z5mpjtzhm

February 17, 2024

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

# Load the data
#data = pd.read_csv('happiness_data.csv')
data=pd.read_csv("C:/Users/prith/OneDrive/Desktop/IU/Spring24/
↪AppliedMachineLearning/jupyter/datasets/happiness_data.csv")

# Print the first few rows of the dataframe
print(data.head())
```

```
  Country name  year  Life Ladder  Log GDP per capita  Social support  \
0  Afghanistan  2008        3.724               7.370           0.451
1  Afghanistan  2009        4.402               7.540           0.552
2  Afghanistan  2010        4.758               7.647           0.539
3  Afghanistan  2011        3.832               7.620           0.521
4  Afghanistan  2012        3.783               7.705           0.521

   Healthy life expectancy at birth  Freedom to make life choices  Generosity  \
0                             50.80                         0.718       0.168
1                             51.20                         0.679       0.190
2                             51.60                         0.600       0.121
3                             51.92                         0.496       0.162
4                             52.24                         0.531       0.236

   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
```

[2]:
```
# Print the shape of the dataframe
print('The size of the dataframe is:', data.shape)

'''
```

```
The size of the dataframe is: (1949, 11)
'''
```

```
The size of the dataframe is: (1949, 11)
```

[2]: '\nThe size of the dataframe is: (1949, 11)\n'

[3]:
```
# Print the data types of the columns
print(data.dtypes)

'''
Attributes that are continuous valued:
1. Life Ladder
2. Log GDP per capita
3. Social support
4. Healthy life expectancy at birth
5. Freedom to make life choices
6. Generosity
7. Perceptions of corruption
8. positive affect
9. negative affect

Attributes that are categorical:
1. Country
2. year
'''
```

```
Country name                         object
year                                  int64
Life Ladder                         float64
Log GDP per capita                  float64
Social support                      float64
Healthy life expectancy at birth    float64
Freedom to make life choices        float64
Generosity                          float64
Perceptions of corruption           float64
Positive affect                     float64
Negative affect                     float64
dtype: object
```

[3]: '\nAttributes that are continuous valued:\n1. Life Ladder\n2. Log GDP per capita\n3. Social support\n4. Healthy life expectancy at birth\n5. Freedom to make life choices\n6. Generosity\n7. Perceptions of corruption\n8. positive affect\n9. negative affect\n\nAttributes that are categorical:\n1. Country\n2. year\n'

The size of the dataframe is: (1949, 11)

Attributes that are continuous valued: 1. Life Ladder 2. Log GDP per capita 3. Social support 4. Healthy life expectancy at birth 5. Freedom to make life choices 6. Generosity 7. Perceptions of corruption 8. positive affect 9. negative affect

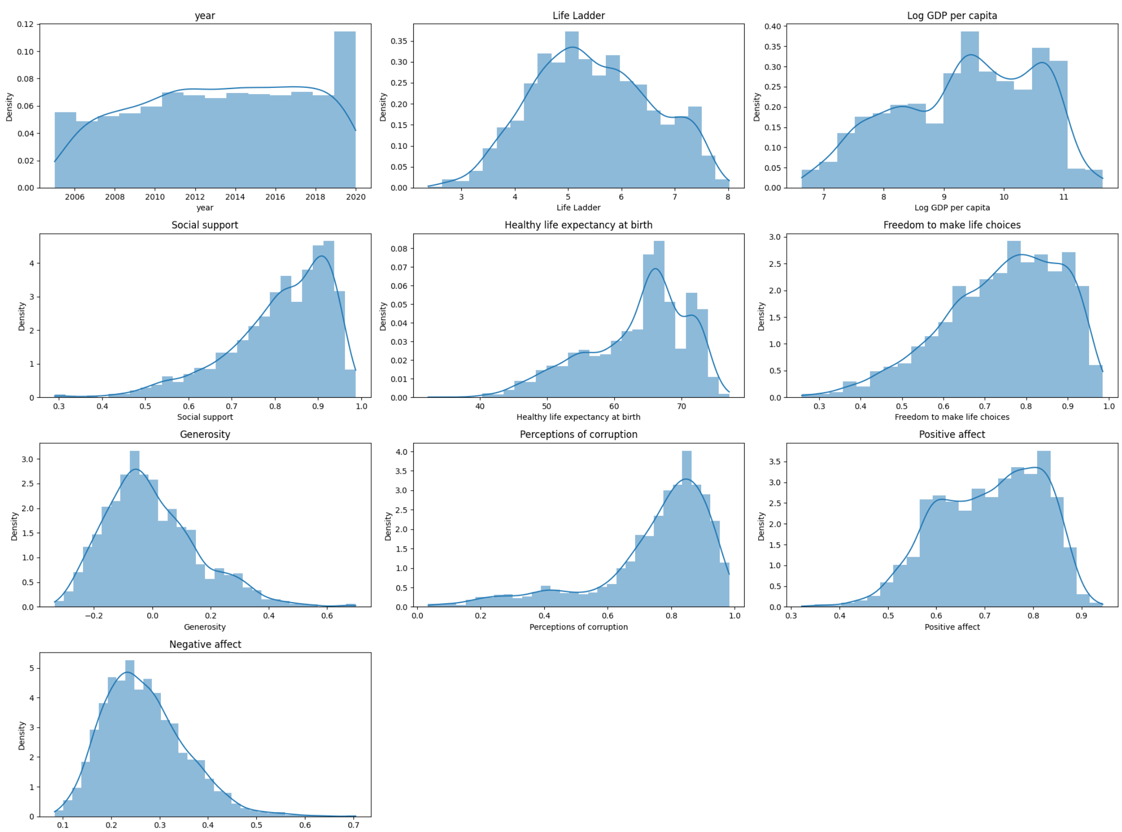
Attributes that are categorical: 1. Country 2. year

[5]:
```python
# 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? [5 points]
import matplotlib.pyplot as plt
import seaborn as sns

# Display statistical values for each attribute
statistics = data.describe()

# Plot histograms for each numerical attribute to visualize distributions
plt.figure(figsize=(20, 15))
for i, column in enumerate(data.select_dtypes(include=['float64', 'int64']).
↪columns, 1):
    plt.subplot(4, 3, i)
    sns.histplot(data[column], kde=True, stat="density", linewidth=0)
    plt.title(column)

plt.tight_layout()
statistics
```

[5]:
```
              year  Life Ladder  Log GDP per capita  Social support  \
count  1949.000000  1949.000000         1913.000000     1936.000000
mean   2013.216008     5.466705            9.368453        0.812552
std       4.166828     1.115711            1.154084        0.118482
min    2005.000000     2.375000            6.635000        0.290000
25%    2010.000000     4.640000            8.464000        0.749750
50%    2013.000000     5.386000            9.460000        0.835500
75%    2017.000000     6.283000           10.353000        0.905000
max    2020.000000     8.019000           11.648000        0.987000

       Healthy life expectancy at birth  Freedom to make life choices  \
count                       1894.000000                   1917.000000
mean                          63.359374                      0.742558
std                            7.510245                      0.142093
min                           32.300000                      0.258000
25%                           58.685000                      0.647000
50%                           65.200000                      0.763000
75%                           68.590000                      0.856000
max                           77.100000                      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
count	1933.000000
mean	0.268544
std	0.085168
min	0.083000
25%	0.206000
50%	0.258000
75%	0.320000
max	0.705000



Life Ladder : The distribution is quite baalanced, there is some skewness towards the left. The mean is 5.466. This indicates that the average life satisfaction is slightly above the midpoint of the scale.

Log GDP per capita: This attribute shows a fairly normal distribution but with a slight left skew. The mean Log GDP per capita is 9.368, reflecting a wide range of economic statuses across countries.

Social support: Most values are clustered towards the higher end (mean = 0.812), suggesting that in most countries, individuals perceive a high level of social support.

Healthy life expectancy at birth: The distribution is slightly left-skewed, with a mean of 63.359 years. This indicates that while many countries have a high life expectancy, a significant number have lower values, pulling the average down.

[7]:
```
#Ignoring year and life ladder is to be be predicted so its our "y" attribute
X = data.drop(['year','Life Ladder'],axis=1)
```

[8]:
```
#special treatment:
#1.Check for null values and fix them

#check null values
X.isnull().sum()
```

[8]:
```
Country name                          0
Log GDP per capita                   36
Social support                       13
Healthy life expectancy at birth     55
Freedom to make life choices         32
Generosity                           89
Perceptions of corruption           110
Positive affect                      22
Negative affect                      16
dtype: int64
```

[10]:
```
#Replacing Null values throughout with their median
X=X.fillna(X.median(numeric_only=True))
X.isnull().sum()
```

[10]:
```
Country name                        0
Log GDP per capita                  0
Social support                      0
Healthy life expectancy at birth    0
Freedom to make life choices        0
Generosity                          0
Perceptions of corruption           0
Positive affect                     0
Negative affect                     0
dtype: int64
```

[11]:
```
#Factorizing Country Name to convert from string to int
#Since Country Name is a categorical data, we encode it and convert into␣
↪numerical data
X['Country name'],_ = pd.factorize(X['Country name'])
X['Country name']
```

[11]:
```
0         0
1         0
2         0
3         0
4         0
        …
1944    165
1945    165
1946    165
1947    165
1948    165
Name: Country name, Length: 1949, dtype: int64
```

[13]:
```
#Calculating the skewness would help us understand the distribution of the data
from scipy import stats
for col in X.columns:
    skewness = stats.skew(X[col])
    print(f"{col} Skewness = {skewness}")
```

```
Country name Skewness = 0.02025630678285456
Log GDP per capita Skewness = -0.31547112391672333
Social support Skewness = -1.116963486077551
Healthy life expectancy at birth Skewness = -0.7732461766043456
Freedom to make life choices Skewness = -0.6344813528475128
Generosity Skewness = 0.8460862253349566
Perceptions of corruption Skewness = -1.577251627639701
Positive affect Skewness = -0.3697065814671494
Negative affect Skewness = 0.7425428204496247
```

C. Analyze 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.

[14]:
```
# Assuming df is your DataFrame and you want to exclude non-numeric columns
numeric_data = data.select_dtypes(include=['float64', 'int64'])

# Compute the correlation matrix
corr_matrix = numeric_data.corr(method='pearson')

# Display the correlation matrix
print(corr_matrix)
```

	year	Life Ladder	Log GDP per capita \
year	1.000000	0.035515	0.078246
Life Ladder	0.035515	1.000000	0.790166
Log GDP per capita	0.078246	0.790166	1.000000
Social support	-0.010093	0.707806	0.692602
Healthy life expectancy at birth	0.164059	0.744506	0.848049
Freedom to make life choices	0.222151	0.528063	0.367932
Generosity	-0.043422	0.190632	-0.000915
Perceptions of corruption	-0.081478	-0.427245	-0.345511
Positive affect	-0.003245	0.532273	0.302282
Negative affect	0.196869	-0.297488	-0.210781

	Social support \
year	-0.010093
Life Ladder	0.707806
Log GDP per capita	0.692602
Social support	1.000000
Healthy life expectancy at birth	0.616037
Freedom to make life choices	0.410402
Generosity	0.067000
Perceptions of corruption	-0.219040
Positive affect	0.432152
Negative affect	-0.395865

	Healthy life expectancy at birth \
year	0.164059
Life Ladder	0.744506
Log GDP per capita	0.848049
Social support	0.616037
Healthy life expectancy at birth	1.000000
Freedom to make life choices	0.388681
Generosity	0.020737
Perceptions of corruption	-0.322461
Positive affect	0.318247
Negative affect	-0.139477

	Freedom to make life choices	Generosity \
year	0.222151	-0.043422
Life Ladder	0.528063	0.190632
Log GDP per capita	0.367932	-0.000915
Social support	0.410402	0.067000
Healthy life expectancy at birth	0.388681	0.020737
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

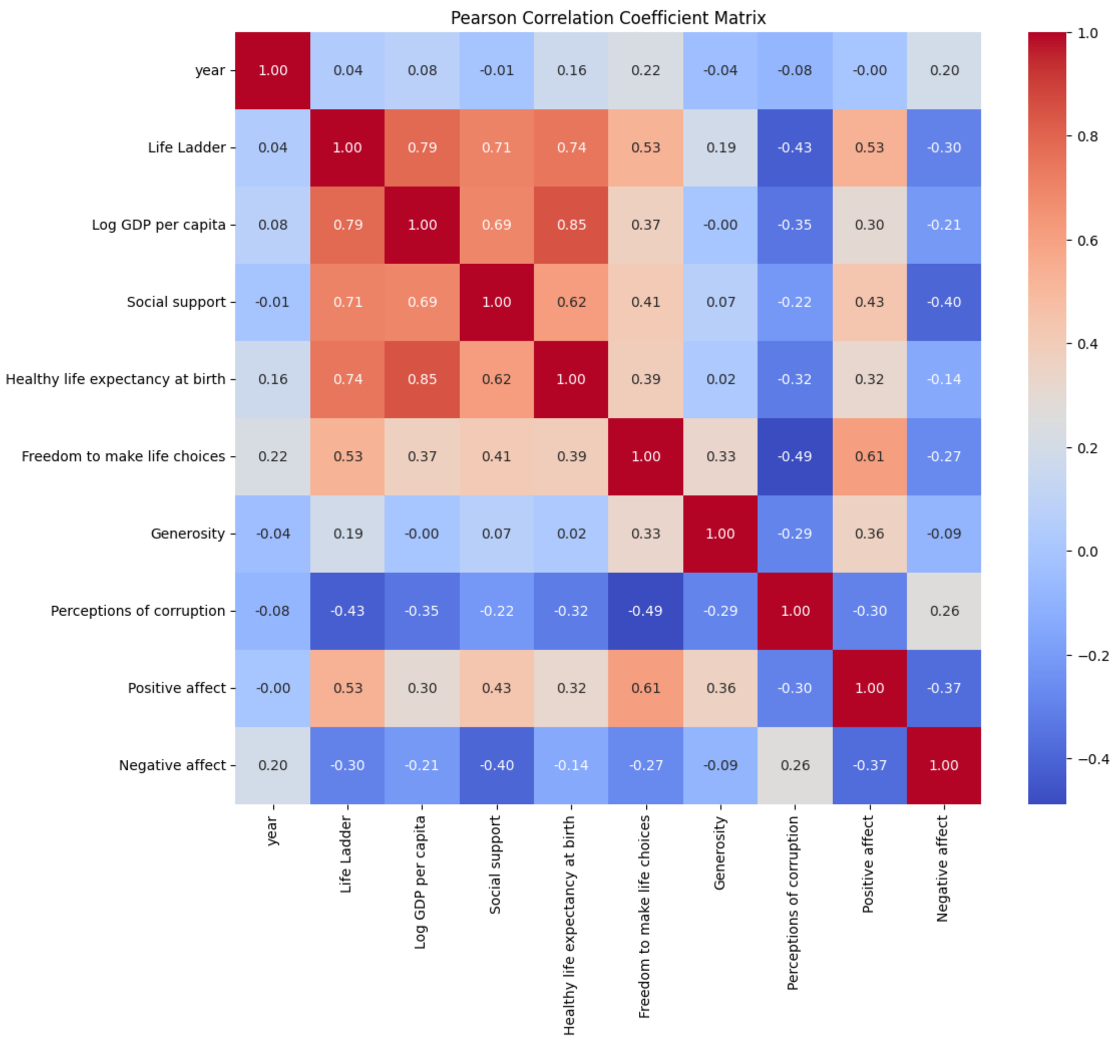
```
                                   Perceptions of corruption  Positive affect  \
year                                               -0.081478        -0.003245
Life Ladder                                        -0.427245         0.532273
Log GDP per capita                                 -0.345511         0.302282
Social support                                     -0.219040         0.432152
Healthy life expectancy at birth                   -0.322461         0.318247
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

                                   Negative affect
year                                      0.196869
Life Ladder                              -0.297488
Log GDP per capita                       -0.210781
Social support                           -0.395865
Healthy life expectancy at birth         -0.139477
Freedom to make life choices             -0.267661
Generosity                               -0.092542
Perceptions of corruption                 0.264225
Positive affect                          -0.374439
Negative affect                           1.000000
```

[15]:
```
corr_matrix["Life Ladder"].sort_values(ascending=False)
```

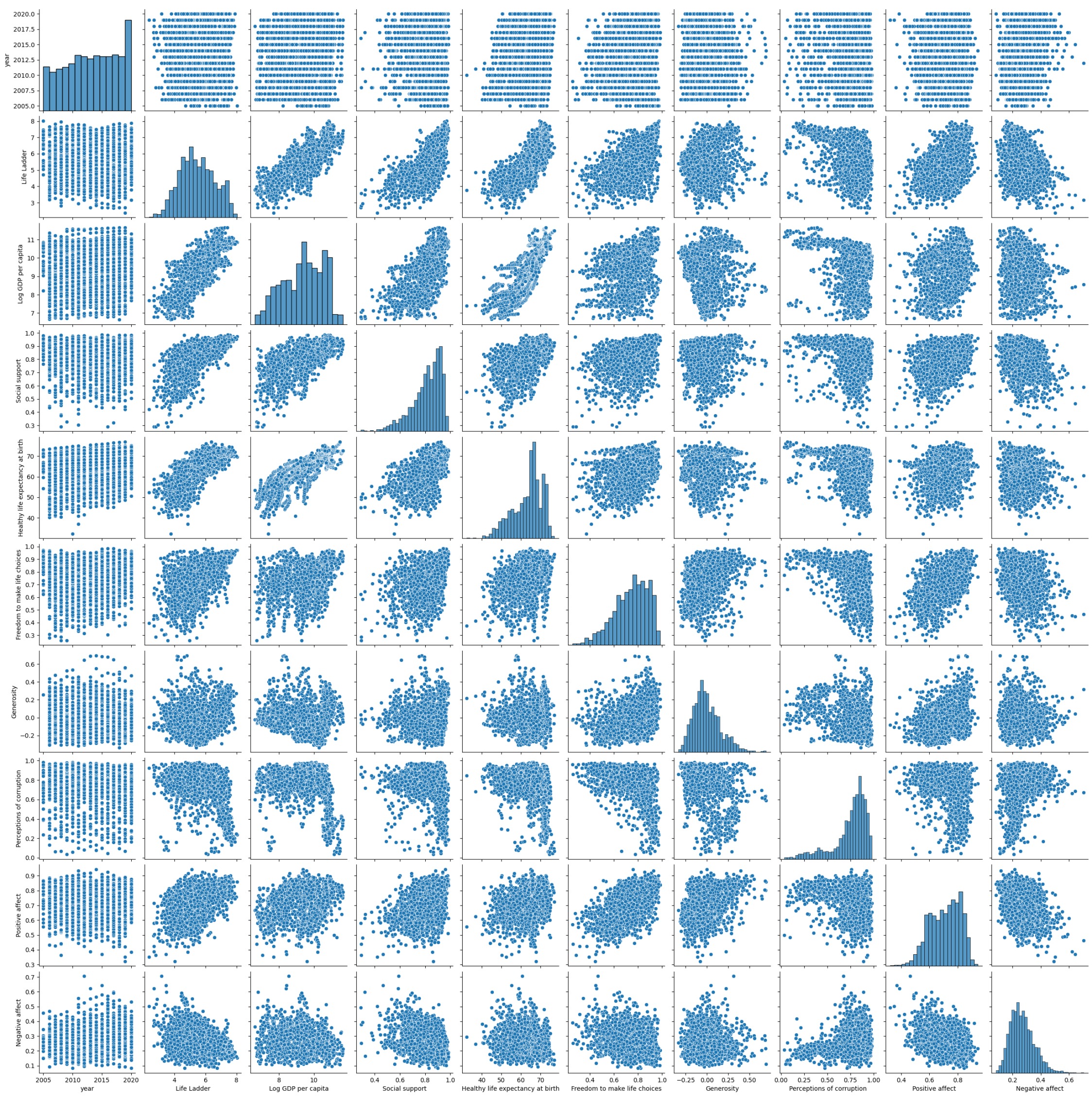
[15]:
```
Life Ladder                         1.000000
Log GDP per capita                  0.790166
Healthy life expectancy at birth    0.744506
Social support                      0.707806
Positive affect                     0.532273
Freedom to make life choices        0.528063
Generosity                          0.190632
year                                0.035515
Negative affect                    -0.297488
Perceptions of corruption          -0.427245
Name: Life Ladder, dtype: float64
```

There is a strong positive corelation between the label and the "Log GDP per capita", "Healthy life expectancy at birth" and "Social support". There is a small negative correlation between the label and "Perceptions of corruption".

[16]:
```
plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Pearson Correlation Coefficient Matrix')
plt.show()
```



[17]:
```
sns.pairplot(data,kind='scatter')
plt.show()
```



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.:

To select 20% of the data for testing and ensure that the test portion is representative of the entire dataset, we use the train_test_split function from sklearn.model_selection. This function splits arrays or matrices into random train and test subsets. By specifying the test_size parameter as 0.2, we ensure that 20% of the data is used for testing. To make the split representative, we use the random_state parameter to ensure reproducibility, this means that every time you run your code with the same random_state, you get the same output, even though the operation involves randomness..

[19]:
```
from sklearn.model_selection import train_test_split
```

[22]:
```
# Data Splitting
y=data['Life Ladder']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[23]:
```
# Verify representativeness of the test data
print("\nTraining data shape:", X_train.shape)
print("Testing data shape:", X_test.shape)

print("\nTraining data shape:", y_train.shape)
print("Testing data shape:", y_test.shape)
```

```

Training data shape: (1559, 9)
Testing data shape: (390, 9)

Training data shape: (1559,)
Testing data shape: (390,)
```

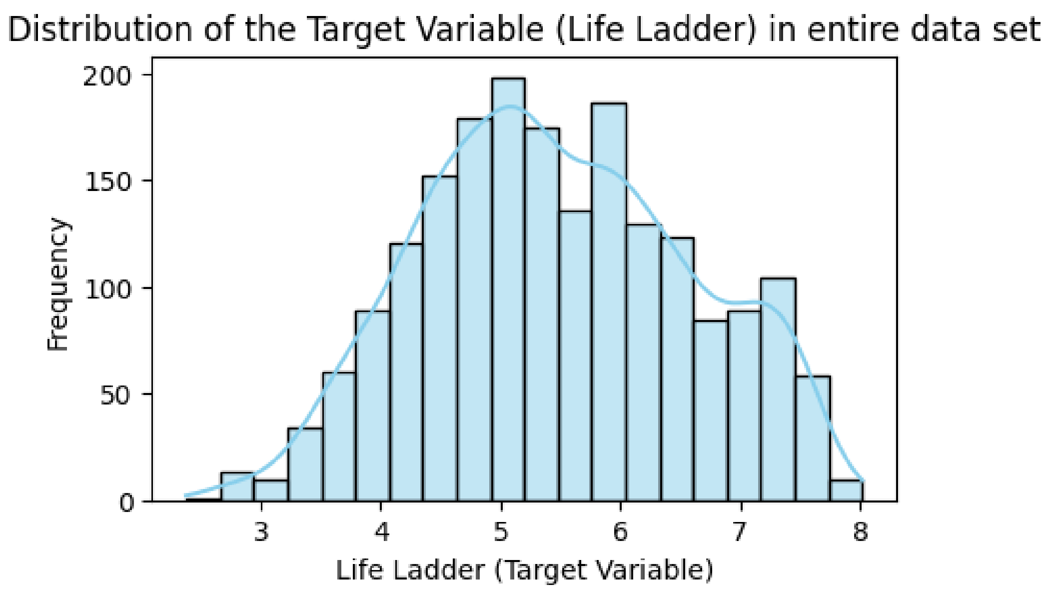
[24]:
```
# Compare means and standard deviations between training and test set for 'Life␣
 ↪ladder'
print(f"Training set 'Life ladder' mean: {y_train.mean()}, std: {y_train.
 ↪std()}")
print(f"Test set 'Life ladder' mean: {y_test.mean()}, std: {y_test.std()}")
```

```
Training set 'Life ladder' mean: 5.469320718409237, std: 1.1132796394483582
Test set 'Life ladder' mean: 5.456251282051283, std: 1.1267538757830755
```

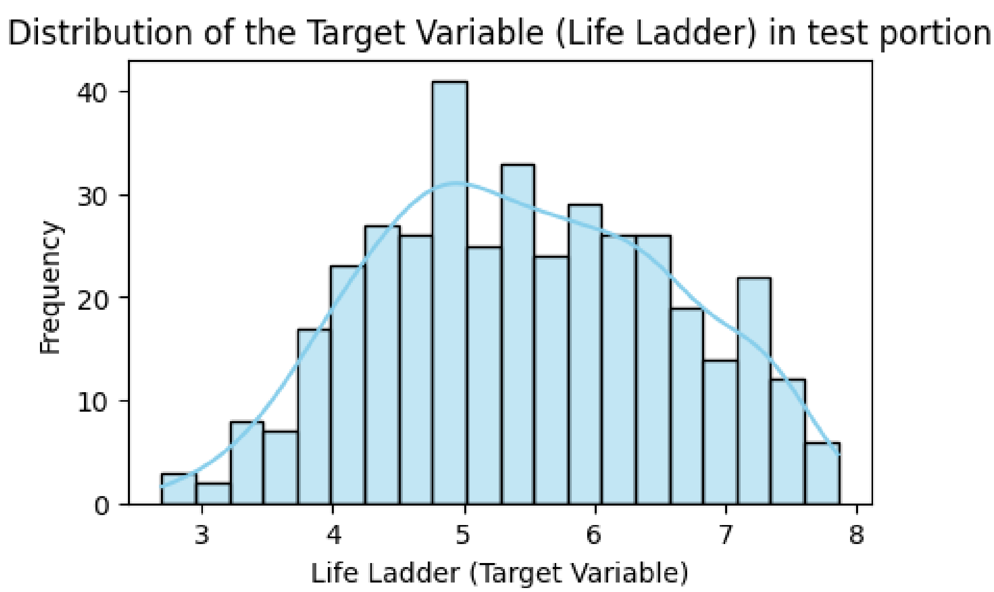
Mean: The average 'Life ladder' score in the training set is approximately 5.46, while in the test set, it is about 5.45. The slight difference between these means suggests that, on average, the happiness levels in both subsets of the data are nearly identical. Standard Deviation: The standard deviation measures the dispersion or variability of the dataset. For the 'Life ladder' variable, the training set has a standard deviation of approximately 1.13, and the test set has a standard deviation of about 1.27. These values are very close, indicating that the range and distribution of happiness scores in both the training and test sets are similar

Representativeness: The close alignment in the means and standard deviations between the training and test sets suggests that the test set is a good representation of the overall dataset. .

[25]:
```
#Plot the distribution of both sets to see the distribution

plt.figure(figsize=(5, 3))
sns.histplot(data['Life Ladder'], bins=20, kde=True, color='skyblue')
plt.xlabel('Life Ladder (Target Variable)')
plt.ylabel('Frequency')
plt.title('Distribution of the Target Variable (Life Ladder) in entire data␣
 ↪set')
plt.show()
```



[26]:
```python
plt.figure(figsize=(5, 3))
sns.histplot(y_test, bins=20, kde=True, color='skyblue')
plt.xlabel('Life Ladder (Target Variable)')
plt.ylabel('Frequency')
plt.title('Distribution of the Target Variable (Life Ladder) in test portion')
plt.show()
```



The distribution is very similar which can infer that test set is representation of 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.

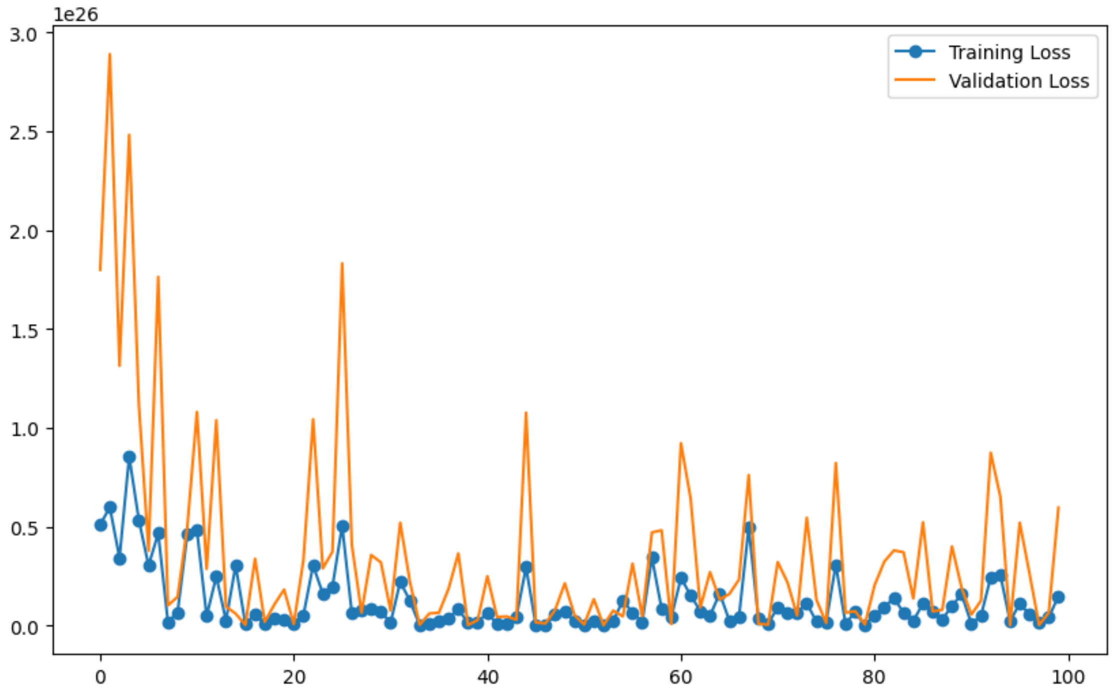
[38]:
```
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

kf = KFold(n_splits=4)
rmse_values = []
r2_values = []

for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

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

    y_pred = model.predict(X_test)

    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    rmse_values.append(rmse)

    r2 = r2_score(y_test, y_pred)
    r2_values.append(r2)

average_rmse = np.mean(rmse_values)
average_r2 = np.mean(r2_values)

print("Average RMSE:", average_rmse)
print("Average R^2:", average_r2)
```

```
Average RMSE: 0.5585263814281751
Average R^2: 0.7449603815317936
```

Model Fit: This value represents the model's fit to the training data. A lower RMSE indicates a better fit. In this case, an RMSE of approximately 0.545 suggests that, on average, the model's predictions are within 0.545 units of the actual 'Life Ladder' scores on the training set. Overfitting/Underfitting: Since this is relatively close to the cross-validated RMSE, it suggests that the model is not significantly overfitting to the training data. Overfitting would be indicated by a much lower training RMSE compared to the cross-validation RMSE.

The RMSE of the model is significantly lower than the standard deviation of the 'Life Ladder' scores. This indicates that your model has good predictive power.

[39]:
```
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_predict

model = SGDRegressor(max_iter=100, tol=1e-3, penalty='l2', alpha = 0.1)
tloss=[]
vloss=[]

for i in range(100):
    model.partial_fit(X_train, y_train)
    tloss.append(mean_squared_error(y_train, model.predict(X_train)))
    vloss.append(mean_squared_error(y_test, model.predict(X_test)))

plt.figure(figsize=(10,6))
plt.plot(tloss, label='Training Loss', marker='o')
plt.plot(vloss, label='Validation Loss')
plt.legend()
plt.show()
```



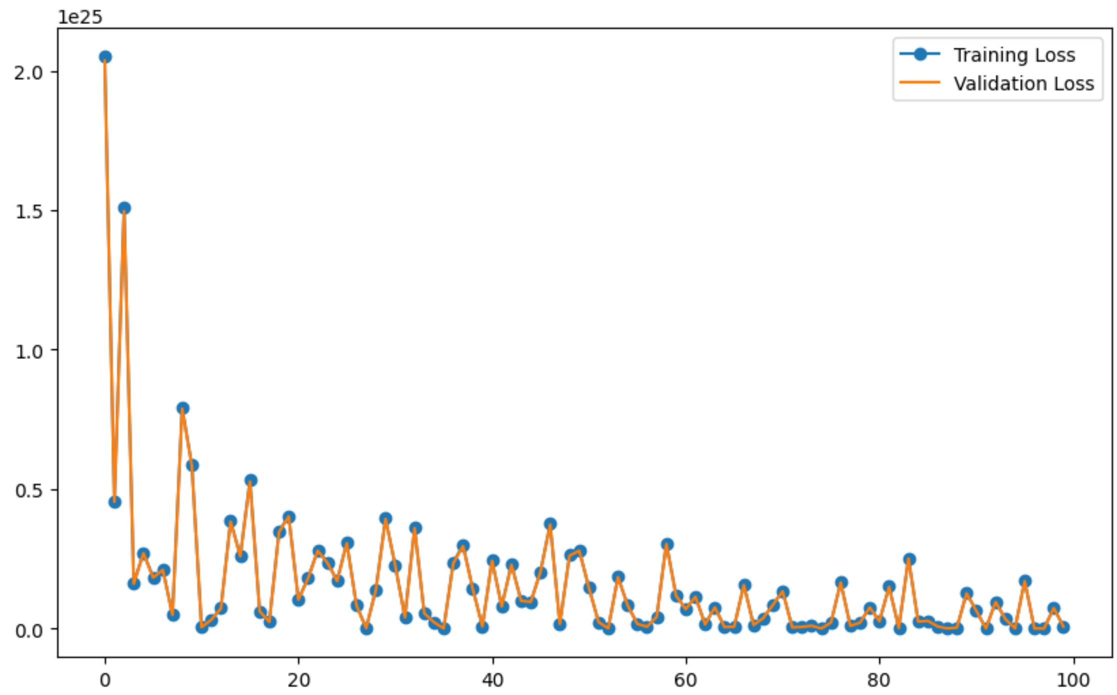
.

[41]:
```
#Model is being overfit, so we drop Country Name and try performing LR again,
↪to see if the performance is improved

X1 = X.drop(['Country name'],axis=1)
```

```
X_train, X_test, y_train, y_test = train_test_split(X1, y, test_size=0.2,␣
 ↪random_state=42)
rmse_values = []
r2=[]
for train_index, test_index in kf.split(X1):
    X_train, X_test = X1.iloc[train_index], X1.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    model = LinearRegression()
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    rmse_values.append(rmse)
    r2=r2_score(y_test, y_pred)

average_rmse = np.mean(rmse_values)
average_rmse
r2
```

[41]: 0.7474044975046146

[42]:
```
model = SGDRegressor(max_iter=100, tol=1e-3, penalty='l2', alpha = 0.1)
tloss=[]
vloss=[]

for i in range(100):
    model.partial_fit(X_train, y_train)
    tloss.append(mean_squared_error(y_train, model.predict(X_train)))
    vloss.append(mean_squared_error(y_test, model.predict(X_test)))

plt.figure(figsize=(10,6))
plt.plot(tloss, label='Training Loss', marker='o')
plt.plot(vloss, label='Validation Loss')
plt.legend()
plt.show()
```



Ridge Regularization (L2) We will train models with different values of the regularization strength (alpha) and observe the impact. We'll also consider the effect of the learning rate (eta0) on SGD.

[47]:
```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error

ridge_pipeline = make_pipeline(StandardScaler(), Ridge())

param_grid = {
    'ridge__alpha': [0.01, 0.1, 1.0],
    'ridge__solver': ['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag',␣
 ↪'saga']
}

grid_search = GridSearchCV(ridge_pipeline, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train, y_train)

best_ridge_model = grid_search.best_estimator_
```

```
best_ridge_model.fit(X_train, y_train)


y_pred = best_ridge_model.predict(X_test)

MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)
r2 = r2_score(y_test, y_pred)

print("Mean Absolute Error (MAE):", MAE)
print("Mean Squared Error (MSE):", MSE)
print("Root Mean Squared Error (RMSE):", RMSE)
print("R-Squared (R2):", r2)

print("Best Hyperparameters:", grid_search.best_params_)
```

```
Mean Absolute Error (MAE): 0.43316466348868826
Mean Squared Error (MSE): 0.35989305753410084
Root Mean Squared Error (RMSE): 0.5999108746589786
R-Squared (R2): 0.7473710679452393
Best Hyperparameters: {'ridge__alpha': 1.0, 'ridge__solver': 'sparse_cg'}
```

The RMSE values obtained for the Ridge regularization with different alpha values show a slight decrease as alpha increases from 0.01 to 1.0. This suggests that a bit of regularization helps to improve the performance of the model on the validation set.

Among the tried values, alpha=1.0 gives the lowest RMSE on the validation set, which implies that it might be the most effective regularization strength of the ones tested. However, since the changes in RMSE are quite small, it suggests that the dataset might not be very sensitive to these values of alpha, or the model is not overfitting much to begin with.

[48]:
```
ridge_model = Ridge(alpha=1.0, solver='lsqr')
ridge_model.fit(X_train, y_train)
y_pred = ridge_model.predict(X_test)
MAE= mean_absolute_error(y_test, y_pred)
MSE= mean_squared_error(y_test, y_pred)
RMSE= np.sqrt(MSE)

r2=r2_score(y_test, y_pred)
print("R-Squared (R2):", r2)
```

```
R-Squared (R2): 0.7486818170050427
```

[49]:
```
pd.DataFrame([MAE, MSE, RMSE], index=['MAE', 'MSE', 'RMSE'], columns=['Ridge␣
↪Metrics'])
```

[49]:

```
      Ridge Metrics
MAE        0.432223
MSE        0.358026
RMSE       0.598353
```

[50]:

```
#Lasso Regularization (L1)

lasso_pipeline = make_pipeline(StandardScaler(), Lasso())

param_grid = {
    'lasso__alpha': [0.01, 0.1, 1.0],
    'lasso__max_iter': [1000, 2000, 3000],
    'lasso__tol': [1e-3, 1e-4, 1e-5]
}

grid_search = GridSearchCV(lasso_pipeline, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train, y_train)

best_lasso_model = grid_search.best_estimator_

best_lasso_model.fit(X_train, y_train)

y_pred = best_lasso_model.predict(X_test)

MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)
r2 = r2_score(y_test, y_pred)

print("Mean Absolute Error (MAE):", MAE)
print("Mean Squared Error (MSE):", MSE)
print("Root Mean Squared Error (RMSE):", RMSE)
print("R-Squared (R2):", r2)

print("Best Hyperparameters:", grid_search.best_params_)
```

```
Mean Absolute Error (MAE): 0.43378244842420854
Mean Squared Error (MSE): 0.3594629507358979
Root Mean Squared Error (RMSE): 0.5995522919111376
R-Squared (R2): 0.7476729838028109
Best Hyperparameters: {'lasso__alpha': 0.01, 'lasso__max_iter': 1000,
'lasso__tol': 0.001}
```

From these results, it seems that the model performs best (lowest RMSE) with alpha=0.01, indicating that this level of regularization provides the best balance between bias and variance in the model. Lower values of alpha tend to reduce overfitting, but too much regularization (higher alpha values) can lead to underfitting.

[51]:
```
lasso_model = Lasso(alpha=0.01, max_iter=1000, tol=0.001 )
lasso_model.fit(X_train, y_train)
y_pred = lasso_model.predict(X_test)
MAE= mean_absolute_error(y_test, y_pred)
MSE= mean_squared_error(y_test, y_pred)
RMSE= np.sqrt(MSE)

r2=r2_score(y_test, y_pred)
print("R-Squared (R2):", r2)
```

```
R-Squared (R2): 0.7444422089617624
```

[52]:
```
pd.DataFrame([MAE, MSE, RMSE], index=['MAE', 'MSE', 'RMSE'], columns=['Lasso␣
↪Metrics'])
```

[52]:

	Lasso Metrics
MAE	0.449278
MSE	0.364065
RMSE	0.603378

[53]:
```
# Elastic Net Regularization

en_pipeline = make_pipeline(StandardScaler(), ElasticNet())

param_grid = {
    'elasticnet__alpha': [0.01, 0.1, 1.0],
    'elasticnet__l1_ratio': [0.1, 0.5, 0.9],
    'elasticnet__max_iter': [1000, 2000, 3000],
    'elasticnet__tol': [1e-3, 1e-4, 1e-5]
}

grid_search = GridSearchCV(en_pipeline, param_grid, cv=5,␣
↪scoring='neg_mean_squared_error', n_jobs=-1)
grid_search.fit(X_train, y_train)

best_en_model = grid_search.best_estimator_

best_en_model.fit(X_train, y_train)

y_pred = best_en_model.predict(X_test)

MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)
r2 = r2_score(y_test, y_pred)

print("Mean Absolute Error (MAE):", MAE)
```

```
print("Mean Squared Error (MSE):", MSE)
print("Root Mean Squared Error (RMSE):", RMSE)
print("R-Squared (R2):", r2)

print("Best Hyperparameters:", grid_search.best_params_)
```

```
Mean Absolute Error (MAE): 0.440768512031975
Mean Squared Error (MSE): 0.366540409143639
Root Mean Squared Error (RMSE): 0.6054258081248594
R-Squared (R2): 0.7427049225363327
Best Hyperparameters: {'elasticnet__alpha': 0.1, 'elasticnet__l1_ratio': 0.1,
'elasticnet__max_iter': 1000, 'elasticnet__tol': 0.001}
```

The Elastic Net model also performs best with alpha=0.1, providing the lowest RMSE.

[54]:
```
EN_model = ElasticNet(alpha=0.1, l1_ratio=0.1, max_iter=1000, tol=0.001)
EN_model.fit(X_train, y_train)
y_pred = EN_model.predict(X_test)
MAE= mean_absolute_error(y_test, y_pred)
MSE= mean_squared_error(y_test, y_pred)
RMSE= np.sqrt(MSE)

r2=r2_score(y_test, y_pred)
print("R-Squared (R2):", r2)
```

```
R-Squared (R2): 0.6786329846062364
```

[55]:
```
pd.DataFrame([MAE, MSE, RMSE], index=['MAE', 'MSE', 'RMSE'],␣
↪columns=['ElasticNet Metrics'])
```

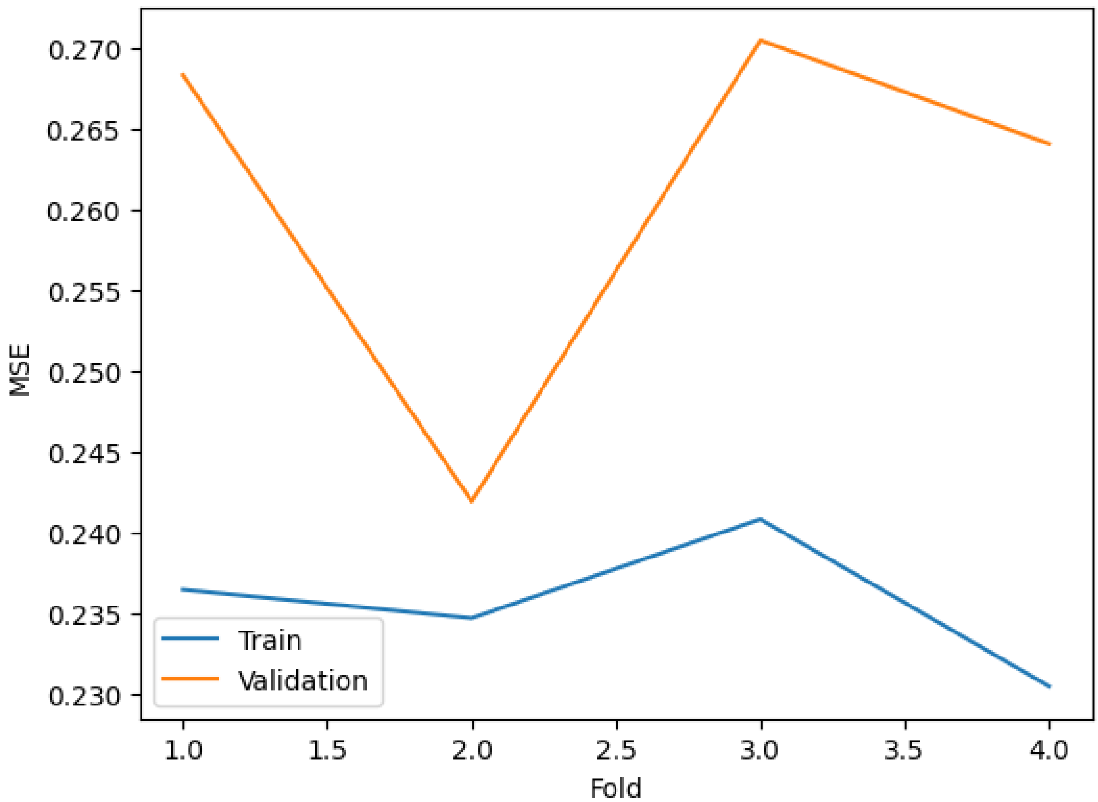
[55]:

	ElasticNet Metrics
MAE	0.540574
MSE	0.457817
RMSE	0.676622

F. Train a Polynomial 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.

[57]:
```
from sklearn.preprocessing import PolynomialFeatures

degree = 2
k = 4

train_errors = []
```

```
val_errors = []

kf = KFold(n_splits=k, shuffle=True)

for train_idx, val_idx in kf.split(X):

    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    y_train_pred = model.predict(X_train_poly)
    y_val_pred = model.predict(X_val_poly)

    train_mse = mean_squared_error(y_train, y_train_pred)
    val_mse = mean_squared_error(y_val, y_val_pred)
    r2=r2_score(y_val, y_val_pred)

    train_errors.append(train_mse)
    val_errors.append(val_mse)

plt.plot(range(1, k+1), train_errors, label='Train')
plt.plot(range(1, k+1), val_errors, label='Validation')
plt.xlabel('Fold')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

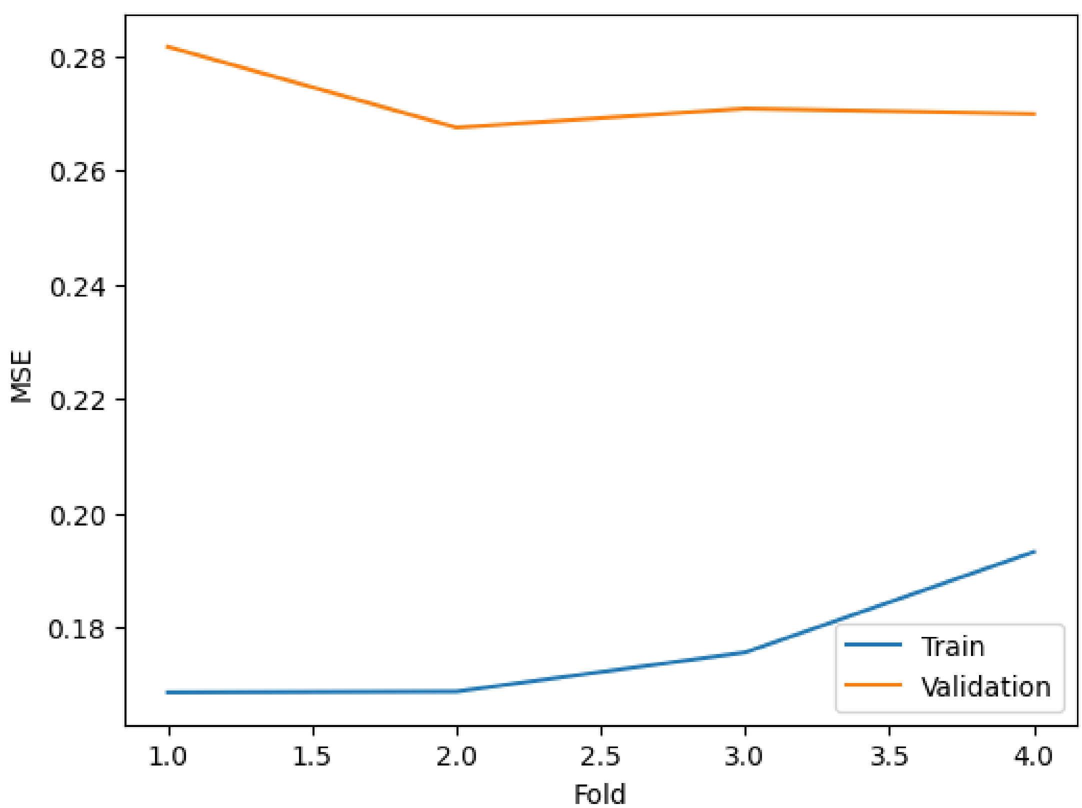


[58]:
```
r2
```

[58]: 0.8004090737908238

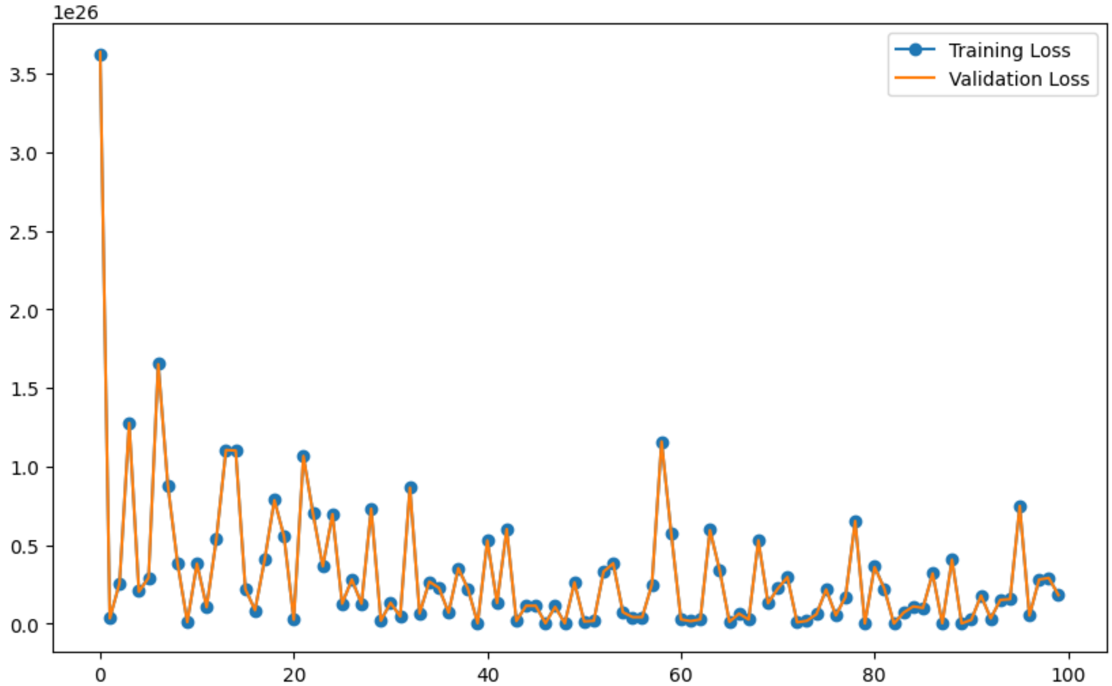
[59]:
```
degree = 3
k = 4

train_errors = []
val_errors = []

kf = KFold(n_splits=k, shuffle=True)

for train_idx, val_idx in kf.split(X):

    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)
```

```
    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    y_train_pred = model.predict(X_train_poly)
    y_val_pred = model.predict(X_val_poly)

    train_mse = mean_squared_error(y_train, y_train_pred)
    val_mse = mean_squared_error(y_val, y_val_pred)
    r2=r2_score(y_val, y_val_pred)

    train_errors.append(train_mse)
    val_errors.append(val_mse)

plt.plot(range(1, k+1), train_errors, label='Train')
plt.plot(range(1, k+1), val_errors, label='Validation')
plt.xlabel('Fold')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



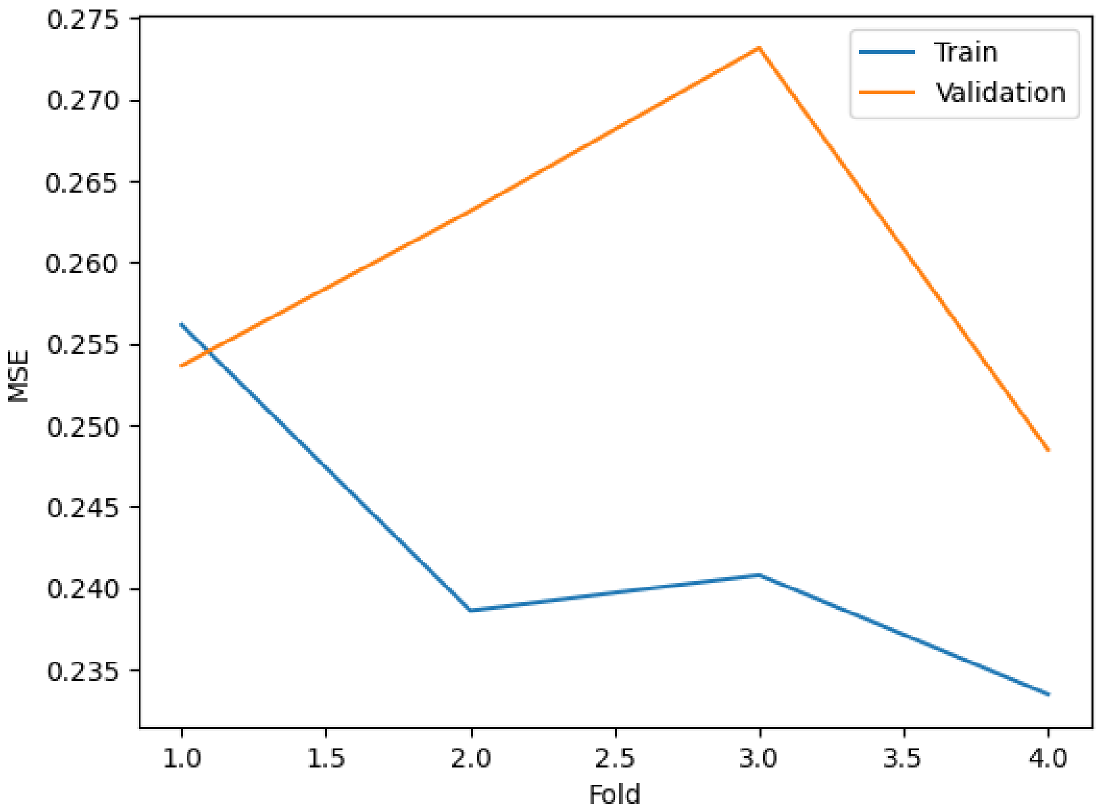
[60]:

```python
degree = 2
k = 4

train_errors = []
val_errors = []

kf = KFold(n_splits=k, shuffle=True)

for train_idx, val_idx in kf.split(X):

    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)

    model = SGDRegressor(max_iter=100, tol=1e-3, penalty='l2', alpha = 0.1)
    tloss=[]
    vloss=[]

    for i in range(100):
        model.partial_fit(X_train, y_train)
        tloss.append(mean_squared_error(y_train, model.predict(X_train)))
        vloss.append(mean_squared_error(y_val, model.predict(X_val)))

plt.figure(figsize=(10,6))
plt.plot(tloss, label='Training Loss', marker='o')
plt.plot(vloss, label='Validation Loss')
plt.legend()
plt.show()
```



[61]:
```
degree = 2
k = 4

train_errors = []
val_errors = []

kf = KFold(n_splits=k, shuffle=True)

for train_idx, val_idx in kf.split(X):

    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    y_train_pred = model.predict(X_train_poly)
    y_val_pred = model.predict(X_val_poly)

    train_mse = mean_squared_error(y_train, y_train_pred)
```

```
    val_mse = mean_squared_error(y_val, y_val_pred)
    r2=r2_score(y_val, y_val_pred)

    train_errors.append(train_mse)
    val_errors.append(val_mse)

plt.plot(range(1, k+1), train_errors, label='Train')
plt.plot(range(1, k+1), val_errors, label='Validation')
plt.xlabel('Fold')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



[62]:
```
alphas = [0.01, 0.1, 1.0]
for alpha in alphas:
    ridge = Ridge(alpha=alpha,max_iter=1000)
    ridge.fit(X_train, y_train)
    y_pred_ridge = ridge.predict(X_val)

    r2_ridge=r2_score(y_val, y_pred_ridge)
    print(f"R-Squared (Ridge) with alpha={alpha}: {r2_ridge}")
```

```
R-Squared (Ridge) with alpha=0.01: 0.7385362957420336
R-Squared (Ridge) with alpha=0.1: 0.73854801067154
R-Squared (Ridge) with alpha=1.0: 0.7382749452409656
```

[65]:
```
for alpha in alphas:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train)
    y_pred_lasso = lasso.predict(X_val)
    r2_lasso=r2_score(y_val, y_pred_ridge)
    print(f"R-Squared (Lasso) with alpha={alpha}: {r2_lasso}")
```

```
R-Squared (Lasso) with alpha=0.01: 0.7382749452409656
R-Squared (Lasso) with alpha=0.1: 0.7382749452409656
R-Squared (Lasso) with alpha=1.0: 0.7382749452409656
```

[66]:
```
for alpha in alphas:
    elastic_net = ElasticNet(alpha=alpha)
    elastic_net.fit(X_train, y_train)
    y_pred_elastic_net = elastic_net.predict(X_val)
    r2_elastic_net=r2_score(y_val, y_pred_elastic_net)
    print(f"R-Squared (Elastic net) with alpha={alpha}: {r2_elastic_net}")
```

```
R-Squared (Elastic net) with alpha=0.01: 0.7198418669573166
R-Squared (Elastic net) with alpha=0.1: 0.5933864344816054
R-Squared (Elastic net) with alpha=1.0: 0.5054962475658304
```

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.

[67]:
```
ridge = Ridge(alpha=0.1,max_iter=1000)
ridge.fit(X_train, y_train)
y_pred_ridge = ridge.predict(X_val)
r2=r2_score(y_val, y_pred_ridge)
print("R2 VALUE:",r2)
```

```
R2 VALUE: 0.73854801067154
```

The best model for regression on the above dataset is using the ridge regularizaion with alpha value=0.1 which reduces the R2 value to 0.73.R2 is the best evaluation metric for the regression model above Future scope could include improving the R2 value of the model and increasing the dataset to prevent overfitting and evaluating more parameters using grid search or randomized search1

0.1 Further Exploration

Logarithimic algorithms can be used to normalise the data. Techniques like grid search or random search can be used to efficiently explore the hyperparameter space and find the optimal combination. Experimenting with different algorithms can help to find the one that best captures the underlying relationships in the data.

0.2 References

Chatgpt Stack Overflow Medium Towards Data Science Scikit Learn Documentation