aml-q1-correct-1

September 30, 2023

Importing Necessary Libraries

[95]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.preprocessing import PolynomialFeatures, StandardScaler,␣
↪OneHotEncoder
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.utils import shuffle
from sklearn.model_selection import cross_val_score, cross_val_predict,␣
↪cross_validate
from sklearn.linear_model import SGDRegressor
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
```

A.Summarize the data. How much data is present? What attributes/features are continuous valued? Which attributes are categorical?

[98]:
```
import warnings
warnings.filterwarnings("ignore")
```

[99]:
```
# Load the dataset
data = pd.read_csv("/content/happiness_data.csv")

# 1. Summarize the Data
# Display basic information about the dataset
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1949 entries, 0 to 1948
Data columns (total 11 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
```

```
 0   Country name                      1949 non-null   object 
 1   year                              1949 non-null   int64  
 2   Life Ladder                       1949 non-null   float64
 3   Log GDP per capita                1913 non-null   float64
 4   Social support                    1936 non-null   float64
 5   Healthy life expectancy at birth  1894 non-null   float64
 6   Freedom to make life choices      1917 non-null   float64
 7   Generosity                        1860 non-null   float64
 8   Perceptions of corruption         1839 non-null   float64
 9   Positive affect                   1927 non-null   float64
 10  Negative affect                   1933 non-null   float64
dtypes: float64(9), int64(1), object(1)
memory usage: 167.6+ KB
None
```

Displaying categorical and continuous attributes from the dataset

[100]:
```python
# Identify continuous and categorical attributes
categorical_attributes_func = data.select_dtypes(np.object)
continuous_attributes_func = data.select_dtypes(np.number)
```

[101]:
```python
continuous_attributes_func
```

[101]:
```
      year  Life Ladder  Log GDP per capita  Social support  \
0     2008        3.724               7.370           0.451   
1     2009        4.402               7.540           0.552   
2     2010        4.758               7.647           0.539   
3     2011        3.832               7.620           0.521   
4     2012        3.783               7.705           0.521   
…     …            …                   …              …   
1944  2016        3.735               7.984           0.768   
1945  2017        3.638               8.016           0.754   
1946  2018        3.616               8.049           0.775   
1947  2019        2.694               7.950           0.759   
1948  2020        3.160               7.829           0.717   

      Healthy life expectancy at birth  Freedom to make life choices  \
0                                50.80                         0.718   
1                                51.20                         0.679   
2                                51.60                         0.600   
3                                51.92                         0.496   
4                                52.24                         0.531   
…                                  …                          …   
1944                             54.40                         0.733   
1945                             55.00                         0.753   
1946                             55.60                         0.763   
1947                             56.20                         0.632   
1948                             56.80                         0.643   
```

```
      Generosity  Perceptions of corruption  Positive affect  Negative affect
0          0.168                      0.882            0.518            0.258
1          0.190                      0.850            0.584            0.237
2          0.121                      0.707            0.618            0.275
3          0.162                      0.731            0.611            0.267
4          0.236                      0.776            0.710            0.268
…            …                          …                …                …
1944      -0.095                      0.724            0.738            0.209
1945      -0.098                      0.751            0.806            0.224
1946      -0.068                      0.844            0.710            0.212
1947      -0.064                      0.831            0.716            0.235
1948      -0.009                      0.789            0.703            0.346

[1949 rows x 10 columns]
```

[102]:
```
categorical_attributes_func
```

[102]:
```
     Country name
0     Afghanistan
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
…             …
1944     Zimbabwe
1945     Zimbabwe
1946     Zimbabwe
1947     Zimbabwe
1948     Zimbabwe

[1949 rows x 1 columns]
```

[103]:
```
# by above observations, we defined these
continuous_attributes = ['Log GDP per capita', 'Social support', 'Freedom to␣
 ↪make life choices',
                         'Generosity', 'Perceptions of corruption', 'Positive␣
 ↪affect', 'Negative affect', 'Healthy life expectancy at birth','Life Ladder']
categorical_attributes = ['Country name']
```

OBSERVATIONS:

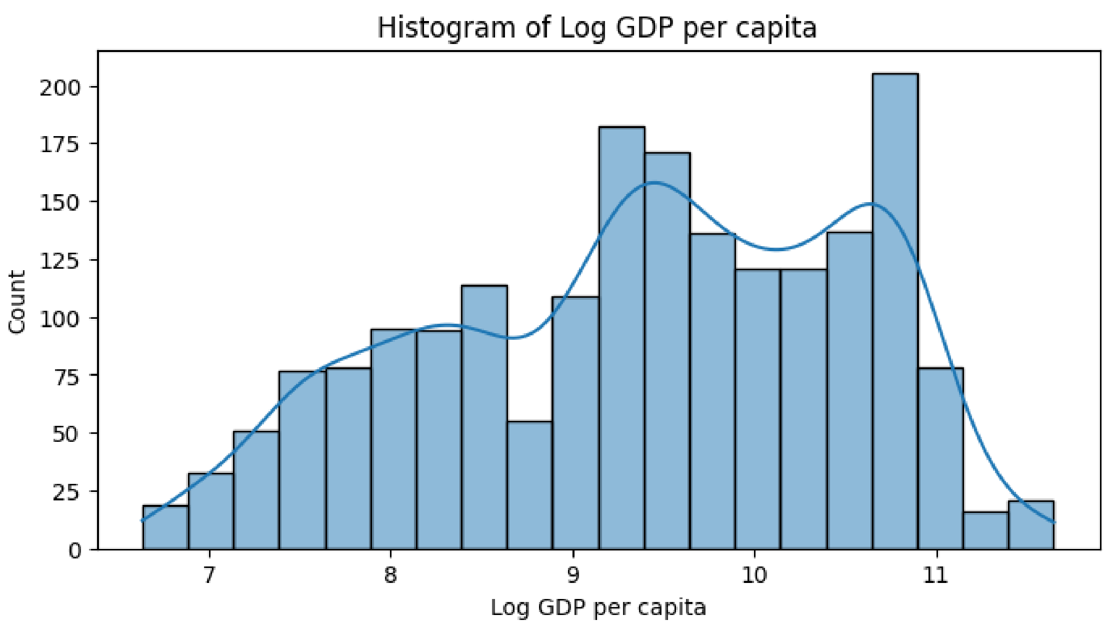
- Total entries = 1949
- Total features = 11
- Country name attribute has categorical values
- remaining all attributes has continuous values (ignoring year attribute)

B.Display the statistical values for each of the attributes, along with visualizations (e.g., histogram) of the distributions for each attribute. Explain noticeable traits for key attributes. Are there any attributes that might require special treatment? If so, what special treatment might they require?

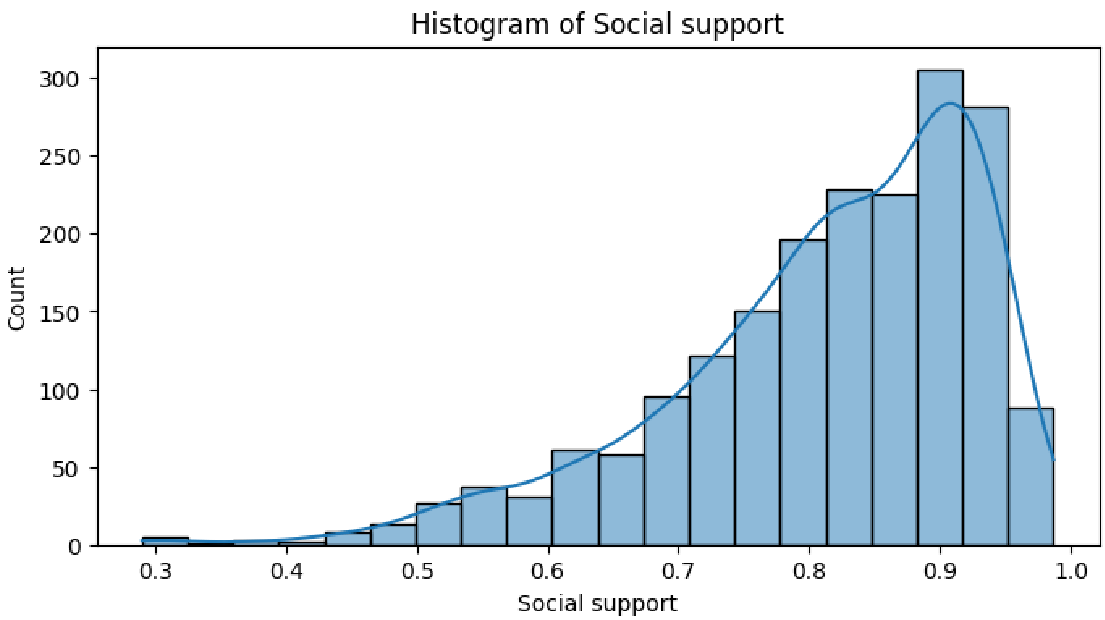
[104]:
```
# 2. Display Statistical Values and Visualizations
# Display statistical values for numerical attributes
print(data[continuous_attributes].info)
# Create histograms for numerical attributes
for attr in continuous_attributes:
    print("for attribute {} Mean : {}, median: {}, standard_deviation :{}" .
 ↪format(attr,data[attr].mean(),data[attr].median(),data[attr].std()))
    plt.figure(figsize=(8, 4))
    sns.histplot(data[attr], bins=20, kde=True)
    plt.title(f'Histogram of {attr}')
    plt.show()
```

```
<bound method DataFrame.info of       Log GDP per capita  Social support
Freedom to make life choices  \
0                   7.370           0.451                           0.718
1                   7.540           0.552                           0.679
2                   7.647           0.539                           0.600
3                   7.620           0.521                           0.496
4                   7.705           0.521                           0.531
…                   …            …                               …
1944                7.984           0.768                           0.733
1945                8.016           0.754                           0.753
1946                8.049           0.775                           0.763
1947                7.950           0.759                           0.632
1948                7.829           0.717                           0.643

      Generosity  Perceptions of corruption  Positive affect  Negative affect  \
0          0.168                      0.882            0.518            0.258
1          0.190                      0.850            0.584            0.237
2          0.121                      0.707            0.618            0.275
3          0.162                      0.731            0.611            0.267
4          0.236                      0.776            0.710            0.268
…          …                         …              …               …
1944      -0.095                      0.724            0.738            0.209
1945      -0.098                      0.751            0.806            0.224
1946      -0.068                      0.844            0.710            0.212
1947      -0.064                      0.831            0.716            0.235
1948      -0.009                      0.789            0.703            0.346

      Healthy life expectancy at birth  Life Ladder
0                                50.80        3.724
1                                51.20        4.402
2                                51.60        4.758
```

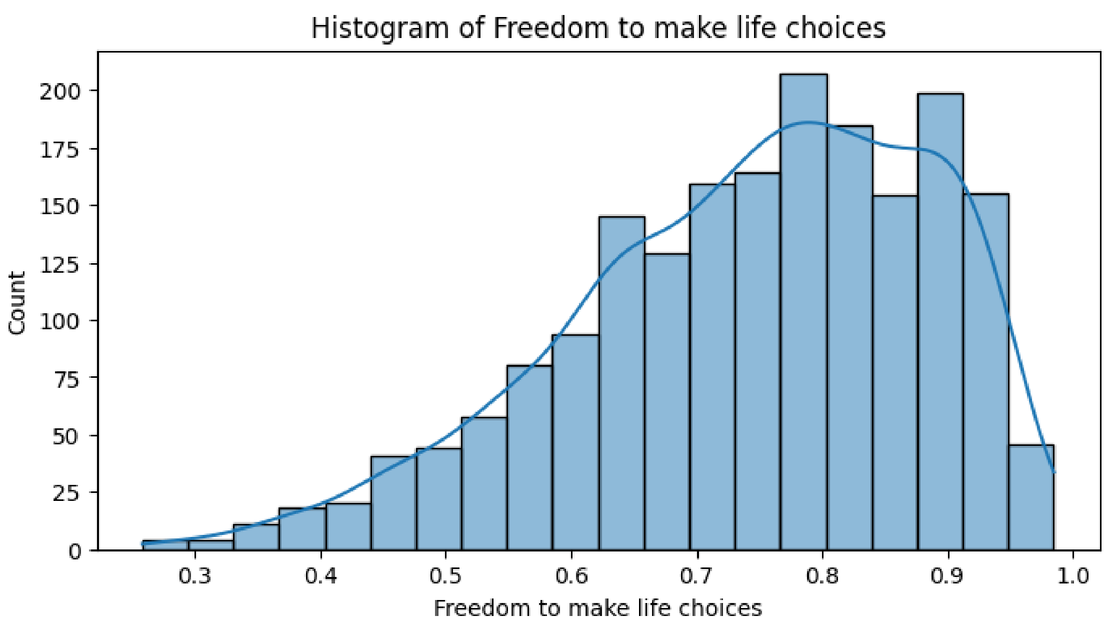
```
3                                 51.92        3.832
4                                 52.24        3.783
…                                 …         …
1944                              54.40        3.735
1945                              55.00        3.638
1946                              55.60        3.616
1947                              56.20        2.694
1948                              56.80        3.160

[1949 rows x 9 columns]>
for attribute Log GDP per capita Mean : 9.368452692106638, median: 9.46,
standard_deviation :1.154084029731952
```



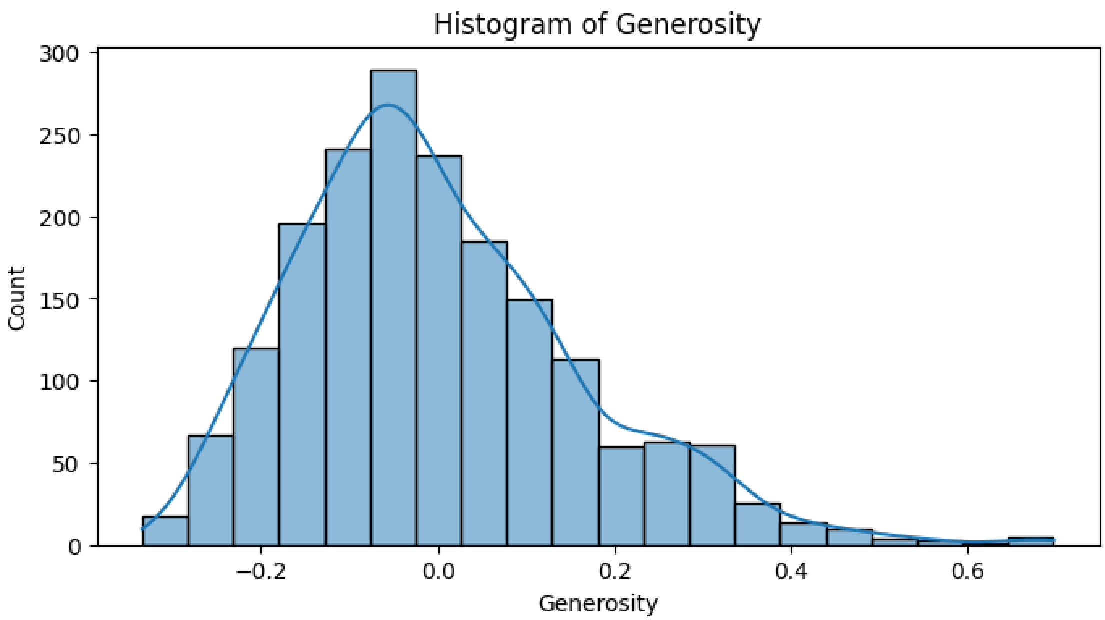

```
for attribute Social support Mean : 0.8125521694214877, median:
0.8354999999999999, standard_deviation :0.11848163156602372
```



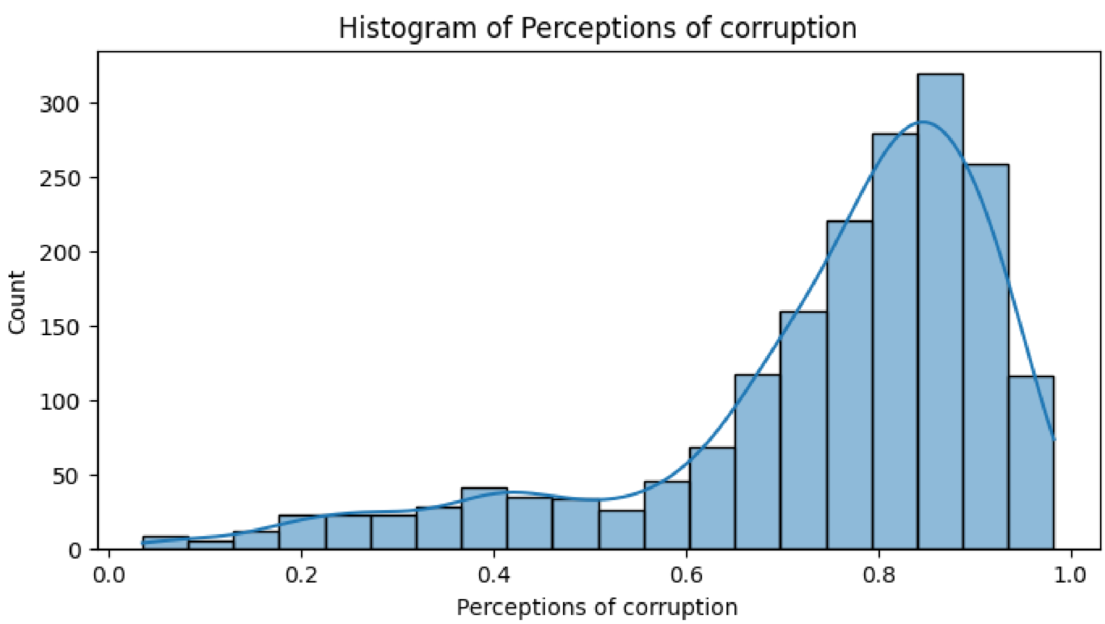
for attribute Freedom to make life choices Mean : 0.7425576421491914, median: 0.763, standard_deviation :0.14209286577975108



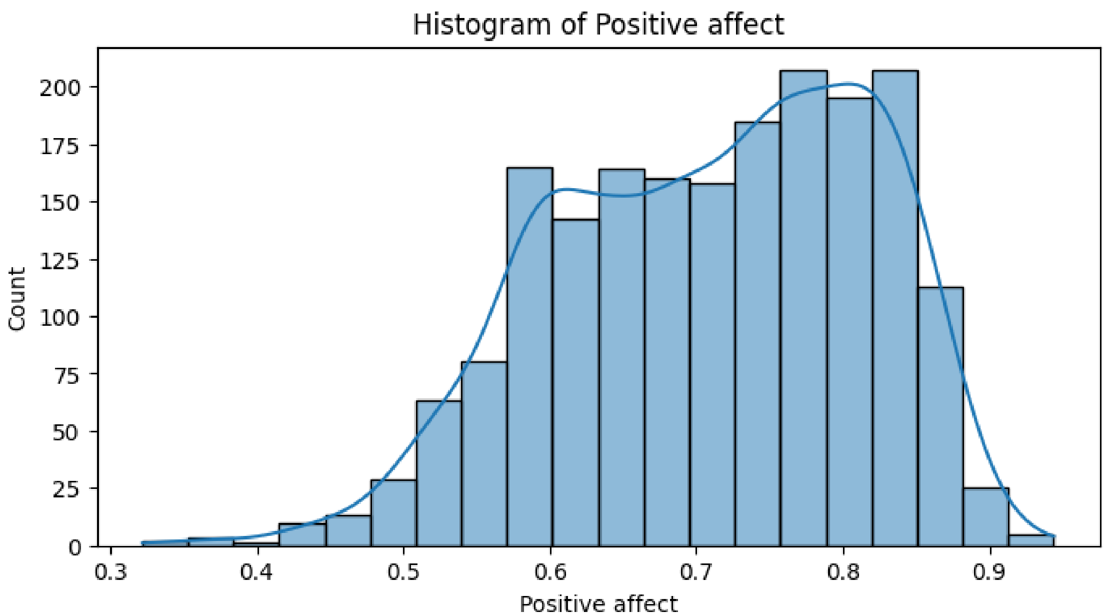
for attribute Generosity Mean : 0.00010322580645161109, median: -0.025500000000000002, standard_deviation :0.16221532880635953



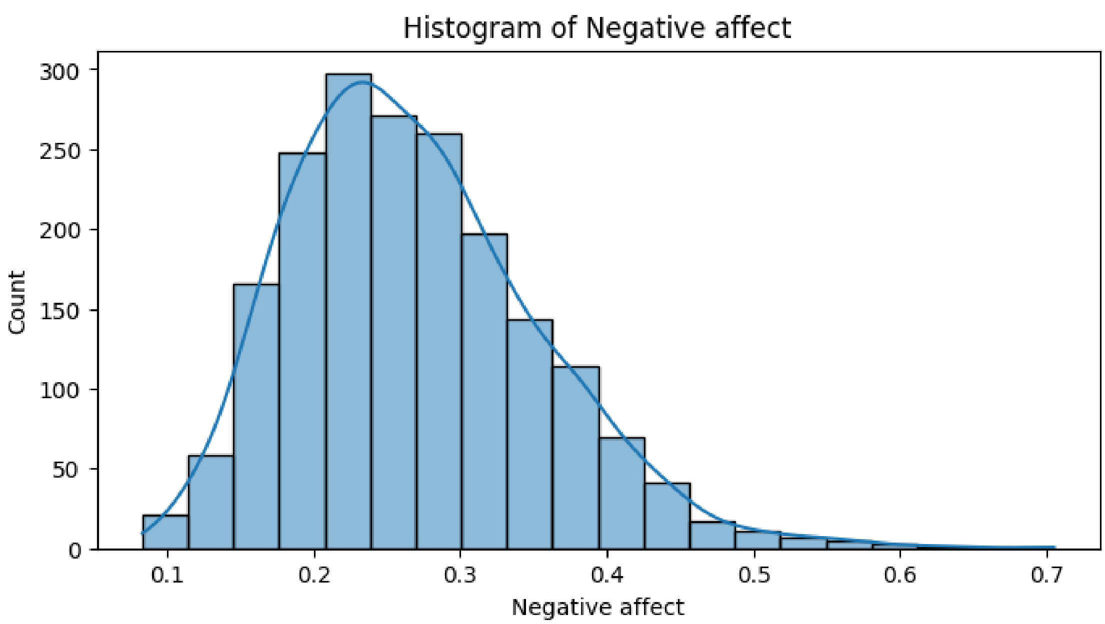
for attribute Perceptions of corruption Mean : 0.7471250679717237, median: 0.802, standard_deviation :0.18678881844350428



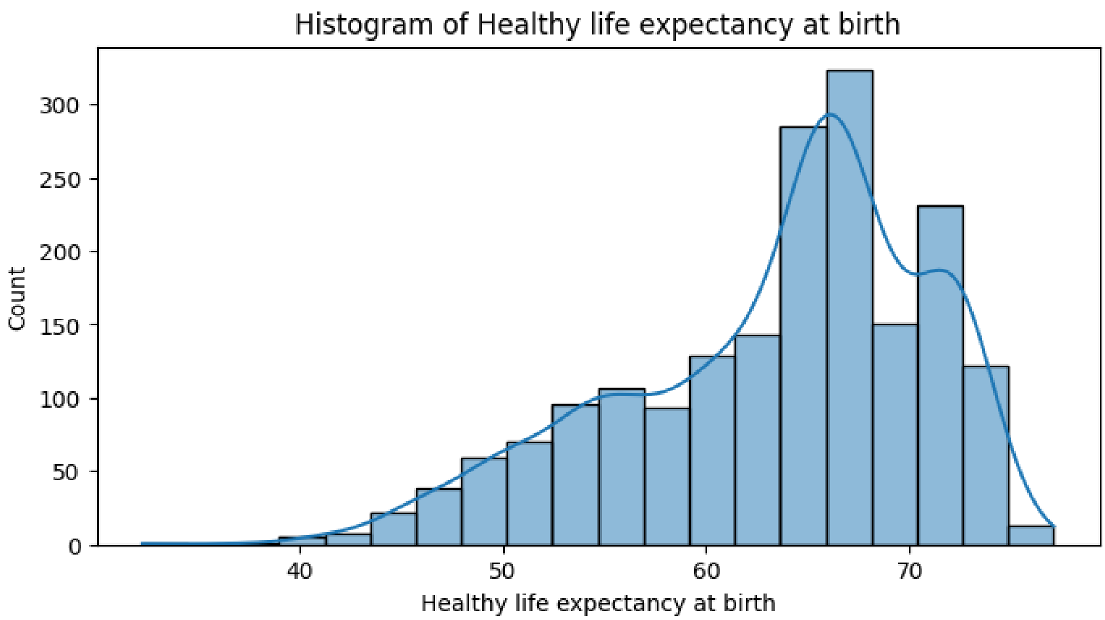
for attribute Positive affect Mean : 0.7100031136481577, median: 0.722, standard_deviation :0.10709993290814633



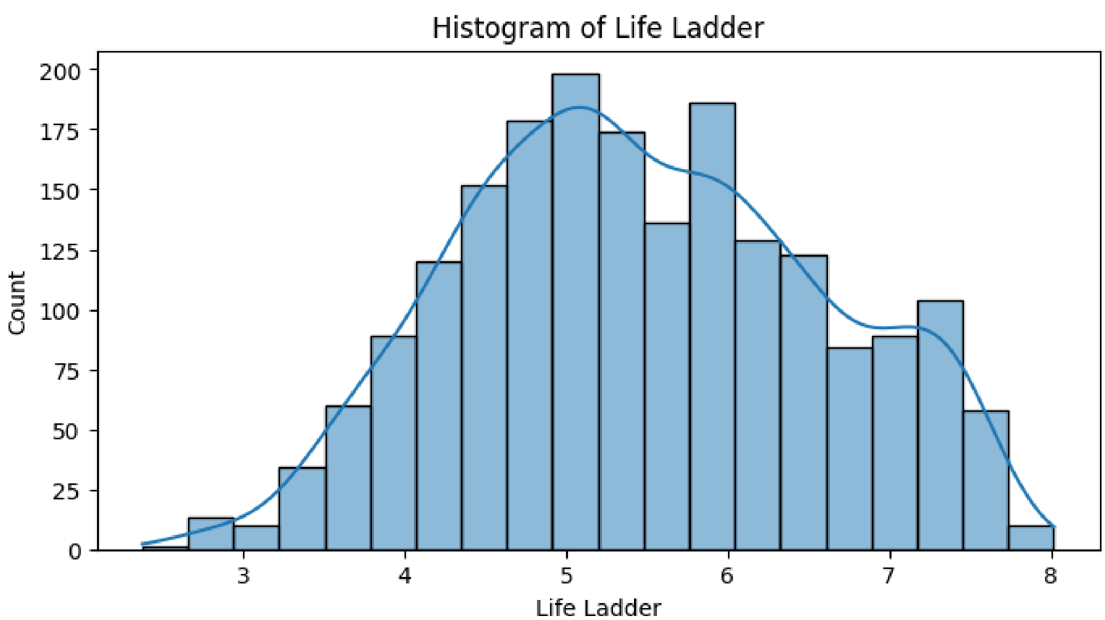
for attribute Negative affect Mean : 0.26854423176409725, median: 0.258, standard_deviation :0.08516806994884693



for attribute Healthy life expectancy at birth Mean : 63.35937381203802, median: 65.2, standard_deviation :7.51024461823635




```
for attribute Life Ladder Mean : 5.46670548999487, median: 5.386,
standard_deviation :1.1157105016473905
```



[105]:
```
#Display the statistical values for each of the attributes,
data[continuous_attributes].describe()
```

[105]:

```
       Log GDP per capita  Social support  Freedom to make life choices  \
count         1913.000000     1936.000000                   1917.000000
mean             9.368453        0.812552                      0.742558
std              1.154084        0.118482                      0.142093
min              6.635000        0.290000                      0.258000
25%              8.464000        0.749750                      0.647000
50%              9.460000        0.835500                      0.763000
75%             10.353000        0.905000                      0.856000
max             11.648000        0.987000                      0.985000

        Generosity  Perceptions of corruption  Positive affect  \
count  1860.000000                1839.000000      1927.000000
mean      0.000103                   0.747125         0.710003
std       0.162215                   0.186789         0.107100
min      -0.335000                   0.035000         0.322000
25%      -0.113000                   0.690000         0.625500
50%      -0.025500                   0.802000         0.722000
75%       0.091000                   0.872000         0.799000
max       0.698000                   0.983000         0.944000

       Negative affect  Healthy life expectancy at birth  Life Ladder
count      1933.000000                       1894.000000  1949.000000
mean          0.268544                         63.359374     5.466705
std           0.085168                          7.510245     1.115711
min           0.083000                         32.300000     2.375000
25%           0.206000                         58.685000     4.640000
50%           0.258000                         65.200000     5.386000
75%           0.320000                         68.590000     6.283000
max           0.705000                         77.100000     8.019000
```

we observed that attributes like social support, freedom to make life choices and percentage of corruption are left skewed and attributes like positive effects and log GDP show similar kind of traits in distribution.

1. We ignored Year and Life Ladder attributes
2. we will see any null values and replace them with median of the each feature
3. we will consider Life ladder as Label and remove it from the data.

C. Analyze and discuss the relationships between the data attributes, and between the data attributes and label. This involves computing the Pearson Correlation Coefficient (PCC) and generating scatter plots.

[106]:

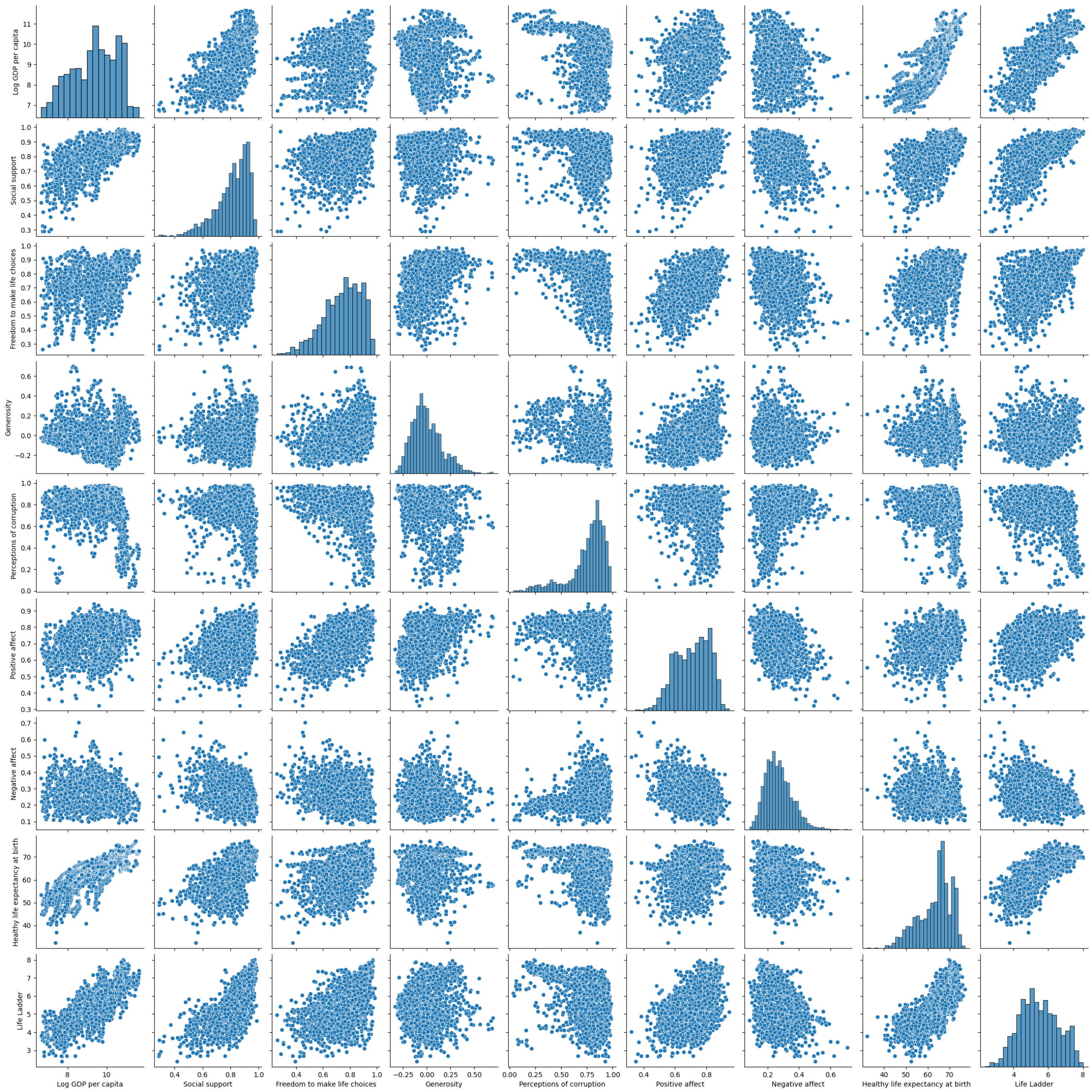
```python
# 3. Analyze Relationships
# Calculate Pearson Correlation Coefficient (PCC)
correlation_matrix = data[continuous_attributes].corr()
print(correlation_matrix)

# Generate scatter plots for key attribute pairs
```

```
sns.pairplot(data[continuous_attributes])
plt.show()
```

```
                                 Log GDP per capita  Social support  \
Log GDP per capita                         1.000000        0.692602
Social support                             0.692602        1.000000
Freedom to make life choices               0.367932        0.410402
Generosity                                -0.000915        0.067000
Perceptions of corruption                 -0.345511       -0.219040
Positive affect                            0.302282        0.432152
Negative affect                           -0.210781       -0.395865
Healthy life expectancy at birth           0.848049        0.616037
Life Ladder                                0.790166        0.707806

                                 Freedom to make life choices  Generosity  \
Log GDP per capita                                   0.367932   -0.000915
Social support                                       0.410402    0.067000
Freedom to make life choices                         1.000000    0.329300
Generosity                                           0.329300    1.000000
Perceptions of corruption                           -0.487883   -0.290706
Positive affect                                      0.606114    0.358006
Negative affect                                     -0.267661   -0.092542
Healthy life expectancy at birth                     0.388681    0.020737
Life Ladder                                          0.528063    0.190632

                                 Perceptions of corruption  Positive affect  \
Log GDP per capita                               -0.345511         0.302282
Social support                                   -0.219040         0.432152
Freedom to make life choices                     -0.487883         0.606114
Generosity                                       -0.290706         0.358006
Perceptions of corruption                         1.000000        -0.296517
Positive affect                                  -0.296517         1.000000
Negative affect                                   0.264225        -0.374439
Healthy life expectancy at birth                 -0.322461         0.318247
Life Ladder                                      -0.427245         0.532273

                                 Negative affect  \
Log GDP per capita                     -0.210781
Social support                         -0.395865
Freedom to make life choices           -0.267661
Generosity                             -0.092542
Perceptions of corruption               0.264225
Positive affect                        -0.374439
Negative affect                         1.000000
Healthy life expectancy at birth       -0.139477
Life Ladder                            -0.297488
```

	Healthy life expectancy at birth \
Log GDP per capita	0.848049
Social support	0.616037
Freedom to make life choices	0.388681
Generosity	0.020737
Perceptions of corruption	-0.322461
Positive affect	0.318247
Negative affect	-0.139477
Healthy life expectancy at birth	1.000000
Life Ladder	0.744506

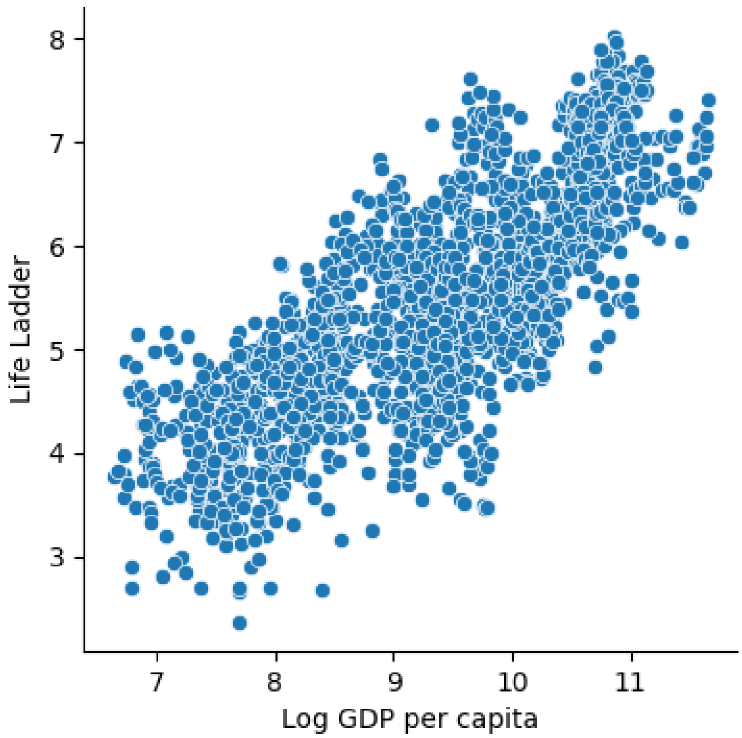
	Life Ladder
Log GDP per capita	0.790166
Social support	0.707806
Freedom to make life choices	0.528063
Generosity	0.190632
Perceptions of corruption	-0.427245
Positive affect	0.532273
Negative affect	-0.297488
Healthy life expectancy at birth	0.744506
Life Ladder	1.000000

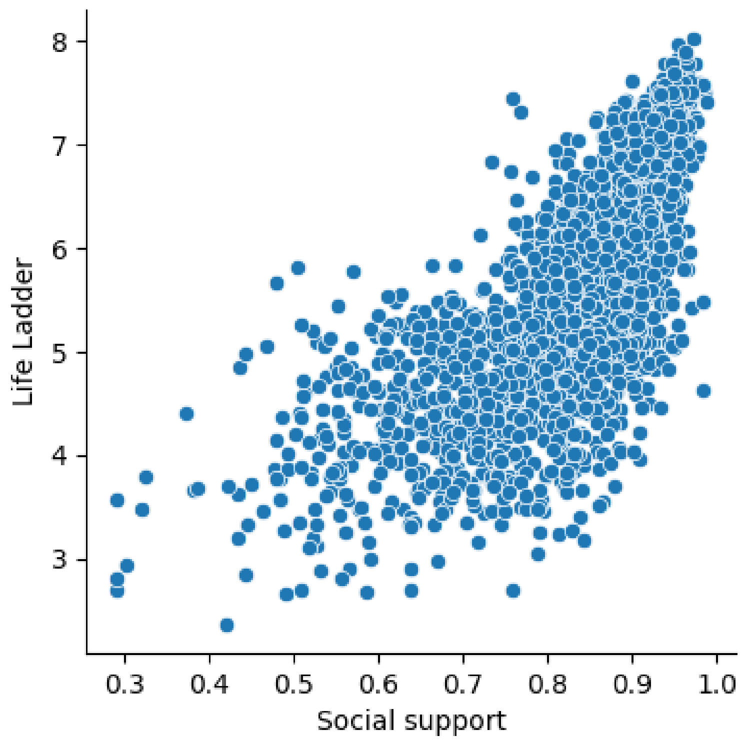


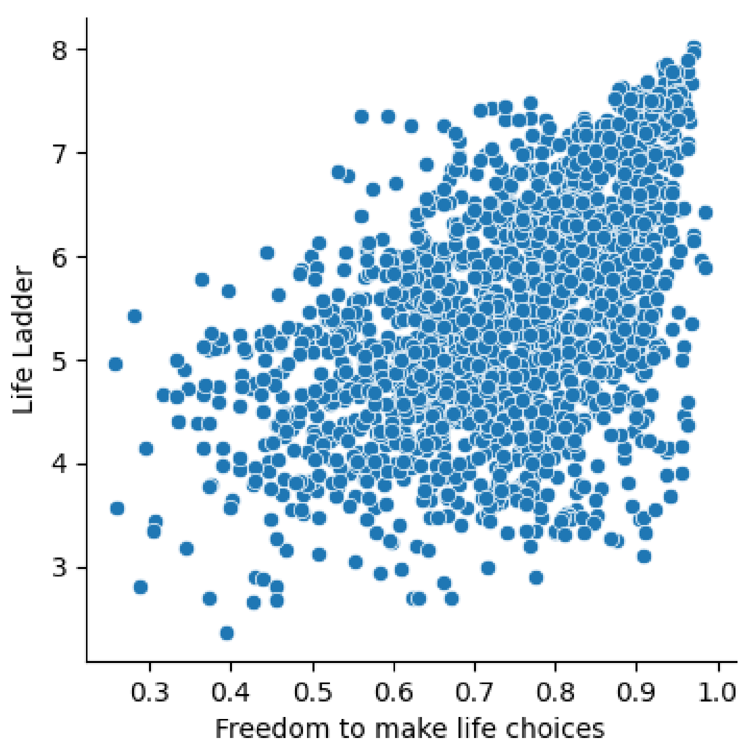
[107]:
```python
# between the Label and the each attribute
correlations_matrix_2 = (data).corr()
print(correlations_matrix_2["Life Ladder"].sort_values(ascending=False))



for attr in data[continuous_attributes]:
  sns.pairplot(data=data, x_vars=[attr], y_vars=["Life Ladder"],␣
 ↪kind="scatter", height=4)
  plt.show()
```

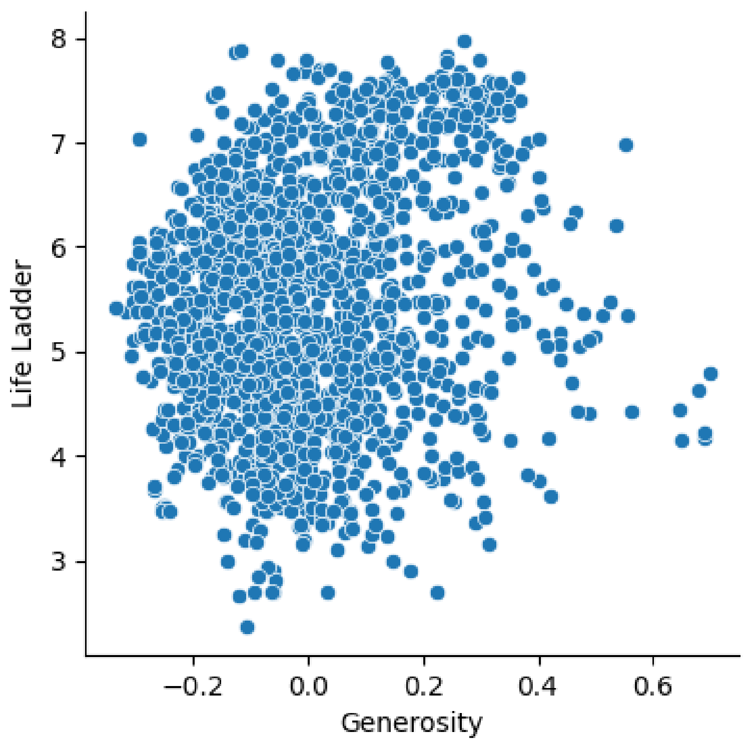
```
Life Ladder                         1.000000
Log GDP per capita                  0.790166
Healthy life expectancy at birth    0.744506
```

```
Social support                       0.707806
Positive affect                      0.532273
Freedom to make life choices         0.528063
Generosity                           0.190632
year                                 0.035515
Negative affect                     -0.297488
Perceptions of corruption           -0.427245
Name: Life Ladder, dtype: float64
```

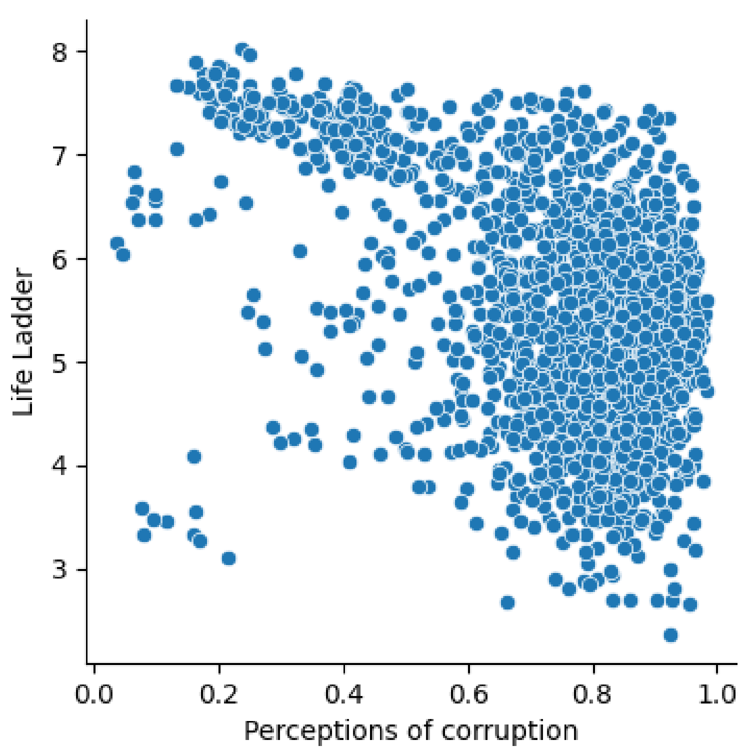



Life Ladder
Social support
8
7
6
5
4
3
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

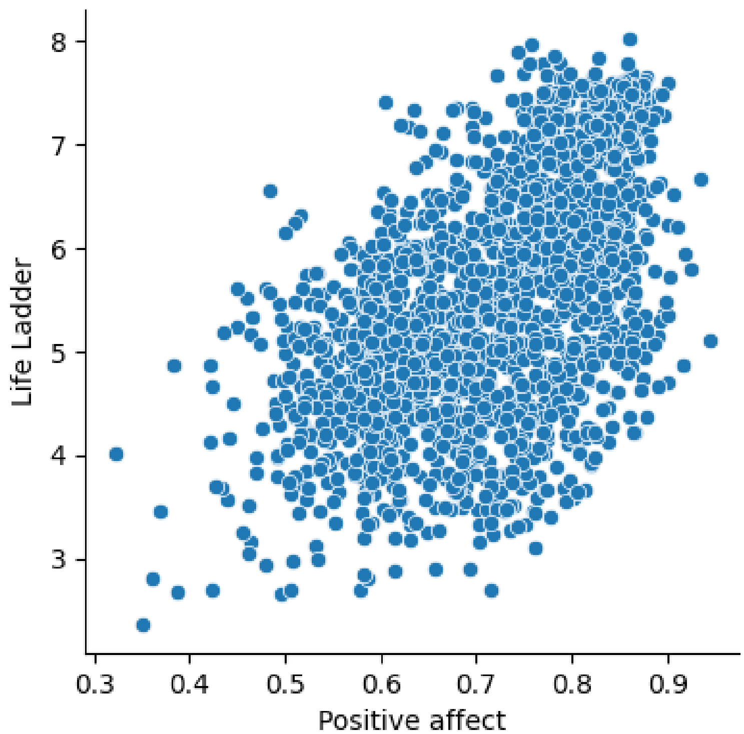

Life Ladder
Freedom to make life choices
8
7
6
5
4
3
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

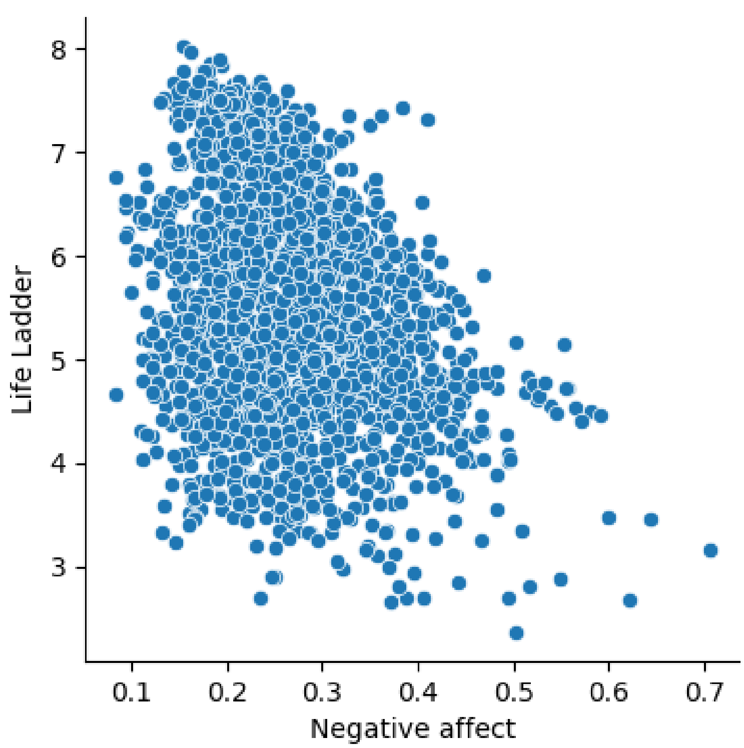


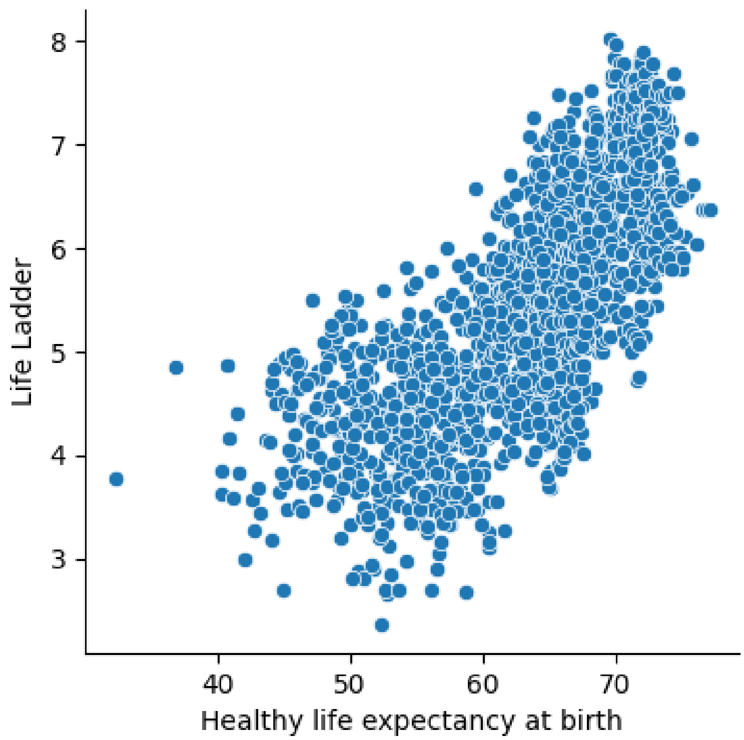
Life Ladder
Generosity



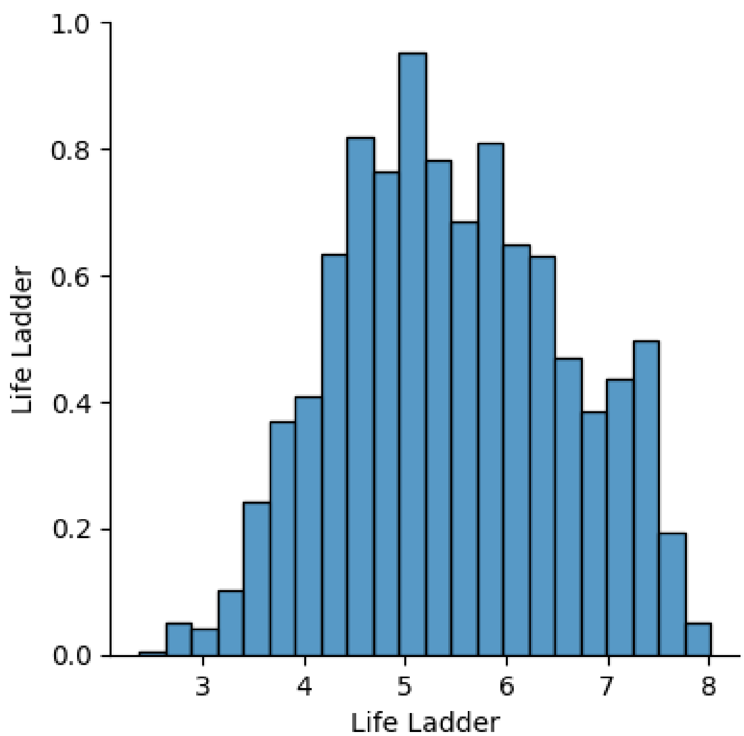
Life Ladder
Perceptions of corruption


Life Ladder
Positive affect
0.3
0.4
0.5
0.6
0.7
0.8
0.9
3
4
5
6
7
8


Life Ladder
Negative affect
0.1
0.2
0.3
0.4
0.5
0.6
0.7
3
4
5
6
7
8



8
7
6
5
4
3
Life Ladder
40
50
60
70
Healthy life expectancy at birth



1.0
0.8
0.6
0.4
0.2
0.0
Life Ladder
3
4
5
6
7
8
Life Ladder

- From the correlation table between the label and the attributes , we have observed that Log GDP per capita , Social support and Health life expectancy at birth are strongly correlated with each other.
- We can remove Generosity attribute and year attribute as the correlation coefficient is very weak with label.

[108]:
```
# drop weak correlated colums and label
data_num_updated=data[continuous_attributes]
data_num_updated.drop(columns=["Life Ladder","Generosity"],axis=1, inplace␣
↪=True)
```

[109]:
```
data_num_updated.head()
```

[109]:
```
   Log GDP per capita  Social support  Freedom to make life choices  \
0               7.370           0.451                         0.718
1               7.540           0.552                         0.679
2               7.647           0.539                         0.600
3               7.620           0.521                         0.496
4               7.705           0.521                         0.531

   Perceptions of corruption  Positive affect  Negative affect  \
0                      0.882            0.518            0.258
1                      0.850            0.584            0.237
2                      0.707            0.618            0.275
3                      0.731            0.611            0.267
4                      0.776            0.710            0.268

   Healthy life expectancy at birth
0                             50.80
1                             51.20
2                             51.60
3                             51.92
4                             52.24
```

[110]:
```
#dropping rows which have null value in the label
data =data.dropna(subset=['Life Ladder'])
```

Preprocessing the dataset

[111]:
```
continuous_attributes =['Log GDP per capita', 'Social support', 'Freedom to␣
↪make life choices',
                        'Perceptions of corruption', 'Positive affect',␣
↪'Negative affect', 'Healthy life expectancy at birth']
num_con_pipeline=␣
↪make_pipeline(StandardScaler(),SimpleImputer(strategy='median'))
```

[112]:
```
categorical_attributes = ['Country name']
cate_pipeline=␣
↪make_pipeline(SimpleImputer(strategy='most_frequent'),OneHotEncoder(handle_unknown="ignore"
```

[113]:
```
prep=␣
↪ColumnTransformer([("cont",num_con_pipeline,continuous_attributes),("cate",cate_pipeline,ca
```

[114]:
```
attri_prep= prep.fit_transform(data)
```

[115]:
```
attributes= pd.DataFrame(attri_prep,columns=prep.
↪get_feature_names_out(),index=data.index)
```

[116]:
```
y= pd.DataFrame(data['Life Ladder'])
```

[117]:
```
attributes.describe()#y.describe()
```

[117]:
```
       cont__Log GDP per capita  cont__Social support  \
count               1949.000000           1949.000000
mean                   0.001466              0.001292
std                    0.991033              0.997040
min                   -2.369123             -4.411546
25%                   -0.771768             -0.519642
50%                    0.079345              0.193733
75%                    0.837721              0.780473
max                    1.975717              1.472742

       cont__Freedom to make life choices  cont__Perceptions of corruption  \
count                         1949.000000                      1949.000000
mean                             0.002363                         0.016585
std                              0.992180                         0.973985
min                             -3.411037                        -3.813498
25%                             -0.658598                        -0.257714
50%                              0.143904                         0.293861
75%                              0.784497                         0.647297
max                              1.706670                         1.263133

       cont__Positive affect  cont__Negative affect  \
count            1949.000000            1949.000000
mean                0.001265              -0.001017
std                 0.994666               0.996205
min                -3.623754              -2.179129
25%                -0.775207              -0.722808
50%                 0.112045              -0.123837
75%                 0.821847               0.592579
max                 2.185413               5.125967

       cont__Healthy life expectancy at birth  cate__Country name_Afghanistan  \
```

```
count                           1949.000000                        1949.000000
mean                               0.006918                           0.006157
std                                0.986878                           0.078245
min                               -4.136693                           0.000000
25%                               -0.593929                           0.000000
50%                                0.245147                           0.000000
75%                                0.671344                           0.000000
max                                1.830068                           1.000000

       cate__Country name_Albania  cate__Country name_Algeria  …  \
count                 1949.000000                 1949.000000  …
mean                     0.006670                    0.004105  …
std                      0.081419                    0.063952  …
min                      0.000000                    0.000000  …
25%                      0.000000                    0.000000  …
50%                      0.000000                    0.000000  …
75%                      0.000000                    0.000000  …
max                      1.000000                    1.000000  …

       cate__Country name_United Arab Emirates  \
count                              1949.000000
mean                                  0.006670
std                                   0.081419
min                                   0.000000
25%                                   0.000000
50%                                   0.000000
75%                                   0.000000
max                                   1.000000

       cate__Country name_United Kingdom  cate__Country name_United States  \
count                        1949.000000                       1949.000000
mean                            0.007696                          0.007696
std                             0.087412                          0.087412
min                             0.000000                          0.000000
25%                             0.000000                          0.000000
50%                             0.000000                          0.000000
75%                             0.000000                          0.000000
max                             1.000000                          1.000000

       cate__Country name_Uruguay  cate__Country name_Uzbekistan  \
count                 1949.000000                    1949.000000
mean                     0.007696                       0.006670
std                      0.087412                       0.081419
min                      0.000000                       0.000000
25%                      0.000000                       0.000000
50%                      0.000000                       0.000000
75%                      0.000000                       0.000000
```

```
max                         1.000000                              1.000000

       cate__Country name_Venezuela  cate__Country name_Vietnam  \
count                   1949.000000                 1949.000000
mean                       0.007696                    0.007183
std                        0.087412                    0.084470
min                        0.000000                    0.000000
25%                        0.000000                    0.000000
50%                        0.000000                    0.000000
75%                        0.000000                    0.000000
max                        1.000000                    1.000000

       cate__Country name_Yemen  cate__Country name_Zambia  \
count               1949.000000                1949.000000
mean                   0.006157                   0.007183
std                    0.078245                   0.084470
min                    0.000000                   0.000000
25%                    0.000000                   0.000000
50%                    0.000000                   0.000000
75%                    0.000000                   0.000000
max                    1.000000                   1.000000

       cate__Country name_Zimbabwe
count                  1949.000000
mean                      0.007696
std                       0.087412
min                       0.000000
25%                       0.000000
50%                       0.000000
75%                       0.000000
max                       1.000000

[8 rows x 173 columns]
```

D.Select 20% of the data for testing. Describe how you did that and verify that your test portion of the data is representative of the entire dataset.

[118]:

```
# 4. Data Splitting
# Split data into training and testing sets


X_train, X_test, y_train, y_test = train_test_split(attributes, y, test_size=0.
 ↪2, random_state=42,shuffle=True)

#imputer = SimpleImputer(strategy='median')
#X_train = imputer.fit_transform(X_train_before_impute)
#X_test = imputer.transform(X_test_before_impute)
```

[119]:
```
# verification of  that our test portion of the data is representative of the␣
↪entire dataset.

attributes.describe()
#for attr in continuous_attributes:
#    print("For attribute in dataset {}: Mean: {}, Median: {}, Standard␣
↪Deviation: {}".format(attr, X[attr].mean(), X[attr].median(), X[attr].std()))
#    print("For attribute in test data {}: Mean: {}, Median: {}, Standard␣
↪Deviation: {}".format(attr, X_test[attr].mean(), X_test[attr].median(),␣
↪X_test[attr].std()))
# Assuming you've already split the dataset and imputed missing values as you␣
↪mentioned

# for attr in continuous_attributes:
#     attr_index = continuous_attributes.index(attr)  # Get the index of the␣
↪attribute
#     print("For attribute in dataset {}: Mean: {:.2f}, Median: {:.2f},␣
↪Standard Deviation: {:.2f}".format(attr, X[:, attr_index].mean(), np.
↪median(X[:, attr_index], axis=0), X[:, attr_index].std()))
#     print("For attribute in test data {}: Mean: {:.2f}, Median: {:.2f},␣
↪Standard Deviation: {:.2f}".format(attr, X_test[:, attr_index].mean(), np.
↪median(X_test[:, attr_index], axis=0), X_test[:, attr_index].std()))
```

[119]:
```
       cont__Log GDP per capita  cont__Social support  \
count               1949.000000           1949.000000
mean                   0.001466              0.001292
std                    0.991033              0.997040
min                   -2.369123             -4.411546
25%                   -0.771768             -0.519642
50%                    0.079345              0.193733
75%                    0.837721              0.780473
max                    1.975717              1.472742

       cont__Freedom to make life choices  cont__Perceptions of corruption  \
count                         1949.000000                      1949.000000
mean                             0.002363                         0.016585
std                              0.992180                         0.973985
min                             -3.411037                        -3.813498
25%                             -0.658598                        -0.257714
50%                              0.143904                         0.293861
75%                              0.784497                         0.647297
max                              1.706670                         1.263133

       cont__Positive affect  cont__Negative affect  \
count            1949.000000            1949.000000
mean                0.001265              -0.001017
std                 0.994666               0.996205
```

```
min               -3.623754              -2.179129
25%               -0.775207              -0.722808
50%                0.112045              -0.123837
75%                0.821847               0.592579
max                2.185413               5.125967

       cont__Healthy life expectancy at birth  cate__Country name_Afghanistan  \
count                             1949.000000                     1949.000000
mean                                 0.006918                        0.006157
std                                  0.986878                        0.078245
min                                 -4.136693                        0.000000
25%                                 -0.593929                        0.000000
50%                                  0.245147                        0.000000
75%                                  0.671344                        0.000000
max                                  1.830068                        1.000000

       cate__Country name_Albania  cate__Country name_Algeria  …  \
count                 1949.000000                 1949.000000  …
mean                     0.006670                    0.004105  …
std                      0.081419                    0.063952  …
min                      0.000000                    0.000000  …
25%                      0.000000                    0.000000  …
50%                      0.000000                    0.000000  …
75%                      0.000000                    0.000000  …
max                      1.000000                    1.000000  …

       cate__Country name_United Arab Emirates  \
count                              1949.000000
mean                                  0.006670
std                                   0.081419
min                                   0.000000
25%                                   0.000000
50%                                   0.000000
75%                                   0.000000
max                                   1.000000

       cate__Country name_United Kingdom  cate__Country name_United States  \
count                        1949.000000                       1949.000000
mean                            0.007696                          0.007696
std                             0.087412                          0.087412
min                             0.000000                          0.000000
25%                             0.000000                          0.000000
50%                             0.000000                          0.000000
75%                             0.000000                          0.000000
max                             1.000000                          1.000000

       cate__Country name_Uruguay  cate__Country name_Uzbekistan  \
```

```
count                  1949.000000                  1949.000000
mean                      0.007696                     0.006670
std                       0.087412                     0.081419
min                       0.000000                     0.000000
25%                       0.000000                     0.000000
50%                       0.000000                     0.000000
75%                       0.000000                     0.000000
max                       1.000000                     1.000000

       cate__Country name_Venezuela  cate__Country name_Vietnam  \
count                   1949.000000                 1949.000000
mean                       0.007696                    0.007183
std                        0.087412                    0.084470
min                        0.000000                    0.000000
25%                        0.000000                    0.000000
50%                        0.000000                    0.000000
75%                        0.000000                    0.000000
max                        1.000000                    1.000000

       cate__Country name_Yemen  cate__Country name_Zambia  \
count               1949.000000                1949.000000
mean                   0.006157                   0.007183
std                    0.078245                   0.084470
min                    0.000000                   0.000000
25%                    0.000000                   0.000000
50%                    0.000000                   0.000000
75%                    0.000000                   0.000000
max                    1.000000                   1.000000

       cate__Country name_Zimbabwe
count                  1949.000000
mean                      0.007696
std                       0.087412
min                       0.000000
25%                       0.000000
50%                       0.000000
75%                       0.000000
max                       1.000000

[8 rows x 173 columns]
```

[120]:
```
X_test.describe()
```

[120]:
```
       cont__Log GDP per capita  cont__Social support  \
count                390.000000            390.000000
mean                   0.020627             -0.003114
std                    0.966890              0.988339
```

```
min                  -2.292852             -4.107623
25%                  -0.745117             -0.591402
50%                   0.080645              0.193733
75%                   0.887123              0.793136
max                   1.864777              1.371435

       cont__Freedom to make life choices  cont__Perceptions of corruption  \
count                          390.000000                       390.000000
mean                             0.022427                        -0.004638
std                              0.982008                         0.974591
min                             -3.411037                        -3.583229
25%                             -0.574124                        -0.272441
50%                              0.143904                         0.293861
75%                              0.788017                         0.635248
max                              1.671472                         1.263133

       cont__Positive affect  cont__Negative affect  \
count             390.000000             390.000000
mean                0.055050               0.041249
std                 0.957890               1.031655
min                -3.184798              -1.944239
25%                -0.688817              -0.731616
50%                 0.112045              -0.123837
75%                 0.875549               0.624877
max                 1.783815               4.397806

       cont__Healthy life expectancy at birth  cate__Country name_Afghanistan  \
count                              390.000000                           390.0
mean                                 0.025709                             0.0
std                                  0.996027                             0.0
min                                 -3.003541                             0.0
25%                                 -0.490043                             0.0
50%                                  0.245147                             0.0
75%                                  0.681333                             0.0
max                                  1.550376                             0.0

       cate__Country name_Albania  cate__Country name_Algeria  …  \
count                  390.000000                  390.000000  …
mean                     0.002564                    0.007692  …
std                      0.050637                    0.087480  …
min                      0.000000                    0.000000  …
25%                      0.000000                    0.000000  …
50%                      0.000000                    0.000000  …
75%                      0.000000                    0.000000  …
max                      1.000000                    1.000000  …

       cate__Country name_United Arab Emirates  \
```

count	390.000000
mean	0.005128
std	0.071519
min	0.000000
25%	0.000000
50%	0.000000
75%	0.000000
max	1.000000

	cate__Country name_United Kingdom	cate__Country name_United States	\
count	390.000000	390.000000	
mean	0.005128	0.002564	
std	0.071519	0.050637	
min	0.000000	0.000000	
25%	0.000000	0.000000	
50%	0.000000	0.000000	
75%	0.000000	0.000000	
max	1.000000	1.000000	

	cate__Country name_Uruguay	cate__Country name_Uzbekistan	\
count	390.000000	390.000000	
mean	0.007692	0.010256	
std	0.087480	0.100883	
min	0.000000	0.000000	
25%	0.000000	0.000000	
50%	0.000000	0.000000	
75%	0.000000	0.000000	
max	1.000000	1.000000	

	cate__Country name_Venezuela	cate__Country name_Vietnam	\
count	390.000000	390.0	
mean	0.015385	0.0	
std	0.123235	0.0	
min	0.000000	0.0	
25%	0.000000	0.0	
50%	0.000000	0.0	
75%	0.000000	0.0	
max	1.000000	0.0	

	cate__Country name_Yemen	cate__Country name_Zambia	\
count	390.000000	390.000000	
mean	0.005128	0.007692	
std	0.071519	0.087480	
min	0.000000	0.000000	
25%	0.000000	0.000000	
50%	0.000000	0.000000	
75%	0.000000	0.000000	

```
max                       1.000000                    1.000000

       cate__Country name_Zimbabwe
count                   390.000000
mean                      0.012821
std                       0.112644
min                       0.000000
25%                       0.000000
50%                       0.000000
75%                       0.000000
max                       1.000000

[8 rows x 173 columns]
```

[121]:
```
#y_train=pd.DataFrame(y_train)
y_train.describe()
```

[121]:
```
       Life Ladder
count  1559.000000
mean      5.469321
std       1.113280
min       2.375000
25%       4.649500
50%       5.374000
75%       6.272500
max       8.019000
```

If we look at the data given by test describe , the mean , median and standard deviation and the quartile range looks similiar. This means test portion of thee data is the representative of the entire dataset.

E.Train a Linear Regression model using the training data with four-fold cross-validation using appropriate evaluation metric. Do this with a closed-form solution (using the Normal Equation or SVD) and with SGD. Perform Ridge, Lasso and Elastic Net regularization – try a few values of penalty term and describe its impact. Explore the impact of other hyperparameters, like batch size and learning rate (no need for grid search). Describe your findings. For SGD, display the training and validation loss as a function of training iteration.

Linear Regression

[122]:
```
linear=LinearRegression().fit(X_train,y_train)
linear_cv=cross_validate(linear,X_train,y_train,scoring=['neg_root_mean_squared_error'],cv=4,r
```

[123]:
```
print("training loss : {:,.3f}".format(-np.
 ↪mean(linear_cv['train_neg_root_mean_squared_error'])))
print("validation loss : {:,.3f}".format(-np.
 ↪mean(linear_cv['test_neg_root_mean_squared_error'])))
```

```
training loss : 0.328
validation loss : 94,289,888,075.400
```

SGD

[124]:
```
sgd=SGDRegressor(max_iter=1000, tol=1e-5,eta0=0.
↪01,n_iter_no_change=100,random_state=42)
sgd.fit(X_train,y_train)
sgd_cv=cross_validate(sgd,X_train,y_train,scoring=['neg_root_mean_squared_error'],cv=4,return_
```

[125]:
```
print("training loss : {:,.3f}".format(-np.
↪mean(sgd_cv['train_neg_root_mean_squared_error'])))
print("Validation loss : {:,.3f}".format(-np.
↪mean(sgd_cv['test_neg_root_mean_squared_error'])))
```

```
training loss : 0.338
Validation loss : 0.394
```

SGD, display the training and validation loss as a function of training iteration.

[126]:
```
t_loss=[]
v_loss=[]
for i in range(1,1001,100):
  sgd1=SGDRegressor(max_iter=i, tol=1e-5,eta0=0.
↪01,n_iter_no_change=100,random_state=42)
  sgd1.fit(X_train,y_train)
␣
↪sgd1_cv=cross_validate(sgd1,X_train,y_train,scoring=['neg_root_mean_squared_error'],cv=4,ret
  t_loss.append(-np.mean(sgd1_cv['train_neg_root_mean_squared_error']))
  v_loss.append(-np.mean(sgd1_cv['test_neg_root_mean_squared_error']))
plt.figure(figsize=(10, 6))
plt.plot(range(1, 1000 + 1,100), t_loss, label='Training Loss')
plt.plot(range(1, 1000 + 1,100), v_loss, label='Validation Loss')
plt.xlabel('Training Iteration')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss vs. Training Iteration')
plt.grid(True)
plt.show()
```

Perform Ridge, Lasso and Elastic Net regularization – try a few values of penalty term and describe its impact.

[127]:
```
# Define a range of alpha (penalty term) values to explore
alphas = [0.01, 0.1, 1.0, 10.0]

# Initialize lists to store results
ridge_results = []
lasso_results = []
elastic_net_results = []

# Ridge Regression
for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    scores = cross_val_score(ridge, X_train, y_train, cv=4,␣
 ↪scoring='neg_mean_squared_error')
    rmse_scores = np.sqrt(-scores)
    rmse_mean = rmse_scores.mean()
    ridge_results.append({'Alpha': alpha, 'RMSE Mean': rmse_mean})

# Lasso Regression
for alpha in alphas:
    lasso = Lasso(alpha=alpha)
```

```python
    scores = cross_val_score(lasso, X_train, y_train, cv=4,
↪scoring='neg_mean_squared_error')
    rmse_scores = np.sqrt(-scores)
    rmse_mean = rmse_scores.mean()
    lasso_results.append({'Alpha': alpha, 'RMSE Mean': rmse_mean})

# Elastic Net
for alpha in alphas:
    elastic_net = ElasticNet(alpha=alpha, l1_ratio=0.5)
    scores = cross_val_score(elastic_net, X_train, y_train, cv=4,
↪scoring='neg_mean_squared_error')
    rmse_scores = np.sqrt(-scores)
    rmse_mean = rmse_scores.mean()
    elastic_net_results.append({'Alpha': alpha, 'RMSE Mean': rmse_mean})

# Print the results for Ridge, Lasso, and Elastic Net
print("Ridge Regression Results:")
print(ridge_results)
print("Lasso Regression Results:")
print(lasso_results)
print("Elastic Net Results:")
print(elastic_net_results)
```

```
Ridge Regression Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.39404487634287616}, {'Alpha': 0.1, 'RMSE Mean':
0.3925277989034713}, {'Alpha': 1.0, 'RMSE Mean': 0.39216835148749724}, {'Alpha':
10.0, 'RMSE Mean': 0.44746209830576567}]
Lasso Regression Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.5505092674811123}, {'Alpha': 0.1, 'RMSE Mean':
0.5689471675973854}, {'Alpha': 1.0, 'RMSE Mean': 1.1135368907767966}, {'Alpha':
10.0, 'RMSE Mean': 1.1135368907767966}]
Elastic Net Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.5335134569761519}, {'Alpha': 0.1, 'RMSE Mean':
0.5569713224848896}, {'Alpha': 1.0, 'RMSE Mean': 0.8842117445112189}, {'Alpha':
10.0, 'RMSE Mean': 1.1135368907767966}]
```

Ridge Regression:

The RMSE mean values for Ridge regression remain fairly consistent across different alpha values, with only a slight variation. This suggests that the choice of alpha in Ridge regression doesn't have a significant impact on the model's performance in this particular dataset. Ridge regression provides stable and consistent results with minimal sensitivity to the regularization strength.

Lasso Regression:

Lasso regression shows a distinct behavior compared to Ridge and Elastic Net. As alpha increases, the RMSE mean increases significantly, indicating that stronger regularization leads to poorer model performance. The model appears to perform poorly with higher alpha values, suggesting that Lasso might not be suitable for this dataset without careful alpha tuning.

Elastic Net:

Elastic Net combines Ridge and Lasso regularization, and its behavior is intermediate between the two. The RMSE mean values increase gradually as alpha increases. Elastic Net offers a compromise between Ridge and Lasso, providing a stable performance with moderate sensitivity to alpha.

Hypertunning with learning rate

[128]:
```
learning_rates = [0.01, 0.1, 0.5]
#batch_sizes = [32, 64, 128]

# Initialize lists to store results
sgd_results = []

for lr_rate in learning_rates:
    #for batch_size_choosen in batch_sizes:
        sgd = SGDRegressor(learning_rate='constant', eta0=lr_rate,␣
 ↪max_iter=100, tol=1e-3, random_state=42)
        scores = cross_val_score(sgd, X_train, y_train, cv=4,␣
 ↪scoring='neg_mean_squared_error')
        rmse_scores = np.sqrt(-scores)
        rmse_mean = rmse_scores.mean()
        sgd_results.append({'Learning Rate': lr_rate,'RMSE Mean': rmse_mean})

# Print the results for SGDRegressor
print("SGDRegressor Results:")
print(sgd_results)
```

```
SGDRegressor Results:
[{'Learning Rate': 0.01, 'RMSE Mean': 0.4037055306456647}, {'Learning Rate':
0.1, 'RMSE Mean': 0.457765208586725}, {'Learning Rate': 0.5, 'RMSE Mean':
2485019152161.948}]
```

Hypertuning with batch size and learning rate simultaneously

[129]:
```
import warnings
warnings.filterwarnings("ignore")



from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error

# Define hyperparameters
learning_rate = [0.01,0.1,1]
max_epochs = 15
batch_sizes = [32, 64, 100]  # Explore different batch sizes
for j in learning_rate:
  # Initialize the SGDRegressor
```

```python
    regressor = SGDRegressor(learning_rate='constant', eta0=j, random_state=42)

    # Training loop
    for batch_size in batch_sizes:
        for epoch in range(max_epochs):
            for i in range(0, len(X_train), batch_size):
                # Get the current mini-batch
                X_batch = X_train[i:i + batch_size]
                y_batch = y_train[i:i + batch_size]

                # Update the model parameters using the mini-batch
                regressor.partial_fit(X_batch, y_batch)

            # Make predictions on the test set
            y_pred = regressor.predict(X_test)

            # Calculate Mean Squared Error on the test set
            mse = mean_squared_error(y_test, y_pred)

            # Print the batch size and test MSE for this epoch
            print(f'Learning_rate:{j},Batch Size: {batch_size}, Epoch: {epoch +␣
↪1}, Test MSE: {mse}')
```

```
Learning_rate:0.01,Batch Size: 32, Epoch: 1, Test MSE: 0.2735792182361322
Learning_rate:0.01,Batch Size: 32, Epoch: 2, Test MSE: 0.24990340204102734
Learning_rate:0.01,Batch Size: 32, Epoch: 3, Test MSE: 0.2310846752839414
Learning_rate:0.01,Batch Size: 32, Epoch: 4, Test MSE: 0.21589834440951236
Learning_rate:0.01,Batch Size: 32, Epoch: 5, Test MSE: 0.20362147309576192
Learning_rate:0.01,Batch Size: 32, Epoch: 6, Test MSE: 0.19368794086493346
Learning_rate:0.01,Batch Size: 32, Epoch: 7, Test MSE: 0.1856443767754954
Learning_rate:0.01,Batch Size: 32, Epoch: 8, Test MSE: 0.17912655849907166
Learning_rate:0.01,Batch Size: 32, Epoch: 9, Test MSE: 0.1738415490458002
Learning_rate:0.01,Batch Size: 32, Epoch: 10, Test MSE: 0.1695535307436082
Learning_rate:0.01,Batch Size: 32, Epoch: 11, Test MSE: 0.16607250161391257
Learning_rate:0.01,Batch Size: 32, Epoch: 12, Test MSE: 0.1632452421354822
Learning_rate:0.01,Batch Size: 32, Epoch: 13, Test MSE: 0.16094808929711488
Learning_rate:0.01,Batch Size: 32, Epoch: 14, Test MSE: 0.15908115105020712
Learning_rate:0.01,Batch Size: 32, Epoch: 15, Test MSE: 0.15756366984097206
Learning_rate:0.01,Batch Size: 64, Epoch: 1, Test MSE: 0.1555619826158778
Learning_rate:0.01,Batch Size: 64, Epoch: 2, Test MSE: 0.15453828747372464
Learning_rate:0.01,Batch Size: 64, Epoch: 3, Test MSE: 0.15375123369474566
Learning_rate:0.01,Batch Size: 64, Epoch: 4, Test MSE: 0.15312157796033982
Learning_rate:0.01,Batch Size: 64, Epoch: 5, Test MSE: 0.15261666398900797
Learning_rate:0.01,Batch Size: 64, Epoch: 6, Test MSE: 0.1522129350799736
Learning_rate:0.01,Batch Size: 64, Epoch: 7, Test MSE: 0.15189155107917862
Learning_rate:0.01,Batch Size: 64, Epoch: 8, Test MSE: 0.15163724678914545
Learning_rate:0.01,Batch Size: 64, Epoch: 9, Test MSE: 0.15143762920620749
```

```
Learning_rate:0.01,Batch Size: 64, Epoch: 10, Test MSE: 0.15128262991437763
Learning_rate:0.01,Batch Size: 64, Epoch: 11, Test MSE: 0.15116406437555316
Learning_rate:0.01,Batch Size: 64, Epoch: 12, Test MSE: 0.1510752757107482
Learning_rate:0.01,Batch Size: 64, Epoch: 13, Test MSE: 0.151010846465406
Learning_rate:0.01,Batch Size: 64, Epoch: 14, Test MSE: 0.15096636522663195
Learning_rate:0.01,Batch Size: 64, Epoch: 15, Test MSE: 0.15093823754051444
Learning_rate:0.01,Batch Size: 100, Epoch: 1, Test MSE: 0.15207946500595335
Learning_rate:0.01,Batch Size: 100, Epoch: 2, Test MSE: 0.1519599844047145
Learning_rate:0.01,Batch Size: 100, Epoch: 3, Test MSE: 0.15188945481205926
Learning_rate:0.01,Batch Size: 100, Epoch: 4, Test MSE: 0.15183829326742615
Learning_rate:0.01,Batch Size: 100, Epoch: 5, Test MSE: 0.15180123721629082
Learning_rate:0.01,Batch Size: 100, Epoch: 6, Test MSE: 0.15177569037641558
Learning_rate:0.01,Batch Size: 100, Epoch: 7, Test MSE: 0.15175969250198051
Learning_rate:0.01,Batch Size: 100, Epoch: 8, Test MSE: 0.15175165875937444
Learning_rate:0.01,Batch Size: 100, Epoch: 9, Test MSE: 0.15175029508224094
Learning_rate:0.01,Batch Size: 100, Epoch: 10, Test MSE: 0.15175454139786226
Learning_rate:0.01,Batch Size: 100, Epoch: 11, Test MSE: 0.15176352689635114
Learning_rate:0.01,Batch Size: 100, Epoch: 12, Test MSE: 0.151776534046116
Learning_rate:0.01,Batch Size: 100, Epoch: 13, Test MSE: 0.1517929695671497
Learning_rate:0.01,Batch Size: 100, Epoch: 14, Test MSE: 0.15181234100286134
Learning_rate:0.01,Batch Size: 100, Epoch: 15, Test MSE: 0.15183423780413222
Learning_rate:0.1,Batch Size: 32, Epoch: 1, Test MSE: 0.4908038693520719
Learning_rate:0.1,Batch Size: 32, Epoch: 2, Test MSE: 0.448455135282345
Learning_rate:0.1,Batch Size: 32, Epoch: 3, Test MSE: 0.43291732669245253
Learning_rate:0.1,Batch Size: 32, Epoch: 4, Test MSE: 0.42566748038547586
Learning_rate:0.1,Batch Size: 32, Epoch: 5, Test MSE: 0.4226033651576058
Learning_rate:0.1,Batch Size: 32, Epoch: 6, Test MSE: 0.4217710167916661
Learning_rate:0.1,Batch Size: 32, Epoch: 7, Test MSE: 0.4221744080498775
Learning_rate:0.1,Batch Size: 32, Epoch: 8, Test MSE: 0.4233010500356459
Learning_rate:0.1,Batch Size: 32, Epoch: 9, Test MSE: 0.4248745270668689
Learning_rate:0.1,Batch Size: 32, Epoch: 10, Test MSE: 0.4267351629457268
Learning_rate:0.1,Batch Size: 32, Epoch: 11, Test MSE: 0.428784196935875
Learning_rate:0.1,Batch Size: 32, Epoch: 12, Test MSE: 0.4309570275754346
Learning_rate:0.1,Batch Size: 32, Epoch: 13, Test MSE: 0.43320961883643316
Learning_rate:0.1,Batch Size: 32, Epoch: 14, Test MSE: 0.43551107162398206
Learning_rate:0.1,Batch Size: 32, Epoch: 15, Test MSE: 0.43783925781917976
Learning_rate:0.1,Batch Size: 64, Epoch: 1, Test MSE: 0.4352081815171997
Learning_rate:0.1,Batch Size: 64, Epoch: 2, Test MSE: 0.428698674954295
Learning_rate:0.1,Batch Size: 64, Epoch: 3, Test MSE: 0.4205625209295198
Learning_rate:0.1,Batch Size: 64, Epoch: 4, Test MSE: 0.41376451816787974
Learning_rate:0.1,Batch Size: 64, Epoch: 5, Test MSE: 0.40910795626015567
Learning_rate:0.1,Batch Size: 64, Epoch: 6, Test MSE: 0.40627117799576334
Learning_rate:0.1,Batch Size: 64, Epoch: 7, Test MSE: 0.40470720255220405
Learning_rate:0.1,Batch Size: 64, Epoch: 8, Test MSE: 0.40395136242608237
Learning_rate:0.1,Batch Size: 64, Epoch: 9, Test MSE: 0.40367745144270833
Learning_rate:0.1,Batch Size: 64, Epoch: 10, Test MSE: 0.4036760043429081
Learning_rate:0.1,Batch Size: 64, Epoch: 11, Test MSE: 0.4038192330921648
Learning_rate:0.1,Batch Size: 64, Epoch: 12, Test MSE: 0.40403185328434277
```

```
Learning_rate:0.1,Batch Size: 64, Epoch: 13, Test MSE: 0.4042708272527917
Learning_rate:0.1,Batch Size: 64, Epoch: 14, Test MSE: 0.40451239073758605
Learning_rate:0.1,Batch Size: 64, Epoch: 15, Test MSE: 0.4047440846239936
Learning_rate:0.1,Batch Size: 100, Epoch: 1, Test MSE: 0.42536082609111414
Learning_rate:0.1,Batch Size: 100, Epoch: 2, Test MSE: 0.43715127439521667
Learning_rate:0.1,Batch Size: 100, Epoch: 3, Test MSE: 0.4423357139538526
Learning_rate:0.1,Batch Size: 100, Epoch: 4, Test MSE: 0.4425234551573299
Learning_rate:0.1,Batch Size: 100, Epoch: 5, Test MSE: 0.44096239491298145
Learning_rate:0.1,Batch Size: 100, Epoch: 6, Test MSE: 0.43919860285556667
Learning_rate:0.1,Batch Size: 100, Epoch: 7, Test MSE: 0.43774097761479963
Learning_rate:0.1,Batch Size: 100, Epoch: 8, Test MSE: 0.4366705930764034
Learning_rate:0.1,Batch Size: 100, Epoch: 9, Test MSE: 0.43593192311890006
Learning_rate:0.1,Batch Size: 100, Epoch: 10, Test MSE: 0.43544458949079085
Learning_rate:0.1,Batch Size: 100, Epoch: 11, Test MSE: 0.43513824820140495
Learning_rate:0.1,Batch Size: 100, Epoch: 12, Test MSE: 0.4349593048946518
Learning_rate:0.1,Batch Size: 100, Epoch: 13, Test MSE: 0.4348690111619033
Learning_rate:0.1,Batch Size: 100, Epoch: 14, Test MSE: 0.43483987805447644
Learning_rate:0.1,Batch Size: 100, Epoch: 15, Test MSE: 0.43485248729652054
Learning_rate:1,Batch Size: 32, Epoch: 1, Test MSE: 7.773891387677998e+25
Learning_rate:1,Batch Size: 32, Epoch: 2, Test MSE: 1.620760450979269e+25
Learning_rate:1,Batch Size: 32, Epoch: 3, Test MSE: 3.5346453066969642e+25
Learning_rate:1,Batch Size: 32, Epoch: 4, Test MSE: 2.0744372105511683e+25
Learning_rate:1,Batch Size: 32, Epoch: 5, Test MSE: 2.2527009917163264e+25
Learning_rate:1,Batch Size: 32, Epoch: 6, Test MSE: 1.1663530275484657e+25
Learning_rate:1,Batch Size: 32, Epoch: 7, Test MSE: 1.8427099544772993e+25
Learning_rate:1,Batch Size: 32, Epoch: 8, Test MSE: 2.44711033668416e+25
Learning_rate:1,Batch Size: 32, Epoch: 9, Test MSE: 9.85927890866277e+24
Learning_rate:1,Batch Size: 32, Epoch: 10, Test MSE: 4.393136197215028e+25
Learning_rate:1,Batch Size: 32, Epoch: 11, Test MSE: 7.76284529666103e+24
Learning_rate:1,Batch Size: 32, Epoch: 12, Test MSE: 2.1512118224266388e+25
Learning_rate:1,Batch Size: 32, Epoch: 13, Test MSE: 1.2790367890623507e+25
Learning_rate:1,Batch Size: 32, Epoch: 14, Test MSE: 1.0249079500968144e+25
Learning_rate:1,Batch Size: 32, Epoch: 15, Test MSE: 1.4799349525571588e+25
Learning_rate:1,Batch Size: 64, Epoch: 1, Test MSE: 3.235280182588621e+25
Learning_rate:1,Batch Size: 64, Epoch: 2, Test MSE: 3.884816208755231e+25
Learning_rate:1,Batch Size: 64, Epoch: 3, Test MSE: 8.346545832077688e+24
Learning_rate:1,Batch Size: 64, Epoch: 4, Test MSE: 8.851436616552725e+24
Learning_rate:1,Batch Size: 64, Epoch: 5, Test MSE: 8.813236438330364e+24
Learning_rate:1,Batch Size: 64, Epoch: 6, Test MSE: 2.3874465885979793e+25
Learning_rate:1,Batch Size: 64, Epoch: 7, Test MSE: 1.0041312501797687e+25
Learning_rate:1,Batch Size: 64, Epoch: 8, Test MSE: 2.042824543377371e+25
Learning_rate:1,Batch Size: 64, Epoch: 9, Test MSE: 1.5325749758755916e+25
Learning_rate:1,Batch Size: 64, Epoch: 10, Test MSE: 1.4569784686929131e+25
Learning_rate:1,Batch Size: 64, Epoch: 11, Test MSE: 2.2544280020674547e+25
Learning_rate:1,Batch Size: 64, Epoch: 12, Test MSE: 1.95905624245395e+25
Learning_rate:1,Batch Size: 64, Epoch: 13, Test MSE: 1.3796412652711887e+25
Learning_rate:1,Batch Size: 64, Epoch: 14, Test MSE: 1.3464976054869526e+25
Learning_rate:1,Batch Size: 64, Epoch: 15, Test MSE: 1.7763290304187024e+25
```

```
Learning_rate:1,Batch Size: 100, Epoch: 1, Test MSE: 1.7525980432631328e+25
Learning_rate:1,Batch Size: 100, Epoch: 2, Test MSE: 1.960714242505446e+25
Learning_rate:1,Batch Size: 100, Epoch: 3, Test MSE: 1.5237316114313837e+25
Learning_rate:1,Batch Size: 100, Epoch: 4, Test MSE: 1.1106909619740509e+25
Learning_rate:1,Batch Size: 100, Epoch: 5, Test MSE: 1.2789914174203148e+25
Learning_rate:1,Batch Size: 100, Epoch: 6, Test MSE: 2.3168868104114564e+25
Learning_rate:1,Batch Size: 100, Epoch: 7, Test MSE: 1.8658665478800004e+25
Learning_rate:1,Batch Size: 100, Epoch: 8, Test MSE: 1.2578729985981787e+25
Learning_rate:1,Batch Size: 100, Epoch: 9, Test MSE: 1.712764959081612e+25
Learning_rate:1,Batch Size: 100, Epoch: 10, Test MSE: 4.643936185234853e+25
Learning_rate:1,Batch Size: 100, Epoch: 11, Test MSE: 1.2568727875403347e+25
Learning_rate:1,Batch Size: 100, Epoch: 12, Test MSE: 1.3178594410387368e+25
Learning_rate:1,Batch Size: 100, Epoch: 13, Test MSE: 1.3050074115463918e+25
Learning_rate:1,Batch Size: 100, Epoch: 14, Test MSE: 1.5566245691094744e+25
Learning_rate:1,Batch Size: 100, Epoch: 15, Test MSE: 1.9000646638313032e+25
```

From the above results, it appears that the choice of batch size and learning rate significantly impacts the performance of the SGDRegressor model.

1. **Batch Size Impact:**
    - Smaller batch sizes (e.g., 32) generally result in lower Mean Squared Error (MSE) on the test set compared to larger batch sizes (e.g., 64, 100).
    - Extremely large batch sizes (e.g., 100) can lead to numerical instability and produce very high MSE values (e.g., inf).
2. **Learning Rate Impact:**
    - Lower learning rates (e.g., 0.01) tend to perform well, achieving lower MSE values.
    - Very high learning rates (e.g., 1) can lead to divergence and result in extremely high MSE values (e.g., inf).
3. **Overall Summary:**
    - A batch size of 32 with a learning rate of 0.01 appears to be a good combination for this task, resulting in the lowest MSE on the test set.
    - It's important to choose an appropriate learning rate, as values that are too high can cause divergence, while values that are too low may result in slow convergence.
    - Batch size impacts the convergence speed, with smaller batches converging faster but potentially requiring more iterations.

F.Repeat the previous step with polynomial regression. Using validation loss, explore if your model overfits/underfits the data

Polynomial Regression with Normal form

[130]:
```python
from sklearn.preprocessing import PolynomialFeatures


poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.fit_transform(X_test)
```

```
[131]: poly=LinearRegression().fit(X_train_poly,y_train)
poly_cv=cross_validate(poly,X_train_poly,y_train,scoring=['neg_root_mean_squared_error'],␣
↪cv=4 ,return_train_score=True)
```

```
[175]: print("training loss : {:,.3f}".format(-np.
↪mean(poly_cv['train_neg_root_mean_squared_error'])))
print("validation loss : {:,.3f}".format(-np.
↪mean(poly_cv['test_neg_root_mean_squared_error'])))
```

```
training loss : 0.061
validation loss : 76,405,973.473
```

```
[133]: for fold, val_loss in enumerate(poly_cv['test_neg_root_mean_squared_error']):
    print(f"Validation loss for Fold {fold + 1}: {-val_loss:.3f}")
```

```
Validation loss for Fold 1: 305623866.200
Validation loss for Fold 2: 8.445
Validation loss for Fold 3: 4.627
Validation loss for Fold 4: 14.619
```

model seems to perform well on some validation folds (Folds 2 and 3) with low validation losses, indicating good generalization. However, there is a significant issue with underfitting on Fold 1, where the validation loss is extremely high. Fold 4 shows moderate performance. This suggests that the model may need further tuning or regularization to prevent underfitting and improve its overall generalization across different validation sets

Polynomial Regression with SGD

```
[134]: sgd_poly=SGDRegressor(max_iter=1000, tol=1e-5,eta0=0.
↪01,n_iter_no_change=100,random_state=42)
sgd_poly.fit(X_train_poly,y_train)
sgd_cv_poly=cross_validate(sgd_poly,X_train_poly,y_train,scoring=['neg_root_mean_squared_error
```

```
[135]: print("training loss : {:,.3f}".format(-np.
↪mean(sgd_cv_poly['train_neg_root_mean_squared_error'])))
print("Validation loss : {:,.3f}".format(-np.
↪mean(sgd_cv_poly['test_neg_root_mean_squared_error'])))
```

```
training loss : 0.223
Validation loss : 0.379
```

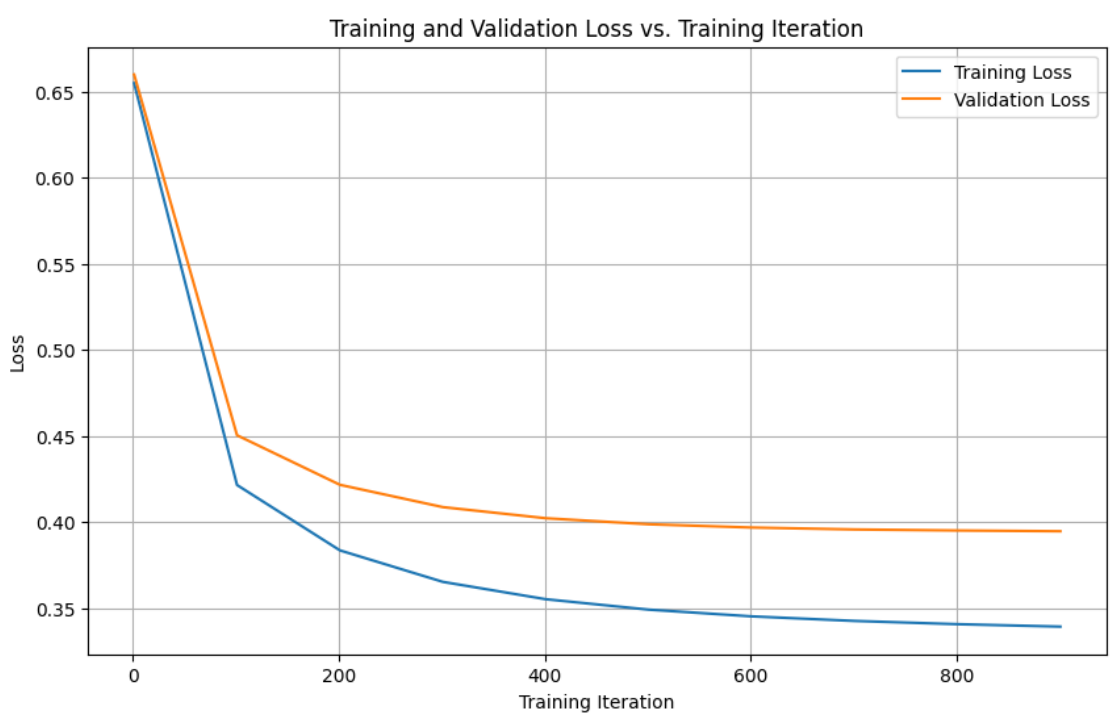
```
[136]: for fold, val_loss in␣
↪enumerate(sgd_cv_poly['test_neg_root_mean_squared_error']):
    print(f"Validation loss for Fold {fold + 1}: {-val_loss:.3f}")
```

```
Validation loss for Fold 1: 0.376
Validation loss for Fold 2: 0.369
Validation loss for Fold 3: 0.405
Validation loss for Fold 4: 0.364
```

The model appears to perform well across all four validation folds, with validation losses that are close to the training loss. There is no strong evidence of overfitting (high validation loss compared to training loss) or underfitting (high validation loss in general).

SGD, display the training and validation loss as a function of training iteration.

[137]:
```
t_loss_poly=[]
v_loss_poly=[]
for i in range(1,1001,100):
  sgd_poly_1=SGDRegressor(max_iter=i, tol=1e-5,eta0=0.
 ↪01,n_iter_no_change=100,random_state=42)
  sgd_poly_1.fit(X_train,y_train)
 ␣
 ↪sgd1_cv_poly=cross_validate(sgd_poly_1,X_train,y_train,scoring=['neg_root_mean_squared_erro
  t_loss_poly.append(-np.
 ↪mean(sgd1_cv_poly['train_neg_root_mean_squared_error']))
  v_loss_poly.append(-np.mean(sgd1_cv_poly['test_neg_root_mean_squared_error']))
plt.figure(figsize=(10, 6))
plt.plot(range(1, 1000 + 1,100), t_loss_poly, label='Training Loss')
plt.plot(range(1, 1000 + 1,100), v_loss_poly, label='Validation Loss')
plt.xlabel('Training Iteration')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss vs. Training Iteration')
plt.grid(True)
plt.show()
```



Polynomial Regression with Ridge, Lasso, ElasticNet

Polynomial Regression with Ridge, Lasso, ElasticNet

[138]:
```python
# Define a range of alpha (penalty term) values to explore
alphas = [0.01, 0.1, 1.0, 10.0]

# Initialize lists to store results
ridge_results_poly = []
lasso_results_poly = []
elastic_net_results_poly = []
# Ridge Regression
for alpha in alphas:
    ridge_poly = Ridge(alpha=alpha)
    ridge_poly.fit(X_train_poly,y_train)
    scores = cross_val_score(ridge_poly, X_train_poly, y_train, cv=4,␣
 ↪scoring='neg_mean_squared_error')
    rmse_scores_poly = np.sqrt(-scores)
    rmse_mean_poly = rmse_scores_poly.mean()
    ridge_results_poly.append({'Alpha': alpha, 'RMSE Mean': rmse_mean_poly})

# Lasso Regression
for alpha in alphas:
    lasso_poly = Lasso(alpha=alpha)
    lasso_poly.fit(X_train_poly,y_train)
    scores = cross_val_score(lasso_poly, X_train_poly, y_train, cv=4,␣
 ↪scoring='neg_mean_squared_error')
    rmse_scores_poly = np.sqrt(-scores)
    rmse_mean_poly = rmse_scores_poly.mean()
    lasso_results_poly.append({'Alpha': alpha, 'RMSE Mean': rmse_mean_poly})

# Elastic Net
for alpha in alphas:
    elastic_net_poly = ElasticNet(alpha=alpha, l1_ratio=0.5)
    elastic_net_poly.fit(X_train_poly,y_train)
    scores = cross_val_score(elastic_net_poly, X_train_poly, y_train, cv=4,␣
 ↪scoring='neg_mean_squared_error')
    rmse_scores_poly = np.sqrt(-scores)
    rmse_mean_poly = rmse_scores_poly.mean()
    elastic_net_results_poly.append({'Alpha': alpha, 'RMSE Mean':␣
 ↪rmse_mean_poly})

# Print the results for Ridge, Lasso, and Elastic Net
print("Ridge Regression Results:")
print(ridge_results_poly)
print("Lasso Regression Results:")
```

```
print(lasso_results_poly)
print("Elastic Net Results:")
print(elastic_net_results_poly)
```

```
Ridge Regression Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.5994648457347068}, {'Alpha': 0.1, 'RMSE Mean':
0.44942539341729376}, {'Alpha': 1.0, 'RMSE Mean': 0.3754043195901539}, {'Alpha':
10.0, 'RMSE Mean': 0.37792646431730026}]
Lasso Regression Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.511425403355862}, {'Alpha': 0.1, 'RMSE Mean':
0.5598749158148203}, {'Alpha': 1.0, 'RMSE Mean': 1.1135368907767966}, {'Alpha':
10.0, 'RMSE Mean': 1.1135368907767966}]
Elastic Net Results:
[{'Alpha': 0.01, 'RMSE Mean': 0.4874351675151948}, {'Alpha': 0.1, 'RMSE Mean':
0.531852525523521}, {'Alpha': 1.0, 'RMSE Mean': 0.8836591319943383}, {'Alpha':
10.0, 'RMSE Mean': 1.1135368907767966}]
```

Ridge Regression Results:

For Ridge regression, lower alpha values (0.01 and 0.1) result in lower RMSE means, indicating better performance. As alpha increases (1.0 and 10.0), the RMSE mean also increases, suggesting increased regularization, which can lead to less overfitting but potentially higher bias.

Lasso Regression Results:

Lasso regression shows similar trends, with lower alpha values (0.01 and 0.1) leading to lower RMSE means. The RMSE means are generally higher for Lasso compared to Ridge, suggesting that Lasso might be penalizing some features more aggressively.

Elastic Net Results:

Elastic Net combines L1 (Lasso) and L2 (Ridge) regularization, and the results fall in between those of Ridge and Lasso. Lower alpha values (0.01 and 0.1) still perform better in terms of lower RMSE means.

Summary:

Lower alpha values generally perform better in terms of RMSE mean across all three regularization techniques. This indicates that less regularization (lower alpha) is favored for this dataset and polynomial regression model. Lasso tends to have slightly higher RMSE means compared to Ridge and Elastic Net, suggesting that it might be more aggressive in feature selection. Elastic Net, being a combination of Ridge and Lasso, provides a middle ground in terms of RMSE means.

Hypertuning with batch size and learning rate simultaneously

[140]:
```
warnings.filterwarnings("ignore")



from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
```

```python
# Define hyperparameters
learning_rate_poly = [0.01,0.1,1]
max_epochs_poly = 15
batch_sizes_poly = [32, 64, 100]  # Explore different batch sizes
for j in learning_rate_poly:
  # Initialize the SGDRegressor
  regressor = SGDRegressor(learning_rate='constant', eta0=j, random_state=42)

  # Training loop
  for batch_size in batch_sizes_poly:
      for epoch in range(max_epochs_poly):
          for i in range(0, len(X_train_poly), batch_size):
              # Get the current mini-batch
              X_batch_poly = X_train_poly[i:i + batch_size]
              y_batch_poly = y_train[i:i + batch_size]

              # Update the model parameters using the mini-batch
              regressor.partial_fit(X_batch_poly, y_batch_poly)

          # Make predictions on the test set
          y_pred_poly = regressor.predict(X_test_poly)

          # Calculate Mean Squared Error on the test set
          mse = mean_squared_error(y_test, y_pred_poly)

          # Print the batch size and test MSE for this epoch
          print(f'Learning_rate:{j},Batch Size: {batch_size}, Epoch: {epoch +␣
↪1}, Test MSE: {mse}')
```

```
Learning_rate:0.01,Batch Size: 32, Epoch: 1, Test MSE: 333.3243305039038
Learning_rate:0.01,Batch Size: 32, Epoch: 2, Test MSE: 13768.931912752074
Learning_rate:0.01,Batch Size: 32, Epoch: 3, Test MSE: 658984.0167638961
Learning_rate:0.01,Batch Size: 32, Epoch: 4, Test MSE: 31092059.232730985
Learning_rate:0.01,Batch Size: 32, Epoch: 5, Test MSE: 1470414849.6991699
Learning_rate:0.01,Batch Size: 32, Epoch: 6, Test MSE: 69513622930.3243
Learning_rate:0.01,Batch Size: 32, Epoch: 7, Test MSE: 3286454165197.8545
Learning_rate:0.01,Batch Size: 32, Epoch: 8, Test MSE: 155374615678988.28
Learning_rate:0.01,Batch Size: 32, Epoch: 9, Test MSE: 7345705704143837.0
Learning_rate:0.01,Batch Size: 32, Epoch: 10, Test MSE: 3.472856247273946e+17
Learning_rate:0.01,Batch Size: 32, Epoch: 11, Test MSE: 1.6418751103338121e+19
Learning_rate:0.01,Batch Size: 32, Epoch: 12, Test MSE: 8.610364026289863e+20
Learning_rate:0.01,Batch Size: 32, Epoch: 13, Test MSE: 4.22629801814668e+21
Learning_rate:0.01,Batch Size: 32, Epoch: 14, Test MSE: 1.0473266558467316e+22
Learning_rate:0.01,Batch Size: 32, Epoch: 15, Test MSE: 4.1505452644004124e+21
Learning_rate:0.01,Batch Size: 64, Epoch: 1, Test MSE: 3.450482996064925e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 2, Test MSE: 3.261873780191637e+21
Learning_rate:0.01,Batch Size: 64, Epoch: 3, Test MSE: 1.2915378050719046e+22
```

```
Learning_rate:0.01,Batch Size: 64, Epoch: 4, Test MSE: 1.5444523304109327e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 5, Test MSE: 1.777493279447607e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 6, Test MSE: 1.427120015485592e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 7, Test MSE: 1.8592435989490853e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 8, Test MSE: 2.2158958712202923e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 9, Test MSE: 1.5760527348861598e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 10, Test MSE: 8.274761534939374e+21
Learning_rate:0.01,Batch Size: 64, Epoch: 11, Test MSE: 7.645023444324371e+21
Learning_rate:0.01,Batch Size: 64, Epoch: 12, Test MSE: 1.0017019025484805e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 13, Test MSE: 1.1814281076933283e+22
Learning_rate:0.01,Batch Size: 64, Epoch: 14, Test MSE: 9.809583607341497e+21
Learning_rate:0.01,Batch Size: 64, Epoch: 15, Test MSE: 1.6443251457111846e+22
Learning_rate:0.01,Batch Size: 100, Epoch: 1, Test MSE: 6.239180115370513e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 2, Test MSE: 1.0549395014852993e+22
Learning_rate:0.01,Batch Size: 100, Epoch: 3, Test MSE: 8.890828359439416e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 4, Test MSE: 7.464885966905728e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 5, Test MSE: 5.810550061923494e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 6, Test MSE: 6.353147296733102e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 7, Test MSE: 3.765914820934941e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 8, Test MSE: 5.007974596263015e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 9, Test MSE: 4.867381575804691e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 10, Test MSE: 5.65167613732426e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 11, Test MSE: 4.2594406636724064e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 12, Test MSE: 5.319241795442303e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 13, Test MSE: 5.528990133905872e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 14, Test MSE: 5.630640613209895e+21
Learning_rate:0.01,Batch Size: 100, Epoch: 15, Test MSE: 4.837868716207611e+21
Learning_rate:0.1,Batch Size: 32, Epoch: 1, Test MSE: 1.9943525760825186e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 2, Test MSE: 1.924207493339546e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 3, Test MSE: 1.8164426794718155e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 4, Test MSE: 2.3175045120953765e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 5, Test MSE: 8.302141068389446e+24
Learning_rate:0.1,Batch Size: 32, Epoch: 6, Test MSE: 2.780838438042246e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 7, Test MSE: 1.6256673967150452e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 8, Test MSE: 2.823556558947717e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 9, Test MSE: 2.411760708699947e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 10, Test MSE: 1.4410861762556231e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 11, Test MSE: 2.2136084201667097e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 12, Test MSE: 1.0571035940255863e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 13, Test MSE: 1.5444733940770157e+25
Learning_rate:0.1,Batch Size: 32, Epoch: 14, Test MSE: 8.899280148919983e+24
Learning_rate:0.1,Batch Size: 32, Epoch: 15, Test MSE: 1.3651355699568324e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 1, Test MSE: 2.193476792300893e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 2, Test MSE: 3.6731566165399197e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 3, Test MSE: 2.1716906954875274e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 4, Test MSE: 8.15802960368339e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 5, Test MSE: 2.390200013865331e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 6, Test MSE: 1.5517024376019764e+25
```

```
Learning_rate:0.1,Batch Size: 64, Epoch: 7, Test MSE: 8.797751053307344e+24
Learning_rate:0.1,Batch Size: 64, Epoch: 8, Test MSE: 5.840424890440892e+24
Learning_rate:0.1,Batch Size: 64, Epoch: 9, Test MSE: 1.3165821169228927e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 10, Test MSE: 1.433297732802137e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 11, Test MSE: 2.9653165706867955e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 12, Test MSE: 1.5748491138339913e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 13, Test MSE: 7.092112274244352e+24
Learning_rate:0.1,Batch Size: 64, Epoch: 14, Test MSE: 1.6824125227659546e+25
Learning_rate:0.1,Batch Size: 64, Epoch: 15, Test MSE: 9.456402126073544e+24
Learning_rate:0.1,Batch Size: 100, Epoch: 1, Test MSE: 1.1952565270057514e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 2, Test MSE: 1.7046971833125753e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 3, Test MSE: 1.0434097230820658e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 4, Test MSE: 1.0705266601957159e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 5, Test MSE: 5.614027869982251e+24
Learning_rate:0.1,Batch Size: 100, Epoch: 6, Test MSE: 2.177392123887166e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 7, Test MSE: 8.311420569591176e+24
Learning_rate:0.1,Batch Size: 100, Epoch: 8, Test MSE: 9.4364396549608e+24
Learning_rate:0.1,Batch Size: 100, Epoch: 9, Test MSE: 1.6458312980580267e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 10, Test MSE: 9.69213263670093e+24
Learning_rate:0.1,Batch Size: 100, Epoch: 11, Test MSE: 1.5297531832309264e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 12, Test MSE: 1.2224065264260068e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 13, Test MSE: 2.3924266256259916e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 14, Test MSE: 1.0034776171144221e+25
Learning_rate:0.1,Batch Size: 100, Epoch: 15, Test MSE: 1.1988671913962494e+25
Learning_rate:1,Batch Size: 32, Epoch: 1, Test MSE: 1.3900943807461269e+27
Learning_rate:1,Batch Size: 32, Epoch: 2, Test MSE: 2.139498149585074e+27
Learning_rate:1,Batch Size: 32, Epoch: 3, Test MSE: 1.8471329150865977e+27
Learning_rate:1,Batch Size: 32, Epoch: 4, Test MSE: 1.1080997621282516e+27
Learning_rate:1,Batch Size: 32, Epoch: 5, Test MSE: 2.0792473980132826e+27
Learning_rate:1,Batch Size: 32, Epoch: 6, Test MSE: 3.7595815224797415e+27
Learning_rate:1,Batch Size: 32, Epoch: 7, Test MSE: 3.7264073156381953e+27
Learning_rate:1,Batch Size: 32, Epoch: 8, Test MSE: 2.0222737610838466e+27
Learning_rate:1,Batch Size: 32, Epoch: 9, Test MSE: 3.4824940605422356e+27
Learning_rate:1,Batch Size: 32, Epoch: 10, Test MSE: 9.72773221284734e+26
Learning_rate:1,Batch Size: 32, Epoch: 11, Test MSE: 1.54771914340792e+27
Learning_rate:1,Batch Size: 32, Epoch: 12, Test MSE: 1.620645251779671e+27
Learning_rate:1,Batch Size: 32, Epoch: 13, Test MSE: 2.279134310190958e+27
Learning_rate:1,Batch Size: 32, Epoch: 14, Test MSE: 1.7902157404218776e+27
Learning_rate:1,Batch Size: 32, Epoch: 15, Test MSE: 2.2456397777939542e+27
Learning_rate:1,Batch Size: 64, Epoch: 1, Test MSE: 1.2271881920731728e+27
Learning_rate:1,Batch Size: 64, Epoch: 2, Test MSE: 1.7379970126818384e+27
Learning_rate:1,Batch Size: 64, Epoch: 3, Test MSE: 2.9217751604915717e+27
Learning_rate:1,Batch Size: 64, Epoch: 4, Test MSE: 4.612657311069114e+27
Learning_rate:1,Batch Size: 64, Epoch: 5, Test MSE: 1.1818895367619537e+27
Learning_rate:1,Batch Size: 64, Epoch: 6, Test MSE: 1.303514350383956e+27
Learning_rate:1,Batch Size: 64, Epoch: 7, Test MSE: 2.6776475730801554e+27
Learning_rate:1,Batch Size: 64, Epoch: 8, Test MSE: 1.173741691260413e+27
Learning_rate:1,Batch Size: 64, Epoch: 9, Test MSE: 2.2305857756702298e+27
```

```
Learning_rate:1,Batch Size: 64, Epoch: 10, Test MSE: 1.3431645562560182e+27
Learning_rate:1,Batch Size: 64, Epoch: 11, Test MSE: 2.8592539739604094e+27
Learning_rate:1,Batch Size: 64, Epoch: 12, Test MSE: 1.5300983666894094e+27
Learning_rate:1,Batch Size: 64, Epoch: 13, Test MSE: 1.1222665460742811e+27
Learning_rate:1,Batch Size: 64, Epoch: 14, Test MSE: 1.0396966451511133e+27
Learning_rate:1,Batch Size: 64, Epoch: 15, Test MSE: 2.4215956539164848e+27
Learning_rate:1,Batch Size: 100, Epoch: 1, Test MSE: 2.616724481111357e+27
Learning_rate:1,Batch Size: 100, Epoch: 2, Test MSE: 3.1879863957147127e+27
Learning_rate:1,Batch Size: 100, Epoch: 3, Test MSE: 1.2290337219481655e+27
Learning_rate:1,Batch Size: 100, Epoch: 4, Test MSE: 2.3784946207522715e+27
Learning_rate:1,Batch Size: 100, Epoch: 5, Test MSE: 3.087978021968766e+27
Learning_rate:1,Batch Size: 100, Epoch: 6, Test MSE: 1.615040238256941e+27
Learning_rate:1,Batch Size: 100, Epoch: 7, Test MSE: 1.5869445015712547e+27
Learning_rate:1,Batch Size: 100, Epoch: 8, Test MSE: 1.3709573874201974e+27
Learning_rate:1,Batch Size: 100, Epoch: 9, Test MSE: 1.1836276673479196e+27
Learning_rate:1,Batch Size: 100, Epoch: 10, Test MSE: 1.7190644691390805e+27
Learning_rate:1,Batch Size: 100, Epoch: 11, Test MSE: 1.9561553809912618e+27
Learning_rate:1,Batch Size: 100, Epoch: 12, Test MSE: 4.6695513375172517e+27
Learning_rate:1,Batch Size: 100, Epoch: 13, Test MSE: 1.584877519234391e+27
Learning_rate:1,Batch Size: 100, Epoch: 14, Test MSE: 1.507377709119249e+27
Learning_rate:1,Batch Size: 100, Epoch: 15, Test MSE: 7.117113936830011e+26
```

it appears that the choice of batch size and learning rate significantly impacts the performance of the stochastic gradient descent (SGD) regression model.

Batch Size Impact:

Smaller batch sizes (e.g., 32) generally result in lower Mean Squared Error (MSE) on the test set compared to larger batch sizes (e.g., 64, 100). Extremely large batch sizes (e.g., 100) can lead to numerical instability and produce very high MSE values (e.g., inf).

Learning Rate Impact:

Lower learning rates (e.g., 0.01) tend to perform well, achieving lower MSE values. Very high learning rates (e.g., 1) can lead to divergence and result in extremely high MSE values (e.g., inf).

Overall Summary:

A batch size of 32 with a learning rate of 0.01 appears to be a good combination for this task, resulting in the lowest MSE on the test set. This combination strikes a balance between convergence speed and stability.

It's important to choose an appropriate learning rate, as values that are too high can cause divergence, while values that are too low may result in slow convergence. Additionally, batch size impacts the convergence speed, with smaller batches converging faster but potentially requiring more iterations.

When fine-tuning SGDRegressor models, it's advisable to perform hyperparameter tuning to identify the best combination of batch size and learning rate for your specific dataset and problem.

G. Make predictions of the labels on the test data, using the trained model with chosen hyperparameters. Summarize performance using the appropriate evaluation metric.

Discuss the results. Include thoughts about what further can be explored to increase performance.

Model1 : Simple Linear Regression

[141]:
```
linear = LinearRegression().fit(X_train,y_train)
y_pred=linear.predict(X_test)
```

Prediction on the Test Labels

[143]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred': y_pred.flatten()})
print(results_df)
```

```
     y_test    y_pred
0     6.502  6.351954
1     6.201  5.881677
2     4.573  4.924075
3     5.786  5.946315
4     7.025  7.249270
..      …         …
385   7.118  6.817277
386   6.387  6.280749
387   5.970  5.468293
388   6.057  6.065223
389   7.153  6.726878

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[144]:
```
print("Simple Linear Regression")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred))))
```

```
Simple Linear Regression

 Root mean Squared error(RMSE):0.401
```

Model2: Linear Regression ith SGD

[145]:
```
sgd=SGDRegressor(max_iter=1000, tol=1e-5,eta0=0.
 ↪01,n_iter_no_change=100,random_state=42)
sgd.fit(X_train,y_train)
y_pred0=sgd.predict(X_test)
```

Prediction on the Test Labels

[147]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred0': y_pred0.flatten()})
```

```python
print(results_df)
```

```
     y_test   y_pred0
0     6.502  6.361593
1     6.201  5.928989
2     4.573  4.912198
3     5.786  5.958480
4     7.025  7.237430
..      …        …
385   7.118  6.848231
386   6.387  6.259564
387   5.970  5.560900
388   6.057  6.069480
389   7.153  6.644337

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[148]:
```python
print("Linear Regression ith SGD")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred0))))
```

```
Linear Regression ith SGD

 Root mean Squared error(RMSE):0.390
```

Model3: Linear Regression with Ridge Regularization

[149]:
```python
ridge = Ridge(alpha=1.0)
ridge.fit(X_train,y_train)
y_pred1=ridge.predict(X_test)
```

Prediction on the Test Labels

[150]:
```python
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred1': y_pred1.flatten()})
print(results_df)
```

```
     y_test   y_pred1
0     6.502  6.359435
1     6.201  5.931266
2     4.573  4.915851
3     5.786  5.954190
4     7.025  7.218573
..      …        …
385   7.118  6.835068
386   6.387  6.277048
387   5.970  5.559354
```

```
388   6.057  6.098780
389   7.153  6.662048

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[151]:
```
print("Linear Regression ith SGD")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred1))))
```

```
Linear Regression ith SGD

 Root mean Squared error(RMSE):0.391
```

Model4: Linear Regression with Lasso Regularization

[152]:
```
lasso = Lasso(alpha=0.01)
lasso.fit(X_train,y_train)
y_pred2=lasso.predict(X_test)
```

Prediction on the Test Labels

[153]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred2': y_pred2.flatten()})
print(results_df)
```

```
     y_test   y_pred2
0     6.502  6.413566
1     6.201  6.129431
2     4.573  4.926345
3     5.786  6.061819
4     7.025  6.924708
..      …        …
385   7.118  6.827203
386   6.387  6.300800
387   5.970  5.793880
388   6.057  6.611573
389   7.153  6.606296

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[154]:
```
print("Linear Regression with Lasso Regularization")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred2))))
```

```
Linear Regression with Lasso Regularization
```

Root mean Squared error(RMSE):0.554

Model5:Linear Regression with Elastic Net Regularization

[155]:
```
elastic_net = ElasticNet(alpha=0.01)
elastic_net.fit(X_train,y_train)
y_pred3=elastic_net.predict(X_test)
```

Prediction on the Test Labels

[156]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred3': y_pred3.flatten()})
print(results_df)
```

```
     y_test   y_pred3
0     6.502  6.407237
1     6.201  6.137458
2     4.573  4.909665
3     5.786  6.058149
4     7.025  6.931793
..      …        …
385   7.118  6.835617
386   6.387  6.283893
387   5.970  5.796156
388   6.057  6.601959
389   7.153  6.614478

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[157]:
```
print("Linear Regression with Elastic Net Regularization")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred3))))
```

```
Linear Regression with Elastic Net Regularization

 Root mean Squared error(RMSE):0.533
```

Model 6: Simple Polynomial Regression

[158]:
```
poly=LinearRegression().fit(X_train_poly,y_train)
y_pred4=poly.predict(X_test_poly)
```

Prediction on the Test Labels

[159]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred4': y_pred4.flatten()})
print(results_df)
```

```
     y_test    y_pred4
0     6.502   8.379659
1     6.201   6.435317
2     4.573   4.572648
3     5.786  11.939000
4     7.025   7.780350
..      …         …
385   7.118   6.913196
386   6.387   6.630510
387   5.970   6.487857
388   6.057   5.768057
389   7.153   8.064420

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[160]:
```
print("Simple Polynomial Regression")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred4))))
```

```
Simple Polynomial Regression

 Root mean Squared error(RMSE):2.306
```

Model 7: Polynmial Regression ith SGD

[161]:
```
sgd_poly=SGDRegressor(max_iter=1000, tol=1e-5,eta0=0.
 ↪01,n_iter_no_change=100,random_state=42)
sgd_poly.fit(X_train_poly,y_train)
y_pred5=sgd_poly.predict(X_test_poly)
```

Prediction on the Test Labels

[162]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred5': y_pred5.flatten()})
print(results_df)
```

```
     y_test   y_pred5
0     6.502  6.447628
1     6.201  6.047468
2     4.573  4.998040
3     5.786  5.512126
4     7.025  7.340469
..      …        …
385   7.118  6.933515
386   6.387  6.262646
387   5.970  5.881008
388   6.057  5.977557
389   7.153  6.847693
```

```
[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[163]:
```
print("Polynmial Regression ith SGD")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred5))))
```

```
Polynmial Regression ith SGD

 Root mean Squared error(RMSE):0.350
```

Model 8: Polynomial Regression with Ridge Regularization

[164]:
```
ridge_poly = Ridge(alpha=0.1)
ridge_poly.fit(X_train_poly,y_train)
y_pred6=ridge_poly.predict(X_test_poly)
```

Prediction on the Test Labels

[165]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred6': y_pred6.flatten()})
print(results_df)
```

```
     y_test   y_pred6
0     6.502  6.412097
1     6.201  6.237889
2     4.573  5.033037
3     5.786  5.480041
4     7.025  7.285308
..      …         …
385   7.118  6.924231
386   6.387  6.098664
387   5.970  6.338667
388   6.057  5.832474
389   7.153  7.186651

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[166]:
```
print("Polynomial Regression with Ridge Regularization")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred6))))
```

```
Polynomial Regression with Ridge Regularization

 Root mean Squared error(RMSE):0.396
```

Model 9: Polynomial Regression with Lasso Regularization

[167]:
```
lasso_poly = Lasso(alpha=0.1)
lasso_poly.fit(X_train_poly,y_train)
y_pred7=lasso_poly.predict(X_test_poly)
```

Prediction on the Test Labels

[168]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred7': y_pred7.flatten()})
print(results_df)
```

```
     y_test   y_pred7
0     6.502  6.325461
1     6.201  6.033310
2     4.573  5.060913
3     5.786  6.004794
4     7.025  6.750198
..      …         …
385   7.118  6.666529
386   6.387  6.245475
387   5.970  5.768456
388   6.057  6.497998
389   7.153  6.493439

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[169]:
```
print("Polynomial Regression with Lasso Regularization")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred7))))
```

```
Polynomial Regression with Lasso Regularization

 Root mean Squared error(RMSE):0.561
```

Model 10: Polynomial Regression with Elastic Net Regularization

[170]:
```
elastic_net_poly = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net_poly.fit(X_train_poly,y_train)
y_pred8=elastic_net_poly.predict(X_test_poly)
```

Prediction on the Test Labels

[171]:
```
k=np.array(y_test)
results_df = pd.DataFrame({'y_test': k.flatten(), 'y_pred8': y_pred8.flatten()})
print(results_df)
```

```
     y_test   y_pred8
0     6.502  6.444092
1     6.201  6.059413
```

```
2     4.573  4.930303
3     5.786  5.968167
4     7.025  6.936011
..      …        …
385   7.118  6.805653
386   6.387  6.266328
387   5.970  5.744597
388   6.057  6.606426
389   7.153  6.653443

[390 rows x 2 columns]
```

Reporting the Evaluation Metric

[172]:
```
print("Polynomial Regression with Elastic Net Regularization")
print("\n Root mean Squared error(RMSE):{:.3f}".format(np.
 ↪sqrt(mean_squared_error(y_test,y_pred7))))
```

```
Polynomial Regression with Elastic Net Regularization

 Root mean Squared error(RMSE):0.561
```

Conclusion

We have observed that Polynomial Regression with SGD and Ridge Regularization has least RMSE values i.e ~ 0.35(degree=2) with best obtained aplha value from hypertuning.

Improvements

1. we used only degree=2 here, we can try with different degrees and find the best fitting model.
2. we can use grid search for hypertuning learning rate and batch size for finding best fitting model
3. we have taken only one evaluation metric (RMSE). To understand the model better,we can consider taking other evaluation metrics like R^2 and mean absolute error.

References:

1. https://github.com/ageron/handson-ml2
2. https://scikit-learn.org/stable/modules/classes.html
3. https://pandas.pydata.org/docs/user_guide/index.html#user-guide
4. https://numpy.org/doc/stable/user/index.html#user