Experiment No 10

* 1. **Aim/Purpose of the Experiment**

To familiarize the students with model evaluation methods using R square, Adjusted R square and VIF for a linear regression.

* 1. **Learning Outcomes**

Knowledge of the model evaluation methods using R square, Adjusted R square and VIF for a linear regression.in python.

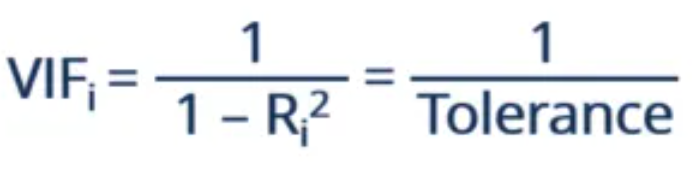
* 1. **Prerequisites**

Basic knowledge of programming, python syntax, matplotlib, seaborn, different libraries.

* 1. **Materials/Equipment/Apparatus / Devices/Software required**

Jupyter Notebook.

* 1. **Introduction and Theory**

1. ***R*2 (R-squared)**:
   * �2*R*2 is a measure of how well the independent variables explain the variability of the dependent variable in the model.
   * It ranges from 0 to 1, with higher values indicating a better fit.
   * However, �2*R*2 tends to increase as you add more independent variables to the model, even if they don't improve the model's predictive power.
   * Therefore, it's essential to use adjusted �2*R*2 when comparing models with different numbers of predictors.
   * R-squared = (TSS-RSS)/TSS
   * TSS = Total variation in target variable is the sum of squares of the difference between the actual values and their mean.
   * RSS = Sum of square of the Residual obtained for a point where Residual in the data is the difference between the actual value and the value predicted by the linear regression model.
2. **Adjusted �2*R*2 (adjusted R-squared)**:
   * Adjusted �2*R*2 takes into account the number of predictors in the model and penalizes overly complex models.
   * It adjusts �2*R*2 downward if adding more predictors doesn't significantly improve the model's fit.
   * Like �2*R*2, it ranges from 0 to 1, with higher values indicating a better fit.
   * Adjusted �2*R*2 is preferred over �2*R*2 for comparing models with different numbers of predictors.
   *  Here,
   * n represents the number of data points in our dataset
   * k represents the number of independent variables, and
   * R represents the R-squared values determined by the model.
3. **Variance Inflation Factor (VIF)**:
   * VIF measures the extent to which the variance of an estimated regression coefficient increases because of multicollinearity.
   * Multicollinearity occurs when independent variables in a regression model are highly correlated with each other.
   * VIF values greater than 10 are often considered indicative of multicollinearity, although the threshold may vary depending on the context.
   * High VIF values suggest that the regression coefficients may be unstable due to multicollinearity, which can lead to unreliable estimates and inflated standard errors.
   * 

To evaluate a linear regression model using these metrics in Python, you can follow these steps:

1. Fit the linear regression model using a library such as scikit-learn or statsmodels.
2. Calculate �2*R*2 and adjusted �2*R*2 to assess the goodness of fit.
3. Calculate VIF for each predictor variable to check for multicollinearity.

Housing Case Study

Problem Statement:

Consider a real estate company that has a dataset containing the prices of properties in the Delhi

region. It wishes to use the data to optimise the sale prices of the properties based on important

factors such as area, bedrooms, parking, etc.

Essentially, the company wants —

• To identify the variables affecting house prices, e.g. area, number of rooms,

bathrooms, etc.

• To create a linear model that quantitatively relates house prices with variables such

as number of rooms, area, number of bathrooms, etc.

• To know the accuracy of the model, i.e. how well these variables can predict house

prices.

So interpretation is important!

Step 1: Reading and Understanding the Data

Let us first import NumPy and Pandas and read the housing dataset

# Supress Warnings

import warnings

warnings.filterwarnings('ignore')

import numpy as np

import pandas as pd

housing = pd.read\_csv("Housing.csv")

# Check the head of the dataset

housing.head()

housing.shape

housing.info()

housing.describe()

Step 2: Visualising the Data

Let's now spend some time doing what is arguably the most important step - understanding the

data.

• If there is some obvious multicollinearity going on, this is the first place to catch it

• Here's where you'll also identify if some predictors directly have a strong association

with the outcome variable

We'll visualise our data using matplotlib and seaborn.

import matplotlib.pyplot as plt

import seaborn as sns

Visualising Numeric Variables

Let's make a pairplot of all the numeric variables

sns.pairplot(housing)

plt.show()

Visualising Categorical Variables

As you might have noticed, there are a few categorical variables as well. Let's make a boxplot for

some of these variables.

plt.figure(figsize=(20, 12))

plt.subplot(2,3,1)

sns.boxplot(x = 'mainroad', y = 'price', data = housing)

plt.subplot(2,3,2)

sns.boxplot(x = 'guestroom', y = 'price', data = housing)

plt.subplot(2,3,3)

sns.boxplot(x = 'basement', y = 'price', data = housing)

plt.subplot(2,3,4)

sns.boxplot(x = 'hotwaterheating', y = 'price', data = housing)

plt.subplot(2,3,5)

sns.boxplot(x = 'airconditioning', y = 'price', data = housing)

plt.subplot(2,3,6)

sns.boxplot(x = 'furnishingstatus', y = 'price', data = housing)

plt.show()

We can also visualise some of these categorical features parallely by using the hue argument.

Below is the plot for furnishingstatus with airconditioning as the hue.

plt.figure(figsize = (10, 5))

sns.boxplot(x = 'furnishingstatus', y = 'price', hue =

'airconditioning', data = housing)

plt.show()

Step 3: Data Preparation

• You can see that your dataset has many columns with values as 'Yes' or 'No'.

• But in order to fit a regression line, we would need numerical values and not string.

Hence, we need to convert them to 1s and 0s, where 1 is a 'Yes' and 0 is a 'No'.

# List of variables to map

varlist = ['mainroad', 'guestroom', 'basement', 'hotwaterheating',

'airconditioning', 'prefarea']

# Defining the map function

def binary\_map(x):

return x.map({'yes': 1, "no": 0})

# Applying the function to the housing list

housing[varlist] = housing[varlist].apply(binary\_map)

# Check the housing dataframe now

housing.head()

Dummy Variables

The variable furnishingstatus has three levels. We need to convert these levels into integer

as well.

For this, we will use something called dummy variables.

# Get the dummy variables for the feature 'furnishingstatus' and store

it in a new variable - 'status'

status = pd.get\_dummies(housing['furnishingstatus'])

# Check what the dataset 'status' looks like

status.head()

Now, you don't need three columns. You can drop the furnished column, as the type of

furnishing can be identified with just the last two columns where —

• 00 will correspond to furnished

• 01 will correspond to unfurnished

• 10 will correspond to semi-furnished

# Let's drop the first column from status df using 'drop\_first = True'

status = pd.get\_dummies(housing['furnishingstatus'], drop\_first =

True)

# Add the results to the original housing dataframe

housing = pd.concat([housing, status], axis = 1)

# Now let's see the head of our dataframe.

housing.head()

# Drop 'furnishingstatus' as we have created the dummies for it

housing.drop(['furnishingstatus'], axis = 1, inplace = True)

housing.head()

Step 4: Splitting the Data into Training and Testing Sets

As you know, the first basic step for regression is performing a train-test split.

from sklearn.model\_selection import train\_test\_split

# We specify this so that the train and test data set always have the

same rows, respectively

np.random.seed(0)

df\_train, df\_test = train\_test\_split(housing, train\_size = 0.7,

test\_size = 0.3, random\_state = 100)

Rescaling the Features

As you saw in the demonstration for Simple Linear Regression, scaling doesn't impact your

model. Here we can see that except for area, all the columns have small integer values. So it is

extremely important to rescale the variables so that they have a comparable scale. If we don't

have comparable scales, then some of the coefficients as obtained by fitting the regression

model might be very large or very small as compared to the other coefficients. This might

become very annoying at the time of model evaluation. So it is advised to use standardization or

normalization so that the units of the coefficients obtained are all on the same scale. As you

know, there are two common ways of rescaling:

1. Min-Max scaling

2. Standardisation (mean-0, sigma-1)

This time, we will use MinMax scaling.

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

# Apply scaler() to all the columns except the 'yes-no' and 'dummy'

variables

num\_vars = ['area', 'bedrooms', 'bathrooms', 'stories',

'parking','price']

df\_train[num\_vars] = scaler.fit\_transform(df\_train[num\_vars])

df\_train.head()

# Let's check the correlation coefficients to see which variables are

highly correlated

plt.figure(figsize = (16, 10))

sns.heatmap(df\_train.corr(), annot = True, cmap="YlGnBu")

plt.show()

As you might have noticed, area seems to the correlated to price the most. Let's see a pairplot

for area vs price.

plt.figure(figsize=[6,6])

plt.scatter(df\_train.area, df\_train.price)

plt.show()

So, we pick area as the first variable and we'll try to fit a regression line to that.

Dividing into X and Y sets for the model building

y\_train = df\_train.pop('price')

X\_train = df\_train

Step 5: Building a linear model

Fit a regression line through the training data using statsmodels. Remember that in

statsmodels, you need to explicitly fit a constant using sm.add\_constant(X) because if we

don't perform this step, statsmodels fits a regression line passing through the origin, by

default.

import statsmodels.api as sm

# Add a constant

X\_train\_lm = sm.add\_constant(X\_train[['area']])

# Create a first fitted model

lr = sm.OLS(y\_train, X\_train\_lm).fit()

# Check the parameters obtained

lr.params

# Let's visualise the data with a scatter plot and the fitted

regression line

plt.scatter(X\_train\_lm.iloc[:, 1], y\_train)

plt.plot(X\_train\_lm.iloc[:, 1], 0.127 + 0.462\*X\_train\_lm.iloc[:, 1],

'r')

plt.show()

# Print a summary of the linear regression model obtained

print(lr.summary())

Adding another variable

The R-squared value obtained is 0.283. Since we have so many variables, we can clearly do

better than this. So let's go ahead and add the second most highly correlated variable, i.e.

bathrooms.

# Assign all the feature variables to X

X\_train\_lm = X\_train[['area', 'bathrooms']]

# Build a linear model

import statsmodels.api as sm

X\_train\_lm = sm.add\_constant(X\_train\_lm)

lr = sm.OLS(y\_train, X\_train\_lm).fit()

lr.params

# Check the summary

print(lr.summary())

We have clearly improved the model as the value of adjusted R-squared as its value has gone up

to 0.477 from 0.281. Let's go ahead and add another variable, bedrooms.

# Assign all the feature variables to X

X\_train\_lm = X\_train[['area', 'bathrooms','bedrooms']]

# Build a linear model

import statsmodels.api as sm

X\_train\_lm = sm.add\_constant(X\_train\_lm)

lr = sm.OLS(y\_train, X\_train\_lm).fit()

lr.params

# Print the summary of the model

print(lr.summary())

Adding all the variables to the model

# Check all the columns of the dataframe

housing.columns

Index(['price', 'area', 'bedrooms', 'bathrooms', 'stories',

'mainroad',

'guestroom', 'basement', 'hotwaterheating', 'airconditioning',

'parking', 'prefarea', 'semi-furnished', 'unfurnished'],

dtype='object')

#Build a linear model

import statsmodels.api as sm

X\_train\_lm = sm.add\_constant(X\_train)

lr\_1 = sm.OLS(y\_train, X\_train\_lm).fit()

lr\_1.params

print(lr\_1.summary())

Looking at the p-values, it looks like some of the variables aren't really significant (in the

presence of other variables).

Maybe we could drop some?

We could simply drop the variable with the highest, non-significant p value. A better way would

be to supplement this with the VIF information.

Checking VIF

Variance Inflation Factor or VIF, gives a basic quantitative idea about how much the feature

variables are correlated with each other. It is an extremely important parameter to test our linear

model. The formula for calculating VIF is:

$ VIF\_i = \frac{1}{1 - {R\_i}^2} $

# Check for the VIF values of the feature variables.

from statsmodels.stats.outliers\_influence import

variance\_inflation\_factor

# Create a dataframe that will contain the names of all the feature

variables and their respective VIFs

vif = pd.DataFrame()

vif['Features'] = X\_train.columns

vif['VIF'] = [variance\_inflation\_factor(X\_train.values, i) for i in

range(X\_train.shape[1])]

vif['VIF'] = round(vif['VIF'], 2)

vif = vif.sort\_values(by = "VIF", ascending = False)

vif

* 1. **Operating Procedure**
* Open Jupyter note book
* Take a new python file
* Type the code
* Run it
* Take inputs from user
* Observe the results
* Verify the results manually
* Store the note book file
  1. **Precautions and/or Troubleshooting**

**Precautions:**

* Save Your Work: Regularly save your Jupyter Notebook to avoid losing your work. You can save your notebook by clicking on the save icon or using the keyboard shortcut Ctrl + S (or Cmd + S on Mac).
* Restart Kernel: If you encounter unexpected behavior or errors, try restarting the kernel. This clears all the variables and imported modules, essentially resetting the notebook's state. You can restart the kernel by going to the "Kernel" menu and selecting "Restart."
* Clear Outputs: To reduce clutter and confusion, consider clearing the outputs of code cells that are no longer relevant. You can do this by selecting "Clear Outputs" from the "Edit" menu.
* Readability: Keep your code and comments clear and well-organized to make it easier to understand and maintain. Use markdown cells for explanations, headings, and documentation.
* Check Dependencies: If you're using external libraries or packages, ensure they are properly installed in your Jupyter environment. You can check the installed packages by running !pip list or !conda list in a code cell.
* Kernel Selection: Make sure you're using the correct kernel for your notebook. The kernel determines the programming language and environment in which your code runs. You can change the kernel by clicking on "Kernel" > "Change kernel" in the menu.
* Resource Usage: Be mindful of the resources your notebook is using, especially if you're working with large datasets or running intensive computations. Check system monitor tools to ensure you're not exhausting memory or CPU resources.

**Troubleshooting:**

* Syntax Errors: Check for syntax errors in your code. Python is sensitive to indentation and syntax, so ensure your code is properly formatted.
* Variable Scope: Be aware of variable scope issues, especially if you're reusing variable names or working with nested functions.
* Library Installation: If you encounter Module Not Found Error or similar errors, ensure that the required libraries are installed in your Jupyter environment. You can install libraries using !pip install <library> or !conda install <library> in a code cell.
* Kernel Crashes: If the kernel crashes frequently, consider reducing the complexity of your code or optimizing resource usage. Large datasets or intensive computations can sometimes overwhelm the kernel.
* Browser Issues: If you experience rendering or responsiveness issues in the notebook interface, try clearing your browser cache or using a different browser.
* Documentation: Consult the official Jupyter documentation and community forums for additional troubleshooting tips and solutions to common problems.
  1. **Observations**

Observe the results obtained in each operation.

* 1. **Calculations & Analysis**

Calculations should be given for each operation.

* 1. **Result & Interpretation**

Result should be printed and pasted in laboratory copy found from Jupyter note book.

* 1. **Follow-up Questions**
  + You need to check the relationship between the two variables. Which graph would you use?
  + You need to check if a variable has outliers. Which graph would you use?
  + You need to perform a univariate analysis. Which graph will you use?
  + What is a data cleaning step?
  + What are the ways to handle missing data?
  + What are some of the methods for univariate analysis?
  + What problems can outliers cause?
  + What is R2 square?
  + What is adjusted R2square?
  + What is VIF?
  1. **Extension and Follow-up Activities (if applicable)**

NA

* 1. **Assessments**
  2. **Suggested reading**

NA