Introduction:-

The Garment Industry is one of the key examples of the industrial globalization of this modern era. It is a highly labour-intensive industry with lots of manual processes. Satisfying the huge global demand for garment products is mostly dependent on the production and delivery performance of the employees in the garment manufacturing companies. So, it is highly desirable among the decision makers in the garments industry to track, analyse and predict the productivity performance of the working teams in their factories.

The dataset which we are working with, includes important attributes of the garment manufacturing process and the productivity of the employees which had been collected manually and also been validated by the industry experts. Here we have to predict the productivity of the employees which is in the range of [0, 1]. Hence making the problem a regression type problem. So let's see how we can solve it.

Regression Problems

Formal Definition:

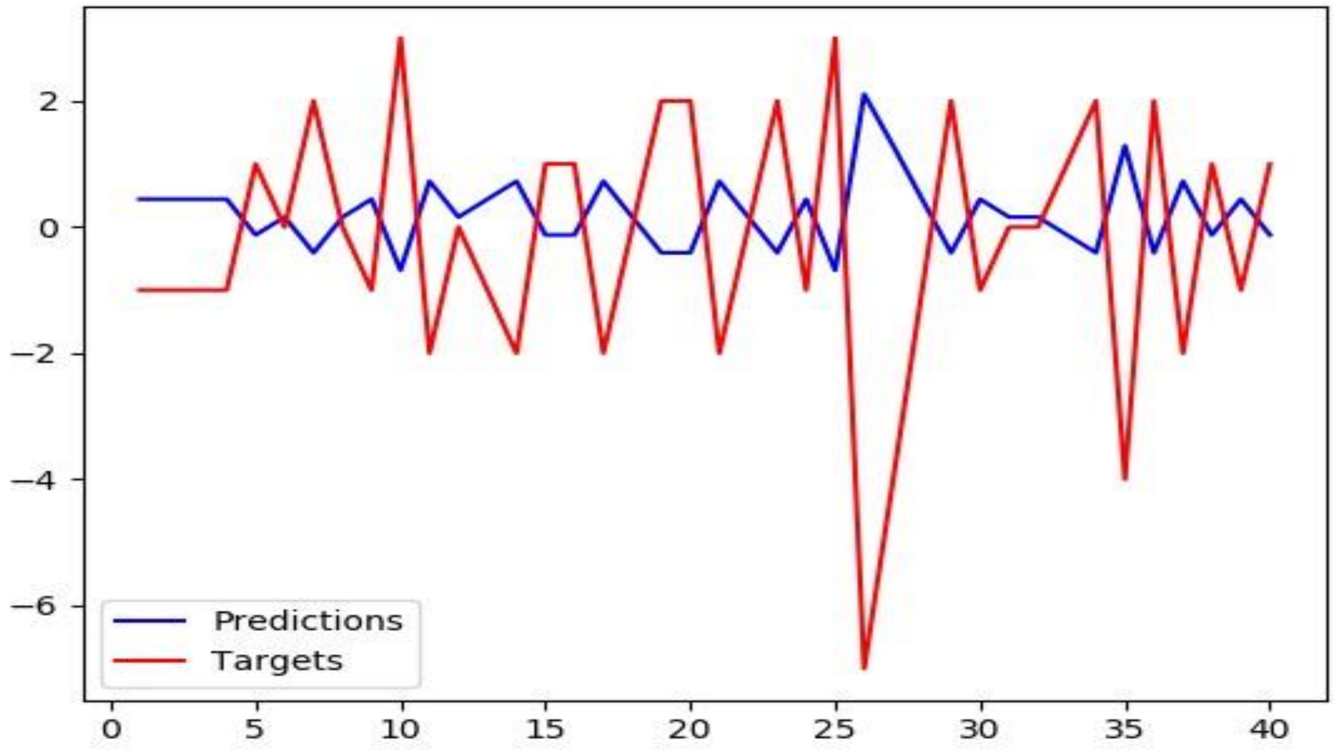
> Regression is a type of problem that use of machine learning algorithms to learn the continuous mapping function.

Taking the example shown in the above image, suppose we want our machine learning algorithm to predict the weather temperature for today. If we solved the above problem as a regression problem, the output would be continuous. It means our ML model will give exact temperature values, e.g., 24 °C, 24.5°C, etc.

To measure the learned mapping function's performance, we measure the prediction's closeness with the accurate labelled validation/test data. In the figure below, blue is the regression model's predicted values, and red is the actual labelled function. The blue line's closeness with the red line will give us a measure of

How good is our model?

how good is our model?



While building the regression model, we define our cost function. It measures the value of the learned values' deviation from the predicted values. Optimizers make sure that this error reduces over the progressive iterations, also called epochs.

Some of the most common error functions (or cost functions) used for regression problems are:

- **Mean Squared Error (MSE):**

$$MSE = \frac{\Sigma_i^N (Y_i - Y_i')^2}{N}$$

- **Root Mean Squared Deviation/Error (RMSD/RMSE):**

$$RMSE = \sqrt{\frac{\Sigma_i^N (Y_i - Y_i')^2}{N}}$$

- **Mean Absolute Error (MAE):**

$$MAE = \frac{\Sigma_i^N | Y_i - Y_i' |}{N}$$

Note: Yi is the predicted value, Yi' is the actual value, and N is the total samples over which prediction is made.

Some famous Examples of regression problems are:

- Predicting the house price based on the size of the house, availability of schools in the area, and other essential factors.
- Predicting the sales revenue of a company based on data such as the previous sales of the company.
- Predicting the temperature of any day based on data such as wind speed, humidity, atmospheric pressure.

Popular Algorithms Used for Regression Problems:

- Linear Regression
- Support Vector Regression ● Regression Tree

Importing Libraries:-
For completing any task we require tools, and we have plenty of tools in python. Let's start with importing the required libraries.

import pandas as pd import
seaborn as sns

#sns.set_theme(style="whitegrid") from tkinter
import messagebox, filedialog tips =
sns.load_dataset("tips")
from sklearn.preprocessing import PowerTransformer from sklearn import preprocessing from sklearn.model_selection import train_test_split from sklearn.preprocessing import StandardScaler from sklearn.linear_model import LinearRegression from sklearn.tree import DecisionTreeRegressor from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Dense import tensorflow as tf

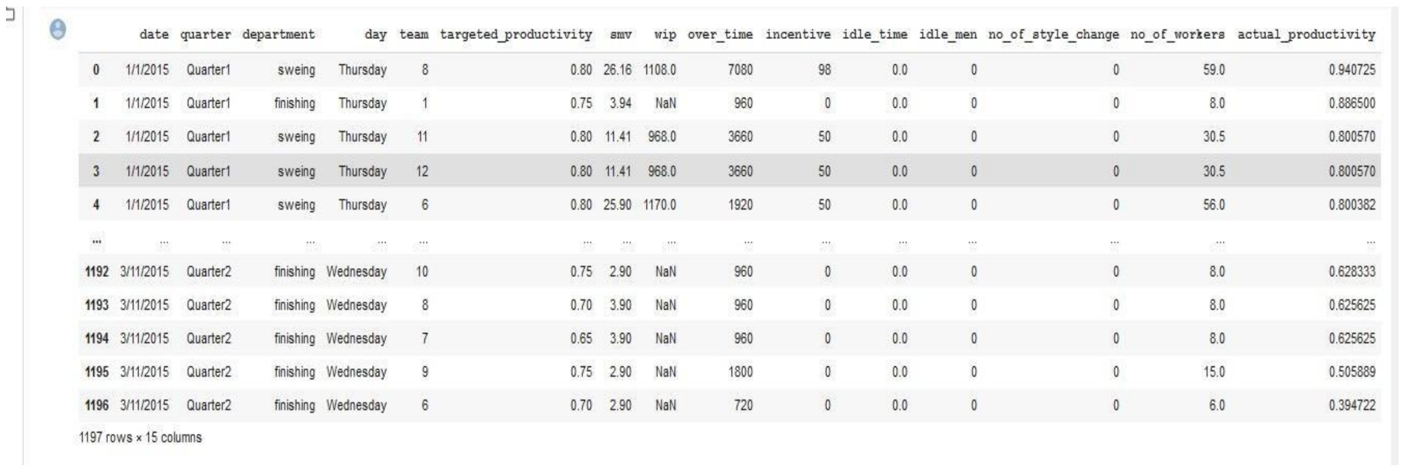
from tensorflow.keras import backend as K

Reading CSV:

The dataset we downloaded from kaggle is in csv (comma separated values) format. Hence we need to read the dataset first to work with it.
With help of this CSV file, we will try to understand the pattern and create our prediction model. We will do this with the panda's library.

# Upload the dataset

df= pd.read_csv ('/content/garments_worker_productivity.csv') #the path of the file is given df



	date	quarter	department	day	team	targeted_productivity	smv	wip	over_time	incentive	idle_time	idle_men	no_of_style_change	no_of_workers	actual_productivity
0	1/1/2015	Quarter1	sweing	Thursday	8	0.80	26.16	1108.0	7080	98	0.0	0	0	59.0	0.940725
1	1/1/2015	Quarter1	finishing	Thursday	1	0.75	3.94	NaN	960	0	0.0	0	0	8.0	0.886500
2	1/1/2015	Quarter1	sweing	Thursday	11	0.80	11.41	968.0	3660	50	0.0	0	0	30.5	0.800570
3	1/1/2015	Quarter1	sweing	Thursday	12	0.80	11.41	968.0	3660	50	0.0	0	0	30.5	0.800570
4	1/1/2015	Quarter1	sweing	Thursday	6	0.80	25.90	1170.0	1920	50	0.0	0	0	56.0	0.800382
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1192	3/11/2015	Quarter2	finishing	Wednesday	10	0.75	2.90	NaN	960	0	0.0	0	0	8.0	0.628333
1193	3/11/2015	Quarter2	finishing	Wednesday	8	0.70	3.90	NaN	960	0	0.0	0	0	8.0	0.625625
1194	3/11/2015	Quarter2	finishing	Wednesday	7	0.65	3.90	NaN	960	0	0.0	0	0	8.0	0.625625
1195	3/11/2015	Quarter2	finishing	Wednesday	9	0.75	2.90	NaN	1800	0	0.0	0	0	15.0	0.505889
1196	3/11/2015	Quarter2	finishing	Wednesday	6	0.70	2.90	NaN	720	0	0.0	0	0	6.0	0.394722

1197 rows × 15 columns

Data Analysis:-

Data analysis involves manipulating, transforming, and visualizing data in order to infer meaningful insights from the results. Individuals, businesses, and even governments often take direction based on these insights.

Data analysts might predict customer behaviour, stock prices, or insurance claims by using basic linear regression. They might create homogeneous clusters using classification and regression trees (CART), or they might gain some impact insight by using graphs to visualize a financial technology company's portfolio.

Until the final decades of the 20th century, human analysts were irreplaceable when it came to finding patterns in data. Today, they're still essential when it comes to feeding the right kind of data to learning algorithms and inferring meaning from algorithmic output, but machines can and do perform much of the analytical work itself.

Data pre-processing:-

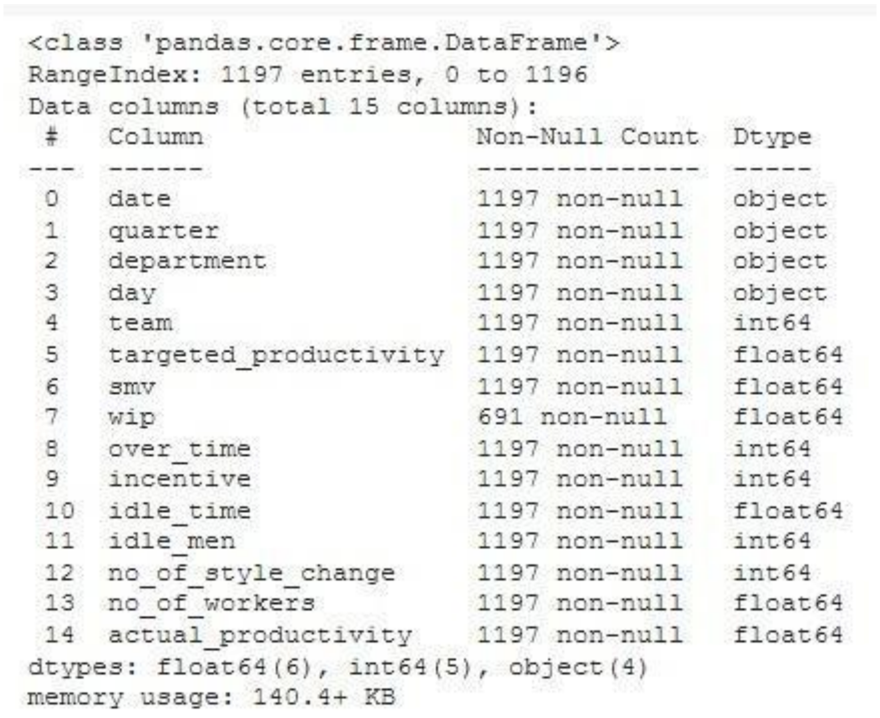
Data pre-processing in Machine Learning is a crucial step that helps enhance the quality of data to promote the extraction of meaningful insights from the data. Data pre-processing in Machine Learning refers to the technique of preparing (cleaning and organizing) the raw data to make it suitable for building and training Machine Learning models. In simple words, data pre-processing in Machine Learning is a data mining technique that transforms raw data into an understandable and readable format.

Data processing:-

Concept of Data processing is collecting and manipulating data into a usable and appropriate form. The automatic processing of data in a predetermined sequence of operations is the manipulation of data. The processing nowadays is automatically done by using computers, which is faster and gives accurate results.

Explain

```
# checking Info of the Current Datasheet
df.info ()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1197 entries, 0 to 1196
Data columns (total 15 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   date                   1197 non-null   object 
 1   quarter                1197 non-null   object 
 2   department             1197 non-null   object 
 3   day                    1197 non-null   object 
 4   team                   1197 non-null   int64  
 5   targeted_productivity  1197 non-null   float64
 6   smv                    1197 non-null   float64
 7   wip                    691 non-null    float64
 8   over_time              1197 non-null   int64  
 9   incentive              1197 non-null   int64  
 10  idle_time              1197 non-null   float64
 11  idle_men               1197 non-null   int64  
 12  no_of_style_change     1197 non-null   int64  
 13  no_of_workers          1197 non-null   float64
 14  actual_productivity    1197 non-null   float64
dtypes: float64(6), int64(5), object(4)
memory usage: 140.4+ KB
```

#from this info we understand that wip have NaN value.

Data Description:-

1. date : Date in MM-DD-YYYY
2. day : Day of the Week
3. quarter : A portion of the month. A month was divided into four quarters
4. department : Associated department with the instance
5. team : Associated team number with the instance
6. no_of_workers : Number of workers in each team
7. no_of_style_change : Number of changes in the style of a particular product
8. targeted_productivity : Targeted productivity set by the Authority for each team for each day.
9. smv : Standard Minute Value, it is the allocated time for a task
10. wip : Work in progress. Includes the number of unfinished items for products
11. over_time : Represents the amount of overtime by each team in minutes
12. incentive : Represents the amount of financial incentive (in BDT) that enables or motivates a particular course of action.
13. idle_time : The amount of time when the production was interrupted due to several reasons
14. idle_men : The number of workers who were idle due to production interruption
15. actual_productivity : The actual % of productivity that was delivered by the workers. It ranges from 0-1

# there are NaN in wip… # showing the feature that we have

```
df. column
```

```
Index(['date', 'quarter', 'department', 'day', 'team', 'targeted_productivity', 'smv', 'wip', 'over_time', 'incentive',
'idle_time', 'idle_men', 'no_of_style_change', 'no_of_workers', 'actual_productivity'], dtype='object')
```

```
df. shape
```

```
(1197, 15)
```

#1197 observations and 15 features are present in this dataset.

# Find how many NaN value in the dataset

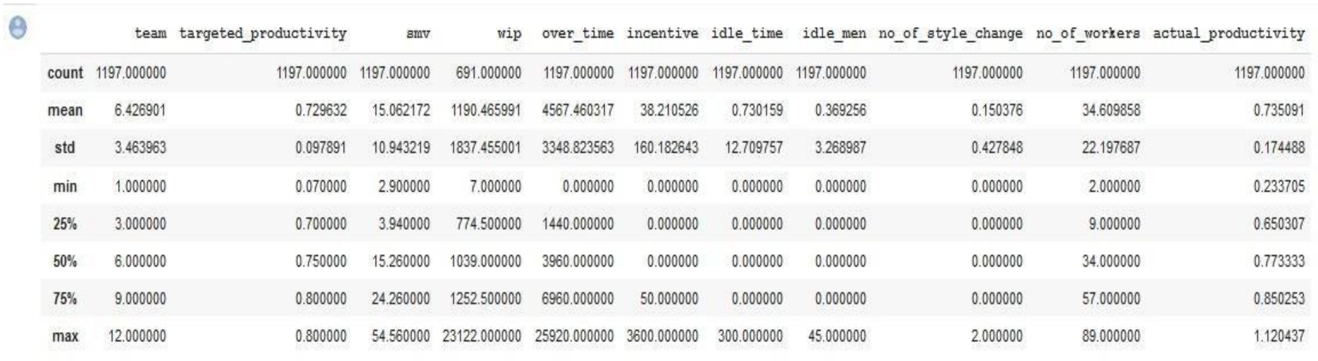
```
df.isna () .sum ()
```

```
date                    0
quarter                 0
department              0
day                     0
team                    0
targeted_productivity   0
smv                     0
wip                     506
over_time               0
```

```
incentive                    0
idle_time                    0
idle_men                     0
no_of_style_change           0
no_of_workers                0
actual_productivity          0
dtype: int64
```

From the above information we understand that wip has 506 NaN values.

```
# showing the features that we have
    df.describe ()
```



	team	targeted_productivity	smv	wip	over_time	incentive	idle_time	idle_men	no_of_style_change	no_of_workers	actual_productivity
count	1197.000000	1197.000000	1197.000000	691.000000	1197.000000	1197.000000	1197.000000	1197.000000	1197.000000	1197.000000	1197.000000
mean	6.426901	0.729632	15.062172	1190.465991	4567.460317	38.210526	0.730159	0.369256	0.150376	34.609858	0.735091
std	3.463963	0.097891	10.943219	1837.455001	3348.823563	160.182643	12.709757	3.268987	0.427848	22.197687	0.174488
min	1.000000	0.070000	2.900000	7.000000	0.000000	0.000000	0.000000	0.000000	0.000000	2.000000	0.233705
25%	3.000000	0.700000	3.940000	774.500000	1440.000000	0.000000	0.000000	0.000000	0.000000	9.000000	0.650307
50%	6.000000	0.750000	15.260000	1039.000000	3960.000000	0.000000	0.000000	0.000000	0.000000	34.000000	0.773333
75%	9.000000	0.800000	24.260000	1252.500000	6960.000000	50.000000	0.000000	0.000000	0.000000	57.000000	0.850253
max	12.000000	0.800000	54.560000	23122.000000	25920.000000	3600.000000	300.000000	45.000000	2.000000	89.000000	1.120437

```
#From this description we can know about mean, mode and median value of all features.
#here copy the dataset into df1 so that the main dataset is not lost during the pre-processing steps.
df1=df.copy ()

# Separate categorical data for simplicity in analysis
categorical_features = [feature for feature in df1.columns if((df1[feature].dtype=='O'))]
categorical_features
   ['quarter', 'department', 'day']
```

The feature is quarter and the number of categories are 5. The feature is department **and the number of categories are 2. The feature is day and the number of categories are 7.**

```
# fix department error
 df1.loc[:,'department'] = df1.loc[:,'department'].str.strip()
 df1  ['department']=  df1['department'].replace(['sewing'],['sewing'] )
```

#explain

from the raw dataset we got three categories (sewing, sewing, finishing) in department feature.

But there have to be two categories (sewing, finishing) in the department feature.

#here replace between two categories (sewing and sewing).

```
# Here we see whether NaN values sewing or not.
        f = 0 c = 0
        for i in sr:
           if i == 'sewing':
              p = float (st[c])
              if pd.isna(p):
                  f = f+1
           c = c+1
        f
0
```

```
# Here we count finishing...

      f = 0
      c = 0
      for i in sr:
          if i == 'finishing':
             p = float (st[c])
             if pd.isna(p):
                 f = f+1
          c = c+1
       f
506
```

```
# Fill NaN value with 0
```

Since all of the missing wip data is in the finishing department. we can see that the finishing department is waiting on work from the sewing department and assign 0 for missing wip data in the finishing department.

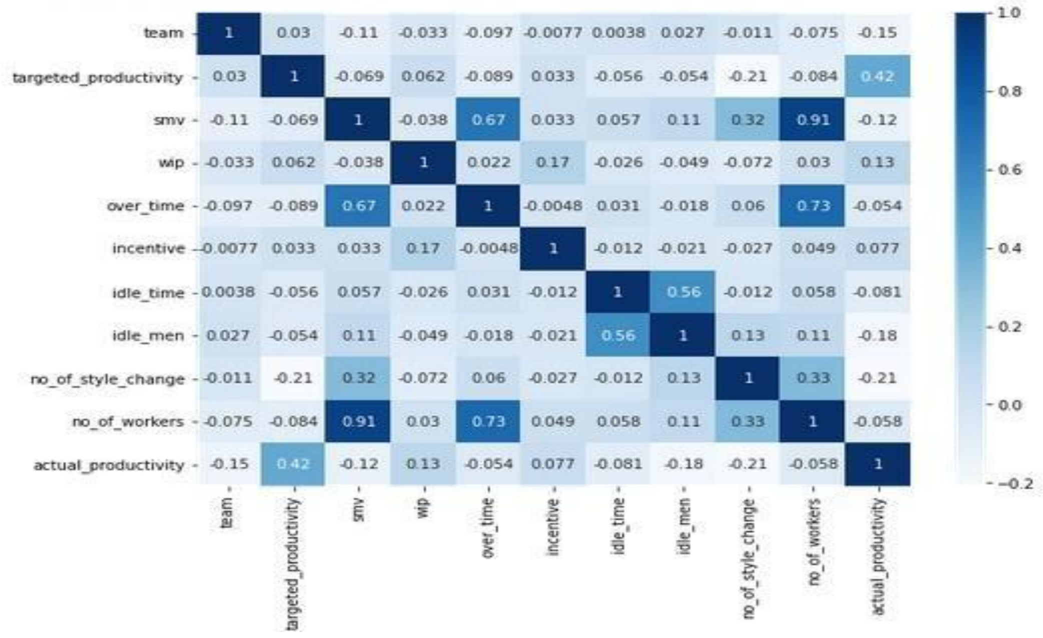
```
   df1 = df1.fillna (0)

   df1.info ()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1197 entries, 0 to 1196
Data columns (total 15 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   date                   1197 non-null   datetime64[ns]
 1   quarter                1197 non-null   object
 2   department             1197 non-null   object
 3   day                    1197 non-null   object
 4   team                   1197 non-null   int64
 5   targeted_productivity  1197 non-null   float64
 6   smv                    1197 non-null   float64
 7   wip                    1197 non-null   float64
 8   over_time              1197 non-null   int64
 9   incentive              1197 non-null   int64
 10  idle_time              1197 non-null   float64
 11  idle_men               1197 non-null   int64
 12  no_of_style_change     1197 non-null   int64
 13  no_of_workers          1197 non-null   float64
 14  actual_productivity    1197 non-null   float64
dtypes: datetime64[ns](1), float64(6), int64(5), object(3)
memory usage: 140.4+ KB
```

Here we can see that no NaN values are available in the Dataset.

```
# checking the correlation among the variables
plt.figure (figsize = (10,7))
sns.heatmap (df.corr(), annot = True, cmap = "Blues")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0305920090>
```

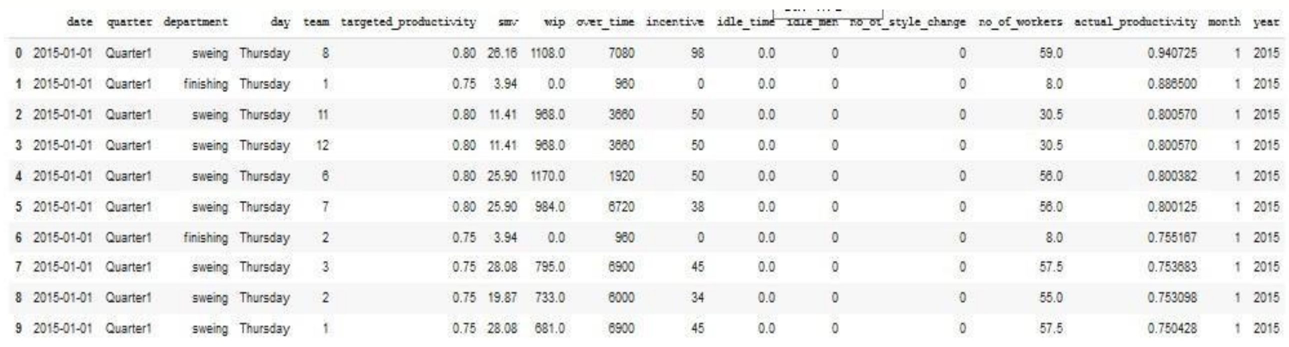


High positive correlation: number_of_workers and smv (0.91), no_of_workers and over_time (0.73), over_time and smv (0.67), idle_men and idle_time (0.56) Positive Correlation: number_of_workers and no_of_style_change (0.36), no_of_style_change and smv (0.32) There is not any obvious negative correlation between features.

Splitting the Month and Year

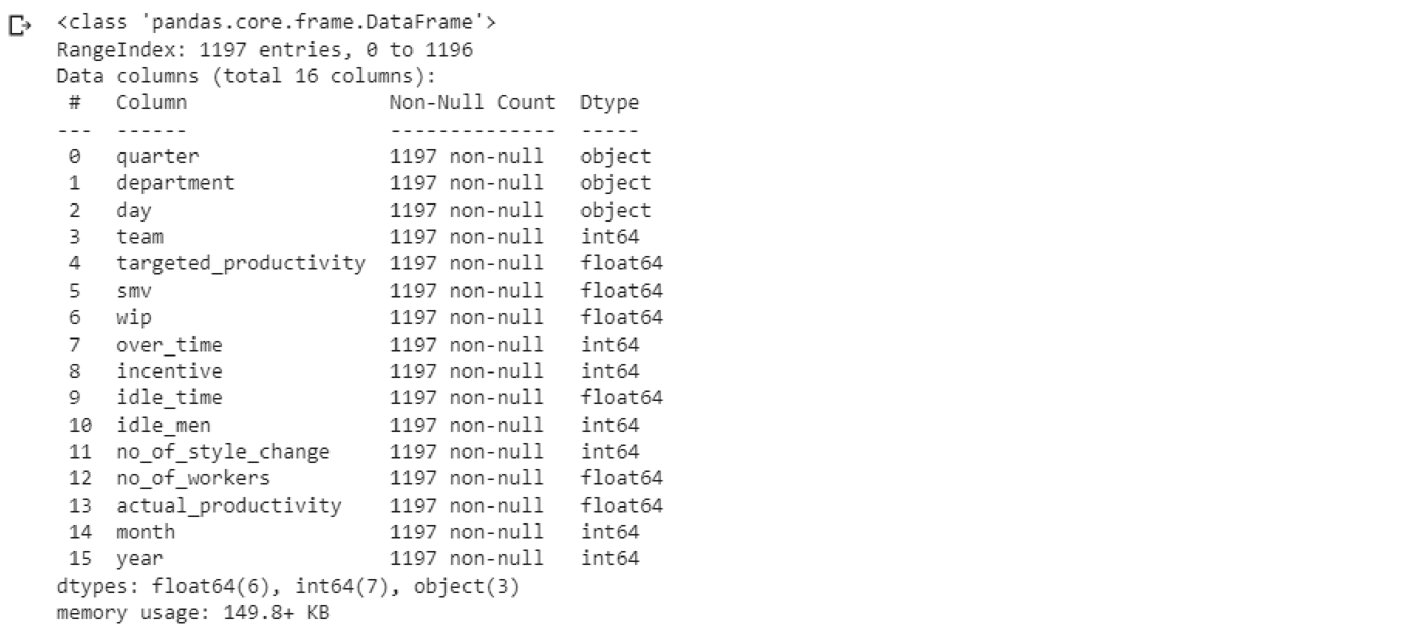
```
# Splitting month
def date2month(x):
  return x.month
```

```
# There we splitting year
def date2year(x):
  return x.year

df1['month']=df1['date'].apply(date2month)
df1.head(10)
```



	date	quarter	department	day	team	targeted_productivity	smv	wip	over_time	incentive	idle_time	idle_men	no_of_style_change	no_of_workers	actual_productivity	month	year
0	2015-01-01	Quarter1	sweing	Thursday	8	0.80	26.16	1108.0	7080	98	0.0	0	0	59.0	0.940725	1	2015
1	2015-01-01	Quarter1	finishing	Thursday	1	0.75	3.94	0.0	960	0	0.0	0	0	8.0	0.886500	1	2015
2	2015-01-01	Quarter1	sweing	Thursday	11	0.80	11.41	968.0	3660	50	0.0	0	0	30.5	0.800570	1	2015
3	2015-01-01	Quarter1	sweing	Thursday	12	0.80	11.41	968.0	3660	50	0.0	0	0	30.5	0.800570	1	2015
4	2015-01-01	Quarter1	sweing	Thursday	6	0.80	25.90	1170.0	1920	50	0.0	0	0	56.0	0.800382	1	2015
5	2015-01-01	Quarter1	sweing	Thursday	7	0.80	25.90	984.0	6720	38	0.0	0	0	56.0	0.800125	1	2015
6	2015-01-01	Quarter1	finishing	Thursday	2	0.75	3.94	0.0	960	0	0.0	0	0	8.0	0.755167	1	2015
7	2015-01-01	Quarter1	sweing	Thursday	3	0.75	28.08	795.0	6900	45	0.0	0	0	57.5	0.753683	1	2015
8	2015-01-01	Quarter1	sweing	Thursday	2	0.75	19.87	733.0	6000	34	0.0	0	0	55.0	0.753098	1	2015
9	2015-01-01	Quarter1	sweing	Thursday	1	0.75	28.08	681.0	6900	45	0.0	0	0	57.5	0.750428	1	2015

```
# Now we will remove the date column because we have extracted the useful information from this feature.

df1=df1.drop(columns='date')
df1.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1197 entries, 0 to 1196
Data columns (total 16 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   quarter                1197 non-null   object
 1   department             1197 non-null   object
 2   day                    1197 non-null   object
 3   team                   1197 non-null   int64
 4   targeted_productivity  1197 non-null   float64
 5   smv                    1197 non-null   float64
 6   wip                    1197 non-null   float64
 7   over_time              1197 non-null   int64
 8   incentive              1197 non-null   int64
 9   idle_time              1197 non-null   float64
 10  idle_men               1197 non-null   int64
 11  no_of_style_change     1197 non-null   int64
 12  no_of_workers          1197 non-null   float64
 13  actual_productivity    1197 non-null   float64
 14  month                  1197 non-null   int64
 15  year                   1197 non-null   int64
dtypes: float64(6), int64(7), object(3)
memory usage: 149.8+ KB
```

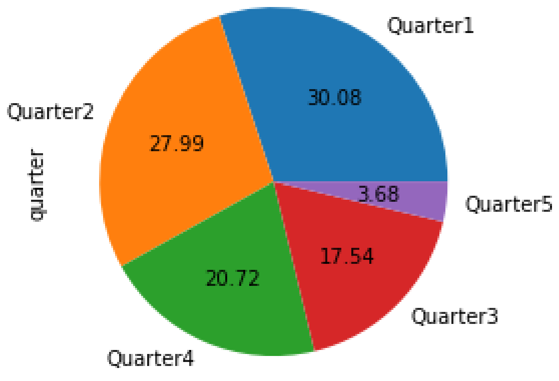
# now the date feature is not present in the dataset.

Visualisation Columns

# Categorical columns visualisation

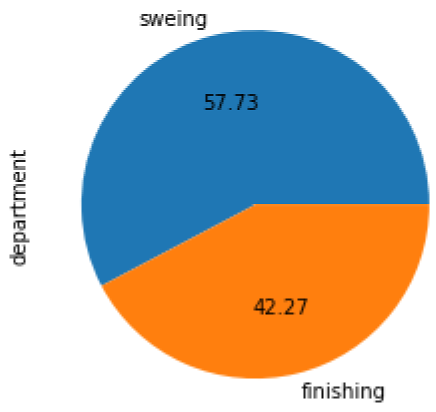
Using pie chart

```
# show quarter column using pie
df1['quarter'].value_counts().plot(kind='pie', autopct="%.2f")
plt.show()
```



From above plot we see that quarter 1 is 30.08%, quarter 2 is 27.72%, quarter 3 is 17.54%, quarter 4 is 20.72% and quarter 5 is 3.68%. From above information we can know that minimum work is done in quarter 5.

```
# show department column
df1['department'].value_counts().plot(kind='pie', autopct="%.2f")
plt.show()
```

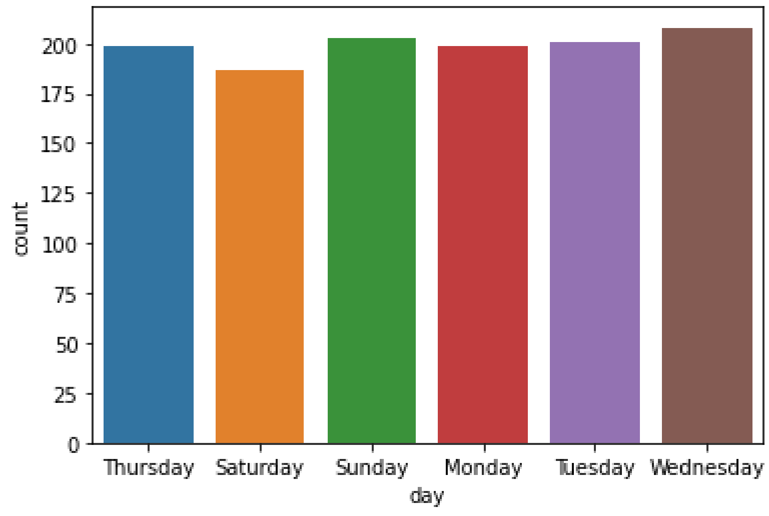


From this plot we see that sweing is 57.73% and finishing is 42.27% and difference between two departments is 15.46% from this information we can know that maximum works done by sweing features and minimum works done by finishing features.

Countplot

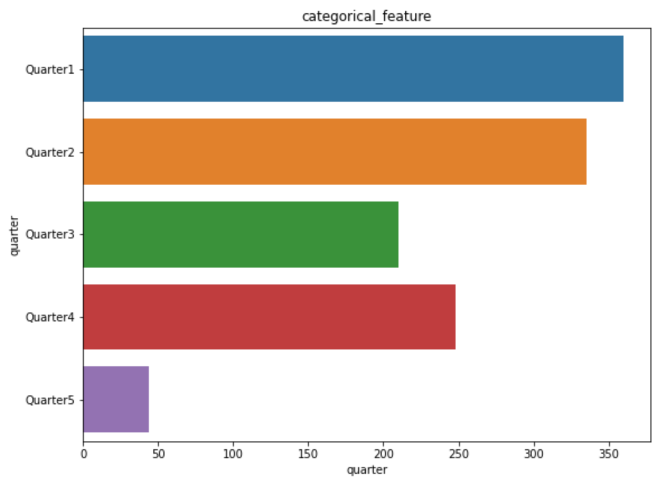
```
sns.countplot(x='day',data=df1)
```

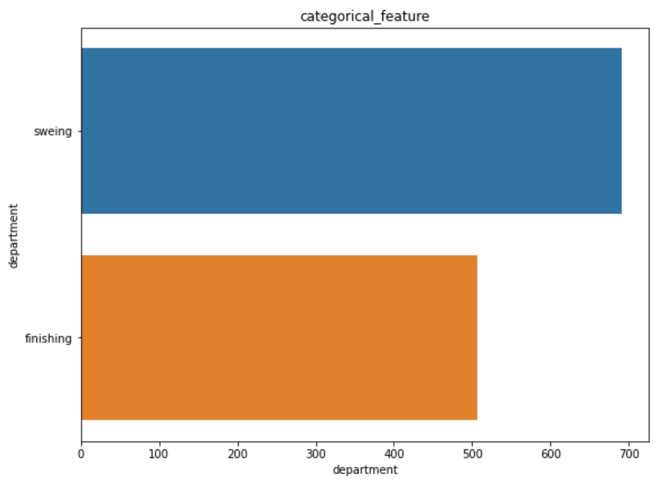
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd72bc110>
```

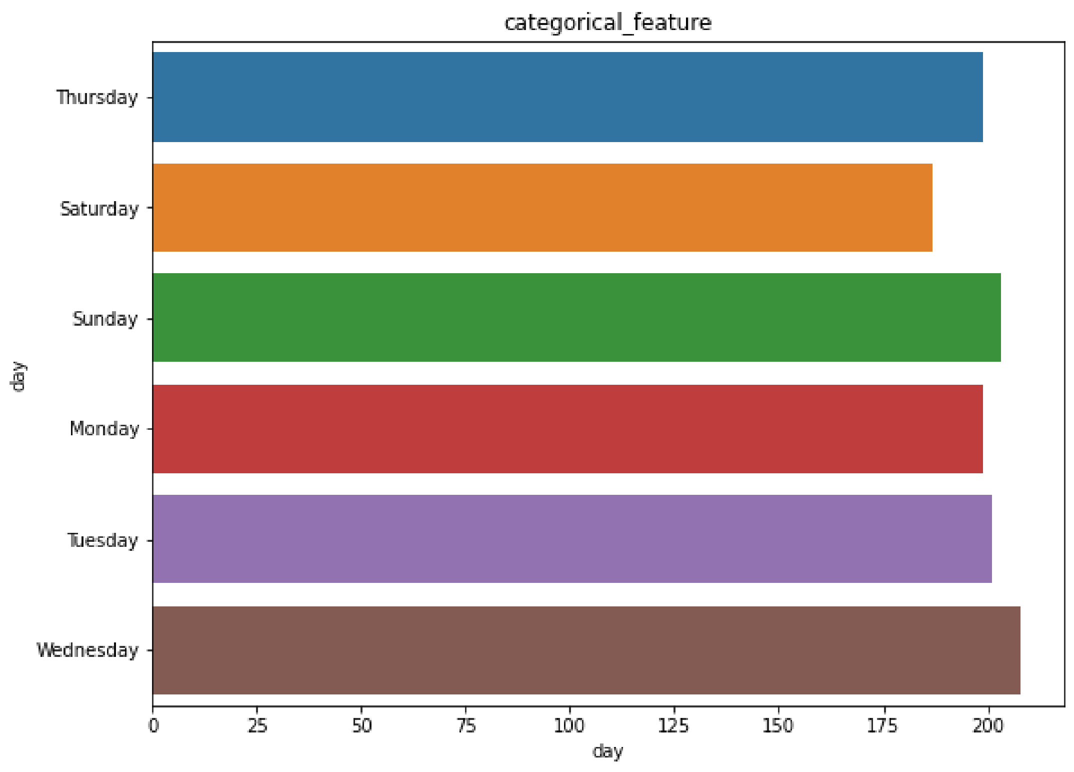


In this plot we see that Wednesday is more than 200 and another day less than equal to 200. Thursday, Sunday, Monday, Tuesday are almost same and Saturday is low of other day.

```
#check count based on categorical features
plt.figure(figsize=(20,80), facecolor='white')
plotnumber =1
for categorical_feature in categorical_features:
    ax = plt.subplot(10,2,plotnumber)
    sns.countplot(y=categorical_feature,data=df1)
    plt.xlabel(categorical_feature)
    plt.title('categorical_feature')
    plotnumber+=1
plt.show()
```







In this plot we saw the most work is done in quarter 1. Most of the work is still going on. In Wednesday most of the work is done. And there we visualization all categorical_features using Countplot.

What are categorical, discrete, and continuous variables?

Quantitative variables can be classified as discrete or continuous.

Categorical variable

Categorical variables contain a finite number of categories or distinct groups. Categorical data might not have a logical order. For example, categorical predictors include gender, material type, and payment method. **Discrete variable**

Discrete variables are numeric variables that have a countable number of values between any two values. A discrete variable is always numeric. For example, the number of customer complaints or the number of flaws or defects.

Continuous variable

Continuous variables are numeric variables that have an infinite number of values between any two values. A continuous variable can be numeric or date/time. For example, the length of a part or the date and time a payment is received.

Numerical Features

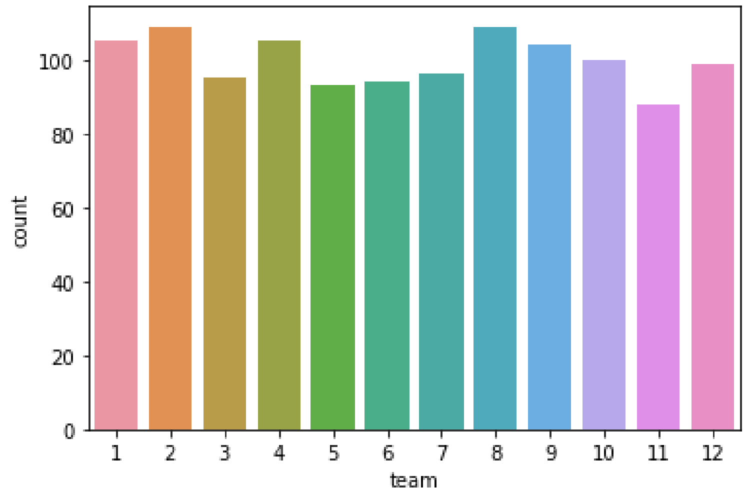
```
# Numerical data simplicity in analysis
numerical_features = [feature for feature in df1.columns if((df1[feature].dtypes!='O'))]
numerical_features
```

```
['team',
 'targeted_productivity',
 'smv',
 'wip',
 'over_time',
 'incentive',
 'idle_time',
 'idle_men',
 'no_of_style_change',
 'no_of_workers',
 'actual_productivity',
 'month',
 'year']
```

Using Count Plot

```
sns.countplot(x='team',data=df1)
```

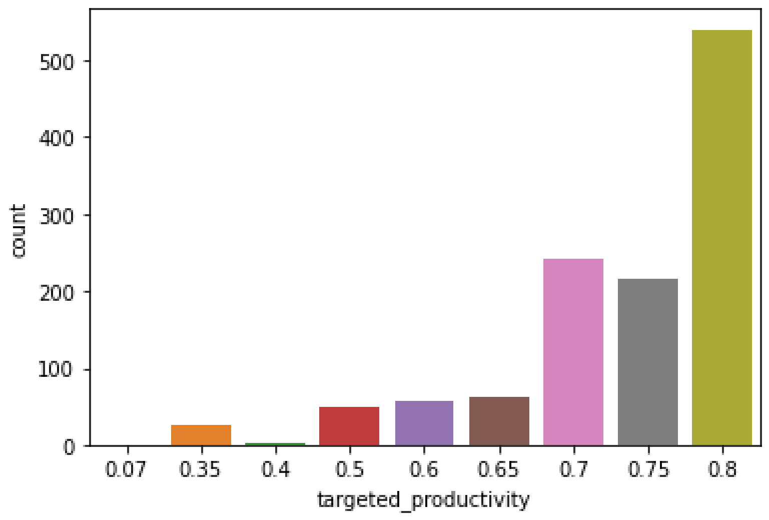
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd7013a10>
```



In this plot we see that 12 teams are present in this plot team1, team2, team4, team8 and team9 are greater than equal to 100 and team 2 and team8 are almost same team2 and team8 are high than other team. But difference among all team is very less. From this information we can know that maximum work is done by team2 and team8 than other team.

```
sns.countplot(x='targeted_productivity',data=df1)
```

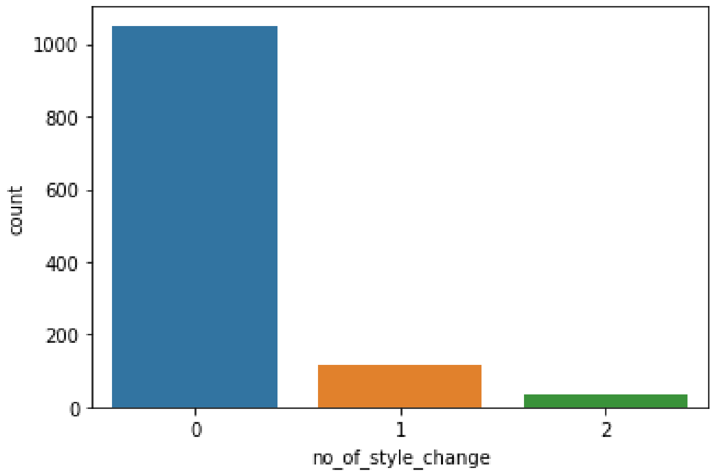
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd6fa8a50>
```



From this plot we see that 0.07 to 0.8 value are present in this plot. Here 0 to 0.65 value are very less and 0.7 to 0.8 value are high 0.8 value in very high than other values.

sns.countplot(x='no_of_style_change',data=df1)

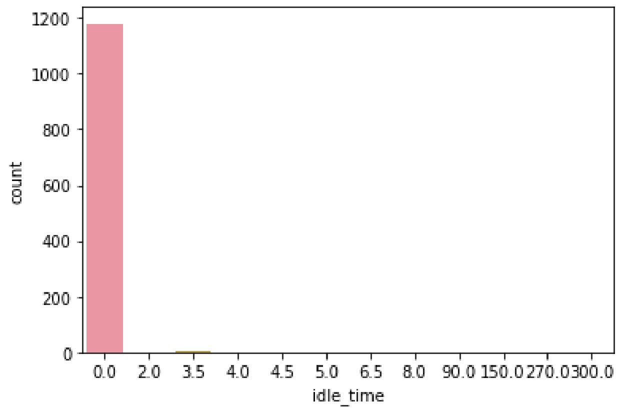
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4d29a50>
```



In this plot 0, 1 and 2 value are present in this plot 0 value is very high and 1, 2 values is very low. It means that maximum 0 value is present in this dataset.

sns.countplot(x='idle_time',data=df1)

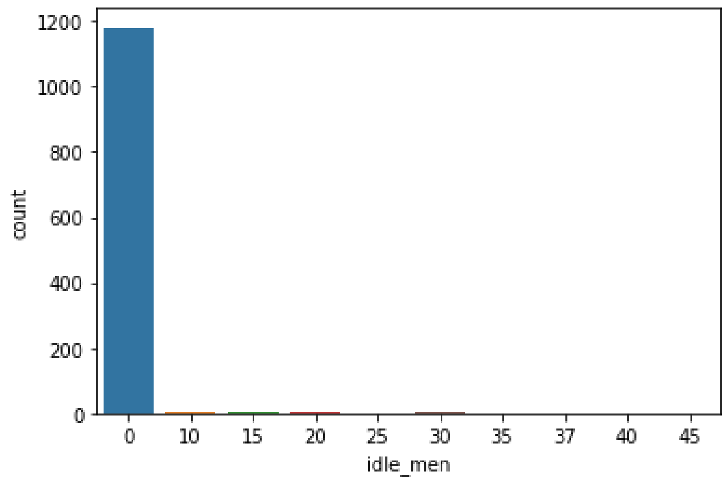
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4e20990>
```



In this plot 0.0 to 300.0 column are present. But every value is present in 0.0 column and little bit value is present in 3.5 column.

sns.countplot(x='idle_men',data=df1)

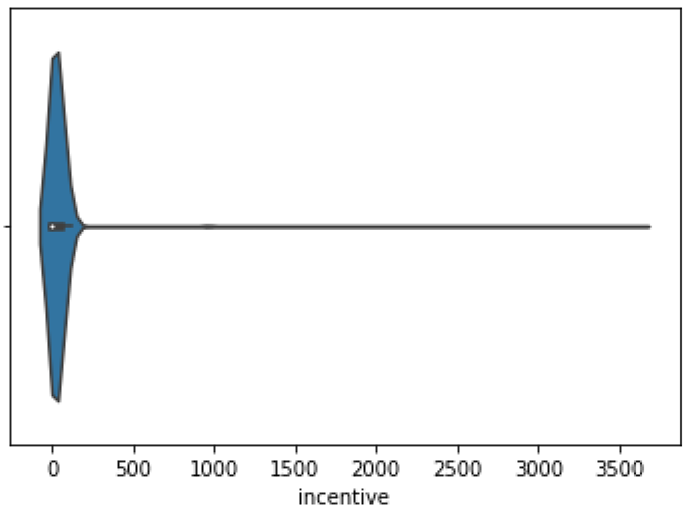
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4ece510>
```



There we use Countplot in idle_men column. There we see only 0 value is present and 10, 15, 20 and 30 there present some value.

Using Violinplot

sns.violinplot(x='incentive',data=df1)
plt.show()

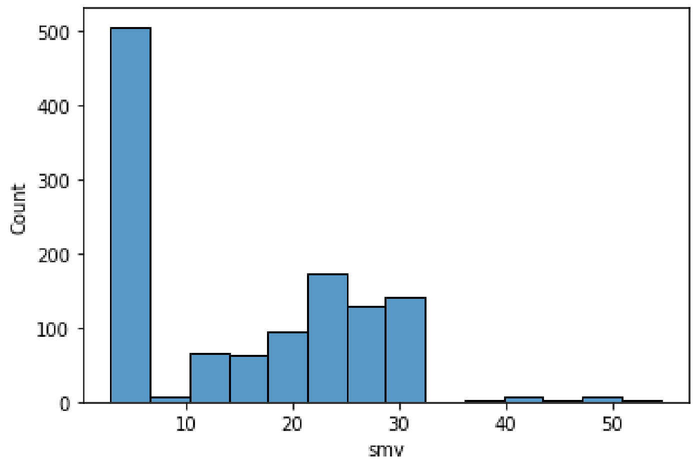


From this plot we see that 0 to 3500 incentive value are present in this features. 500 to 3500 incentive value is very low and o value is very high.

Using Histplot

sns.histplot(x='smv',data=df1)

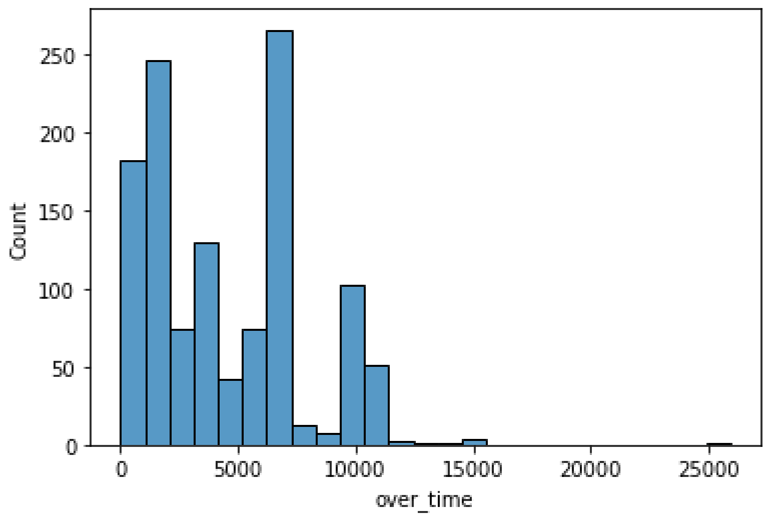
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4c2e890>
```



From above plot we see that 0 to 50 value present in this feature. Here 0 to 30 value are high and 30 to 50 value in very low. 0-10 value are very high than other value. From this information we can know that in SMV 0 to 10 value so much present.

sns.histplot(x='over_time',data=df1)

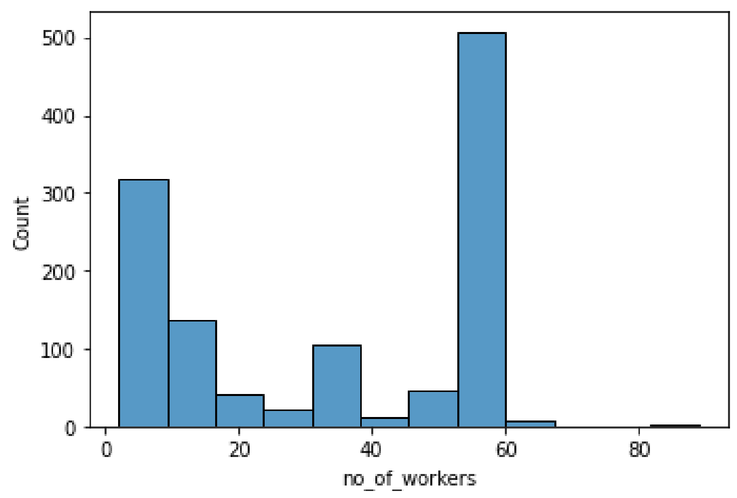
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd6fa2f10>
```



From this plot we see that 0 to 25000 value are present in this feature. Here 0 to 10000 value is high and 15000 and 25000 value are low. From this information we can understand in over_time o to 10000 value so much present.

sns.histplot(x='no_of_workers',data=df1)

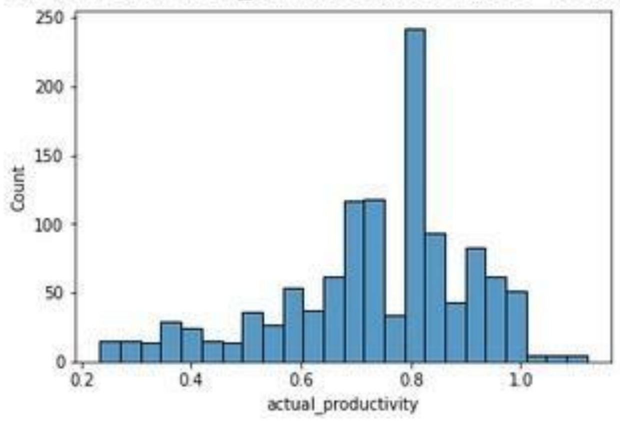
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4b8e350>
```



From this plot we see that 0 to 80 value are present. In this features near 60 value is very high than other value. Near 70 value are empty. 0 to 40 value are low then neat 60 value. From this information we understand that in no_of_workers 0 to 60 values are present.

sns.histplot(x='actual_productivity',data=df1)

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0304e51cd0>
```

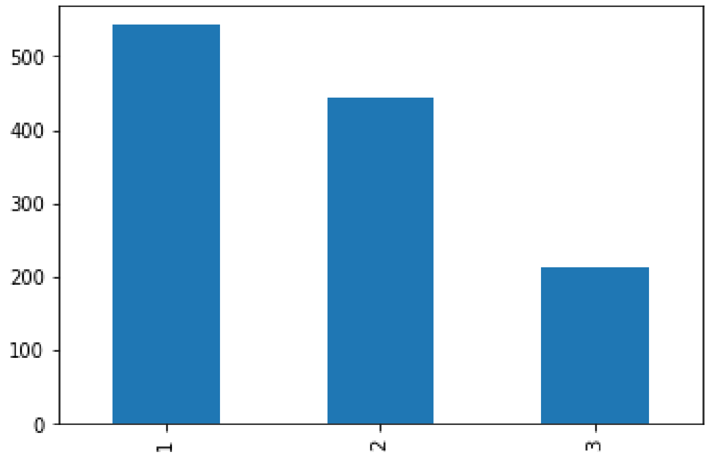


In this plot we see that 0.2 – 1.0 value are present in this feature. 0.8 Value is very high than other value. From this information we understan that near 1.0 value so mouch present.

Using bar plot

df1.month.value_counts().plot(kind='bar')

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4ac8650>
```

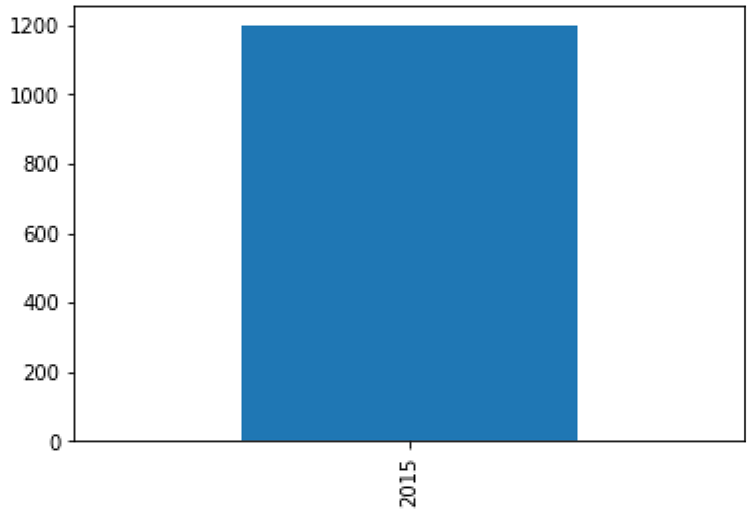


This plot we see that here 3 value are present 1 mean January, 2 mean February and 3 mean March. 1 is very high it touch 500 and 2 is medium it touch 400 and 3 it is low more than outer value its touch 200. Form this information we understand that maximum work is done in January and minimum work is done in March.
df1['month'].value_counts()

```
1    542
2    443
3    212
Name: month, dtype: int64
```

df1.year.value_counts().plot(kind='bar')

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd4a288d0>
```



In this plot we have only one vale in this features 2015. All of work are done in this year.

df1['year'].value_counts()

```
2015    1197
Name: year, dtype: int64
```

What is Skew-Symmetric Data?

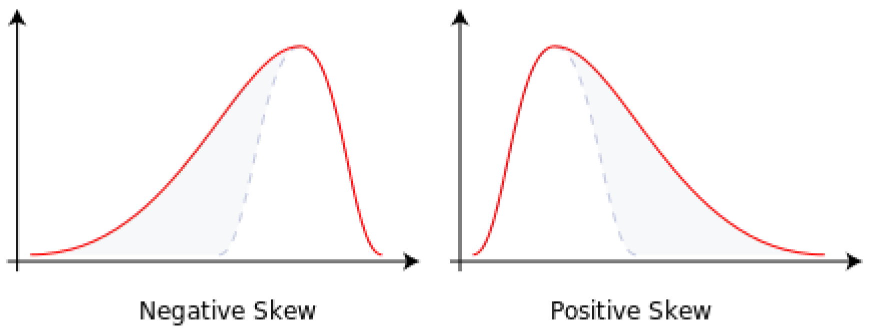
Skew-Symmetric data, on the other hand, may have skewness or noise such that the data appears at irregular or haphazard intervals

.

Skew-Symmetric are two types:

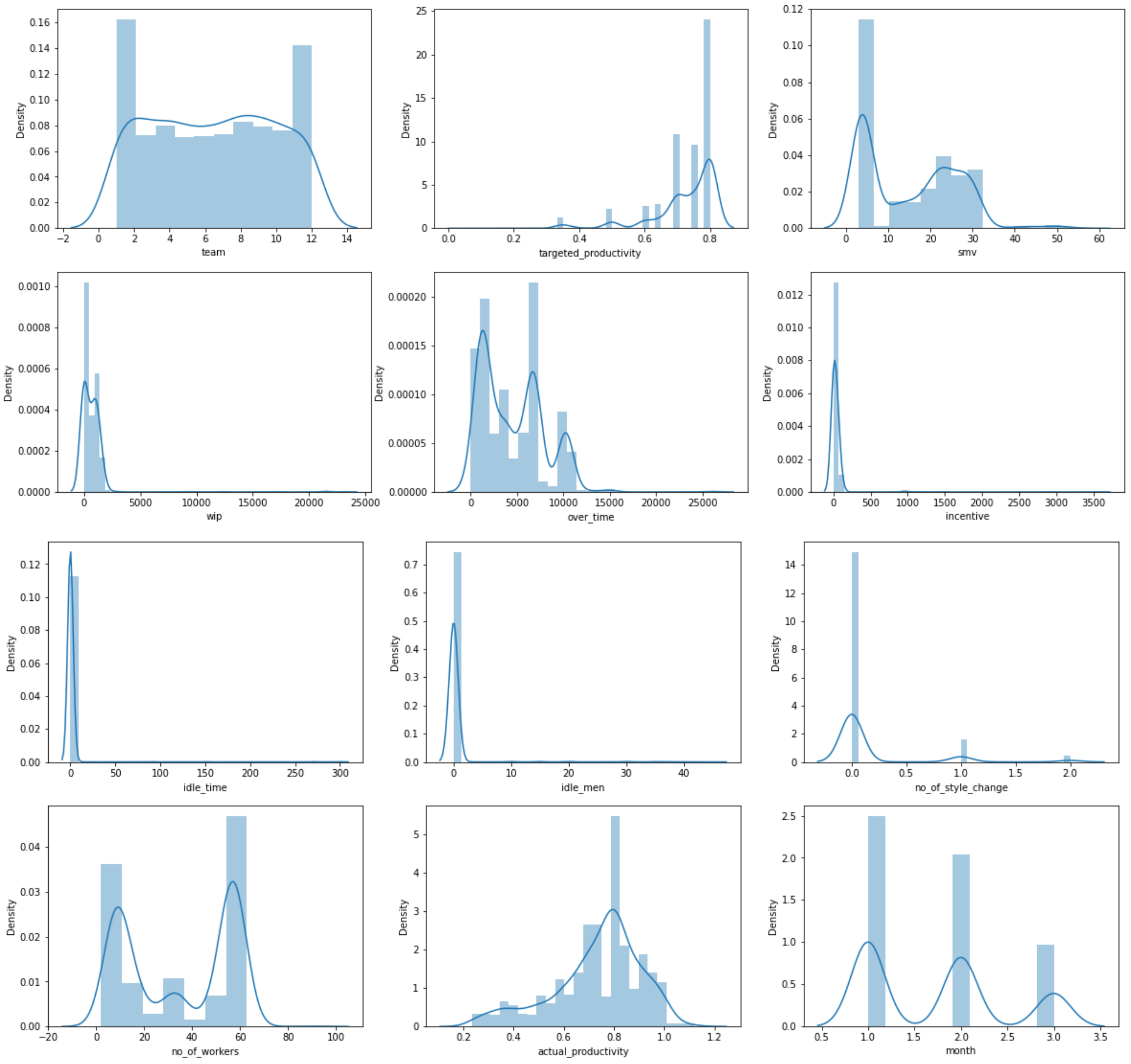
1. Negative skew: -The left tail is longer; the mass of the distribution is concentrated on the right of the figure. The distribution is said to be *left-skewed*, *left-tailed*, or *skewed to the left*, despite the fact that the curve itself appears to be skewed or leaning to the right; *left* instead refers to the left tail being drawn out and, often, the mean being skewed to the left of a typical centre of the data. A left-skewed distribution usually appears as a *right-leaning* curve.

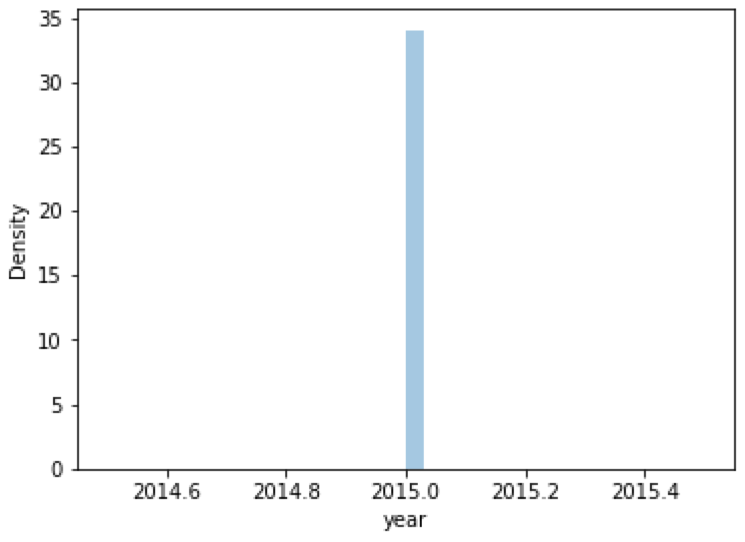
2. Positive skew: -The right tail is longer; the mass of the distribution is concentrated on the left of the figure. The distribution is said to be *right-skewed*, *right-tailed*, or *skewed to the right*, despite the fact that the curve itself appears to be skewed or leaning to the left; *right* instead refers to the right tail being drawn out and, often, the mean being skewed to the right of a typical centre of the data. A right-skewed distribution usually appears as a *left-leaning* curve.



Now we visualize whether our feature is Symmetric or Skew-Symmetric show all numerical_features in distplot.

```
# show all numerical_features in distplot

plt.figure(figsize=(20,60), facecolor='white')
plotnumber =1
for numerical_features in numerical_features:
    ax = plt.subplot(12,3,plotnumber)
    sns.distplot(df1[numerical_features])
    plt.xlabel(numerical_features)
    plotnumber+=1
plt.show()
```





In those graph we are understand that actual_productivity, **incentive, over_time, smv, targeted_productivity, idle_men and idle_time are** skewness.

Why do you want to transform skewness? You can clearly see that actual_productivity, incentive, over_time, smv, targeted_productivity, idle_men and idle_time are skewness. Now, let's say you want to use those as a feature for the model which will predict the Garment_Worker_productivity.

Here we have **targeted_productivity feature from all those graphs**, it means that it has a higher number of data points having high values. So when we train our model on this data, it will perform better at predicting the Garment_Worker_productivity with high targeted_productivity as compared to those with lower value.

Also, skewness tells us about the direction of outliers. You can see that our distribution is ***negative skew*** skewed and most of the outliers are present on the right side of the distribution.

Top 3 methods to handling Skew-Symmetric data:

Log Transform:

Log transformation is most likely the first thing you should do to remove skewness from the predictor. It can be easily done via Numpy, just by calling the log() function on the desired column

Square Root Transform:

The square root sometimes works great and sometimes isn't the best suitable option. In this case, I still expect the transformed distribution to look somewhat exponential, but just due to taking a square root the range of the variable will be smaller.

You can apply a square root transformation via Numpy, by calling the sqrt() function.

Box-Cox Transform:

This is the last transformation method I want to explore today. As I don't want to drill down into the math behind, You should only know that it is just another way of handling skewed data. To use it, your data must be positive — so that can be a bummer sometimes. Here also we use **yeo-johnson.**

Yeo-Johnson Transform

The Yeo-Johnson transform is also named for the authors.

Unlike the Box-Cox transform, it does not require the values for each input variable to be strictly positive. It supports zero values and negative values. This means we can apply it to our dataset without scaling it first.

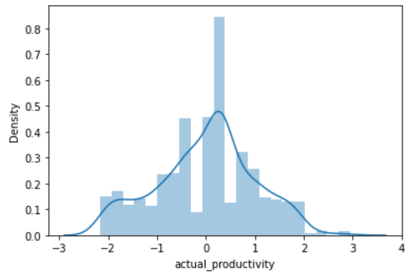
We can apply the transform by defining a power Transform object and setting the "method" argument to "yeo-johnson" (the default).

There we covert all skew-symmetric data

```
# Convert actual_productivity

from sklearn.preprocessing import PowerTransformer
```

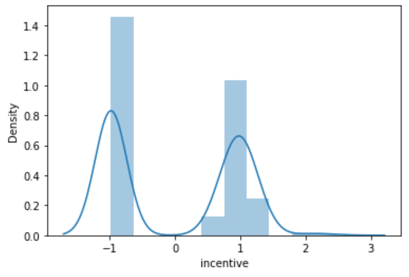
```
p = PowerTransformer(method='box-cox')
df1['actual_productivity']=p.fit_transform(df1[['actual_productivity']])
sns.distplot(df1.actual_productivity)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd8c56950>
```



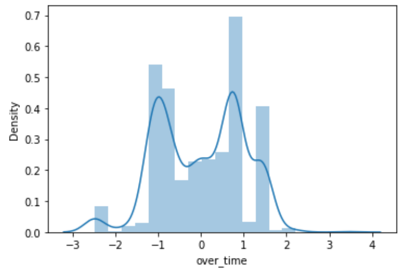
```
# convert incentive using yeo-johnson
p = PowerTransformer(method='yeo-johnson')
df1['incentive']=p.fit_transform(df1[['incentive']])
sns.distplot(df1.incentive)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd3f71510>
```



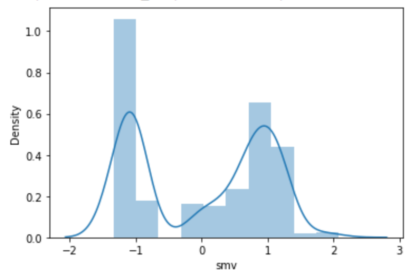
```
# convert over_time
p = PowerTransformer(method='yeo-johnson')
df1['over_time']=p.fit_transform(df1[['over_time']])
sns.distplot(df1.over_time)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd3eefa50>
```



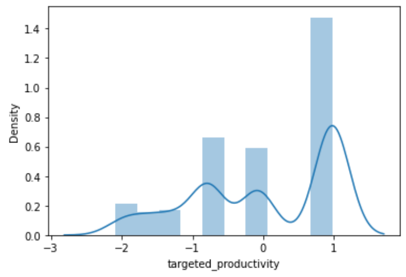
```
# convert over_time
from sklearn.preprocessing import PowerTransformer
p = PowerTransformer(method='yeo-johnson')
df1['smv']=p.fit_transform(df1[['smv']])
sns.distplot(df1.smv)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd3cc53d0>
```



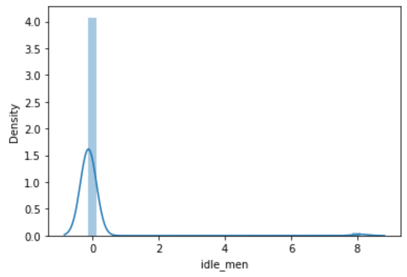
```
# convert targeted_productivity
p = PowerTransformer(method='yeo-johnson')
df1['targeted_productivity']=p.fit_transform(df1[['targeted_productivity']])
sns.distplot(df1.targeted_productivity)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd3b26e10>
```



```
# convert idle_men
p = PowerTransformer(method='yeo-johnson')
df1['idle_men']=p.fit_transform(df1[['idle_men']])
sns.distplot(df1.idle_men)
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please ad
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fd39f79d0>
```

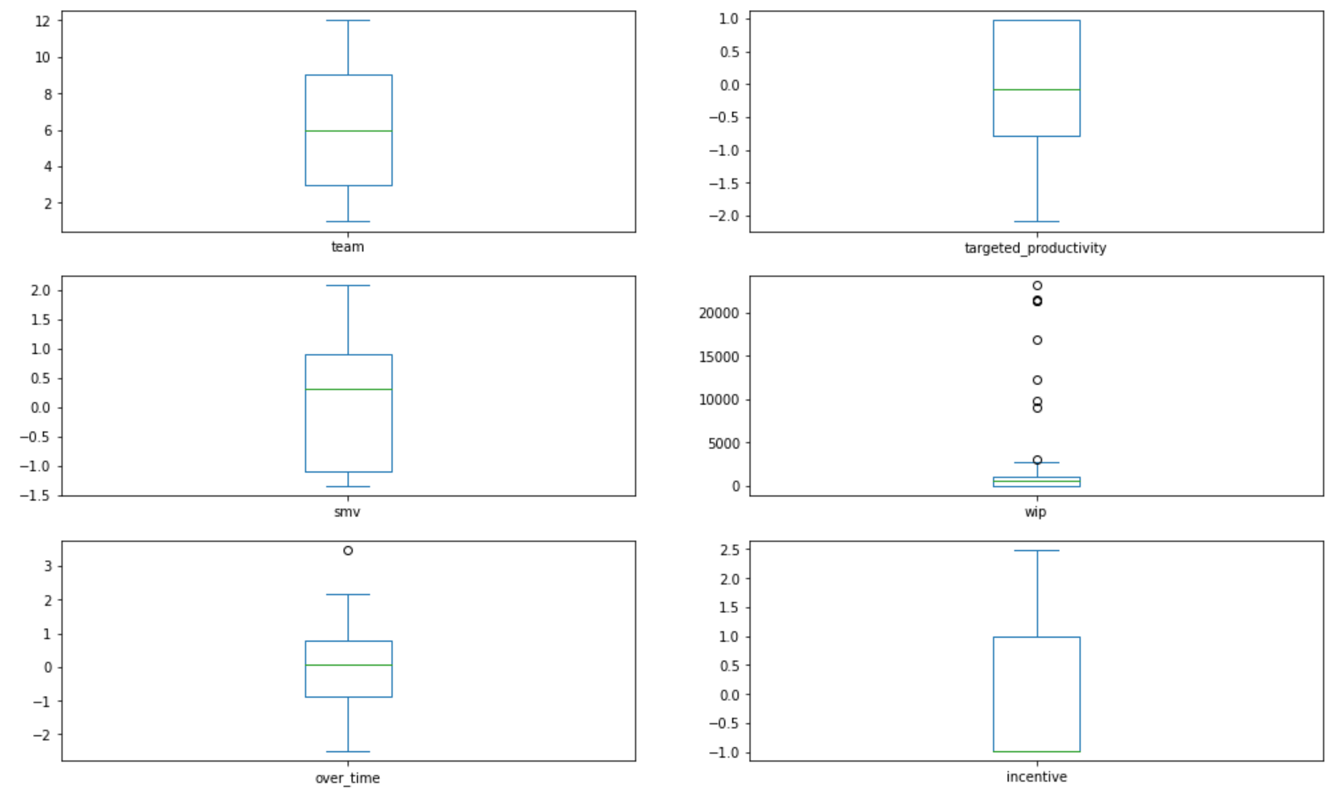


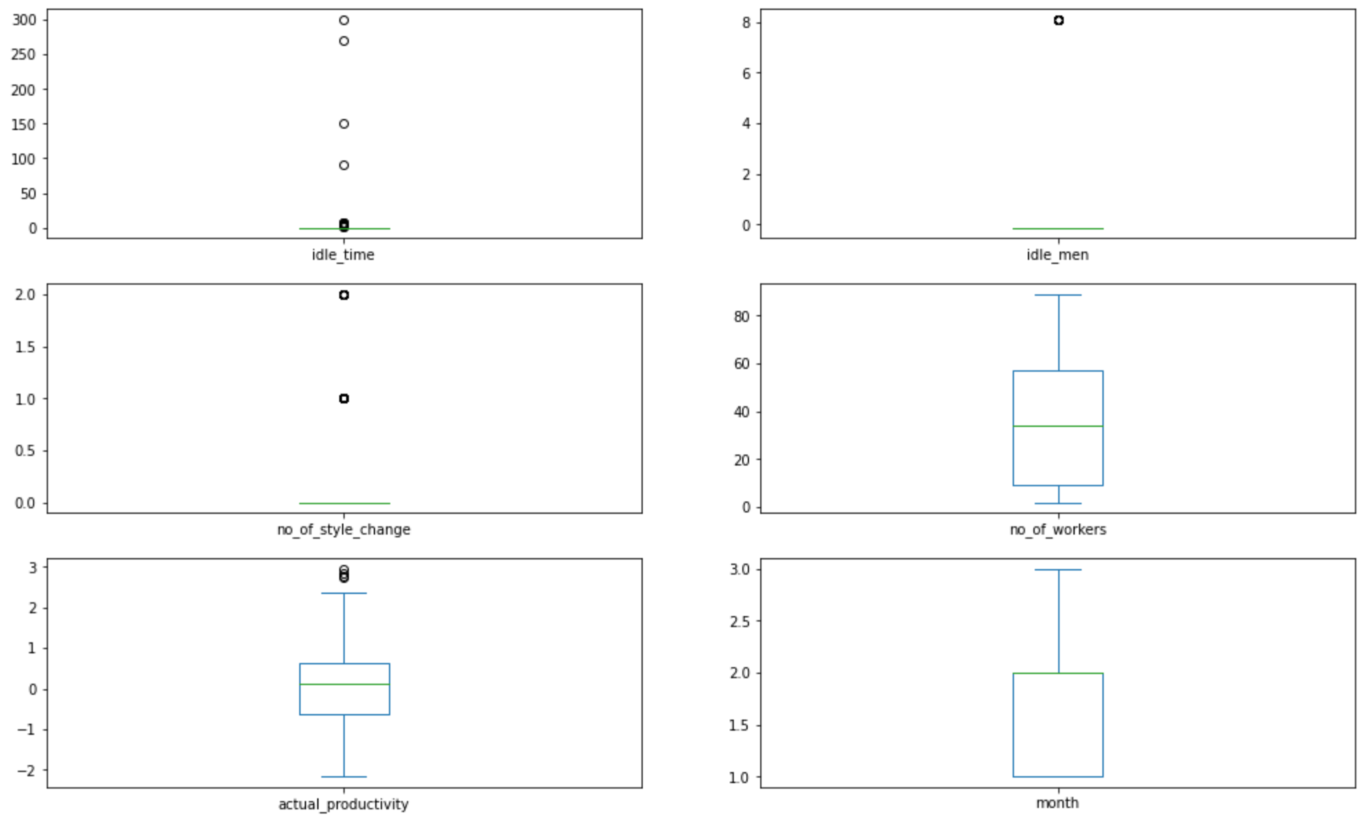
What is outlier?

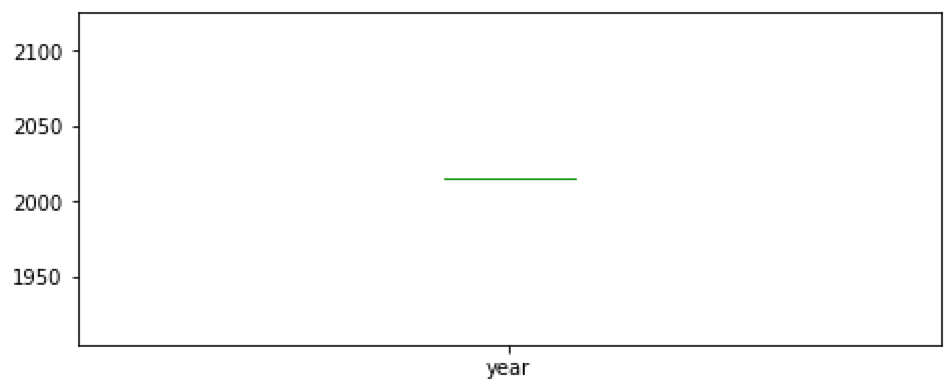
An outlier is a single data point that goes far outside the average value of a group of statistics. Outliers may be exceptions that stand outside individual samples of populations as well. In a more general context, an outlier is an individual that is markedly different from the norm in some respect.

Outliers are an important factor in statistics as they can have a considerable effect on overall results. In especially small sample sizes, a single outlier may dramatically affect averages and skew the study's final results.

```
# Chack outlier data...
df1.plot(kind="box",subplots=True,layout=(7,2),figsize=(17,25));
```







From the above diagrams you can see, the dots standing above the lines are outliers. Hence wip, over_time, actual_productivity, on_of_style_change, ideal_man and ideal_time are having outliers.

Why remove them?
Often outliers are discarded because of their effect on the total distribution and statistical analysis of the dataset. This is certainly a good approach if the outliers are due to an error of some kind (measurement error, data corruption, etc.), however often the source of the outliers is unclear. There are many situations where occasional 'extreme' events cause an outlier that is outside the usual distribution of the dataset but is a valid measurement and not due to an error. In these situations, the choice of how to deal with the outliers is not necessarily clear and the choice has a significant impact on the results of any statistical analysis done on the dataset. The decision about how to deal with outliers depends on the goals and context of the research and should be detailed in any explanation about the methodology.

Remove Outlier Data

there we delete all outliers data in our data set.

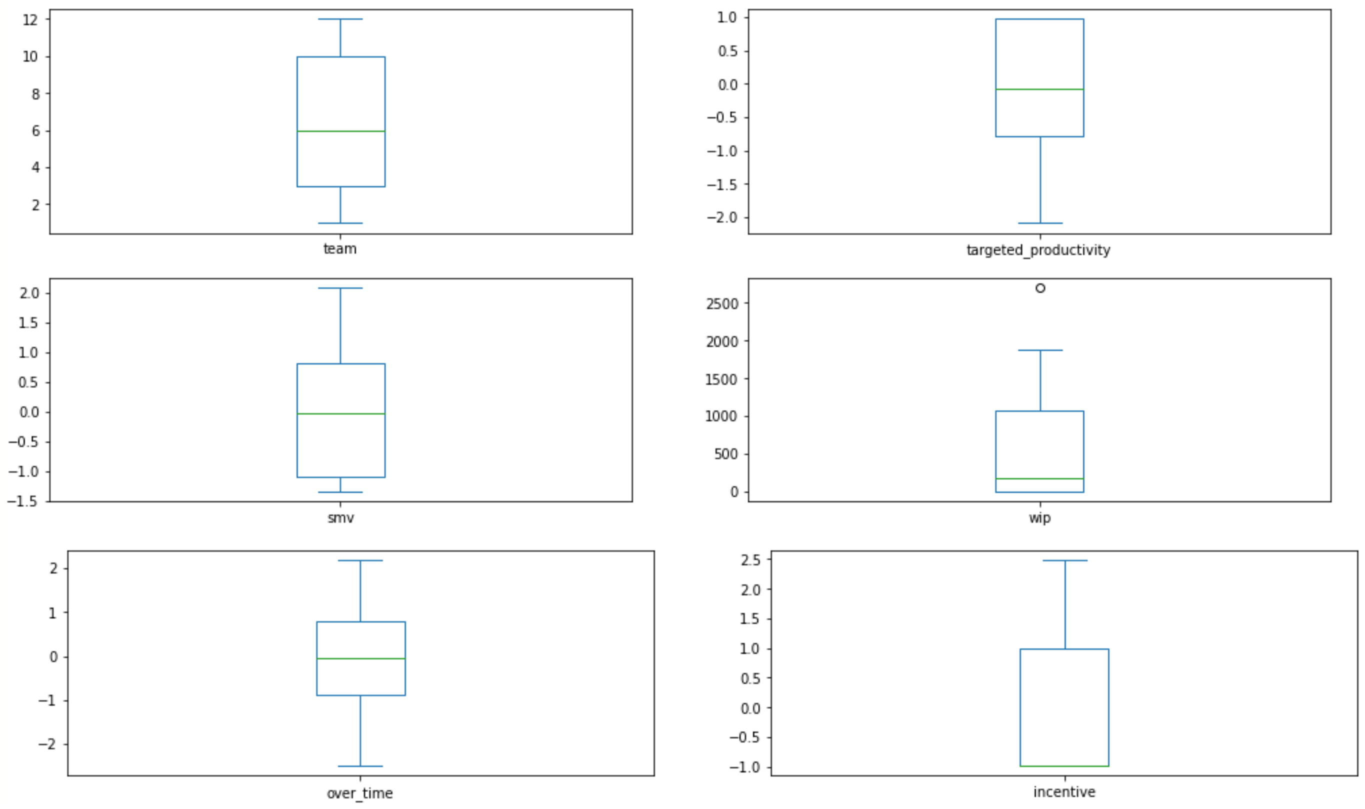
```
def outliers(df1, ft):
  Q1 = df1[ft].quantile(0.25)
  Q3 = df1[ft].quantile(0.75)
  IQR = Q3 - Q1

  lower_bound = Q1 - 1.5 * IQR
  upper_bound = Q3 + 1.5 * IQR

  ls = df1.index[(df1[ft]< lower_bound) | (df1[ft]> upper_bound)]
  return ls

index_list = []
for feature in ['targeted_productivity', 'wip', 'incentive', 'idle_men', 'no_of_style_change', 'idle_time', 'over_time', 'actual_productivity','smv']:
  index_list.extend(outliers(df1, feature))
```

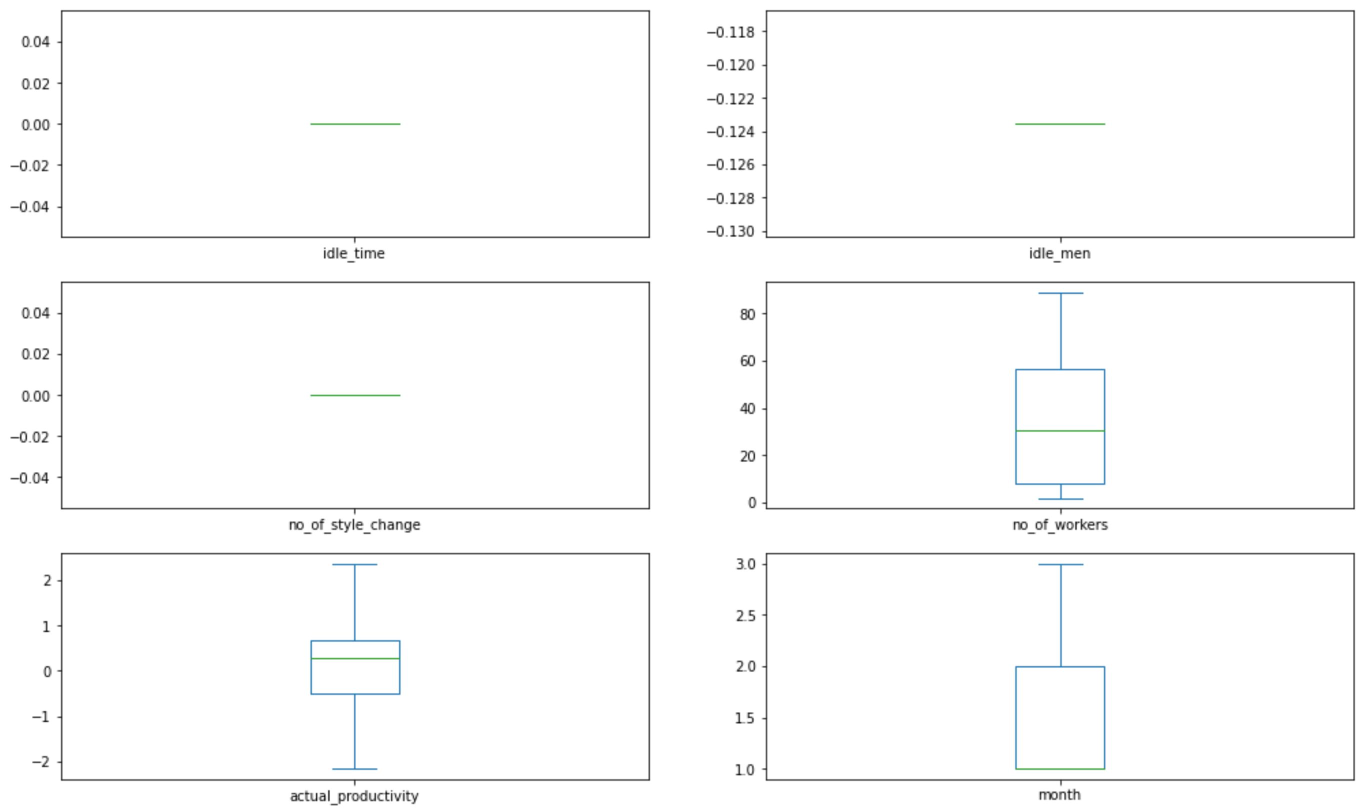
Remove the outlier:

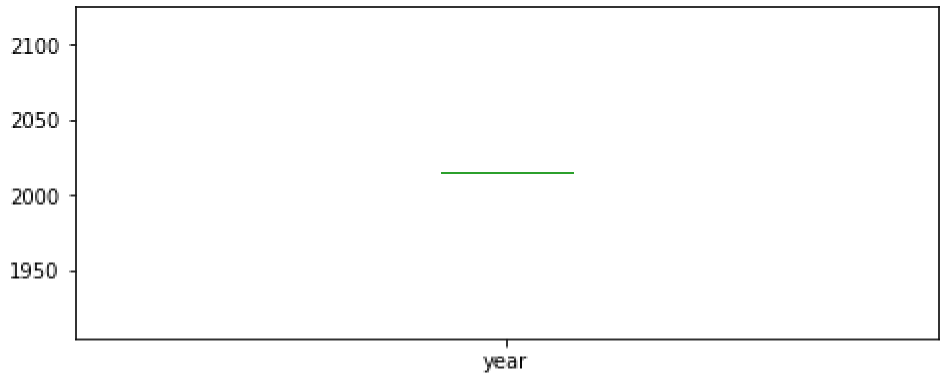
```
def remove(df1, ls):
  ls = sorted(set(ls))
  df1 = df1.drop(ls)
  return df1

df1_cleaned = remove(df1, index_list)

df1_cleaned.shape
```

(1027, 16)

```
df1_cleaned.plot(kind="box",subplots=True,layout=(7,2),figsize=(17,25));
```







There we will remove all outlier data in our dataset.

Data Pre-processing:-

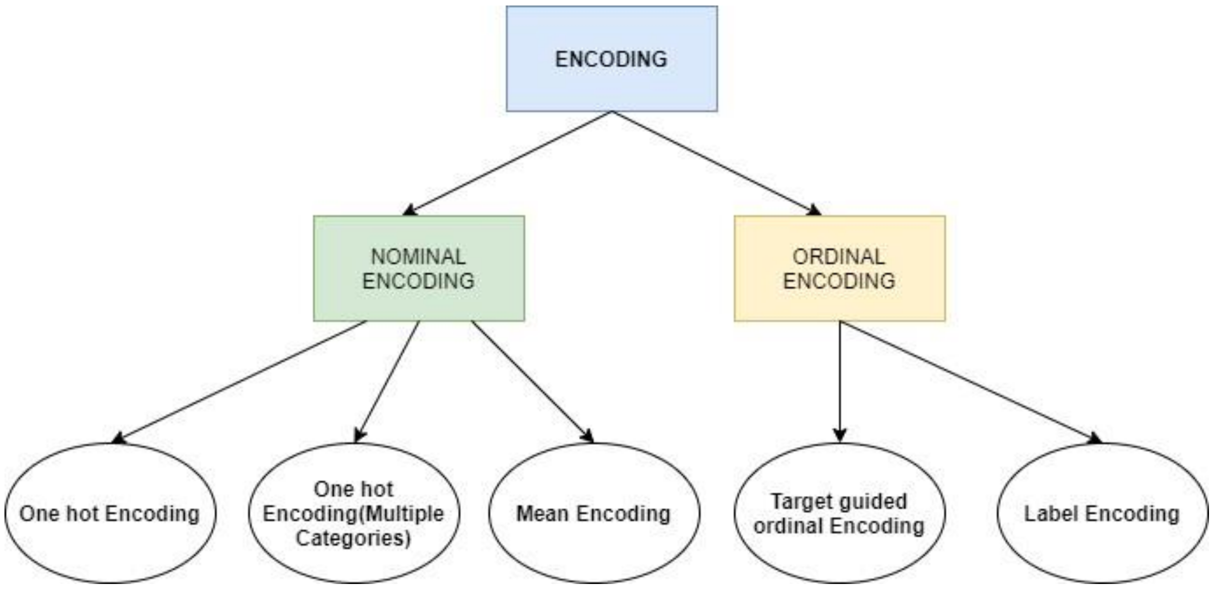
Encoding:-Encoding is a technique of converting categorical variables into numerical values so that it could be easily fitted to a machine learning model. Different types of categorical variables.

Ordinal Categorical variable:

Ordinal categories are those in which we have to worry about the rank. These categories can be rearranged based on ranks.

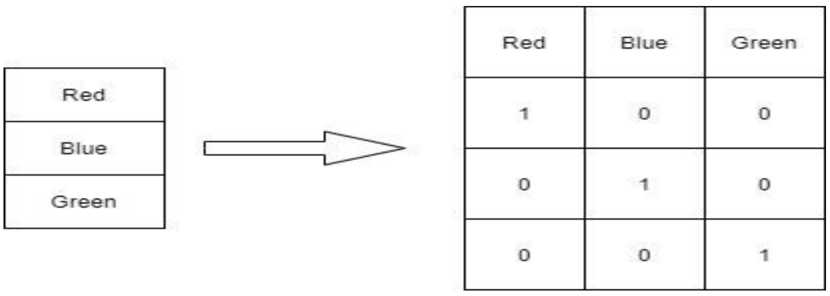
Now that we have discussed about the type of categorical variables, let’s see the different types of encoding:

1. Nominal Encoding
2. Ordinal Encoding



1. One Hot Encoding

This method is applied to nominal categorical variables. Example, suppose we have a column containing 3 categorical variables, then in one hot encoding 3 columns will be created each for a categorical variable.



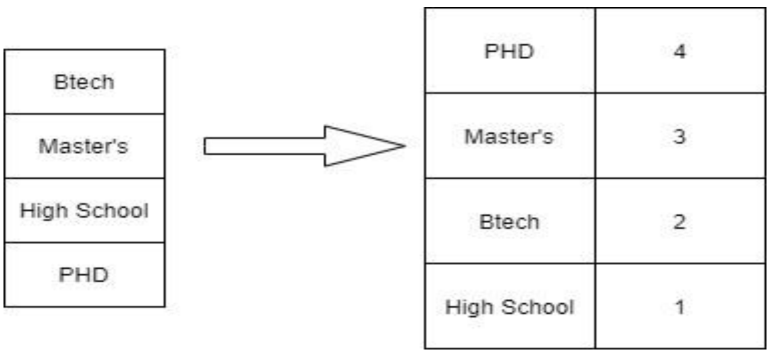
Red
Blue
Green

Red	Blue	Green
1	0	0
0	1	0
0	0	1

One Hot Encoding

2. Label Encoding

This technique will be used only for Ordinal categories. Ranks are provided based on the importance of the category. Below table illustrates that PHD is considered as the highest degree, so the highest label is given to it and so on.

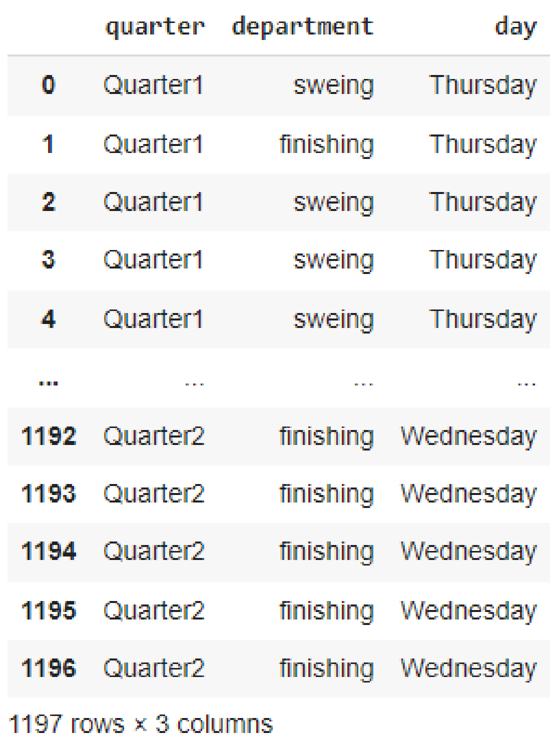


Btech
Master's
High School
PHD

PHD	4
Master's	3
Btech	2
High School	1

Data Pre-processing Imputing

```
df1[['quarter', 'department', 'day']]
```



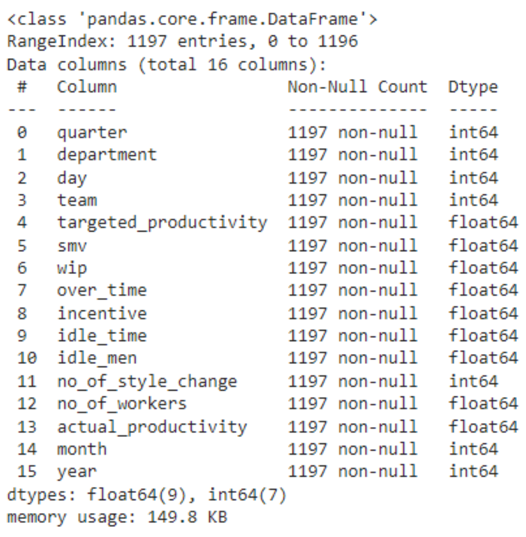
	quarter	department	day
0	Quarter1	sweing	Thursday
1	Quarter1	finishing	Thursday
2	Quarter1	sweing	Thursday
3	Quarter1	sweing	Thursday
4	Quarter1	sweing	Thursday
...	...	...	...
1192	Quarter2	finishing	Wednesday
1193	Quarter2	finishing	Wednesday
1194	Quarter2	finishing	Wednesday
1195	Quarter2	finishing	Wednesday
1196	Quarter2	finishing	Wednesday

1197 rows × 3 columns

```
# Using Label encoding....

from sklearn import preprocessing

label_encoder = preprocessing.LabelEncoder()
df1['quarter']= label_encoder.fit_transform(df1['quarter'])
df1['day']= label_encoder.fit_transform(df1['day'])
df1['department']= label_encoder.fit_transform(df1['department'])
```

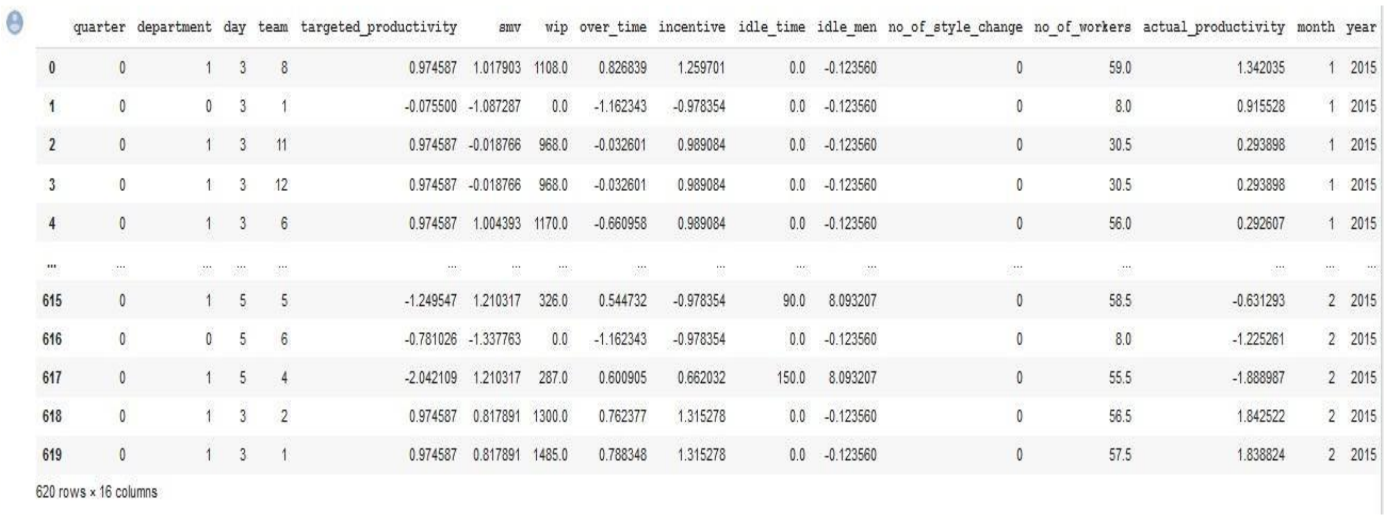
#Here we use label encoding because according to the above explanation quarter, day and department belong to ordinal categories. now we see all feature data ,after using label encoding

```
df1.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1197 entries, 0 to 1196
Data columns (total 16 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   quarter                1197 non-null   int64
 1   department             1197 non-null   int64
 2   day                    1197 non-null   int64
 3   team                   1197 non-null   int64
 4   targeted_productivity  1197 non-null   float64
 5   smv                    1197 non-null   float64
 6   wip                    1197 non-null   float64
 7   over_time              1197 non-null   float64
 8   incentive              1197 non-null   float64
 9   idle_time              1197 non-null   float64
 10  idle_men               1197 non-null   float64
 11  no_of_style_change     1197 non-null   int64
 12  no_of_workers          1197 non-null   float64
 13  actual_productivity    1197 non-null   float64
 14  month                  1197 non-null   int64
 15  year                   1197 non-null   int64
dtypes: float64(9), int64(7)
memory usage: 149.8 KB
```

df1.head(620)



	quarter	department	day	team	targeted_productivity	smv	wip	over_time	incentive	idle_time	idle_men	no_of_style_change	no_of_workers	actual_productivity	month	year
0	0	1	3	8	0.974587	1.017903	1108.0	0.826839	1.259701	0.0	-0.123560	0	59.0	1.342035	1	2015
1	0	0	3	1	-0.075500	-1.087287	0.0	-1.162343	-0.978354	0.0	-0.123560	0	8.0	0.915528	1	2015
2	0	1	3	11	0.974587	-0.018766	968.0	-0.032601	0.989084	0.0	-0.123560	0	30.5	0.293898	1	2015
3	0	1	3	12	0.974587	-0.018766	968.0	-0.032601	0.989084	0.0	-0.123560	0	30.5	0.293898	1	2015
4	0	1	3	6	0.974587	1.004393	1170.0	-0.660958	0.989084	0.0	-0.123560	0	56.0	0.292607	1	2015
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
615	0	1	5	5	-1.249547	1.210317	326.0	0.544732	-0.978354	90.0	8.093207	0	58.5	-0.631293	2	2015
616	0	0	5	6	-0.781026	-1.337763	0.0	-1.162343	-0.978354	0.0	-0.123560	0	8.0	-1.225261	2	2015
617	0	1	5	4	-2.042109	1.210317	287.0	0.600905	0.662032	150.0	8.093207	0	55.5	-1.888987	2	2015
618	0	1	3	2	0.974587	0.817891	1300.0	0.762377	1.315278	0.0	-0.123560	0	56.5	1.842522	2	2015
619	0	1	3	1	0.974587	0.817891	1485.0	0.788348	1.315278	0.0	-0.123560	0	57.5	1.838824	2	2015

620 rows × 16 columns

Here we can see that in quarter feature data are changed for example:-”quater1” is 0, ”quater2” is 1,”quater3” is 2,”quater4” is 3,”quater5” is 4.

In day feature data are changed for example:-”Monday” is 0,”**Tuesday” is 1, “Wednesday” is 2,” Thursday” is 3, “Friday” is 4, “Saturday” is 5 and “Sunday” is 6.**

In day feature data are changed for example:-”sewing “ is 0 and “finishing” is 1.

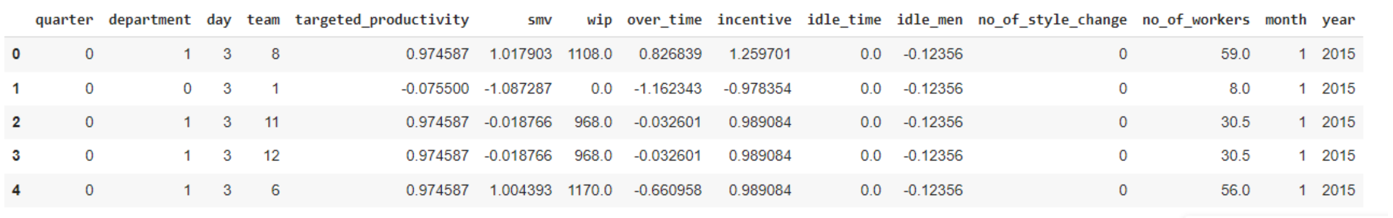
Train Model:-

Before sending the dataset to the machine, we need to mention which of these are input features and which of these are target feature.

```
X=df1.drop(columns='actual_productivity')
y=df1['actual_productivity']
```

We have taken all features in X without target feature.
We have taken only the target feature in y.
Here we will show which feature are present in X.

X.head()



	quarter	department	day	team	targeted_productivity	smv	wip	over_time	incentive	idle_time	idle_men	no_of_style_change	no_of_workers	month	year
0	0	1	3	8	0.974587	1.017903	1108.0	0.826839	1.259701	0.0	-0.12356	0	59.0	1	2015
1	0	0	3	1	-0.075500	-1.087287	0.0	-1.162343	-0.978354	0.0	-0.12356	0	8.0	1	2015
2	0	1	3	11	0.974587	-0.018766	968.0	-0.032601	0.989084	0.0	-0.12356	0	30.5	1	2015
3	0	1	3	12	0.974587	-0.018766	968.0	-0.032601	0.989084	0.0	-0.12356	0	30.5	1	2015
4	0	1	3	6	0.974587	1.004393	1170.0	-0.660958	0.989084	0.0	-0.12356	0	56.0	1	2015

Here we will show which feature are present in y

y.head ()

```
0       1.342035
1       0.915528
2       0.293898
3       0.293898
4       0.292607
Name: actual_productivity, dtype: float64
```

Now this dataset will be divided into training and testing datasets.

MultiCollinearity

What is MultiCollinearity?

MultiCollinearity occurs when two or more independent variables are highly correlated with one another in a regression model.

This means that an independent variable can be predicted from another independent variable in a regression model. For example, height and weight, household income and water consumption, mileage and price of a car, study time and leisure time.

The Problem with having MultiCollinearity

MultiCollinearity can be a problem in a regression model because we would not be able to distinguish between the individual effects of the independent variables on the dependent variable. For example, let's assume that in the following linear equation:

$$Y = W0+W1*X1+W2*X2$$

Coefficient W1 is the increase in Y for a unit increase in X1 while keeping X2 constant. But since X1 and X2 are highly correlated, changes in X1 would also cause changes in X2 and we would not be able to see their individual effect on Y.

"This makes the effects of X1 on Y difficult to distinguish from the effects of X2 on Y."

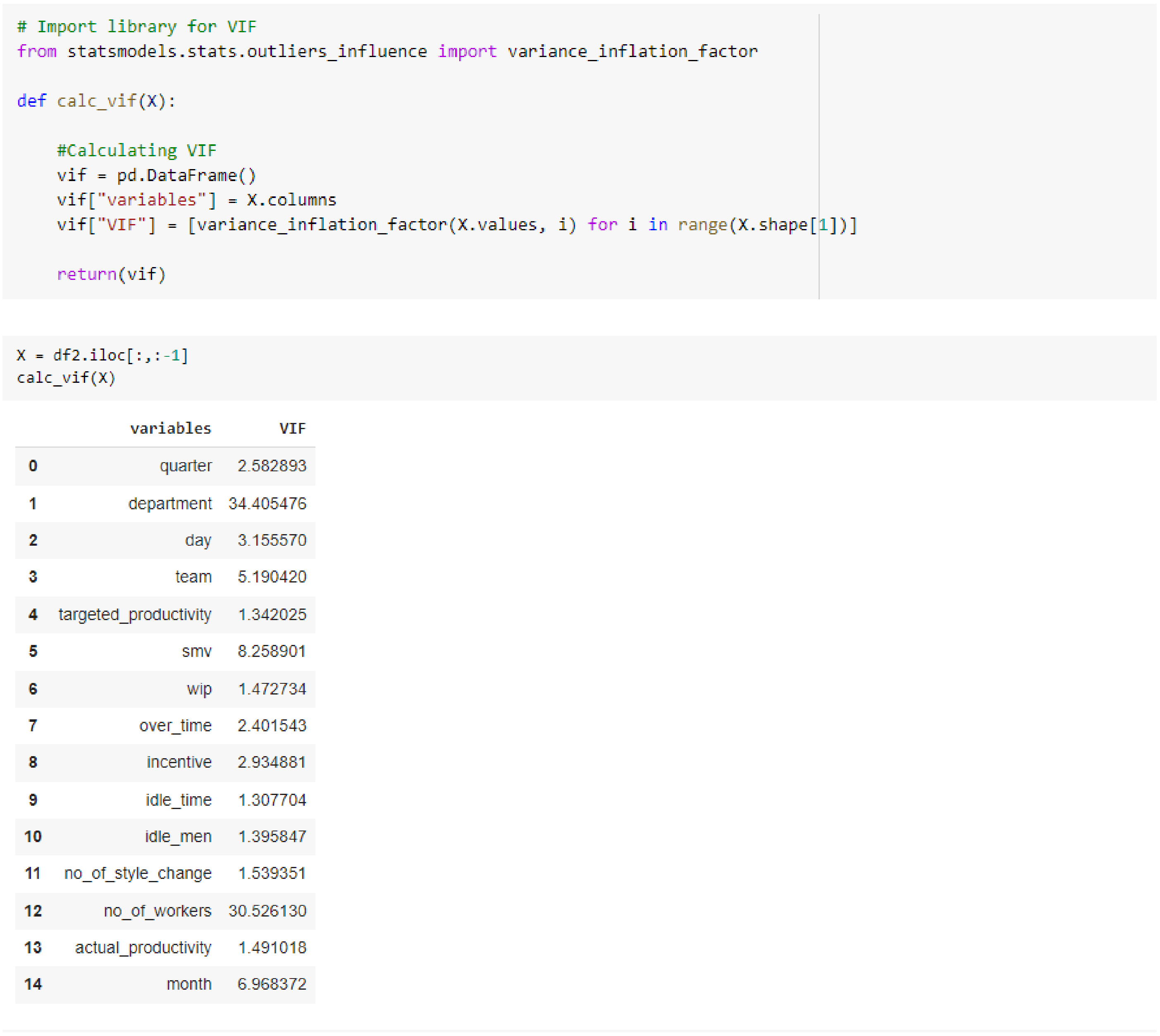
MultiCollinearity may not affect the accuracy of the model as much. But we might lose reliability in determining the effects of individual features in your model – and that can be a problem when it comes to interpretability.

Detecting MultiCollinearity using VIF

Let’s try detecting MultiCollinearity in the dataset.

MultiCollinearity can be detected via various methods. In this article, we will focus on the most common one – **VIF (Variable Inflation Factors)**.

VIF determines the strength of the correlation between the independent variables. It is predicted by taking a variable and regressing it against every other variable.



```
# Import library for VIF
from statsmodels.stats.outliers_influence import variance_inflation_factor

def calc_vif(X):

    #Calculating VIF
    vif = pd.DataFrame()
    vif["variables"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

    return(vif)
```

```
X = df2.iloc[:,:-1]
calc_vif(X)
```

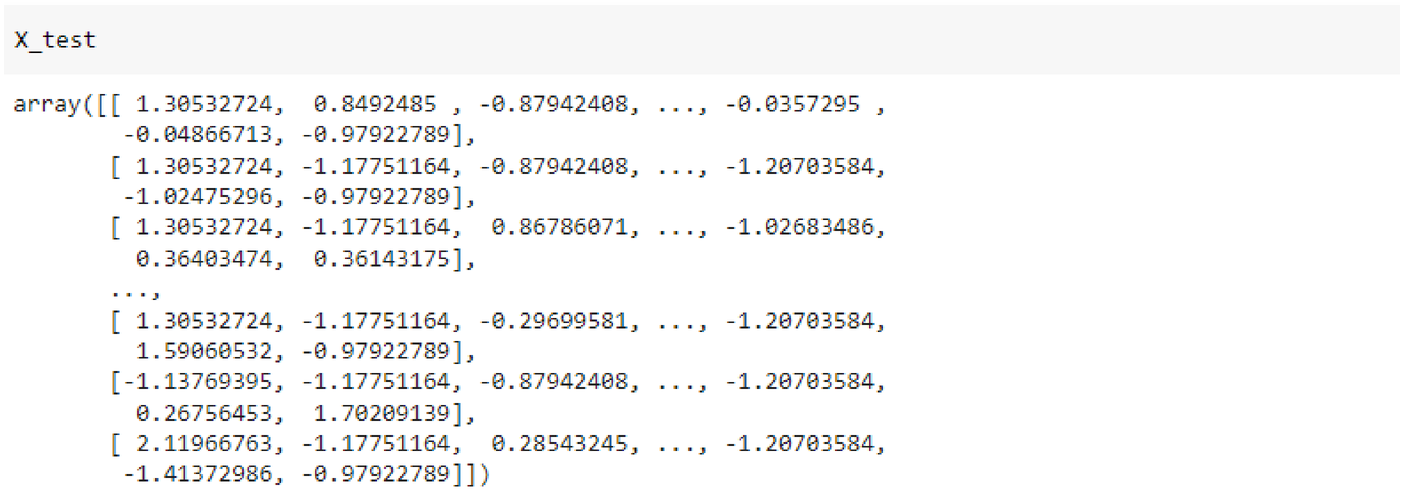
	variables	VIF
0	quarter	2.582893
1	department	34.405476
2	day	3.155570
3	team	5.190420
4	targeted_productivity	1.342025
5	smv	8.258901
6	wip	1.472734
7	over_time	2.401543
8	incentive	2.934881
9	idle_time	1.307704
10	idle_men	1.395847
11	no_of_style_change	1.539351
12	no_of_workers	30.526130
13	actual_productivity	1.491018
14	month	6.968372

The training dataset will be used to train the model and the testing dataset will be used to evaluate and test the model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

X_train.shape
(957, 15)

#The X_train variable has 957 rows and 15 columns.


y_train.shape
(957,)
```

Now we need to scale the features so that all of them are in the same range.

```
from sklearn.preprocessing import StandardScaler
std = StandardScaler()

X_train=std.fit_transform(X_train)
X_test=std.transform(X_test)
```



```
X_test
```

```
array([[ 1.30532724,  0.8492485 , -0.87942408, ..., -0.0357295 ,
        -0.04866713, -0.97922789],
       [ 1.30532724, -1.17751164, -0.87942408, ..., -1.20703584,
        -1.02475296, -0.97922789],
       [ 1.30532724, -1.17751164,  0.86786071, ..., -1.02683486,
         0.36403474,  0.36143175],
       ...,
       [ 1.30532724, -1.17751164, -0.29699581, ..., -1.20703584,
         1.59060532, -0.97922789],
       [-1.13769395, -1.17751164, -0.87942408, ..., -1.20703584,
         0.26756453,  1.70209139],
       [ 2.11966763, -1.17751164,  0.28543245, ..., -1.20703584,
        -1.41372986, -0.97922789]])
```

Matrix

Here we have changed the matrix of our data set using Mae, Mse, Rmse and R2.

```
from sklearn import metrics
models_metrics = pd.DataFrame(columns = [ 'models','mae','mse', 'rmse' ,'mape', 'R2'])
def evaluate_model(model,Y_actual,Y_Predicted, df):
    mape = np.mean(np.abs((Y_actual - Y_Predicted)/Y_actual))*100
    mae=metrics.mean_absolute_error(Y_actual, Y_Predicted)
    mse=metrics.mean_squared_error(Y_actual, Y_Predicted)
    rmse=np.sqrt(metrics.mean_squared_error(Y_actual, Y_Predicted))
    r2 = metrics.r2_score(Y_actual, Y_Predicted)
    df2 = {'models':model,'mae':mae,'mse':mse, 'rmse':rmse, 'mape':mape, 'R2': r2}
    df2 = df2.append(df2, ignore_index = True)
    return df2

    print (rmse)
```

```
from sklearn.model_selection import train_test_split,cross_val_score, cross_val_predict


#using the function to show ANN models in bar plot.
model1 = [ ]
model2 = [ ]
model3 = [ ]
ann_result = np.zeros(3)
ann_model_name= ['ANN 1 ', 'ANN 2 ', 'ANN 3']
#========================================================
```

Build the Models

There we use some models to show the result in our dataset. We use LinearRegression, DecisionTreeRegersSor, SVM (Support Vector Machine), and use 3 ANN (Artificial Neural Network) models. But of all the models, the best result give ANN model.

Linear Regression

In the simplest words, **Linear Regression** is the supervised Machine Learning model in which the **model finds the best fit linear line between the independent and dependent variable** i.e it finds the linear relationship between the dependent and independent variable.

Linear Regression is of two types: **Simple and Multiple**. **Simple Linear Regression** is where only one independent variable is present and the model has to find the linear relationship of it with the dependent variable

Whereas, In **Multiple Linear Regression** there are more than one independent variables for the model to find the relationship.

Equation of Simple Linear Regression, where b_o is the intercept, b_1 is coefficient or slope, x is the independent variable and y is the dependent variable.

$$y = b_o + b_1 x$$

Equation of Multiple Linear Regression, where do is the intercept, $b_1,b_2,b_3,b_4\ldots,b_n$ are coefficients or slopes of the independent variables $x_1,x_2,x_3,x_4\ldots, x_n$ and y is the dependent variable.

$$y = b_o + b_1 x_1 + b_2 x_2 + b_3 x_3 \ldots + b_n x_n$$

Using LinearRegression

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Model prediction
y_pred = lin_reg.predict(X_test)
```

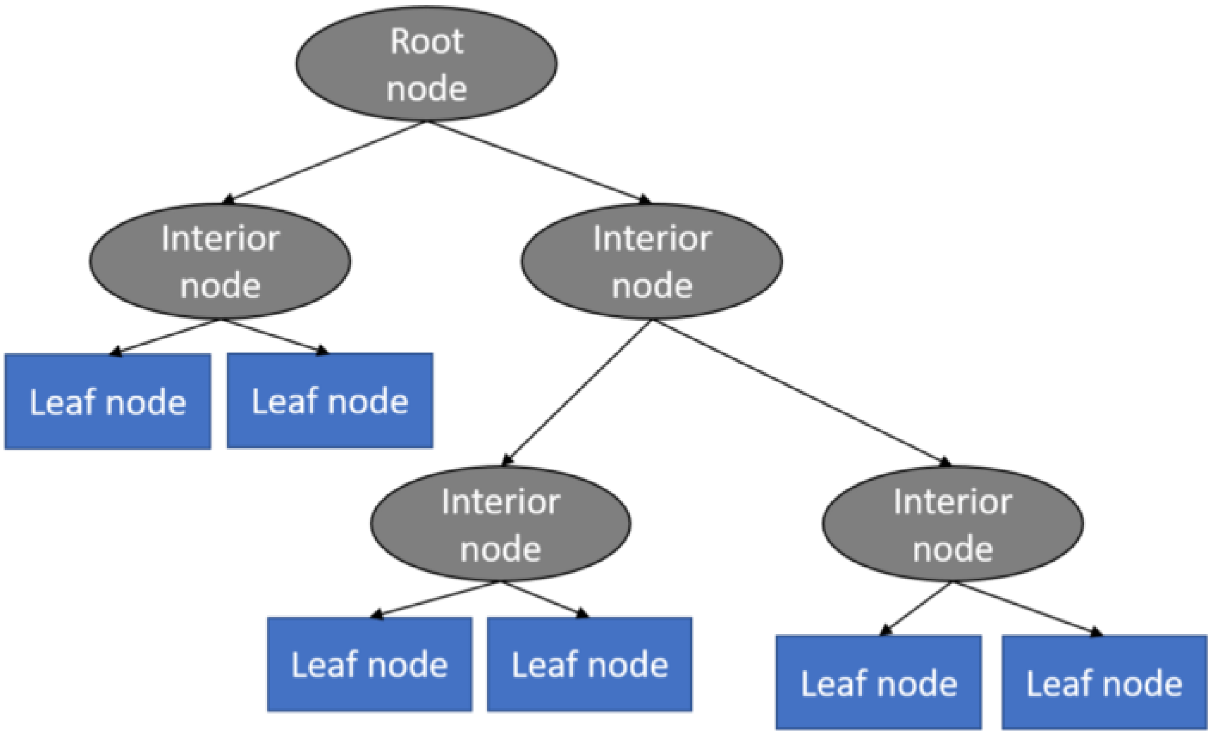
```
mae=metrics.mean_absolute_error(y_test, y_pred)
mse=metrics.mean_squared_error(y_test, y_pred)

# Printing the metrics
print('Linear Regression')
print('R2 square:',metrics.r2_score(y_test, y_pred))
model1.append(metrics.r2_score(y_test, y_pred))
print('MAE: ', mae)
print('MSE: ', mse)
print('')
```

```
Linear Regression

R2 square: 1.0

MAE:  9.710006236562929e-16

MSE:  1.4595210731231492e-30
```

DecisionTreeRegressor:-

Decision Tree is one of the most commonly used, practical approaches for supervised learning. It can be used to solve both Regression and Classification tasks with the latter being put more into practical application.

It is a tree-structured classifier with three types of nodes. The *Root Node* is the initial node which represents the entire sample and may get split further into further nodes. The *Interior Nodes* represent the features of a data set and the branches represent the decision rules. Finally, the *Leaf Nodes* represent the outcome. This algorithm is very useful for solving decision-related problems.



Using DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor
dec_re = DecisionTreeRegressor(random_state=0)
dec_re.fit(X_train, y_train)

# Model prediction
y_pred = dec_re.predict(X_test)
mae=metrics.mean_absolute_error(y_test, y_pred)
mse=metrics.mean_squared_error(y_test, y_pred)

# Printing the metrics
print('Decision Tree Regressor')
print('R2 square:',metrics.r2_score(y_test, y_pred))
model2.append(metrics.r2_score(y_test, y_pred))
print('MAE: ', mae)
print('MSE: ', mse)
print('')
```

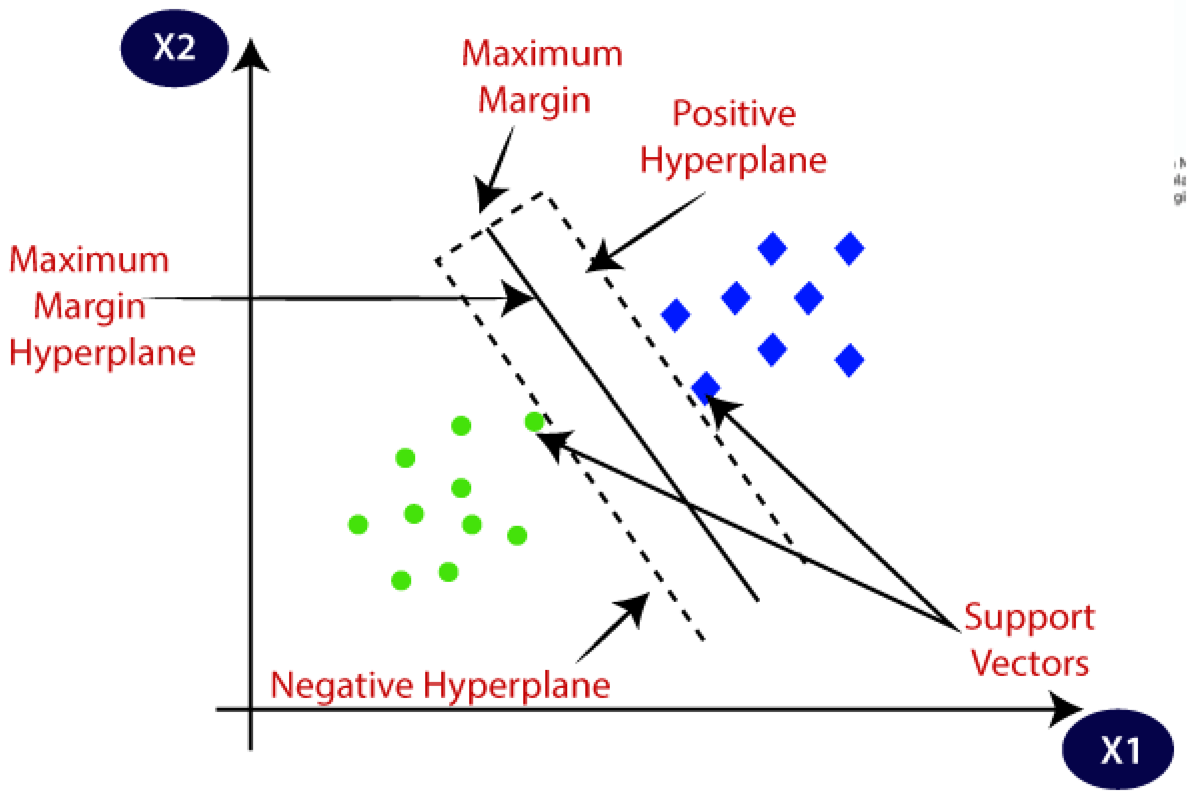
```
Decision Tree Regressor
R2 square: 0.9996376466448735
MAE:  0.007430554211216142
MSE:  0.00038797499410692544
```

SVM:-

Support Vector Machine or SVM is one of the most popular Supervised
Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence the algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



```
from sklearn.svm import SVR
svr = SVR(kernel = 'rbf')
svr.fit(X_train, y_train)

# Model prediction
y_pred = svr.predict(X_test)
mae=metrics.mean_absolute_error(y_test, y_pred)
mse=metrics.mean_squared_error(y_test, y_pred)

# Printing the metrics
print('SVM')
print('R2 square:',metrics.r2_score(y_test, y_pred))
model3.append(metrics.r2_score(y_test, y_pred))
print('MAE: ', mae)
print('MSE: ', mse)
print('')
```

```
SVM
R2 square: 0.977066737610753
MAE:  0.09025105387203199
MSE:  0.02455485016059866
```

ANN Models:

Artificial Neural Networks (ANN) or neural networks are computational algorithms.

It intended to simulate the behaviour of biological systems composed of “neurons”, ANN are computational models inspired by an animal‘s central nervous systems. It is capable of machine learning as well as pattern recognition. These presented as systems of interconnected “neurons” which can compute values from inputs.

A neural network is a machine learning algorithm based on the model of a human neuron. The human brain consists of millions of neurons. It sends and process signals in the form of electrical and chemical signals. These neurons are connected with a special structure known as synapses. Synapses allow neurons to pass signals. From large numbers of simulated neurons neural networks forms.

Artificial Neural Network Layers

Artificial Neural Network is typically organized in layers. Layers are being made up of many interconnected ‘Nodes’ which contain and ‘activation function’. A neural network may contain the following 3 layers:

a. Input layer: The activity of the input units represents the raw information that can fees into the network.

b. Hidden layer: To determine the activity of each hidden unit. The activities of the input units and the weights on the connections between the input and the hidden units. There may be one or more hidden layers.

c. Output layer: The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import tensorflow as tf


# ANN1 model

ANN = Sequential()
ANN.add(Dense(32,activation='relu'))
ANN.add(Dense(32,activation='relu'))
ANN.add(Dense(32,activation='relu'))
ANN.add(Dense(16,activation='relu'))
ANN.add(Dense(16,activation='relu'))
ANN.add(Dense(8,activation='relu'))
ANN.add(Dense(8,activation='relu'))
ANN.add(Dense(2,activation='relu'))
ANN.add(Dense(2,activation='relu'))
ANN.add(Dense(1))

from tensorflow.keras import backend as K

def coeff_determination(y_true, y_pred):
    SS_res =  K.sum(K.square( y_true-y_pred ))
    SS_tot = K.sum(K.square( y_true - K.mean(y_true) ) )
    return ( 1 - SS_res/(SS_tot + K.epsilon()) )
```

Training the model

Now we will train our model. To train, we will use the 'fit()' function on our model with the following five parameters: training data (train_X), target data (train_y), validation split, the number of epochs and call-backs.

The validation split will randomly split the data into use for training and testing. During training, we will be able to see the validation loss, which gives the mean squared error of our model on the validation set. We will set the validation split at 0.2, which means that 20% of the training data we provide in the model will be set aside for testing model performance.

The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that point, the model will stop improving during each epoch. In addition, the more epochs, the longer the model will take to run. To monitor this, we will use 'early stopping'.

Early stopping will stop the model from training before the number of epochs is reached if the model stops improving. We will set our early stopping monitor to 3. This means that after 3 epochs in a row in which the model doesn't improve, training will stop. Sometimes, the validation loss can stop improving then improve in the next epoch, but after 3 epochs in which the validation loss doesn't improve, it usually won't improve again.

Optimizer

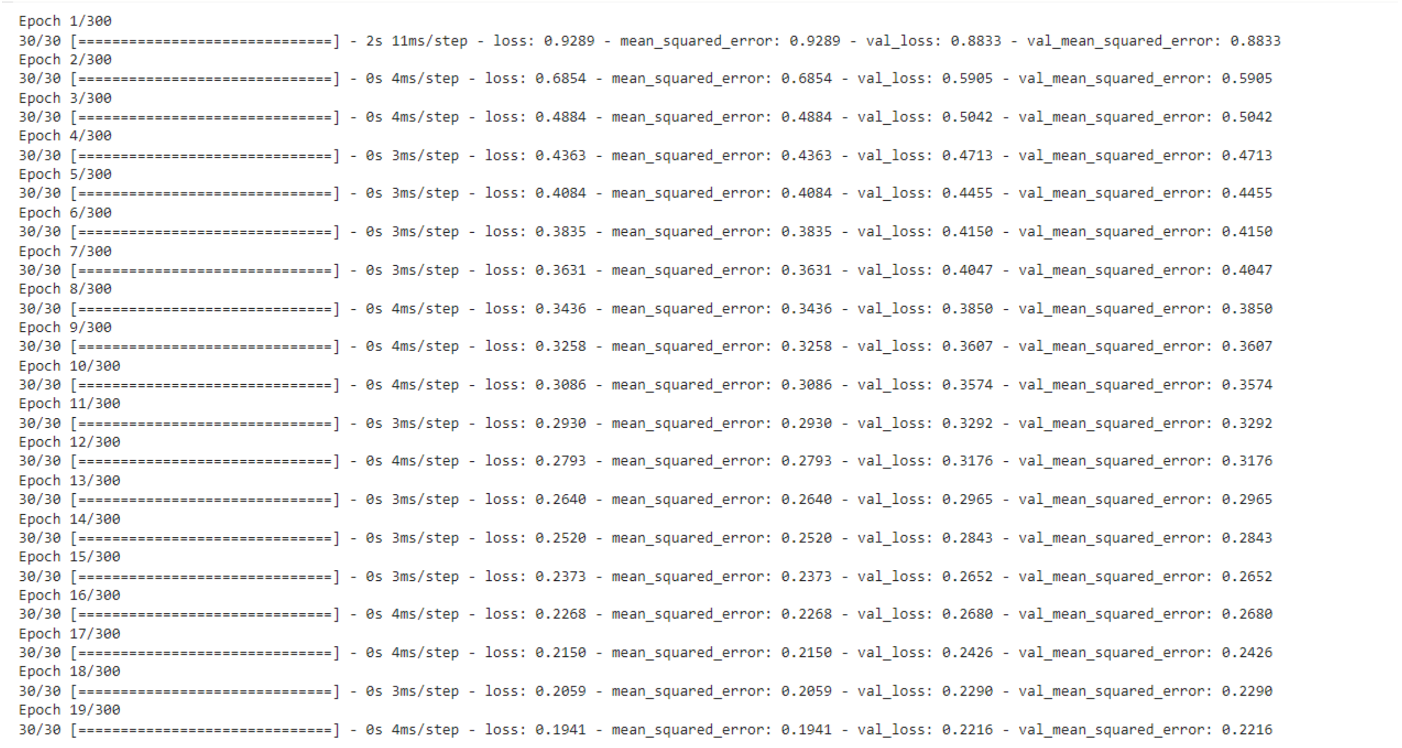
An optimizer is a function of an algorithm that modifies the attributes of the neural network, such as weights and learning rate. Thus, it helps in reducing the overall loss and improve the accuracy.

In our model we use RMS prop Optimizer.

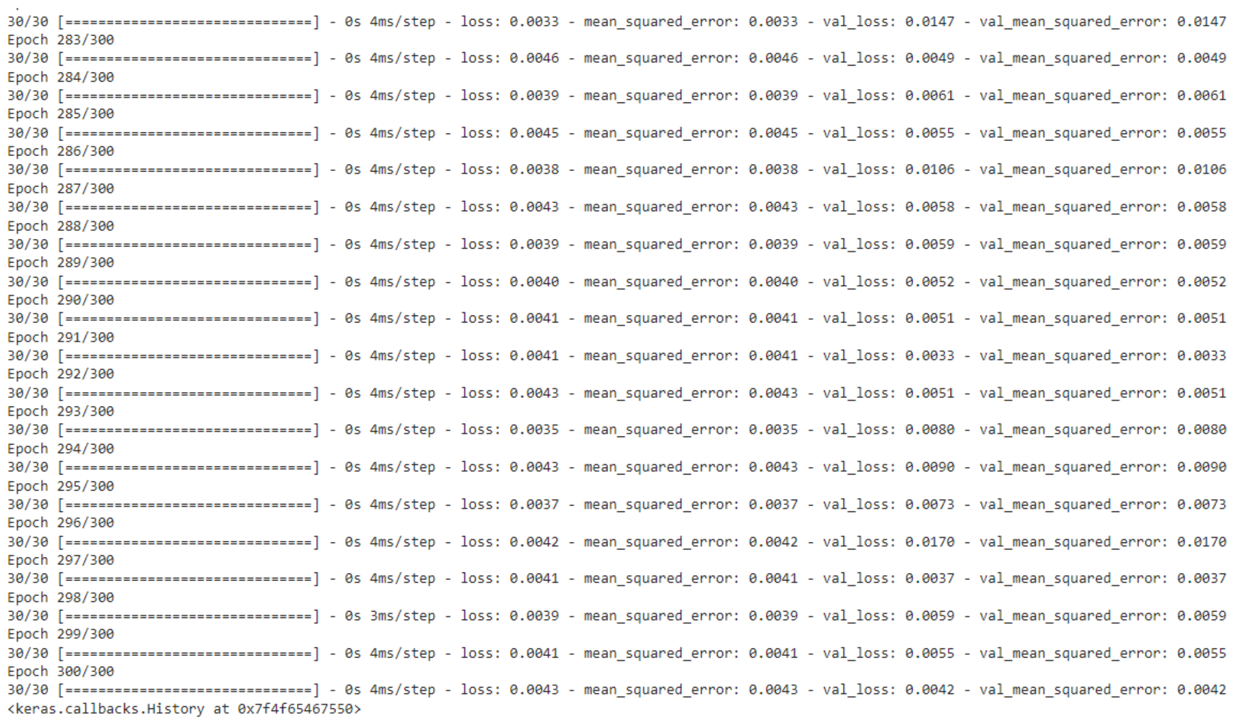
RMS Prop:
RMS prop is one of the popular optimizer among deep learning enthusiasts. This is maybe because it hasn't been published but still very well known in the community. RMS prop is ideally an extension of the work RPPROP.

Adam Optimizer:
The name adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training. Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually.

```
from typing_extensions import Annotated
ANN.compile(optimizer=tf.keras.optimizers.RMSprop(),
            loss='mean_squared_error',
            metrics=['mean_squared_error'])

ANN.fit(x=X_train,y=y_train,
        validation_data=(X_test,y_test),
        batch_size=32,epochs=300)
```



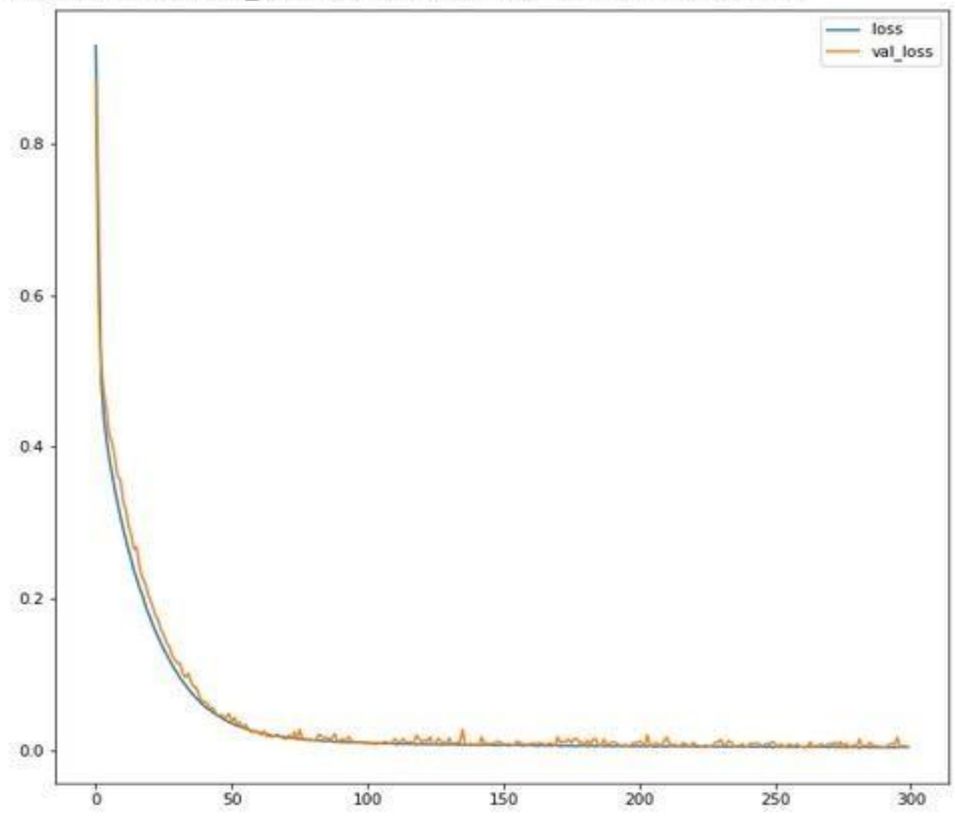
```
Epoch 1/300
30/30 [==============================] - 2s 11ms/step - loss: 0.9289 - mean_squared_error: 0.9289 - val_loss: 0.8833 - val_mean_squared_error: 0.8833
Epoch 2/300
30/30 [==============================] - 0s 4ms/step - loss: 0.6854 - mean_squared_error: 0.6854 - val_loss: 0.5905 - val_mean_squared_error: 0.5905
Epoch 3/300
30/30 [==============================] - 0s 4ms/step - loss: 0.4884 - mean_squared_error: 0.4884 - val_loss: 0.5042 - val_mean_squared_error: 0.5042
Epoch 4/300
30/30 [==============================] - 0s 3ms/step - loss: 0.4363 - mean_squared_error: 0.4363 - val_loss: 0.4713 - val_mean_squared_error: 0.4713
Epoch 5/300
30/30 [==============================] - 0s 3ms/step - loss: 0.4084 - mean_squared_error: 0.4084 - val_loss: 0.4455 - val_mean_squared_error: 0.4455
Epoch 6/300
30/30 [==============================] - 0s 3ms/step - loss: 0.3835 - mean_squared_error: 0.3835 - val_loss: 0.4150 - val_mean_squared_error: 0.4150
Epoch 7/300
30/30 [==============================] - 0s 3ms/step - loss: 0.3631 - mean_squared_error: 0.3631 - val_loss: 0.4047 - val_mean_squared_error: 0.4047
Epoch 8/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3436 - mean_squared_error: 0.3436 - val_loss: 0.3850 - val_mean_squared_error: 0.3850
Epoch 9/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3258 - mean_squared_error: 0.3258 - val_loss: 0.3607 - val_mean_squared_error: 0.3607
Epoch 10/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3086 - mean_squared_error: 0.3086 - val_loss: 0.3574 - val_mean_squared_error: 0.3574
Epoch 11/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2930 - mean_squared_error: 0.2930 - val_loss: 0.3292 - val_mean_squared_error: 0.3292
Epoch 12/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2793 - mean_squared_error: 0.2793 - val_loss: 0.3176 - val_mean_squared_error: 0.3176
Epoch 13/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2640 - mean_squared_error: 0.2640 - val_loss: 0.2965 - val_mean_squared_error: 0.2965
Epoch 14/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2520 - mean_squared_error: 0.2520 - val_loss: 0.2843 - val_mean_squared_error: 0.2843
Epoch 15/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2373 - mean_squared_error: 0.2373 - val_loss: 0.2652 - val_mean_squared_error: 0.2652
Epoch 16/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2268 - mean_squared_error: 0.2268 - val_loss: 0.2680 - val_mean_squared_error: 0.2680
Epoch 17/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2150 - mean_squared_error: 0.2150 - val_loss: 0.2426 - val_mean_squared_error: 0.2426
Epoch 18/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2059 - mean_squared_error: 0.2059 - val_loss: 0.2290 - val_mean_squared_error: 0.2290
Epoch 19/300
30/30 [==============================] - 0s 4ms/step - loss: 0.1941 - mean_squared_error: 0.1941 - val_loss: 0.2216 - val_mean_squared_error: 0.2216
```



```
30/30 [==============================] - 0s 4ms/step - loss: 0.0033 - mean_squared_error: 0.0033 - val_loss: 0.0147 - val_mean_squared_error: 0.0147
Epoch 283/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0046 - mean_squared_error: 0.0046 - val_loss: 0.0049 - val_mean_squared_error: 0.0049
Epoch 284/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0039 - mean_squared_error: 0.0039 - val_loss: 0.0061 - val_mean_squared_error: 0.0061
Epoch 285/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0045 - mean_squared_error: 0.0045 - val_loss: 0.0055 - val_mean_squared_error: 0.0055
Epoch 286/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0038 - mean_squared_error: 0.0038 - val_loss: 0.0106 - val_mean_squared_error: 0.0106
Epoch 287/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0058 - val_mean_squared_error: 0.0058
Epoch 288/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0039 - mean_squared_error: 0.0039 - val_loss: 0.0059 - val_mean_squared_error: 0.0059
Epoch 289/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0040 - mean_squared_error: 0.0040 - val_loss: 0.0052 - val_mean_squared_error: 0.0052
Epoch 290/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0051 - val_mean_squared_error: 0.0051
Epoch 291/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0033 - val_mean_squared_error: 0.0033
Epoch 292/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0051 - val_mean_squared_error: 0.0051
Epoch 293/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0035 - mean_squared_error: 0.0035 - val_loss: 0.0080 - val_mean_squared_error: 0.0080
Epoch 294/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0090 - val_mean_squared_error: 0.0090
Epoch 295/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0037 - mean_squared_error: 0.0037 - val_loss: 0.0073 - val_mean_squared_error: 0.0073
Epoch 296/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0042 - mean_squared_error: 0.0042 - val_loss: 0.0170 - val_mean_squared_error: 0.0170
Epoch 297/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0037 - val_mean_squared_error: 0.0037
Epoch 298/300
30/30 [==============================] - 0s 3ms/step - loss: 0.0039 - mean_squared_error: 0.0039 - val_loss: 0.0059 - val_mean_squared_error: 0.0059
Epoch 299/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0055 - val_mean_squared_error: 0.0055
Epoch 300/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0042 - val_mean_squared_error: 0.0042
<keras.callbacks.History at 0x7f4f65467550>
```

```
dfl_RMS[['loss', 'val_loss']].plot(figsize=(10,10))
```

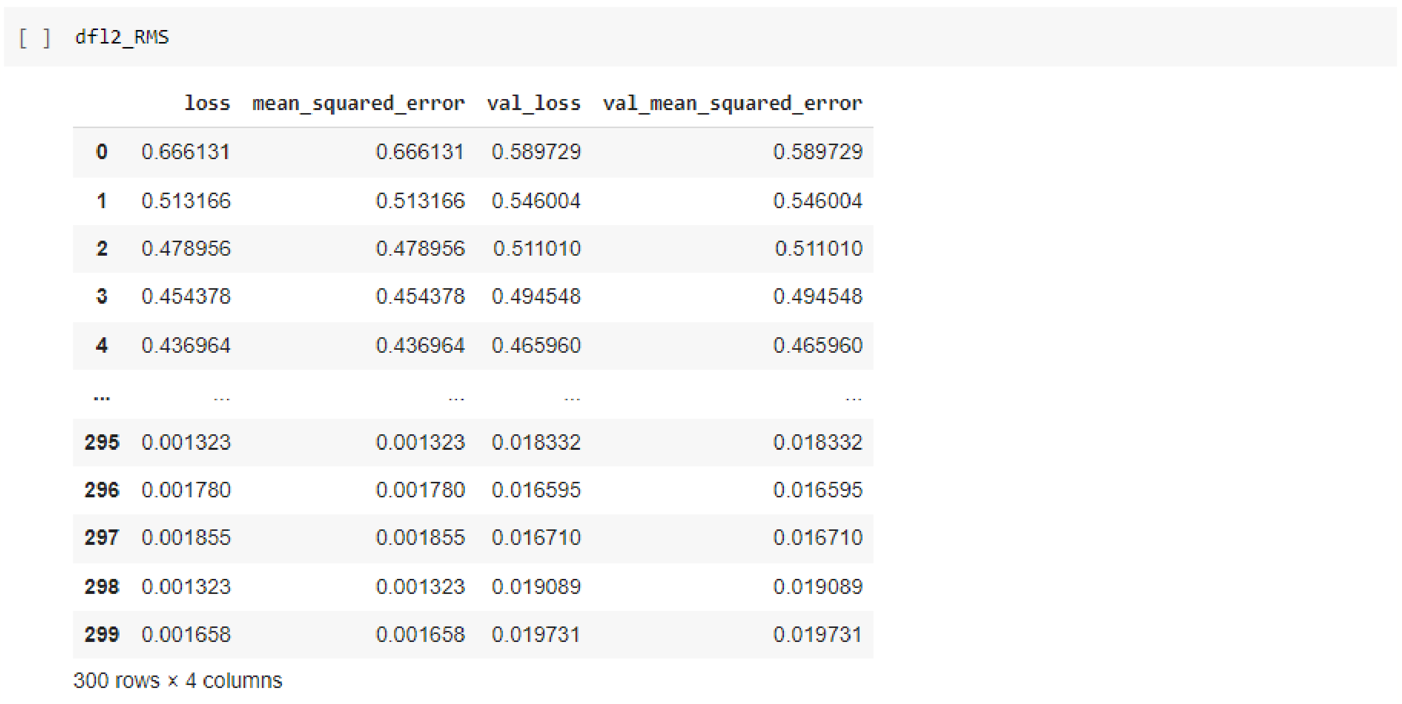
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4f63af2810>
```



```
# collecting the data in 1st ANN model
ann_result[0]=np.round(ANN.evaluate(X_test, y_test, verbose=0)[1], 3)

ann_result[0]
0.004


# ANN2 model

ANN1 = Sequential()
ANN1.add(Dense(64,activation='relu'))
ANN1.add(Dense(32,activation='relu'))

ANN1.add(Dense(16,activation='relu'))

ANN1.add(Dense(8,activation='relu'))

ANN1.add(Dense(2,activation='relu'))

ANN1.add(Dense(1))

ANN1.compile(optimizer=tf.keras.optimizers.RMSprop(),
          loss='mean_squared_error',
          metrics=['mean_squared_error'])

N1.fit(x=X_train,y=y_train,
        validation_data=(X_test,y_test),
batch_size=32,epochs=300)
```

```
Epoch 1/300
30/30 [==============================] - 1s 9ms/step - loss: 0.6661 - mean_squared_error: 0.6661 - val_loss: 0.5897 - val_mean_squared_error: 0.5897
Epoch 2/300
30/30 [==============================] - 0s 4ms/step - loss: 0.5132 - mean_squared_error: 0.5132 - val_loss: 0.5460 - val_mean_squared_error: 0.5460
Epoch 3/300
30/30 [==============================] - 0s 4ms/step - loss: 0.4790 - mean_squared_error: 0.4790 - val_loss: 0.5110 - val_mean_squared_error: 0.5110
Epoch 4/300
30/30 [==============================] - 0s 3ms/step - loss: 0.4544 - mean_squared_error: 0.4544 - val_loss: 0.4945 - val_mean_squared_error: 0.4945
Epoch 5/300
30/30 [==============================] - 0s 3ms/step - loss: 0.4370 - mean_squared_error: 0.4370 - val_loss: 0.4660 - val_mean_squared_error: 0.4660
Epoch 6/300
30/30 [==============================] - 0s 3ms/step - loss: 0.4133 - mean_squared_error: 0.4133 - val_loss: 0.4459 - val_mean_squared_error: 0.4459
Epoch 7/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3967 - mean_squared_error: 0.3967 - val_loss: 0.4222 - val_mean_squared_error: 0.4222
Epoch 8/300
30/30 [==============================] - 0s 3ms/step - loss: 0.3780 - mean_squared_error: 0.3780 - val_loss: 0.4097 - val_mean_squared_error: 0.4097
Epoch 9/300
30/30 [==============================] - 0s 3ms/step - loss: 0.3616 - mean_squared_error: 0.3616 - val_loss: 0.3859 - val_mean_squared_error: 0.3859
Epoch 10/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3459 - mean_squared_error: 0.3459 - val_loss: 0.3678 - val_mean_squared_error: 0.3678
Epoch 11/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3307 - mean_squared_error: 0.3307 - val_loss: 0.3527 - val_mean_squared_error: 0.3527
Epoch 12/300
30/30 [==============================] - 0s 4ms/step - loss: 0.3136 - mean_squared_error: 0.3136 - val_loss: 0.3335 - val_mean_squared_error: 0.3335
Epoch 13/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2996 - mean_squared_error: 0.2996 - val_loss: 0.3191 - val_mean_squared_error: 0.3191
Epoch 14/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2861 - mean_squared_error: 0.2861 - val_loss: 0.3031 - val_mean_squared_error: 0.3031
Epoch 15/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2729 - mean_squared_error: 0.2729 - val_loss: 0.3032 - val_mean_squared_error: 0.3032
Epoch 16/300
30/30 [==============================] - 0s 3ms/step - loss: 0.2609 - mean_squared_error: 0.2609 - val_loss: 0.2753 - val_mean_squared_error: 0.2753
Epoch 17/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2478 - mean_squared_error: 0.2478 - val_loss: 0.2650 - val_mean_squared_error: 0.2650
Epoch 18/300
30/30 [==============================] - 0s 4ms/step - loss: 0.2378 - mean_squared_error: 0.2378 - val_loss: 0.2498 - val_mean_squared_error: 0.2498
```

```
Epoch 282/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0015 - mean_squared_error: 0.0015 - val_loss: 0.0158 - val_mean_squared_error: 0.0158
Epoch 283/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0016 - mean_squared_error: 0.0016 - val_loss: 0.0197 - val_mean_squared_error: 0.0197
Epoch 284/300
30/30 [==============================] - 0s 3ms/step - loss: 0.0017 - mean_squared_error: 0.0017 - val_loss: 0.0178 - val_mean_squared_error: 0.0178
Epoch 285/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0016 - mean_squared_error: 0.0016 - val_loss: 0.0198 - val_mean_squared_error: 0.0198
Epoch 286/300
30/30 [==============================] - 0s 3ms/step - loss: 0.0017 - mean_squared_error: 0.0017 - val_loss: 0.0175 - val_mean_squared_error: 0.0175
Epoch 287/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0014 - mean_squared_error: 0.0014 - val_loss: 0.0158 - val_mean_squared_error: 0.0158
Epoch 288/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0016 - mean_squared_error: 0.0016 - val_loss: 0.0205 - val_mean_squared_error: 0.0205
Epoch 289/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0014 - mean_squared_error: 0.0014 - val_loss: 0.0229 - val_mean_squared_error: 0.0229
Epoch 290/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0018 - mean_squared_error: 0.0018 - val_loss: 0.0170 - val_mean_squared_error: 0.0170
Epoch 291/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0015 - mean_squared_error: 0.0015 - val_loss: 0.0207 - val_mean_squared_error: 0.0207
Epoch 292/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0017 - mean_squared_error: 0.0017 - val_loss: 0.0102 - val_mean_squared_error: 0.0102
Epoch 293/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0017 - mean_squared_error: 0.0017 - val_loss: 0.0228 - val_mean_squared_error: 0.0228
Epoch 294/300
30/30 [==============================] - 0s 3ms/step - loss: 0.0012 - mean_squared_error: 0.0012 - val_loss: 0.0229 - val_mean_squared_error: 0.0229
Epoch 295/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0019 - mean_squared_error: 0.0019 - val_loss: 0.0175 - val_mean_squared_error: 0.0175
Epoch 296/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0013 - mean_squared_error: 0.0013 - val_loss: 0.0183 - val_mean_squared_error: 0.0183
Epoch 297/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0018 - mean_squared_error: 0.0018 - val_loss: 0.0166 - val_mean_squared_error: 0.0166
Epoch 298/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0019 - mean_squared_error: 0.0019 - val_loss: 0.0167 - val_mean_squared_error: 0.0167
Epoch 299/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0013 - mean_squared_error: 0.0013 - val_loss: 0.0191 - val_mean_squared_error: 0.0191
Epoch 300/300
30/30 [==============================] - 0s 4ms/step - loss: 0.0017 - mean_squared_error: 0.0017 - val_loss: 0.0197 - val_mean_squared_error: 0.0197
<keras.callbacks.History at 0x7f4f652d0d90>
```



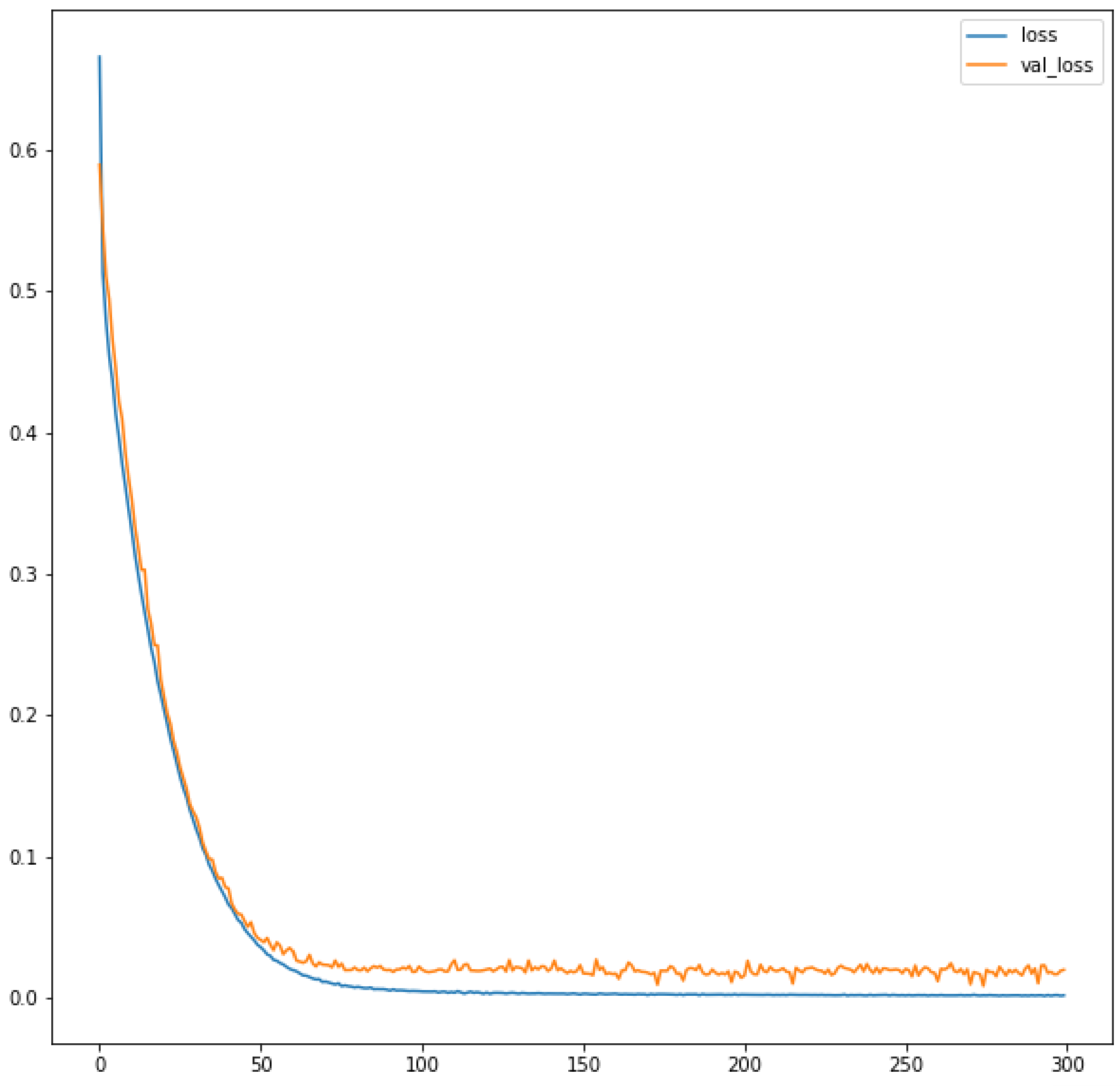
```
[ ] dfl2_RMS
```

	loss	mean_squared_error	val_loss	val_mean_squared_error
0	0.666131	0.666131	0.589729	0.589729
1	0.513166	0.513166	0.546004	0.546004
2	0.478956	0.478956	0.511010	0.511010
3	0.454378	0.454378	0.494548	0.494548
4	0.436964	0.436964	0.465960	0.465960
...	...	...	...	...
295	0.001323	0.001323	0.018332	0.018332
296	0.001780	0.001780	0.016595	0.016595
297	0.001855	0.001855	0.016710	0.016710
298	0.001323	0.001323	0.019089	0.019089
299	0.001658	0.001658	0.019731	0.019731

300 rows × 4 columns

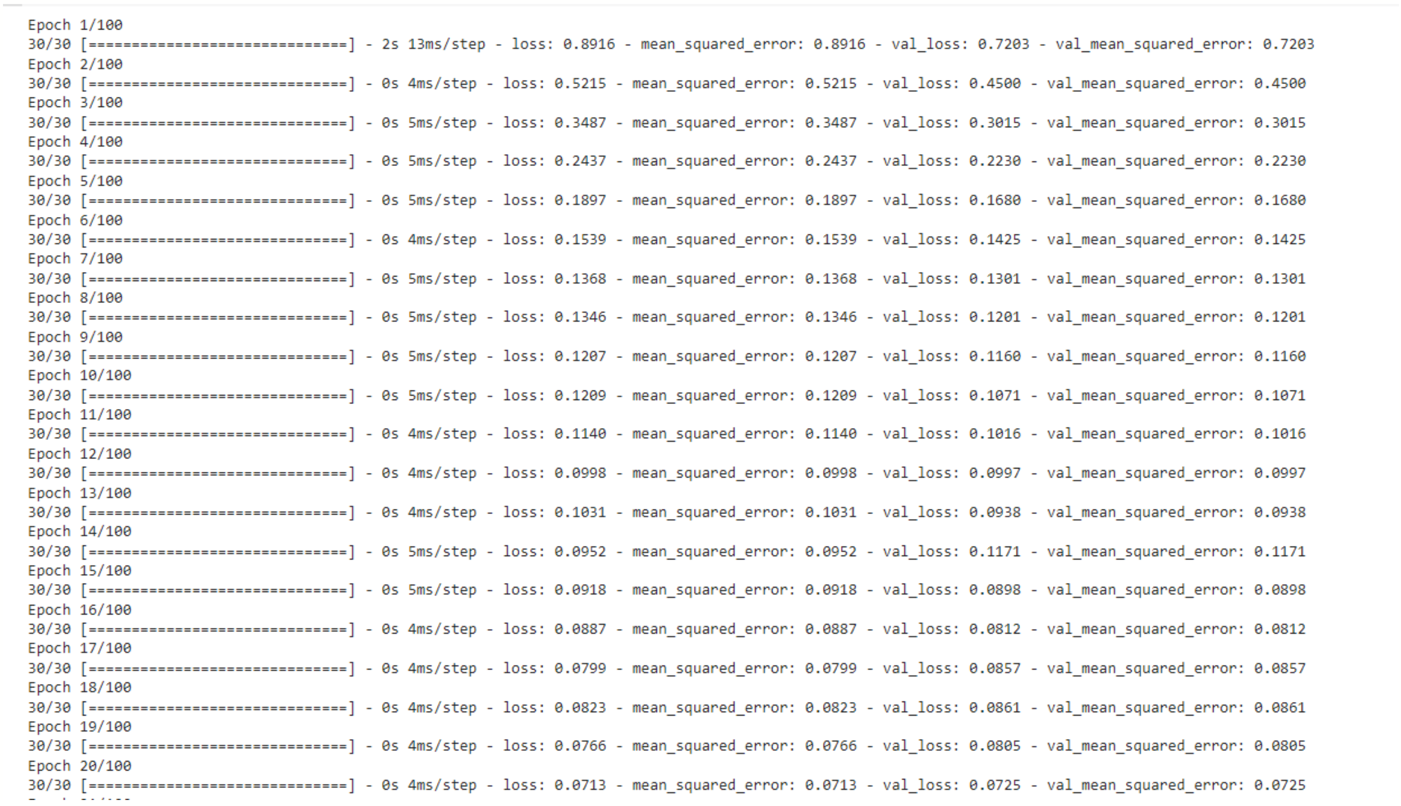
```
# There we see loss and val loss.
dfl2_RMS[['loss', 'val_loss']].plot(figsize=(10,10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4f65055cd0>
```



```
#collecting the data in 2ed ANN model
ann_result[1]=np.round(ANN1.evaluate(X_test, y_test, verbose=0)[1], 3)

ann_result[1]
0.02

#ANN3 model

ANN2 = Sequential()
ANN2.add(Dense(64,activation='relu'))
ANN2.add(Dense(64,activation='relu'))
ANN2.add(Dense(32,activation='relu'))
ANN2.add(Dense(32,activation='relu'))
ANN2.add(Dense(16,activation='relu'))
ANN2.add(Dense(16,activation='relu'))
ANN2.add(Dense(8,activation='relu'))
ANN2.add(Dense(8,activation='relu'))
ANN2.add(Dense(2,activation='relu'))
ANN2.add(Dense(2,activation='sigmoid'))
```

```
ANN2.add(Dense(1))

ANN2.compile(optimizer=tf.keras.optimizers.RMSprop(),
             loss='mean_squared_error',
             metrics=['mean_squared_error'])

ANN2.fit(x=X_train,y=y_train,
         validation_data=(X_test,y_test),
         batch_size=32,epochs=100)
```



```
Epoch 1/100
30/30 [==============================] - 2s 13ms/step - loss: 0.8916 - mean_squared_error: 0.8916 - val_loss: 0.7203 - val_mean_squared_error: 0.7203
Epoch 2/100
30/30 [==============================] - 0s 4ms/step - loss: 0.5215 - mean_squared_error: 0.5215 - val_loss: 0.4500 - val_mean_squared_error: 0.4500
Epoch 3/100
30/30 [==============================] - 0s 5ms/step - loss: 0.3487 - mean_squared_error: 0.3487 - val_loss: 0.3015 - val_mean_squared_error: 0.3015
Epoch 4/100
30/30 [==============================] - 0s 5ms/step - loss: 0.2437 - mean_squared_error: 0.2437 - val_loss: 0.2230 - val_mean_squared_error: 0.2230
Epoch 5/100
30/30 [==============================] - 0s 5ms/step - loss: 0.1897 - mean_squared_error: 0.1897 - val_loss: 0.1680 - val_mean_squared_error: 0.1680
Epoch 6/100
30/30 [==============================] - 0s 4ms/step - loss: 0.1539 - mean_squared_error: 0.1539 - val_loss: 0.1425 - val_mean_squared_error: 0.1425
Epoch 7/100
30/30 [==============================] - 0s 5ms/step - loss: 0.1368 - mean_squared_error: 0.1368 - val_loss: 0.1301 - val_mean_squared_error: 0.1301
Epoch 8/100
30/30 [==============================] - 0s 5ms/step - loss: 0.1346 - mean_squared_error: 0.1346 - val_loss: 0.1201 - val_mean_squared_error: 0.1201
Epoch 9/100
30/30 [==============================] - 0s 5ms/step - loss: 0.1207 - mean_squared_error: 0.1207 - val_loss: 0.1160 - val_mean_squared_error: 0.1160
Epoch 10/100
30/30 [==============================] - 0s 5ms/step - loss: 0.1209 - mean_squared_error: 0.1209 - val_loss: 0.1071 - val_mean_squared_error: 0.1071
Epoch 11/100
30/30 [==============================] - 0s 4ms/step - loss: 0.1140 - mean_squared_error: 0.1140 - val_loss: 0.1016 - val_mean_squared_error: 0.1016
Epoch 12/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0998 - mean_squared_error: 0.0998 - val_loss: 0.0997 - val_mean_squared_error: 0.0997
Epoch 13/100
30/30 [==============================] - 0s 4ms/step - loss: 0.1031 - mean_squared_error: 0.1031 - val_loss: 0.0938 - val_mean_squared_error: 0.0938
Epoch 14/100
30/30 [==============================] - 0s 5ms/step - loss: 0.0952 - mean_squared_error: 0.0952 - val_loss: 0.1171 - val_mean_squared_error: 0.1171
Epoch 15/100
30/30 [==============================] - 0s 5ms/step - loss: 0.0918 - mean_squared_error: 0.0918 - val_loss: 0.0898 - val_mean_squared_error: 0.0898
Epoch 16/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0887 - mean_squared_error: 0.0887 - val_loss: 0.0812 - val_mean_squared_error: 0.0812
Epoch 17/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0799 - mean_squared_error: 0.0799 - val_loss: 0.0857 - val_mean_squared_error: 0.0857
Epoch 18/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0823 - mean_squared_error: 0.0823 - val_loss: 0.0861 - val_mean_squared_error: 0.0861
Epoch 19/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0766 - mean_squared_error: 0.0766 - val_loss: 0.0805 - val_mean_squared_error: 0.0805
Epoch 20/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0713 - mean_squared_error: 0.0713 - val_loss: 0.0725 - val_mean_squared_error: 0.0725
```

```
Epoch 81/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0143 - mean_squared_error: 0.0143 - val_loss: 0.0199 - val_mean_squared_error: 0.0199
Epoch 82/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0151 - mean_squared_error: 0.0151 - val_loss: 0.0289 - val_mean_squared_error: 0.0289
Epoch 83/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0134 - mean_squared_error: 0.0134 - val_loss: 0.0167 - val_mean_squared_error: 0.0167
Epoch 84/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0128 - mean_squared_error: 0.0128 - val_loss: 0.0317 - val_mean_squared_error: 0.0317
Epoch 85/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0137 - mean_squared_error: 0.0137 - val_loss: 0.0196 - val_mean_squared_error: 0.0196
Epoch 86/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0130 - mean_squared_error: 0.0130 - val_loss: 0.0363 - val_mean_squared_error: 0.0363
Epoch 87/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0133 - mean_squared_error: 0.0133 - val_loss: 0.0232 - val_mean_squared_error: 0.0232
Epoch 88/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0126 - mean_squared_error: 0.0126 - val_loss: 0.0218 - val_mean_squared_error: 0.0218
Epoch 89/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0134 - mean_squared_error: 0.0134 - val_loss: 0.0186 - val_mean_squared_error: 0.0186
Epoch 90/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0107 - mean_squared_error: 0.0107 - val_loss: 0.0249 - val_mean_squared_error: 0.0249
Epoch 91/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0133 - mean_squared_error: 0.0133 - val_loss: 0.0169 - val_mean_squared_error: 0.0169
Epoch 92/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0120 - mean_squared_error: 0.0120 - val_loss: 0.0203 - val_mean_squared_error: 0.0203
Epoch 93/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0121 - mean_squared_error: 0.0121 - val_loss: 0.0168 - val_mean_squared_error: 0.0168
Epoch 94/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0119 - mean_squared_error: 0.0119 - val_loss: 0.0150 - val_mean_squared_error: 0.0150
Epoch 95/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0116 - mean_squared_error: 0.0116 - val_loss: 0.0143 - val_mean_squared_error: 0.0143
Epoch 96/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0115 - mean_squared_error: 0.0115 - val_loss: 0.0156 - val_mean_squared_error: 0.0156
Epoch 97/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0116 - mean_squared_error: 0.0116 - val_loss: 0.0116 - val_mean_squared_error: 0.0116
Epoch 98/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0110 - mean_squared_error: 0.0110 - val_loss: 0.0166 - val_mean_squared_error: 0.0166
Epoch 99/100
30/30 [==============================] - 0s 5ms/step - loss: 0.0088 - mean_squared_error: 0.0088 - val_loss: 0.0317 - val_mean_squared_error: 0.0317
Epoch 100/100
30/30 [==============================] - 0s 4ms/step - loss: 0.0120 - mean_squared_error: 0.0120 - val_loss: 0.0145 - val_mean_squared_error: 0.0145
<keras.callbacks.History at 0x7f4f64ff58d0>
```

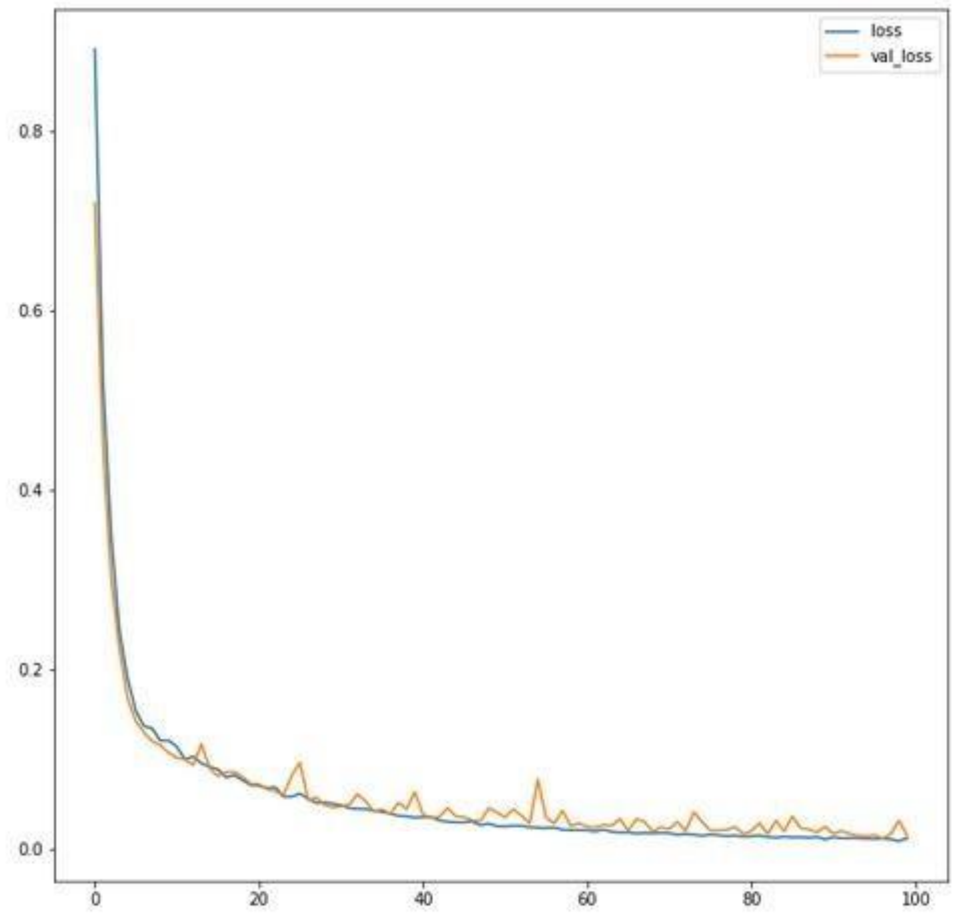
```
dfl3_RMS
```

	loss	mean_squared_error	val_loss	val_mean_squared_error
0	0.891632	0.891632	0.720253	0.720253
1	0.521454	0.521454	0.450020	0.450020
2	0.348655	0.348655	0.301520	0.301520
3	0.243745	0.243745	0.223027	0.223027
4	0.189683	0.189683	0.167990	0.167990
...	...	...	...	...
95	0.011464	0.011464	0.015609	0.015609
96	0.011643	0.011643	0.011588	0.011588
97	0.010951	0.010951	0.016629	0.016629
98	0.008824	0.008824	0.031710	0.031710
99	0.011976	0.011976	0.014527	0.014527

100 rows × 4 columns

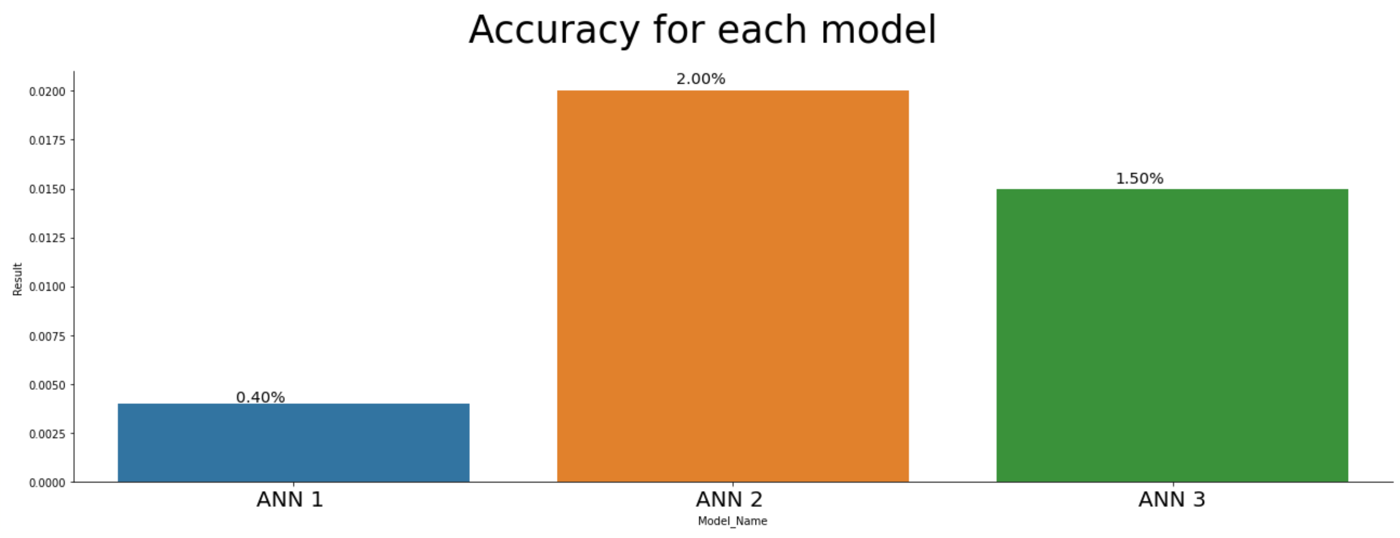
```
dfl3_RMS[['loss', 'val_loss']].plot(figsize=(10,10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4f64cb3f10>
```



```
# collecting the data in 3ed ANN model

ann_result[2]=np.round(ANN2.evaluate(X_test, y_test, verbose=0)[1], 3)

ann_result[2]
0.015
```

```
# we make a data frame for models results
df = pd.DataFrame(data=ann_model_name,columns=['Model_Name'])
df1 = pd.DataFrame(data=ann_result,columns=['Result'])
result = pd.concat([df,df1],axis=1)


g = sns.catplot(x='Model_Name', y='Result', data=result,
            height=6, aspect=3, kind='bar', legend=True)
g.fig.suptitle('Accuracy for each model', size=35, y=1.1)
ax = g.facet_axis(0,0)
ax.tick_params(axis='x', which='major', labelsize=20)
#=========================================================
# for printing percentage
for p in ax.patches:
   ax.text(p.get_x() + 0.27,
         p.get_height() * 1.02,
        '{0:.2f}%'.format(p.get_height()*100),
         color='black',
         rotation='horizontal',
         size='x-large')
```



There we Visualization which ANN model give the best result. There we see ANN 2 model give the best result in other 2 model.