CST4050 - Individual Assessment (comp 2)

Student:

Name: Angad Partap

Surname: Singh

Student number: M00912257

Campus: London

Loading Data / Loading Basic Libraries

In [3]:
```
## Importing Basic Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

IMPORTING THE DATA

In [4]:
```
# Loading the Data
# Importing the data

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

Out[4]:

	x1	x2	x3	x4	x5	x6	x7	x8	y
0	0.592109	0.545496	0.199054	2.370339	1.032510	1.137605	1.199802	-0.833456	32.175299
1	0.911316	1.260952	0.446375	1.564526	0.820080	2.172268	1.441165	-0.373705	6.521285
2	0.968683	3.744257	1.931173	1.472553	0.102181	4.712941	1.954805	1.828992	5.688139
3	0.748656	2.741351	1.790573	1.696788	0.464028	3.490008	0.948294	1.326545	8.488786
4	0.320502	3.196858	1.050494	2.813036	0.204284	3.517361	1.273237	0.846210	63.570984

Our dataset contains 8 Independent features (x1, x2, x3, x4,..., x8).

There is One Dependent Feature (y).

Our target attribute (y) is numerical.

As our outcome (y) is numerical, this is a 'REGRESSION PROBLEM'.

TARGET ATTRIBUTE / DEPENDENT FEATURE

1. Our target feature 'y' is of datatype 'float'. The describe() technique lets us know more about our target feature (mean, standard deviation etc, 25th,50th, 75th percentiles etc). There are 150 observations in the target comlumn. Min possible value is 0.081 and max possible value is 80.673

In [91]:
```
print(df['y'].describe())
```

```
count    150.000000
mean      15.573634
std       21.532456
min        0.081166
25%        0.855930
50%        4.188566
75%       22.455500
max       80.673096
Name: y, dtype: float64
```

Target = Numercial / Nominal.

Our target feature is NUMERICAL.

Dataset : Low-Dimension/ High Dimensional ???

Since our number of observations outnumbers the features in our data set (150 x 9).

Our Dataset is a LOW-DIMENSIONAL dataset.

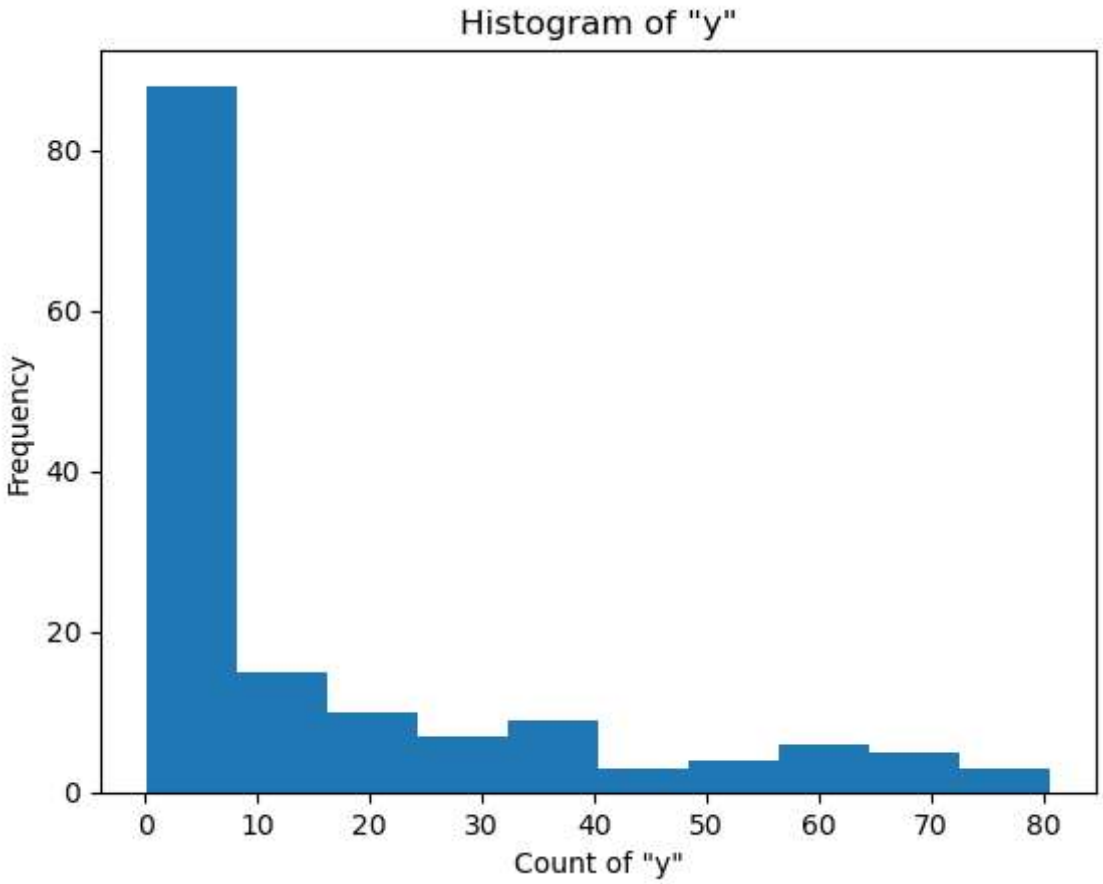
Supervised Problem = Classification / Regression.

The approach we would follow is REGRESSION because our target variable is continous in nature.

Frequency Distribution of Target Attribute.

The x-axis shows the values of 'y' and the y_axis shows the feaquency of occurences of values of 'y'.The histogram shows that more then 85% of the values of 'y' are between 0-10. The rest of the values are spread out between 10-80. Approximately 10%-15% values are between 10-40, And less then 10% values are between 40-80. The histograms is RIGHT-SKEWED. The left side being more populated with values then right, thus non-uniform distribution of values.

In [92]:
```
plt.hist(df['y'])
plt.xlabel('Count of "y"')
plt.ylabel('Frequency')
plt.title('Histogram of "y"')
plt.show()
```



Checking the features

After the removal of 'y', all the remaining features are our INDEPENDENT FEATURES. We would DROP 'y' so as to get the remaining Independent features. Axis = 1 helps us to drop columns. By default its axis=0 (drop a rox), so we need to mention axis = 1 specifically

In [96]:
```
features = df.drop('y', axis=1)
features
```

Out[96]:

	x1	x2	x3	x4	x5	x6	x7	x8
0	0.592109	0.545496	0.199054	2.370339	1.032510	1.137605	1.199802	-0.833456
1	0.911316	1.260952	0.446375	1.564526	0.820080	2.172268	1.441165	-0.373705
2	0.968683	3.744257	1.931173	1.472553	0.102181	4.712941	1.954805	1.828992
3	0.748656	2.741351	1.790573	1.696788	0.464028	3.490008	0.948294	1.326545
4	0.320502	3.196858	1.050494	2.813036	0.204284	3.517361	1.273237	0.846210
...	...	...	...	...	...	...	...	...
145	0.558799	4.116570	1.908279	1.656721	0.207313	4.675369	1.170346	1.700966
146	0.086132	0.238454	0.410397	2.399203	0.511865	0.324586	0.215530	-0.101468
147	0.789870	3.326645	0.981375	1.035684	0.820560	4.116515	1.295516	0.160816
148	0.328342	0.642812	0.066570	0.759085	0.084414	0.971154	1.312087	-0.017844
149	0.230653	2.720530	1.963448	1.058873	0.472453	2.951183	0.395287	1.490995

150 rows × 8 columns

Shape of data : Rows x Columns

Our dataset contains 150 rows and 8 columns. Meaning 150 Observations in 8 Columns. The 8 Columns represent our Independent features.

Standadization of data

We need to know whether our data is standardized or not.

In [97]:

```
## Checking whther our data is standardized or not.

# Calculate the mean and standard deviation of each feature
mean = df.mean()
std = df.std()

# Print the mean and standard deviation of each feature
print('Mean:\n', mean)
print('\nStandard deviation:\n', std)

# Check if the mean is close to 0 and standard deviation is close to 1 for each fe
is_standardized = (mean.abs() < 0.1) & (std.abs() > 0.9) & (std.abs() < 1.1)

# Print the result
print('Are features standardized:', is_standardized.all())

## We get the information now that are features are not standardized.
## Thus we need to standardize our data.
## We would do the standardization once we divide the data into train and test spl
```

```
Mean:
 x1     0.526873
x2     2.532959
x3     1.004755
x4     1.437147
x5     0.608091
x6     3.059833
x7     1.005223
x8     0.396664
y     15.573634
dtype: float64

Standard deviation:
 x1     0.287178
x2     1.366323
x3     0.591567
x4     0.869658
x5     0.365670
x6     1.405730
x7     0.418172
x8     0.666874
y     21.532456
dtype: float64
Are features standardized: False
```

This shows that our data is NOT STANDARDIZED.

We need to know whether our data's mean is 0 and standard deviation is 1. This is the basis of a standardization. If the condition is not met, means our data is not standardized.

The above lines of code shows that our data is NOT STANDARDIZED.

Please note that we would follow the RANDOM FOREST TECHNIQUE and Decision Tree. As our dataset is small, we wouldn't need to standardize the data. However in case we would have a huge data, is the time we might need standardizing.

Machine Learning Pipeline (Training & Testing)

The following steps need to be followed for building our pipeline.

1. DATA PREPROCESSING : The methods like feature scaling, standardization of the data are done in this step. It is made sure that our data should be on same scale.

2. FEATURE SELECTION: Relevant features need to be selected for better performance of the model.

3. MODEL SELECTION : A proper machine-learning-model is sleected for better output and performance.

4. TRAINING THE MODEL : Our selected model is trained on trained data and then tested on the test data. The train-test split method comes handy in this step.

5. TUNING THE MODEL :This is a very important step. The best hyperparameters, and their best values are selected such that we get the best predictions from our model.

6. DEPLOYING OUR MODEL : Our trained and tested model is then traveresed from the unseen data to check how it is performing on the new dataset introduced.

The reason for Selection of Proposed Pipeline.

The above pipeline consists of all the necessary steps to be taken during a regression problem. We do the Feature Engineering, Model Selection, Tuning, Hyperparameters Tuning. This all steps always come handy when we are dealing with a Regression Problem.

Our Suggested Pipeline standardizes the data ?

Our suggested pipeline does standardize the data. The need of the same is that the data needs to be on the same scale before putting it through a model.

However, please note, our suggested Models do not require Standardizing as Random Forest Regressor stands Robust when dealing with Non-standardized data.

Whether Our mentioned Pipeline transforms the data ?

Our pipeline would not transform the data. As it is a small dataset.

In case we introduce a larger dataset is the time we might require to standardize it.

Splitting the dataset into Train and Test splits.

Tuning & Validation Split

Splitting the data into Tuning and Valiation Splits.

Model would be trained on Tuning Data.

Accuracy would be checked on Validation data. Thus Validation Data is kept as UNSEEN.

TUNING DATA = 70% of total data

VALIDATION DATA = 30% of total data

Our random state = 42 tells us the random seed = 42 meaning that every time the same split is generated when the code is run.

X_TRAIN = training subset of the Independent features.

X_test = Testing subset of the Independent features.

y_train = training subset of the Dependent Feature.

y_test = testing subset of the Dependent feature.

In [101…
```
#separate dataset into train - test split

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_
```

Features of Tuning Data have 105 Observations.

And that of Validation Data have 45 Observations.

In [102…
```
X_train.shape, X_test.shape
```

Out[102]:
```
((105, 8), (45, 8))
```

Training our Model / Pipeline with Random Forest Regressor

We would Import Random Forest Regressor from SciKit Learn Library. Creating a pipeling of our model and making sure that the pipeline first fits on our train data.

Post that, the pipeline is used to do predictions and then the accuracy of the model is calculated.

We've created the empty vectors of accuracies for the accuracy values. The 'append' is sued to fill out the values of accuracies.

In [105…
```
## Importing Basic Libraries
import pandas as pd
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler  ## Libraries for Scaling
from sklearn.ensemble import RandomForestRegressor  ## Libraries for Random Forest
from sklearn import datasets, metrics  ## Libraries for checking the metrics for th

# Creating the Empty vectors of scores
R2_train = []
R2_test = []

steps=[("Regressor",RandomForestRegressor())] #To create our pipeline

pipe.fit(X_train,y_train) #Fitting our pipeline to training data.

# Use the trained model to create predictions on train and test data
y_train_pred = pipe.predict(X_train)
y_test_pred = pipe.predict(X_test)


## R-squared value is used to cehck the goodness of our model.
## The value varies from 0-1. The more it is towards 1, the more our model is bette
# Computing the accuracy of each model
from sklearn.metrics import r2_score
R2_score_train = pipe.score(X_train, y_train)
R2_score_test = pipe.score(X_test, y_test)


# Updating the vector of accuracies
R2_train.append( R2_score_train )
R2_test.append( R2_score_test )


print("Mean R2 on train:", np.mean( R2_train ))
print("Mean R2 on test:", np.mean( R2_test ))
```

```
Mean R2 on train: 1.0
Mean R2 on test: 0.9921529032430193
```

Comparing the accuracy to the baseline.

In [107…

```
# Calculate baseline using mean target variable
baseline = y_test.mean()
baseline_rmse = mean_squared_error(y_test, [baseline]*len(y_test), squared=False)
baseline_r2 = r2_score(y_test, [baseline]*len(y_test))
print('Baseline RMSE is:', baseline_rmse)
print('Baseline R-squared is:', baseline_r2)
```

```
Baseline RMSE is: 19.49181528074548
Baseline R-squared is: 0.0
```

The comparison with baseline accuracy is important for the performance of the model.

The variable 'Baseline' contains the mean of the target variable (y_test).

The values of actual y_test and baselien prediction is stored in variable 'Baseline RMSE'.

tHE r_squared value is calculated as a part of accuracy score of the regressor model.

Model OVERFITS / UNDERFITS

Once we perform the test both on test and train data, we would be able to deduce whether our model overfits or underfits. The comparison would be done both on training data and test data and basis MSE AND R2 value we can decide the above.

The R2 on trian is 1.0 and R2 on test 0.99 means our model is perfect.

Tuning the Proposed Pipeline.

The Tuning is done using GridSearch Cross Validation.

Grid-Search Cross Validation

Hyperparameter Tuning

We'll check how we can alter the values of our parameters to get the best output.

The Cross Valdiation would work on Tuning Data and the best parameters selected would then be used to predict outcomes.

In [112…

```
from sklearn.model_selection import GridSearchCV

parameter = {
    'criterion':['friedman_mse'],
    'n_estimators':[1,2,3,4,5,6,7,8,9,10],
    'max_depth':[1,2,3,4,5,6,7,8,9,10],
    'random_state':[1,2,3,4,5,42,50]
}
```

Scatter plot ACTUAL DATA vs. PREDICTED DATA

We can check how our model is performing with the graph below.

It shows the linear relation with the actual data and the predicted data.

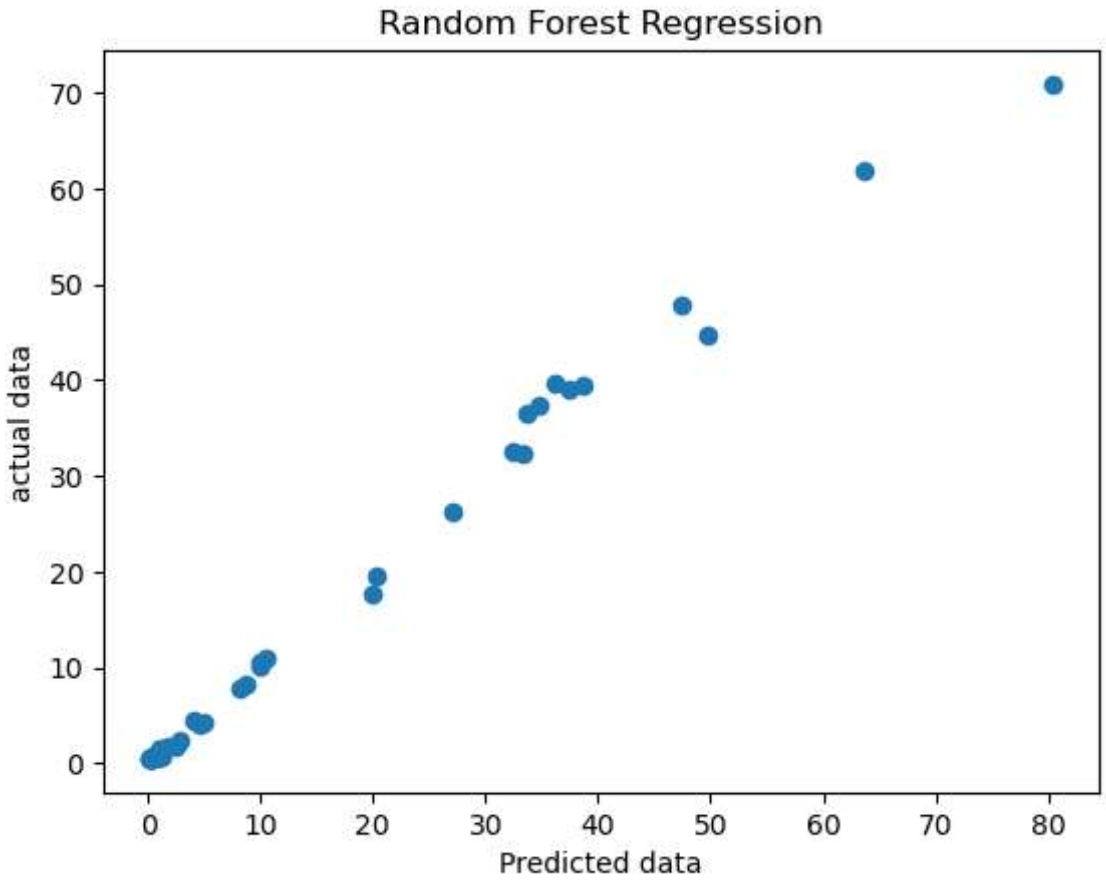
Thus Random Forest Model is working amazing.

The linear relation of predicted and actual data validates our assumption.

Thus our model is working well

In [141…

```
plt.scatter(y_test, y_pred)
plt.title('Random Forest Regression')
plt.xlabel('Predicted data')
plt.ylabel('actual data')
plt.show()
```



It shows the linear relation with the actual data and the predicted data.

DEDUCTIONS (Decision Tree or Random Forest)

We can deduce from above that Random Forest works better then Decision Tree.However our dataset is small so there is not much of the variation that we can tell.

The R2 value in case of Random Forest is greater in training data then in Test data. This means our Random Forest model is not Overfitting and can give better predictions.

Any Fine Tuned phase required.

Our model doesnt require any fine tuned phase as the data is less and the GridSearch CV is enough to get the best predictions.

Is there any requirement of data to be split into train and test, k-fold,

validation set, or what?

There is no requirement of splitting the data as already mentioned that the data is so less that Random forest works well with the normal Cross Validation and doesnt require any

more splitting.

How the model is tuned and which protion is selected for tuning ?

The model is tuned basis GridSearchCV with the n_estimators, max_depth, random_state,and mse.

All these methods are enough for our tuning phase.

n_estimators tells us how many Decision trees need to be used. max_depth tells us how many splits are required for each of the tree. The random_state generates a seed value such that there is a same split every time the code is run, and mse describes the accuracy of the model.

Validating the Random Forest Regressor Model.

Now we can go ahead and make use of Unseen Data with Random Forest Model.

The Mse and R2 value accuracies would justify that our moel is a 'good-fit'

In [113…

```
from sklearn.metrics import mean_squared_error # Library to assess the accuracy of

steps=[("Regressor",RandomForestRegressor())]

pipe.fit(X_train,y_train)

y_train_pred = pipe.predict(X_train)
y_test_pred = pipe.predict(X_test)

# Calculating the accuracies

#mean_squared_error
## This helps to measure the amount of errors in the moel.
## it calculates the the average squared difference between the observed and predi
## The lesser the mse, the better the predicitons.

from sklearn.metrics import mean_squared_error
mse_test = mean_squared_error(y_train_pred, y_train)
mse_train = mean_squared_error(y_train_pred, y_train)
R2_train = pipe.score(X_train, y_train)
R2_test = pipe.score(X_test, y_test)

print("MSE on tuning data:", mse_test)
print("MSE on validation data:", mse_train)
print(" --- ")
print("R2 on tuning data:", R2_train)
print("R2 on validation data:", R2_test)
```

```
MSE on tuning data: 0.0
MSE on validation data: 0.0
 ---
R2 on tuning data: 1.0
R2 on validation data: 0.9923621057677046
```

The R2 value on the Unseen Data is 0.99.

Thats an Unbiased approach making our model fit for the predictions with Low Bias and Low Variance.

The HYPER-PARAMETERS we are tuning.

Reason for their selection?

Their role in Bias-Variance Trade-off ?

The hyperparametrs we are tuning are n_estimators, max_depth, random_state, mse.

n_estimators tells us how many Decision trees need to be used. max_depth tells us how many splits are required for each of the tree. The random_state generates a seed value such that there is a same split every time the code is run, and mse describes the accuracy of the model.

Range of n_estimators is taken from (1 to 10). Max_depth range is (1 to 10). Random_state (1 to 50).

The reason for selection of these paramets and their value ranges si that our dataset is small and we do not want to overcomplicate thing by taking highers values. Based on our domain knowledge, we know that our model would perform wel with these parameters and their values.

The number of trees can be increased however,

Increasing the number of trees and the maximum depth of the trees can help us to reduce bias. However, increasing the minimum number of samples required to split a node can reduce the depth of the tree and prevent overfitting, thus reducing variance. By acheiving appropiate values for these hyperparameters can help us balance the bias and variance trade-off and result in better performance on new data.

Validation of Our tuned Parameters.

We have taken an obejct regresssord such that it would take only the best parameters from the GridSearchCV and would apply those parameters to train out model and same way get predictions out of it.

Implementing Random Forest for the predictions and checking the Coefficients.

We've used GridSearchCV for best parameters.

In [122…
```
## Hyperparameter Tuning
## We'll check how we can alter the values of our parameters to get the best outpu

## Library for Random Forest Regression
from sklearn.ensemble import RandomForestRegressor

parameter = {
    'criterion':['friedman_mse'],
    'n_estimators':[1,2,3,4,5,6,7,8,9,10],
    'max_depth':[1,2,3,4,5,6,7,8,9,10],
    'random_state':[1,2,3,4,5,42,50]
}
regressord = RandomForestRegressor()
```

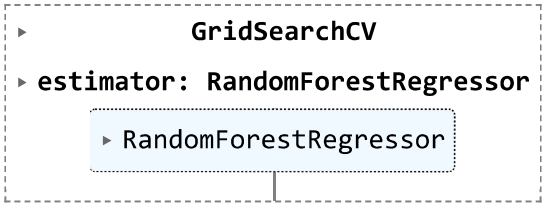
In [126…
```
## WGrid Search Cross Validation would help us with the best values of our paramet
## Such that we get the best model.

from sklearn.model_selection import GridSearchCV
regressorcvd = GridSearchCV(regressord, param_grid = parameter, cv=5, scoring ='ne
```

In [127…
```
## We would fit the best parameters with our train data first

regressorcvd.fit(X_train, y_train)
```

Out[127]:



In [128…
```
## The best values for our parameters would be:

regressorcvd.best_params_
```

Out[128]:
```
{'criterion': 'friedman_mse',
 'max_depth': 5,
 'n_estimators': 9,
 'random_state': 4}
```

In [135…
```
## We need to use 9 decision trees (n_estimators = 9).
## Further, the random_state (4) would ensure that our process would output the sa
## max_depth = 5, depicts the number of splits each decision is allowed to make.
## friedman_mse tells represents average squared residual.

y_pred = regressorcvd.predict(X_test)
```

In [137…
```
regressor.feature_importances_
```

Out[137]:
```
array([7.05290392e-04, 1.23996790e-04, 1.42842356e-03, 9.96235473e-01,
       8.21619252e-04, 3.08713377e-04, 1.08628646e-04, 2.67855475e-04])
```

In [131…
```
Beta = pd.DataFrame(regressor.feature_importances_)
Beta.describe()
```

Out[131]:

	0
count	8.000000
mean	0.125000
std	0.352033
min	0.000109
25%	0.000232
50%	0.000507
75%	0.000973
max	0.996235

Random Forest generates coefficients which are almost close to Zero. Thus a better model for both tuning and testing data.

Tuned Model Overfits or Underfits

Our model is performing well as we have checked with the coefficients, r2 value, and MSE values

Interpretation of our Model

. By Human : Our model is easy to be interpreted as it does not contains many lines of codes. Also the Data Preprocessing steps as well as Feature Engineering isn't that complex which cant be interpreted by Humans.

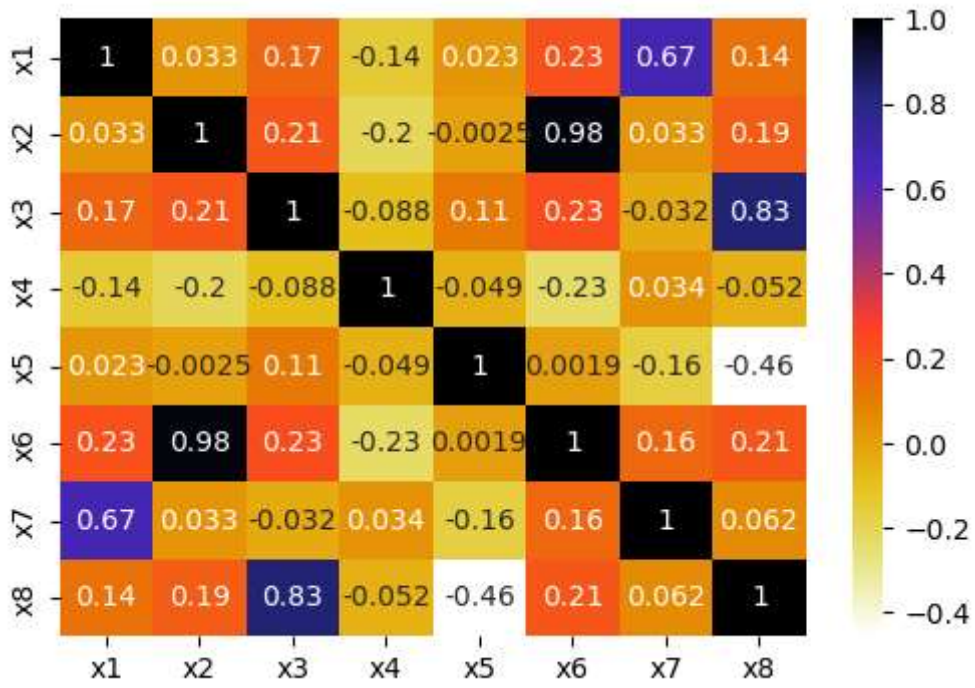
Analyzing Multicollinearity

Checking the collinearity of the Independent Features Using Pearson Correlation.

If there is collinearity among the Independent features, the model tends to Overfit.

In [132…

```
plt.figure(figsize=(6,4))
cor = X_test.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.CMRmap_r)
plt.show()
```



Keeping our threshold collinearity at 80%, we can see from the above heatmap that:

'x6' and 'x2' are 98% collinear.

'x3' and 'x8' are 84% collinear.

Thus, we can drop 'x6' and 'x3', else our model would Overfit.

Feature Importance

The Multicollinearity shows that apart from x6 and x3, rest all teh features are important for calculations of our predictions. The array below shows that all the features are important, but we know that two features are collinear, so omitting them would not put any difference to our model.

In [138…
```
regressor.feature_importances_
```

Out[138]:
```
array([7.05290392e-04, 1.23996790e-04, 1.42842356e-03, 9.96235473e-01,
       8.21619252e-04, 3.08713377e-04, 1.08628646e-04, 2.67855475e-04])
```

Visualizing the feature Importance.

We can see that Xx6 and x2 are collinear.

Similarly x3 and x8 are collinear.

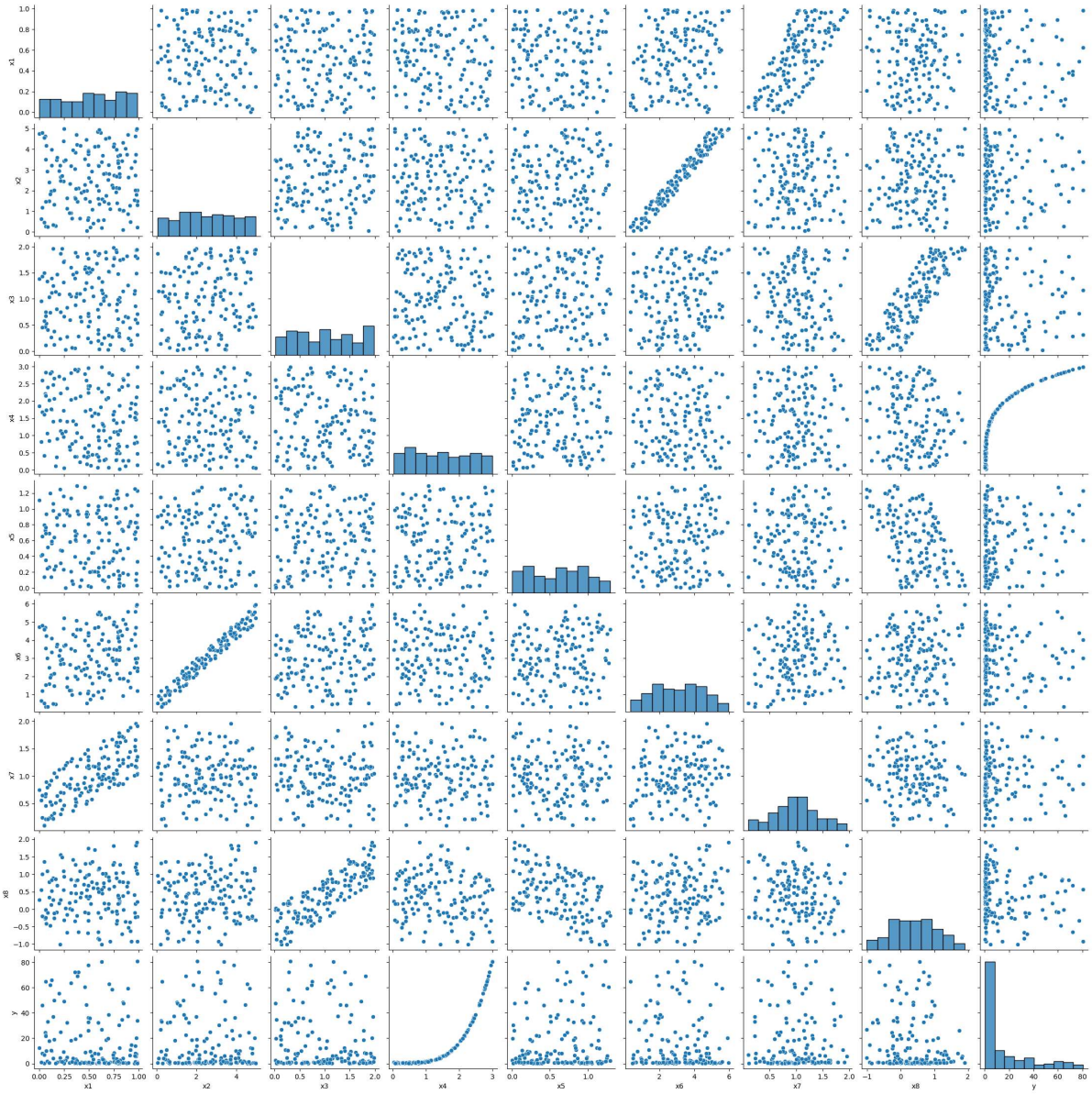
Thus x6 and x3 can be dropped.

Further x4 has maximum relation with output feature y. Thus it has most collinearity with y and which is very amazing for our predictions

The other features are equally important and we can see from the array above, of feature importance.

In [5]:
```
import seaborn as sns
sns.pairplot(df)
```

Out[5]:
```
<seaborn.axisgrid.PairGrid at 0x1e8aa3bf0a0>
```



Stability of our Findings.

Based on our analysis, we can deduce that our model is predicting well and it is stable for its predictions.

The value of MSE and R2 value also adds up to the stability of our model.

Discussion about Inferref Function f(x1,...,xm)

The inferred function f(x1, ..., xm) in our model represents is mathematical formula that maps the input features (x1,..,x8) to the target variable(Life_expectancy) allowing us to make accurate predictions for new data points.

Our Random Forests creates a set of Decision Trees and each tree makes prediction for each sample.

Finally the ultimate prediction is taken by majority vote of the mean of the majority of the outputs from the Decision Trees.

Time Complexity of the Model

The Time Complexity of a Random Forest depends upon the

Number of Trees

Depth of Each Tree

Number of features in the Dataset

This can be best assumed by estimating the time complexity of building a single decision tree, and number of trees in forest.

As this model is VERY EFFICIENT & SCALABLE, therefore it works well with larger datasets as well. As we have checked, our Random Forest contains

n_estimators = 9 max_depth = 5 random_state = 4

For n x m samples, ('n' samples & 'm' features), time complexity for a decision tree with max depth of 8 is O(n *m* log(n)). Now our model contains 8 trees, we can multiply the time complexity of one decision tree with the number of trees in our forest. Which gives us a value of O(8 *n* m * log(n)).The prediction of each tree is averaged and each tree is moved through to make a prediciton for each sample.

Now we have n = 150 (150 observations) and m = 7 (7 features, as we would drop 2 collinear features.)

Time complexity for one decision tree = O(n *m* log(n)) = O(150 *7* log(150)) = O(2198).

Time complexity for 9 trees(n_estimators = 9) = O(9 * 2198) = O(19782).

Time complexity for the predictions = O(9 * log(150)) = O(84).

Time Complexity Majorly Affected by :

Our time complexity is majorly affected by Data Preprocessing Steps.

The GridSearchCV took a lot of time to deduce the best parameters thus elaying in the model perforemnce.

Further, the Random Forests involve many decision trees, and working of all the decision trees takes time to implement, thus delaying out predictions

Summary

The Pipeline introduced here is for Random Forest Regressor.

We did initial analysis to get better understanding of our data.

The R2 value on the Unseen Data is 0.99.

Thats an Unbiased approach making our model fit for the predictions with Low Bias and Low Variance.

The Cross Validation helped us to decide the best parameters and their values for our model and thus our predictions.

Pros/Cons (Strengths / Limitations)

Pros

Random Forest Regressor works amazingly well with the unstandaized data.

There we do not need to standardize the data to implement this model.

The data was small thus it was easy to deduce the predictions.

Cons

Since the data was small, this moelmight not work well with the bigger dataset.

The model might Overfit with the bigger set of data as we aren't aware from where would we get the data and how it would be affected by outliers.

How can we imporve it

We can improve the working of this model by getting a bigger dataset.

Further the feature selection technique can also help us 9PCA), however for that we need a bigger dataset which can largely justify the importance of each feature.

In [ ]: