Sardar Patel Institute of Technology, Mumbai
Department of Electronics and Telecommunication Engineering
B.E. Sem-VII (2022-2023)
OEIT6 - Data Analytics

Experiment: Time Series Forecasting

Name: Pushkar Sutar **Roll No. 2019110060**

Objective : Based on competition available at Kaggle, https://www.kaggle.com/c/house-prices-advanced-regressiontechniques/overview/description, revolving around predicting the house price through regression techniques. Your objective is to build a regression model and measure the performance of the model in terms of accuracy, recall, sensitivity, specificity, ROC curves, precision recall curves and loss function for regression.

Introduction:

A time series is a chronological sequence of observations on a particular variable. Usually the observations are taken at regular intervals (days, months, years), but the sampling could be irregular. A time series analysis consists of steps:
(1) building a model that represents a time series
(2) validating the model proposed
(3) using the model to predict (forecast) future values and/or impute missing values.
If a time series has a regular pattern, then a value of the series should be a function of previous values.
The goal of building a time series model is the same as the goal for other types of predictive models, which is to create a model such that the error between the predicted value of the target variable and the actual value is as small as possible.
The primary difference between time series models and other types of models is that lag values of the target variable are used as predictor variables, whereas traditional models use other variables as predictors, and the concept of a lag value doesn't apply because the observations don't represent a chronological sequence.
From an experimental point of view There are two general approaches to modelling time series data that contain a trend and/or seasonal variation.
1. First model the trend and seasonality in the data, and then use a stationary time series model to represent the short-term correlation.
2. Model the trend, seasonality and short-term correlation in the data simultaneously using a nonstationary time series model.

Dataset Description:

Here's a brief version of what you'll find in the data description file.

- SalePrice - the property's sale price in dollars. This is the target variable that you're trying to predict.
- MSSubClass: The building class
- MSZoning: The general zoning classification
- LotFrontage: Linear feet of street connected to property
- LotArea: Lot size in square feet
- Street: Type of road access
- Alley: Type of alley access
- LotShape: General shape of property
- LandContour: Flatness of the property
- Utilities: Type of utilities available
- LotConfig: Lot configuration
- LandSlope: Slope of property
- Neighborhood: Physical locations within Ames city limits
- Condition1: Proximity to main road or railroad
- Condition2: Proximity to main road or railroad (if a second is present)
- BldgType: Type of dwelling
- HouseStyle: Style of dwelling
- OverallQual: Overall material and finish quality
- OverallCond: Overall condition rating
- YearBuilt: Original construction date
- YearRemodAdd: Remodel date
- RoofStyle: Type of roof
- RoofMatl: Roof material
- Exterior1st: Exterior covering on house
- Exterior2nd: Exterior covering on house (if more than one material)
- MasVnrType: Masonry veneer type
- MasVnrArea: Masonry veneer area in square feet
- ExterQual: Exterior material quality
- ExterCond: Present condition of the material on the exterior
- Foundation: Type of foundation
- BsmtQual: Height of the basement
- BsmtCond: General condition of the basement
- BsmtExposure: Walkout or garden level basement walls
- BsmtFinType1: Quality of basement finished area
- BsmtFinSF1: Type 1 finished square feet

- BsmtFinType2: Quality of second finished area (if present)
- BsmtFinSF2: Type 2 finished square feet
- BsmtUnfSF: Unfinished square feet of basement area
- TotalBsmtSF: Total square feet of basement area
- Heating: Type of heating
- HeatingQC: Heating quality and condition
- CentralAir: Central air conditioning
- Electrical: Electrical system
- 1stFlrSF: First Floor square feet
- 2ndFlrSF: Second floor square feet
- LowQualFinSF: Low quality finished square feet (all floors)
- GrLivArea: Above grade (ground) living area square feet
- BsmtFullBath: Basement full bathrooms
- BsmtHalfBath: Basement half bathrooms
- FullBath: Full bathrooms above grade
- HalfBath: Half baths above grade
- Bedroom: Number of bedrooms above basement level
- Kitchen: Number of kitchens
- KitchenQual: Kitchen quality
- TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)
- Functional: Home functionality rating
- Fireplaces: Number of fireplaces
- FireplaceQu: Fireplace quality
- GarageType: Garage location
- GarageYrBlt: Year garage was built
- GarageFinish: Interior finish of the garage
- GarageCars: Size of garage in car capacity
- GarageArea: Size of garage in square feet
- GarageQual: Garage quality
- GarageCond: Garage condition
- PavedDrive: Paved driveway
- WoodDeckSF: Wood deck area in square feet
- OpenPorchSF: Open porch area in square feet
- EnclosedPorch: Enclosed porch area in square feet
- 3SsnPorch: Three season porch area in square feet
- ScreenPorch: Screen porch area in square feet
- PoolArea: Pool area in square feet
- PoolQC: Pool quality
- Fence: Fence quality
- MiscFeature: Miscellaneous feature not covered in other categories

- MiscVal: $Value of miscellaneous feature
- MoSold: Month Sold
- YrSold: Year Sold
- SaleType: Type of sale
- SaleCondition: Condition of sale

Code and Output:

1. Create time series data -

We first import all necessary modules -

```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

Load train and test dataset available

```
[2] df_train=pd.read_csv('/content/train.csv')
    df_train.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN

5 rows × 81 columns

```
[3] df_test=pd.read_csv('/content/test.csv')
    df_test.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	ScreenPorch	PoolArea	PoolQC	Fence
0	1461	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	AllPub	...	120	0	NaN	MnPrv
1	1462	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN
2	1463	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	MnPrv
3	1464	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN
4	1465	120	RL	43.0	5005	Pave	NaN	IR1	HLS	AllPub	...	144	0	NaN	NaN

5 rows × 80 columns

We see that the train dataset contains 81 rows and 1460 instances.

```
[4] df_train.shape
```

```
(1460, 81)
```

We prepare and clean our data before creating a time series representation.

```
[5]  df_train.isnull().sum()
```

```
Id                 0
MSSubClass         0
MSZoning           0
LotFrontage      259
LotArea            0
                ...
MoSold             0
YrSold             0
SaleType           0
SaleCondition      0
SalePrice          0
Length: 81, dtype: int64
```

```
[6]  percent_missing = df_train.isnull().sum() * 100 / len(df_train)
     missing_value_df = pd.DataFrame({'column_name': df_train.columns,
                                      'percent_missing': percent_missing})
```

```
[7]  missing_value_df.sort_values('percent_missing', inplace=True)
     missing_value_df.tail(20)
```

Utilities	Utilities	0.000000
Electrical	Electrical	0.068493
MasVnrType	MasVnrType	0.547945
MasVnrArea	MasVnrArea	0.547945
BsmtQual	BsmtQual	2.534247
BsmtCond	BsmtCond	2.534247
BsmtFinType1	BsmtFinType1	2.534247
BsmtFinType2	BsmtFinType2	2.602740
BsmtExposure	BsmtExposure	2.602740
GarageQual	GarageQual	5.547945
GarageFinish	GarageFinish	5.547945
GarageYrBlt	GarageYrBlt	5.547945
GarageType	GarageType	5.547945
GarageCond	GarageCond	5.547945
LotFrontage	LotFrontage	17.739726
FireplaceQu	FireplaceQu	47.260274
Fence	Fence	80.753425
Alley	Alley	93.767123
MiscFeature	MiscFeature	96.301370
PoolQC	PoolQC	99.520548

We drop the columns with more than half rows that have null values.

```
[8]  df_train.drop(['FireplaceQu', 'Fence','Alley','MiscFeature','PoolQC'], inplace=True, axis=1)
```

```
[9]  df_test.drop(['FireplaceQu', 'Fence','Alley','MiscFeature','PoolQC'], inplace=True, axis=1)
```

We drop all the remaining null rows.

```
[10] df_train.dropna()
     df_test.dropna()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	...	OpenPorchSF
0	1461	20	RH	80.0	11622	Pave	Reg	Lvl	AllPub	Inside	...	0
1	1462	20	RL	81.0	14267	Pave	IR1	Lvl	AllPub	Corner	...	36
2	1463	60	RL	74.0	13830	Pave	IR1	Lvl	AllPub	Inside	...	34
3	1464	60	RL	78.0	9978	Pave	IR1	Lvl	AllPub	Inside	...	36
4	1465	120	RL	43.0	5005	Pave	IR1	HLS	AllPub	Inside	...	82
...	...	...	...	...	...	...	...	...	...	...	...	...
1451	2912	20	RL	80.0	13384	Pave	Reg	Lvl	AllPub	Inside	...	0
1452	2913	160	RM	21.0	1533	Pave	Reg	Lvl	AllPub	Inside	...	0
1455	2916	160	RM	21.0	1894	Pave	Reg	Lvl	AllPub	Inside	...	24
1456	2917	20	RL	160.0	20000	Pave	Reg	Lvl	AllPub	Inside	...	0
1458	2919	60	RL	74.0	9627	Pave	Reg	Lvl	AllPub	Inside	...	48

We check how many columns are remaining after cleaning.

```
[11] df_train.columns
```

```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope',
       'Neighborhood', 'Condition1', 'Condition2', 'BldgType', 'HouseStyle',
       'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'RoofStyle',
       'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType', 'MasVnrArea',
       'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual', 'BsmtCond',
       'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1', 'BsmtFinType2',
       'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating', 'HeatingQC',
       'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
       'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
       'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRmsAbvGrd',
       'Functional', 'Fireplaces', 'GarageType', 'GarageYrBlt', 'GarageFinish',
       'GarageCars', 'GarageArea', 'GarageQual', 'GarageCond', 'PavedDrive',
       'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch', '3SsnPorch',
       'ScreenPorch', 'PoolArea', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

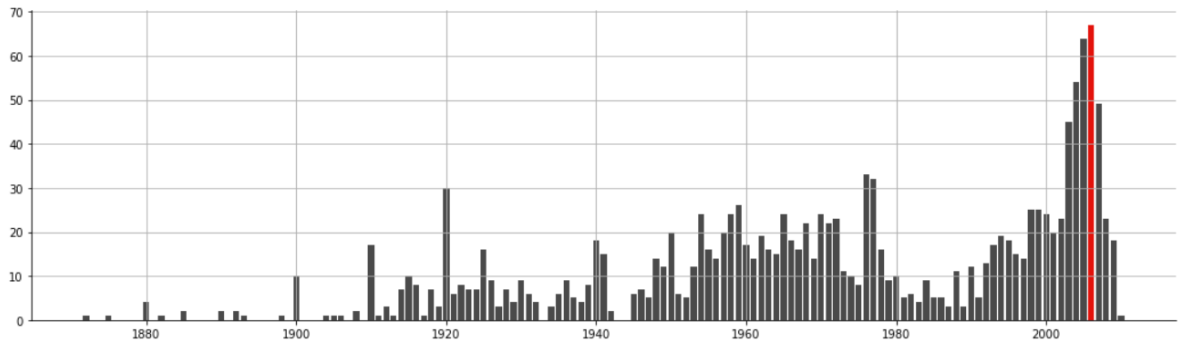
2. Accommodate trend, as well as seasonal and event-related variation, in time series models.

• Time series plots: Basic visualisation of ts objects and differentiating trends, seasonality, and cycle variation.

• Seasonal plots: Plotting seasonality trends in time series data.

We create a bar plot for visualisation of the number of sales per year.

```
built = df_train['YearBuilt'].value_counts().sort_index()
fig, ax = plt.subplots(1, 1, figsize=(18, 5))
color = ['#4a4a4a' if val != max(built) else '#e3120b' for val in built]
ax.bar(built.index, built, color=color)

for s in ['top', 'right']:
    ax.spines[s].set_visible(False)

ax.grid()

plt.show()
```



We can see that the year 2007 has the maximum number of sales.

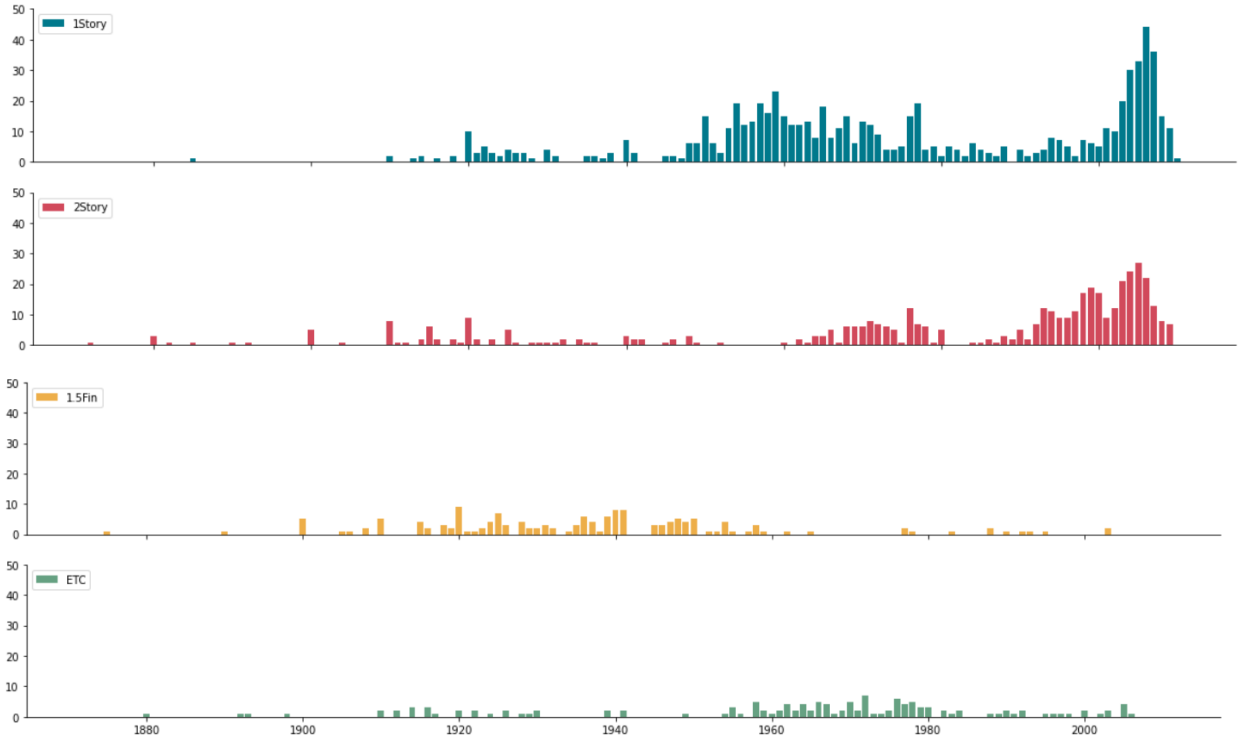
We see that there are many different house styles. We can combine the house styles with lesser value counts in the same variable.

```
[15] df_train['HouseStyle'].value_counts()
```

```
1Story    726
2Story    445
1.5Fin    154
SLvl       65
SFoyer     37
1.5Unf     14
2.5Unf     11
2.5Fin      8
Name: HouseStyle, dtype: int64
```

```
[19] df_train['HouseStyle'] = df_train['HouseStyle'].apply(lambda x : 'ETC' if x in ['SLvl', 'SFoyer', '1.5Unf', '2.5Unf', '2.5Fin'] else x)
     df_test['HouseStyle'] = df_test['HouseStyle'].apply(lambda x : 'ETC' if x in ['SLvl', 'SFoyer', '1.5Unf', '2.5Unf', '2.5Fin'] else x)
```

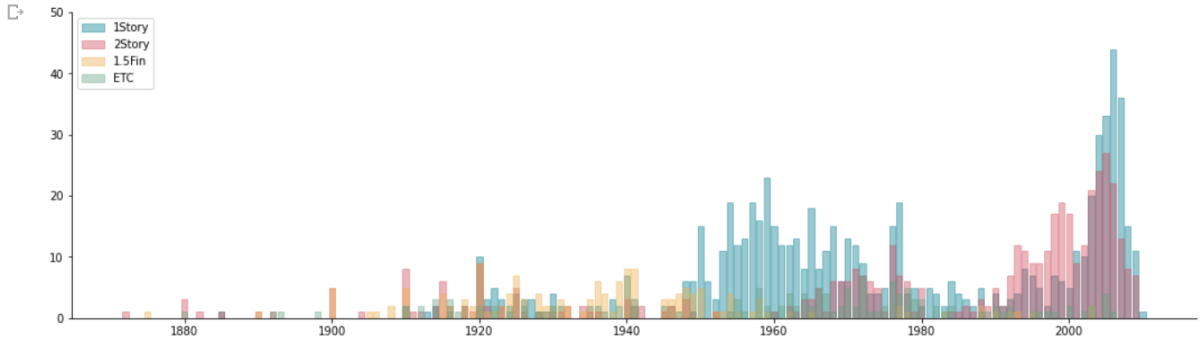
```
[20] fig, ax = plt.subplots(4, 1, figsize=(20, 12), sharex=True)
     color = ["#00798c", "#d1495b", '#edae49', '#66a182']

     for i, hs in enumerate(df_train['HouseStyle'].value_counts().index):
         hs_built = df_train[df_train['HouseStyle']==hs]['YearBuilt'].value_counts()
         ax[i].bar(hs_built.index, hs_built, color=color[i], label=hs)
         ax[i].set_ylim(0, 50)
         ax[i].legend(loc='upper left')
         for s in ['top', 'right']:
             ax[i].spines[s].set_visible(False)



     plt.show()
```



We can observe that more 1 Story houses are sold after the year 2000. People preferred 2 story and 1 story houses the most after 2000.

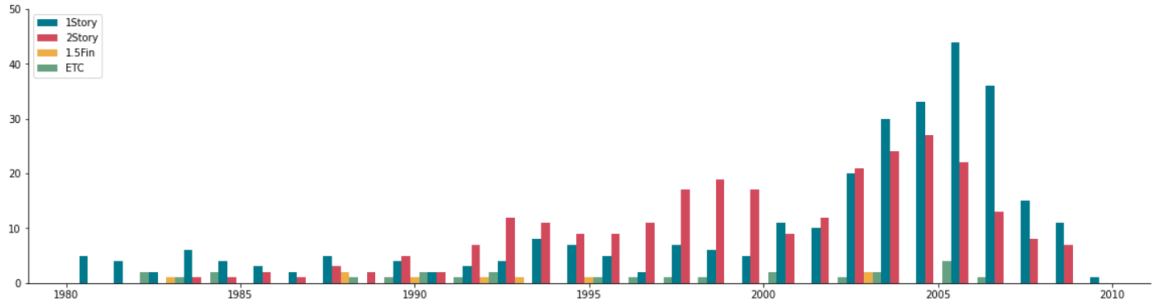
We can also visualise these plots together on the same graph.

```
[22] fig, ax = plt.subplots(1, 1, figsize=(18, 5))
     color = ["#00798c", "#d1495b", '#edae49', '#66a182']

     for i, hs in enumerate(df_train['HouseStyle'].value_counts().index):
         hs_built = df_train[df_train['HouseStyle']==hs]['YearBuilt'].value_counts()
         ax.bar(hs_built.index, hs_built, color=color[i], label=hs, alpha=0.4, edgecolor=color[i])

     for s in ['top', 'right']:
         ax.spines[s].set_visible(False)

     ax.set_ylim(0, 50)
     ax.legend(loc='upper left')
     plt.show()
```



We can also visualise this using separate bar graphs.

```
[24] fig, ax = plt.subplots(1, 1, figsize=(20, 5))
     color = ["#00798c", "#d1495b", '#edae49', '#66a182']

     width = 0.25
     for i, hs in enumerate(df_train['HouseStyle'].value_counts().index):
         hs_built = df_train[(df_train['HouseStyle']==hs)&(df_train['YearBuilt']>1980)]['YearBuilt'].value_counts()
         ax.bar(hs_built.index+(width*(i-2)), hs_built, width, color=color[i], label=hs)

     for s in ['top', 'right']:
         ax.spines[s].set_visible(False)

     ax.set_ylim(0, 50)
     ax.legend(loc='upper left')
     plt.show()
```



We can also plot a line graph indicating the number of houses built by year.

```
[25] built = df_train['YearBuilt'].value_counts().sort_index()
     fig, ax = plt.subplots(1, 1, figsize=(18, 5))

     ax.plot(built.index, built, color='#4a4a4a')

     for s in ['top', 'right']:
         ax.spines[s].set_visible(False)

     ax.grid()

     plt.show()
```

We can clearly see that as population increases the demand for houses increases and more houses are built in later years.

• Creating time series objects: Convert your data to a time series object for time series analysis

```
[27] df_train["Date"] = df_train['YrSold'].astype(str) +"-"+ df_train["MoSold"].astype(str)
     df_test["Date"] = df_test['YrSold'].astype(str) +"-"+ df_test["MoSold"].astype(str)
```

```
[34] df_train["Date"] = pd.to_datetime(df_train["Date"], format='%Y-%m-%d')
     df_test["Date"] = pd.to_datetime(df_test["Date"], format='%Y-%m-%d')
```

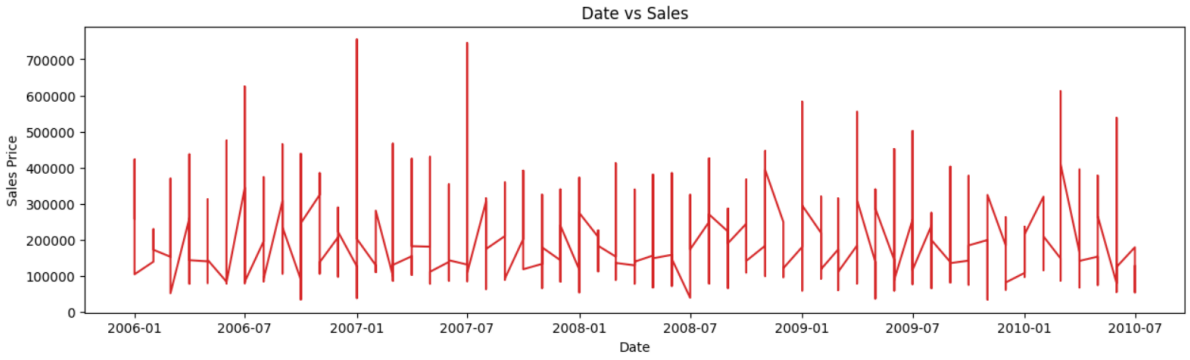
We have created a time series object “Date” for further analysis with sales price of houses.

```
[43] result_df = df_train.sort_values(by = ['Date','MoSold'])
     result_df = pd.DataFrame(result_df, columns=['Date', 'SalePrice']).reset_index()
```

We have only kept two attributes for analysis in this dataframe.

We now plot sales vs date for the given data.

```
[44] def plot_df(df, x, y, title="", xlabel='Date', ylabel='Sales Price', dpi=100):
         plt.figure(figsize=(15,4), dpi=dpi)
         plt.plot(x, y, color='tab:red')
         plt.gca().set(title=title, xlabel=xlabel, ylabel=ylabel)
         plt.show()


     plot_df(result_df, x=result_df['Date'], y=result_df['SalePrice'], title='Date vs Sales')
```



```
[49] result_df.head()
     result_df.drop("index",inplace = True,axis=1)
```

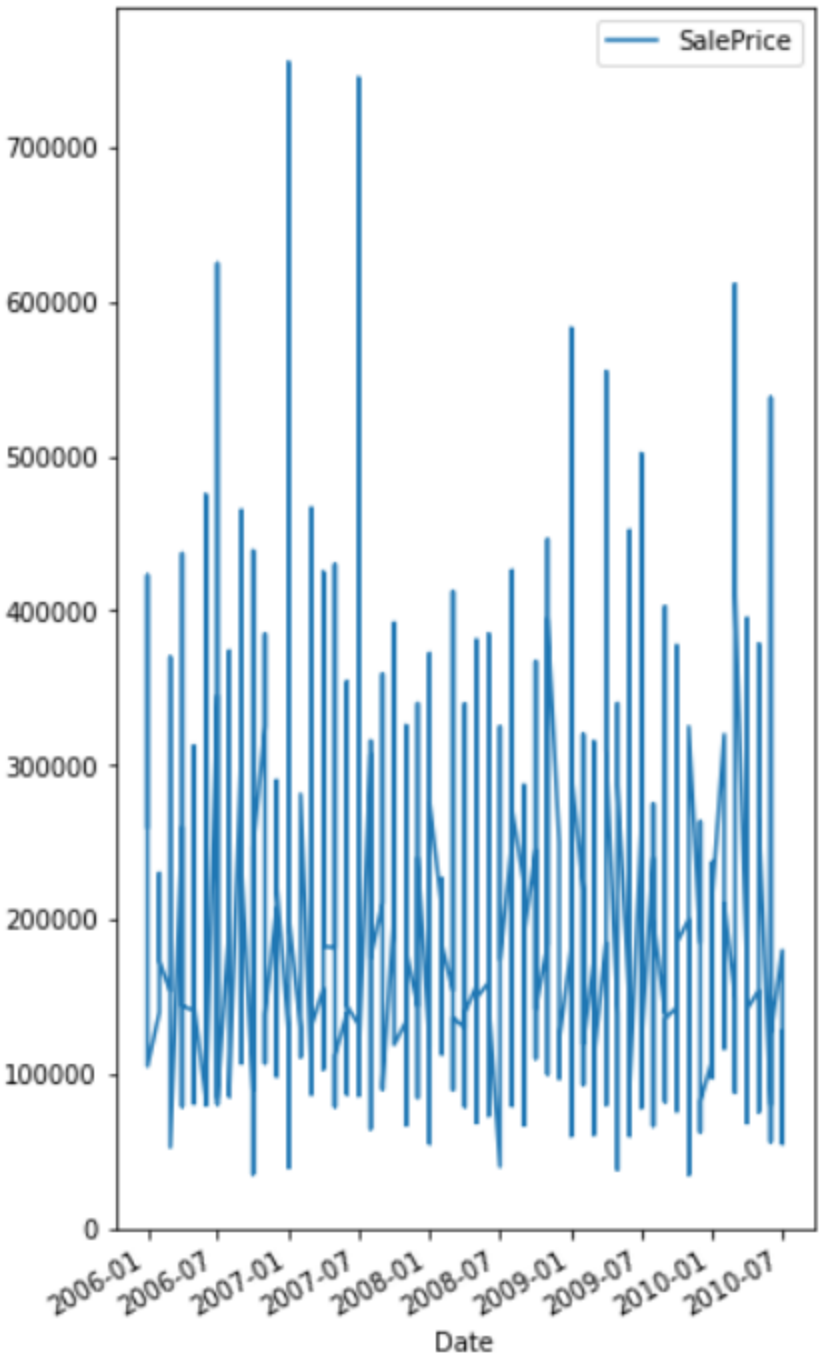
```
[51] result_df.head()
```

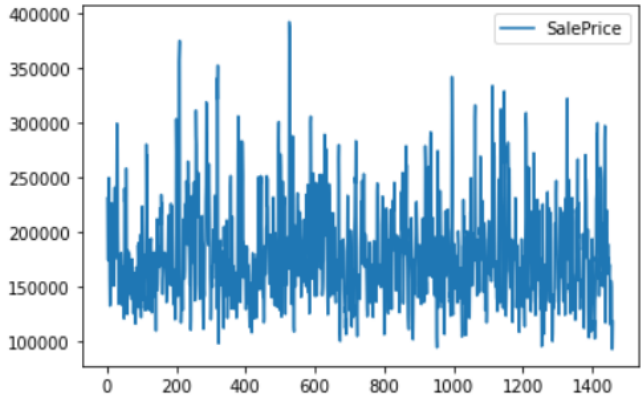
	Date	SalePrice
0	2006-01-01	260000
1	2006-01-01	228000
2	2006-01-01	205000
3	2006-01-01	172400
4	2006-01-01	145000

3. Stationary and Autocorrelation of time series: Computing constant mean and variance and visualising autocorrelation.

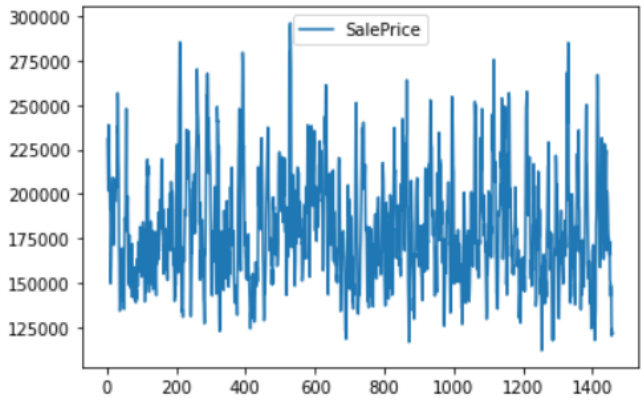
When plotting the time series data, these fluctuations may prevent us to clearly gain insights about the peaks and troughs in the plot. So to clearly get value from the data, we use the rolling average concept to make the time series plot.

```
[52] result_df.set_index('Date')
     result_df=result_df.fillna(0)

     fig, axs = plt.subplots(figsize=(5,10))
     result_df.plot(x='Date', ax=axs)
     plt.show()


     result_df.rolling(window=3,min_periods=3).mean().plot()
     plt.show()
     print("The larger the window coefficient the smoother the line will appear")
     print('The min_periods is the minimum number of observations in the window required to have a value')

     result_df.rolling(window=6,min_periods=3).mean().plot()
     plt.show()
```


SalePrice
700000
600000
500000
400000
300000
200000
100000
0
2006-01
2006-07
2007-01
2007-07
2008-01
2008-07
2009-01
2009-07
2010-01
2010-07
Date



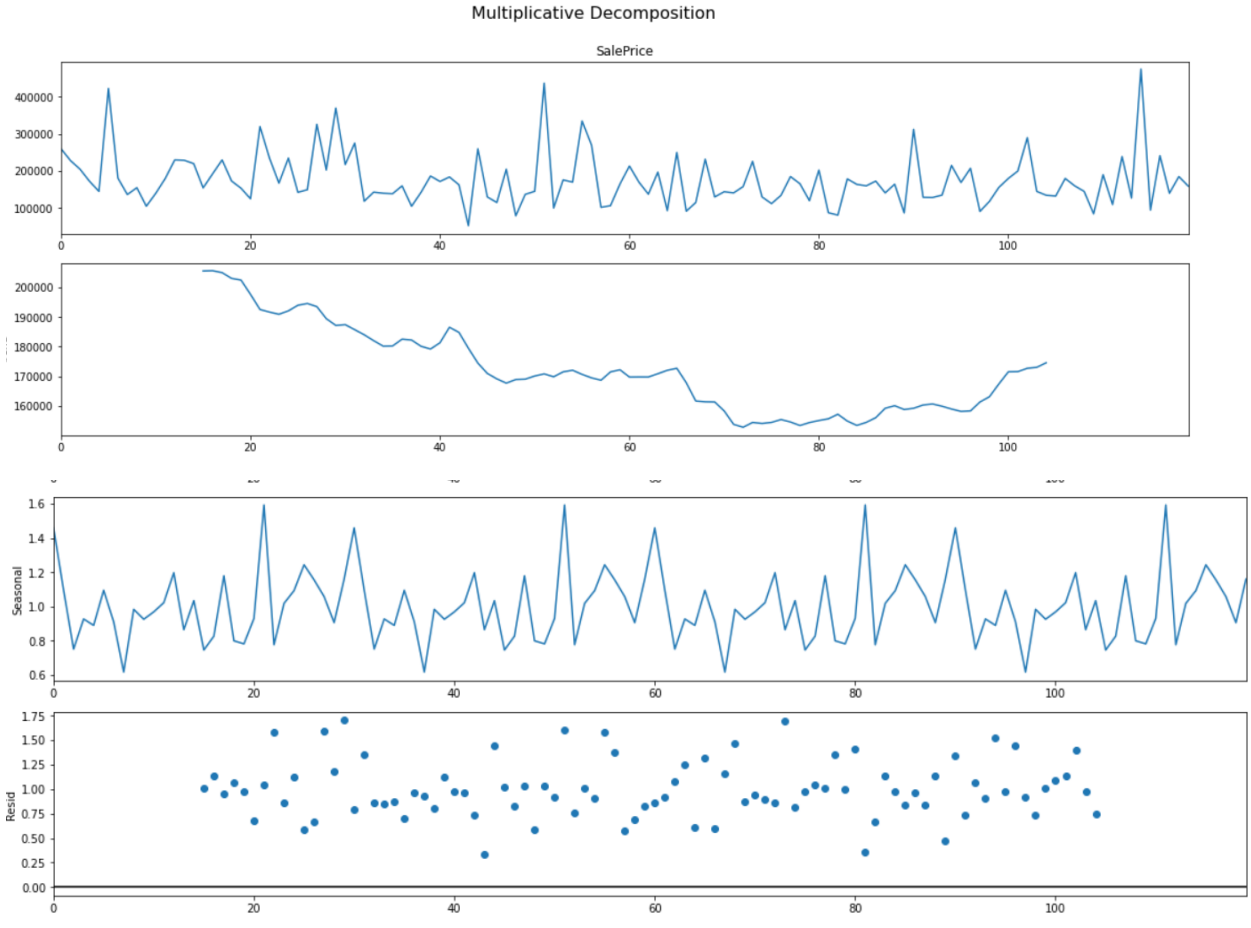
The larger the window coefficient the smoother the line will appear
The min_periods is the minimum number of observations in the window required to have a value

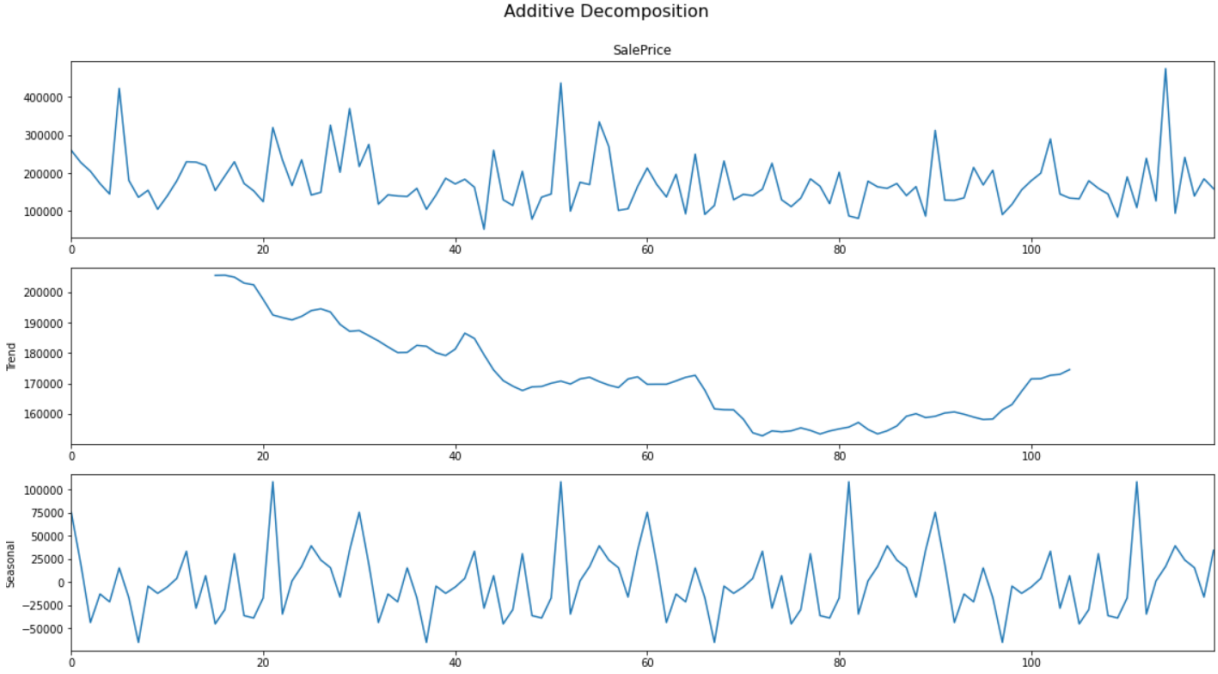


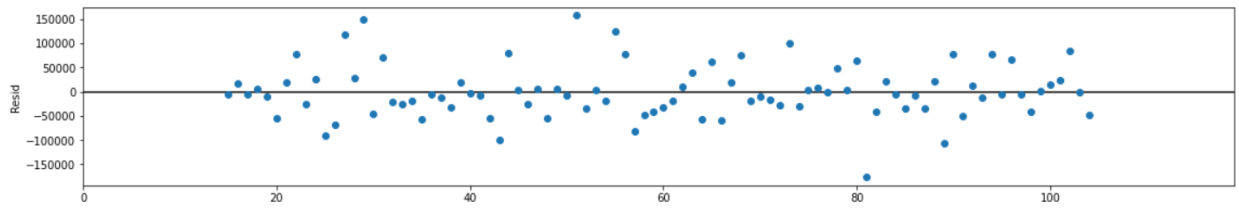
We can see that it is very difficult to gain insights from these plots as data fluctuates a lot.

Decomposition -

```
[53] from statsmodels.tsa.seasonal import seasonal_decompose
     from dateutil.parser import parse

     new_df=result_df.iloc[0:120,:]

     # Multiplicative Decomposition
     multiplicative_decomposition = seasonal_decompose(new_df['SalePrice'], model='multiplicative', period=30)

     # Additive Decomposition
     additive_decomposition = seasonal_decompose(new_df['SalePrice'], model='additive', period=30)

     # Plot
     plt.rcParams.update({'figure.figsize': (16,12)})
     multiplicative_decomposition.plot().suptitle('Multiplicative Decomposition', fontsize=16)
     plt.tight_layout(rect=[0, 0.03, 1, 0.95])

     additive_decomposition.plot().suptitle('Additive Decomposition', fontsize=16)
     plt.tight_layout(rect=[0, 0.03, 1, 0.95])

     plt.show()
```


Multiplicative Decomposition
SalePrice
Seasonal
Resid

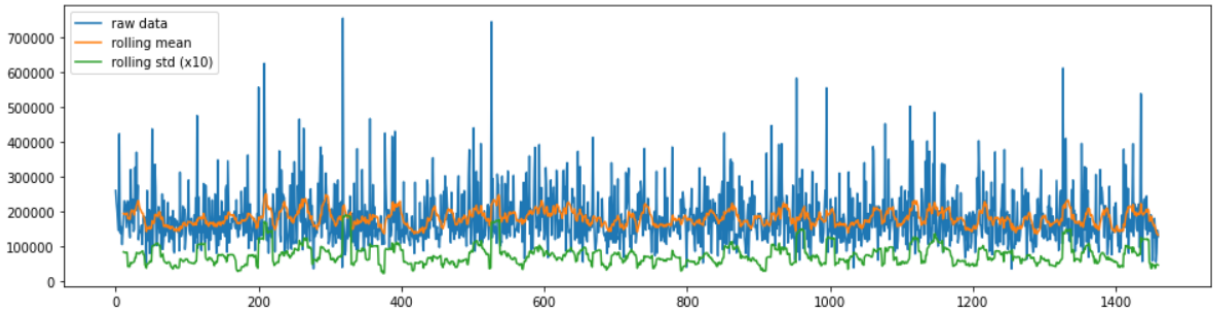

Additive Decomposition
SalePrice
Trend
Seasonal



Testing stationarity -

Stationarity is a key part of time series analysis. Simply put, stationarity means that the manner in which time series data changes is constant. A stationary time series will not have any trends or seasonal patterns.

```
[55] def test_stationarity(timeseries, title):

        #Determing rolling statistics
        rolmean = pd.Series(timeseries).rolling(window=12).mean()
        rolstd = pd.Series(timeseries).rolling(window=12).std()

        fig, ax = plt.subplots(figsize=(16, 4))
        ax.plot(timeseries, label= title)
        ax.plot(rolmean, label='rolling mean');
        ax.plot(rolstd, label='rolling std (x10)');
        ax.legend()
    pd.options.display.float_format = '{:.8f}'.format
    test_stationarity(series,'raw data')
```

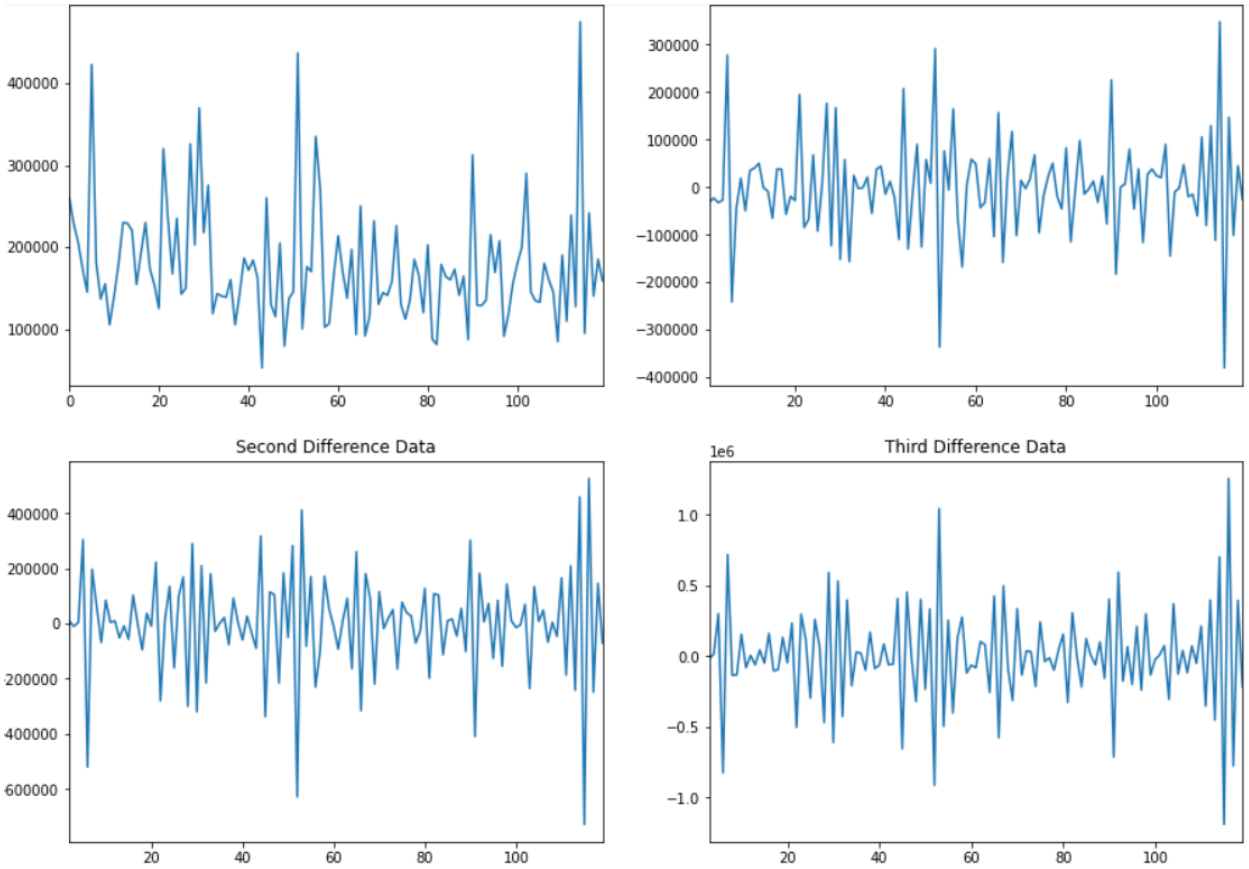



```
[56] result = adfuller(series, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'n_lags: {result[1]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')
```

```
ADF Statistic: -38.54252468592106
n_lags: 0.0
p-value: 0.0
Critial Values:
   1%, -3.4348399537053482
Critial Values:
   5%, -2.8635230163107837
Critial Values:
   10%, -2.5678257404326903
```

We can see from the Dickey Fuller test that the p value is less than 5 percent. Hence the data is stationary.

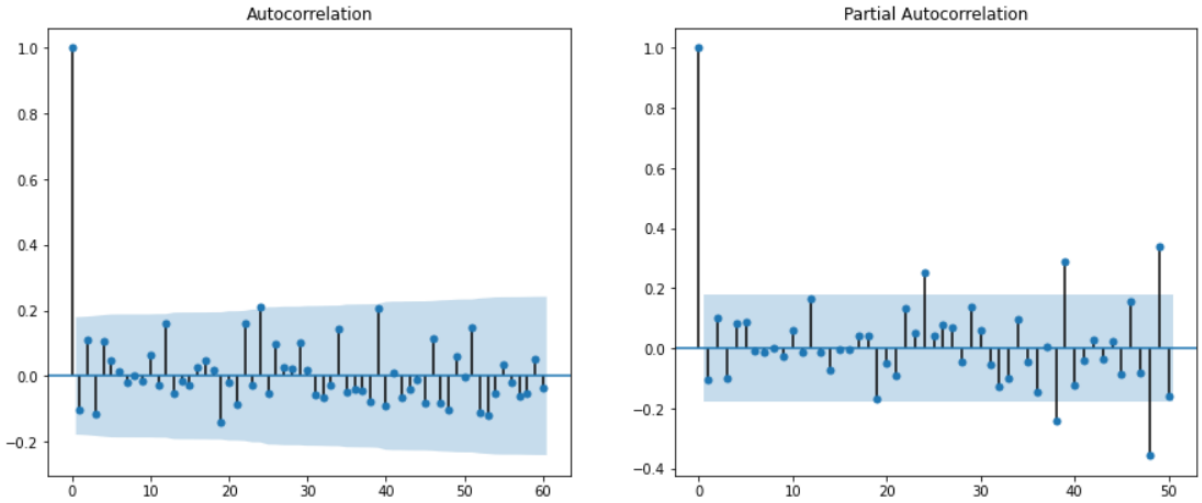
```
[57] from statsmodels.tsa.statespace.tools import diff
     fig, ax = plt.subplots(nrows=2, ncols=2,figsize=(15, 11))

     new_df['SalePriceDiff1'] = diff(new_df['SalePrice'],k_diff=1)
     new_df['SalePriceDiff2'] = diff(new_df['SalePrice'],k_diff=2)
     new_df['SalePriceDiff3'] = diff(new_df['SalePrice'],k_diff=3)

     new_df['SalePrice'].plot(title="Initial Data",ax=ax[0][0]).autoscale(axis='x',tight=True);
     new_df['SalePriceDiff1'].plot(title="First Difference Data",ax=ax[0][1]).autoscale(axis='x',tight=True);
     new_df['SalePriceDiff2'].plot(title="Second Difference Data",ax=ax[1][0]).autoscale(axis='x',tight=True);
     new_df['SalePriceDiff3'].plot(title="Third Difference Data",ax=ax[1][1]).autoscale(axis='x',tight=True);
```



This can be used to remove the series dependence on time, so-called temporal dependence. This includes structures like trends and seasonality.

We perform autocorrelation with different lags.

```
[58] from statsmodels.graphics.tsaplots import plot_acf,plot_pacf,acf
     fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(15, 6))
     autocorr = acf(new_df['SalePrice'], nlags=60) # just the numbers
     plot_acf(new_df['SalePrice'].tolist(), lags=60, ax=ax[0]); # just the plot
     plot_pacf(new_df['SalePrice'].tolist(), lags=50, ax=ax[1]); # just the plot
```




```
[60] autocorrelation_lag1 = result_df['SalePrice'].autocorr(lag=1)
     print("One Month Lag: ", autocorrelation_lag1)
```

```
One Month Lag:  -0.00951139838053
```

```
[61] autocorrelation_lag3 = result_df['SalePrice'].autocorr(lag=3)
     print("Three Month Lag: ", autocorrelation_lag3)

     autocorrelation_lag6 = result_df['SalePrice'].autocorr(lag=6)
     print("Six Month Lag: ", autocorrelation_lag6)

     autocorrelation_lag9 = result_df['SalePrice'].autocorr(lag=9)
     print("Nine Month Lag: ", autocorrelation_lag9)
```

```
Three Month Lag:  -0.0038466304887539364
Six Month Lag:  -0.02171094116713255
Nine Month Lag:  -0.02987550731914915
```

We see that, even with a nine-month lag, the data is not highly autocorrelated. This is further illustration of absence of the short- and long-term trends in the data.

Forecasting -

```
[65] impt_col = list(df_train.corr()["SalePrice"][(df_train.corr()["SalePrice"]>0.5) | (df_train.corr()["SalePrice"]<-0.5)].index)
```

```
[67] imp_df = df_train[impt_col]
```

```
[68] imp_df.isnull().sum()
```

```
OverallQual     0
YearBuilt       0
YearRemodAdd    0
TotalBsmtSF     0
1stFlrSF        0
GrLivArea       0
FullBath        0
TotRmsAbvGrd    0
GarageCars      0
GarageArea      0
SalePrice       0
dtype: int64
```

We choose only those attributes which are highly correlated with the target.

```
[69] x = imp_df.drop("SalePrice", axis = 1)
     y = imp_df["SalePrice"]
```

```
[70] from sklearn.model_selection import train_test_split, cross_val_score
     xtrain, xtest, ytrain, ytest = train_test_split(x,y, test_size =0.2, random_state = 42)
```

```
[71] from sklearn.linear_model import LinearRegression
     import statsmodels.api as sm
     lin_reg = LinearRegression()
```

```
[72] lin_reg.fit(xtrain,ytrain)
```

We split the train dataset into train and test and fit a linear regression model to the data.

```
[73] prediction = lin_reg.predict(xtest)
```

```
[74] from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
     lin_reg.score(xtrain,ytrain)
```

```
0.7647556828674686
```

We can see that we get an accuracy of 76.47%.

We now obtain predictions on the test data with the trained model.

```
[79] imp_df.columns
```

```
Index(['OverallQual', 'YearBuilt', 'YearRemodAdd', 'TotalBsmtSF', '1stFlrSF',
       'GrLivArea', 'FullBath', 'TotRmsAbvGrd', 'GarageCars', 'GarageArea',
       'SalePrice'],
      dtype='object')
```

```
[80] features = ['OverallQual', 'YearBuilt', 'YearRemodAdd', 'TotalBsmtSF', '1stFlrSF',
       'GrLivArea', 'FullBath', 'TotRmsAbvGrd', 'GarageCars', 'GarageArea']
     x_test = df_test[features]
```

```
[81] x_test.fillna(method='bfill', inplace=True)
```

```
[82] prediction_with_linear = lin_reg.predict(x_test)
```

Conclusions :

- We can conclude from auto-correlation scores that there is no short-term or long-term trend in the data.
- The stationary test concluded that the data is stationary, which further confirms that the data doesn't have any trends or seasonal patterns.
- The linear regression model fit to the train data gives an accuracy of 76.6%.
- We have used only those attributes from the data that are highly correlated with the target output.
- Trend decomposition is another useful way to visualise the trends in time series data. From the plot, we can see a decreasing trend at first and then an increasing trend in house sale prices.