Project1

November 13, 2021

1 Mercedes Benz Greener Manufacturing

In this project our main target is to reduce the time a Mercedes-Benz spends on the test bench. First we import all required libraries and datasets. Then we read, understand, clean the data and after that we analyze it.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
df_train=pd.read_csv("train.csv")
df_test=pd.read_csv("test.csv")
df_train.head()
```

[2]:
```
   ID       y  X0 X1  X2 X3 X4 X5 X6 X8  …  X375  X376  X377  X378  X379  \
0   0  130.81   k  v  at  a  d  u  j  o  …     0     0     1     0     0
1   6   88.53   k  t  av  e  d  y  l  o  …     1     0     0     0     0
2   7   76.26  az  w   n  c  d  x  j  x  …     0     0     0     0     0
3   9   80.62  az  t   n  f  d  x  l  e  …     0     0     0     0     0
4  13   78.02  az  v   n  f  d  h  d  n  …     0     0     0     0     0

   X380  X382  X383  X384  X385
0     0     0     0     0     0
1     0     0     0     0     0
2     0     1     0     0     0
3     0     0     0     0     0
4     0     0     0     0     0

[5 rows x 378 columns]
```

Here 'y' is our target variable and the rest are independent variables.

[3]:
```
df_test.head()
```

```
[3]:    ID  X0 X1  X2 X3 X4 X5 X6 X8  X10  …  X375  X376  X377  X378  X379  X380  \
     0   1  az  v   n  f  d  t  a  w    0  …     0     0     0     1     0     0
     1   2   t  b  ai  a  d  b  g  y    0  …     0     0     1     0     0     0
     2   3  az  v  as  f  d  a  j  j    0  …     0     0     0     1     0     0
     3   4  az  l   n  f  d  z  l  n    0  …     0     0     0     1     0     0
     4   5   w  s  as  c  d  y  i  m    0  …     1     0     0     0     0     0

        X382  X383  X384  X385
     0     0     0     0     0
     1     0     0     0     0
     2     0     0     0     0
     3     0     0     0     0
     4     0     0     0     0

     [5 rows x 377 columns]
```

In both datasets 'ID' column is not required, so we drop it.

```
[4]: df_train.drop(['ID'],axis=1,inplace=True)
     df_test.drop(['ID'],axis=1,inplace=True)
```

```
[5]: df_train.shape          # 4209 rows and 377 columns in trainning data
```

```
[5]: (4209, 377)
```

```
[6]: df_test.shape
```

```
[6]: (4209, 376)
```

```
[7]: # Check if there are any null values
     print(df_train.isnull().any().sum())
     print(df_test.isnull().any().sum())
```

```
0
0
```

Both values are zero means there is no null value in both datasets.

```
[8]: # Look at basic statistics of train data.
     df_train.describe().T
```

```
[8]:       count        mean        std    min    25%    50%     75%     max
     y    4209.0  100.669318  12.679381  72.11  90.82  99.15  109.01  265.32
     X10  4209.0    0.013305   0.114590   0.00   0.00   0.00    0.00    1.00
     X11  4209.0    0.000000   0.000000   0.00   0.00   0.00    0.00    0.00
     X12  4209.0    0.075077   0.263547   0.00   0.00   0.00    0.00    1.00
     X13  4209.0    0.057971   0.233716   0.00   0.00   0.00    0.00    1.00
     …       …         …          …      …     …     …      …       …
```

```
X380  4209.0  0.008078  0.089524  0.00  0.00  0.00  0.00  1.00
X382  4209.0  0.007603  0.086872  0.00  0.00  0.00  0.00  1.00
X383  4209.0  0.001663  0.040752  0.00  0.00  0.00  0.00  1.00
X384  4209.0  0.000475  0.021796  0.00  0.00  0.00  0.00  1.00
X385  4209.0  0.001426  0.037734  0.00  0.00  0.00  0.00  1.00

[369 rows x 8 columns]
```

[9]:

```python
# Check correlation of trainning data
df_train.corr()
```

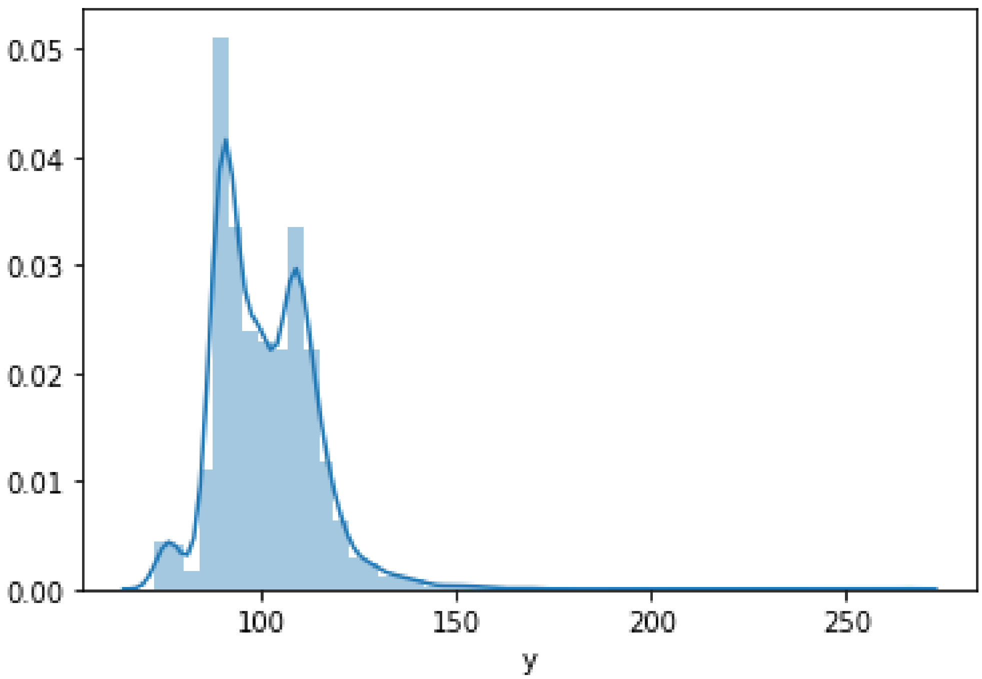
[9]:

```
             y       X10  X11       X12       X13       X14       X15  \
y     1.000000 -0.026985  NaN  0.089792  0.048276  0.193643  0.023116
X10  -0.026985  1.000000  NaN -0.033084 -0.028806 -0.100474 -0.002532
X11        NaN       NaN  NaN       NaN       NaN       NaN       NaN
X12   0.089792 -0.033084  NaN  1.000000  0.214825 -0.246513 -0.006212
X13   0.048276 -0.028806  NaN  0.214825  1.000000 -0.083141 -0.005409
…          …         …  …         …         …         …         …
X380  0.040932 -0.010479  NaN -0.005566  0.023045  0.007743 -0.001968
X382 -0.159815 -0.010164  NaN -0.024937 -0.021713  0.012713 -0.001908
X383  0.040291 -0.004740  NaN -0.011628 -0.010125  0.023604 -0.000890
X384 -0.004591 -0.002532  NaN -0.006212  0.041242  0.025199 -0.000475
X385 -0.022280 -0.004387  NaN -0.010765 -0.009373  0.043667 -0.000824

           X16       X17       X18  …      X375      X376      X377  \
y     0.048946 -0.159815 -0.001789  …  0.029100  0.114005  0.061403
X10  -0.005944 -0.010164 -0.010323  …  0.165277 -0.028618 -0.074244
X11        NaN       NaN       NaN  …       NaN       NaN       NaN
X12  -0.014584 -0.024937 -0.025327  … -0.107864 -0.070214  0.030134
X13  -0.012698 -0.021713 -0.010525  … -0.169721 -0.061136  0.357229
…          …         …         …  …         …         …         …
X380 -0.004619 -0.007899 -0.008022  … -0.061741 -0.022240 -0.061168
X382 -0.004480  1.000000  0.085256  … -0.059883 -0.021571 -0.059327
X383 -0.002089 -0.003572  0.062481  … -0.015413 -0.010059  0.035107
X384 -0.001116 -0.001908 -0.001938  … -0.014917 -0.005373  0.008694
X385 -0.001934 -0.003307 -0.003359  …  0.055225 -0.009311 -0.025610

          X378      X379      X380      X382      X383      X384      X385
y    -0.258679  0.067919  0.040932 -0.159815  0.040291 -0.004591 -0.022280
X10  -0.016870 -0.011374 -0.010479 -0.010164 -0.004740 -0.002532 -0.004387
X11        NaN       NaN       NaN       NaN       NaN       NaN       NaN
X12  -0.016043 -0.027907 -0.005566 -0.024937 -0.011628 -0.006212 -0.010765
X13  -0.036040 -0.024299  0.023045 -0.021713 -0.010125  0.041242 -0.009373
…          …         …         …         …         …         …         …
X380 -0.013110 -0.008839  1.000000 -0.007899 -0.003683 -0.001968 -0.003410
X382 -0.012716 -0.008573 -0.007899  1.000000 -0.003572 -0.001908 -0.003307
X383 -0.005930 -0.003998 -0.003683 -0.003572  1.000000 -0.000890 -0.001542
```

```
X384 -0.003168 -0.002136 -0.001968 -0.001908 -0.000890  1.000000 -0.000824
X385 -0.005489 -0.003701 -0.003410 -0.003307 -0.001542 -0.000824  1.000000

[369 rows x 369 columns]
```

[10]:
```
# Now  we split response and indenpendent variables. There is no y_test value␣
 ↪as we have to predict it.
X_train=df_train.drop(['y'],axis=1)
y_train=df_train['y']
X_test=df_test.iloc[:,:]
```

[11]:
```
# Distribution plot of y
sns.distplot(y_train)
plt.show()
```



In correlation matrix there are some nan values and also in description some columns have same max. and min. values that means some columns have constant values or only zero values, so we drop them.

[12]:
```
# X_train data
a=list()
for i in X_train.columns:
    if X_train[i].min() == X_train[i].max():
        a.append(i)
print(a)
```

```
X_train.drop(a,axis=1,inplace=True)
```

```
['X11', 'X93', 'X107', 'X233', 'X235', 'X268', 'X289', 'X290', 'X293', 'X297',
'X330', 'X347']
```

[13]:
```
# X_test data
b=list()
for i in X_test.columns:
    if X_test[i].min() == X_test[i].max():
        b.append(i)
print(b)
X_test.drop(a,axis=1,inplace=True)
```

```
['X257', 'X258', 'X295', 'X296', 'X369']
```

Now we encode object type columns 'X0', 'X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X8' as we do not want object type data.

[14]:
```
# Check unique categories of object type data.
print(X_train['X0'].unique())
print(X_test['X0'].unique())
```

```
['k' 'az' 't' 'al' 'o' 'w' 'j' 'h' 's' 'n' 'ay' 'f' 'x' 'y' 'aj' 'ak' 'am'
 'z' 'q' 'at' 'ap' 'v' 'af' 'a' 'e' 'ai' 'd' 'aq' 'c' 'aa' 'ba' 'as' 'i'
 'r' 'b' 'ax' 'bc' 'u' 'ad' 'au' 'm' 'l' 'aw' 'ao' 'ac' 'g' 'ab']
['az' 't' 'w' 'y' 'x' 'f' 'ap' 'o' 'ay' 'al' 'h' 'z' 'aj' 'd' 'v' 'ak'
 'ba' 'n' 'j' 's' 'af' 'ax' 'at' 'aq' 'av' 'm' 'k' 'a' 'e' 'ai' 'i' 'ag'
 'b' 'am' 'aw' 'as' 'r' 'ao' 'u' 'l' 'c' 'ad' 'au' 'bc' 'g' 'an' 'ae' 'p'
 'bb']
```

[15]:
```
# Check their length is same or not.
print(len(X_train['X0'].unique()))
print(len(X_test['X0'].unique()))
```

```
47
49
```

[16]:
```
# Check it for another column.
print(len(X_train['X2'].unique()))
print(len(X_test['X2'].unique()))
```

```
44
45
```

As object type columns have different categories so we have to encode them manually.

[17]:
```
variable=['X0','X1','X2','X3','X4','X5','X6','X8']
for i in variable:
    d=list(X_train[i].unique())
```

```
    f=list(X_test[i].unique())
    for j in f:
        if j not in d:
            d.append(j)
    enco=dict(zip(d,range(len(d))))
    X_train[i]=X_train[i].replace(enco)
    X_test[i]=X_test[i].replace(enco)
```

[18]:
```
X_train.head()
```

[18]:
```
   X0  X1  X2  X3  X4  X5  X6  X8  X10  X12  …  X375  X376  X377  X378  \
0   0   0   0   0   0   0   0   0    0    0  …     0     0     1     0
1   0   1   1   1   0   1   1   0    0    0  …     1     0     0     0
2   1   2   2   2   0   2   0   1    0    0  …     0     0     0     0
3   1   1   2   3   0   2   1   2    0    0  …     0     0     0     0
4   1   0   2   3   0   3   2   3    0    0  …     0     0     0     0

   X379  X380  X382  X383  X384  X385
0     0     0     0     0     0     0
1     0     0     0     0     0     0
2     0     0     1     0     0     0
3     0     0     0     0     0     0
4     0     0     0     0     0     0

[5 rows x 364 columns]
```

[19]:
```
X_test.head()
```

[19]:
```
   X0  X1  X2  X3  X4  X5  X6  X8  X10  X12  …  X375  X376  X377  X378  \
0   1   0   2   3   0  29   5  16    0    0  …     0     0     0     1
1   2   3   7   0   0  30   6  18    0    0  …     0     0     1     0
2   1   0   4   3   0  31   0  13    0    0  …     0     0     0     1
3   1   5   2   3   0  32   1   3    0    0  …     0     0     0     1
4   5   6   4   2   0   1   4   8    0    0  …     1     0     0     0

   X379  X380  X382  X383  X384  X385
0     0     0     0     0     0     0
1     0     0     0     0     0     0
2     0     0     0     0     0     0
3     0     0     0     0     0     0
4     0     0     0     0     0     0

[5 rows x 364 columns]
```

In both data there are too many columns so we apply Principal Component Analysis to extract important features.

[20]:
```
from sklearn.decomposition import PCA
```

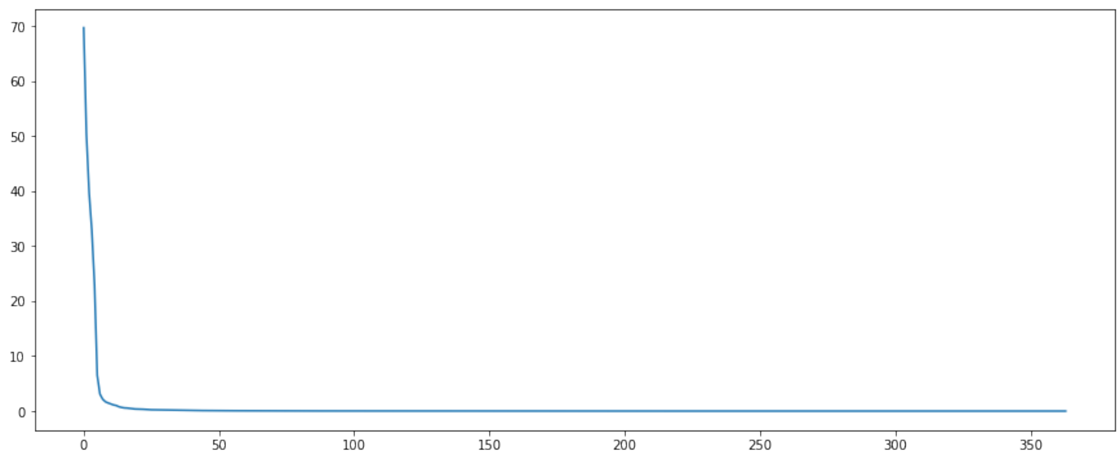
[21]:
```
# First we plot PCA with all components and see where we get smooth curve
pca = PCA(n_components=X_train.shape[1])
pca.fit(X_train)
```

[21]:
```
PCA(n_components=364)
```

[22]:
```
pca.n_components_
```

[22]:
```
364
```

[23]:
```
plt.figure(figsize = (15,6))
sns.lineplot(data=pca.explained_variance_)
plt.show()
# In this graph we get maximum information less than 25 components.
```



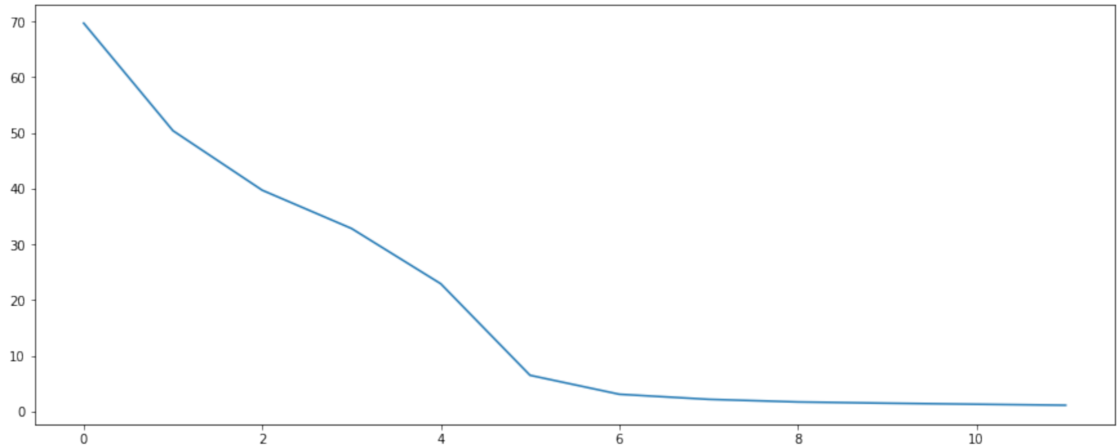
[24]:
```
# Now check where we get 95% variance explained.
pca = PCA(n_components = 0.95)
X_train_pca = pca.fit_transform(X_train)
print(X_train_pca.shape)
```

```
(4209, 12)
```

[25]:
```
X_test_pca = pca.transform(X_test)
print(X_test_pca.shape)
```

```
(4209, 12)
```

[26]:
```
plt.figure(figsize = (15,6))
sns.lineplot(data=pca.explained_variance_)
plt.show()
```



Above graph n=5 component has a elbow shape curve which explained maximum variance. So, we reduce PCA to 5 components.

[27]:
```
pca = PCA(n_components = 5)
X_train_pca = pca.fit_transform(X_train)
print(X_train_pca.shape)
```

```
(4209, 5)
```

[28]:
```
X_test_pca = pca.transform(X_test)
print(X_test_pca.shape)
```

```
(4209, 5)
```

Now data is clean, dimension is reduced and ready to fit in model. First we use eXtreme Gradient Boosting Regressor of ensemble method. Then we try to fit in another models.

[29]:
```
import xgboost
from sklearn.metrics import r2_score, mean_squared_error
```

[30]:
```
xgb_model = xgboost.XGBRegressor(n_estimators=1000)
xgb_model.fit(X_train_pca, y_train,eval_metric='rmse')
```

[30]:
```
XGBRegressor(base_score=0.5, booster=None, colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
             importance_type='gain', interaction_constraints=None,
             learning_rate=0.300000012, max_delta_step=0, max_depth=6,
             min_child_weight=1, missing=nan, monotone_constraints=None,
             n_estimators=1000, n_jobs=0, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method=None, validate_parameters=False, verbosity=None)
```

[31]:
```
xgb_pred = xgb_model.predict(X_train_pca)
```

[32]:
```
r2_score(y_pred = xgb_pred, y_true=y_train)  # r square value
```

[32]: 0.9713683061811162

[33]:
```
mean_squared_error(y_pred = xgb_pred, y_true=y_train)  # mse
```

[33]: 4.601929593445253

[34]:
```
mean_squared_error(y_pred = xgb_pred, y_true=y_train,squared=False)  # rmse
```

[34]: 2.145210850579787

Here XGB model fits very well and R square value is 0.97 (very high) that means it explained 97% variance of model. Mean square error and Root mean square error is 4.6 and 2.1 respectively, which means low error.

[35]:
```
# Now predict y_test
xgb_pred = xgb_model.predict(X_test_pca)
xgb_pred
```

[35]:
```
array([ 75.20987,  87.23393,  80.63685, …, 103.73881, 114.57336,
        93.11466], dtype=float32)
```

[36]:
```
y_test=pd.DataFrame(X_test_pca,xgb_pred)
```

[37]:
```
# Final result for X_test
y_test.head()
```

[37]:

	0	1	2	3	4
75.209869	-10.157275	7.308688	11.069408	-5.872025	-7.740356
87.233932	-8.533170	9.980976	11.856146	-1.535415	-4.083620
80.636848	-9.981792	5.078707	13.852080	-4.190252	-6.625144
77.718788	-10.507914	-4.100156	17.032831	-6.478453	-1.472961
110.676888	-8.827351	-6.798141	-14.186807	-2.166010	-0.546464

Now we try to fit the data in Linear Regression model.

[38]:
```
from sklearn.linear_model import LinearRegression
```

[39]:
```
lr_model=LinearRegression()
lr_model.fit(X_train,y_train)
```

[39]: LinearRegression()

[40]:
```
lr_pred=lr_model.predict(X_train)
```

[41]:
```
r2_score(y_pred = lr_pred, y_true=y_train)  # r square value
```

[41]: 0.5919691916499641

[42]:
```
mean_squared_error(y_pred = lr_pred, y_true=y_train)  # mse
```

[42]: 65.58218538733398

Here Linear Regression model does not fit well as R square value is very low and MSE is very high.

Lets try Ridge, Lasso and ElasticNet model.

[43]:
```
from sklearn.linear_model import Ridge, Lasso, ElasticNet
```

[44]:
```
# Ridge regression
ridge_model = Ridge(alpha=0.1)
ridge_model.fit(X_train,y_train)
```

[44]: Ridge(alpha=0.1)

[45]:
```
ridge_pred = ridge_model.predict(X_train)
```

[46]:
```
r2_score(y_pred = ridge_pred, y_true=y_train)  # r square value
```

[46]: 0.5918099470610183

[47]:
```
mean_squared_error(y_pred = ridge_pred, y_true=y_train)  # mse
```

[47]: 65.6077805334368

[48]:
```
# Lasso regression
lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train,y_train)
```

[48]: Lasso(alpha=0.1)

[49]:
```
lasso_pred = ridge_model.predict(X_train)
```

[50]:
```
r2_score(y_pred = lasso_pred, y_true=y_train)  # r square value
```

[50]: 0.5918099470610183

[51]:
```
mean_squared_error(y_pred = lasso_pred, y_true=y_train)  # mse
```

[51]: 65.6077805334368

[52]:
```
# ElasticNet Regression
enet_model = ElasticNet(alpha=0.1, l1_ratio=0.5)
```

```
enet_model.fit(X_train,y_train)
enet_pred = enet_model.predict(X_train)
```

[53]:
```
r2_score(y_pred = enet_pred, y_true=y_train)  # r square value
```

[53]: 0.5380441696500797

[54]:
```
mean_squared_error(y_pred = enet_pred, y_true=y_train)  # mse
```

[54]: 74.24947402691784

Here Ridge, Lasso and ElasticNet model does not fit well as R square value is very low and MSE is very high like LR model.

Till now XGBoost performs best. Let us try XGBoost model with Grid Search Cross Validation.

[55]:
```
from sklearn.model_selection import GridSearchCV
```

[56]:
```
param_grid = {'C': [0.1,1], 'gamma': [1,0.1]}
```

[57]:
```
xgb_grid=GridSearchCV(xgb_model,param_grid,refit=True,verbose=2)
```

[58]:
```
xgb_grid.fit(X_train_pca,y_train)
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
[CV] END …C=0.1, gamma=1; total time=   6.2s
[CV] END …C=0.1, gamma=1; total time=   6.0s
[CV] END …C=0.1, gamma=1; total time=   6.9s
[CV] END …C=0.1, gamma=1; total time=   6.3s
[CV] END …C=0.1, gamma=1; total time=   5.9s
[CV] END …C=0.1, gamma=0.1; total time=   6.9s
[CV] END …C=0.1, gamma=0.1; total time=   7.4s
[CV] END …C=0.1, gamma=0.1; total time=   6.2s
[CV] END …C=0.1, gamma=0.1; total time=   6.7s
[CV] END …C=0.1, gamma=0.1; total time=   6.3s
[CV] END …C=1, gamma=1; total time=   6.9s
[CV] END …C=1, gamma=1; total time=   6.3s
[CV] END …C=1, gamma=1; total time=   6.9s
[CV] END …C=1, gamma=1; total time=   6.7s
[CV] END …C=1, gamma=1; total time=   5.9s
[CV] END …C=1, gamma=0.1; total time=   6.3s
[CV] END …C=1, gamma=0.1; total time=   6.3s
[CV] END …C=1, gamma=0.1; total time=   6.2s
[CV] END …C=1, gamma=0.1; total time=   6.1s
[CV] END …C=1, gamma=0.1; total time=   6.6s
```

[58]:
```
GridSearchCV(estimator=XGBRegressor(base_score=0.5, booster=None,
                                    colsample_bylevel=1, colsample_bynode=1,
```

```
                                            colsample_bytree=1, gamma=0, gpu_id=-1,
                                            importance_type='gain',
                                            interaction_constraints=None,
                                            learning_rate=0.300000012, max_delta_step=0,
                                            max_depth=6, min_child_weight=1,
                                            missing=nan, monotone_constraints=None,
                                            n_estimators=1000, n_jobs=0,
                                            num_parallel_tree=1, random_state=0,
                                            reg_alpha=0, reg_lambda=1,
                                            scale_pos_weight=1, subsample=1,
                                            tree_method=None, validate_parameters=False,
                                            verbosity=None),
                       param_grid={'C': [0.1, 1], 'gamma': [1, 0.1]}, verbose=2)
```

[59]:
```
grid_predictions = xgb_grid.predict(X_train_pca)
```

[60]:
```
r2_score(y_train,grid_predictions)
```

[60]:
```
0.9664775171018932
```

[61]:
```
mean_squared_error(y_pred = grid_predictions, y_true=y_train)  # mse
```

[61]:
```
5.3880188531778055
```

[62]:
```
mean_squared_error(y_pred = grid_predictions, y_true=y_train,squared=False)  #
↪rmse
```

[62]:
```
2.3212106438619062
```

This model also performs well as R square value is 0.96 (very high) that means it explained 96.6% variance of model. Mean square error and Root mean square error is 5.3 and 2.3 respectively, which means low error.

[63]:
```
# Predict y_test with this model.
grid_pred = xgb_grid.predict(X_test_pca)
grid_pred
```

[63]:
```
array([ 80.864975,  90.027245,  82.5852  , …, 101.24793 , 115.40703 ,
        95.76787 ], dtype=float32)
```

[65]:
```
y_test2=pd.DataFrame(X_test_pca,grid_pred)
```

[66]:
```
y_test2.head()      # final result with Grid search cv
```

[66]:

	0	1	2	3	4
80.864975	-10.157275	7.308688	11.069408	-5.872025	-7.740356
90.027245	-8.533170	9.980976	11.856146	-1.535415	-4.083620
82.585197	-9.981792	5.078707	13.852080	-4.190252	-6.625144

```
75.685059  -10.507914 -4.100156  17.032831 -6.478453 -1.472961
107.953415  -8.827351 -6.798141 -14.186807 -2.166010 -0.546464
```

[67]:
```python
y_test.head()          # final result without Grid search cv
```

[67]:
```
                     0          1           2          3          4
75.209869  -10.157275  7.308688  11.069408 -5.872025 -7.740356
87.233932   -8.533170  9.980976  11.856146 -1.535415 -4.083620
80.636848   -9.981792  5.078707  13.852080 -4.190252 -6.625144
77.718788  -10.507914 -4.100156  17.032831 -6.478453 -1.472961
110.676888  -8.827351 -6.798141 -14.186807 -2.166010 -0.546464
```

Both results are very good we can use any one of them but without Grid search result is better from with grid search result.

[ ]:
```python
# END
```