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

In [ ]:
```
!pip install category_encoders
```

```
Requirement already satisfied: category_encoders in /usr/local/lib/python3.10/
dist-packages (2.6.2)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.10/dist
-packages (from category_encoders) (1.23.5)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.
10/dist-packages (from category_encoders) (1.2.2)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.10/dist-
packages (from category_encoders) (1.11.2)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.1
0/dist-packages (from category_encoders) (0.14.0)
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.10/dist
-packages (from category_encoders) (1.5.3)
Requirement already satisfied: patsy>=0.5.1 in /usr/local/lib/python3.10/dist-
packages (from category_encoders) (0.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python
3.10/dist-packages (from pandas>=1.0.5->category_encoders) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.0.5->category_encoders) (2023.3.post1)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
(from patsy>=0.5.1->category_encoders) (1.16.0)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
-packages (from scikit-learn>=0.20.0->category_encoders) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
10/dist-packages (from scikit-learn>=0.20.0->category_encoders) (3.2.0)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/di
st-packages (from statsmodels>=0.9.0->category_encoders) (23.1)
```

In [ ]:
```
df = pd.read_csv("/content/happiness_data.csv")
```

In [ ]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

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

In [ ]:
```
df
```

Out[ ]:

	Country name	year	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptio corrupti
0	Afghanistan	2008	3.724	7.370	0.451	50.80	0.718	0.168	0.8
1	Afghanistan	2009	4.402	7.540	0.552	51.20	0.679	0.190	0.8
2	Afghanistan	2010	4.758	7.647	0.539	51.60	0.600	0.121	0.7
3	Afghanistan	2011	3.832	7.620	0.521	51.92	0.496	0.162	0.7
4	Afghanistan	2012	3.783	7.705	0.521	52.24	0.531	0.236	0.7
...	...	...	...	...	...	...	...	...	
1944	Zimbabwe	2016	3.735	7.984	0.768	54.40	0.733	-0.095	0.7
1945	Zimbabwe	2017	3.638	8.016	0.754	55.00	0.753	-0.098	0.7
1946	Zimbabwe	2018	3.616	8.049	0.775	55.60	0.763	-0.068	0.8
1947	Zimbabwe	2019	2.694	7.950	0.759	56.20	0.632	-0.064	0.8
1948	Zimbabwe	2020	3.160	7.829	0.717	56.80	0.643	-0.009	0.7

1949 rows × 11 columns

How much data is present?

1949 Rows and 11 Columns

In [ ]:
```
df.shape
```

Out[ ]:
```
(1949, 11)
```

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['Country name', 'year', 'Life Ladder', 'Log GDP per capita',
       'Social support', 'Healthy life expectancy at birth',
       'Freedom to make life choices', 'Generosity',
       'Perceptions of corruption', 'Positive affect', 'Negative affect'],
      dtype='object')
```

What attributes/features are continuous valued?

- Everything except "Country name" and "year" is continuous valued.
- Country Name is a categorical attribute
- "year" is a Discrete attribute

Which attributes are categorical?

- Country Name is a categorical attribute

In [ ]:
```
df.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
```

Missing values

In [ ]:
```
df.isna().sum()
```

Out[ ]:
```
Country name                        0
year                                0
Life Ladder                         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
```

We are dropping the rows which contain Null or NaN because we do not have enough context on the data and any assumption for replacing those could introduce bias in the data. (Part B - special treatment for missing values)

In [ ]:
```
## dropping nan values and rows
df.dropna(how="any", axis = 0, inplace = True)
```

In [ ]:
```
## removing year column as mentioned by the question

df.drop(columns = ["year"], errors = "ignore", inplace = True)
```

In [ ]:
```
df_numerical = df.select_dtypes(include=np.number)
```

In [ ]:
```
df_numerical.head()
```

Out[ ]:

	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption	Positive affect	Negative affect
0	3.724	7.370	0.451	50.80	0.718	0.168	0.882	0.518	0.258
1	4.402	7.540	0.552	51.20	0.679	0.190	0.850	0.584	0.237
2	4.758	7.647	0.539	51.60	0.600	0.121	0.707	0.618	0.275
3	3.832	7.620	0.521	51.92	0.496	0.162	0.731	0.611	0.267
4	3.783	7.705	0.521	52.24	0.531	0.236	0.776	0.710	0.268

Part 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?

Statistical distribution of numerical columns

In [ ]:
```
df.describe()
```

Out[ ]:

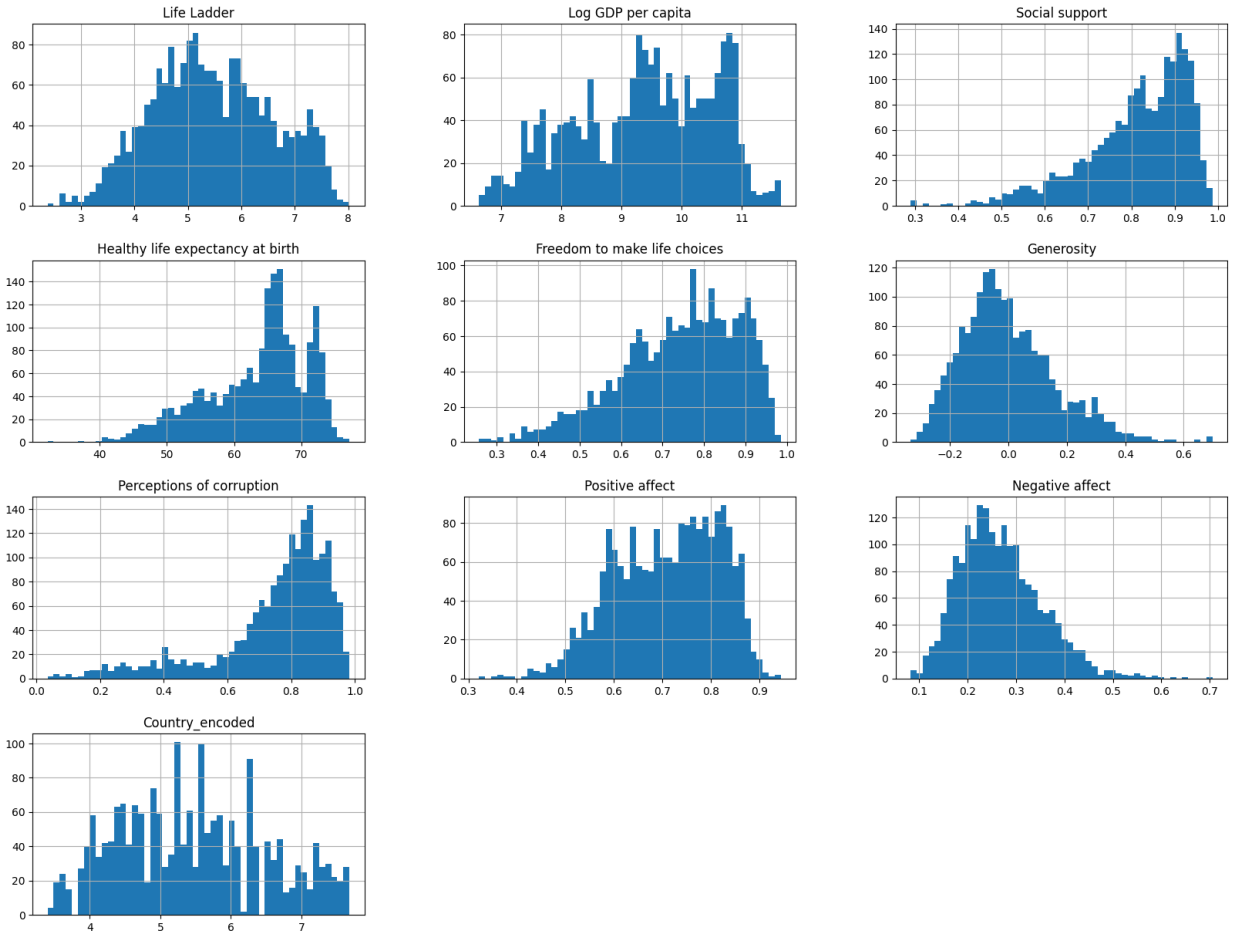
	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Percep corru
count	1708.000000	1708.000000	1708.000000	1708.000000	1708.000000	1708.000000	1708.0(
mean	5.446680	9.321709	0.810321	63.225465	0.739442	-0.000638	0.75
std	1.136592	1.158344	0.121638	7.687011	0.142846	0.162103	0.18
min	2.375000	6.635000	0.290000	32.300000	0.258000	-0.335000	0.03
25%	4.595000	8.394000	0.741000	58.175000	0.644000	-0.111250	0.6(
50%	5.364000	9.456500	0.835000	65.100000	0.757500	-0.025500	0.8(
75%	6.259000	10.272000	0.908000	68.685000	0.852000	0.089000	0.8
max	7.971000	11.648000	0.987000	77.100000	0.985000	0.689000	0.9(

In [ ]:
```
## Target encoding Categorical column Country name
import category_encoders as ce

# Initialize and fit the TargetEncoder
encoder = ce.TargetEncoder(cols=['Country name'])
encoder.fit(df['Country name'], df['Life Ladder'])

# Transform the 'Category' column
df['Country name encoded'] = encoder.transform(df['Country name'])
```

We are target encoding the Country name variable because label encoding would give it an irrelevant inherent order and one hot encoding would introduce far too many features in the feature space. Target encoding calculates the grouped mean of each unique value in the column and encodes it using the mean values.

Histogram visualisation for data distribution of attribute

In [ ]:
```
df_numerical.hist(bins=50, figsize=(20,15))
plt.show()
```



Are there any attributes that might require special treatment?

Missing values, outliers, scaling etc.

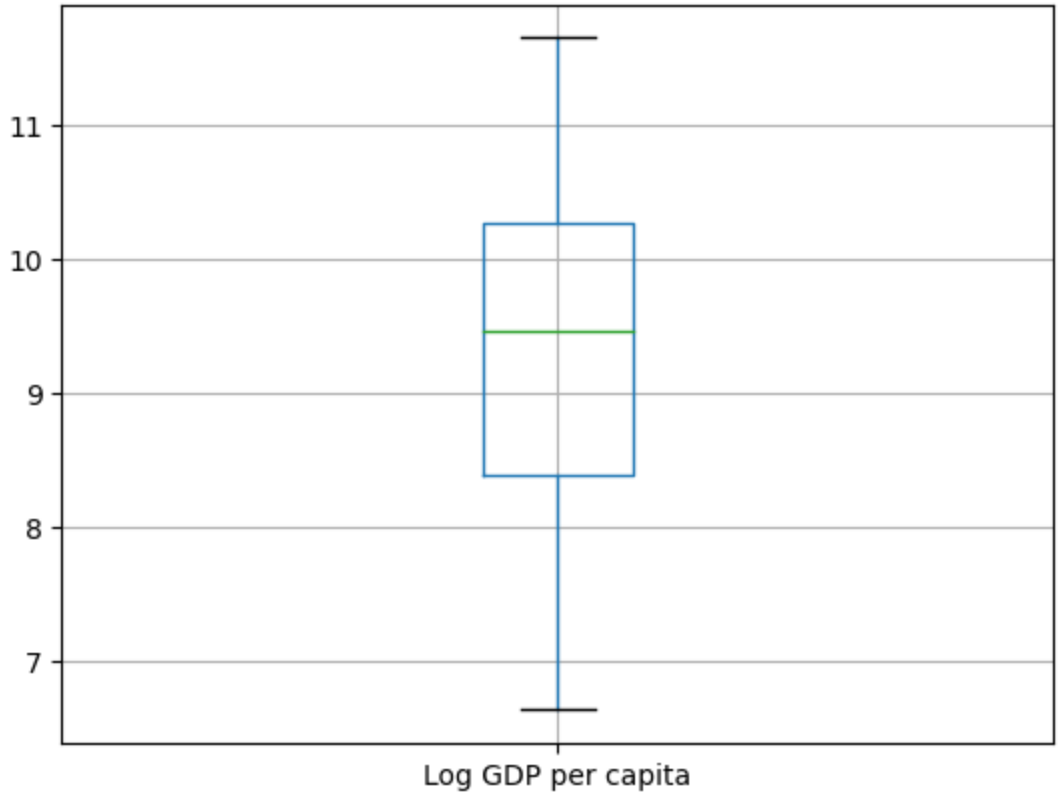
Special Treatment - Because of skewed distributions (as we can see below), it is best to apply a feature scaling method such as Normalization or Standardization.

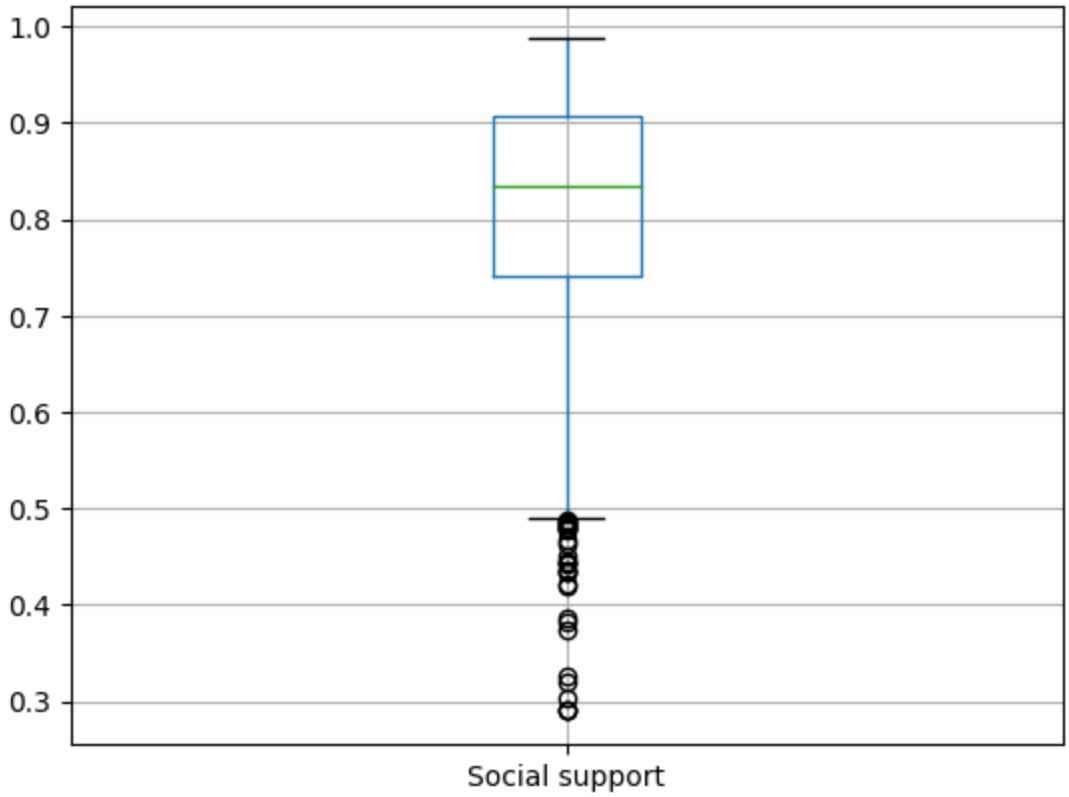
We have also dropped rows containing missing values.

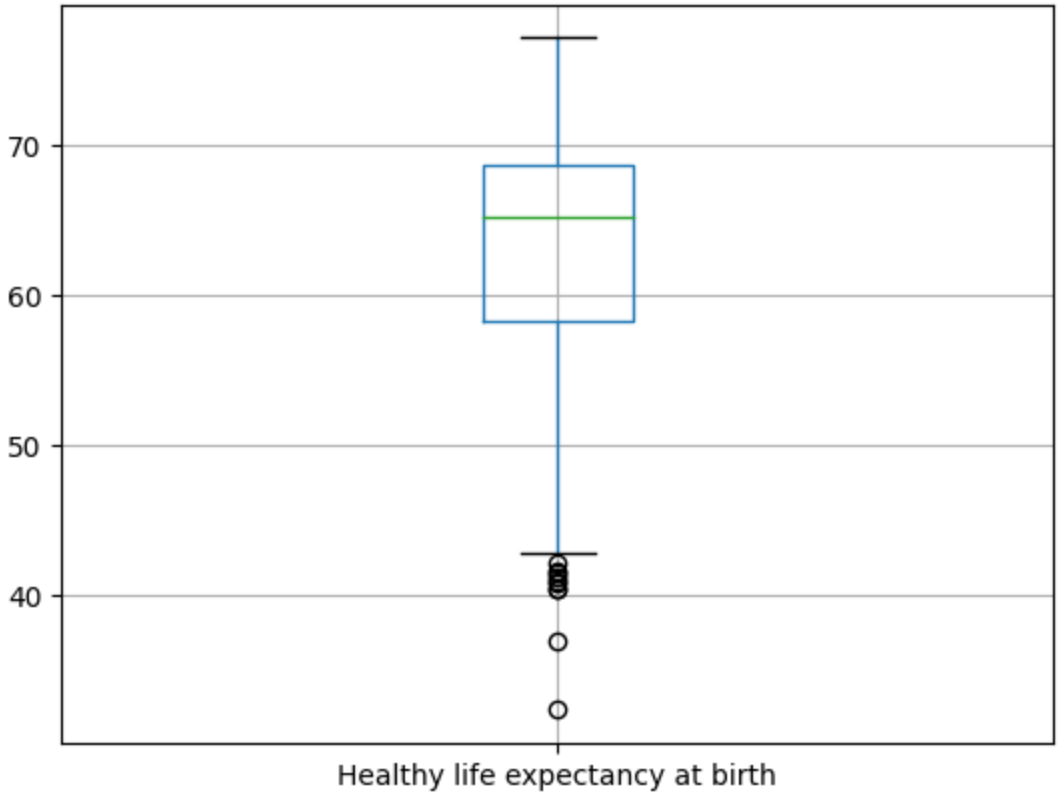
There are some outliers in each attribute but we will keep them because they might contribute in our prediction of life ladder.

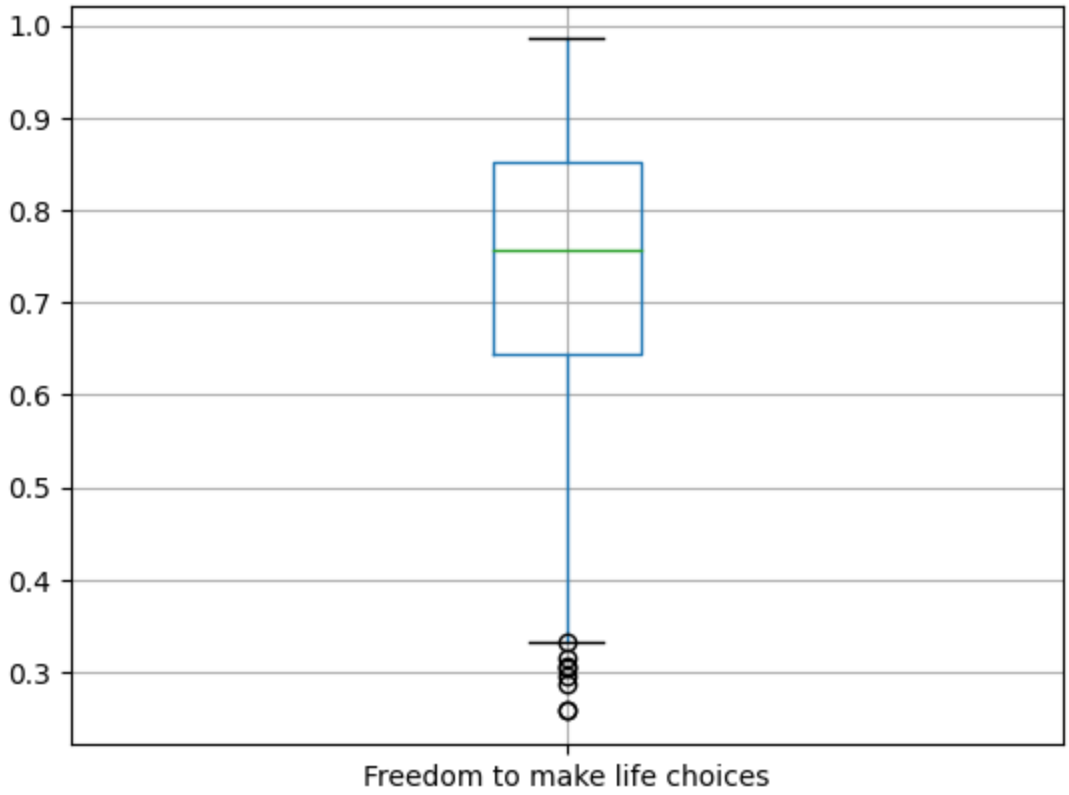
In [ ]:
```
## boxplots
for column in df_numerical:
```

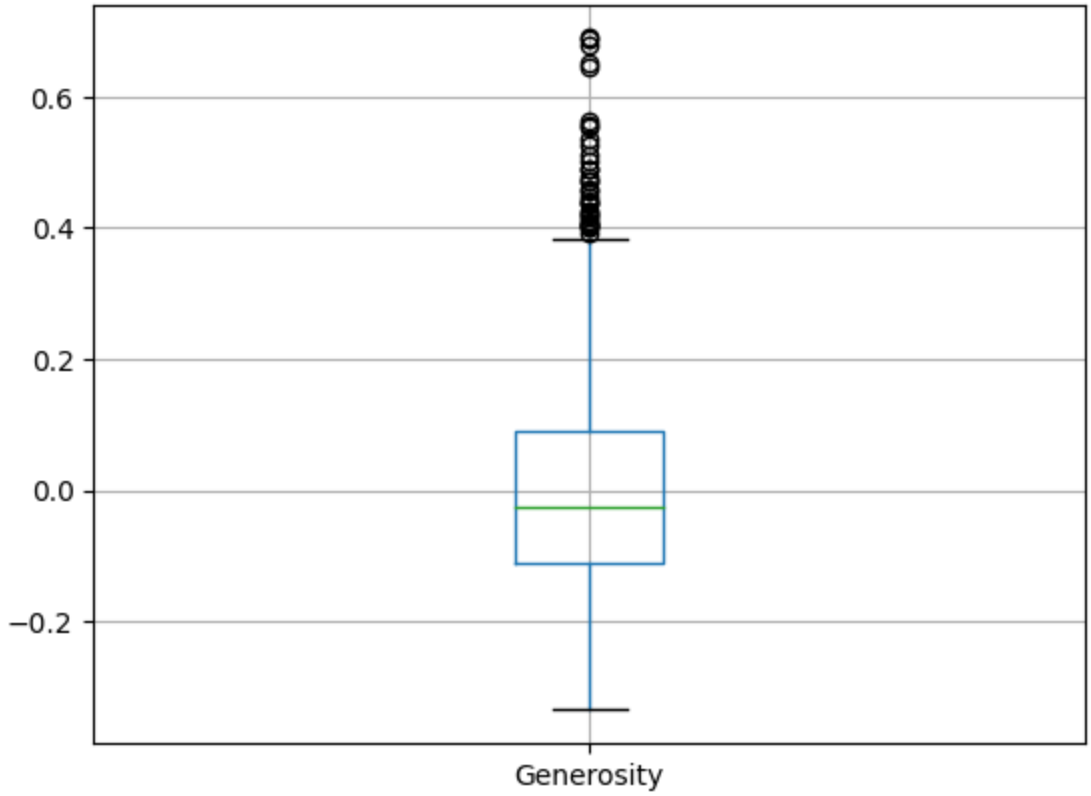
```
    plt.figure()
    df_numerical.boxplot([column])
```

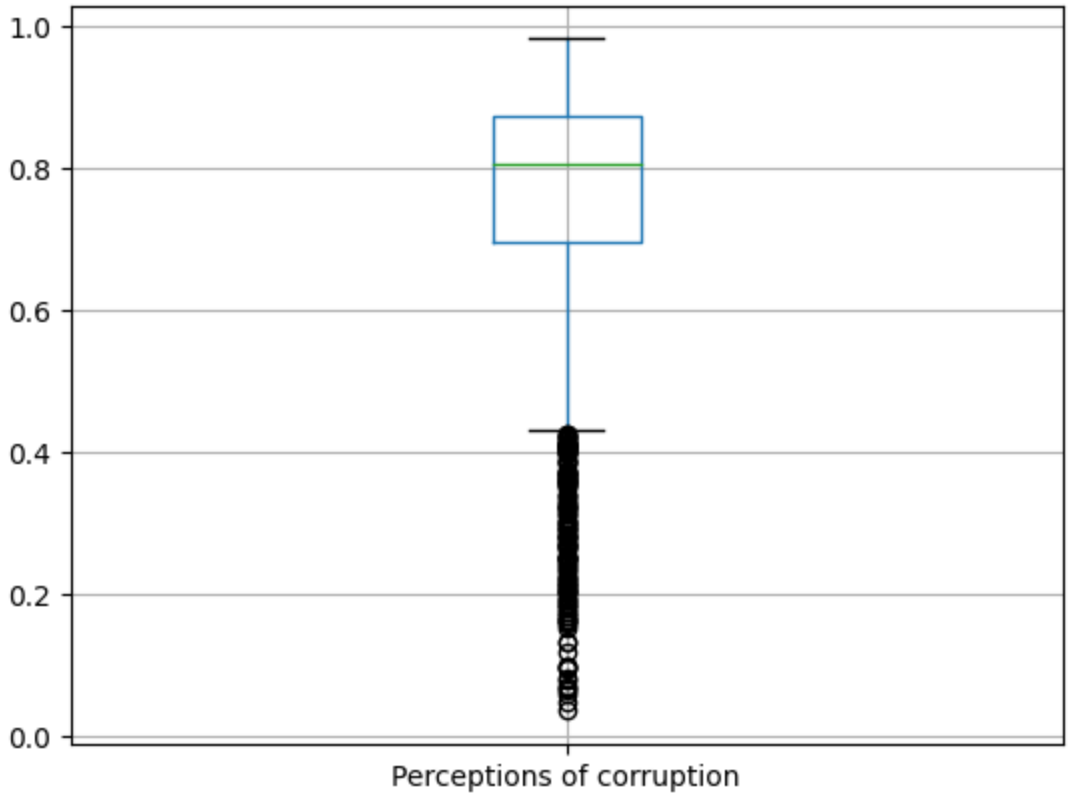


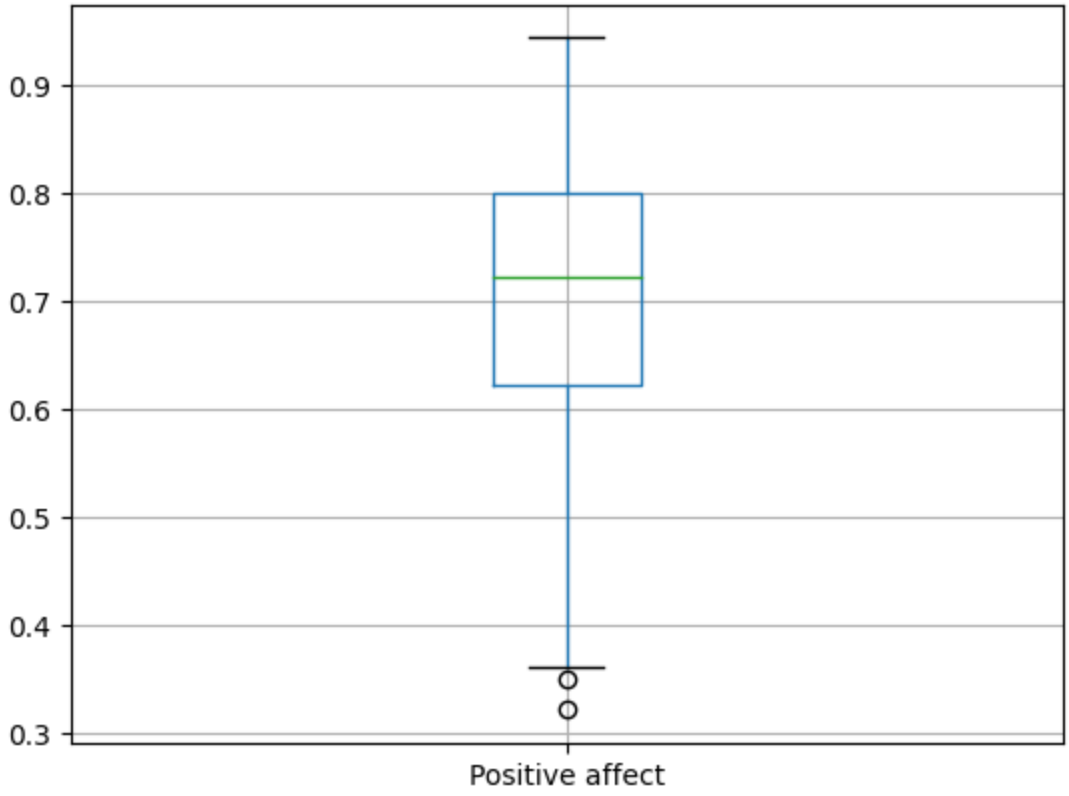

1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Social support

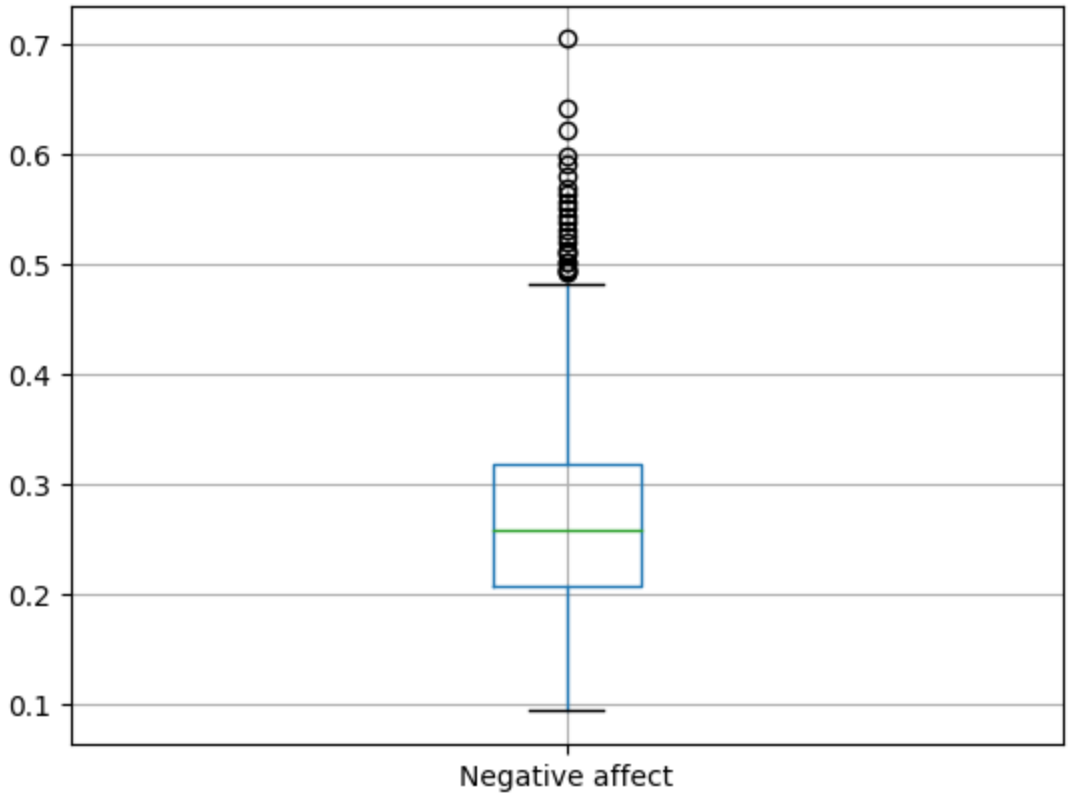

70
60
50
40
Healthy life expectancy at birth


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Freedom to make life choices


0.6
0.4
0.2
0.0
−0.2
Generosity


1.0
0.8
0.6
0.4
0.2
0.0
Perceptions of corruption


0.9
0.8
0.7
0.6
0.5
0.4
0.3
Positive affect



Part 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.

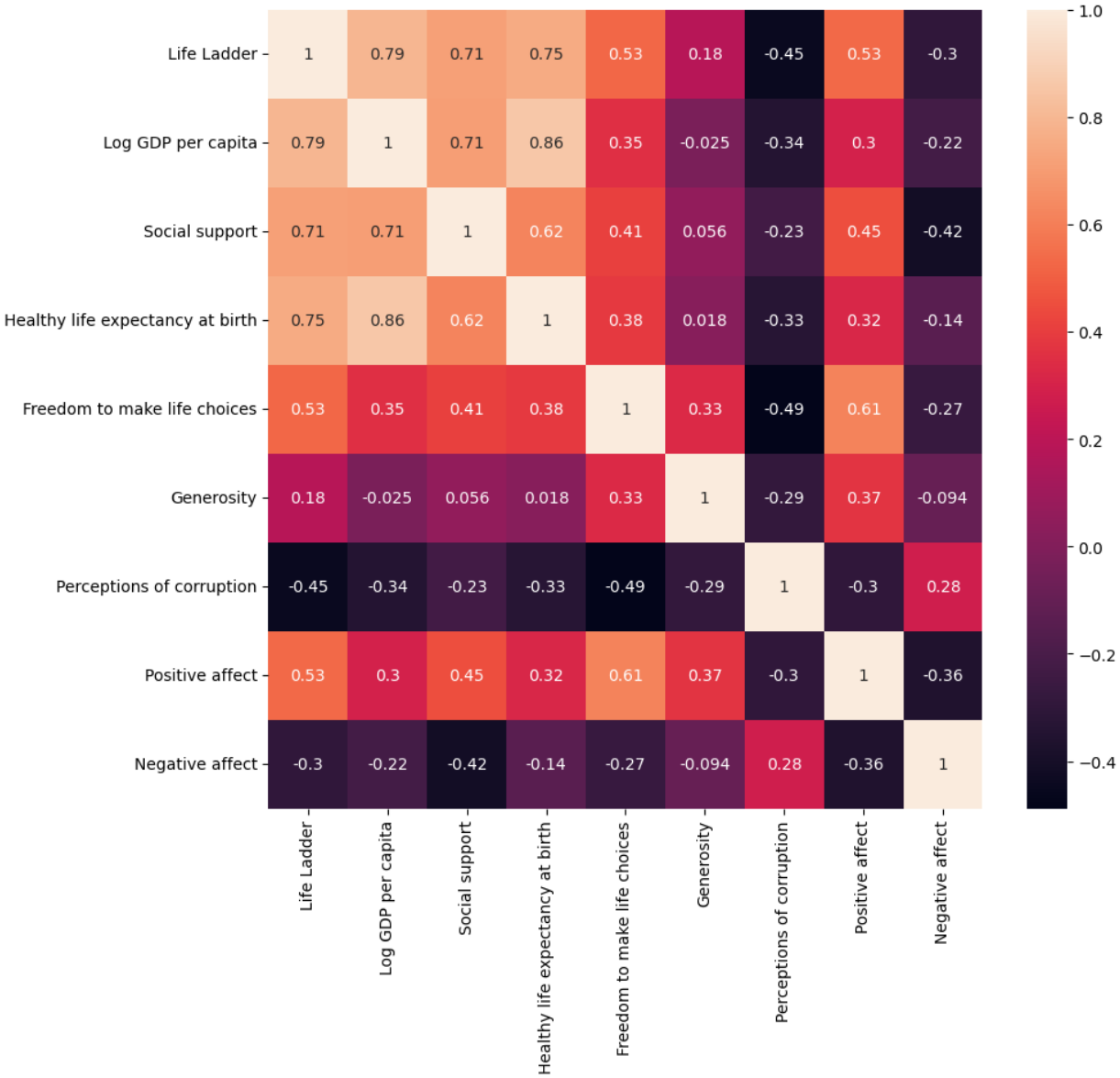
In [ ]:
```
df_numerical.corr()
```

Out[ ]:

	Life Ladder	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption
Life Ladder	1.000000	0.792848	0.713211	0.754697	0.525089	0.182758	-0.448157
Log GDP per capita	0.792848	1.000000	0.705972	0.860345	0.353182	-0.024565	-0.343468
Social support	0.713211	0.705972	1.000000	0.617446	0.411719	0.056131	-0.226855
Healthy life expectancy at birth	0.754697	0.860345	0.617446	1.000000	0.384829	0.018188	-0.334990
Freedom to make life choices	0.525089	0.353182	0.411719	0.384829	1.000000	0.326313	-0.488072
Generosity	0.182758	-0.024565	0.056131	0.018188	0.326313	1.000000	-0.288467
Perceptions of corruption	-0.448157	-0.343468	-0.226855	-0.334990	-0.488072	-0.288467	1.000000
Positive affect	0.533092	0.296473	0.449969	0.318886	0.611673	0.371530	-0.301383
Negative affect	-0.300466	-0.224806	-0.415358	-0.143014	-0.267349	-0.094401	0.276518

In [ ]:
```
fig,ax=plt.subplots(figsize=(10,9))
sns.heatmap(df_numerical.corr(),annot=True)
```

Out[ ]:
```
<Axes: >
```



From the PCC table and the heatmap, we can observe that our target - life ladder is highly positively correlated with Country encoded, healthy life expectancy at birth, social support, log GDP per capita.

Life Ladder is poorly and negatively correlated with negative effect.

It has a moderate correlation with positive effect (positively), fredom to make life choices (positively), and perceptions of corruption (negatively).

Hence, we have good predictors and we do not need to perform any further feature engineering.

In [ ]:
```
df_label = df["Life Ladder"]
df_attributes = df.drop(columns = ["Life Ladder"])
```

In [ ]:
```
df_label
```

Out[ ]:
```
0       3.724
1       4.402
2       4.758
3       3.832
4       3.783
        ...
1944    3.735
1945    3.638
1946    3.616
1947    2.694
1948    3.160
Name: Life Ladder, Length: 1708, dtype: float64
```

In [ ]:
```
df_attributes
```

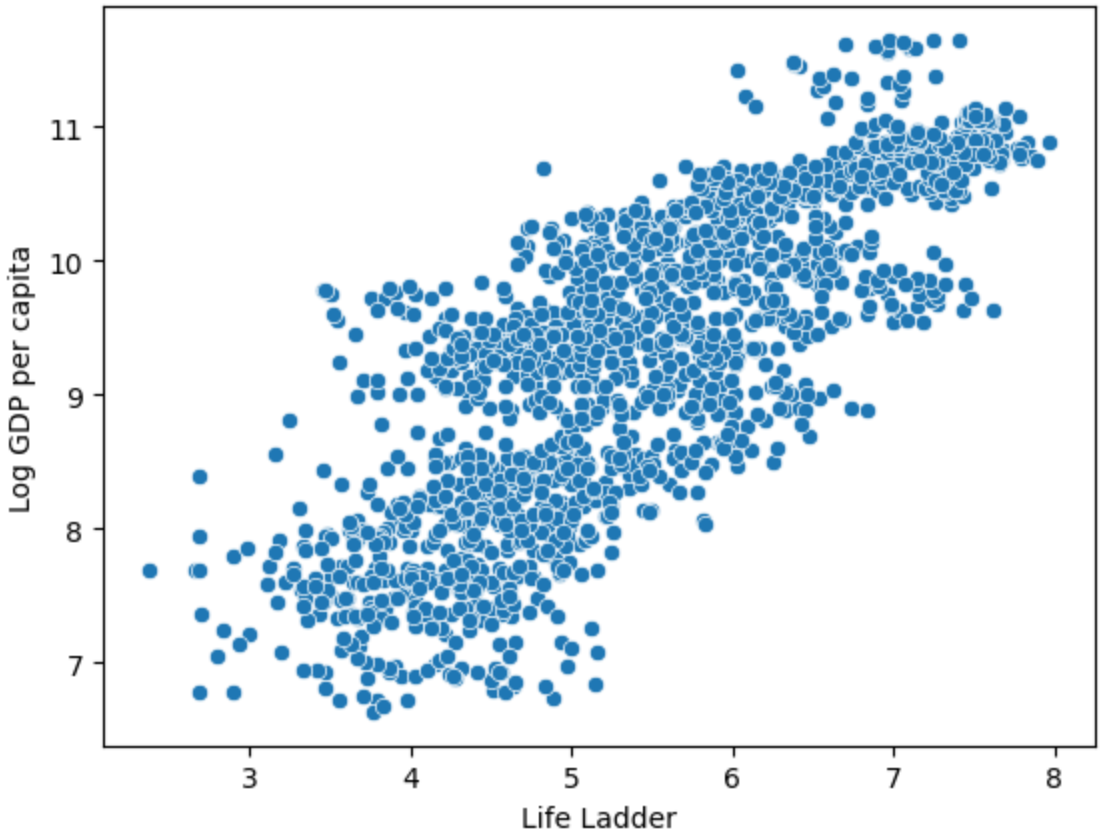
Out[ ]:

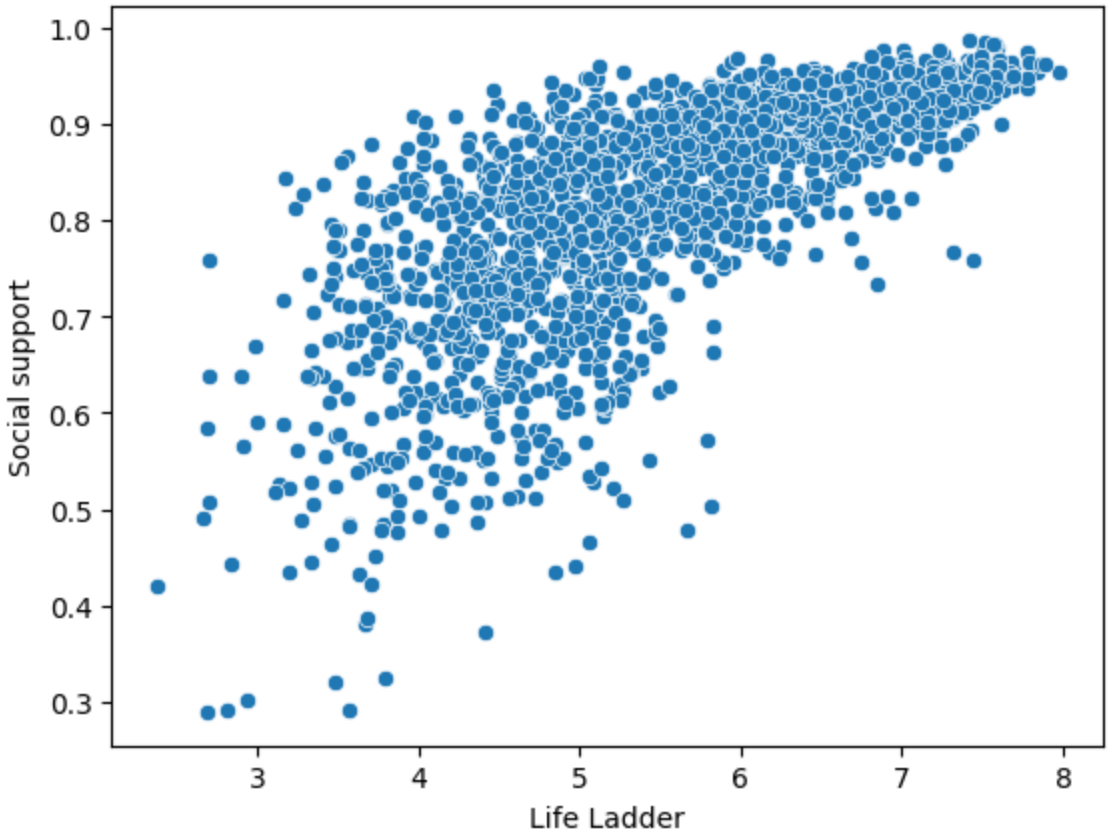
	Country name	Log GDP per capita	Social support	Healthy life expectancy at birth	Freedom to make life choices	Generosity	Perceptions of corruption	Positive affect	Ne
0	Afghanistan	7.370	0.451	50.80	0.718	0.168	0.882	0.518	
1	Afghanistan	7.540	0.552	51.20	0.679	0.190	0.850	0.584	
2	Afghanistan	7.647	0.539	51.60	0.600	0.121	0.707	0.618	
3	Afghanistan	7.620	0.521	51.92	0.496	0.162	0.731	0.611	
4	Afghanistan	7.705	0.521	52.24	0.531	0.236	0.776	0.710	
...	...	...	...	...	...	...	...	...	
1944	Zimbabwe	7.984	0.768	54.40	0.733	-0.095	0.724	0.738	
1945	Zimbabwe	8.016	0.754	55.00	0.753	-0.098	0.751	0.806	
1946	Zimbabwe	8.049	0.775	55.60	0.763	-0.068	0.844	0.710	
1947	Zimbabwe	7.950	0.759	56.20	0.632	-0.064	0.831	0.716	
1948	Zimbabwe	7.829	0.717	56.80	0.643	-0.009	0.789	0.703	

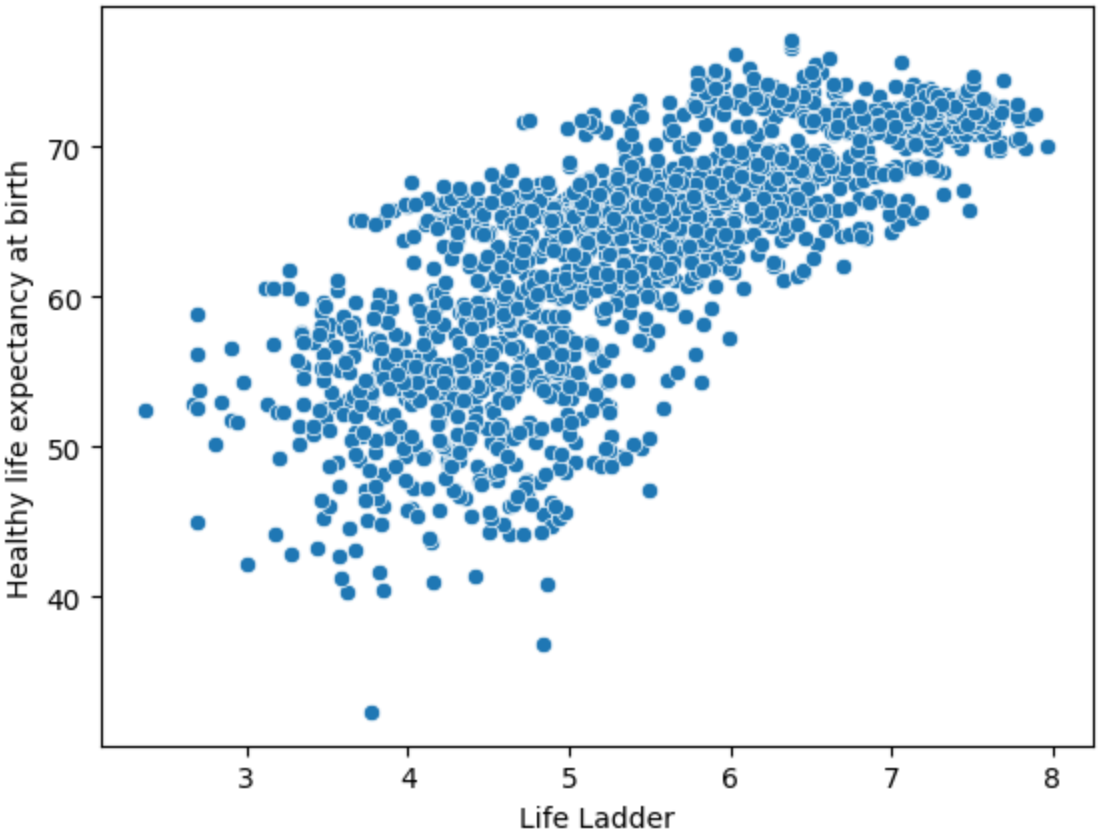
1708 rows × 10 columns

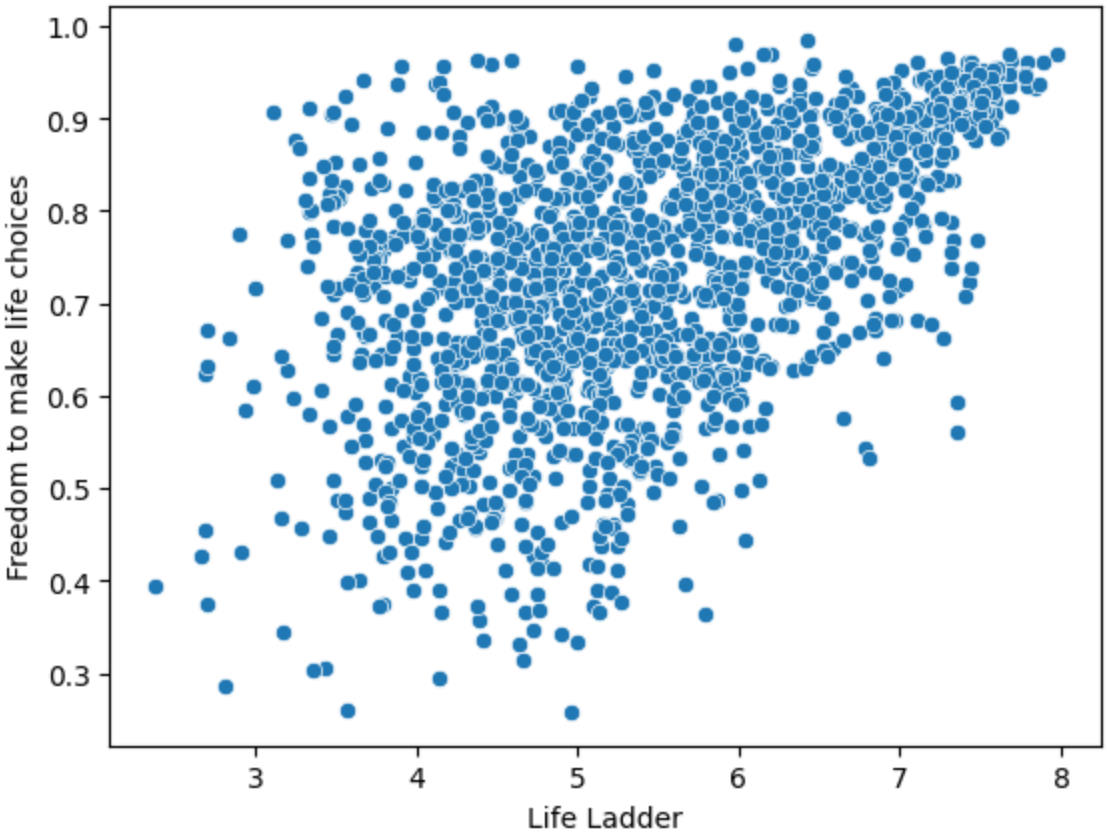
In [ ]:
```
# sns.pairplot(df_attributes)
```

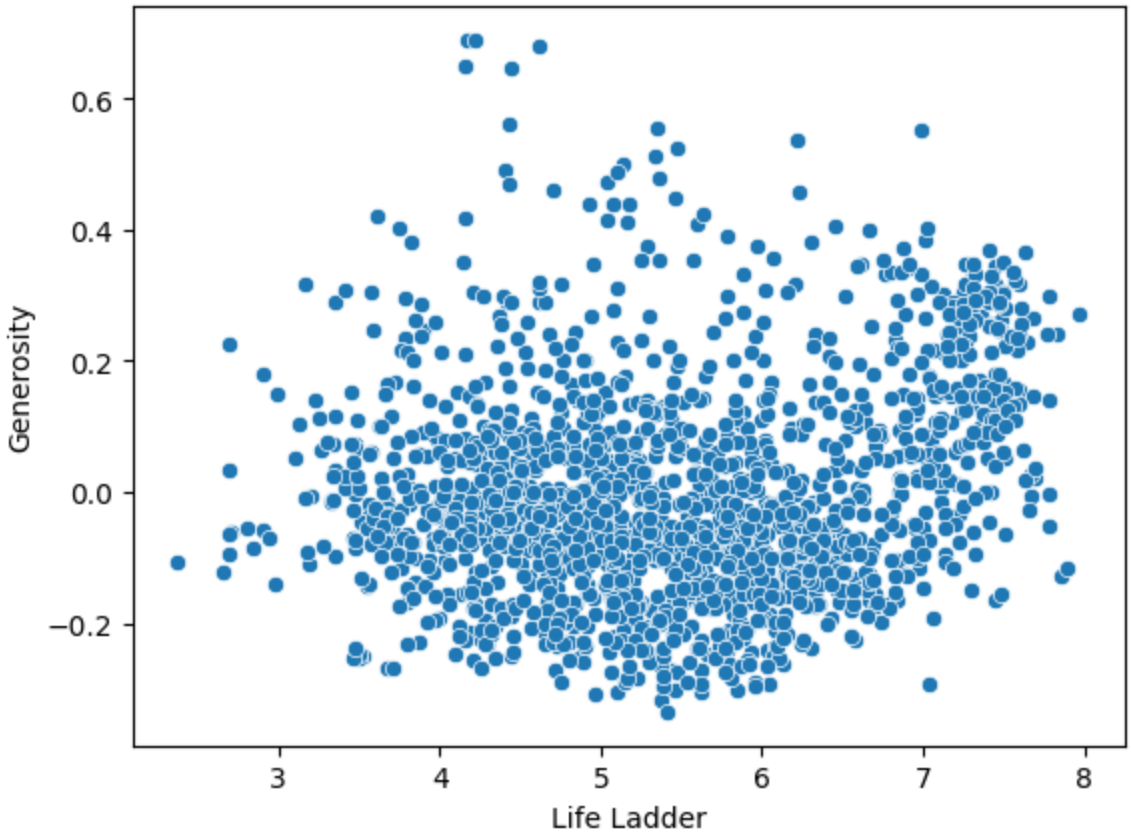
In [ ]:
```
df_wo_countryname = df.drop(['Country name'], axis=1)
for colname in df_attributes:
  sns.scatterplot(data=df_wo_countryname, x="Life Ladder", y=colname)
  plt.show()
```

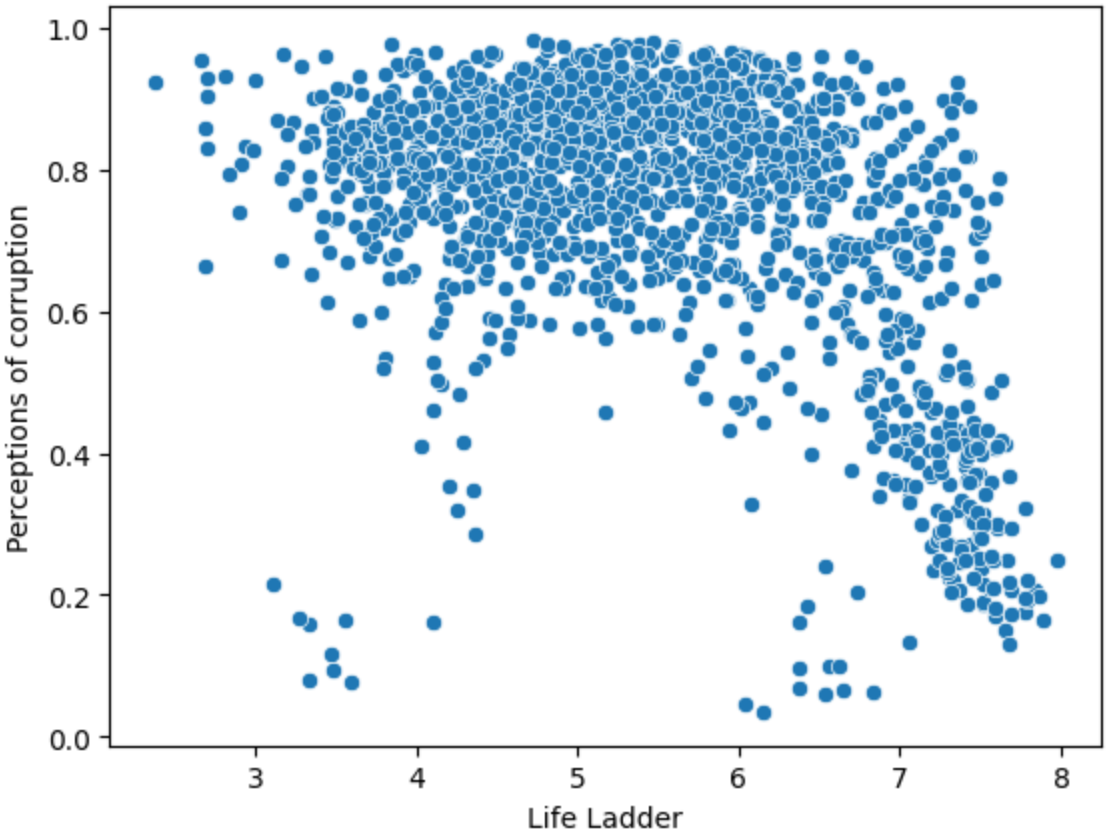

Log GDP per capita
11
10
9
8
7
3
4
5
6
7
8
Life Ladder

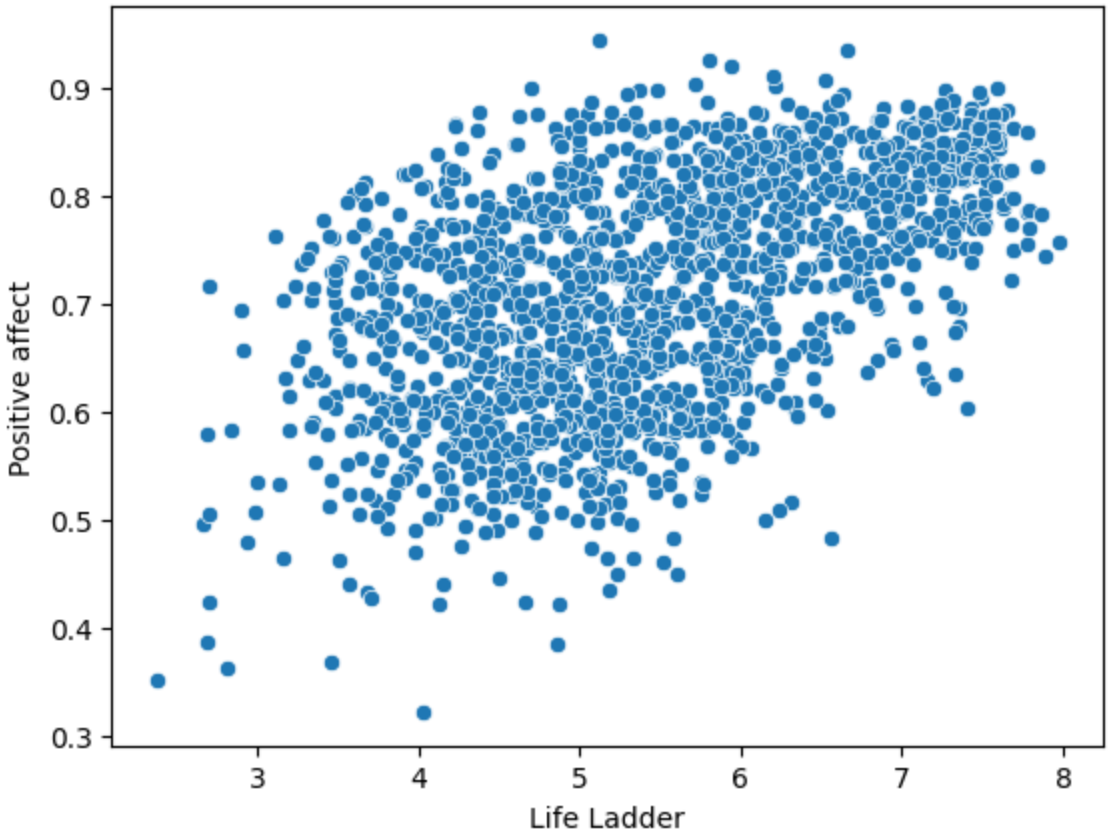

Social support
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
3
4
5
6
7
8
Life Ladder

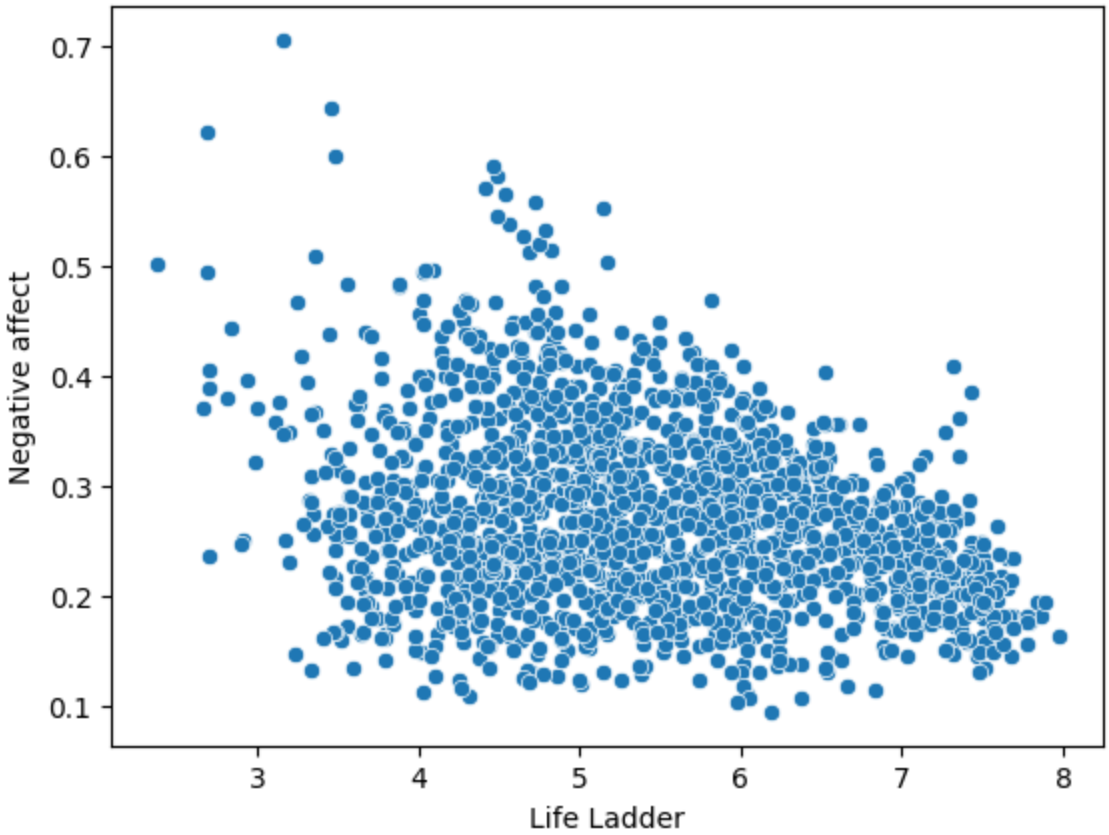

Healthy life expectancy at birth
70
60
50
40
3
4
5
6
7
8
Life Ladder

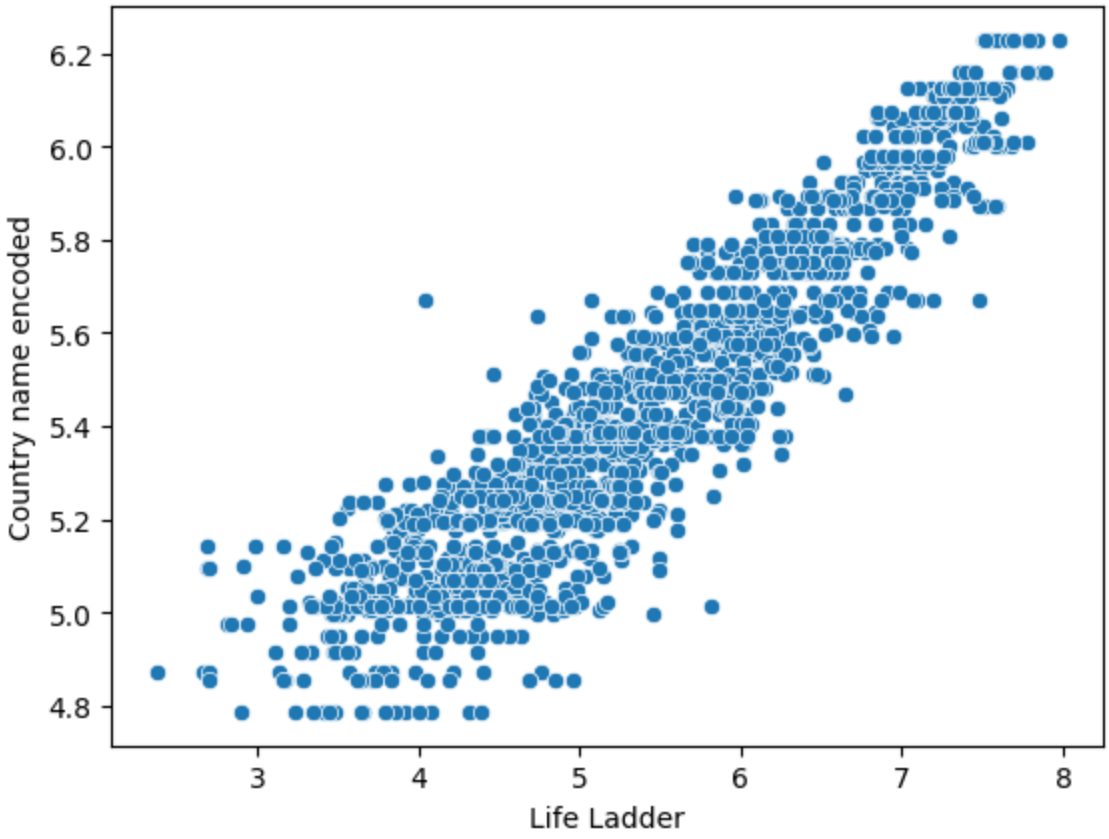

Freedom to make life choices
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
3
4
5
6
7
8
Life Ladder


Generosity
0.6
0.4
0.2
0.0
−0.2
3
4
5
6
7
8
Life Ladder


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


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


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



Looking at the scatter plots of attributes against the target (life ladder) we can further confirm about their correlations and predictive power.

The linearity is evident.

Part 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.

In [ ]:
```
df_attributes.drop(columns = ["Country name"], inplace = True)
```

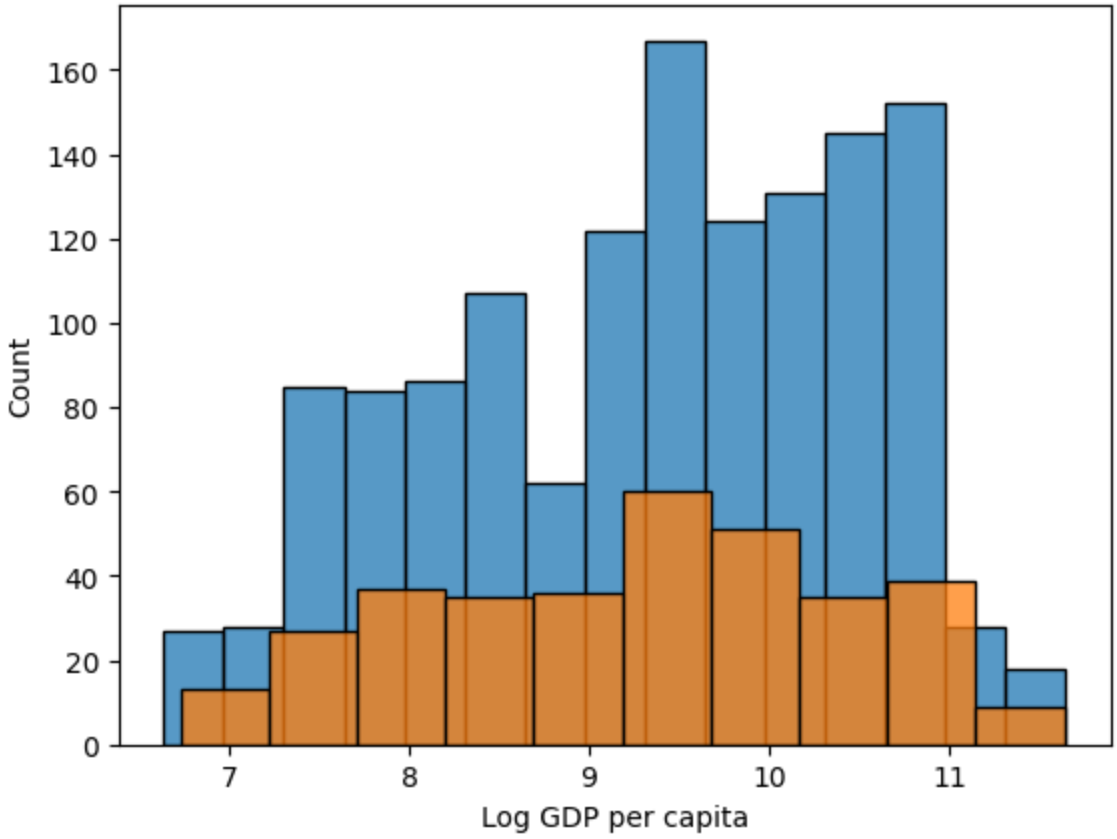
In [ ]:
```
from sklearn.model_selection import train_test_split
```

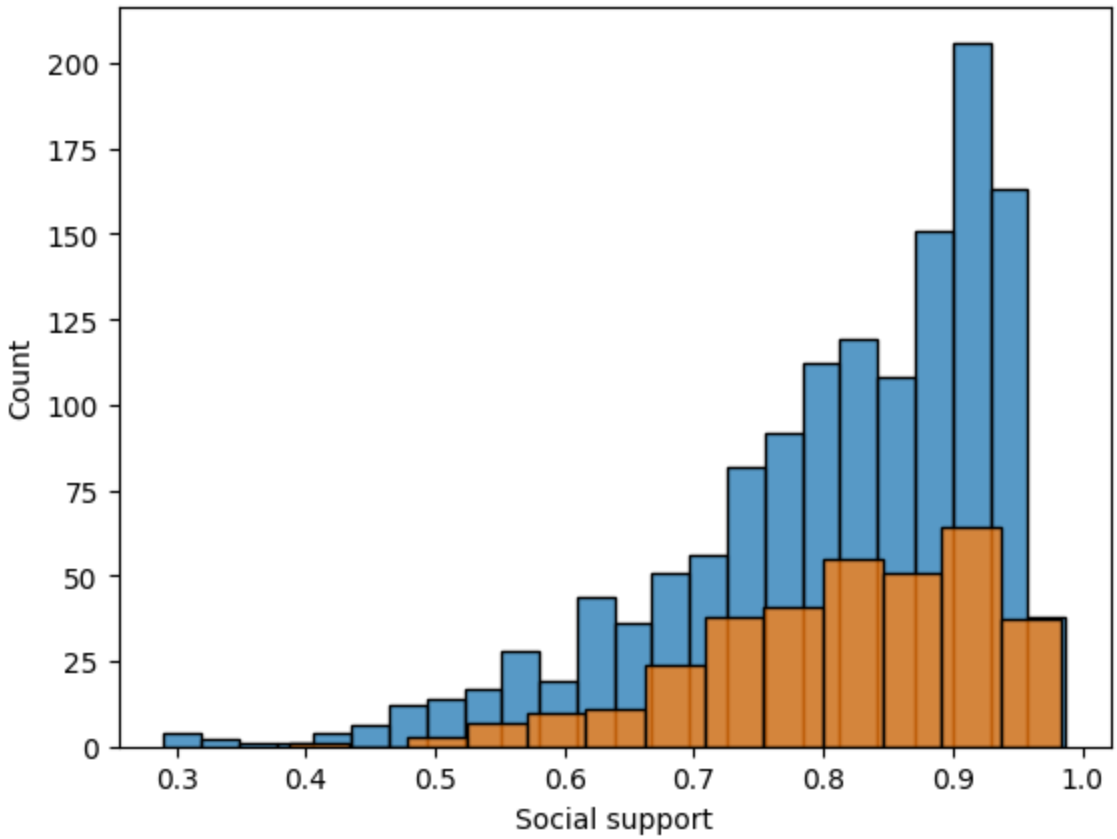
In [ ]:
```
## We use train_test_split with parameter shuffle set to True and test_size as
X_train, X_test, y_train, y_test = train_test_split(df_attributes, df_label, te
```

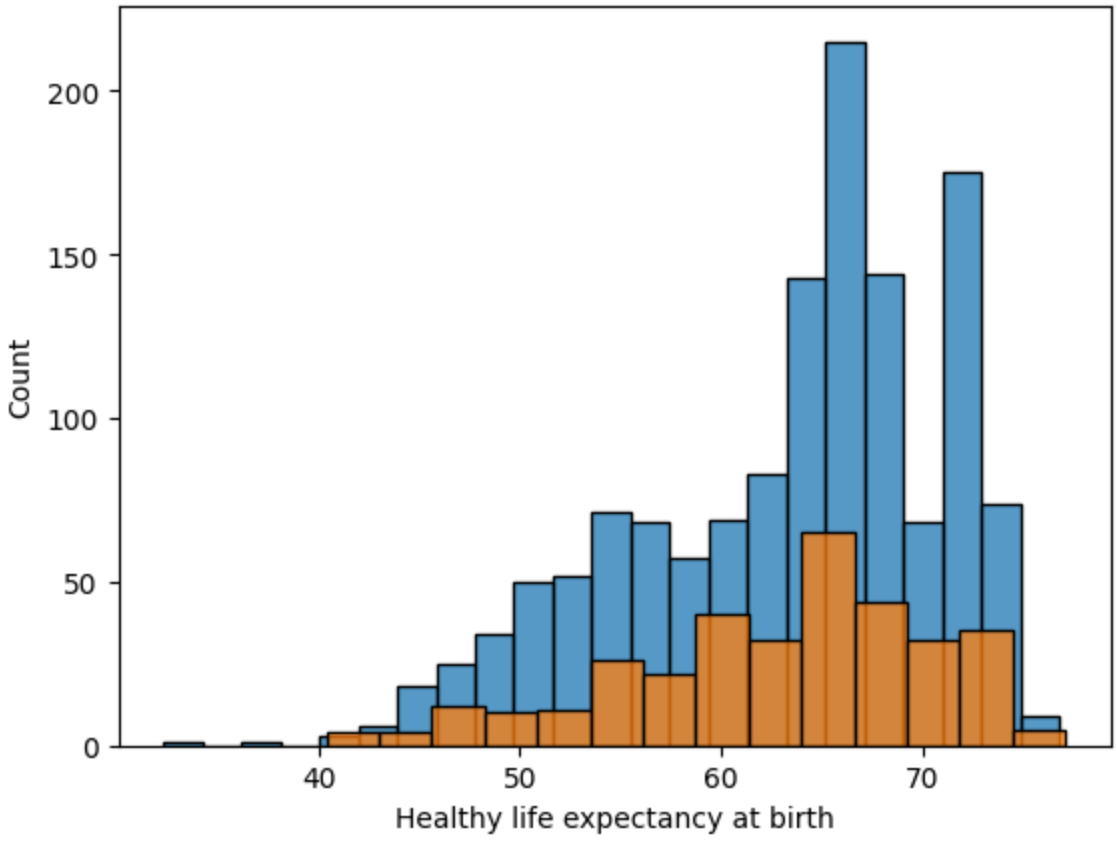
In [ ]:
```
X_train_numerical = X_train.select_dtypes(include=np.number)
X_test_numerical = X_test.select_dtypes(include=np.number)
```

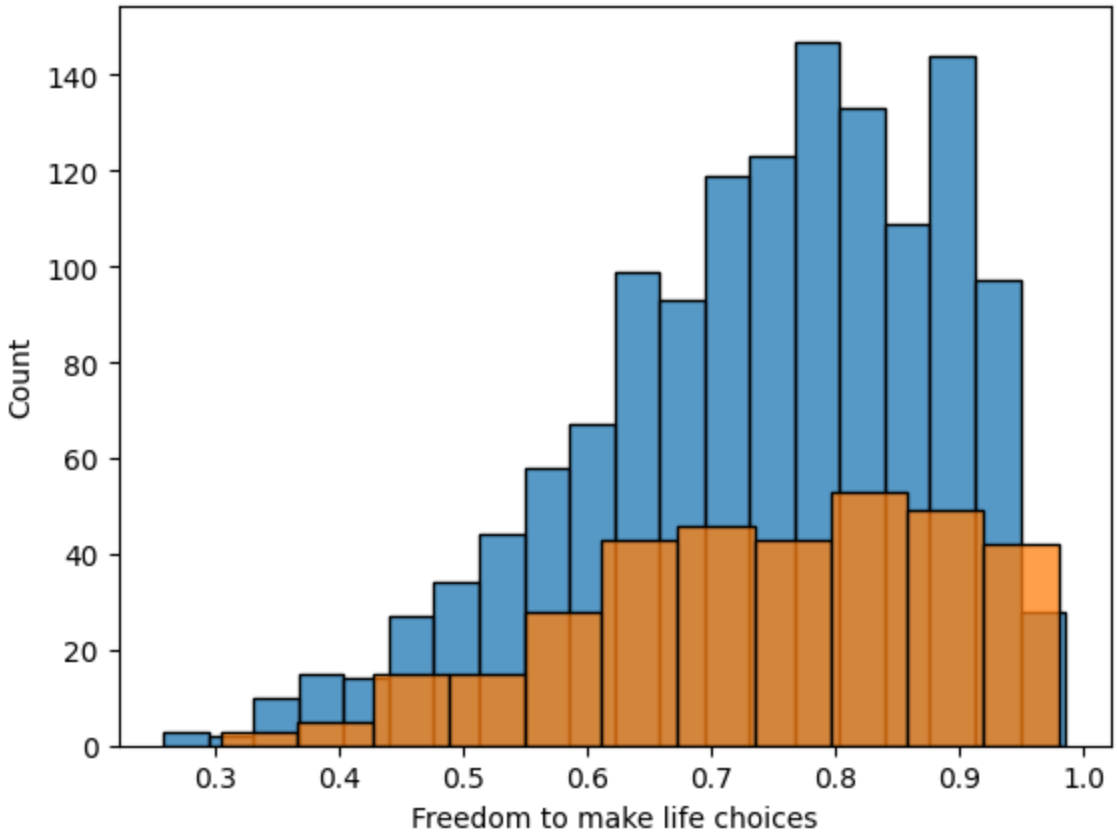
In [ ]:
```
for colname in df_attributes.columns:
  plt.subplot(111)
  sns.histplot(data=X_train_numerical,x=colname)

  plt.subplot(111)
  sns.histplot(data=X_test_numerical,x=colname)
```

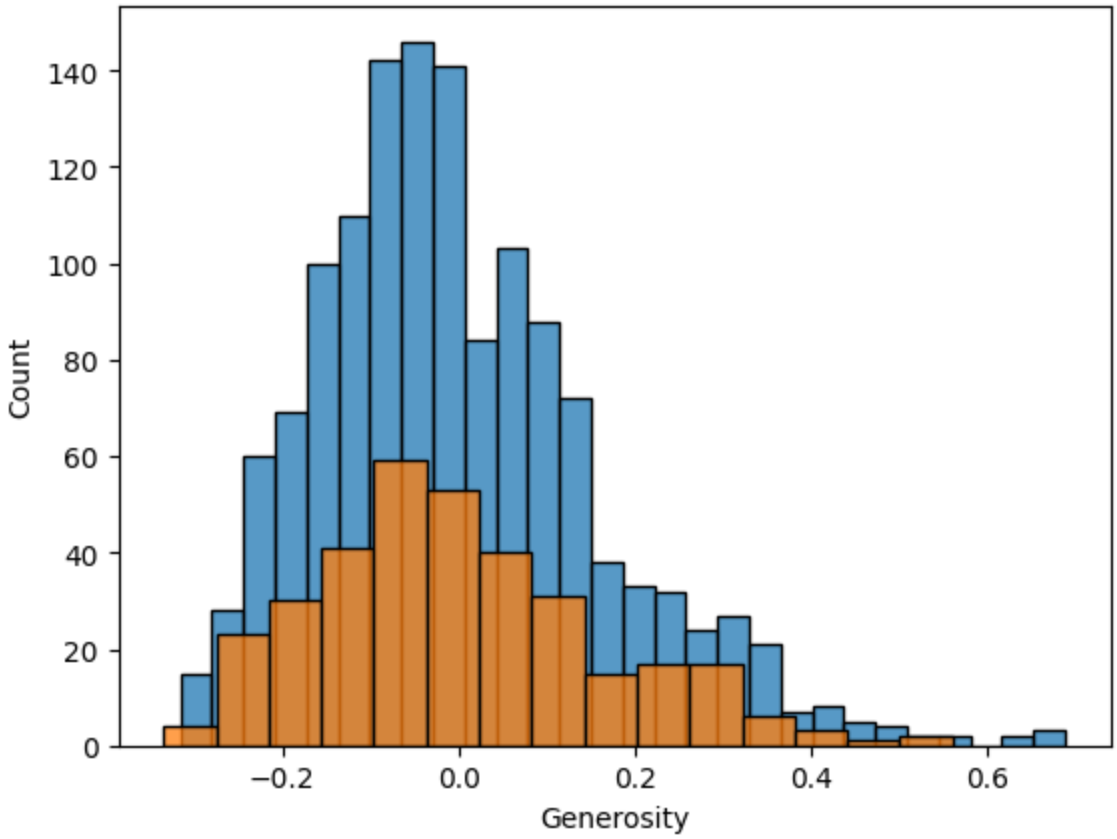
```
plt.show()
```

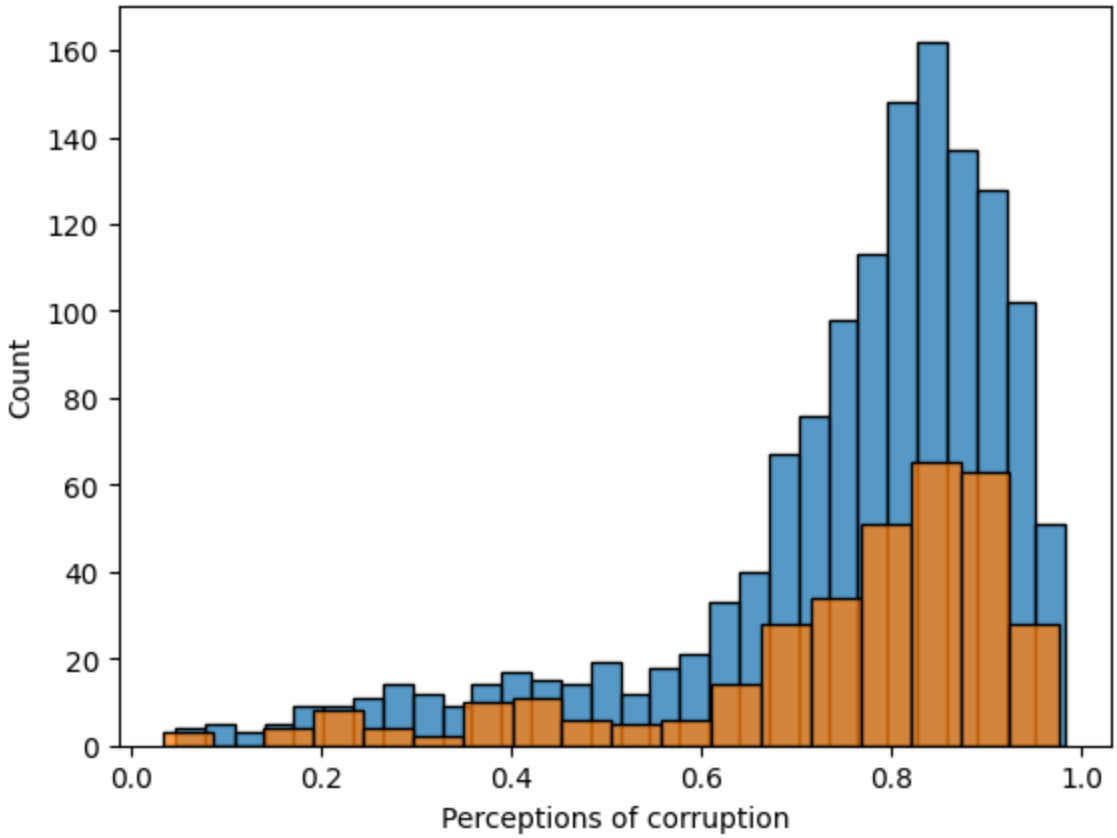

Count
Log GDP per capita
0
20
40
60
80
100
120
140
160
7
8
9
10
11

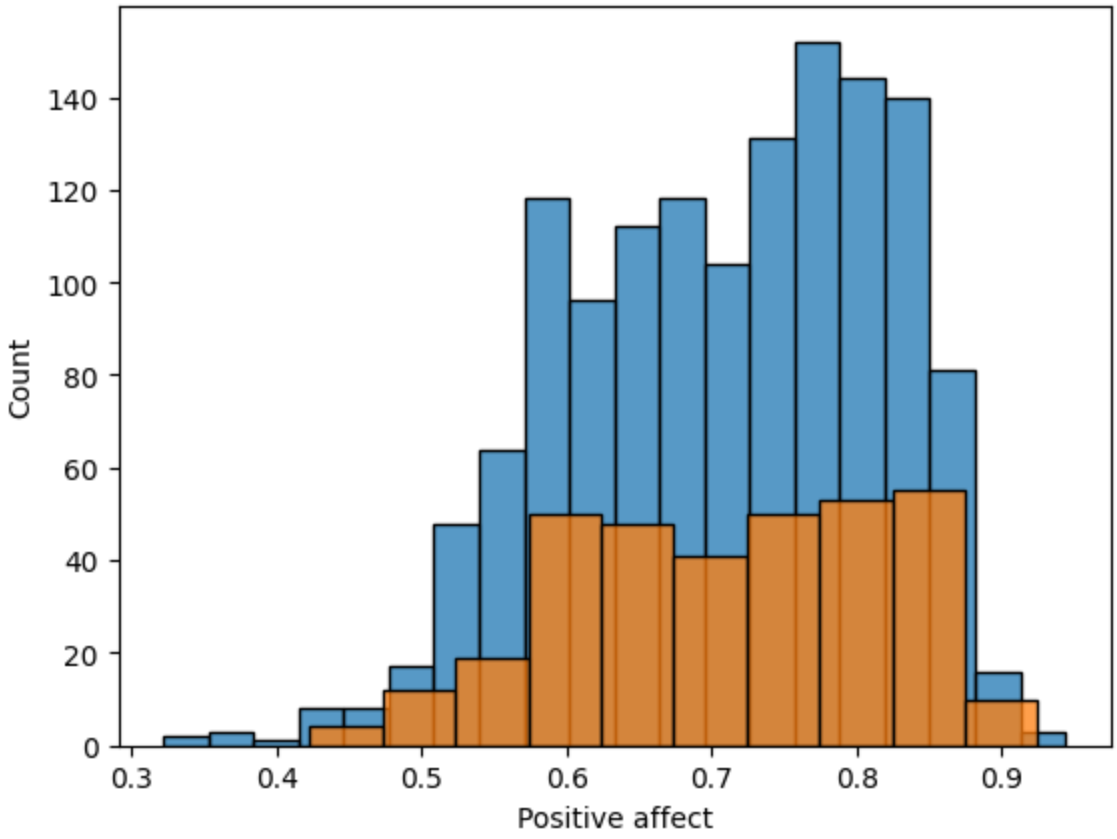

Count
Social support
0
25
50
75
100
125
150
175
200
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

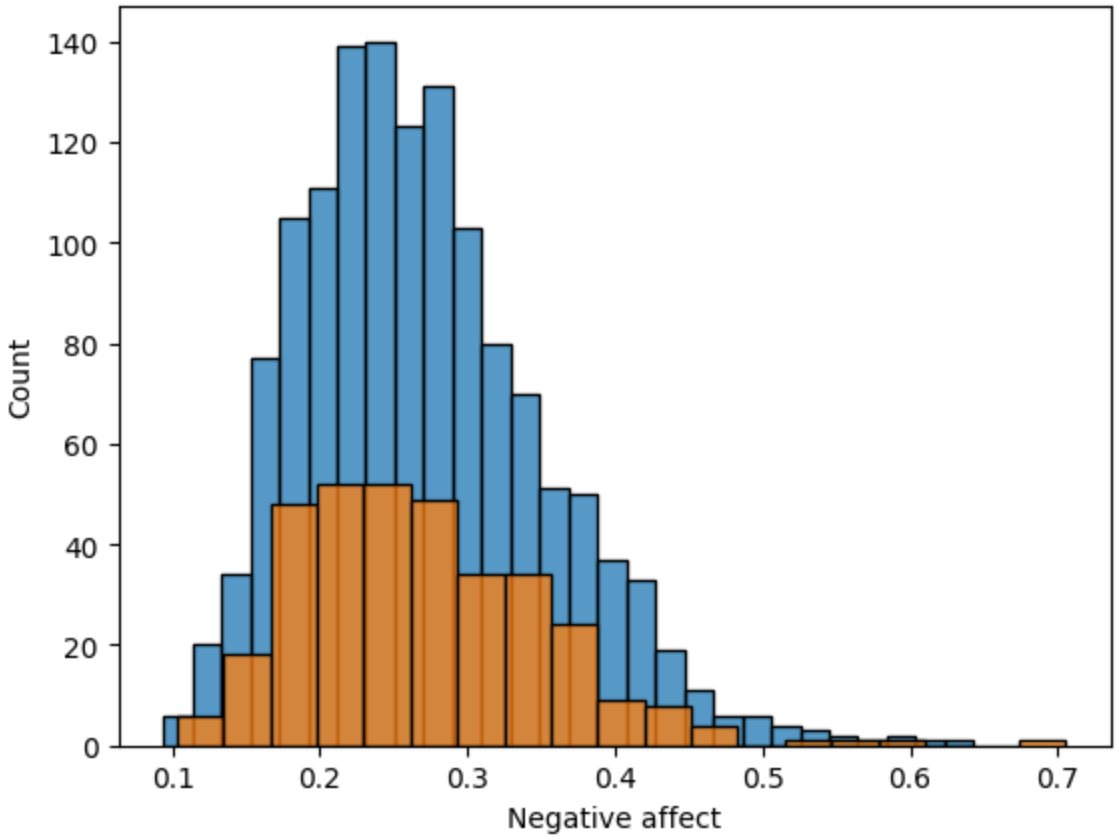

Count
200
150
100
50
0
40
50
60
70
Healthy life expectancy at birth

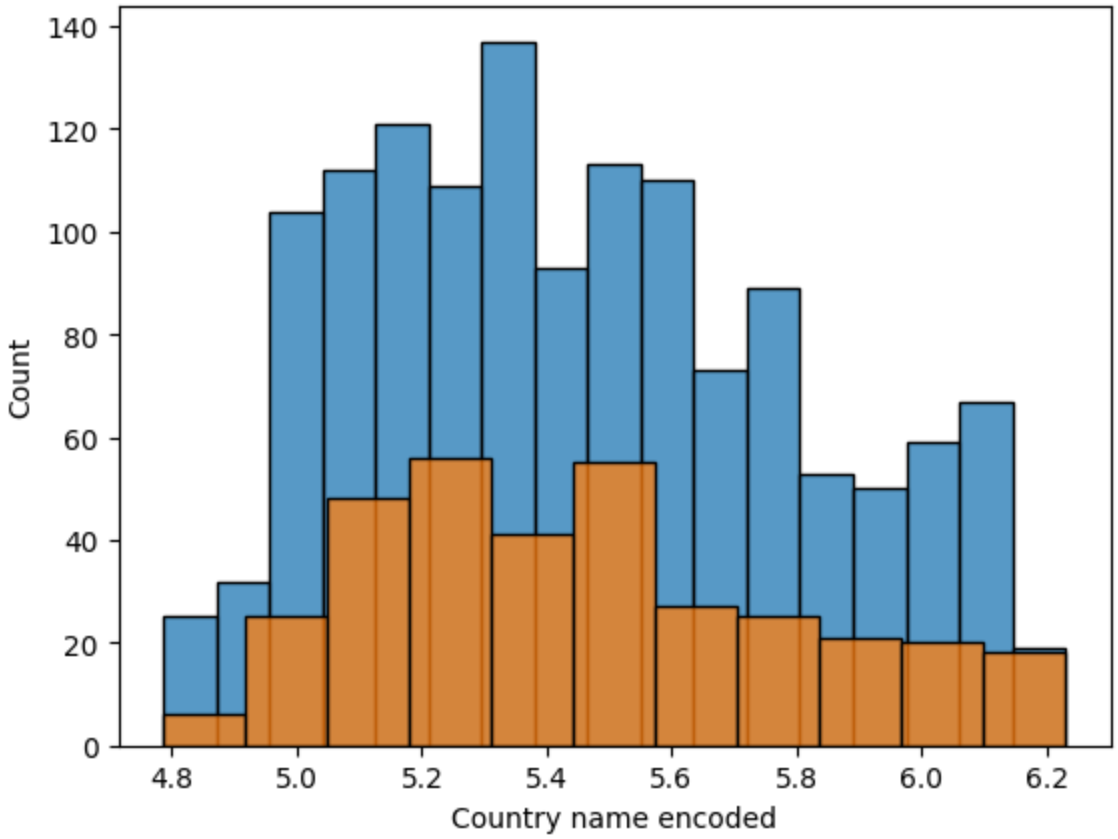

Count
140
120
100
80
60
40
20
0
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Freedom to make life choices


Count
Generosity
0
20
40
60
80
100
120
140
−0.2
0.0
0.2
0.4
0.6


Count
Perceptions of corruption
0
20
40
60
80
100
120
140
160
0.0
0.2
0.4
0.6
0.8
1.0


Count
Positive affect
0
20
40
60
80
100
120
140
0.3
0.4
0.5
0.6
0.7
0.8
0.9


Count
Negative affect
0
20
40
60
80
100
120
140
0.1
0.2
0.3
0.4
0.5
0.6
0.7



As we can see, the distribution (via histograms) for our test set and train set follow similar patterns and hence we can verify that we have randomly picked a representative train and test set

Let us also normalize our data before feeding it into the model.

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = pd.DataFrame(X_train)
X_test = pd.DataFrame(X_test)
```

Part 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.

In [ ]:
```
X_train
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Out[ ]:

	0	1	2	3	4	5	6	7	
0	0.722920	0.781923	0.887640	0.376891	0.043781	0.830128	0.672026	0.333333	0.47
1	0.528227	0.459110	0.784270	0.551582	0.106468	0.899573	0.348875	0.617486	0.31
2	0.857770	0.958393	0.901124	0.931224	0.332338	0.111111	0.802251	0.204007	1.00
3	0.248354	0.492109	0.503371	0.671252	0.257711	0.504274	0.413183	0.158470	0.38
4	0.646918	0.618364	0.739326	0.577717	0.387065	0.963675	0.326367	0.449909	0.34
...	...	...	...	...	...	...	...	...	
1361	0.073210	0.582496	0.419326	0.779917	0.237811	0.709402	0.527331	0.207650	0.16
1362	0.700977	0.890961	0.701124	0.627235	0.134328	0.938034	0.504823	0.087432	0.47
1363	0.625773	0.599713	0.744719	0.546080	0.310448	0.933761	0.284566	0.573770	0.34
1364	0.834829	0.922525	0.897079	0.944979	0.515423	0.216880	0.826367	0.207650	0.92
1365	0.052464	0.690100	0.367191	0.855571	0.305473	0.465812	0.655949	0.038251	0.16

1366 rows × 9 columns

In [ ]:
```
y_train
```

Out[ ]:
```
644     5.623
1751    4.315
461     7.649
467     4.369
972     5.032
        ...
1260    3.716
1450    5.855
975     4.983
1645    7.239
1256    4.267
Name: Life Ladder, Length: 1366, dtype: float64
```

In [ ]:
```
#Linear Regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

kf = KFold(n_splits=4)
lin_reg = LinearRegression()
# Create a list to store the performance metrics (e.g., Mean Squared Error) fo
```

```
performance_metrics = []

for train_index, test_index in kf.split(X_train):
  X_train_kFold, X_test_kFold = X_train.iloc[train_index,:], X_train.iloc[test_
  y_train_kFold, y_test_kFold = y_train.iloc[train_index], y_train.iloc[test_in


  # Fit the linear regression model on the training data
  lin_reg.fit(X_train_kFold, y_train_kFold)

  # Make predictions on the test data
  y_pred_kFold = lin_reg.predict(X_test_kFold)

  # Calculate the Mean Squared Error for this iteration
  mse = mean_squared_error(y_test_kFold, y_pred_kFold)

  # Store the performance metric
  performance_metrics.append(mse)

# Calculate the average performance metric across all iterations
average_performance = np.mean(performance_metrics)
```

In [ ]:
```
lin_reg.predict(X_test)
```

Out[ ]:
```
array([6.93034052, 4.41454288, 3.95719212, 4.70371092, 5.83943597,
       5.96298206, 5.74150359, 4.24292889, 4.92486858, 4.11869586,
       7.73646312, 5.82985437, 5.27509584, 4.47967613, 4.55746704,
       6.18139884, 3.98298804, 4.36986818, 4.27475062, 5.07769202,
       5.21756222, 6.14085159, 5.37416422, 5.13716997, 6.05076571,
       4.41833471, 4.21590871, 5.69706128, 4.61557747, 5.41667736,
       4.08623727, 4.96229469, 6.45875521, 7.13450609, 5.80324821,
       5.51211973, 5.57031038, 5.48729866, 6.14167803, 3.89636378,
       7.76959695, 3.58835219, 4.45646147, 5.39350571, 7.5338581 ,
       4.57704553, 4.66286408, 4.73073812, 4.24069807, 7.03692365,
       4.07276094, 3.62914398, 5.84255744, 4.24710407, 4.67611774,
       6.96027622, 3.70603034, 7.49678216, 4.91082621, 5.02264466,
       5.8956455 , 7.13615922, 6.57059862, 5.46206356, 5.58167065,
       5.6558349 , 6.14597545, 5.7760002 , 4.3319221 , 4.28294621,
       7.27824274, 5.39804637, 7.11623805, 7.34220736, 6.9534849 ,
       7.29888917, 6.72083276, 5.59763188, 4.81248238, 4.64761718,
       6.06951579, 4.47577671, 6.54857198, 7.35257   , 4.55563708,
       4.08452167, 7.72719092, 4.53961028, 4.72088987, 5.72870855,
       4.35487974, 4.005095  , 5.09402124, 4.19803382, 4.03337046,
       3.75821123, 5.61419025, 4.61700481, 4.74366894, 7.52811079,
       7.06107984, 5.9309589 , 7.78845549, 4.38102105, 4.59507001,
       4.92663429, 6.29014898, 4.99634252, 7.23922373, 4.87153078,
       6.1579765 , 3.95571992, 7.0521888 , 6.20435568, 4.32810836,
       4.8667632 , 3.76314905, 4.71444364, 4.81483976, 7.47582446,
       5.5579489 , 6.36000039, 4.72661001, 6.44769778, 5.22134096,
       4.63576499, 6.45909104, 5.93644352, 7.49455537, 4.38141172,
       4.91211106, 5.62731985, 5.43690282, 4.39929427, 5.41141633,
       6.41564235, 4.69112652, 6.39655414, 6.99235599, 5.87961946,
       7.10779431, 5.04678453, 4.41004791, 4.11744639, 5.3644257 ,
       5.47906148, 7.56234019, 6.16484186, 5.58448495, 7.27203621,
       4.72872583, 6.07846128, 5.366423  , 6.20123281, 6.56300463,
       3.60607583, 5.29814033, 4.35398219, 6.77443873, 4.36439446,
       5.14868969, 4.92742622, 7.45171154, 6.23070613, 4.1116089 ,
       4.3305339 , 5.04303308, 6.0255695 , 5.50735591, 7.75276178,
       7.31421837, 6.77935161, 4.02927773, 5.68994034, 5.37070075,
       4.35755841, 5.45621646, 7.27137742, 5.69863812, 7.44481782,
       4.45171574, 6.7275226 , 5.39587083, 5.78054169, 4.52393536,
       5.95315281, 6.6844999 , 5.31484502, 4.70760293, 4.46203031,
       5.92858698, 5.61993372, 5.21261885, 4.95231888, 7.31206451,
       4.48494047, 7.5356344 , 3.98250612, 6.39511083, 5.537332  ,
       6.48166754, 6.63271601, 6.2290538 , 5.10646919, 4.29542152,
       3.72106407, 4.29750836, 7.33139229, 4.8674667 , 5.89616787,
       5.70726674, 5.6345627 , 6.65584534, 4.54063735, 5.49850236,
       4.99372456, 5.56254195, 4.7156402 , 6.30483619, 4.73791697,
       7.16255101, 4.94801376, 4.36675483, 6.65717224, 4.43452238,
       6.29200082, 4.66192254, 4.5326661 , 4.38581952, 6.17476157,
       4.69422346, 6.29688175, 7.27262232, 7.26605797, 4.14081495,
       4.6806593 , 7.34184235, 5.69332917, 4.64326728, 6.96932474,
       5.3428835 , 5.32159248, 6.99201885, 5.60135923, 4.64702189,
       6.07150858, 6.86017505, 6.03858762, 5.885618  , 5.63927459,
       5.34745792, 5.02262483, 4.53803228, 5.54669121, 4.04089857,
       4.45162031, 4.99245454, 7.29170803, 3.96082036, 4.10506299,
       7.49204876, 6.02646684, 5.39008352, 4.70292124, 5.35964951,
       4.46265988, 4.71754993, 5.58124029, 5.70274839, 3.82014438,
       5.50461302, 5.72213223, 4.21977844, 7.47110015, 3.73994378,
       4.90164432, 5.0460675 , 5.64116992, 6.26926341, 4.35827664,
       4.60191606, 5.6524589 , 4.88458668, 5.41458572, 5.03566793,
       6.26938306, 5.61343942, 5.79834461, 5.88469682, 5.27607064,
       5.68931687, 4.76940539, 5.58157804, 4.86561663, 5.01053154,
       4.3823221 , 4.33738015, 7.03563588, 4.48677565, 4.88219823,
```

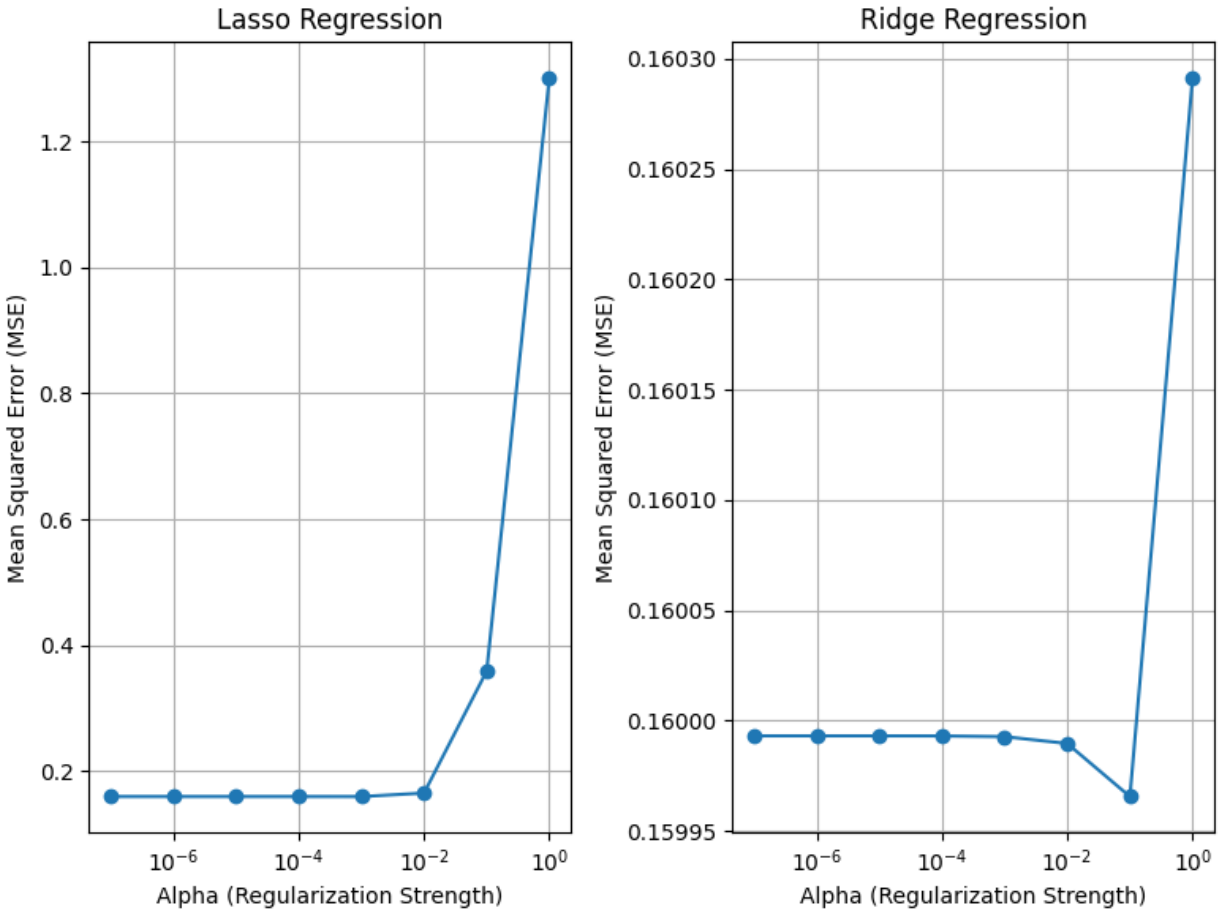
```
       4.501142  , 5.58591296, 6.32508383, 4.15890131, 5.44106641,
       6.96634429, 4.60543728, 4.41229849, 5.83646822, 4.43883194,
       5.51508547, 7.75578008, 5.46401959, 7.12487628, 5.39736699,
       3.93322234, 4.61690244, 4.26721223, 3.84235448, 5.77322991,
       4.32416078, 3.88040757, 5.94642365, 6.45320946, 5.00151752,
       7.53347151, 6.67925875, 5.03487863, 5.02604033, 4.40882521,
       6.37911238, 4.50993249, 4.8227262 , 6.78679572, 4.48654384,
       5.2233609 , 4.91790728, 5.79640973, 5.61416997, 5.56564841,
       6.50565212, 4.07376739])
```

In [ ]:

```
from sklearn.linear_model import Lasso, Ridge
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt

warnings.filterwarnings("ignore")

# Define a list of alpha values to try
alphas = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1]

# Initialize empty lists to store the MSE values
lasso_mse_values = []
ridge_mse_values = []

for alpha in alphas:
    lasso_mse_fold = []
    ridge_mse_fold = []

    for train_index, test_index in kf.split(X_train):
        X_train_fold, X_test_fold = X_train.iloc[train_index,:], X_train.iloc[t
        y_train_fold, y_test_fold = y_train.iloc[train_index], y_train.iloc[tes

        # Create Lasso and Ridge models with the current alpha
        lasso_reg = Lasso(alpha=alpha)
        ridge_reg = Ridge(alpha=alpha)

        # Fit the Lasso model
        lasso_reg.fit(X_train_fold, y_train_fold)

        # Make predictions with the trained Lasso model
        lasso_predictions = lasso_reg.predict(X_test_fold)

        # Calculate MSE for Lasso
        lasso_mse = mean_squared_error(y_test_fold, lasso_predictions)
        lasso_mse_fold.append(lasso_mse)

        # Fit the Ridge model
        ridge_reg.fit(X_train_fold, y_train_fold)

        # Make predictions with the trained Ridge model
        ridge_predictions = ridge_reg.predict(X_test_fold)

        # Calculate MSE for Ridge
        ridge_mse = mean_squared_error(y_test_fold, ridge_predictions)
        ridge_mse_fold.append(ridge_mse)

    # Calculate the mean MSE across the four folds for each alpha
    lasso_mse_values.append(np.mean(lasso_mse_fold))
    ridge_mse_values.append(np.mean(ridge_mse_fold))

    # Print the MSE values for each alpha
```

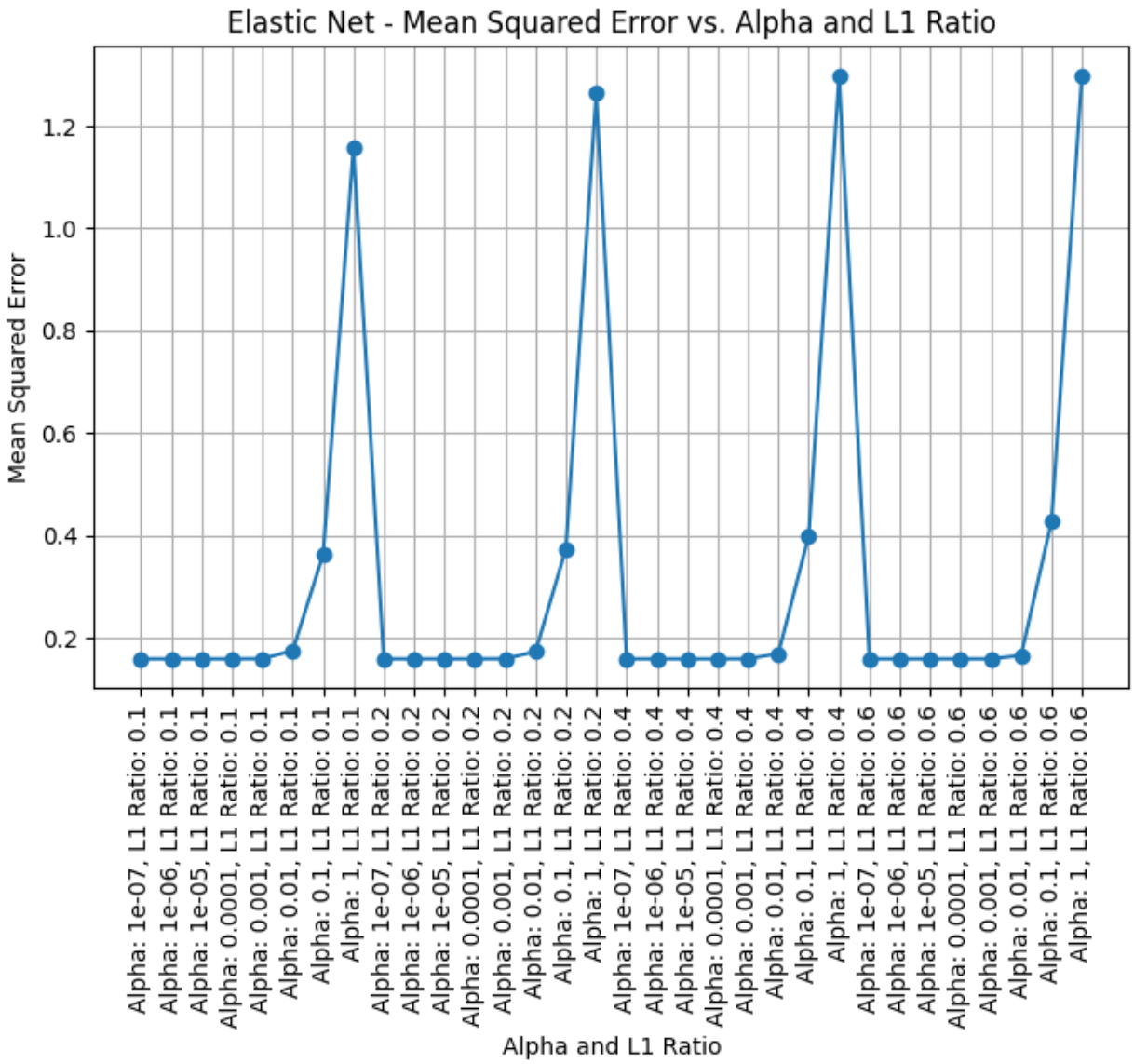
```
    print(f"Alpha = {alpha}")
    print("Lasso Regression Mean CV MSE:", np.mean(lasso_mse_fold))
    print("Ridge Regression Mean CV MSE:", np.mean(ridge_mse_fold))
    print("===")

# Create individual plots for each regularization technique
plt.figure(figsize=(12, 6))

# Lasso Regression Plot
plt.subplot(131)
plt.semilogx(alphas, lasso_mse_values, marker='o')
plt.xlabel('Alpha (Regularization Strength)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Lasso Regression')
plt.grid(True)

# Ridge Regression Plot
plt.subplot(132)
plt.semilogx(alphas, ridge_mse_values, marker='o')
plt.xlabel('Alpha (Regularization Strength)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Ridge Regression')
plt.grid(True)

plt.tight_layout()
plt.show()

warnings.resetwarnings()
```

```
Alpha = 1e-07
Lasso Regression Mean CV MSE: 0.15999303425277747
Ridge Regression Mean CV MSE: 0.159993062427645
===
Alpha = 1e-06
Lasso Regression Mean CV MSE: 0.15999275997514273
Ridge Regression Mean CV MSE: 0.15999306210877534
===
Alpha = 1e-05
Lasso Regression Mean CV MSE: 0.15999315522016264
Ridge Regression Mean CV MSE: 0.15999305892015053
===
Alpha = 0.0001
Lasso Regression Mean CV MSE: 0.15998863271612268
Ridge Regression Mean CV MSE: 0.1599930270411281
===
Alpha = 0.001
Lasso Regression Mean CV MSE: 0.15994774144939425
Ridge Regression Mean CV MSE: 0.15999270897329
===
Alpha = 0.01
Lasso Regression Mean CV MSE: 0.1654925630243752
Ridge Regression Mean CV MSE: 0.15998960036525417
===
Alpha = 0.1
Lasso Regression Mean CV MSE: 0.35899065854785966
Ridge Regression Mean CV MSE: 0.15996555623363307
===
Alpha = 1
Lasso Regression Mean CV MSE: 1.300302382784274
Ridge Regression Mean CV MSE: 0.16029124133238964
===
```



In [ ]:
```
import warnings
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold

warnings.filterwarnings("ignore")

# Define the number of folds (k)
k = 4

# Create lists of alpha (regularization strength) and l1_ratio (L1 mixing parar
alpha_list = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
l1_ratio_list = [0.1, 0.2, 0.4, 0.6]  # Different L1 ratios for Elastic Net

alpha_l1_list = []  # Store alpha and l1_ratio combinations
elastic_mse_vals = []  # Store Elastic Net MSE values

for l1_ratio in l1_ratio_list:
    for alpha in alpha_list:
        elastic_net_model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter

        # Initialize KFold cross-validator
        kf = KFold(n_splits=k, shuffle=True, random_state=42)

        fold_training_losses = []  # Store training losses for each fold
        fold_validation_losses = []  # Store validation losses for each fold

        for train_index, test_index in kf.split(X_train):  # You should replace
```

```
            X_train_fold, X_test_fold = X_train.iloc[train_index,:], X_train.il
            y_train_fold, y_test_fold = y_train.iloc[train_index], y_train.iloc

            # Fit the Elastic Net model
            elastic_net_model.fit(X_train_fold, y_train_fold)

            # Predict on the training data
            y_train_pred = elastic_net_model.predict(X_train_fold)

            # Calculate training loss (Mean Squared Error) and append to the li
            train_loss = mean_squared_error(y_train_fold, y_train_pred)
            fold_training_losses.append(train_loss)

            # Predict on the validation data
            y_val_pred = elastic_net_model.predict(X_test_fold)

            # Calculate validation loss (Mean Squared Error) and append to the
            val_loss = mean_squared_error(y_test_fold, y_val_pred)
            fold_validation_losses.append(val_loss)

        # Calculate the mean training and validation loss across folds for this
        mean_training_loss = np.mean(fold_training_losses)
        mean_validation_loss = np.mean(fold_validation_losses)  # Fix this line

        alpha_l1_list.append((alpha, l1_ratio))
        elastic_mse_vals.append(mean_validation_loss)


# Plot training and validation loss as a function of alpha
plt.figure(figsize=(8, 5))
plt.plot(range(len(alpha_l1_list)), elastic_mse_vals, marker='o')
plt.xticks(range(len(alpha_l1_list)), [f"Alpha: {alpha}, L1 Ratio: {l1_ratio}"
plt.xlabel('Alpha and L1 Ratio')
plt.ylabel('Mean Squared Error')
plt.title('Elastic Net - Mean Squared Error vs. Alpha and L1 Ratio')
plt.grid()
plt.show()

warnings.resetwarnings()
```



In [ ]:
```
# As asked in the question, we would be running a four-fold valiadation, hence
k = 4

# Storing the theta values for each fold
theta_values = []

# Initialize KFold cross-validator
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Initialize an empty list to store the MSE values for each fold
mse_values = []

# Perform k-fold cross-validation
for train_index, test_index in kf.split(X_train):
    X_train_fold, X_val_fold = X_train.iloc[train_index,:], X_train.iloc[test_
    y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[test_in

    # Calculating the coefficients using the normal equation
    X_transpose = np.transpose(X_train_fold)
    X_transpose_X = np.dot(X_transpose, X_train_fold)
    X_transpose_X_inv = np.linalg.inv(X_transpose_X)
    X_transpose_y = np.dot(X_transpose, y_train_fold)
    theta = np.dot(X_transpose_X_inv, X_transpose_y)
```

```
    # Append the theta values for this fold to the list
    theta_values.append(theta)

    # Make predictions using the calculated theta values
    y_val_pred = np.dot(X_val_fold, theta)

    # Calculate and append the MSE for this fold
    mse_fold = mean_squared_error(y_val_fold, y_val_pred)
    mse_values.append(mse_fold)

# Calculate the average coefficients (theta) across all folds
average_theta = np.mean(theta_values, axis=0)

# Calculate the average MSE across all folds
average_mse = np.mean(mse_values)

print("Average Coefficients (theta) across all folds:", average_theta)
print("Average MSE across all folds:",average_mse)
```

```
Average Coefficients (theta) across all folds: [0.13760749 1.32009442 1.374610
47 0.9762109  0.5060611  0.91092974
 0.680486   0.55040688 2.82104474]
Average MSE across all folds: 0.24713120603072275
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
```

In [ ]:
```
## Predict / Plot
```

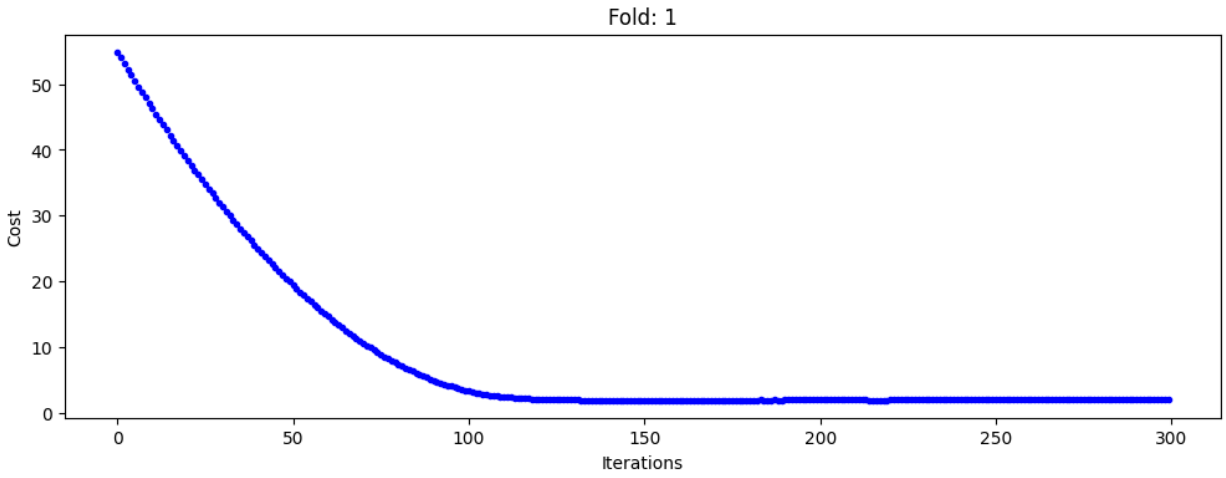
```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
```

In [ ]:
```
# Inside the minibatch_gradient_descent function
def minibatch_gradient_descent(X, y, theta, lr, n_iter, batch_size, gradient_cl
    cost_history = []
    for _ in range(n_iter):
        indices = np.random.choice(len(X), batch_size, replace=False)
        X_batch = X[indices]
        y_batch = y_batch = y[indices].reshape(-1, 1)  # Reshape y_batch to (ba
        gradient = (1 / batch_size) * X_batch.T.dot(X_batch.dot(theta) - y_batc

        # Clip gradients to prevent extreme updates
        gradient = np.clip(gradient, -gradient_clip_threshold, gradient_clip_th

        theta -= lr * gradient
        cost = np.mean((X.dot(theta) - y) ** 2)
        cost_history.append(cost)
    return theta, cost_history
```

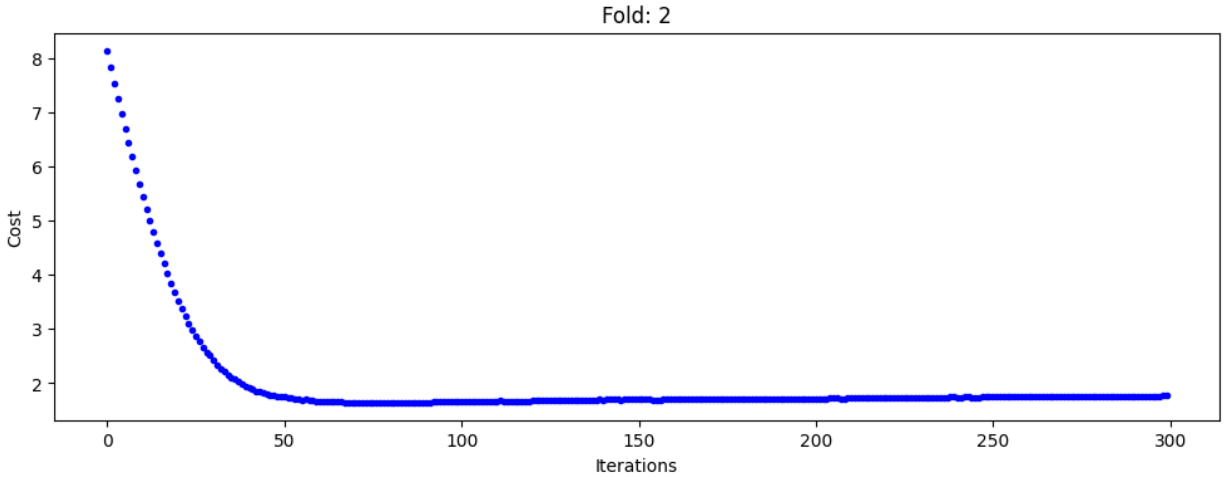
In [ ]:
```
X_train_array = X_train.values
y_train_array = y_train.values
```

In [ ]:
```
gradient_clip_threshold = 1.0  # Adjust this threshold as needed

batch_sizes = [5,10,20,50,100]
for batch_size in batch_sizes:
  print("*******************************************************")
  print(f"The following graphs are for batch size: {batch_size}")
  print("*******************************************************")
  i = 1
  for train_index, val_index in kf.split(X_train_array):
      X_train_fold, X_val_fold = X_train_array[train_index], X_train_array[val_
      y_train_fold, y_val_fold = y_train_array[train_index], y_train_array[val_

      # Add bias column to fold data
      X_train_fold = np.c_[np.ones((len(X_train_fold), 1)), X_train_fold]
      X_val_fold = np.c_[np.ones((len(X_val_fold), 1)), X_val_fold]

      lr = 0.01
      n_iter = 300
      batch_size = 20
      theta = np.random.randn(10, 1)

      # Train the model using minibatch gradient descent with gradient clipping
      theta, cost_history = minibatch_gradient_descent(X_train_fold, y_train_fo

      print('Predictions for fold:', i)
      print('Intercept value:', theta[0, 0], '\nWeight values:', theta[1:])

      # Use the validation data and mean square error to evaluate performance
      N = len(y_val_fold)
      y_hat = np.dot(X_val_fold, theta)
      mse_val = (1 / N) * (np.sum(np.square(y_hat - y_val_fold)))
      print('Mean Square Error:', round(mse_val, 3), '\n')

      fig, ax = plt.subplots(figsize=(12, 4))
      plt.title('Fold: ' + str(i))
      ax.set_ylabel('Cost')
      ax.set_xlabel('Iterations')
      _ = ax.plot(range(n_iter), cost_history, 'b.')
      plt.show()

      i += 1
```

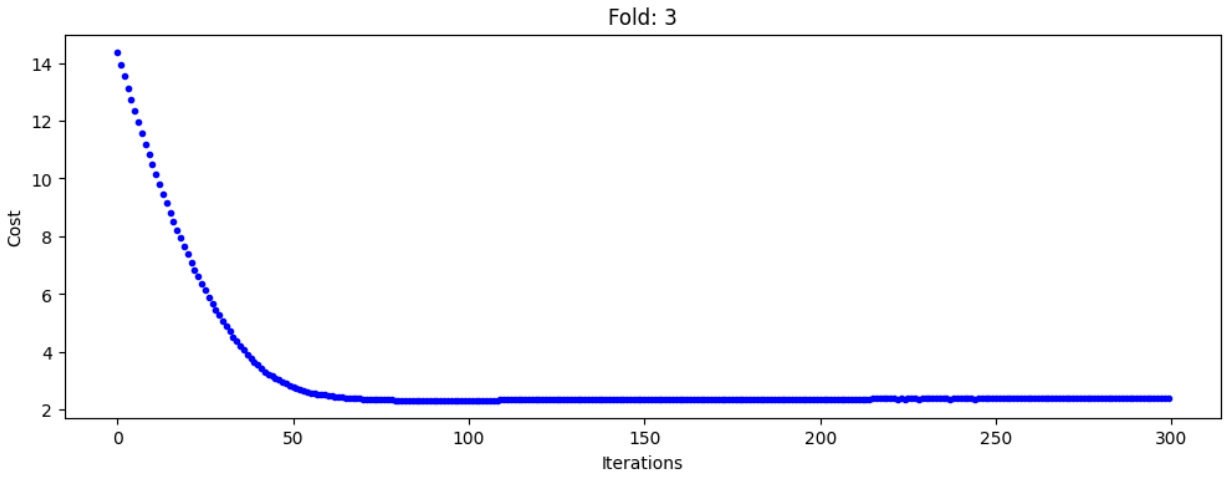
```
*******************************************************
The following graphs are for batch size: 5
*******************************************************
Predictions for fold: 1
Intercept value: 1.3058198105592675
Weight values: [[-0.83260333]
 [ 1.6518994 ]
 [ 1.02606331]
 [ 2.35593935]
 [ 0.29794748]
 [ 1.12623562]
 [ 0.63648359]
 [-1.31050196]
 [ 0.46071214]]
Mean Square Error: 683.122
```



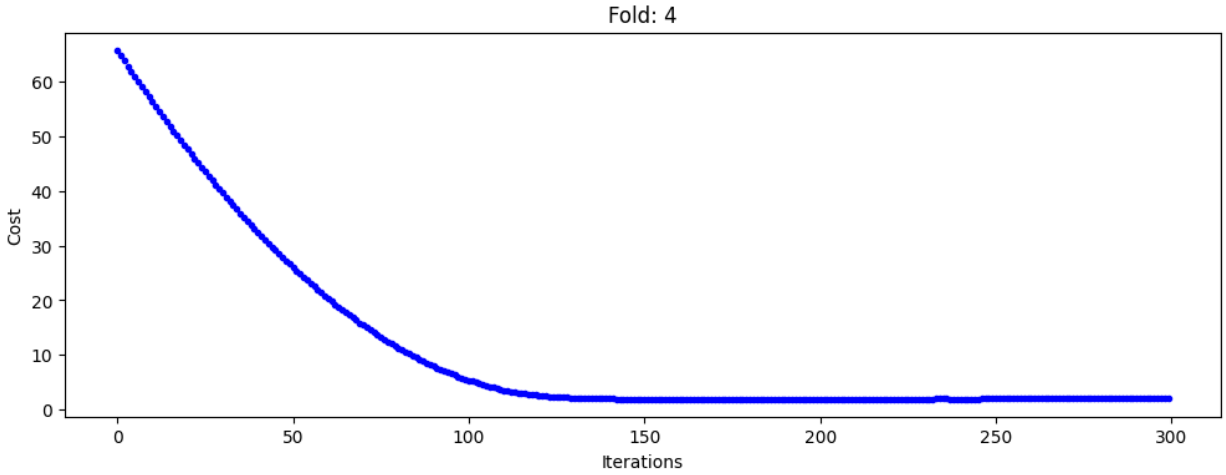

```
Predictions for fold: 2
Intercept value: 1.1058841748288901
Weight values: [[-0.1685344 ]
 [ 1.83397287]
 [ 0.19488735]
 [ 1.55973431]
 [ 1.39980118]
 [ 0.98477718]
 [ 0.29629449]
 [ 0.78344864]
 [ 0.71009191]]
Mean Square Error: 582.246
```



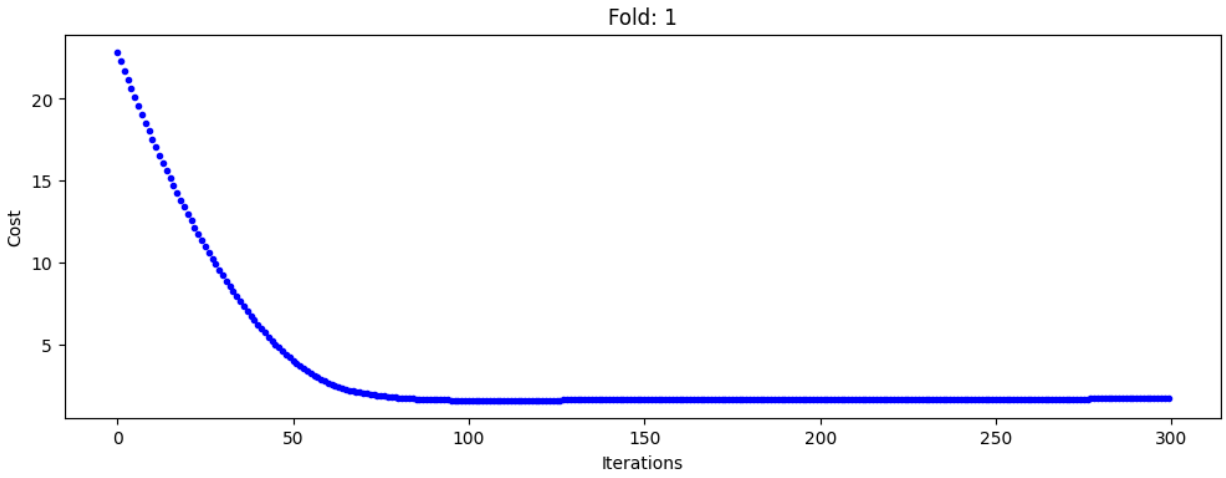

```
Predictions for fold: 3
Intercept value: 1.8556767234293854
Weight values: [[ 0.72892664]
 [-0.25303772]
 [ 1.91055809]
 [ 1.15786615]
 [-1.54846026]
 [ 0.35372458]
 [ 0.55636359]
 [ 0.57453043]
 [ 2.18584175]]
Mean Square Error: 797.607
```



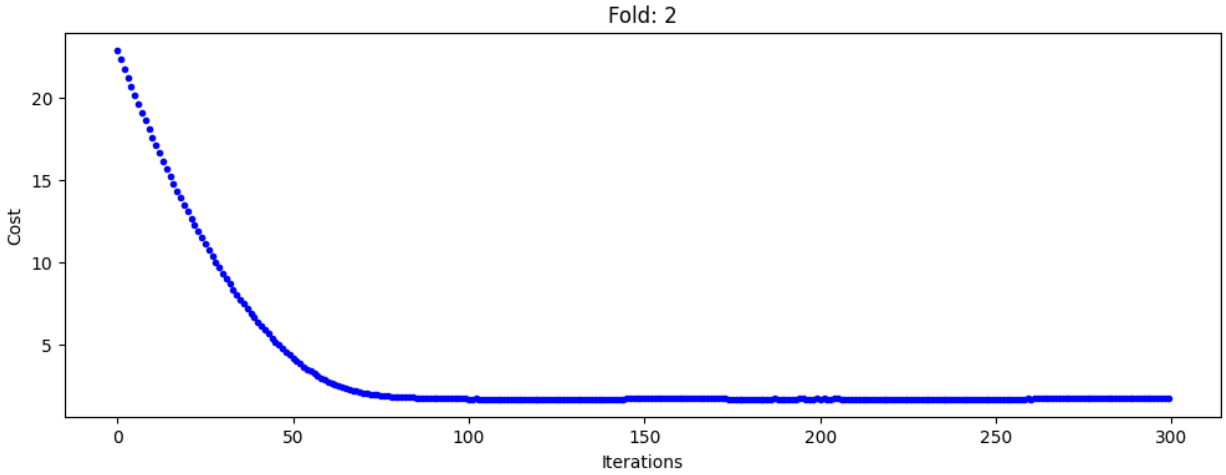

```
Predictions for fold: 4
Intercept value: 0.6066522990336737
Weight values: [[-0.43453143]
 [ 0.1560962 ]
 [ 2.34245751]
 [ 2.54119961]
 [ 1.49853631]
 [ 0.84894266]
 [-0.38937038]
 [ 1.11940969]
 [ 0.92517036]]
Mean Square Error: 669.069
```



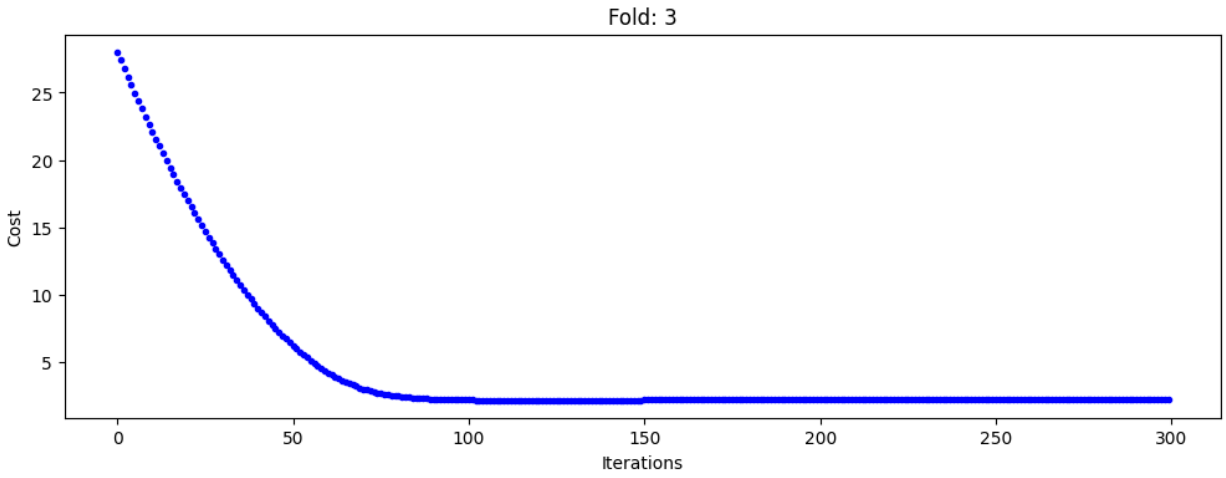

```
**********************************************************
The following graphs are for batch size: 10
**********************************************************
Predictions for fold: 1
Intercept value: 0.8803011017450312
Weight values: [[ 1.22431415]
 [ 2.19528361]
 [ 0.36214831]
 [ 0.58474207]
 [ 0.51512581]
 [ 0.8873021 ]
 [ 1.33410741]
 [ 1.10240911]
 [-0.74639315]]
Mean Square Error: 605.216
```



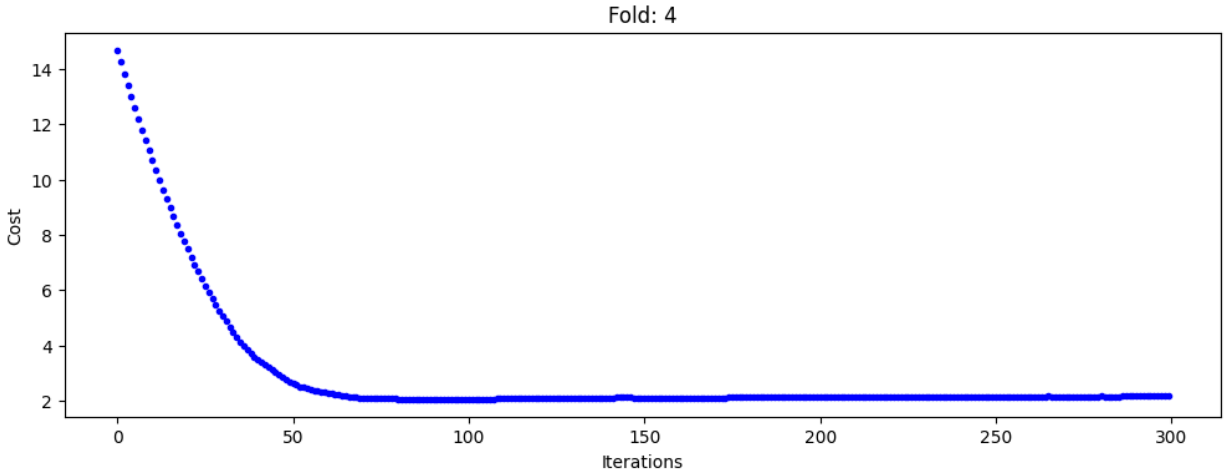

```
Predictions for fold: 2
Intercept value: 1.7983257529753742
Weight values: [[ 1.53978367]
 [ 0.05758434]
 [ 1.11198803]
 [ 0.59650058]
 [-0.94439664]
 [ 1.89736934]
 [ 0.43054666]
 [ 1.05265067]
 [-0.06589365]]
Mean Square Error: 575.57
```



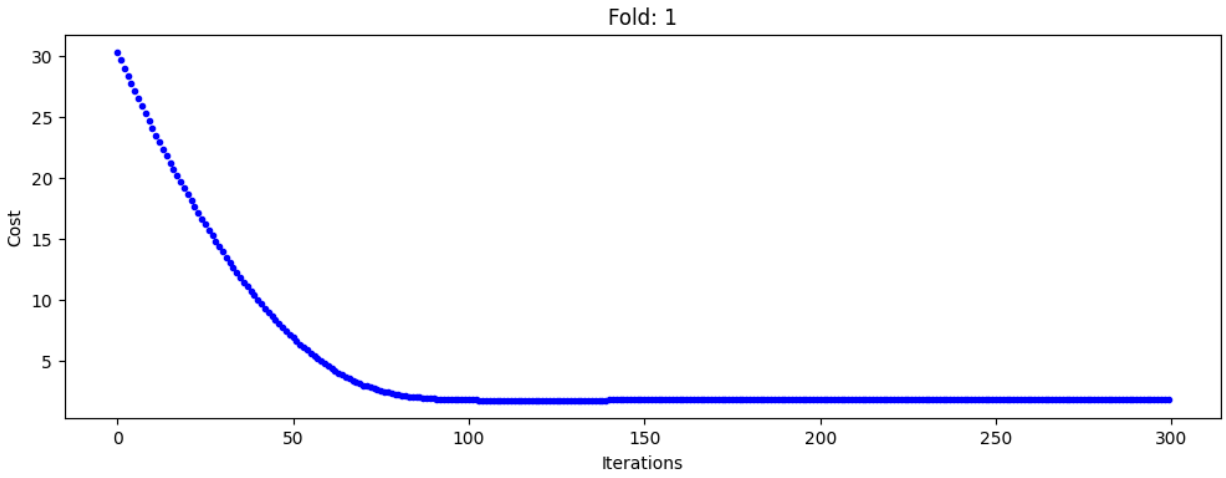

```
Predictions for fold: 3
Intercept value: 0.3123571640142426
Weight values: [[ 2.65511247]
 [ 1.04450479]
 [-0.27334851]
 [ 2.41795589]
 [ 0.30098433]
 [ 1.01613486]
 [ 1.39746757]
 [ 0.29387806]
 [-0.60906418]]
Mean Square Error: 738.484
```



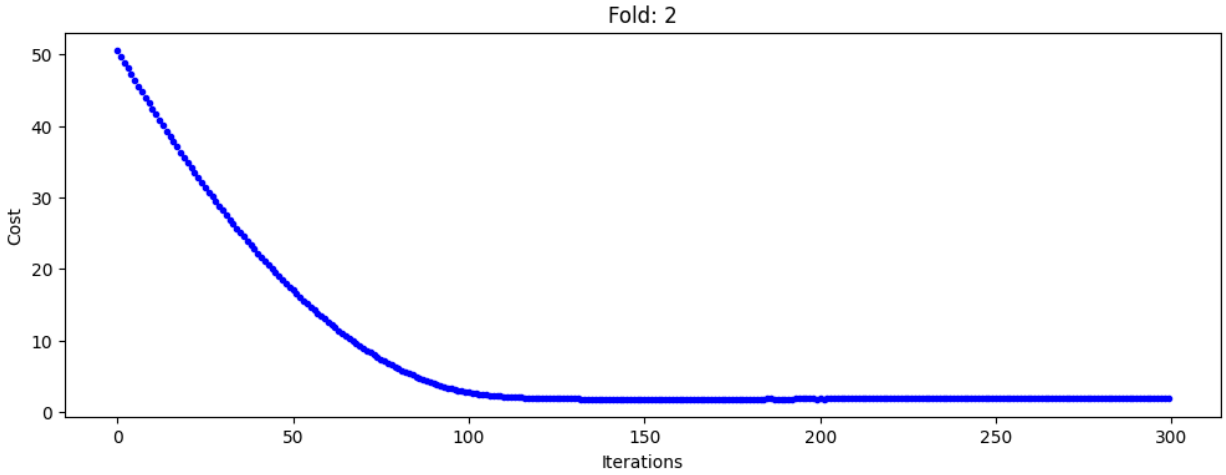

```
Predictions for fold: 4
Intercept value: 1.7107617560015387
Weight values: [[ 2.32604286]
 [ 1.14956343]
 [ 0.95899964]
 [ 0.96664767]
 [ 1.66125646]
 [ 1.28910175]
 [-0.67458782]
 [-2.09094438]
 [-0.15582347]]
Mean Square Error: 778.013
```



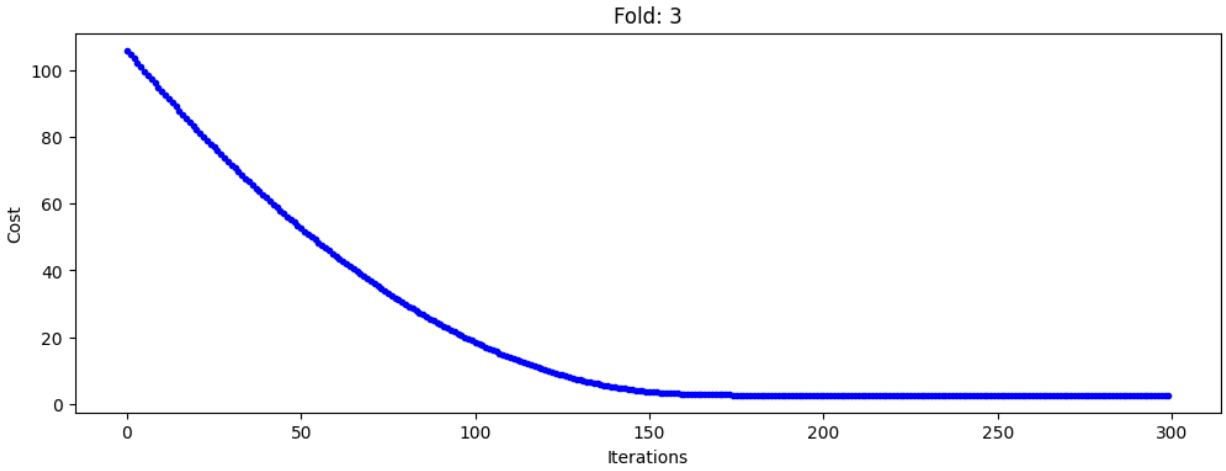

```
**********************************************************
The following graphs are for batch size: 20
**********************************************************
Predictions for fold: 1
Intercept value: 0.40648304852976347
Weight values: [[ 0.91344989]
 [ 1.8427132 ]
 [ 0.20897705]
 [ 1.10326105]
 [ 0.06425347]
 [ 1.85842348]
 [ 1.44601997]
 [ 0.02430904]
 [-0.03353738]]
Mean Square Error: 647.036
```



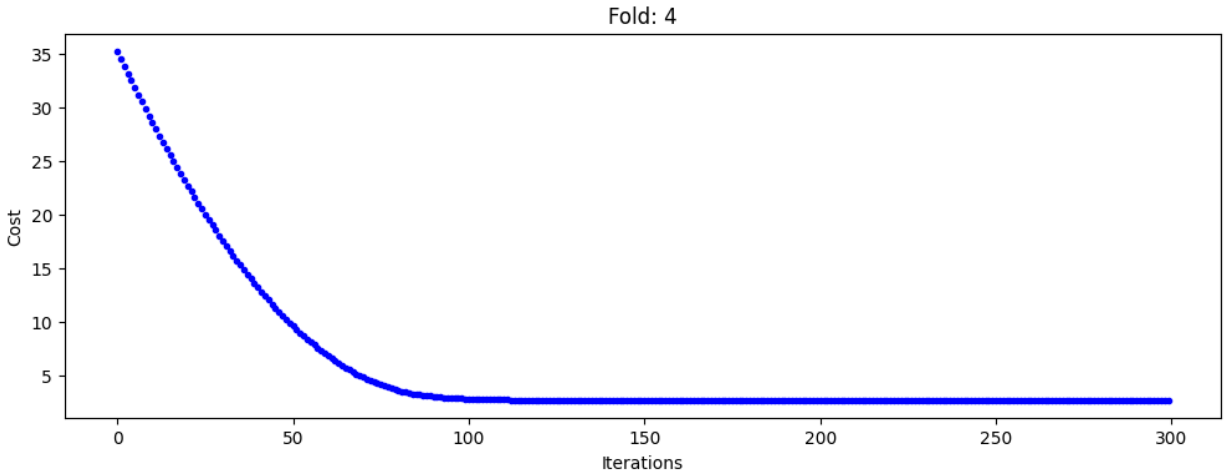

```
Predictions for fold: 2
Intercept value: 0.7683155040653272
Weight values: [[-0.38237775]
 [ 1.47400811]
 [-0.06307803]
 [ 1.3751641 ]
 [ 1.3265875 ]
 [ 1.51637355]
 [ 0.70941034]
 [ 0.73666571]
 [ 1.65175459]]
Mean Square Error: 605.857
```



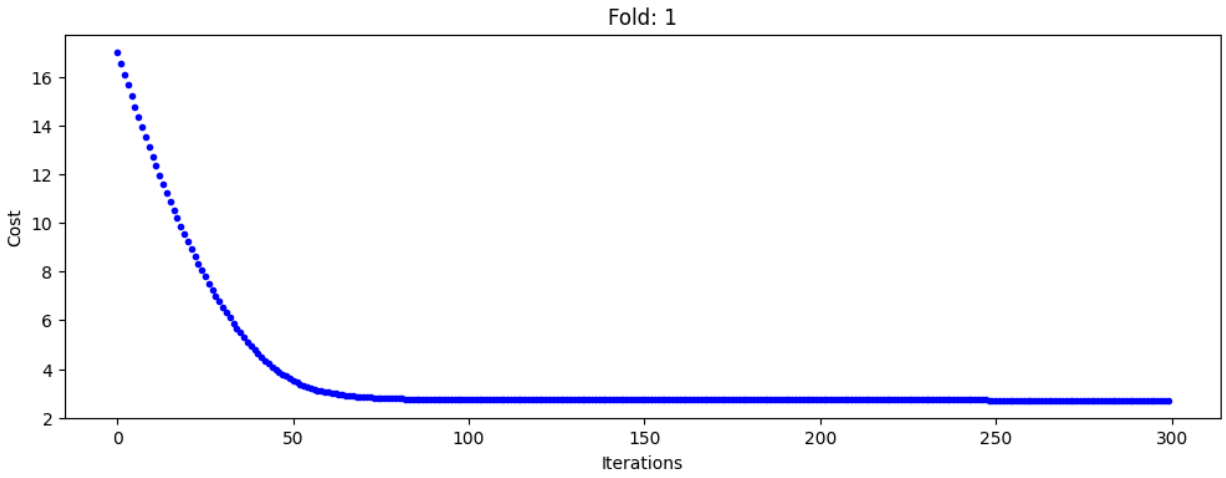

```
Predictions for fold: 3
Intercept value: -0.479964475025906
Weight values: [[1.42668671]
 [0.43215092]
 [0.72897471]
 [0.87498272]
 [1.41446517]
 [1.20096539]
 [1.86417165]
 [0.92947096]
 [1.95456455]]
Mean Square Error: 875.175
```



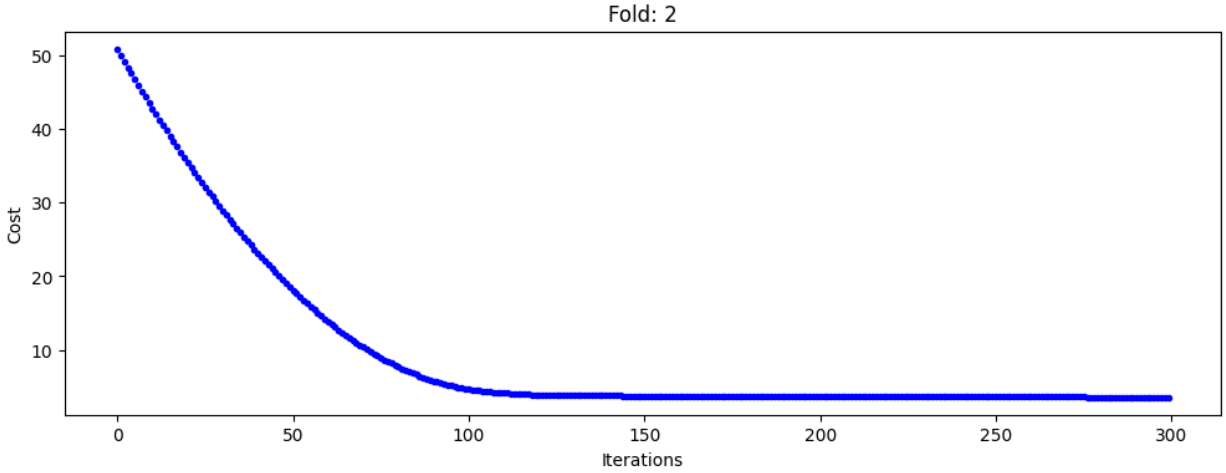

```
Predictions for fold: 4
Intercept value: 0.00875358578621343
Weight values: [[ 1.30519277]
 [-0.36088058]
 [ 2.56662124]
 [ 1.47695344]
 [ 0.11639203]
 [ 1.52893068]
 [-0.03124507]
 [ 0.07496587]
 [ 2.17710575]]
Mean Square Error: 909.223
```



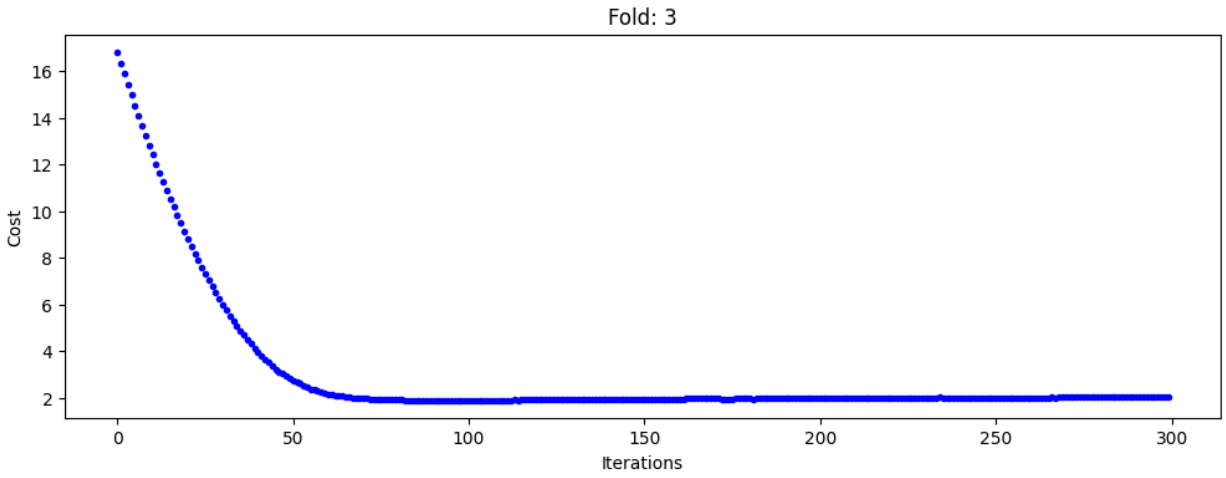

```
**********************************************************
The following graphs are for batch size: 50
**********************************************************
Predictions for fold: 1
Intercept value: 0.32462118795548794
Weight values: [[ 0.16863407]
 [ 2.60442254]
 [ 1.14951044]
 [ 0.2467027 ]
 [-0.2823736 ]
 [ 0.03448251]
 [ 1.58290563]
 [ 0.6826036 ]
 [ 2.00325908]]
Mean Square Error: 980.894
```



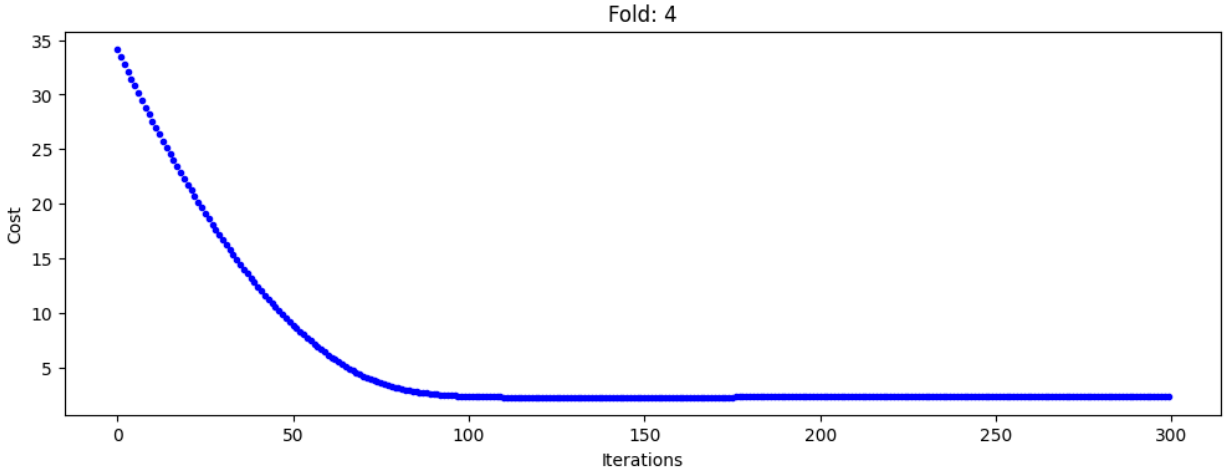

```
Predictions for fold: 2
Intercept value: 1.156985494068983
Weight values: [[ 1.90549246]
 [ 0.91479222]
 [-0.05480651]
 [ 0.2748935 ]
 [ 1.85039537]
 [-1.51854641]
 [ 1.41105528]
 [ 2.41872652]
 [ 2.82148806]]
Mean Square Error: 1189.518
```



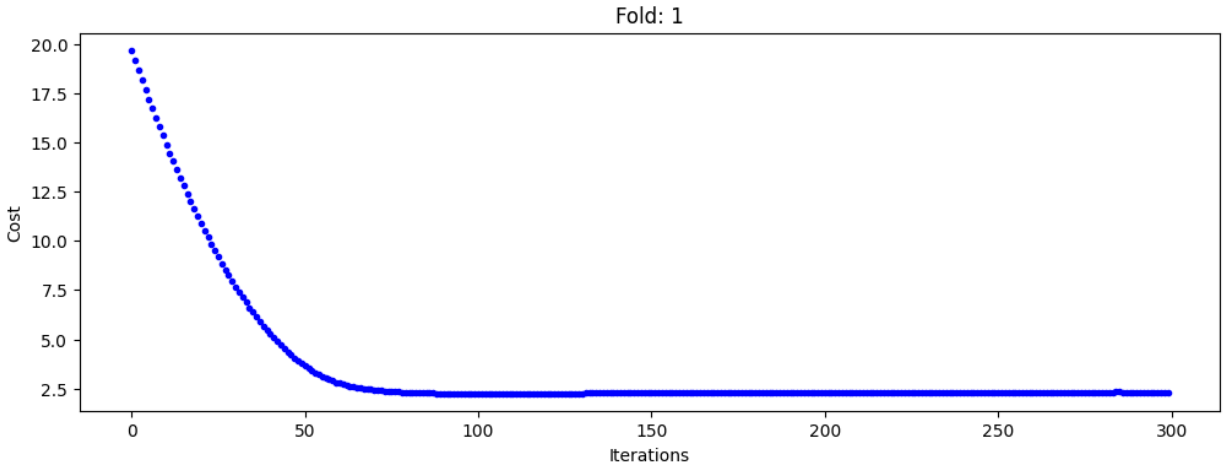

```
Predictions for fold: 3
Intercept value: 0.8099328142768258
Weight values: [[1.22896402]
 [1.14741418]
 [0.62865291]
 [0.91024931]
 [0.46557703]
 [0.69884905]
 [1.14326222]
 [1.10277139]
 [0.75499363]]
Mean Square Error: 668.557
```



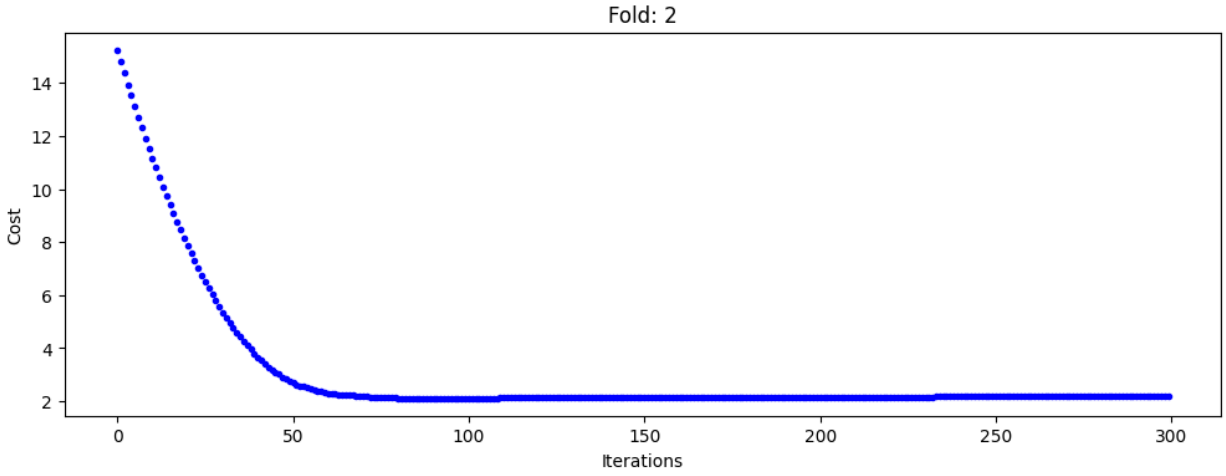

```
Predictions for fold: 4
Intercept value: -0.23692428746418043
Weight values: [[ 1.8776917 ]
 [-0.01357011]
 [ 0.83357714]
 [ 1.06618504]
 [ 3.13336798]
 [ 0.86107081]
 [ 2.03577696]
 [ 1.11510936]
 [ 0.22740362]]
Mean Square Error: 835.293
```



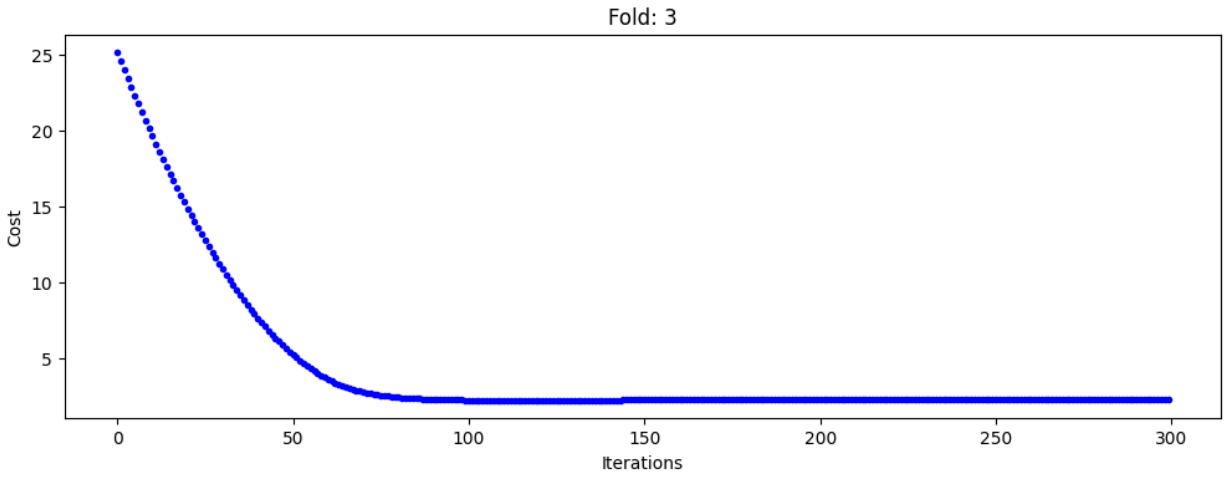

```
**********************************************************
The following graphs are for batch size: 100
**********************************************************
Predictions for fold: 1
Intercept value: 0.7681530037795047
Weight values: [[ 1.92553915]
 [ 1.4670118 ]
 [ 0.32135477]
 [-0.54927601]
 [ 0.61940848]
 [ 0.37423531]
 [ 2.82599078]
 [ 0.59409942]
 [ 0.59944734]]
Mean Square Error: 851.403
```



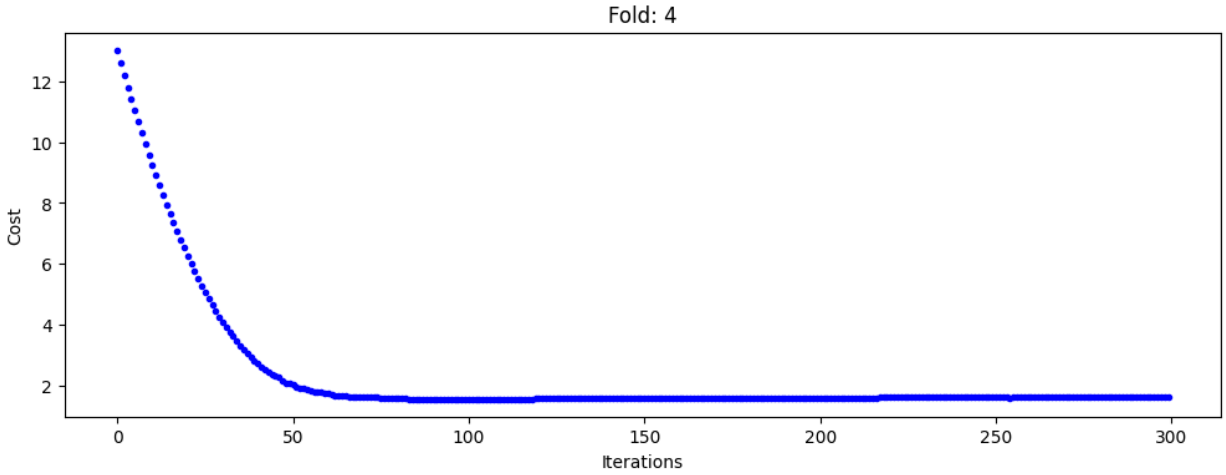

```
Predictions for fold: 2
Intercept value: 1.1515508271293404
Weight values: [[ 1.4860854 ]
 [ 1.342675  ]
 [ 0.20909531]
 [ 1.83616436]
 [ 0.59377937]
 [ 0.49544508]
 [ 0.95238645]
 [-0.35075107]
 [ 0.22470766]]
Mean Square Error: 728.226
```




```
Predictions for fold: 3
Intercept value: 1.5734777930079236
Weight values: [[ 0.21363993]
 [ 1.36981825]
 [ 1.95891883]
 [ 0.23639047]
 [ 0.95096285]
 [ 0.23512187]
 [ 0.86909953]
 [-0.85684717]
 [ 1.06856242]]
Mean Square Error: 712.815
```



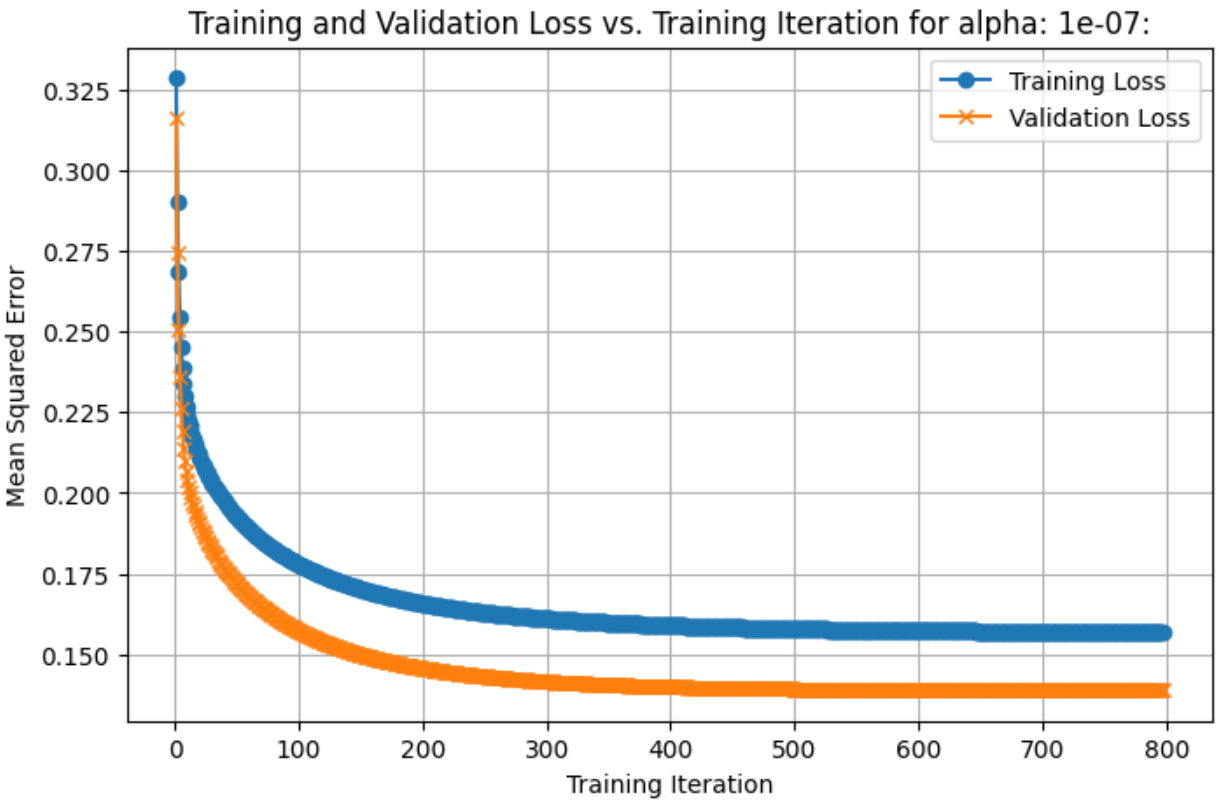

```
Predictions for fold: 4
Intercept value: 1.7872496148083925
Weight values: [[-0.22203706]
 [ 0.6378978 ]
 [ 0.89380434]
 [ 1.41140867]
 [-1.36184969]
 [ 1.73241667]
 [ 0.50713331]
 [ 0.26530276]
 [ 0.97401144]]
Mean Square Error: 552.405
```



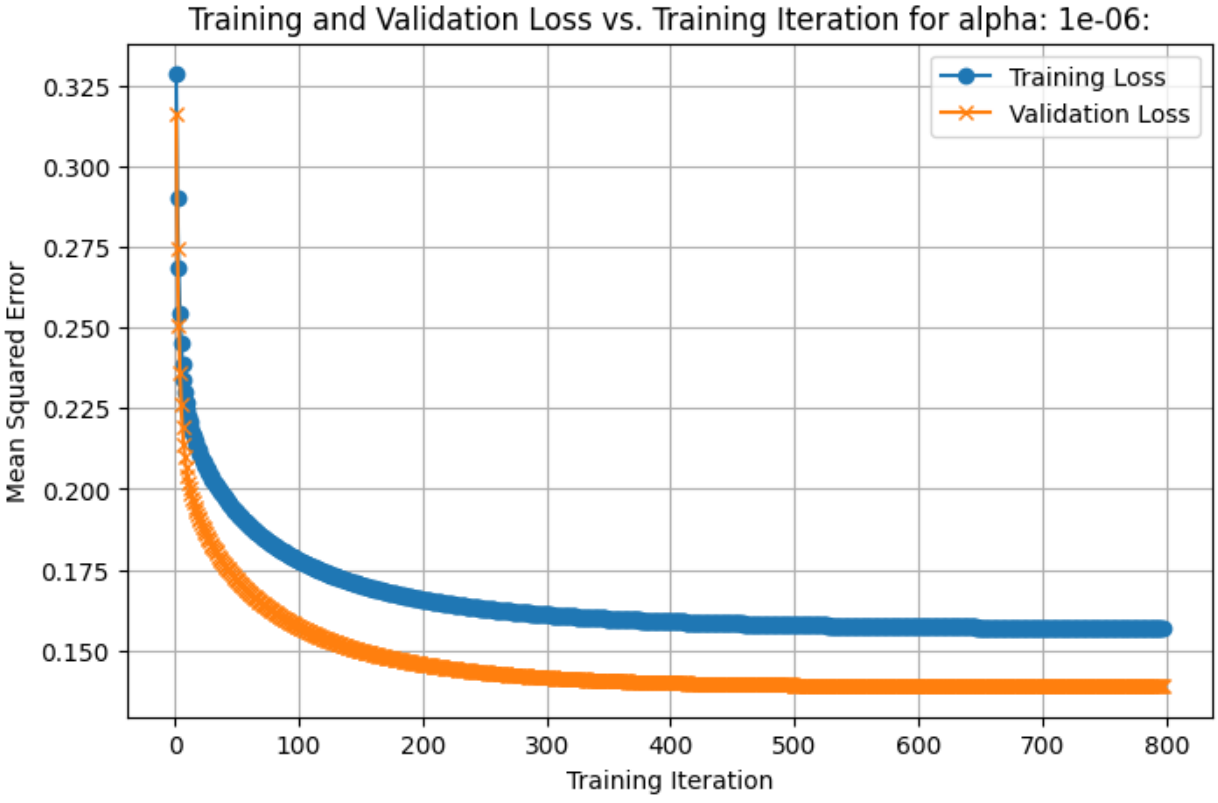
In [ ]:
```
# Running SGD Regression and plotting the training and validation loss
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error

# Create the SGDRegressor model with appropriate hyperparameters
alphaList = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
for alpha in alphaList:
  sgd_model = SGDRegressor(loss='squared_error', alpha=alpha, max_iter=1000, ra

  # Initialize lists to store training and validation loss
  training_loss = []
  validation_loss = []
```

```
# Number of training iterations
n_iterations = 1000

# Early stopping parameters
early_stopping_rounds = 10  # Number of iterations with no improvement to wai
best_val_loss = float('inf')  # Initialize the best validation loss to positi
no_improvement_count = 0  # Initialize the count of iterations with no improv

for iteration in range(n_iterations):
    # Fit the model for one iteration (one pass through the training data)
    sgd_model.partial_fit(X_train, y_train)

    # Predict on the training data
    y_train_pred = sgd_model.predict(X_train)

    # Calculate training loss (Mean Squared Error) and append to the list
    train_loss = mean_squared_error(y_train, y_train_pred)
    training_loss.append(train_loss)

    # Predict on the validation data
    y_val_pred = sgd_model.predict(X_test)

    # Calculate validation loss (Mean Squared Error) and append to the list
    val_loss = mean_squared_error(y_test, y_val_pred)
    validation_loss.append(val_loss)

    # Check for early stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        no_improvement_count = 0
    else:
        no_improvement_count += 1

    if no_improvement_count >= early_stopping_rounds:
        print(f"Early stopping at iteration {iteration + 1} due to no improve
        break

    # Plot training and validation loss as a function of training iteration
plt.figure(figsize=(8, 5))
plt.plot(range(1, iteration + 2), training_loss, label='Training Loss', marke
plt.plot(range(1, iteration + 2), validation_loss, label='Validation Loss', m
plt.xlabel('Training Iteration')
plt.ylabel('Mean Squared Error')
plt.title(f'Training and Validation Loss vs. Training Iteration for alpha: {a
plt.legend()
plt.grid()
plt.show()

# Final model evaluation
y_pred_sgd = sgd_model.predict(X_test)
final_mse = mean_squared_error(y_test, y_pred_sgd)
print(f"Final Mean Squared Error (SGD) for alpha: {alpha}:", final_mse)
```

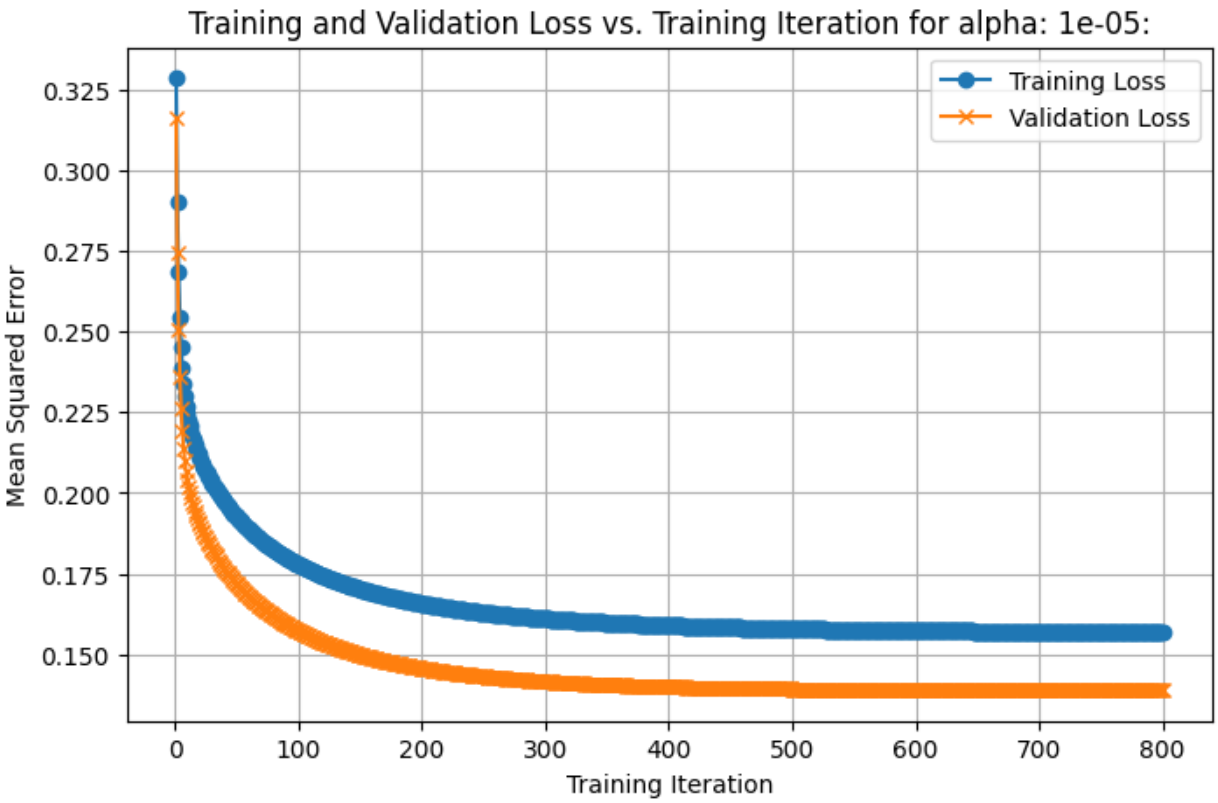
```
Early stopping at iteration 797 due to no improvement in validation loss.
```



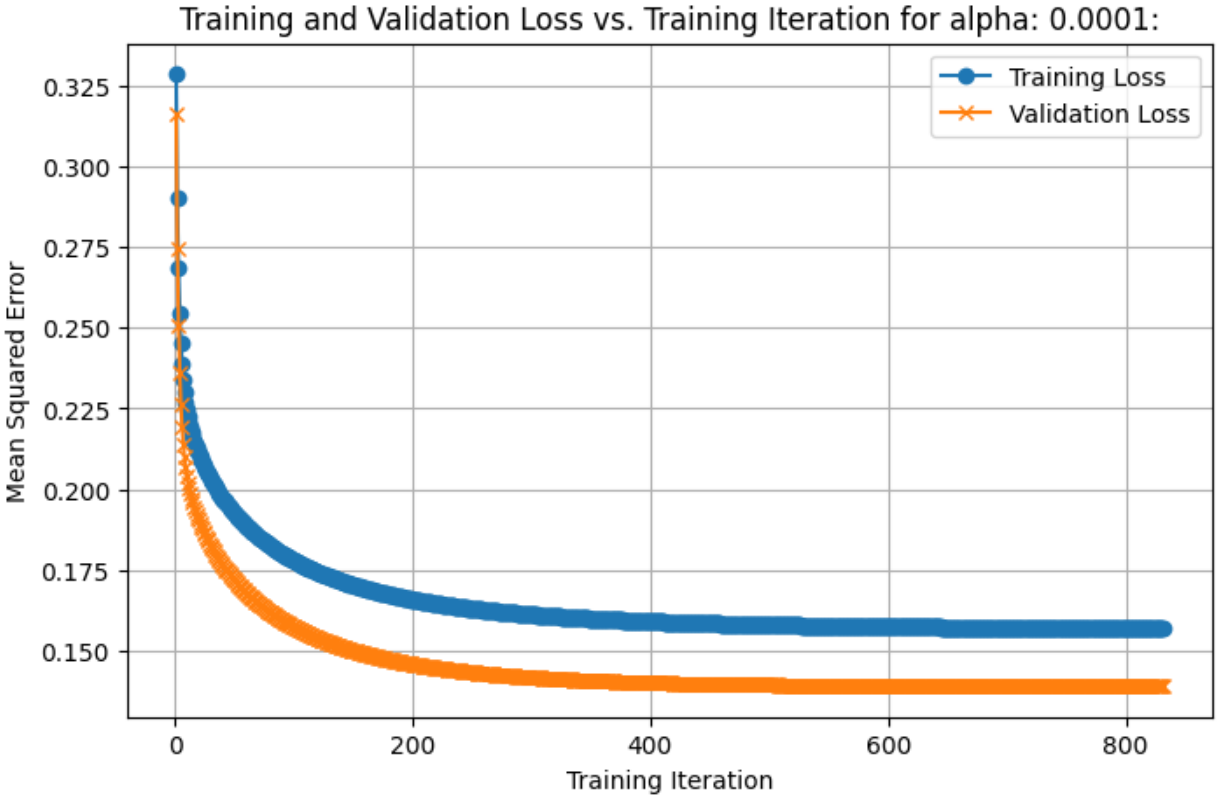

```
Final Mean Squared Error (SGD) for alpha: 1e-07: 0.13883458169225515
Early stopping at iteration 797 due to no improvement in validation loss.
```



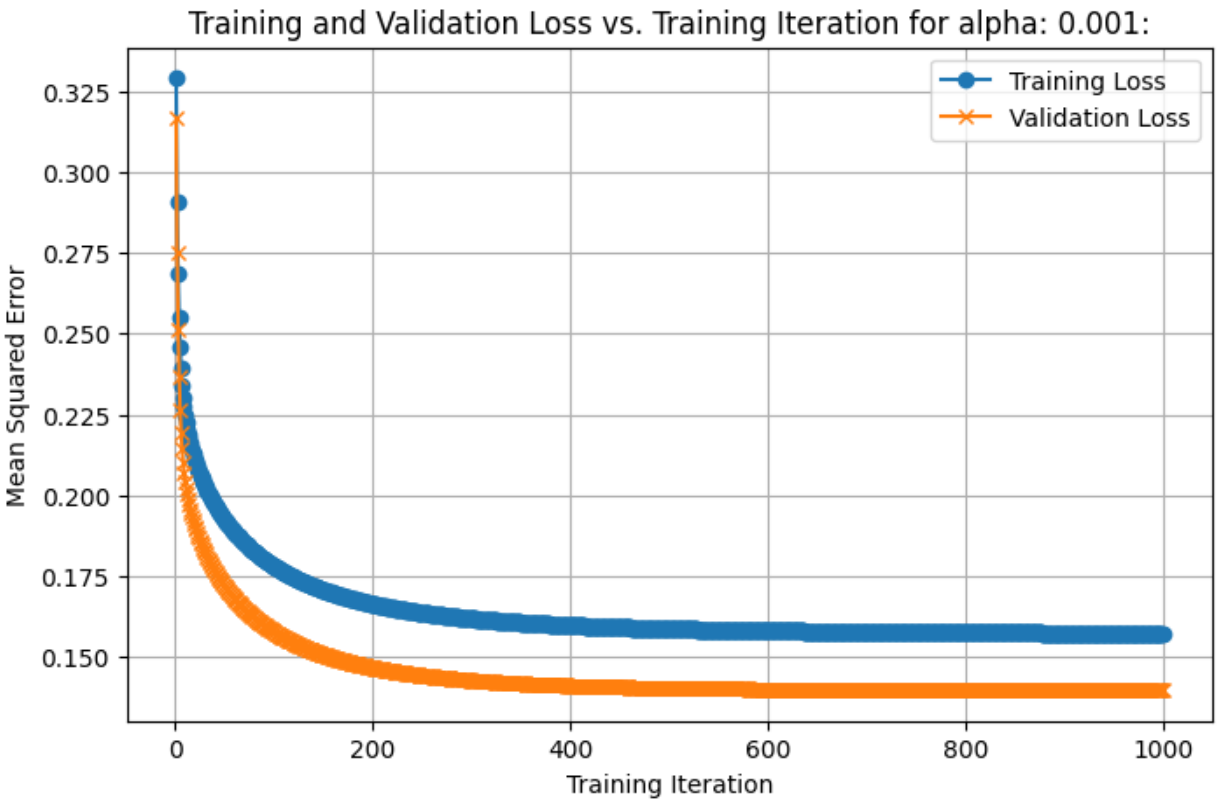

```
Final Mean Squared Error (SGD) for alpha: 1e-06: 0.13883515924755457
Early stopping at iteration 800 due to no improvement in validation loss.
```



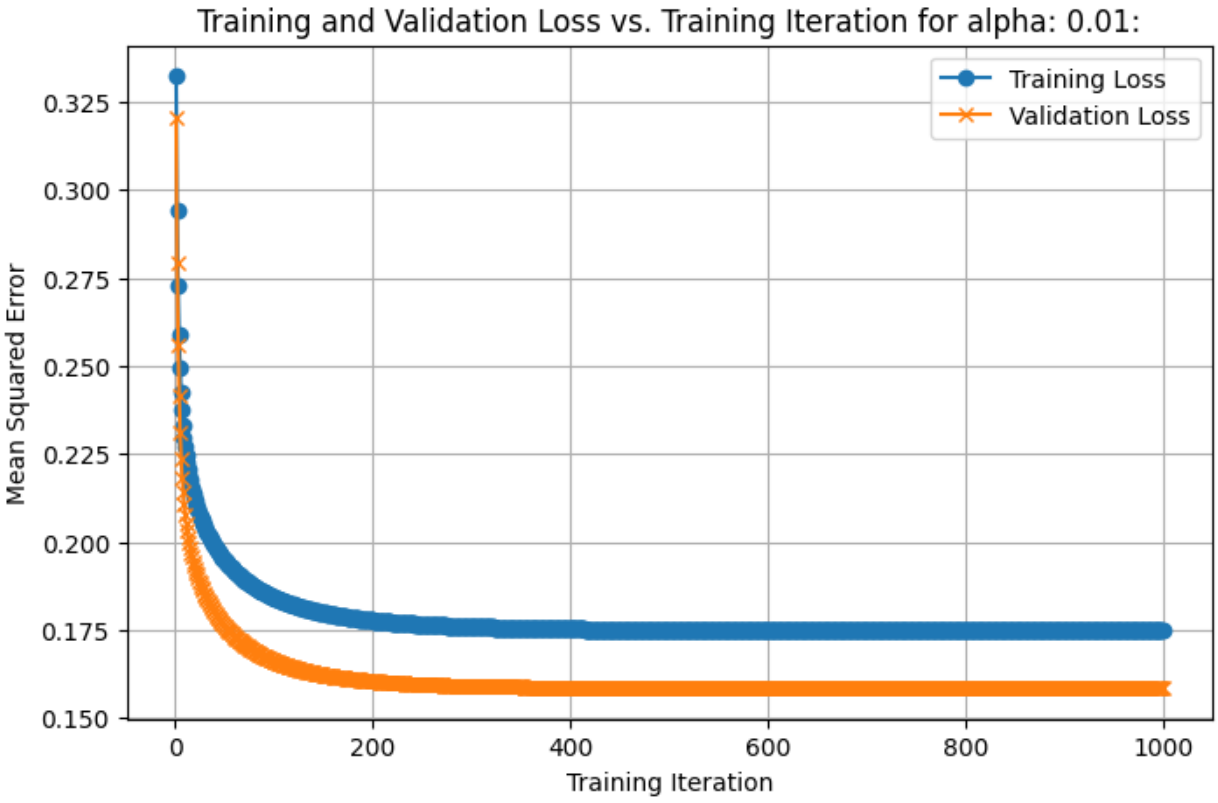

```
Final Mean Squared Error (SGD) for alpha: 1e-05: 0.13884106006121624
Early stopping at iteration 831 due to no improvement in validation loss.
```



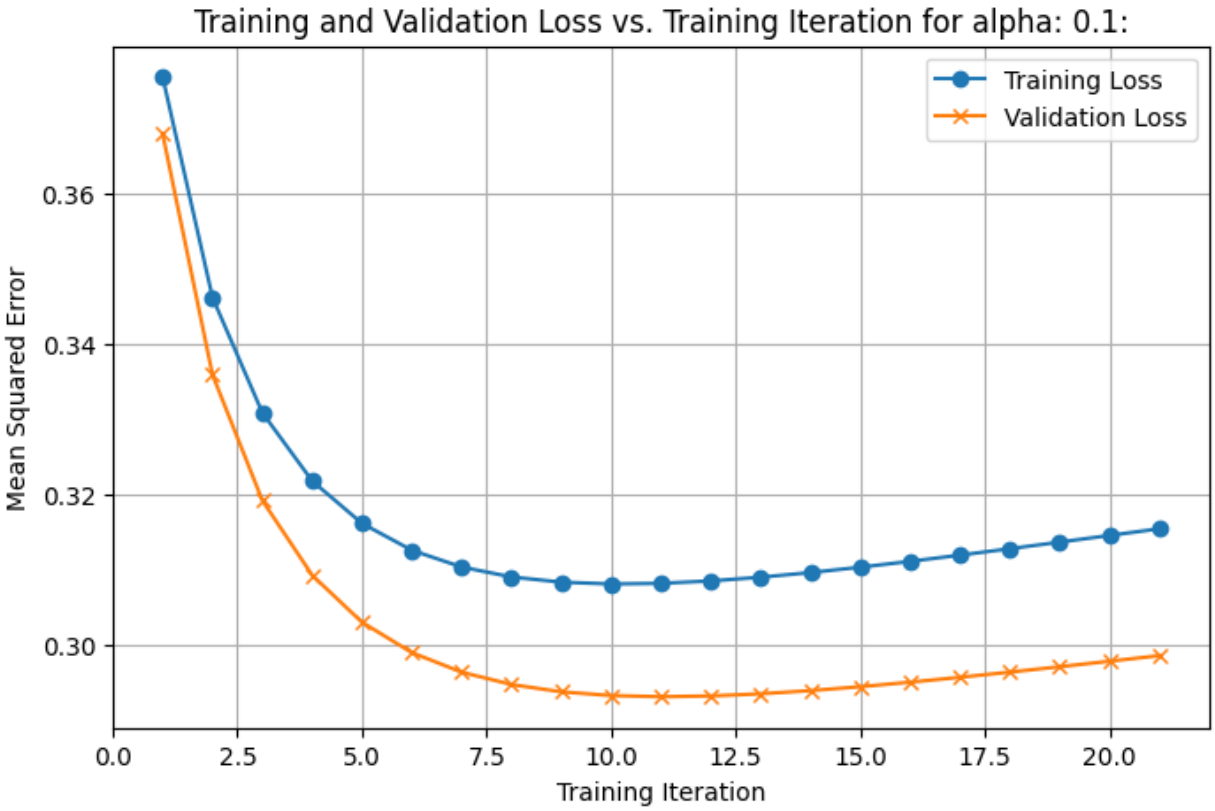

```
Final Mean Squared Error (SGD) for alpha: 0.0001: 0.1389027463196937
```




```
Final Mean Squared Error (SGD) for alpha: 0.001: 0.13981321083989706
```




```
Final Mean Squared Error (SGD) for alpha: 0.01: 0.15843002471574336
Early stopping at iteration 21 due to no improvement in validation loss.
```




```
Final Mean Squared Error (SGD) for alpha: 0.1: 0.2985342973169805
Early stopping at iteration 13 due to no improvement in validation loss.
```


```
Final Mean Squared Error (SGD) for alpha: 1: 0.970938087331455
```

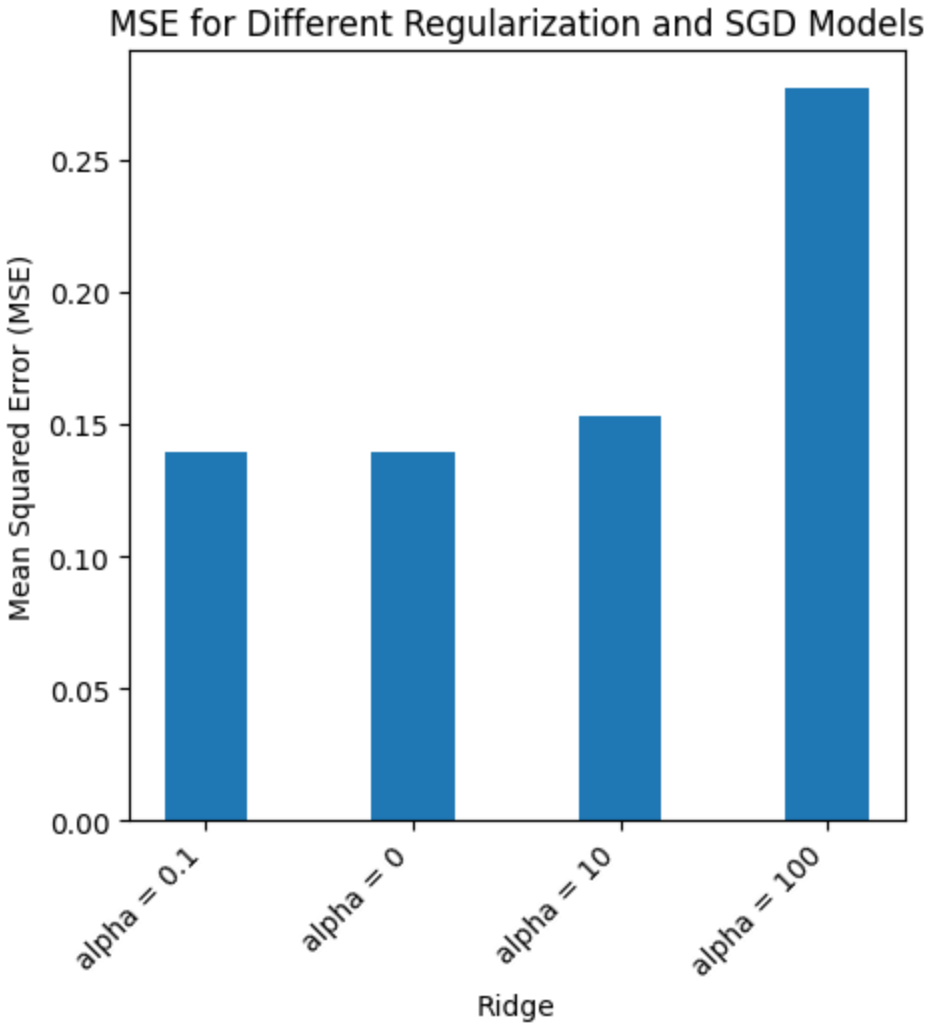
In [ ]:
```
#Amended code to calculate the Ridge, Lasso and Elastic Net
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

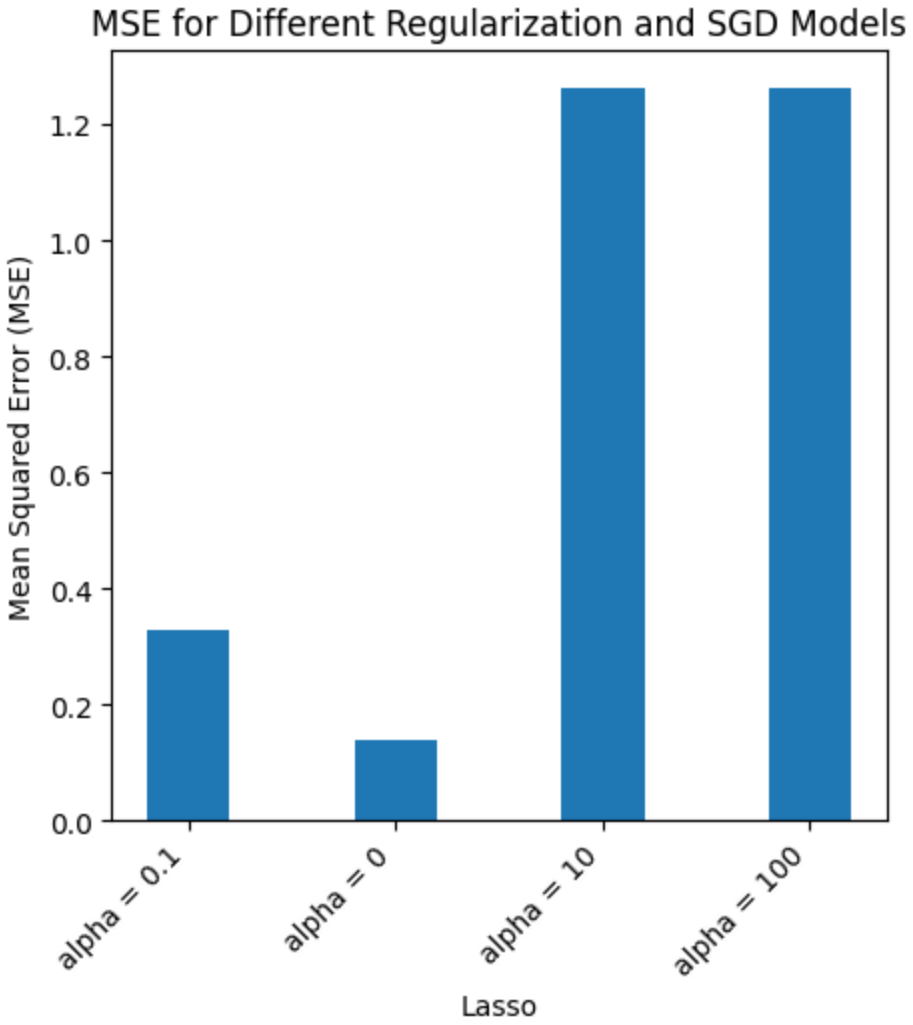
```
from sklearn.linear_model import Ridge, Lasso, ElasticNet, SGDRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Initialize lists to store performance metrics
model_performance = pd.DataFrame(columns=('Model', 'penalty', 'MSE','R2-sq err
model_r2 = {}

# Regularization penalties to try
penalties = [0.1, 0, 10, 100]

# Define the maximum number of iterations for each model
max_iterations = 1000

# Create a dictionary to store model names for reference
model_names = {
    'Ridge': Ridge(),
    'Lasso': Lasso(),
    'Elastic Net': ElasticNet()
}

# Iterate through different penalty terms for Ridge, Lasso, and Elastic Net
for penalty in penalties:
    # Ridge
    ridge_model = Ridge(alpha=penalty, max_iter=max_iterations)
    ridge_model.fit(X_train, y_train)
    y_pred_ridge = ridge_model.predict(X_test)
    mse_ridge = mean_squared_error(y_test, y_pred_ridge)
    r2_ridge = r2_score(y_test, y_pred_ridge)
    model_performance = model_performance.append({'Model':'Ridge','penalty': f

    # Lasso
    lasso_model = Lasso(alpha=penalty, max_iter=max_iterations)
    lasso_model.fit(X_train, y_train)
    y_pred_lasso = lasso_model.predict(X_test)
    mse_lasso = mean_squared_error(y_test, y_pred_lasso)
    r2_lasso = r2_score(y_test, y_pred_lasso)
    model_performance = model_performance.append({'Model':'Lasso','penalty': f

    # Elastic Net
    elastic_net_model = ElasticNet(alpha=penalty, max_iter=max_iterations)
    elastic_net_model.fit(X_train, y_train)
    y_pred_elastic_net = elastic_net_model.predict(X_test)
    mse_elastic_net = mean_squared_error(y_test, y_pred_elastic_net)
    r2_elastic_net = r2_score(y_test, y_pred_elastic_net)
    model_performance = model_performance.append({'Model':'Elastic Net','penalt

print(model_performance)

# Plot the MSE values for different models
for m in model_names.keys():
    plt.figure(figsize=(5, 5))
    plt.bar(model_performance[model_performance['Model'] == m]['penalty'], mode
    plt.xlabel(m)
    plt.ylabel('Mean Squared Error (MSE)')
    plt.title('MSE for Different Regularization and SGD Models')
    plt.xticks(rotation=45, ha='right')
    plt.show()
```

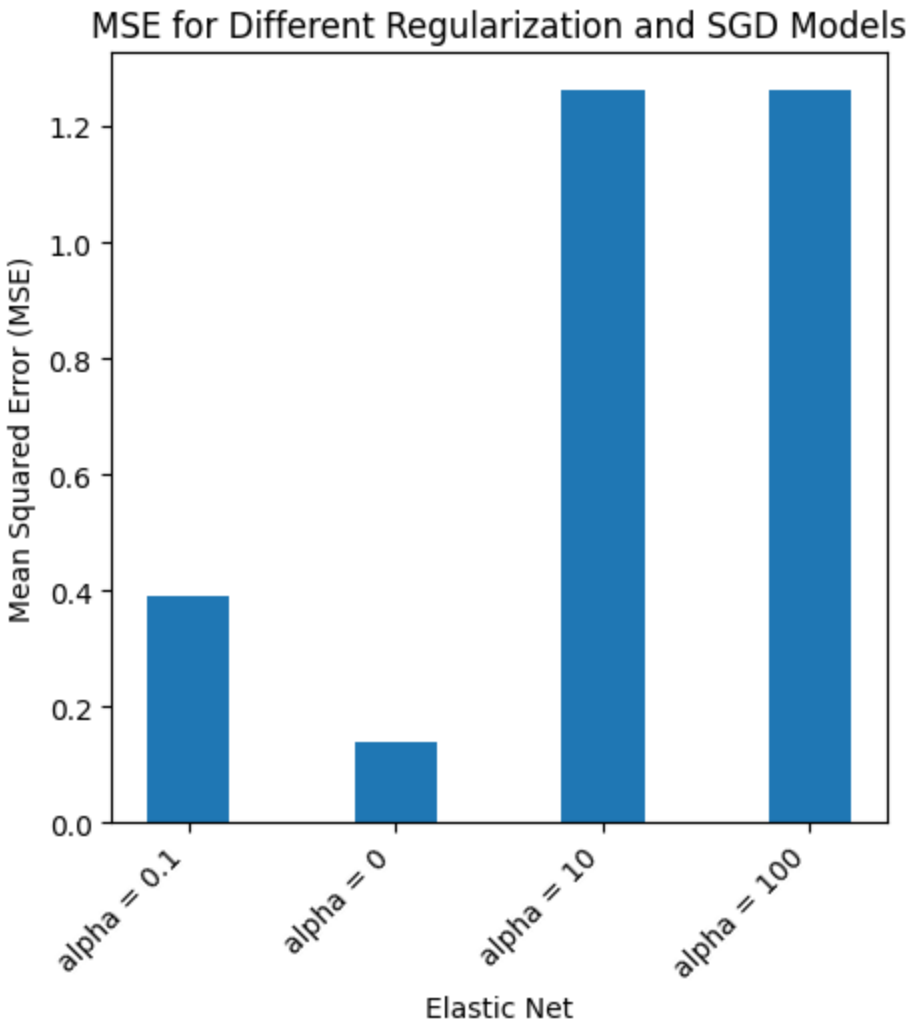
```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
<ipython-input-53-8f247c5b1350>:33: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Ridge','penalty': f'a
lpha = {penalty}','MSE': mse_ridge, 'R2-sq err': r2_ridge}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:41: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Lasso','penalty': f'a
lpha = {penalty}','MSE': mse_lasso, 'R2-sq err': r2_lasso}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:49: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Elastic Net','penalt
y': f'alpha = {penalty}','MSE': mse_elastic_net, 'R2-sq err': r2_elastic_net},
ignore_index=True)
<ipython-input-53-8f247c5b1350>:33: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Ridge','penalty': f'a
lpha = {penalty}','MSE': mse_ridge, 'R2-sq err': r2_ridge}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:37: UserWarning: With alpha=0, this algorithm
does not converge well. You are advised to use the LinearRegression estimator
  lasso_model.fit(X_train, y_train)
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: UserWarning: Coordinate descent with no regularization may lead to
unexpected results and is discouraged.
  model = cd_fast.enet_coordinate_descent(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 1.069e+02, tolerance: 1.773e-01 Linear
regression models with null weight for the l1 regularization term are more eff
iciently fitted using one of the solvers implemented in sklearn.linear_model.R
idge/RidgeCV instead.
  model = cd_fast.enet_coordinate_descent(
<ipython-input-53-8f247c5b1350>:41: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Lasso','penalty': f'a
lpha = {penalty}','MSE': mse_lasso, 'R2-sq err': r2_lasso}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:45: UserWarning: With alpha=0, this algorithm
does not converge well. You are advised to use the LinearRegression estimator
  elastic_net_model.fit(X_train, y_train)
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: UserWarning: Coordinate descent with no regularization may lead to
unexpected results and is discouraged.
  model = cd_fast.enet_coordinate_descent(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 1.069e+02, tolerance: 1.773e-01 Linear
regression models with null weight for the l1 regularization term are more eff
iciently fitted using one of the solvers implemented in sklearn.linear_model.R
```

```
idge/RidgeCV instead.
  model = cd_fast.enet_coordinate_descent(
<ipython-input-53-8f247c5b1350>:49: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Elastic Net','penalt
y': f'alpha = {penalty}','MSE': mse_elastic_net, 'R2-sq err': r2_elastic_net},
ignore_index=True)
<ipython-input-53-8f247c5b1350>:33: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Ridge','penalty': f'a
lpha = {penalty}','MSE': mse_ridge, 'R2-sq err': r2_ridge}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:41: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Lasso','penalty': f'a
lpha = {penalty}','MSE': mse_lasso, 'R2-sq err': r2_lasso}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:49: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Elastic Net','penalt
y': f'alpha = {penalty}','MSE': mse_elastic_net, 'R2-sq err': r2_elastic_net},
ignore_index=True)
<ipython-input-53-8f247c5b1350>:33: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Ridge','penalty': f'a
lpha = {penalty}','MSE': mse_ridge, 'R2-sq err': r2_ridge}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:41: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Lasso','penalty': f'a
lpha = {penalty}','MSE': mse_lasso, 'R2-sq err': r2_lasso}, ignore_index=True)
<ipython-input-53-8f247c5b1350>:49: FutureWarning: The frame.append method is
deprecated and will be removed from pandas in a future version. Use pandas.con
cat instead.
  model_performance = model_performance.append({'Model':'Elastic Net','penalt
y': f'alpha = {penalty}','MSE': mse_elastic_net, 'R2-sq err': r2_elastic_net},
ignore_index=True)
```

	Model	penalty	MSE	R2-sq err
0	Ridge	alpha = 0.1	0.139139	8.898437e-01
1	Lasso	alpha = 0.1	0.329192	7.393790e-01
2	Elastic Net	alpha = 0.1	0.392239	6.894649e-01
3	Ridge	alpha = 0	0.139131	8.898497e-01
4	Lasso	alpha = 0	0.139131	8.898497e-01
5	Elastic Net	alpha = 0	0.139131	8.898497e-01
6	Ridge	alpha = 10	0.152665	8.791354e-01
7	Lasso	alpha = 10	1.263106	-8.826513e-08
8	Elastic Net	alpha = 10	1.263106	-8.826513e-08
9	Ridge	alpha = 100	0.277220	7.805248e-01
10	Lasso	alpha = 100	1.263106	-8.826513e-08
11	Elastic Net	alpha = 100	1.263106	-8.826513e-08


MSE for Different Regularization and SGD Models
Mean Squared Error (MSE)
0.25
0.20
0.15
0.10
0.05
0.00
alpha = 0.1
alpha = 0
alpha = 10
alpha = 100
Ridge


MSE for Different Regularization and SGD Models
Mean Squared Error (MSE)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
alpha = 0.1
alpha = 0
alpha = 10
alpha = 100
Lasso



In [ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Define the hyperparameters to tune
batch_sizes = [1, 8, 16, 32, 64, 128]
learning_rates = [0.01, 0.1]

results = []

for batch_size in batch_sizes:
    for learning_rate in learning_rates:
        # Create and train the linear regression model with specified hyperpara
        model = LinearRegression()

        # Assuming you want to set the learning rate (eta0) for the model
        model_eta0=learning_rate

        # Train the model using the current batch size
        model.fit(X_train, y_train,learning_rate)

        # Evaluate the model on the test data
        y_pred = model.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        results.append((batch_size, learning_rate, mse))

print(results)
```

```
# Find the best combination of hyperparameters based on the lowest RMSE
best_result = min(results, key=lambda x: x[2])
best_batch_size, best_learning_rate, best_rmse = best_result

print(f"Best Batch Size: {best_batch_size}")
print(f"Best Learning Rate: {best_learning_rate}")
print(f"Lowest RMSE: {best_rmse}")
```

```
[(1, 0.01, 0.13913141171548538), (1, 0.1, 0.13913141171548538), (8, 0.01, 0.13
913141171548538), (8, 0.1, 0.13913141171548538), (16, 0.01, 0.1391314117154853
8), (16, 0.1, 0.13913141171548538), (32, 0.01, 0.13913141171548538), (32, 0.1,
0.13913141171548538), (64, 0.01, 0.13913141171548538), (64, 0.1, 0.13913141171
548538), (128, 0.01, 0.13913141171548538), (128, 0.1, 0.13913141171548538)]
Best Batch Size: 1
Best Learning Rate: 0.01
Lowest RMSE: 0.13913141171548538
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
```

Describing the impact for Linear Regression

SGD vs OLS

One of the characteristics of SGD is sensitive to feature scaling. This has been displayed in our regression model wherein while comparing the MSE values between the OLS regression and SGD, we notice that MSE derived from SGD is similar to the MSE value derived from OLS.

Ridge vs Lasso vs Elastic Net

- Lasso can be best used when the dataset has features with poor predictive power. While ridge regression is useful for the grouping effect, in which colinear features can be selected together.
- in our model, we can identify that ridge regression performs the best consistently across the alpha values on the training set. Additionally, our model is performing better with lower alpha values, eg: 0.0001, 0.001 as compared to 0.1 and 1.

Mini Batch Gradient Desent

- While a lower batch would give us a better MSE, small batch sizes can be more susceptible to random fluctuations in the training data, while larger batch sizes are more resistant to these fluctuations but may converge more slowly. Hence, this trade-off needs to be kept in mind while deciding the batch size and learning rates. We can choose the ideal batch size through further experimentation of the model.

From the above graphs, we can observe that as the batch size increases, the convergence of the cost function does not change to a large extent. However, if we have a lower batch size the average MSE tends to lower, thereby indicating that a better performance of the model.

For this model, batch size 5 and learning rate 0.1, indicates the best performance measured in terms of MSE.

Part 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.

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

poly_degree = 2 # generating only 2nd dergree because 3rd degree would be a hug
poly_features = PolynomialFeatures(degree = poly_degree, include_bias=False)
X_train_poly = pd.DataFrame(poly_features.fit_transform(X_train))
X_test_poly  = pd.DataFrame(poly_features.transform(X_test))

performance_metrics_poly = []

i = 1
for train_index, test_index in kf.split(X_train_poly):

    print(f'Fold {i}')
    X_train_poly_kFold, X_test_poly_kFold = X_train_poly.iloc[train_index, :],
    y_train_kFold, y_test_kFold = y_train.iloc[train_index], y_train.iloc[test_

    # Fit the polynomial linear regression model on the training data
    poly_reg = LinearRegression()
    poly_reg.fit(X_train_poly_kFold, y_train_kFold)

    # Make predictions on the test data
    y_pred_kFold = poly_reg.predict(X_test_poly_kFold)

    # Calculate the Mean Squared Error for this iteration
    rmse = np.sqrt(mean_squared_error(y_test_kFold, y_pred_kFold))
    print('Root Mean Squared Error: ', rmse)

    # Store the performance metric
    performance_metrics_poly.append(rmse)
    i+=1
    print()

# Calculate the average performance metric across all iterations
average_performance_poly = np.mean(performance_metrics_poly)
print('average_performance of sklearn\'s Polynomial Regression in terms of RMSE
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
Fold 1
Root Mean Squared Error:  0.3970239571196309

Fold 2
Root Mean Squared Error:  0.40330550728639586

Fold 3
Root Mean Squared Error:  0.3972748808529861

Fold 4
Root Mean Squared Error:  0.3775186846514285

average_performance of sklearn's Polynomial Regression in terms of RMSE:  0.39
37807574776104
```

In [ ]:

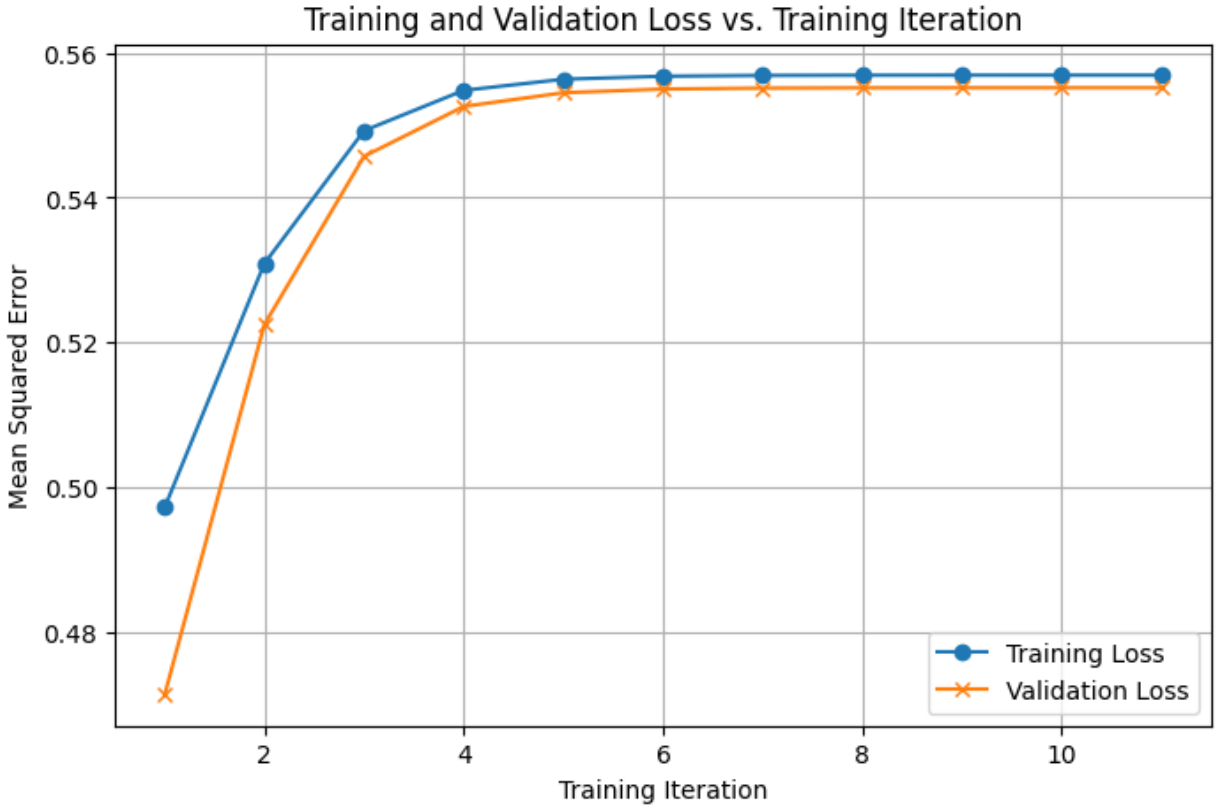
```
# Running SGD Regression and plotting the training and validation loss
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error

i = 1
for train_index, test_index in kf.split(X_train_poly):

    print(f'Fold {i}')
    X_train_poly_kFold, X_test_poly_kFold = X_train_poly.iloc[train_index, :],
    y_train_kFold, y_test_kFold = y_train.iloc[train_index], y_train.iloc[test_

    # Create the SGDRegressor model with appropriate hyperparameters
    for alpha in [0.00001, 0.0001, 0.001, 0.01, 0.1, 1]:

        sgd_model_poly = SGDRegressor(learning_rate='adaptive' , loss='squared_

        # Initialize lists to store training and validation loss
        training_loss_poly = []
        validation_loss_poly = []

        # Number of training iterations
        n_iterations_poly = 1000

        # Early stopping parameters
        early_stopping_rounds_poly = 10  # Number of iterations with no improve
        best_val_loss_poly = float('inf')  # Initialize the best validation los
        no_improvement_count_poly = 0  # Initialize the count of iterations wi

        for iteration in range(n_iterations_poly):
        # Fit the model for one iteration (one pass through the training data)
            sgd_model_poly.partial_fit(X_train_poly_kFold, y_train_kFold)

            # Predict on the training data
            y_train_pred_sgd_poly = sgd_model_poly.predict(X_train_poly_kFold)

            # Calculate training loss (Mean Squared Error) and append to the li
            train_loss_poly = mean_squared_error(y_train_kFold, y_train_pred_sg
```

```
            training_loss_poly.append(train_loss_poly)

            # Predict on the validation data
            y_val_pred_sgd_poly = sgd_model_poly.predict(X_test_poly_kFold)

            # Calculate validation loss (Mean Squared Error) and append to the
            val_loss_poly = mean_squared_error(y_test_kFold, y_val_pred_sgd_pol
            validation_loss_poly.append(val_loss_poly)

            # Check for early stopping
            if val_loss_poly < best_val_loss_poly:
                best_val_loss_poly = val_loss_poly
                no_improvement_count_poly = 0
            else:
                no_improvement_count_poly += 1

            if no_improvement_count_poly >= early_stopping_rounds_poly:
                print(f"Early stopping at iteration {iteration + 1} due to no i
                break

    # Plot training and validation loss as a function of training iteration
    plt.figure(figsize=(8, 5))
    plt.plot(range(1, iteration + 2), training_loss_poly, label='Training Loss
    plt.plot(range(1, iteration + 2), validation_loss_poly, label='Validation L
    plt.xlabel('Training Iteration')
    plt.ylabel('Mean Squared Error')
    plt.title(f'Training and Validation Loss vs. Training Iteration')
    plt.legend()
    plt.grid()
    plt.show()

    # Final model evaluation
    y_pred_sgd_poly = sgd_model_poly.predict(X_test_poly_kFold)
    final_rmse_sgd_poly = mean_squared_error(y_test_kFold, y_pred_sgd_poly)
    print(f"Final Mean Squared Error (SGD) for alpha: {alpha}: ", final_rmse_sg
```

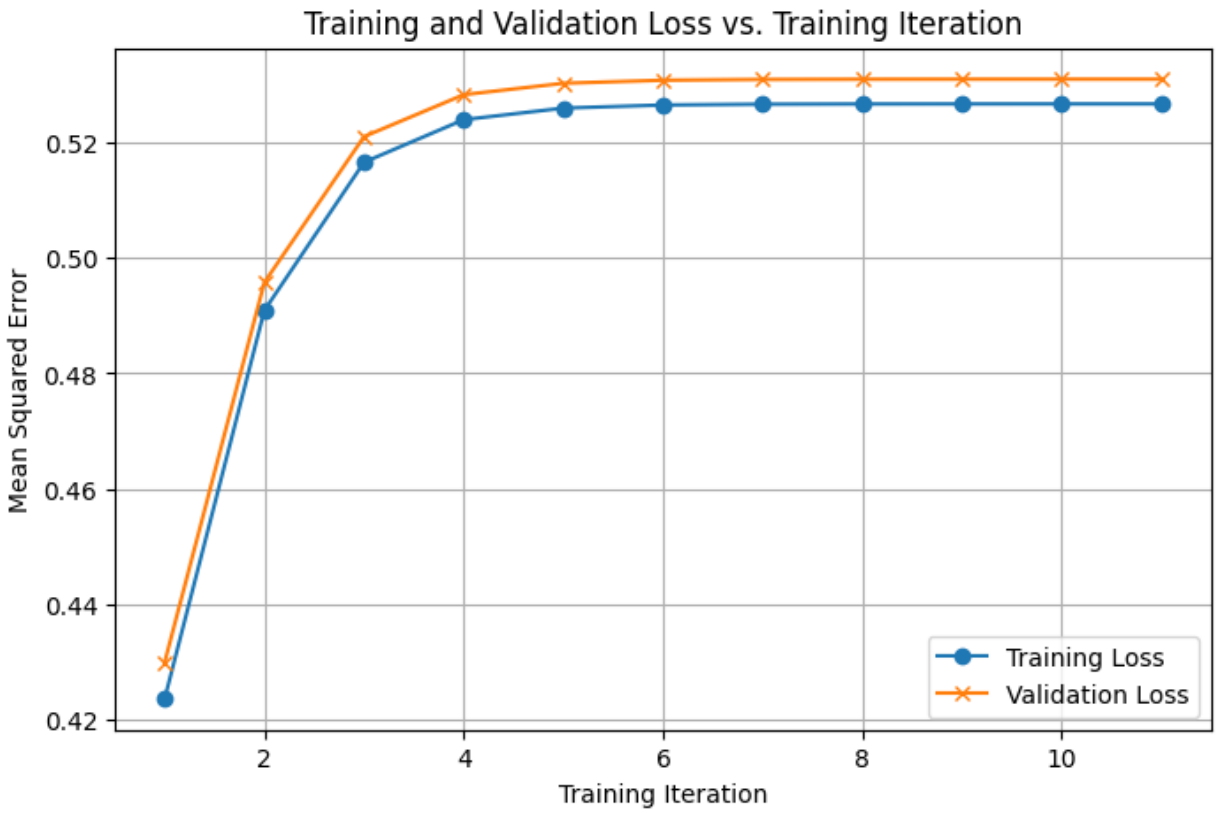
```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecation
Warning: `should_run_async` will not call `transform_cell` automatically in th
e future. Please pass the result to `transformed_cell` argument and any except
ion that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
17 and above.
  and should_run_async(code)
Fold 1
Early stopping at iteration 64 due to no improvement in validation loss.
Early stopping at iteration 65 due to no improvement in validation loss.
Early stopping at iteration 67 due to no improvement in validation loss.
Early stopping at iteration 49 due to no improvement in validation loss.
Early stopping at iteration 19 due to no improvement in validation loss.
Early stopping at iteration 11 due to no improvement in validation loss.
```


```
Final Mean Squared Error (SGD) for alpha: 1:  0.4203389681500594
Fold 1
Early stopping at iteration 198 due to no improvement in validation loss.
Early stopping at iteration 325 due to no improvement in validation loss.
Early stopping at iteration 487 due to no improvement in validation loss.
Early stopping at iteration 111 due to no improvement in validation loss.
Early stopping at iteration 11 due to no improvement in validation loss.
```

```
Final Mean Squared Error (SGD) for alpha: 1:  0.42569473648884254
Fold 1
Early stopping at iteration 350 due to no improvement in validation loss.
Early stopping at iteration 428 due to no improvement in validation loss.
Early stopping at iteration 187 due to no improvement in validation loss.
Early stopping at iteration 28 due to no improvement in validation loss.
Early stopping at iteration 17 due to no improvement in validation loss.
Early stopping at iteration 11 due to no improvement in validation loss.
```




```
Final Mean Squared Error (SGD) for alpha: 1:  0.5551610079684546
Fold 1
Early stopping at iteration 169 due to no improvement in validation loss.
Early stopping at iteration 196 due to no improvement in validation loss.
Early stopping at iteration 563 due to no improvement in validation loss.
Early stopping at iteration 519 due to no improvement in validation loss.
Early stopping at iteration 27 due to no improvement in validation loss.
Early stopping at iteration 11 due to no improvement in validation loss.
```




```
Final Mean Squared Error (SGD) for alpha: 1:  0.5310473081855498
```

In [ ]:

```
## poly features normal equation

# As asked in the question, we would be running a four-fold valiadation, hence
k = 4

# Storing the theta values for each fold
theta_values = []

# Initialize KFold cross-validator
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Initialize an empty list to store the MSE values for each fold
mse_values = []

# Perform k-fold cross-validation
for train_index, test_index in kf.split(X_train_poly):
    X_train_poly_kFold, X_test_poly_kFold = X_train_poly.iloc[train_index,:], X
    y_train_kFold, y_test_kFold = y_train.iloc[train_index], y_train.iloc[test_

    # Calculating the coefficients using the normal equation
    X_transpose = np.transpose(X_train_fold)
    X_transpose_X = np.dot(X_transpose, X_train_fold)
    X_transpose_X_inv = np.linalg.inv(X_transpose_X)
    X_transpose_y = np.dot(X_transpose, y_train_fold)
    theta = np.dot(X_transpose_X_inv, X_transpose_y)

    # Append the theta values for this fold to the list
    theta_values.append(theta)

    # Make predictions using the calculated theta values
    y_val_pred = np.dot(X_val_fold, theta)
```

```
    # Calculate and append the MSE for this fold
    mse_fold = mean_squared_error(y_val_fold, y_val_pred)
    mse_values.append(mse_fold)

# Calculate the average coefficients (theta) across all folds
average_theta = np.mean(theta_values, axis=0)

# Calculate the average MSE across all folds
average_mse = np.mean(mse_values)

print("Average Coefficients (theta) across all folds:", average_theta)
print("Average MSE across all folds:",average_mse)
```

```
Average Coefficients (theta) across all folds: [ 2.81349577  0.51597649  0.462
27272  0.14790629  0.38245103  0.15431259
 -0.0769383   0.26294941 -0.21479337  3.37261408]
Average MSE across all folds: 0.14397315332559257
```

In [ ]:
```
X_train_poly_array = X_train_poly.values
y_train_poly_array = y_train.values
```

In [ ]:
```
print(X_train_poly_array.shape)
```

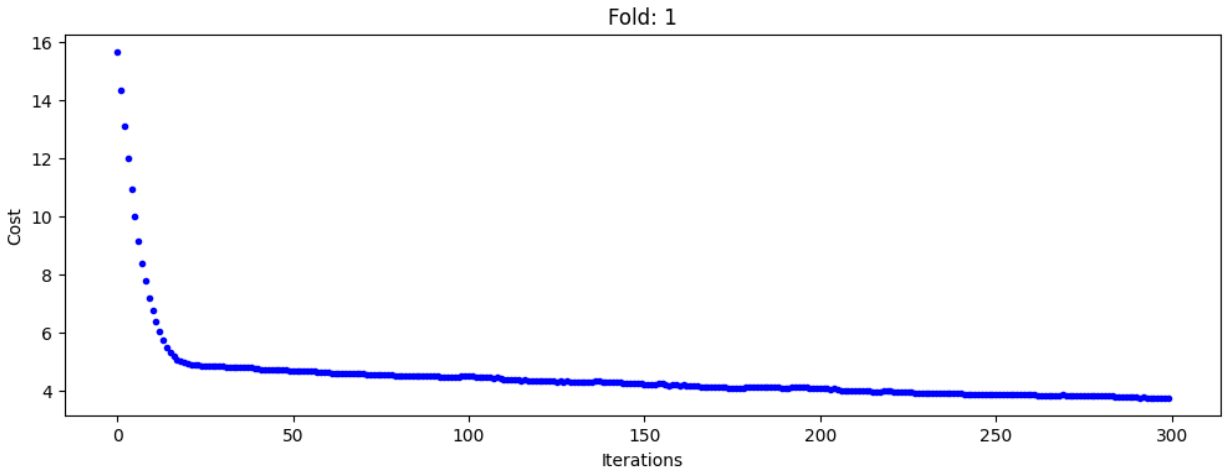
```
(1366, 54)
```

In [ ]:
```
gradient_clip_threshold = 1.0  # Adjust this threshold as needed

batch_sizes = [5,10,20,50,100]
for batch_size in batch_sizes:
  print("*********************************************************")
  print(f"The following graphs are for batch size: {batch_size}")
  print("*********************************************************")
  i = 1
  for train_index, val_index in kf.split(X_train_poly_array):
      X_train_fold, X_val_fold = X_train_poly_array[train_index], X_train_poly_
      y_train_fold, y_val_fold = y_train_poly_array[train_index], y_train_poly_

      # Add bias column to fold data
      X_train_fold = np.c_[np.ones((len(X_train_fold), 1)), X_train_fold]
      X_val_fold = np.c_[np.ones((len(X_val_fold), 1)), X_val_fold]

      lr = 0.01
      n_iter = 300
      batch_size = 20
      theta = np.random.randn(55, 1)

      # Train the model using minibatch gradient descent with gradient clipping
      theta, cost_history = minibatch_gradient_descent(X_train_fold, y_train_fo

      print('Predictions for fold:', i)
      print('Intercept value:', theta[0, 0], '\nWeight values:', theta[1:])

      # Use the validation data and mean square error to evaluate performance
      N = len(y_val_fold)
      y_hat = np.dot(X_val_fold, theta)
      mse_val = (1 / N) * (np.sum(np.square(y_hat - y_val_fold)))
      print('Mean Square Error:', round(mse_val, 3), '\n')

      fig, ax = plt.subplots(figsize=(12, 4))
      plt.title('Fold: ' + str(i))
```

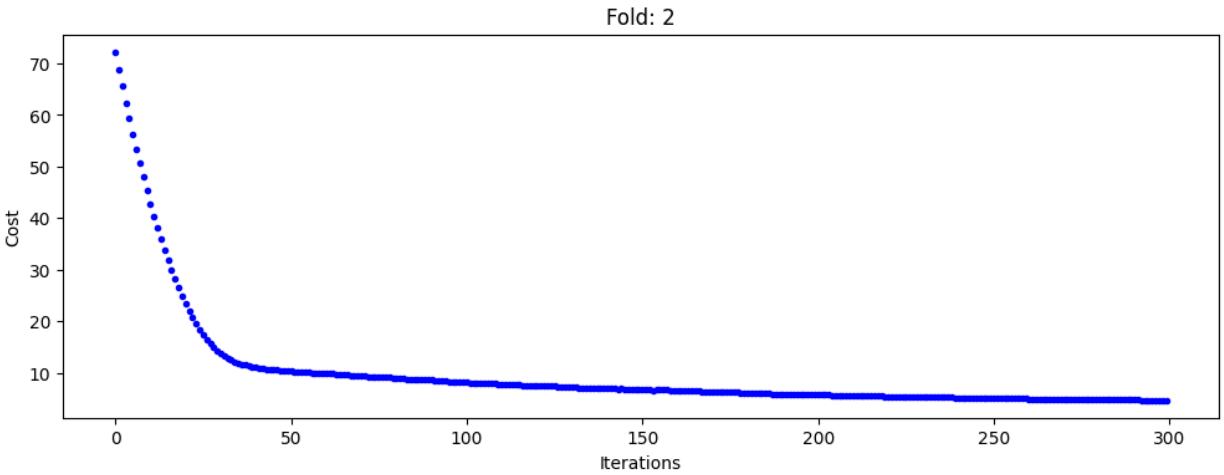
```
        ax.set_ylabel('Cost')
        ax.set_xlabel('Iterations')
        _ = ax.plot(range(n_iter), cost_history, 'b.')
        plt.show()

        i += 1
```

```
**********************************************************
The following graphs are for batch size: 5
**********************************************************
Predictions for fold: 1
Intercept value: 0.7106297637200157
Weight values: [[ 1.03642164]
 [-0.43619187]
 [-1.86932608]
 [-0.07725265]
 [ 1.48444811]
 [ 0.65240678]
 [ 2.08530341]
 [ 0.08170197]
 [ 0.5011207 ]
 [ 0.52013845]
 [ 1.05828198]
 [-0.13033013]
 [-0.0584638 ]
 [ 0.60500365]
 [ 0.4157783 ]
 [-0.40122047]
 [-2.45641363]
 [-1.04634113]
 [ 0.68658062]
 [ 0.06640946]
 [ 0.22221703]
 [ 1.25392118]
 [ 0.76669673]
 [ 0.28582345]
 [ 0.82236441]
 [ 1.92815759]
 [ 1.02221049]
 [-0.97982733]
 [-0.24062405]
 [ 0.62480025]
 [-1.67462943]
 [ 0.93226774]
 [ 0.57443741]
 [-1.5750473 ]
 [-0.69784194]
 [-0.77103908]
 [ 0.47595418]
 [ 0.26374049]
 [ 0.39303505]
 [ 0.81473281]
 [-0.51020976]
 [ 0.76230665]
 [-0.5684855 ]
 [-0.42856271]
 [ 1.17717965]
 [ 0.14452839]
 [-1.34773399]
 [ 0.1446961 ]
 [ 0.9879596 ]
 [ 2.08086743]
 [-0.38221718]
 [ 0.20789634]
 [-0.11316734]
 [ 2.22670272]]
```

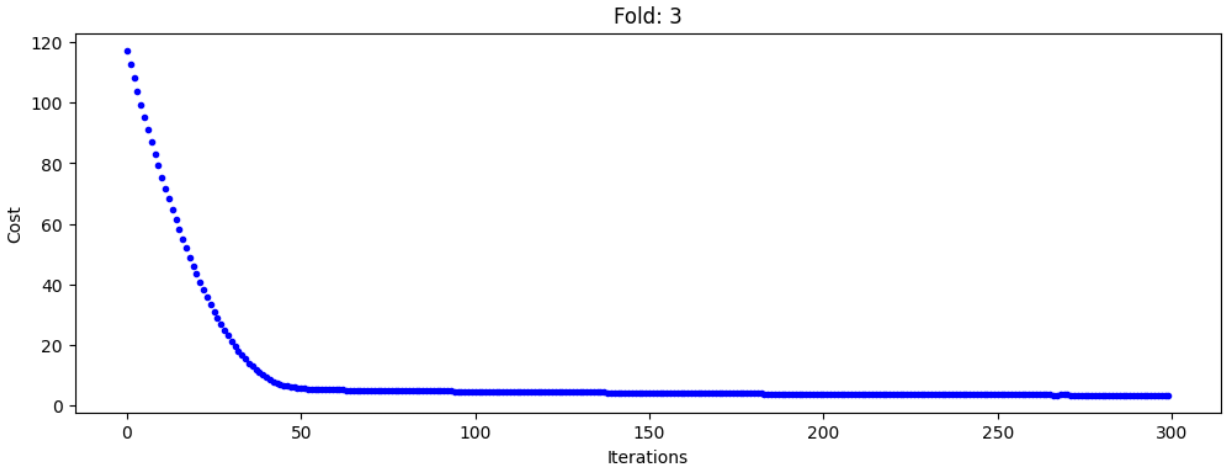
```
Mean Square Error: 1355.814
```



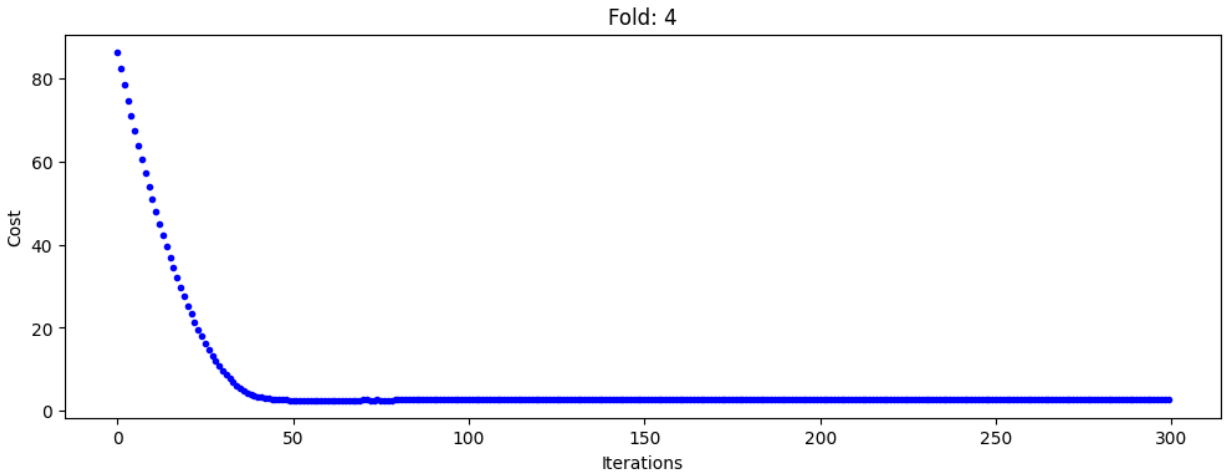
```
Predictions for fold: 2
Intercept value: -0.2767884077783834
Weight values: [[ 1.32020319]
 [ 0.11680518]
 [-0.51933499]
 [-0.54971832]
 [ 0.37694304]
 [-0.18152621]
 [ 0.11848367]
 [ 0.46711316]
 [-0.00369894]
 [ 0.05508408]
 [-0.94079334]
 [-0.05955971]
 [-1.56707294]
 [-1.98383625]
 [ 1.05404693]
 [ 0.4755514 ]
 [ 2.20401834]
 [-0.34469288]
 [ 0.53277089]
 [ 0.08171399]
 [ 1.60789924]
 [-0.08781109]
 [ 0.69320073]
 [ 0.27427277]
 [ 0.71403478]
 [ 1.65257443]
 [ 0.93745988]
 [ 1.56581532]
 [ 0.010179  ]
 [ 1.12410206]
 [ 0.43301491]
 [-0.09744775]
 [ 0.302972  ]
 [-0.33520048]
 [ 0.62008977]
 [-0.66702094]
 [ 0.51680394]
 [-0.79739802]
 [ 0.80492229]
 [ 0.43566608]
 [ 2.34531911]
 [ 0.77179187]
 [ 1.95348982]
 [ 0.16268765]
 [-0.89376648]
 [ 0.93028587]
 [ 1.17776035]
 [ 0.18628571]
 [ 1.3809688 ]
 [-1.30910686]
 [-0.05370988]
 [-1.12549868]
 [-0.64684252]
 [ 0.15648837]]
Mean Square Error: 1430.033
```


Fold: 2
Cost
Iterations
70
60
50
40
30
20
10
0
50
100
150
200
250
300

```
Predictions for fold: 3
Intercept value: 0.09518469145597867
Weight values: [[ 2.50264567]
 [ 1.32250125]
 [-0.25815415]
 [ 1.78322393]
 [ 0.65909025]
 [-0.47857758]
 [ 0.57230007]
 [ 2.54672135]
 [ 0.93871384]
 [-0.98521246]
 [-0.22427405]
 [-1.03864465]
 [ 1.01248213]
 [ 2.06384371]
 [ 0.04336646]
 [-1.39142556]
 [ 2.37050062]
 [-0.35060586]
 [ 0.34712264]
 [ 0.38463119]
 [ 0.03303086]
 [ 0.78319936]
 [ 0.05170823]
 [ 0.52717012]
 [ 0.44449797]
 [-1.26620421]
 [ 1.05381819]
 [-0.30343442]
 [ 0.30683554]
 [-0.0772615 ]
 [ 1.28712528]
 [ 0.76268447]
 [-0.44325238]
 [-1.37060416]
 [-2.24960694]
 [ 0.63350605]
 [ 0.42586711]
 [-0.39923553]
 [ 0.44041002]
 [ 0.21437991]
 [-0.73260527]
 [ 1.68882392]
 [ 0.14644323]
 [-0.54850621]
 [-0.44425301]
 [ 0.10928449]
 [-0.1813896 ]
 [ 0.0101565 ]
 [-0.75934561]
 [-0.32827811]
 [ 0.49783671]
 [-1.21767876]
 [ 0.66556891]
 [ 1.10084003]]
Mean Square Error: 1080.826
```

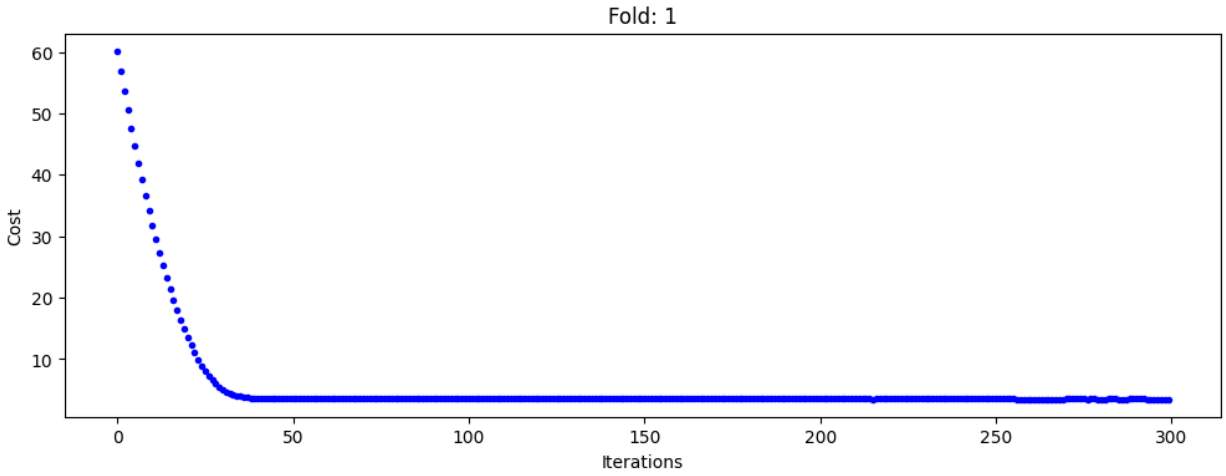

Fold: 3
Cost
Iterations
0
20
40
60
80
100
120
0
50
100
150
200
250
300

```
Predictions for fold: 4
Intercept value: 0.07548728506640338
Weight values: [[ 0.44403547]
 [ 1.98689059]
 [ 1.1296457 ]
 [-0.39797767]
 [ 0.35361871]
 [ 0.33503225]
 [ 1.28857353]
 [ 1.10371164]
 [-0.2278066 ]
 [-0.26685396]
 [-0.24252061]
 [ 0.68700186]
 [-0.22384637]
 [-0.21658752]
 [-0.40356104]
 [ 0.84918376]
 [-1.44907278]
 [-0.03902822]
 [-0.01172976]
 [-0.67220803]
 [ 1.82901019]
 [-0.60140505]
 [-1.32259048]
 [-1.3963325 ]
 [-1.0063047 ]
 [ 2.13130222]
 [ 0.55974927]
 [-0.1553718 ]
 [-0.83455145]
 [ 1.26049261]
 [-0.19979704]
 [ 1.42870224]
 [-0.85626036]
 [ 0.3139197 ]
 [ 1.26497513]
 [-1.19387168]
 [-0.32677351]
 [ 0.09457201]
 [ 0.26153091]
 [-0.65889673]
 [ 1.24809909]
 [ 0.87105542]
 [-0.47967823]
 [ 2.00852247]
 [-0.82467083]
 [ 2.01773342]
 [ 2.4815877 ]
 [ 0.8745251 ]
 [-0.0955946 ]
 [ 0.04944195]
 [-0.52510463]
 [ 0.85368913]
 [-0.29979745]
 [ 0.27376818]]
Mean Square Error: 962.541
```

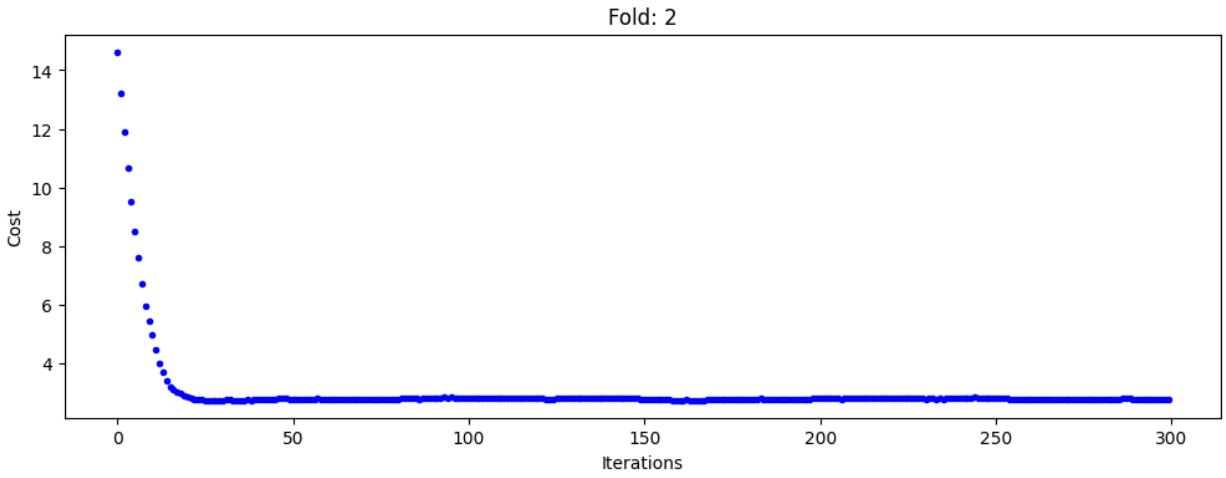

Fold: 4
Cost
Iterations
0
20
40
60
80
0
50
100
150
200
250
300

```
**********************************************************
The following graphs are for batch size: 10
**********************************************************
Predictions for fold: 1
Intercept value: -0.4205243623372434
Weight values: [[ 0.71169089]
 [ 1.16566975]
 [ 0.55815474]
 [-0.36642392]
 [-0.12771041]
 [ 0.20233521]
 [ 0.41962705]
 [ 2.00455967]
 [ 0.20843706]
 [-0.58768122]
 [ 1.66951994]
 [ 0.0572358 ]
 [-0.35376867]
 [-0.76790754]
 [ 1.81688386]
 [ 0.52886594]
 [ 1.47310993]
 [-1.07446109]
 [ 0.33335308]
 [ 1.2136364 ]
 [ 0.98749629]
 [-0.23386874]
 [ 0.53561602]
 [ 0.7334413 ]
 [ 1.47873162]
 [-0.82038109]
 [-0.17834915]
 [ 0.17233752]
 [ 1.56129388]
 [-0.55581593]
 [-0.77986893]
 [ 1.48296762]
 [-0.55158443]
 [-0.41859372]
 [ 0.29838393]
 [ 1.02069303]
 [-0.43849674]
 [-0.14646054]
 [-0.27826656]
 [ 0.24925915]
 [-1.22462939]
 [-0.05579764]
 [-1.02867721]
 [ 0.51320568]
 [-0.20172542]
 [-1.46751145]
 [ 0.82696324]
 [ 0.59676074]
 [ 2.34485349]
 [-0.66019396]
 [ 1.2629335 ]
 [-1.20809043]
 [-1.04372281]
 [-0.09320634]]
```

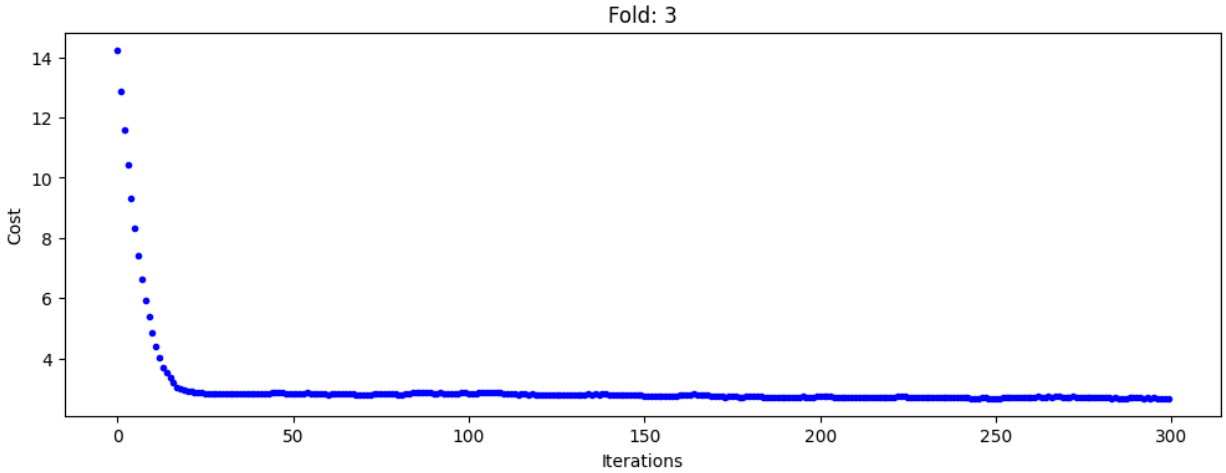
```
Mean Square Error: 1280.286
```



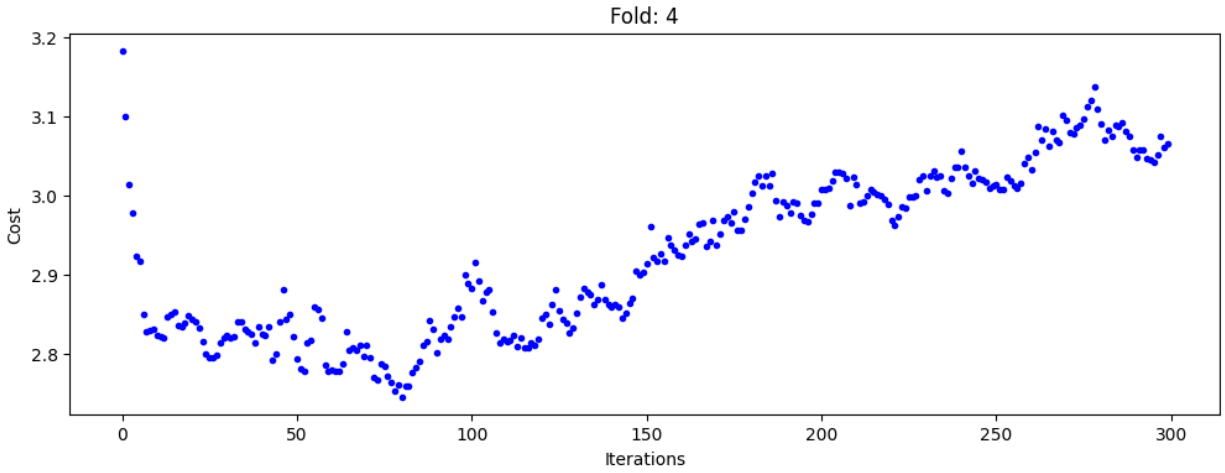
```
Predictions for fold: 2
Intercept value: 1.5692795351140911
Weight values: [[ 0.28542919]
 [ 1.61530043]
 [-0.76822216]
 [ 0.59306342]
 [ 0.31034043]
 [ 0.22103154]
 [ 0.74491338]
 [ 0.98081409]
 [ 0.2530348 ]
 [-0.02791956]
 [ 1.65433263]
 [ 0.88634981]
 [-0.45533617]
 [ 0.01080171]
 [-0.59370864]
 [ 0.20825286]
 [-1.1442945 ]
 [ 0.78535951]
 [ 2.85469047]
 [ 1.29040031]
 [ 0.2631523 ]
 [ 0.06178008]
 [-1.95711081]
 [-0.13119372]
 [ 0.22624332]
 [-1.90166859]
 [-2.3135474 ]
 [ 0.93044784]
 [ 1.38008574]
 [ 2.15991973]
 [ 0.0679746 ]
 [-1.33634741]
 [-1.18120496]
 [ 0.50969921]
 [ 0.51543885]
 [-1.54622198]
 [-0.66896687]
 [ 1.27663143]
 [-0.90144976]
 [ 1.8142791 ]
 [-0.82980081]
 [-0.99237959]
 [ 0.58338532]
 [ 1.21120728]
 [-0.44445484]
 [ 1.22717866]
 [-0.89824846]
 [ 2.16403151]
 [-1.69043393]
 [ 0.67514798]
 [ 0.19467255]
 [ 0.98971377]
 [ 1.20376495]
 [ 0.33821226]]
Mean Square Error: 905.76
```


Fold: 2
Cost
Iterations
14
12
10
8
6
4
0
50
100
150
200
250
300

```
Predictions for fold: 3
Intercept value: 1.5821878436034722
Weight values: [[ 0.34291418]
 [ 1.06245559]
 [ 1.87770507]
 [ 1.06317128]
 [ 1.54917768]
 [-1.24542399]
 [-0.04066443]
 [ 1.07044329]
 [ 0.0222707 ]
 [ 0.96568417]
 [ 1.70197163]
 [ 0.13848516]
 [ 0.18192859]
 [-2.16652738]
 [ 0.41941289]
 [ 0.81179212]
 [ 1.62621688]
 [ 0.18635529]
 [-0.21658538]
 [-0.76357563]
 [ 1.4981727 ]
 [-0.39986508]
 [ 0.17774982]
 [-1.25598729]
 [-0.0361113 ]
 [ 1.11279632]
 [-1.05549637]
 [ 0.60228543]
 [-0.75011185]
 [ 0.95186228]
 [ 0.81326298]
 [-0.56891546]
 [ 0.52415504]
 [-0.08287161]
 [ 2.14245839]
 [-1.18395795]
 [-1.56000233]
 [-0.18669922]
 [-0.29727089]
 [-0.39023835]
 [ 0.22524593]
 [-0.09855143]
 [ 0.37121021]
 [ 0.86740372]
 [ 0.29387982]
 [ 0.37082224]
 [ 0.64088479]
 [-1.49834281]
 [ 0.63153026]
 [-0.40051556]
 [-0.14762202]
 [ 0.17335301]
 [-0.39541162]
 [-1.61257776]]
Mean Square Error: 850.248
```

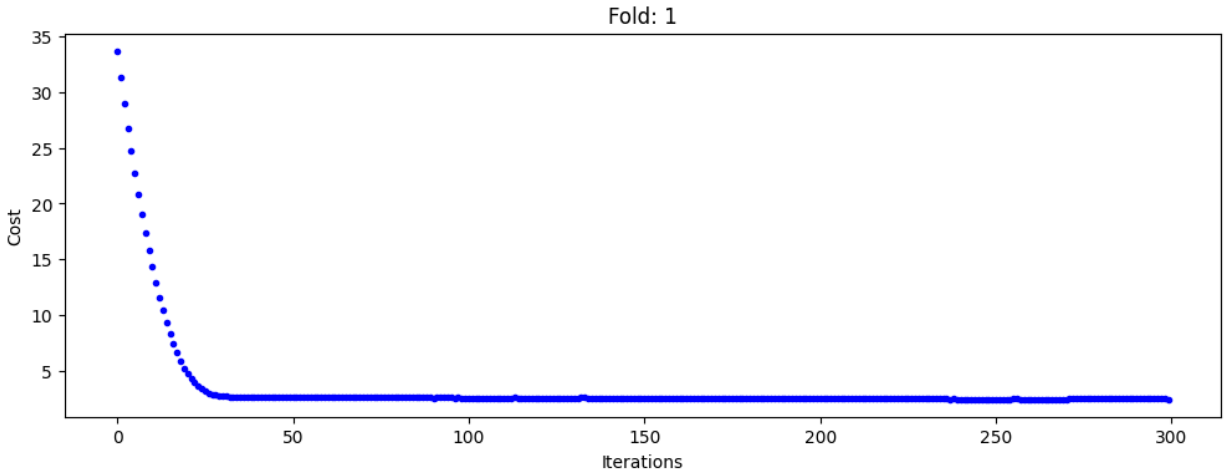

Fold: 3
Cost
Iterations
14
12
10
8
6
4
0
50
100
150
200
250
300

```
Predictions for fold: 4
Intercept value: -0.40469455755135353
Weight values: [[-0.74013936]
 [ 1.36441736]
 [ 0.10948438]
 [ 0.38616025]
 [-0.09727638]
 [ 1.1928493 ]
 [ 0.08159767]
 [ 0.44901051]
 [ 0.2784204 ]
 [ 1.4480537 ]
 [ 1.16327618]
 [ 0.44249658]
 [-0.20300147]
 [-0.18027938]
 [ 1.56765933]
 [-0.25526558]
 [ 0.20142282]
 [ 1.07923983]
 [-0.52040845]
 [-0.2681398 ]
 [-1.08146579]
 [-0.49186619]
 [ 0.15875485]
 [-1.10505408]
 [-1.2528053 ]
 [-0.44359509]
 [ 1.25630803]
 [ 0.03401412]
 [ 0.86144766]
 [ 1.52577324]
 [ 1.03453961]
 [-1.30493353]
 [-1.0307028 ]
 [ 0.44522036]
 [-0.40966087]
 [-0.46382963]
 [ 0.89218346]
 [ 1.54696974]
 [ 0.10376848]
 [ 0.23896951]
 [ 0.51703484]
 [ 2.20814145]
 [ 0.09490044]
 [-0.2265325 ]
 [-0.18824712]
 [ 1.11099197]
 [ 1.14484295]
 [-0.7142612 ]
 [ 0.71251409]
 [ 0.07870823]
 [ 0.44211225]
 [-0.04376485]
 [-0.43041812]
 [ 0.77039103]]
Mean Square Error: 1070.982
```

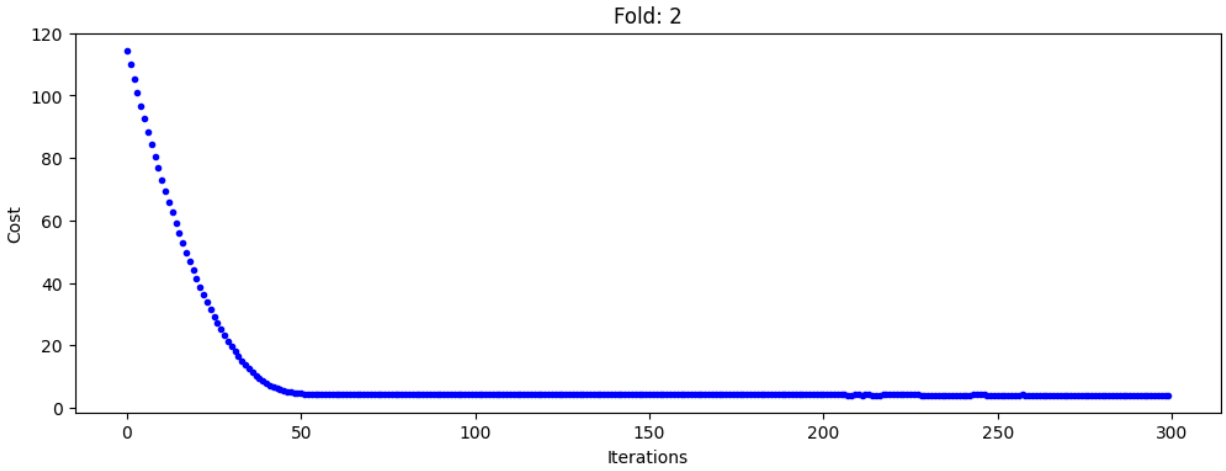

Fold: 4
Cost
Iterations
3.2
3.1
3.0
2.9
2.8
0
50
100
150
200
250
300

```
**********************************************************
The following graphs are for batch size: 20
**********************************************************
Predictions for fold: 1
Intercept value: 1.8591245017436862
Weight values: [[-0.36174725]
 [ 1.61026019]
 [ 0.74869206]
 [ 0.38040119]
 [ 1.12003652]
 [-0.56520051]
 [ 0.39297885]
 [ 0.92312082]
 [ 0.2321482 ]
 [-0.35120228]
 [ 0.15729136]
 [-0.1774318 ]
 [ 0.36811435]
 [ 1.6740581 ]
 [ 0.25057043]
 [-0.65358935]
 [ 0.46815398]
 [ 0.39838109]
 [ 1.41403025]
 [ 1.27439159]
 [-0.79790707]
 [-0.49172905]
 [-0.51316591]
 [-1.96943032]
 [-0.32494726]
 [ 0.83608107]
 [ 0.04848001]
 [-1.45371949]
 [-0.57140996]
 [-0.02053339]
 [ 1.05213618]
 [-1.11263754]
 [-1.06228749]
 [ 0.68259818]
 [ 0.61325114]
 [-0.49501032]
 [ 0.66463358]
 [-0.61524087]
 [ 1.14351766]
 [-0.98171777]
 [-0.58917451]
 [ 1.16385344]
 [-1.73326451]
 [ 0.80843065]
 [ 1.10805147]
 [-0.26182729]
 [ 0.49082146]
 [-0.67839658]
 [ 1.40311395]
 [ 0.5539305 ]
 [-0.76508289]
 [-1.06973794]
 [ 1.74228568]
 [ 0.29851191]]
```

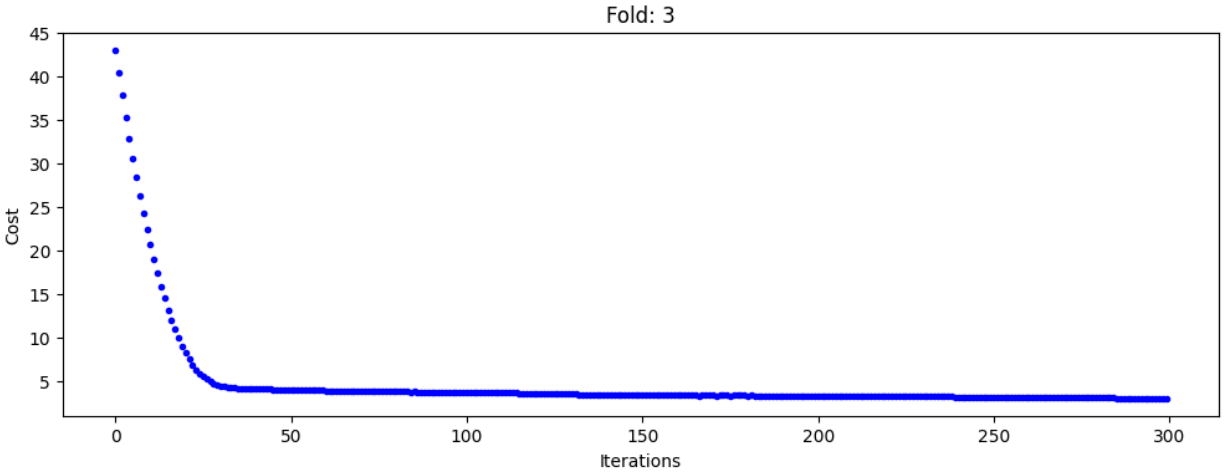
```
Mean Square Error: 924.798
```



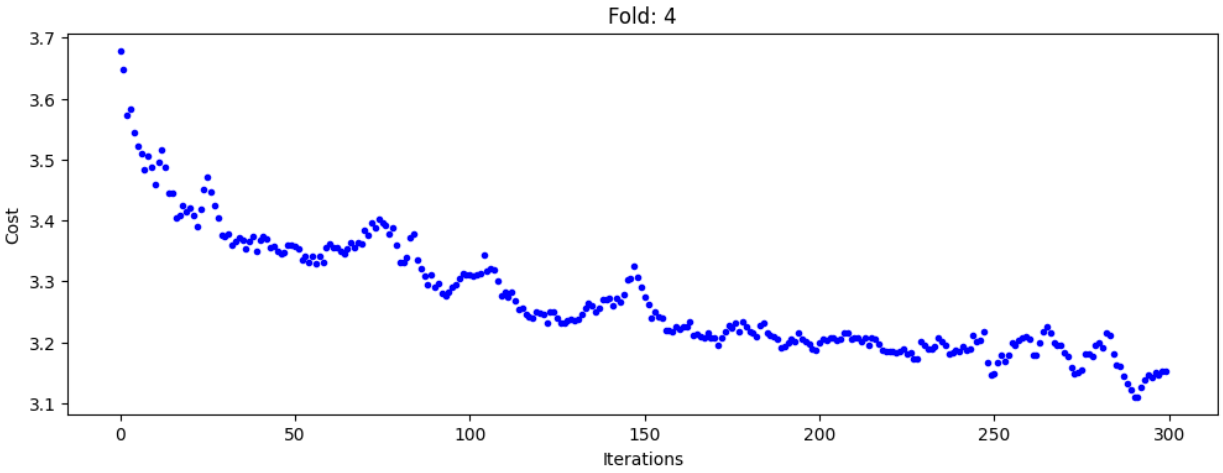
```
Predictions for fold: 2
Intercept value: -0.9838182425698019
Weight values: [[-0.37708456]
 [ 0.24854547]
 [ 0.30911399]
 [ 1.15337149]
 [ 0.52375223]
 [ 0.82686654]
 [-0.87760664]
 [ 1.11603813]
 [-1.03133939]
 [ 0.0430121 ]
 [ 0.79480704]
 [ 0.36929532]
 [-0.03972187]
 [-0.24870545]
 [ 1.08332924]
 [ 1.0183812 ]
 [ 2.15317305]
 [ 0.57556147]
 [-1.20714634]
 [ 1.48700693]
 [-0.1514047 ]
 [-0.25011261]
 [ 2.11237977]
 [ 0.29812336]
 [-0.45905708]
 [-0.01681599]
 [-1.9964196 ]
 [ 0.91673088]
 [-1.07231293]
 [ 0.20571333]
 [ 1.3051553 ]
 [ 0.69999237]
 [ 0.71516475]
 [ 1.26077753]
 [-0.34612073]
 [ 1.59081676]
 [-0.37477677]
 [-1.23092412]
 [ 0.78948794]
 [-0.33255911]
 [ 0.9617312 ]
 [ 1.43211646]
 [ 0.44205142]
 [ 1.69729457]
 [-0.35026458]
 [ 0.508365  ]
 [ 0.01893888]
 [ 0.31815662]
 [-0.17040269]
 [ 0.66130366]
 [ 0.10878716]
 [-0.40331409]
 [-0.24789611]
 [ 0.48799972]]
Mean Square Error: 1239.124
```


Fold: 2
Cost
Iterations
0
20
40
60
80
100
120
0
50
100
150
200
250
300

```
Predictions for fold: 3
Intercept value: 1.1550366912970171
Weight values: [[-0.9846084 ]
 [ 0.60572956]
 [ 1.95495655]
 [-0.31245123]
 [ 0.65459855]
 [-0.51470689]
 [ 0.96424537]
 [-0.68386578]
 [ 2.23928455]
 [ 0.12132107]
 [-1.85642756]
 [ 1.83263705]
 [-0.1060339 ]
 [-0.11746374]
 [-1.61123803]
 [ 0.28920109]
 [ 0.12914271]
 [-0.26439448]
 [ 1.78228126]
 [-0.63105518]
 [ 0.88649971]
 [ 1.08733242]
 [ 0.59019278]
 [-1.17188479]
 [ 1.39725143]
 [-1.16444096]
 [ 1.08736759]
 [-0.20619538]
 [-1.46907814]
 [-0.686586  ]
 [ 0.22964365]
 [ 1.23524304]
 [ 0.7742824 ]
 [-0.35426199]
 [-1.10636079]
 [ 0.28858396]
 [ 1.16786391]
 [ 0.68494368]
 [ 0.60141171]
 [-1.12564157]
 [ 0.64397503]
 [ 0.11886289]
 [-1.02919015]
 [ 2.699268  ]
 [ 0.91406711]
 [-0.87496412]
 [ 0.52863027]
 [-0.32171307]
 [-0.74649459]
 [ 0.41099243]
 [ 0.51794804]
 [ 0.57333624]
 [ 0.59842475]
 [-0.32270417]]
Mean Square Error: 965.691
```

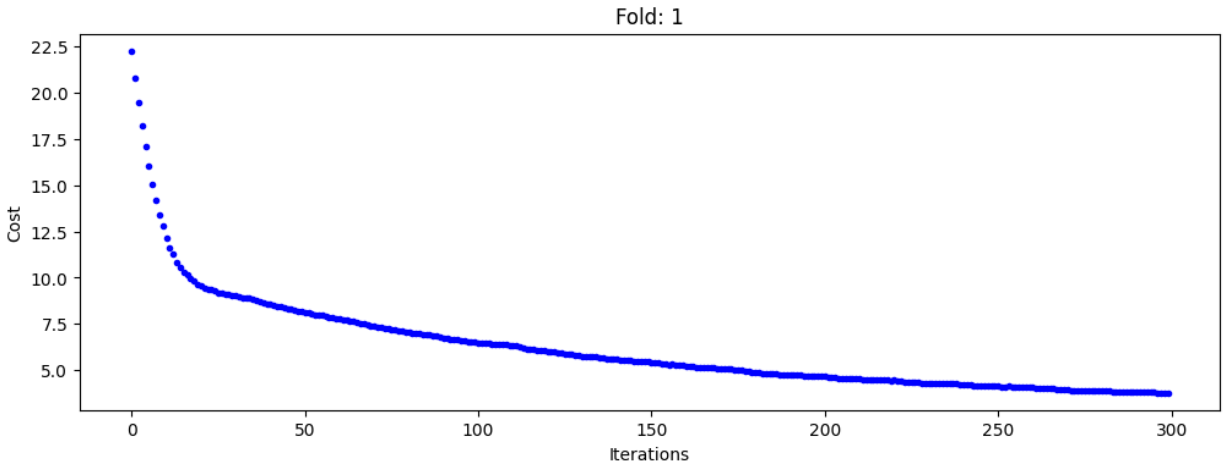

Fold: 3
Cost
Iterations
45
40
35
30
25
20
15
10
5
0
50
100
150
200
250
300

```
Predictions for fold: 4
Intercept value: 0.8859640835050525
Weight values: [[ 0.71986211]
 [ 0.12675169]
 [-0.50322505]
 [-0.15925345]
 [ 1.36573457]
 [ 1.27990835]
 [-0.87694686]
 [ 0.23702168]
 [-0.3692539 ]
 [-1.67916531]
 [ 0.3522301 ]
 [ 0.5309795 ]
 [ 2.51957972]
 [ 1.25839755]
 [ 1.71773544]
 [ 0.31223286]
 [-0.68497368]
 [ 0.81039277]
 [-0.8841541 ]
 [ 0.30833254]
 [ 1.26212171]
 [ 0.63060064]
 [-0.61937431]
 [-0.1113899 ]
 [-0.12760842]
 [ 0.80792764]
 [-0.56854244]
 [ 0.55991562]
 [ 0.46628947]
 [ 0.96326135]
 [-1.91123181]
 [ 0.26598005]
 [ 0.34269295]
 [ 1.33325013]
 [ 0.20567968]
 [-0.7392313 ]
 [ 0.22038738]
 [ 1.84680362]
 [-0.16358726]
 [-0.63463069]
 [-0.03380912]
 [ 0.74899748]
 [-0.5967266 ]
 [-0.00648246]
 [ 0.65420406]
 [-0.72168968]
 [ 0.44462867]
 [ 0.36622288]
 [ 1.51622757]
 [-0.23856738]
 [ 0.69338936]
 [ 0.06512342]
 [-0.33561943]
 [-0.51777137]]
Mean Square Error: 1100.566
```

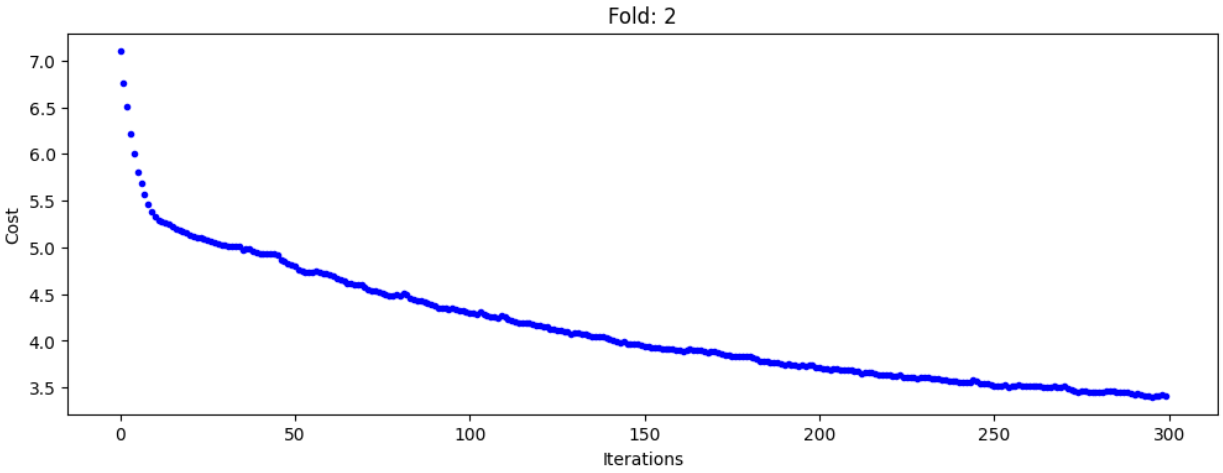

Fold: 4
Cost
Iterations
3.7
3.6
3.5
3.4
3.3
3.2
3.1
0
50
100
150
200
250
300

```
**********************************************************
The following graphs are for batch size: 50
**********************************************************
Predictions for fold: 1
Intercept value: 1.687087898576935
Weight values: [[ 1.22434535]
 [-0.76052266]
 [ 0.07186853]
 [ 0.04775907]
 [-0.7809041 ]
 [ 0.08696536]
 [-0.85962294]
 [-0.58395492]
 [-0.08300046]
 [ 0.73154957]
 [-0.77574438]
 [-0.48907364]
 [ 0.92480103]
 [ 0.4017231 ]
 [-0.44249889]
 [ 0.88066249]
 [ 1.75827548]
 [ 0.43188275]
 [ 0.48427497]
 [ 0.42132619]
 [ 0.67822378]
 [ 0.66092345]
 [ 0.40690664]
 [-1.40983442]
 [ 0.17749989]
 [-0.2133246 ]
 [ 1.8531101 ]
 [ 0.5753246 ]
 [ 0.19601384]
 [ 0.02885466]
 [ 0.41266115]
 [ 0.13907772]
 [-0.73244977]
 [-0.49864756]
 [ 0.6123714 ]
 [ 0.19860664]
 [ 2.05441496]
 [-2.52728648]
 [ 2.02857668]
 [-0.10598584]
 [ 1.00134225]
 [ 0.27722254]
 [ 0.08913205]
 [-0.54420786]
 [ 0.26847448]
 [ 0.04957733]
 [ 0.02370974]
 [ 1.03797922]
 [-0.67574696]
 [ 2.32503462]
 [-0.22655559]
 [-0.1307441 ]
 [ 0.31421676]
 [ 0.12709854]]
```

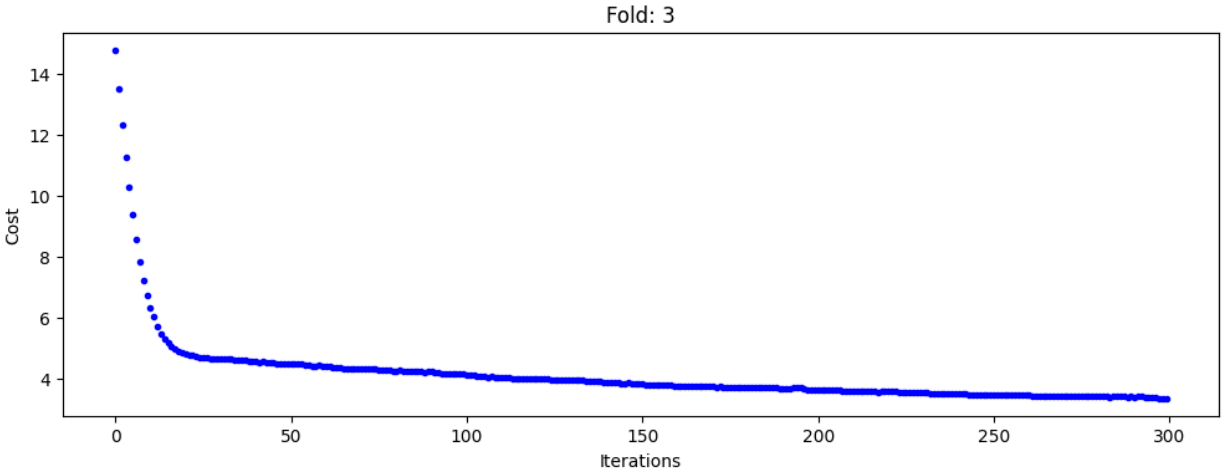
```
Mean Square Error: 1439.596
```



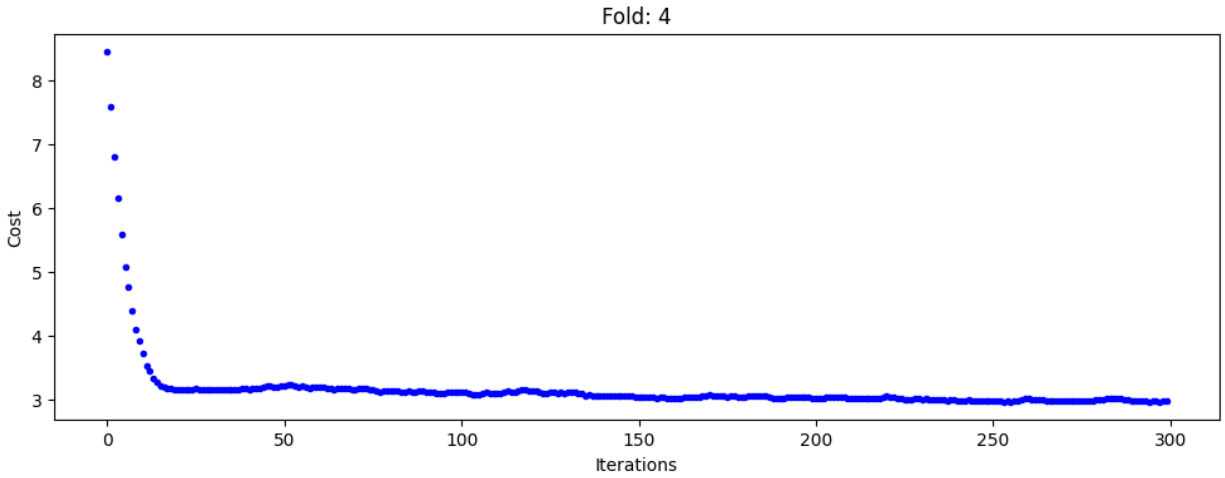
```
Predictions for fold: 2
Intercept value: -0.3440329359356002
Weight values: [[ 2.9660126 ]
 [-0.43770473]
 [ 1.70840165]
 [-0.03521241]
 [ 0.73048204]
 [-0.91038514]
 [ 0.30144079]
 [-0.44808623]
 [ 0.5239548 ]
 [-0.38064936]
 [-0.04896075]
 [-1.42149217]
 [ 1.10420286]
 [-0.07581607]
 [-0.03415832]
 [-0.42791613]
 [-2.2542127 ]
 [-0.23601352]
 [ 0.50288494]
 [-0.5263755 ]
 [-1.28723678]
 [ 1.5397516 ]
 [ 0.975268  ]
 [ 0.71339553]
 [ 0.14176076]
 [ 0.707298  ]
 [-1.0102366 ]
 [ 1.08299974]
 [ 1.19653408]
 [ 0.77650339]
 [ 0.77289434]
 [ 0.65143789]
 [ 0.91836138]
 [ 1.10635   ]
 [-1.69440743]
 [-0.01329683]
 [ 0.52599596]
 [-0.68113683]
 [ 0.20252179]
 [-0.8327596 ]
 [ 1.63879494]
 [ 0.26351789]
 [ 0.25183166]
 [-0.85295311]
 [ 1.55727793]
 [-0.39762791]
 [ 1.45421831]
 [-1.28397797]
 [ 0.33593131]
 [ 0.13850517]
 [-0.80206549]
 [ 0.8842713 ]
 [ 1.99836805]
 [ 1.71786327]]
Mean Square Error: 1058.121
```


Fold: 2
Cost
Iterations
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
0
50
100
150
200
250
300

```
Predictions for fold: 3
Intercept value: -0.099816907610039
Weight values: [[ 1.03668716]
 [ 0.9110025 ]
 [ 0.18006235]
 [ 0.27332333]
 [ 0.14036991]
 [ 0.71467521]
 [-0.06813985]
 [-0.11985399]
 [-0.18622513]
 [ 1.24211602]
 [-0.40285275]
 [ 0.4789494 ]
 [-0.24179148]
 [-0.01547685]
 [ 0.66524128]
 [-0.48405555]
 [ 0.38789157]
 [ 0.39813271]
 [ 0.69954928]
 [ 1.16697427]
 [ 0.45193409]
 [-0.94441756]
 [ 0.14303645]
 [ 0.57335496]
 [-0.05329737]
 [ 0.81490329]
 [ 0.21702419]
 [ 0.42863214]
 [-0.24420769]
 [-1.30981232]
 [ 2.21126635]
 [-0.52515695]
 [-0.45647288]
 [ 0.24073837]
 [ 1.48986882]
 [ 0.261991  ]
 [-0.004942  ]
 [ 1.25002736]
 [-1.39197423]
 [ 1.84625494]
 [-2.05900621]
 [ 0.17046334]
 [-0.66353986]
 [-0.22963958]
 [ 1.41472572]
 [ 0.0156753 ]
 [ 1.55501601]
 [-0.28093608]
 [ 0.07532719]
 [-0.45836874]
 [-0.17091288]
 [-1.30445094]
 [-1.15807649]
 [ 0.5177827 ]]
Mean Square Error: 1080.518
```

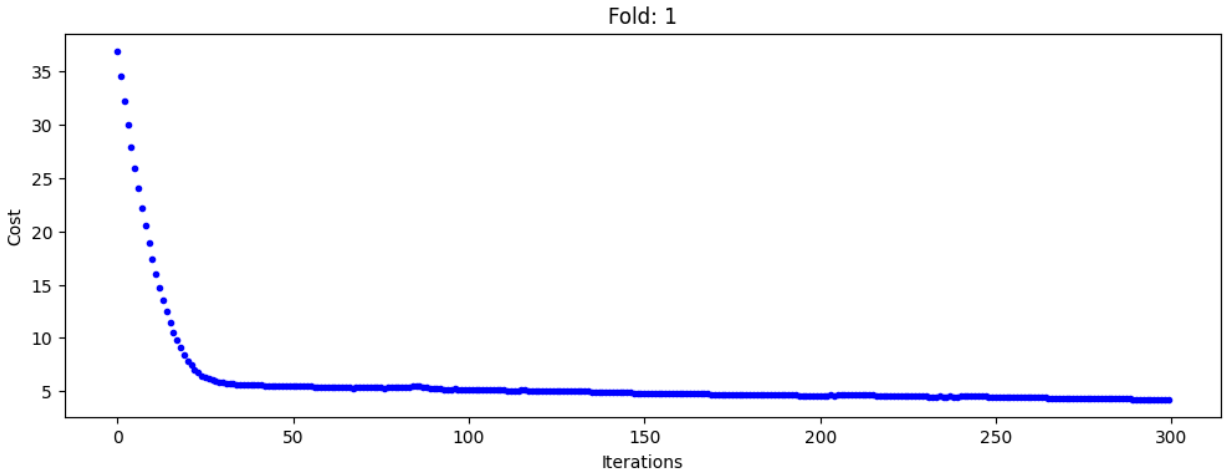

Fold: 3
Cost
Iterations
14
12
10
8
6
4
0
50
100
150
200
250
300

```
Predictions for fold: 4
Intercept value: 1.9319851787405609
Weight values: [[ 0.98879535]
 [-0.0433371 ]
 [ 0.71274147]
 [-0.84416275]
 [ 0.2306835 ]
 [-0.01798583]
 [ 1.10265806]
 [-0.88825454]
 [-0.25447767]
 [-0.45905219]
 [ 0.07279489]
 [-0.65286064]
 [-0.72835817]
 [-0.48298165]
 [ 1.08228887]
 [ 0.66124328]
 [ 0.62430411]
 [-0.40162769]
 [ 1.28422141]
 [-0.19485306]
 [ 2.37503751]
 [ 0.60474445]
 [-0.74880089]
 [-0.87110632]
 [ 0.13046524]
 [ 1.21260518]
 [ 0.39410854]
 [-1.16121812]
 [-0.48208635]
 [ 1.23475604]
 [ 0.89302324]
 [ 0.83568268]
 [ 2.28681391]
 [ 0.63404428]
 [-0.69110594]
 [ 0.18817483]
 [-0.28247894]
 [-0.7488481 ]
 [ 0.23383614]
 [ 0.91194838]
 [ 0.14877933]
 [-2.25155333]
 [ 2.22476514]
 [ 0.9939343 ]
 [-0.17507377]
 [-1.54072643]
 [ 0.57976403]
 [ 1.02791778]
 [ 0.10115281]
 [ 0.85433494]
 [-0.6821778 ]
 [ 0.98568044]
 [-0.49293982]
 [-0.74597323]]
Mean Square Error: 1004.493
```

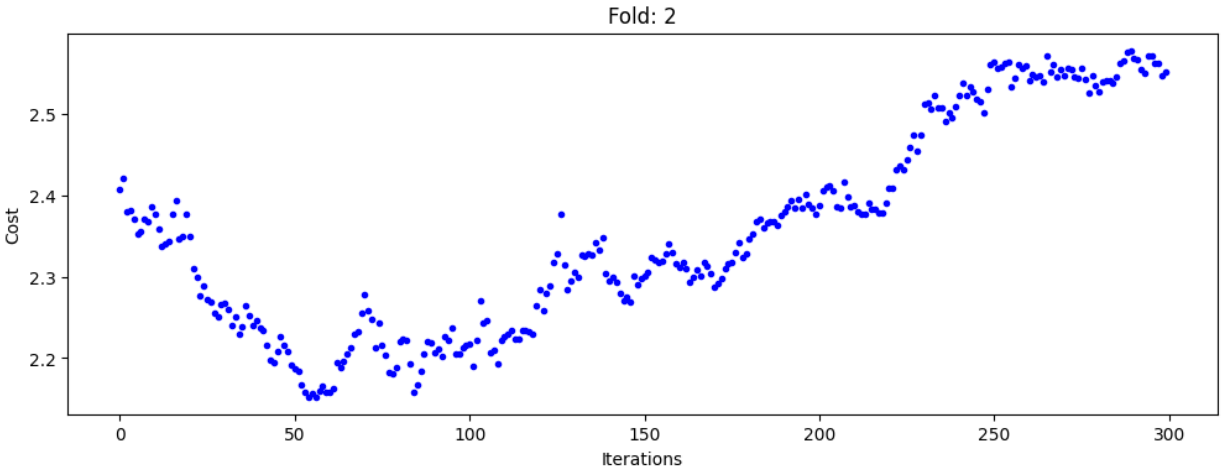

Fold: 4
Cost
Iterations
8
7
6
5
4
3
0
50
100
150
200
250
300

```
**********************************************************
The following graphs are for batch size: 100
**********************************************************
Predictions for fold: 1
Intercept value: -0.7201288369436148
Weight values: [[-0.93441839]
 [ 0.28102159]
 [ 0.06141375]
 [ 0.88692265]
 [ 0.49089989]
 [ 0.12177744]
 [ 0.0685765 ]
 [ 0.4283112 ]
 [-0.537569  ]
 [-0.61711371]
 [-0.8830913 ]
 [ 1.47220527]
 [-0.80370451]
 [ 1.27077556]
 [ 0.76460666]
 [ 0.92293649]
 [-0.7378662 ]
 [ 0.07379759]
 [-0.63186338]
 [ 0.90469296]
 [-0.79259255]
 [ 0.38359461]
 [ 2.18688161]
 [-0.2676849 ]
 [ 0.70394275]
 [ 0.28391745]
 [ 0.42732833]
 [ 0.18746806]
 [ 1.15500638]
 [ 1.25743951]
 [ 3.04417945]
 [ 0.88906921]
 [ 0.96464518]
 [ 0.30785865]
 [ 0.97901765]
 [ 1.17598108]
 [-1.18943486]
 [-0.17056301]
 [ 0.42740944]
 [-0.87827467]
 [-1.34836723]
 [ 0.03564239]
 [ 0.54844012]
 [ 0.54403332]
 [-0.66980264]
 [-0.51200167]
 [ 1.2349429 ]
 [ 1.06995541]
 [ 0.20002437]
 [ 1.22456806]
 [ 0.67289809]
 [-1.47376862]
 [ 0.1303189 ]
 [ 0.09208737]]
```

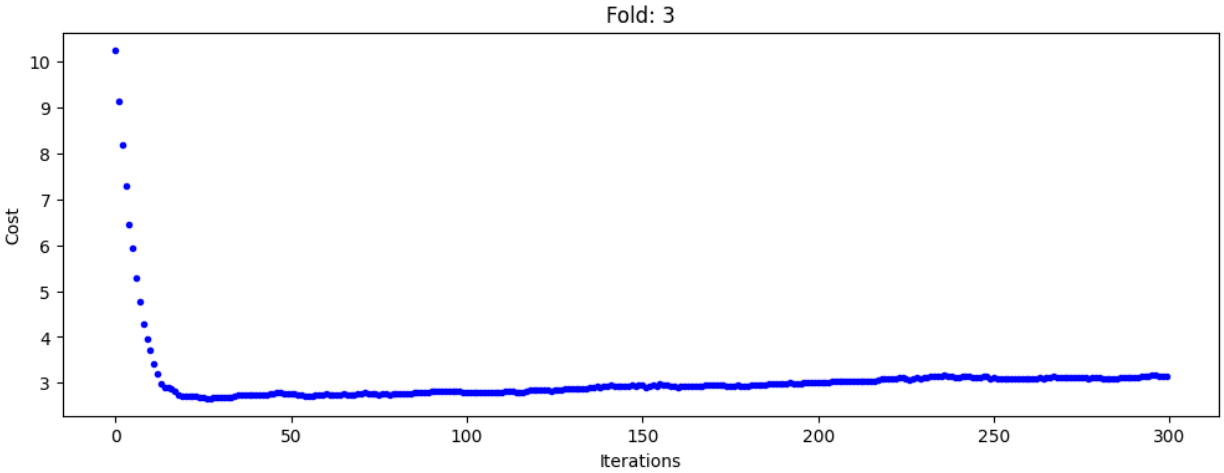
```
Mean Square Error: 1594.244
```



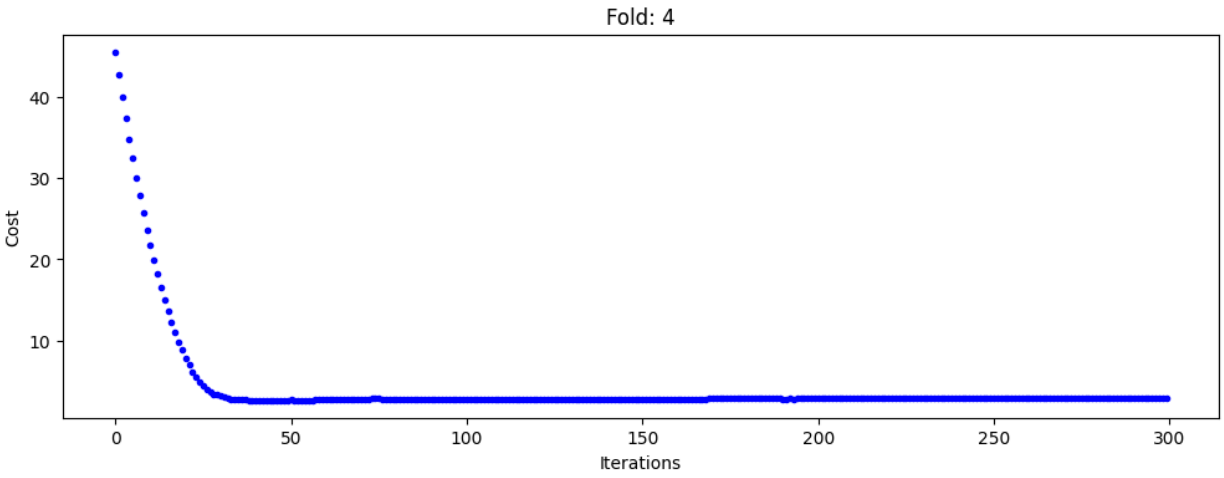
```
Predictions for fold: 2
Intercept value: 1.0696734003242623
Weight values: [[ 0.36850424]
 [ 0.28179706]
 [ 1.41210848]
 [ 0.27872364]
 [-1.33944605]
 [ 0.57313251]
 [ 0.91315899]
 [-0.63670222]
 [ 1.35486526]
 [ 0.49028588]
 [ 0.60647855]
 [ 0.0327628 ]
 [ 1.03495846]
 [-0.8138502 ]
 [-0.23907731]
 [ 0.50451514]
 [ 1.00979739]
 [ 1.2292788 ]
 [ 1.6888931 ]
 [-1.26116269]
 [ 1.0724585 ]
 [ 0.07660599]
 [-0.15687923]
 [-0.55522052]
 [ 2.32079439]
 [-1.7705184 ]
 [-2.53596958]
 [ 0.5492092 ]
 [ 1.30512788]
 [ 0.39741721]
 [-0.91411001]
 [-0.81220479]
 [-0.08906001]
 [ 1.33133779]
 [ 0.72018718]
 [-0.03773384]
 [ 1.20538325]
 [ 0.23303266]
 [-0.06571174]
 [-0.61353764]
 [ 0.94802594]
 [-0.28536672]
 [-0.44025375]
 [ 1.15894512]
 [ 0.9942756 ]
 [-0.34052765]
 [-0.25837523]
 [ 0.3320594 ]
 [-1.26439327]
 [-1.35030549]
 [-0.86602058]
 [ 1.25429788]
 [ 0.26844713]
 [ 0.18134393]]
Mean Square Error: 830.758
```


Fold: 2
Cost
2.5
2.4
2.3
2.2
0
50
100
150
200
250
300
Iterations

```
Predictions for fold: 3
Intercept value: -0.49910722230165483
Weight values: [[-1.22592021]
 [ 1.09051363]
 [ 1.10325363]
 [ 0.7021441 ]
 [-1.12332432]
 [ 1.41201982]
 [-1.09942251]
 [ 0.38006183]
 [ 0.49013285]
 [ 0.24430402]
 [-0.04087466]
 [-1.04814148]
 [ 0.02859024]
 [ 0.43213338]
 [ 0.62138735]
 [-0.30784091]
 [-1.7001115 ]
 [-0.83094905]
 [-0.11608223]
 [ 1.96741631]
 [-0.02691598]
 [-0.69240648]
 [ 0.08038954]
 [-0.20424261]
 [ 0.11863279]
 [-0.29085399]
 [-0.83547516]
 [ 0.95011242]
 [-0.57405877]
 [ 0.26739701]
 [ 2.60681574]
 [ 0.56832535]
 [ 1.67154414]
 [-0.21654869]
 [-0.79944563]
 [ 0.45394263]
 [-0.26189611]
 [ 1.96753956]
 [ 0.81398009]
 [ 0.19890911]
 [ 0.49921882]
 [ 2.21497727]
 [ 2.07333911]
 [ 1.88746487]
 [ 0.57038767]
 [-0.26225435]
 [-0.25722496]
 [-0.02310287]
 [-0.42931284]
 [-0.38659989]
 [ 1.08824615]
 [ 2.52515871]
 [-1.10759667]
 [ 0.85842253]]
Mean Square Error: 1119.169
```


Fold: 3
Cost
Iterations
10
9
8
7
6
5
4
3
0
50
100
150
200
250
300

```
Predictions for fold: 4
Intercept value: 1.283212187821903
Weight values: [[-0.06630534]
 [ 2.32745205]
 [-0.13405773]
 [-0.06338858]
 [ 1.95432821]
 [-1.16030982]
 [ 0.01769891]
 [ 0.79655003]
 [-0.42364268]
 [-0.07979591]
 [ 0.20712091]
 [ 0.87252704]
 [ 1.03721979]
 [-0.30450109]
 [ 1.55262016]
 [-0.73371995]
 [-0.63599391]
 [-0.28069152]
 [ 0.16105599]
 [-1.47166972]
 [-0.2231297 ]
 [-0.21335799]
 [-0.65754042]
 [ 1.96459558]
 [-1.03444101]
 [ 0.63542133]
 [ 0.56240012]
 [-1.08259546]
 [-0.6466897 ]
 [ 0.00521121]
 [ 1.43665477]
 [ 0.41072839]
 [ 0.58702405]
 [-1.46138412]
 [-0.96644603]
 [-0.52132748]
 [ 0.40407933]
 [-0.09517972]
 [ 0.13805604]
 [ 0.86913494]
 [ 1.03814005]
 [ 1.05704179]
 [ 0.92227946]
 [ 1.35273658]
 [ 0.9880736 ]
 [ 0.92479146]
 [ 1.43744333]
 [ 0.0557074 ]
 [-0.82477254]
 [ 1.28767145]
 [-0.95183791]
 [ 0.1501523 ]
 [ 0.5393098 ]
 [ 2.11721577]]
Mean Square Error: 1074.95
```

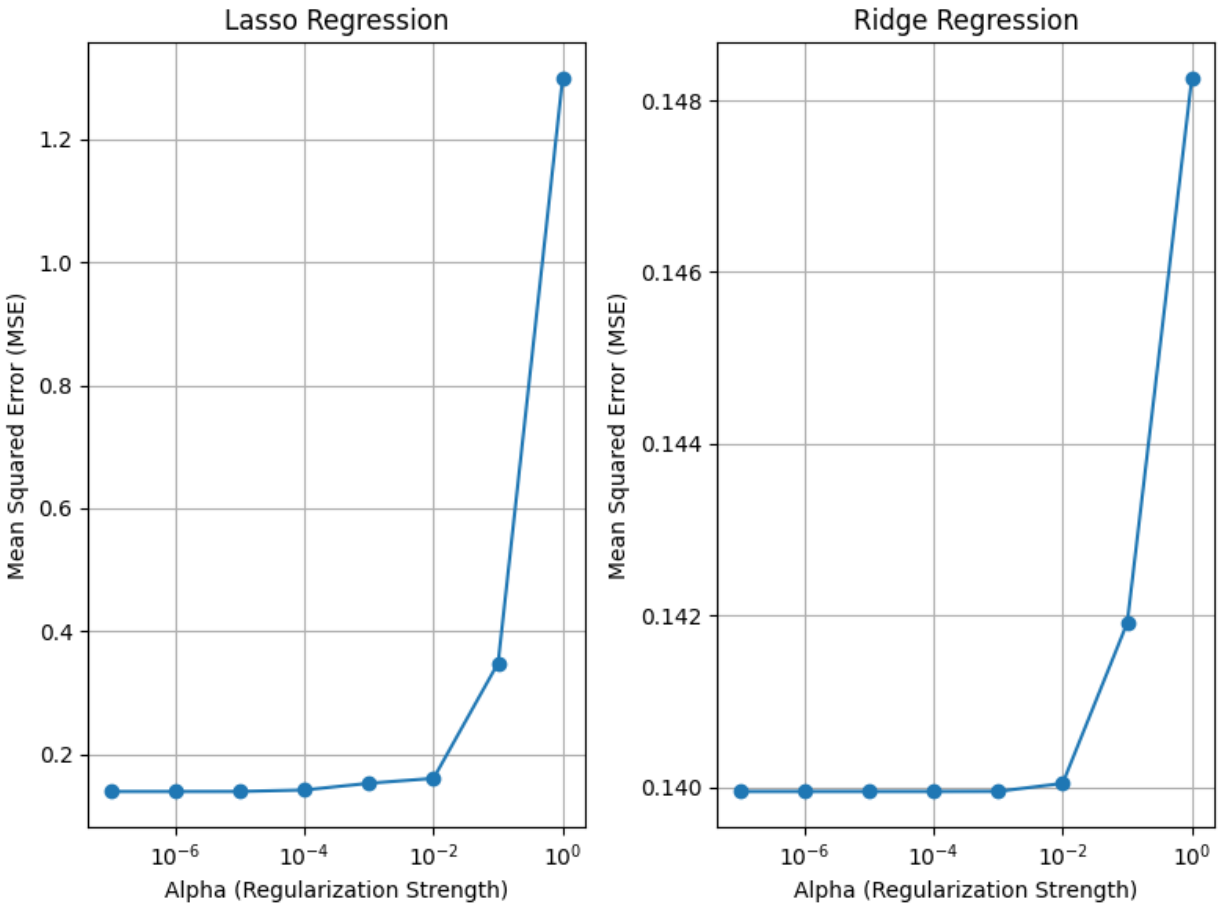


In [ ]:
```
from sklearn.linear_model import Lasso, Ridge
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt


# Define a list of alpha values to try
alphas = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1]

# Initialize empty lists to store the MSE values
lasso_mse_values_poly = []
ridge_mse_values_poly = []

for alpha in alphas:
    lasso_reg_poly = Lasso(alpha=alpha)
    ridge_reg_poly = Ridge(alpha=alpha)

    # Fit the Lasso model
    lasso_reg_poly.fit(X_train_poly, y_train)

    # Make predictions with the trained Lasso model
    lasso_predictions_poly = lasso_reg_poly.predict(X_train_poly)

    # Calculate MSE for Lasso
    lasso_mse_poly = mean_squared_error(y_train, lasso_predictions_poly)
    lasso_mse_values_poly.append(lasso_mse_poly)

    # Fit the Ridge model
    ridge_reg_poly.fit(X_train_poly, y_train)

    # Make predictions with the trained Ridge model
    ridge_predictions_poly = ridge_reg_poly.predict(X_train_poly)

    # Calculate MSE for Ridge
    ridge_mse_poly = mean_squared_error(y_train, ridge_predictions_poly)
    ridge_mse_values_poly.append(ridge_mse_poly)

    # Print the MSE values for each alpha
    print(f"Alpha = {alpha}")
    print("Lasso Regression Train MSE:", lasso_mse_poly)
    print("Ridge Regression Train MSE:", ridge_mse_poly)
    print("===")

# Create individual plots for each regularization technique
plt.figure(figsize=(12, 6))
```

```
# Lasso Regression Plot
plt.subplot(131)
plt.semilogx(alphas, lasso_mse_values_poly, marker='o')
plt.xlabel('Alpha (Regularization Strength)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Lasso Regression')
plt.grid(True)

# Ridge Regression Plot
plt.subplot(132)
plt.semilogx(alphas, ridge_mse_values_poly, marker='o')
plt.xlabel('Alpha (Regularization Strength)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Ridge Regression')
plt.grid(True)

plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 9.550e+01, tolerance: 1.773e-01
  model = cd_fast.enet_coordinate_descent(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 9.290e+01, tolerance: 1.773e-01
  model = cd_fast.enet_coordinate_descent(
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 6.490e+01, tolerance: 1.773e-01
  model = cd_fast.enet_coordinate_descent(
Alpha = 1e-07
Lasso Regression Train MSE: 0.14024320992884445
Ridge Regression Train MSE: 0.13994898955772997
===
Alpha = 1e-06
Lasso Regression Train MSE: 0.14024578460767928
Ridge Regression Train MSE: 0.13994898955904733
===
Alpha = 1e-05
Lasso Regression Train MSE: 0.14029793304241933
Ridge Regression Train MSE: 0.13994898969073832
===
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_desce
nt.py:631: ConvergenceWarning: Objective did not converge. You might want to i
ncrease the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 1.749e+00, tolerance: 1.773e-01
  model = cd_fast.enet_coordinate_descent(
```

```
Alpha = 0.0001
Lasso Regression Train MSE: 0.14269960481275495
Ridge Regression Train MSE: 0.13994900281463876
===
Alpha = 0.001
Lasso Regression Train MSE: 0.15362454156724387
Ridge Regression Train MSE: 0.1399502713708702
===
Alpha = 0.01
Lasso Regression Train MSE: 0.16158811659883954
Ridge Regression Train MSE: 0.1400438462368528
===
Alpha = 0.1
Lasso Regression Train MSE: 0.348891458819652
Ridge Regression Train MSE: 0.14191421695610337
===
Alpha = 1
Lasso Regression Train MSE: 1.298089382073318
Ridge Regression Train MSE: 0.14825716037778605
===
```



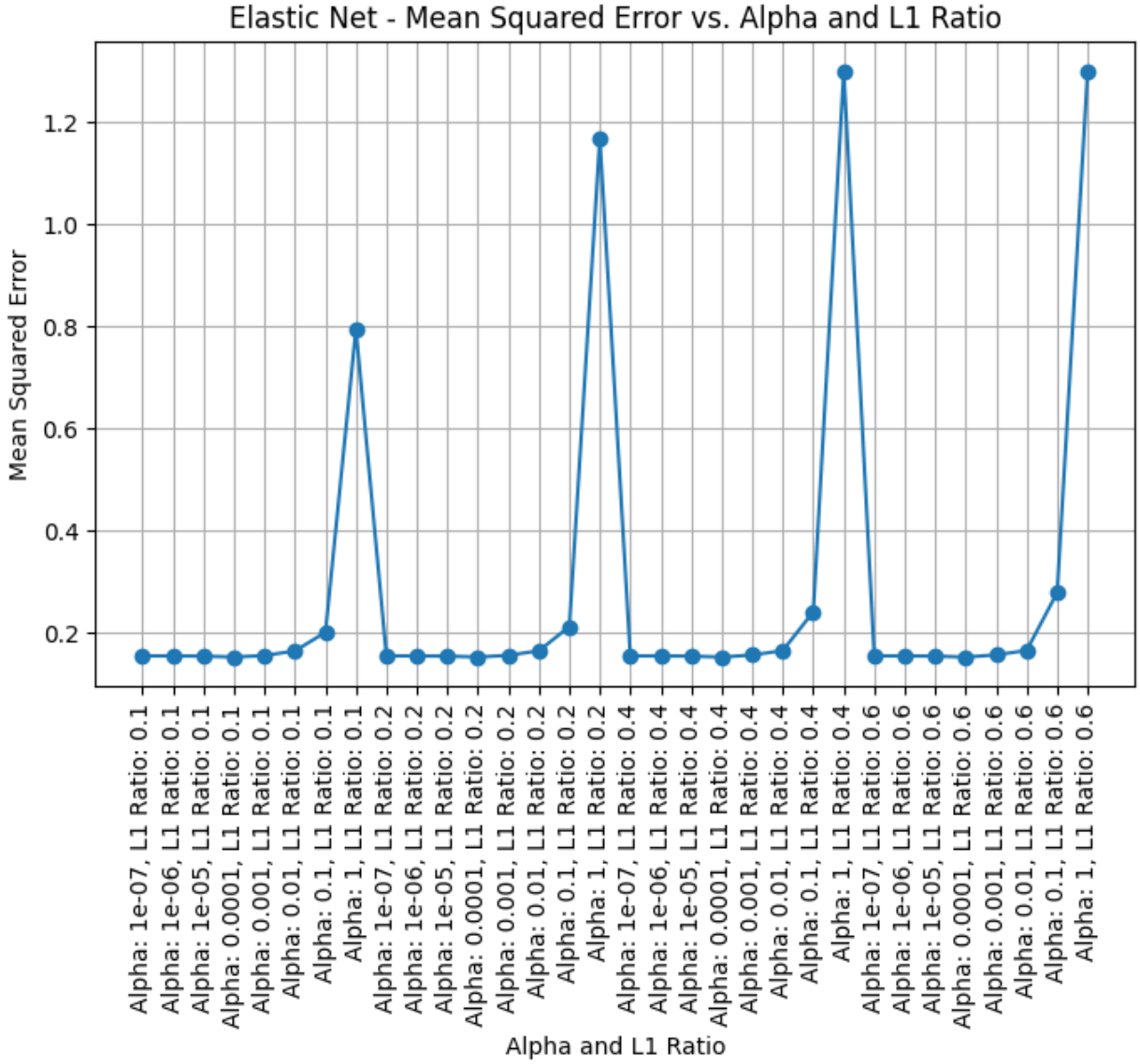
In [ ]:
```
import warnings
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
import warnings
warnings.filterwarnings("ignore")

# Define the number of folds (k)
```

```
k = 4

# Create lists of alpha (regularization strength) and l1_ratio (L1 mixing paran
alpha_list = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
l1_ratio_list = [0.1, 0.2, 0.4, 0.6]  # Different L1 ratios for Elastic Net

alpha_l1_list = []  # Store alpha and l1_ratio combinations
elastic_mse_vals = []  # Store Elastic Net MSE values

for l1_ratio in l1_ratio_list:
    for alpha in alpha_list:
        elastic_net_model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter

        # Initialize KFold cross-validator
        kf = KFold(n_splits=k, shuffle=True, random_state=42)

        fold_training_losses = []  # Store training losses for each fold
        fold_validation_losses = []  # Store validation losses for each fold

        for train_index, test_index in kf.split(X_train_poly):  # Use the polyn
            X_train_fold, X_val_fold = X_train_poly.iloc[train_index,:], X_trai
            y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[

            # Fit the Elastic Net model
            elastic_net_model.fit(X_train_fold, y_train_fold)

            # Predict on the training data
            y_train_pred = elastic_net_model.predict(X_train_fold)

            # Calculate training loss (Mean Squared Error) and append to the li
            train_loss = mean_squared_error(y_train_fold, y_train_pred)
            fold_training_losses.append(train_loss)

            # Predict on the validation data
            y_val_pred = elastic_net_model.predict(X_val_fold)

            # Calculate validation loss (Mean Squared Error) and append to the
            val_loss = mean_squared_error(y_val_fold, y_val_pred)
            fold_validation_losses.append(val_loss)

        # Calculate the mean training and validation loss across folds for this
        mean_training_loss = np.mean(fold_training_losses)
        mean_validation_loss = np.mean(fold_validation_losses)

        alpha_l1_list.append((alpha, l1_ratio))
        elastic_mse_vals.append(mean_validation_loss)

        # Final model evaluation
        elastic_net_model.fit(X_train_poly, y_train)  # Fit on the entire polyn
        y_pred_elastic_net = elastic_net_model.predict(X_test_poly)  # Use poly
        final_mse = mean_squared_error(y_test, y_pred_elastic_net)
        print(f"Alpha: {alpha}, L1 Ratio: {l1_ratio}, Final Mean Squared Error

# Plot training and validation loss as a function of alpha
plt.figure(figsize=(8, 5))
plt.plot(range(len(alpha_l1_list)), elastic_mse_vals, marker='o')
plt.xticks(range(len(alpha_l1_list)), [f"Alpha: {alpha}, L1 Ratio: {l1_ratio}"
plt.xlabel('Alpha and L1 Ratio')
plt.ylabel('Mean Squared Error')
plt.title('Elastic Net - Mean Squared Error vs. Alpha and L1 Ratio')
```

```
plt.grid()
plt.show()


warnings.resetwarnings()
```

```
Alpha: 1e-07, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.1465466
279579683
Alpha: 1e-06, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.1464986
3755954745
Alpha: 1e-05, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.1460467
4116829505
Alpha: 0.0001, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.143419
67131100997
Alpha: 0.001, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.1395464
7794381947
Alpha: 0.01, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.14122815
106406097
Alpha: 0.1, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.177792571
67150309
Alpha: 1, L1 Ratio: 0.1, Final Mean Squared Error (Elastic Net): 0.76155488770
24577
Alpha: 1e-07, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.1465466
8983437655
Alpha: 1e-06, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.1464991
5794262308
Alpha: 1e-05, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.1460493
9966321423
Alpha: 0.0001, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.143402
83592416014
Alpha: 0.001, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.1393670
5689934142
Alpha: 0.01, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.14103077
787127447
Alpha: 0.1, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 0.186814089
35178578
Alpha: 1, L1 Ratio: 0.2, Final Mean Squared Error (Elastic Net): 1.13150742065
1459
Alpha: 1e-07, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.1465468
1363702052
Alpha: 1e-06, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.1465001
8621467387
Alpha: 1e-05, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.1460583
8840635318
Alpha: 0.0001, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.143418
7151528157
Alpha: 0.001, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.1393389
071598924
Alpha: 0.01, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.14060992
210877757
Alpha: 0.1, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 0.213401507
0750539
Alpha: 1, L1 Ratio: 0.4, Final Mean Squared Error (Elastic Net): 1.26310550637
8901
Alpha: 1e-07, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.1465469
3741928254
Alpha: 1e-06, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.1465012
4238696677
Alpha: 1e-05, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.1460695
8578165108
Alpha: 0.0001, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.143475
66096872988
Alpha: 0.001, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.1397879
1105519686
Alpha: 0.01, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.14059735
71707867
```

```
Alpha: 0.1, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 0.247968296
3782455
Alpha: 1, L1 Ratio: 0.6, Final Mean Squared Error (Elastic Net): 1.26310550637
8901
```



Describing the impact for polynomial regression:

- Polynomial regression uses a linear model to fit non-linear data. The evaluation model has not improved as compared to the linear regression model. This would primarily be since our data has linear data as compared to non-linear and since polynomial works better with non-linear data, polynomial regression did not show an improvement in MSE for other regression models either (Lasso, Ridge and Elastic Net)

- Further, a high degree of polynomial tries to overfit the data, and for smaller values of degree, the model tries to underfit. In our model, we have used the degree 2 for polynomial features, which tends towards underfitting.

- Additionally, analyzing the batch size and learning rates, we observe that the performance of the model is not better as compared to the OLS model. The average MSE is significantly higher while using polynomial features.

Ref: Code from chapter 2 of Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow (3rd edition).

Part 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.

In [ ]:
```
import pandas as pd
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, SG
from sklearn.metrics import mean_squared_error

warnings.filterwarnings("ignore")

# Create a DataFrame to store MSE values
mse_table = pd.DataFrame(columns=['Model', 'MSE'])

# Linear Regression
linear_reg = LinearRegression()
linear_reg.fit(X_train, y_train)
linear_reg_predictions = linear_reg.predict(X_test)
linear_reg_mse = mean_squared_error(y_test, linear_reg_predictions)
mse_table = mse_table.append({'Model': 'Linear Regression', 'MSE': linear_reg_r

# Ridge Regression
ridge_reg = Ridge(alpha=0.0001)
ridge_reg.fit(X_train, y_train)
ridge_reg_predictions = ridge_reg.predict(X_test)
ridge_reg_mse = mean_squared_error(y_test, ridge_reg_predictions)
mse_table = mse_table.append({'Model': 'Ridge Regression', 'MSE': ridge_reg_mse

# Lasso Regression
lasso_reg = Lasso(alpha=0.0001)
lasso_reg.fit(X_train, y_train)
lasso_reg_predictions = lasso_reg.predict(X_test)
lasso_reg_mse = mean_squared_error(y_test, lasso_reg_predictions)
mse_table = mse_table.append({'Model': 'Lasso Regression', 'MSE': lasso_reg_mse

# Elastic Net Regression
elastic_net = ElasticNet(alpha=0.0001, l1_ratio=0.1)
elastic_net.fit(X_train, y_train)
elastic_net_predictions = elastic_net.predict(X_test)
elastic_net_mse = mean_squared_error(y_test, elastic_net_predictions)
mse_table = mse_table.append({'Model': 'Elastic Net', 'MSE': elastic_net_mse},

# SGDRegressor (Stochastic Gradient Descent)
sgd_reg = SGDRegressor(alpha=0.0001, max_iter=1000, random_state=42)
sgd_reg.fit(X_train, y_train)
sgd_reg_predictions = sgd_reg.predict(X_test)
sgd_reg_mse = mean_squared_error(y_test, sgd_reg_predictions)
mse_table = mse_table.append({'Model': 'SGD Regression', 'MSE': sgd_reg_mse}, i
```

```
# Display the MSE values
print(mse_table)

warnings.resetwarnings()
```

```
                Model       MSE
0   Linear Regression  0.139131
1    Ridge Regression  0.139131
2    Lasso Regression  0.139078
3         Elastic Net  0.139136
4      SGD Regression  0.191477
```

Here, with the best set of hyperparamters for each model with different regularization, Lasso is performing the best with least MSE which implies a sparse solution will be best for the model. This means: (i) Features could be noisy (ii) Shrinking the feature set or reducing coefficient of certain features to 0 is helping means not all features are relevant for determining the output.

Elastic net is performing worse than L1, that means L2 regularization drops the performance even with combination.

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

# Create a DataFrame to store MSE values
mse_table_poly = pd.DataFrame(columns=['Model', 'MSE_poly'])

# Linear Regression
linear_reg_poly = LinearRegression()
linear_reg_poly.fit(X_train_poly, y_train)
linear_reg_predictions_poly = linear_reg_poly.predict(X_test_poly)
linear_reg_mse_poly = mean_squared_error(y_test, linear_reg_predictions_poly)
mse_table_poly = mse_table_poly.append({'Model': 'Linear Regression', 'MSE_poly

# Ridge Regression
ridge_reg_poly = Ridge(alpha=0.0001)
ridge_reg_poly.fit(X_train_poly, y_train)
ridge_reg_predictions_poly = ridge_reg_poly.predict(X_test_poly)
ridge_reg_mse_poly = mean_squared_error(y_test, ridge_reg_predictions_poly)
mse_table_poly = mse_table_poly.append({'Model': 'Ridge Regression', 'MSE_poly

# Lasso Regression
lasso_reg_poly = Lasso(alpha=0.0001)
lasso_reg_poly.fit(X_train_poly, y_train)
lasso_reg_predictions_poly = lasso_reg_poly.predict(X_test_poly)
lasso_reg_mse_poly = mean_squared_error(y_test, lasso_reg_predictions_poly)
mse_table_poly = mse_table_poly.append({'Model': 'Lasso Regression', 'MSE_poly

# Elastic Net Regression
elastic_net_poly = ElasticNet(alpha=0.0001, l1_ratio=0.1)  # Adjust l1_ratio as
elastic_net_poly.fit(X_train_poly, y_train)
elastic_net_predictions_poly = elastic_net_poly.predict(X_test_poly)
elastic_net_mse_poly = mean_squared_error(y_test, elastic_net_predictions_poly)
mse_table_poly = mse_table_poly.append({'Model': 'Elastic Net', 'MSE_poly': ela

# SGDRegressor (Stochastic Gradient Descent)
sgd_reg_poly = SGDRegressor(alpha=0.0001, max_iter=1000, random_state=42)
sgd_reg_poly.fit(X_train_poly, y_train)
sgd_reg_predictions_poly = sgd_reg_poly.predict(X_test_poly)
```

```
sgd_reg_mse_poly = mean_squared_error(y_test, sgd_reg_predictions_poly)
mse_table_poly = mse_table_poly.append({'Model': 'SGD Regression', 'MSE_poly':

# Display the MSE values
print(mse_table_poly)

warnings.resetwarnings()
```

```
               Model  MSE_poly
0  Linear Regression  0.145150
1   Ridge Regression  0.145136
2   Lasso Regression  0.141824
3        Elastic Net  0.141941
4     SGD Regression  0.157209
```

Based on the best possible combination of hyperparameters selected for each model Lasso Regression with polynomial features has the least MSE, so Lasso is performing best with polynomial regression, which implies that a sparse solution is better suited for the data. This means that there are features which are not affecting the labels or the features are noisy. Also the MSE is higher as compared to the MSE for Lasso Linear Regression, which means we are not gaining much from polynomial regression .

Possible improvements:

(i) Using a feature reduction or dimensionality reduction technique while preparing dataset for training to have a better set of features.

(ii) Increasing the number of data points, if possible.

(iii) Use ensemble such as bagging (Random Forest) and boosting methods (Extreme Gradient Boosting, Adaptive Boosting) while training the dataset.

In [ ]:

In [ ]:
```
for iteration in tqdm(range(n_iterations)):
            # Fit the model for one iteration (one pass through the training da
            sgd_model.partial_fit(X_train_sc, y_train)

            # Predict on the training data
            y_train_pred = sgd_model.predict(X_train_sc)

            # Calculate training loss (Mean Squared Error) and append to the li
            train_loss = mean_squared_error(y_train, y_train_pred)
            training_loss.append(train_loss)

            # Predict on the validation data
            y_val_pred = sgd_model.predict(X_test_sc)

            # Calculate validation loss (Mean Squared Error) and append to the
            val_loss = mean_squared_error(y_test, y_val_pred)
            validation_loss.append(val_loss)

            # Check for early stopping
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                no_improvement_count = 0
```

```
            else:
                no_improvement_count += 1

            if no_improvement_count >= early_stopping_rounds:
                print(f"Early stopping at iteration {iteration + 1} due to no i
                break

        # Plot training and validation loss as a function of training iterati
        plt.figure(figsize=(8, 5))
        plt.plot(range(1, iteration + 2), training_loss, label='Training Loss',
        plt.plot(range(1, iteration + 2), validation_loss, label='Validation Lo
        plt.xlabel('Training Iteration')
        plt.ylabel('Mean Squared Error')
        plt.title('Training and Validation Loss vs. Training Iteration')
        plt.legend()
        plt.grid()
        plt.show()

        # Final model evaluation
        y_pred_sgd = sgd_model.predict(X_test_sc)
        final_rmse = mean_squared_error(y_test_sc, y_pred_sgd)
        print("Final Root Mean Squared Error (SGD):", final_rmse)
```