#!/usr/bin/env python

# coding: utf-8

# # Handling Missing Data

# In this section, we will study ways to identify and treat missing data. We will:

#

# Identify missing data in dataframes

# Treat (delete or impute) missing values

# There are various reasons for missing data, such as, human-errors during data-entry, non availability at the end of the user (e.g. DOB of certain people), etc. Most often, the reasons are simply unknown.

#

# In python, missing data is represented using either of the two objects NaN (Not a Number) or NULL. We'll not get into the differences between them and how Python stores them internally etc. We'll focus on studying ways to identify and treat missing values in Pandas dataframes.

#

# There are four main methods to identify and treat missing data:

#

# isnull(): Indicates presence of missing values, returns a boolean

# notnull(): Opposite of isnull(), returns a boolean

# dropna(): Drops the missing values from a data frame and returns the rest

# fillna(): Fills (or imputes) the missing values by a specified value

# For this exercise, we will use the Melbourne house pricing dataset.

# In[49]:

import numpy as np

import pandas as pd

#library to deal with warning

import warnings

warnings.filterwarnings('ignore')

# In[68]:

df = pd.read\_csv('C:/Users/HP/Desktop/Data Science/Python Sessions/Python Session 20/melbourne1.csv.csv')

df.head()

# In[69]:

df.tail()

# In[70]:

df.shape

# In[71]:

df.keys()

# In[72]:

df.describe() # its give numerical columns and their Aggression funtion

# In[73]:

df.describe(include='all').T

# - The first few rows contain missing values, represented as NaN.

# - Let's quickly look at the structure of the data frame, types of columns, etc.

# In[74]:

# approx 23k rows, 21 columns

print(df.shape)

print(df.info())

# ## Identifying Missing Values

# The methods isnull() and notnull() are the most common ways of identifying missing values.

#

# While handling missing data, you first need to identify the rows and columns containing missing values, count the number of missing values, and then decide how you want to treat them.

#

# It is important that you treat missing values in each column separately, rather than implementing a single solution (e.g. replacing NaNs by the mean of a column) for all columns.

#

# isnull() returns a boolean (True/False) which can then be used to find the rows or columns containing missing values.

# In[75]:

# isnull()

df.isnull()

# ## Identifying Missing Values in Columns

# Let's first compute the total number of missing values in the data frame. You can calculate the number of missing values in each column by

# df.isnull().sum()

# In[76]:

# summing up the missing values (column-wise)

df.isnull().sum()

# df.notnull()

#

# Note that some columns have extremely large number of missing values, such as Price, Bedroom2, Bathroom, BuildingArea, YearBuilt etc. In such cases, one should be careful in handling missing values, since if you replace them by arbitrary numbers such as mean, median etc., the entire further analysis may throw unrealistic or unexpected results.

#

# The functions any() and all() are quite useful to identify rows and columns having missing values:

#

# - any() returns True when at least one value satisfies a condition (equivalent to logical or)

# - all() returns True when all the values satisfy a condition (equivalent to logical and)

# In[77]:

# columns having at least one missing value

df.isnull().any()

# above is equivalent to axis=0 (by default, any() operates on columns)

df.isnull().any(axis=0)

# We have identified columns having missing values and have computed the number of missing values in each. Let's do the same for rows.

# ## Identifying Missing Values in Rows

# The methods any() and all() can be used to identify rows having at least one and all missing values respectively. To specify that the operation should be done on rows, you need to use axis=1 as an argument.

# In[79]:

# rows having at least one missing value

df.isnull().any(axis=1)

# In[80]:

# rows having all missing values

df.isnull().any(axis=1).sum()

# In[81]:

# sum it up to check how many rows have all missing values

df.isnull().all(axis=1).sum()

# Thus, there are no rows having all missing values (we'd remove them if there were any).

#

# Often, you may also want to remove the rows having more than a certain threshold number of missing values. To do that, you need to count the number of missing values in each row using sum().

# In[82]:

# sum of misisng values in each row

df.isnull().sum(axis=1)

# We have now identified:

#

# - The number of missing values in columns

# - The number of missing values in rows

# Let's now move ahead and treat the missing values.

#

# ## Treating Missing Values

# There are broadly two ways to treat missing values:

#

# 1.Delete: Delete the missing values

# 2.Impute:

# - Imputing by a simple statistic: Replace the missing values by another value, commonly the mean, median, mode etc.

# - Predictive techniques: Use statistical models such as k-NN, SVM etc. to predict and impute missing values

# In general, imputation makes assumptions about the missing values and replaces missing values by arbitrary numbers such as mean, median etc. It should be used only when you are reasonably confident about the assumptions.

#

# Otherwise, deletion is often safer and recommended. You may lose some data, but will not make any unreasonable assumptions.

#

# Caution: Always have a backup of the original data if you're deleting missing values.

#

# \*\*Additional Stuff for Nerds\*\*

# How you treat missing values should ideally depend upon an understnading of why missing values occur. The reasons are classified into categories such as missing completely at random, missing at random, misisngness that depends on the missing value itself etc.

#

# We'll not discuss why missing values occur, though you can read this article if interested: http://www.stat.columbia.edu/~gelman/arm/missing.pdf

#

# ## Treating Missing Values in Columns

# Let's now treat missing values in columns. Let's look at the number of NaNs in each column again, this time as the percentage of missing values in each column. Notice that we calculate the number of rows as len(df.index).

# In[83]:

print(df.isnull().sum())

len(df.index)

# In[85]:

# summing up the missing values (column-wise)

round(100\*(df.isnull().sum()/len(df.index)), 2)

# In[86]:

# summing up the missing values (column-wise)

round(100\*(df.isnull().sum()/len(df.index)), 0)

# Notice that there are columns having almost 22%, 19%, 26%, 57% etc. missing values. When dealing with columns, you have two simple choices - either delete or retain the column. If you retain the column, you'll have to treat (i.e. delete or impute) the rows having missing values.

#

# If you delete the missing rows, you lose data. If you impute, you introduce bias.

#

# Apart from the number of missing values, the decision to delete or retain a variable depends on various other factors, such as:

#

# - the analysis task at hand

# - the usefulness of the variable (based on your understanding of the problem)

# - the total size of available data (if you have enough, you can afford to throw away some of it)

# - etc.

# For e.g. let's say that we want to build a (linear regression) model to predict the house prices in Melbourne. Now, even though the variable Price has about 22% missing values, you cannot drop the variable, since that is what you want to predict.

#

# Similarly, you would expect some other variables such as Bedroom2, Bathroom, Landsize etc. to be important predictors of Price, and thus cannot remove those columns.

#

# There are others such as BuildingArea, which although seem important, have more than 50% missing values. It is impossible to either delete or impute the rows corresponding to such large number of missing values without losing a lot of data or introducing heavy bias.

#

# Thus, for this exercise, let's remove the columns having more than 30% missing values, i.e. BuildingArea, YearBuilt, CouncilArea.

# In[87]:

# removing the three columns

df = df.drop('BuildingArea', axis=1)

df = df.drop('YearBuilt', axis=1)

df = df.drop('CouncilArea', axis=1)

round(100\*(df.isnull().sum()/len(df.index)), 2)

# We now have columns having maximum 26% missing values (Landsize). Next, we need to treat the rows.

#

# ## Treating Missing Values in Rows

# Now, we need to either delete or impute the missing values. First, let's see if there are any rows having a significant number of missing values. If so, we can drop those rows, and then take a decision to delete or impute the rest.

#

# After dropping three columns, we now have 18 columns to work with. Just to inspect rows with missing values, let's have a look at the rows having more than 5 missing values.

# In[88]:

df[df.isnull().sum(axis=1) > 5]

# Notice an interesting pattern - many rows have multiple columns missing. Since each row represents a house, it indicates that there are houses (observations) whose majority data has either not been collected or is unavailable. Such observations are anyway unlikely to contribute to prediction of prices.

#

# Thus we can remove the rows with (say) more than 5 missing values.

# In[89]:

# count the number of rows having > 5 missing values

# use len(df.index)

len(df[df.isnull().sum(axis=1) > 5].index)

# In[90]:

# 4278 rows have more than 5 missing values

# calculate the percentage

100\*(len(df[df.isnull().sum(axis=1) > 5].index) / len(df.index))

# Thus, about 18% rows have more than 5 missing values. Let's remove these rows and count the number of missing values remaining.

# In[91]:

# retaining the rows having <= 5 NaNs

df = df[df.isnull().sum(axis=1) <= 5]

# look at the summary again

round(100\*(df.isnull().sum()/len(df.index)), 2)

# Notice that now, we have removed most of the rows where multiple columns (Bedroom2, Bathroom, Landsize) were missing.

#

# Now, we still have about 21% missing values in the column Price and 9% in Landsize. Since Price still contains a lot of missing data (and imputing 21% values of a variable you want to predict will introduce heavy bias), its a bad idea to impute those values.

#

# Thus, let's remove the missing rows in Price as well. Notice that you can use np.isnan(df['column']) to filter out the corresonding rows, and use a ~ to discard the values satisfying the condition.

# In[92]:

# removing NaN Price rows

df = df[~np.isnan(df['Price'])]

round(100\*(df.isnull().sum()/len(df.index)), 2)

# In[93]:

df = df[~np.isnan(df['Price'])]

df

# In[94]:

len(df)

# Now, you have Landsize as the only variable having a significant number of missing values. Let's give this variable a chance and consider imputing the NaNs.

#

# The decision (whether and how to impute) will depend upon the distribution of the variable. For e.g., if the variable is such that all the observations lie in a short range (say between 800 sq. ft to 820 sq.ft), you can take a call to impute the missing values by something like the mean or median Landsize.

#

# Let's look at the distribution.

# In[95]:

df.dtypes

# In[96]:

df['Landsize'].describe()

# Notice that the minimum is 0, max is 433014, the mean is 558 and median (50%) is 440. There's a significant variation in the 25th and the 75th percentile as well (176 to 651).

#

# Thus, imputing this with mean/median seems quite biased, and so we should remove the NaNs.

# In[97]:

df[~np.isnan(df['Landsize'])]

# In[98]:

len(df[~np.isnan(df['Landsize'])])

# In[99]:

# removing NaNs in Landsize

df = df[~np.isnan(df['Landsize'])]

round(100\*(df.isnull().sum()/len(df.index)), 2)

# We have reduced the NaNs significantly now. Only the variables Bathroom, Car, Lattitude and Longitude have a small number of missing values (most likely, the same rows will have Lattitude and Longitude missing).

#

# Let's first look at Lattitude and Longitude.

# In[100]:

# rows having Lattitude and Longitude missing

df[np.isnan(df['Lattitude'])]

# As expected, the same rows have Lattitude and Longitude missing. Let's look at the summary stats of these.

# In[101]:

df.loc[:, ['Lattitude', 'Longtitude']].describe()

# Notice that the distribution of both Lattitude and Longitude is quite narrow.

#

# A good way to estimate the 'spread of data' is to look at the difference between the mean and the median (lower the better), and the variation from 25th to 75th percentile (quite small in this case).

#

# Thus, let's impute the missing values by the mean value of Lattitude and Longitude respectively.

# In[102]:

# imputing Lattitude and Longitude by mean values

df.loc[np.isnan(df['Lattitude']), ['Lattitude']] = df['Lattitude'].mean()

df.loc[np.isnan(df['Longtitude']), ['Longtitude']] = df['Longtitude'].mean()

round(100\*(df.isnull().sum()/len(df.index)), 2)

# In[104]:

df['Lattitude'].fillna(df['Lattitude'].mean(),inplace=True)

# In[105]:

df['Lattitude'].mean()

# In[106]:

df['Lattitude'].isnull().any()

# We now have Bathroom and Car with 0.01% and 0.46% NaNs respectively.

#

# Since these are very small number of rows, it does not really matter whether you delete or impute. However, let's have a look at the distributions.

# In[107]:

df.loc[:, ['Bathroom', 'Car']].describe()

# These two are integer type variables, and thus have values 0, 1, 2 etc. You cannot impute the NaNs by the mean or the median (1.53 bathrooms does not make sense!).

#

# Thus, you need to impute them by the mode - the most common occurring value.

# In[108]:

# converting to type 'category'

df['Car'] = df['Car'].astype('category')

df['Car'].head()

# In[109]:

df.dtypes

# In[110]:

df['Car'].unique()

# In[111]:

df['Car'].nunique()

# In[112]:

df['Car'].value\_counts()

# In[113]:

df['Car'].fillna(2.0,inplace=True)

# In[114]:

# displaying frequencies of each category

df['Car'].isnull().any()

# In[115]:

df['Car'].shape

# The most common value of Car is 2 (dtype is int), so let's impute the NaNs by that.

# In[116]:

# imputing NaNs by 2.0

df.loc[pd.isnull(df['Car']), ['Car']] = 2

round(100\*(df.isnull().sum()/len(df.index)), 2)

# Similarly for Bathroom:

# In[117]:

# converting to type 'category'

df['Bathroom'] = df['Bathroom'].astype('category')

# displaying frequencies of each category

df['Bathroom'].value\_counts()

# In[118]:

# imputing NaNs by 1

df.loc[pd.isnull(df['Bathroom']), ['Bathroom']] = 1

round(100\*(df.isnull().sum()/len(df.index)), 2)

# We now have a dataframe with no missing values. Let's finally look at how many rows (apart from three columns) we have lost in the process (originally we had 23547):

# In[119]:

df.shape

# In[120]:

# fraction of rows lost

len(df.index)/23547

# Thus, we have lost about 42% observations in cleaning the missing values.

# In[121]:

df.dtypes

# In[122]:

df.groupby(by=['Type']).count()

# In[123]:

df.groupby(by=['Type']).median()

# #Pivot\_Table

# Pivot Tables

#

# - Pandas provides a function â€˜pivot\_tableâ€™ to create MS-Excel spreadsheet style pivot tables. It can take following arguments:

# - data: DataFrame object,

# - values: column to aggregate,

# - index: row labels,

# - columns: column labels,

# - Aggfunc: aggregation function to be used on values, default is NumPy.mean

# In[124]:

pd.pivot\_table(df, index = ['Type','Bathroom'])

# In[125]:

import matplotlib.pyplot as plt

get\_ipython().run\_line\_magic('matplotlib', 'inline')

# In[127]:

df['Price'].hist(bins=30)

# In[128]:

df.boxplot('Price')

# In[129]:

df.boxplot('Price', by= 'Type', figsize= (10,10))

# In[130]:

a = pd.pivot\_table(df, index = ['Car'], aggfunc= 'mean')

# In[131]:

a.plot(kind='bar')

# In[132]:

a.plot(kind='bar', stacked = True)

# In[ ]:

# In[ ]:

# In[ ]: