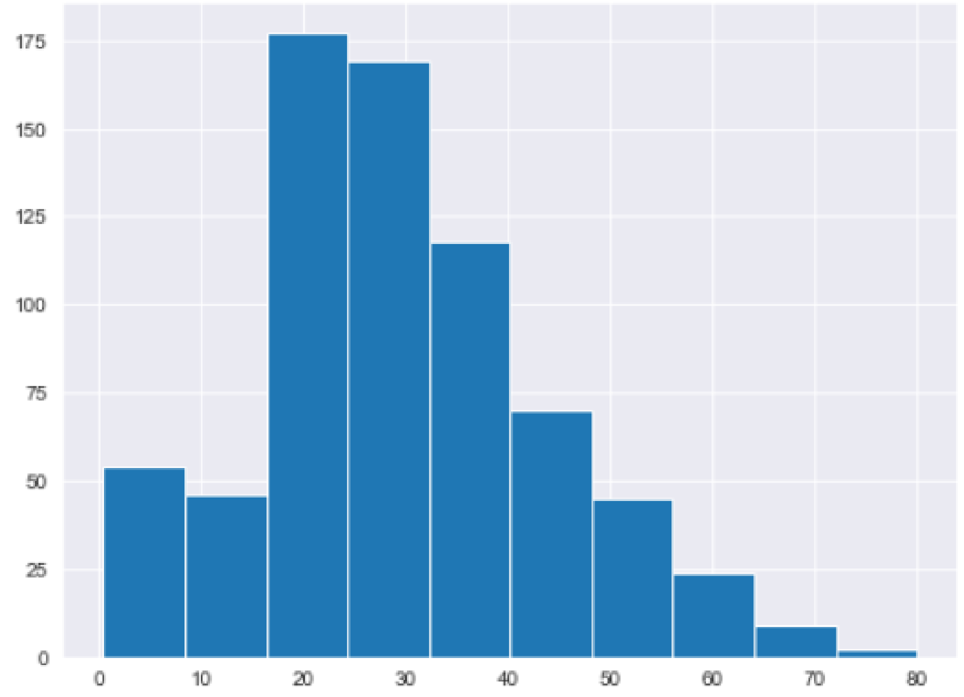
Disadvantages:

- The distortion of data distribution, variance, and covariance.

3. Arbitrary Value Imputation (1/2)

- In arbitrary value imputation, the values used to replace missing values are selected arbitrarily.
- The arbitrary values are selected in a way that **they do not belong to the dataset; rather, they signify the missing values.**
- A good value to select is **99, 999**, or **any number containing 9s**.
- If the dataset contains only positive value, a **−1** can be chosen as an arbitrary number.
- Use **hist()** function to know the data distribution

```
td1.Age.hist()
```



  - The output shows that the maximum positive value is around 80. Therefore, 99 can be a very good arbitrary value.
  - Furthermore, since the age column only contains positive values, −1 can be another very useful arbitrary value. Let's replace the missing values in the age column first by 99, and then by −1

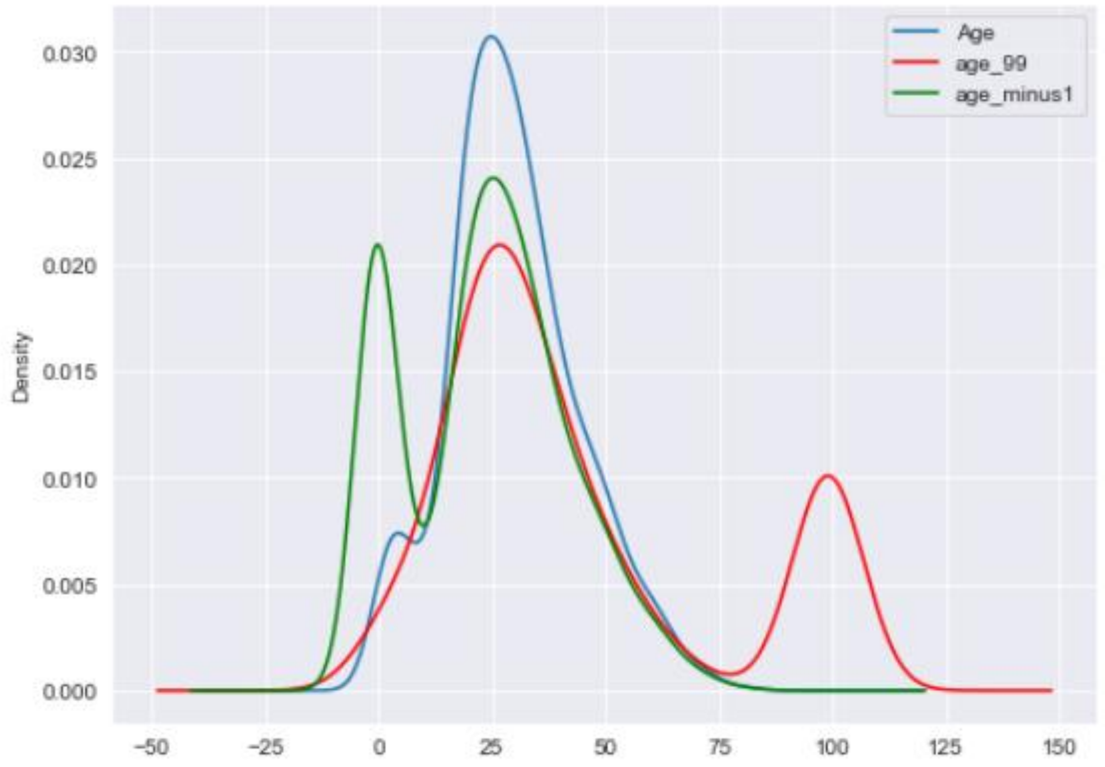
3. Arbitrary Value Imputation (2/2)

- The final step is to plot the kernel density plots for the original age column and for the age columns where the missing values are replaced by 99 and −1.

```
#### Plot KDE
plt.rcParams["figure.figsize"] = [8,6]
fig = plt.figure()
ax = fig.add_subplot(111)


td1['Age'] .plot(kind='kde', ax=ax)
td1['age_99'] .plot(kind='kde', ax=ax, color='red')
td1['age_minus1'] .plot(kind='kde', ax=ax, color='green')


lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



Empty Cells / Missing Values

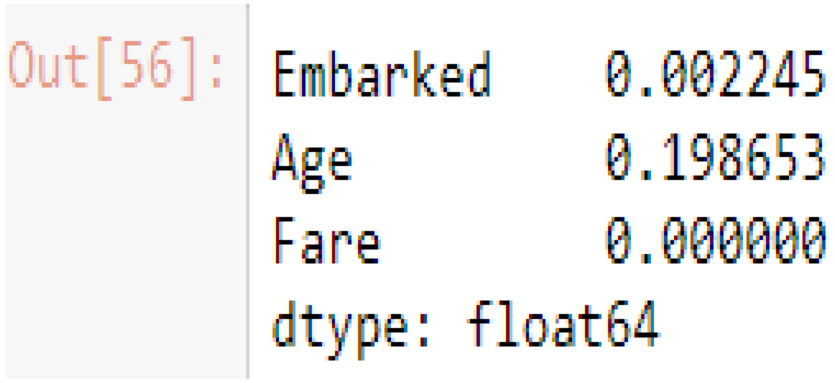
- **Complete Case Analysis (CCA)**
- **Handling Missing Numerical Data**
  - Mean, Median, Mode imputation
  - End of Distribution Imputation
  - Arbitrary Value Imputation
- **Handling Missing Categorical Data**
  - Frequent Category Imputation
  - Missing Category Imputation

1. Frequent Category Imputation (1/5)

- **Frequent Category Imputation**
- One of the most common ways of handling missing values in a categorical column is to replace the missing values with the most frequently occurring values, i.e., the mode of the column.
- It is also called as **mode imputation.**
- STEP1: Identify the categorical columns and separate them
- STEP2: Calculate mean for the dataset

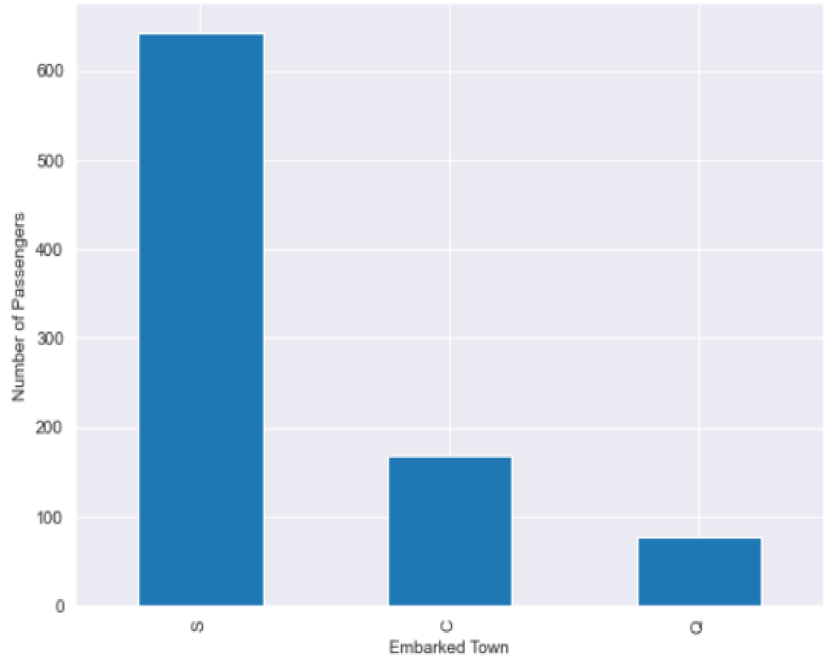
1. Frequent Category Imputation (2/5)

```
td2 = TD[[“Embarked", “Age", “Fare"]]
td2.head()
td2.isnull().mean()
```



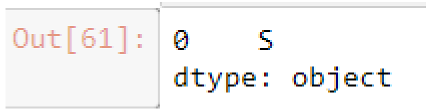
```
Out[56]: Embarked    0.002245
         Age         0.198653
         Fare        0.000000
         dtype: float64
```

- The output shows that embarked and age columns have missing values. The ratio of missing values for the embarked column is very less.
- Let’s plot the bar plot that shows each category in the embarked column against the number of passengers.



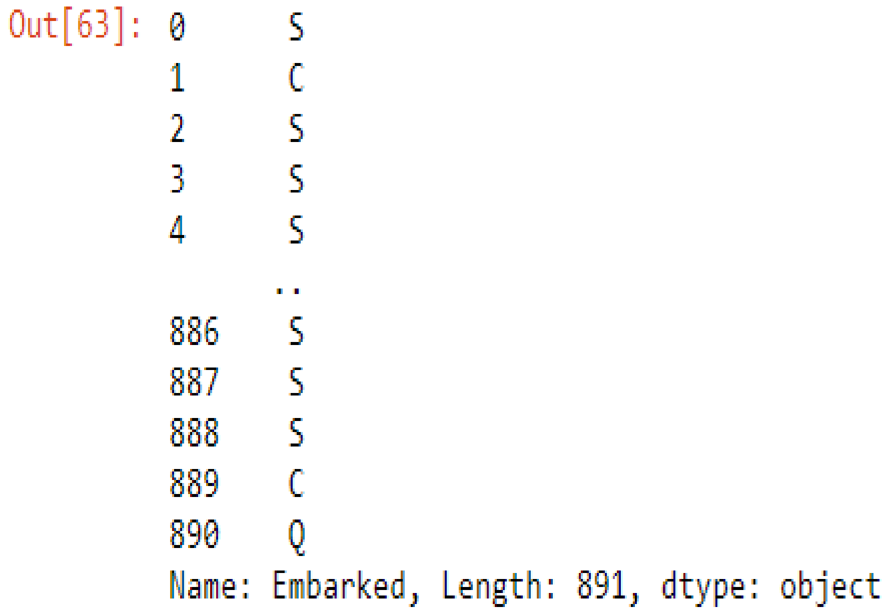
1. Frequent Category Imputation (3/5)

- Now, let’s check if Southampton is actually the mode value for the embarked column or not
- **td2.embarked.mode()**



```
Out[61]: 0    S
         dtype: object
```

- Next, simply replace the missing values in the Embarked column by ‘S’
- **td2.Embarked.fillna(‘S’)** # td2.Embarked.fillna(‘S’ , inplace=True)

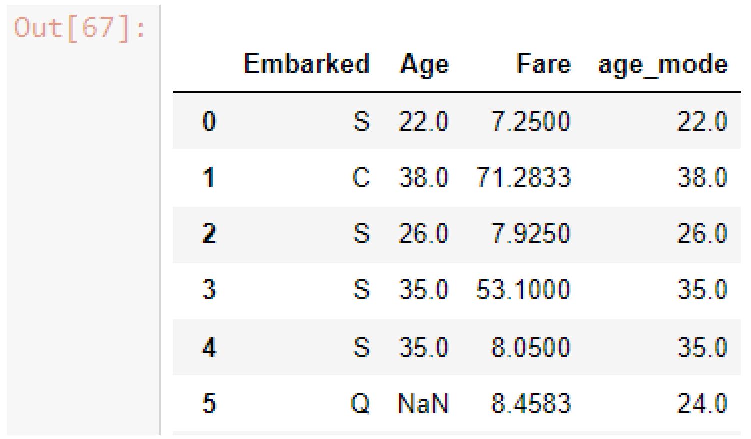


```
Out[63]: 0      S
         1      C
         2      S
         3      S
         4      S
               ..
         886    S
         887    S
         888    S
         889    C
         890    Q
         Name: Embarked, Length: 891, dtype: object
```

1. Frequent Category Imputation (4/5)

- Calculate mode on 'Age' column

```
td2.Age.mode()
td2['age_mode'] = td2.Age.fillna(24)
td2.head(20)
```



Out[67]:

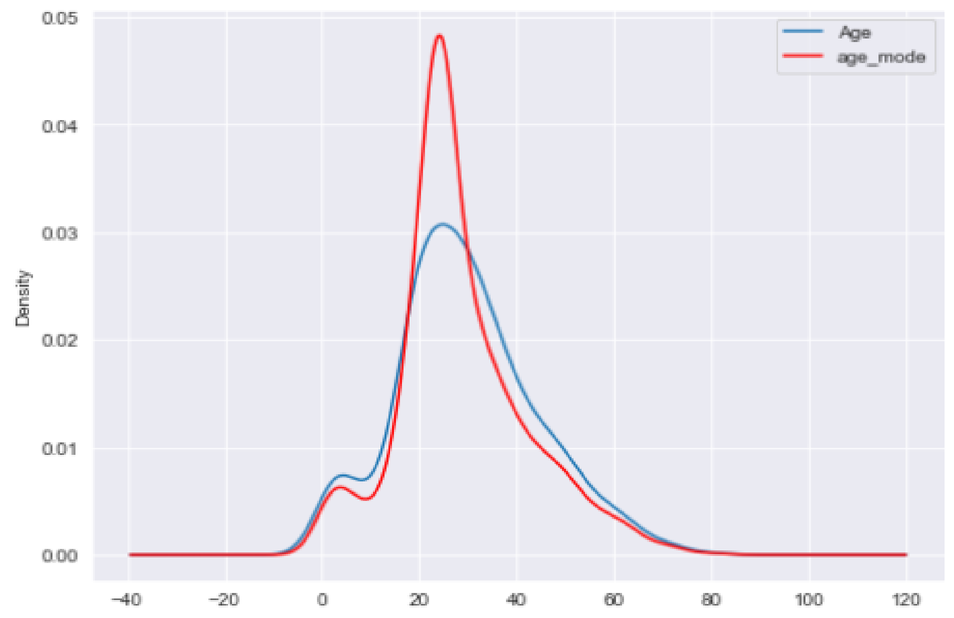
	Embarked	Age	Fare	age_mode
0	S	22.0	7.2500	22.0
1	C	38.0	71.2833	38.0
2	S	26.0	7.9250	26.0
3	S	35.0	53.1000	35.0
4	S	35.0	8.0500	35.0
5	Q	NaN	8.4583	24.0

```
# let's plot the kernel density estimation plot for the
#original age column and the age column that contains the
# mode of the values in place of the missing values.

plt.rcParams["figure.figsize"] = [8,6]
fig = plt.figure()
ax = fig.add_subplot(111)

td2['Age'] .plot(kind='kde', ax=ax)
td2['age_mode'] .plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

```
Out[68]: <matplotlib.legend.Legend at 0x1b521e48a00>
```



1. Frequent Category Imputation (5/5)

Advantages:

- The frequent category imputation is easier to implement on large datasets. Frequent category distribution doesn’t make any assumption on the data and can be used in a production environment.

Disadvantages:

- The downside of frequent category imputation is that it can overrepresent the most frequently occurring category in case there are too many missing values in the original dataset. In the case of very small values in the original dataset, the frequent category imputation can result in a new label containing rare values.

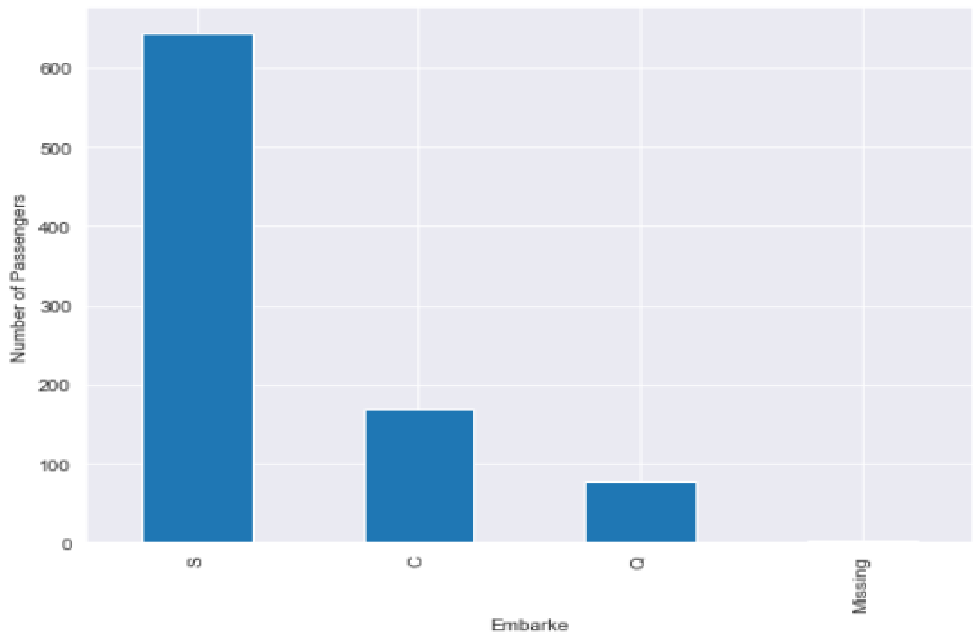
Missing Category Imputation

- Missing category imputation is similar to arbitrary value imputation. In the case of categorical value, missing value imputation adds an arbitrary category, e.g., **missing** in place of the missing values.
- Take a look at an example of missing value imputation.

```
td2.Embarked.fillna('Missing', inplace=True)
```

- After applying missing value imputation, plot the bar plot for the embarked column. You can see that we have a very small, almost negligible plot for the missing column.

```
td2.Embarked.value_counts().sort_ values(ascending=False).plot.bar()
plt.xlabel('Embark Town’)
plt.ylabel('Number of Passengers')
```



Data of Wrong Format

- Cells with data of wrong format can make it difficult, or even impossible, to analyze data.
- To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.
- Example : **Date format**

Pandas has a `to_datetime()` method for this:

Date Conversion:

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

Removing Rows

- The result from the converting in the example above gave us a **NaN** value, which can be handled as a NULL value, and we can remove the row by using the `dropna()` method.

- Example
  Remove rows with a NULL value in the "Date" column:
  ```
  df.dropna(subset=['Date'], inplace = True)
  ```

Wrong Data

- "Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".
- Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

Replacing Values

- One way to fix wrong values is to replace them with something else.
- Examples:
  - it is most likely a typo, and the value should be "45" instead of "450
  - `df.loc[7, 'Duration'] = 45`
- For small data sets you might be able to replace the wrong data one by one, but not for big data sets.
- To replace wrong data for larger data sets you can create some rules,
  - For instance, set some boundaries for legal values, and replace any values that are outside of the boundaries.
  - Example
    - Loop through all values in the "Duration" column.
    - If the value is higher than 120, set it to 120:

```
for x in df.index:
    if df.loc[x, "Duration"] > 120:
        df.loc[x, "Duration"] = 120
```

Removing Rows

- Another way of handling wrong data is to remove the rows that contains wrong data.
- This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.
- Example

  Delete rows where "Duration" is higher than 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

Removing Duplicates

- To discover duplicates, we can use the **`duplicated()`** method.
- The **`duplicated()`** method returns a Boolean values for each row:

- Example
  - Returns **`True`** for every row that is a duplicate, othwerwise **`False`**:
  - `print(df.duplicated())`

- To remove duplicates, use the **drop_duplicates()** method.

- Example

  Remove all duplicates:

  df.**drop_duplicates**(inplace = True)