r76nmm11o

October 5, 2024

0.0.1 Applied Machine Learning Homework 1 Question 1

In this question the goal is to explore how to properly analyze, visualize, split, clean and format data and perform linear regression, polynomial regression and regularization. The Walmart Sales data is taken and it will predict the Unemployment based on factors like Temperature, Weekly salary, Number of holidays, Fuel Prices, Etc

[59]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

# Load dataset
df = pd.read_csv('Walmart_Sales.csv')

print("Number of rows: ", df.shape[0]," \nNumber of columns: ", df.shape[1])

# Separate continuous and categorical features
continuous_cols = df.select_dtypes(include=['float64', 'int64']).columns
categorical_cols = df.select_dtypes(include=['object', 'category']).columns

print(f"\nContinuous Features:\n{continuous_cols}")
print(f"\nCategorical Features:\n{categorical_cols}")
```

```
Number of rows:  6435
Number of columns:  8

Continuous Features:
Index(['Store', 'Weekly_Sales', 'Holiday_Flag', 'Temperature', 'Fuel_Price',
       'CPI', 'Unemployment'],
      dtype='object')

Categorical Features:
Index(['Date'], dtype='object')
```

[36]:
```
print(df.describe())
```

```
             Store  Weekly_Sales  Holiday_Flag  Temperature   Fuel_Price  \
count  6435.000000  6.435000e+03   6435.000000  6435.000000  6435.000000
mean     23.000000  1.046965e+06      0.069930    60.663782     3.358607
std      12.988182  5.643666e+05      0.255049    18.444933     0.459020
min       1.000000  2.099862e+05      0.000000    -2.060000     2.472000
25%      12.000000  5.533501e+05      0.000000    47.460000     2.933000
50%      23.000000  9.607460e+05      0.000000    62.670000     3.445000
75%      34.000000  1.420159e+06      0.000000    74.940000     3.735000
max      45.000000  3.818686e+06      1.000000   100.140000     4.468000

               CPI  Unemployment
count  6435.000000   6435.000000
mean    171.578394      7.999151
std      39.356712      1.875885
min     126.064000      3.879000
25%     131.735000      6.891000
50%     182.616521      7.874000
75%     212.743293      8.622000
max     227.232807     14.313000
```

[37]:

```python
print(df.head())
```

```
   Store        Date  Weekly_Sales  Holiday_Flag  Temperature  Fuel_Price  \
0      1  05-02-2010    1643690.90             0        42.31       2.572
1      1  12-02-2010    1641957.44             1        38.51       2.548
2      1  19-02-2010    1611968.17             0        39.93       2.514
3      1  26-02-2010    1409727.59             0        46.63       2.561
4      1  05-03-2010    1554806.68             0        46.50       2.625

          CPI  Unemployment
0  211.096358         8.106
1  211.242170         8.106
2  211.289143         8.106
3  211.319643         8.106
4  211.350143         8.106
```

[38]:

```python
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6435 entries, 0 to 6434
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Store         6435 non-null   int64
 1   Date          6435 non-null   object
 2   Weekly_Sales  6435 non-null   float64
 3   Holiday_Flag  6435 non-null   int64
 4   Temperature   6435 non-null   float64
```

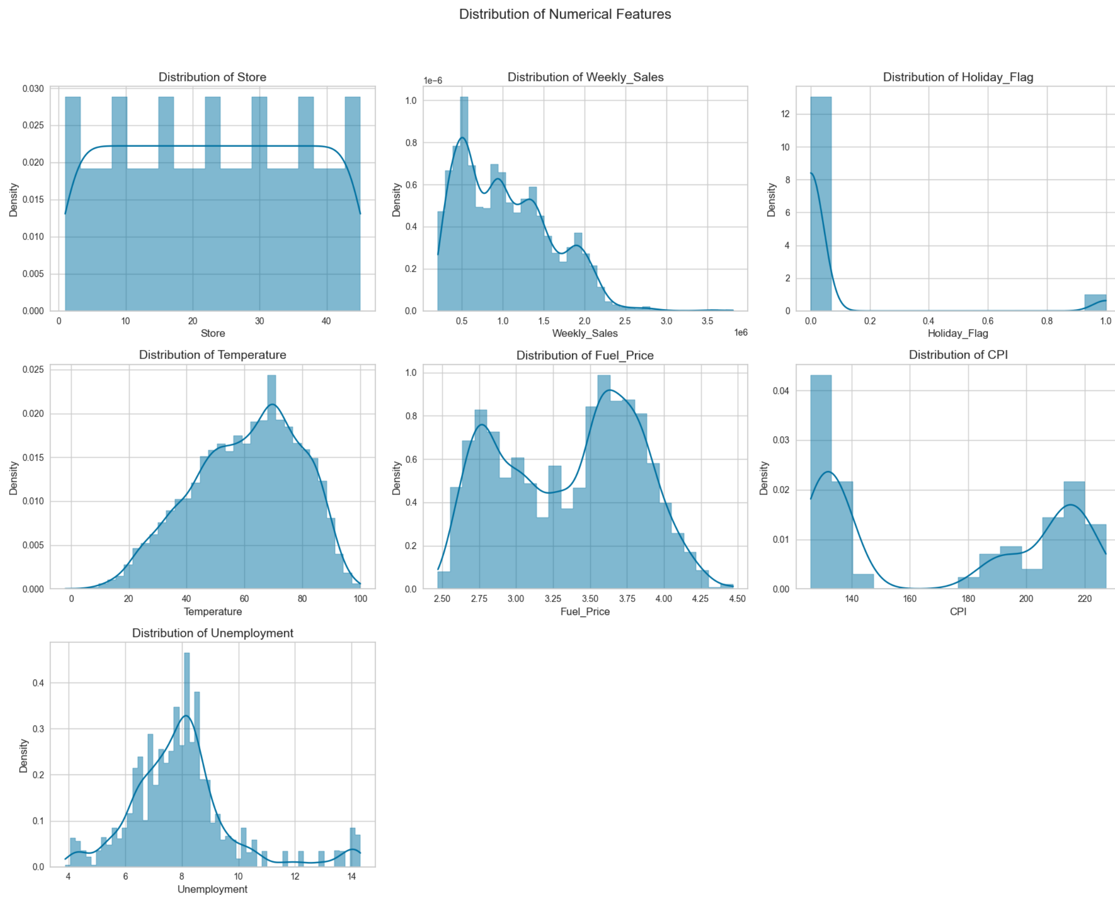
```
 5   Fuel_Price    6435 non-null   float64
 6   CPI           6435 non-null   float64
 7   Unemployment  6435 non-null   float64
dtypes: float64(5), int64(2), object(1)
memory usage: 402.3+ KB
None
```

0.0.2 Summarization of Data

In this subquestion the shape of the data is determined with the categorical and continuous features. The data statistics are further analysed.

[69]:

```python
# Histogram
numerical_columns = df.select_dtypes(include=['number']).columns

num_cols = 3
num_rows = (len(numerical_columns) + num_cols - 1) // num_cols

fig, axes = plt.subplots(num_rows, num_cols, figsize=(18, 5 * num_rows))
fig.suptitle("Distribution of Numerical Features", fontsize=16)

axes = axes.flatten()

for i, col in enumerate(numerical_columns):
    sns.histplot(df[col], kde=True, ax=axes[i], palette="viridis",␣
 ↪element='step', stat='density')
    axes[i].set_title(f'Distribution of {col}', fontsize=14)
    axes[i].set_xlabel(col, fontsize=12)
    axes[i].set_ylabel('Density', fontsize=12)

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

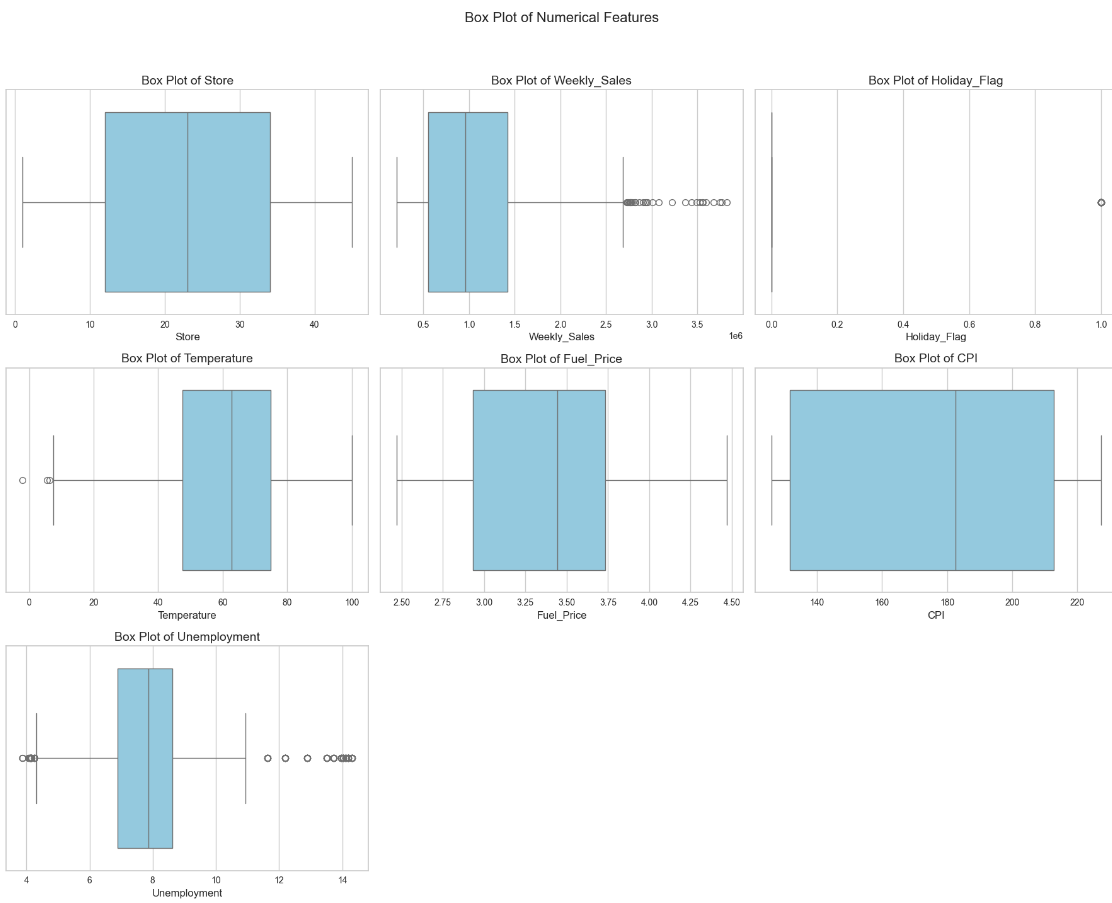


0.0.3 Histogram of Continuous Features

Store: The distribution is skewed to the right, indicating that there are more stores with lower IDs. Weekly_Sales: The distribution is skewed to the right, suggesting that most stores have lower weekly sales, with a few stores having significantly higher sales. Holiday_Flag: The distribution is bimodal, indicating two distinct groups of stores: those with a holiday flag and those without. Temperature: The distribution is approximately normal, with a slight skew to the right. Fuel_Price: The distribution is also approximately normal, with a slight skew to the right. CPI: The distribution is skewed to the right, suggesting that most stores are located in areas with lower CPI. Unemployment: The distribution is skewed to the right, indicating that most stores are located in areas with lower unemployment rates.

[70]:
```
# Box Plot
numerical_columns = df.select_dtypes(include=['number']).columns

num_cols = 3
num_rows = (len(numerical_columns) + num_cols - 1) // num_cols

fig, axes = plt.subplots(num_rows, num_cols, figsize=(18, 5 * num_rows))
```

```
fig.suptitle('Box Plot of Numerical Features', fontsize=16)

axes = axes.flatten()

for i, col in enumerate(numerical_columns):
    sns.boxplot(x=df[col], ax=axes[i], color="skyblue")
    axes[i].set_title(f'Box Plot of {col}', fontsize=14)
    axes[i].set_xlabel(col, fontsize=12)

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



0.0.4 Box Plot for Continuous Features

Store: The data is relatively evenly distributed, with no significant outliers. Weekly_Sales: There are some outliers on the higher end, indicating a few stores with exceptionally high weekly sales.

Holiday_Flag: The data is skewed to the left, with a majority of stores having a holiday flag of 0. Temperature: The data is relatively evenly distributed, with no significant outliers. Fuel_Price: The data is slightly skewed to the left, with a majority of stores having fuel prices around 3.5. CPI: The data is slightly skewed to the left, with a majority of stores located in areas with lower CPI. Unemployment: The data is skewed to the left, indicating that most stores are located in areas with lower unemployment rates.

[8]:

```
# Check for missing values
print("\nMissing Values:")
missing_values = df.isnull().sum()
print(missing_values)
```

```
Missing Values:
Store           0
Date            0
Weekly_Sales    0
Holiday_Flag    0
Temperature     0
Fuel_Price      0
CPI             0
Unemployment    0
dtype: int64
```

[9]:

```
# Check for outliers
def detect_outliers(df, col):
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
    return len(outliers)

print("\nNumber of outliers:")
for col in continuous_cols:
    print(f"{col}: {detect_outliers(df, col)}")

# Handle Outliers
for column in df.select_dtypes(include=['number']).columns:
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    df[column] = df[column].clip(lower=lower_bound, upper=upper_bound)
```

```
Number of outliers:
Store: 0
Weekly_Sales: 34
Holiday_Flag: 450
Temperature: 3
Fuel_Price: 0
CPI: 0
Unemployment: 481
```

0.0.5 Missing Values:

The data is checked for missing values in the dataset using isnull().sum(), and found that there are no missing values in any of the columns. This is a positive trait, as it means you won't need to deal with data imputation or removal due to missing data.

0.0.6 Outlier Detection:

Implemented a function detect_outliers() to identify outliers in continuous attributes using the Interquartile Range (IQR) method. This method defines outliers as any points that fall. Furthermore, the outliers are handled by capping the data.

[99]:
```
# Pearson Correlation
df.corr(method='pearson', numeric_only=True)
```

[99]:
```
                     Store  Weekly_Sales  Holiday_Flag  Temperature  \
Store         1.000000e+00     -0.335332 -4.386841e-16    -0.022659
Weekly_Sales -3.353320e-01      1.000000  3.689097e-02    -0.063810
Holiday_Flag -4.386841e-16      0.036891  1.000000e+00    -0.155091
Temperature  -2.265908e-02     -0.063810 -1.550913e-01     1.000000
Fuel_Price    6.002295e-02      0.009464 -7.834652e-02     0.144982
CPI          -2.094919e-01     -0.072634 -2.162091e-03     0.176888
Unemployment  2.235313e-01     -0.106176  1.096028e-02     0.101158

              Fuel_Price       CPI  Unemployment
Store           0.060023 -0.209492      0.223531
Weekly_Sales    0.009464 -0.072634     -0.106176
Holiday_Flag   -0.078347 -0.002162      0.010960
Temperature     0.144982  0.176888      0.101158
Fuel_Price      1.000000 -0.170642     -0.034684
CPI            -0.170642  1.000000     -0.302020
Unemployment   -0.034684 -0.302020      1.000000
```

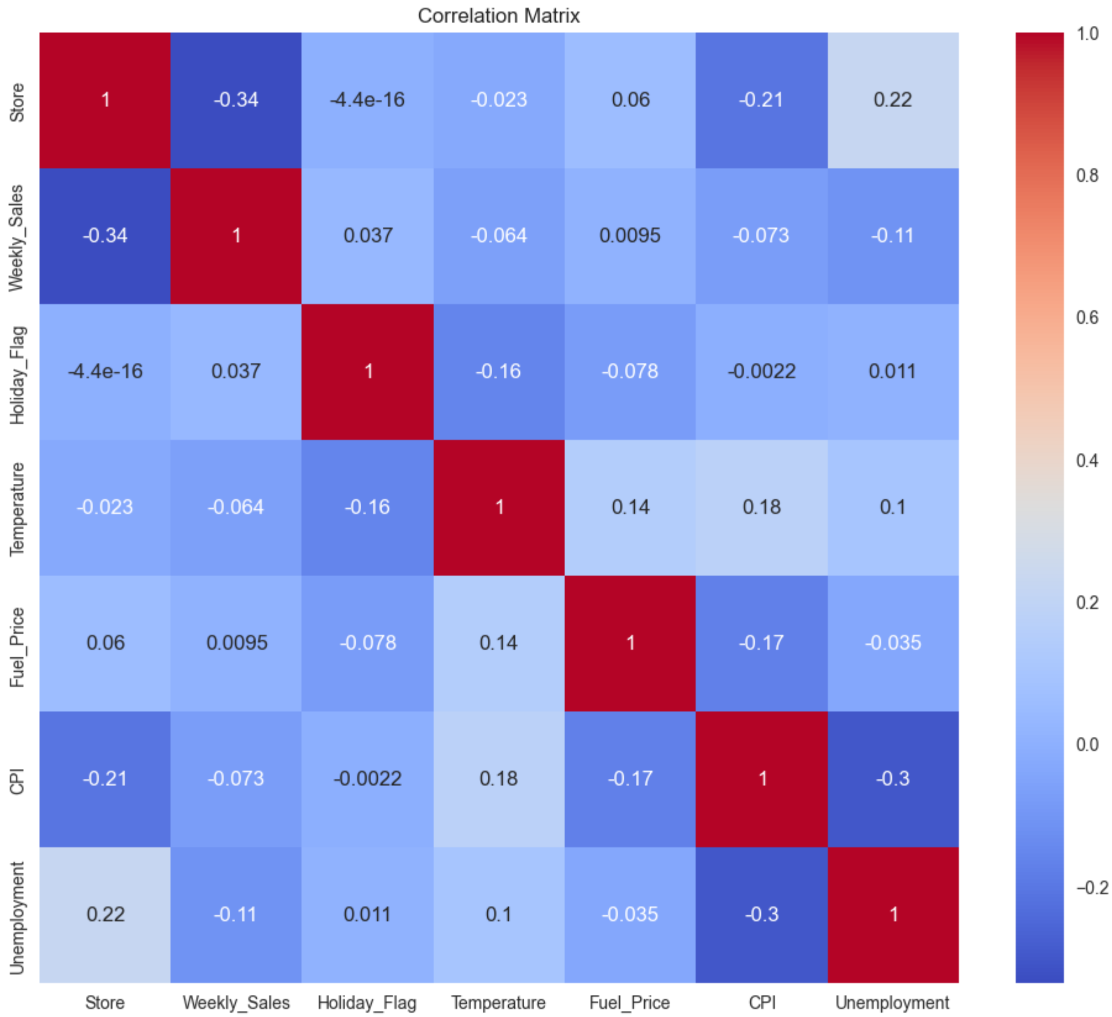
0.0.7 Pearson Correlation

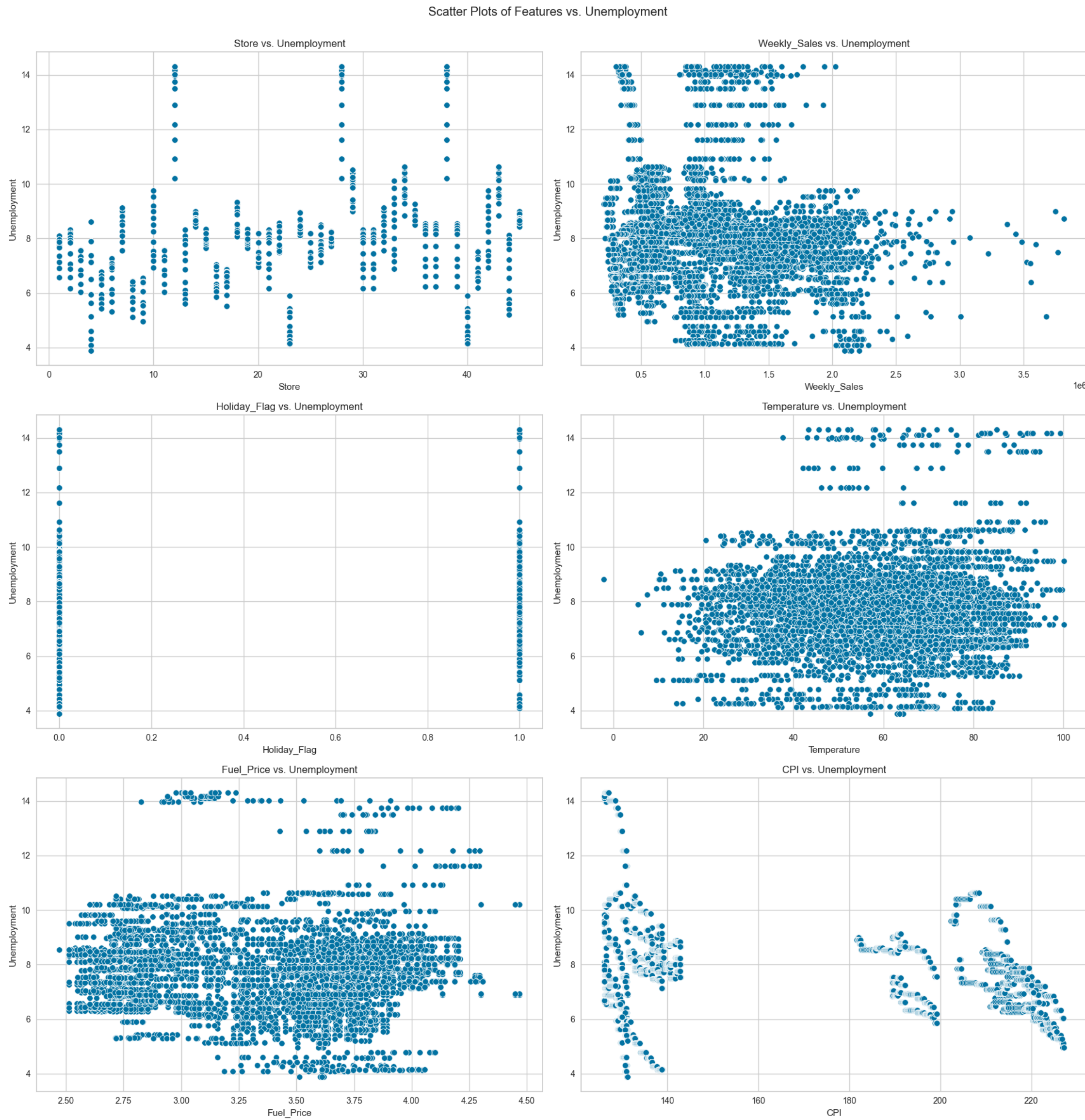
Store & Weekly_Sales: Moderate negative correlation (-0.335). Weekly_Sales & Holiday_Flag: Very weak positive correlation (0.037). Temperature: Weak negative correlation with sales (-0.064). Fuel_Price: Negligible correlation (0.009). CPI & Unemployment: Notable negative correlation (-0.302).

The most significant correlation is between CPI and Unemployment (-0.302), suggesting that as CPI increases, unemployment tends to decrease.

[71]:

```
# Heatmap of continuous values
correlation_matrix = df[continuous_cols].corr()

plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()



# Plot of Scatter for Unemployment vs continuous values
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(3, 2, figsize=(18, 18))
x_vars = ['Store', 'Weekly_Sales', 'Holiday_Flag', 'Temperature', 'Fuel_Price',␣
 ↪'CPI']

for ax, x_var in zip(axes.flatten(), x_vars):
    sns.scatterplot(data=df, x=x_var, y='Unemployment', ax=ax)
    ax.set_title(f'{x_var} vs. Unemployment')
    ax.set_xlabel(x_var)
    ax.set_ylabel('Unemployment')

plt.tight_layout()
plt.suptitle('Scatter Plots of Features vs. Unemployment', y=1.02)
plt.show()
```


Correlation Matrix
Store
Weekly_Sales
Holiday_Flag
Temperature
Fuel_Price
CPI
Unemployment
1 -0.34 -4.4e-16 -0.023 0.06 -0.21 0.22
-0.34 1 0.037 -0.064 0.0095 -0.073 -0.11
-4.4e-16 0.037 1 -0.16 -0.078 -0.0022 0.011
-0.023 -0.064 -0.16 1 0.14 0.18 0.1
0.06 0.0095 -0.078 0.14 1 -0.17 -0.035
-0.21 -0.073 -0.0022 0.18 -0.17 1 -0.3
0.22 -0.11 0.011 0.1 -0.035 -0.3 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2



0.0.8 Correlation matrix Heatmap

Store size or location could be a significant factor influencing weekly sales. Economic conditions (CPI and Unemployment) might have a weaker impact on weekly sales compared to store-specific factors. Seasonal factors (related to holidays or temperature) might have a minimal effect on the overall sales.

0.0.9 Scatter Plots

The scatter plots show the relationship between unemployment and various other features.

Store vs. Unemployment: There seems to be no clear relationship between store and unemployment. Weekly Sales vs. Unemployment: There is a slight negative correlation, suggesting that higher unemployment might be associated with slightly lower weekly sales. Holiday Flag vs. Unemployment: There is no clear relationship between holiday flag and unemployment. Temperature vs. Unemployment: There is a weak negative correlation, suggesting that higher unemployment might be associated with slightly lower temperatures. Fuel Price vs. Unemployment: There is a slight negative correlation, suggesting that higher unemployment might be associated with slightly lower fuel prices. CPI vs. Unemployment: There is a strong negative correlation, suggesting that higher unemployment is associated with lower CPI.

[42]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

target = 'Unemployment'
drop = [target, 'Date']
X = df.drop(columns=drop, axis = 1)
y = df[target]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=42)

print("Full dataset mean:", y.mean())
print("Test set mean:", y_test.mean())

print("\nFull dataset std:", y.std())
print("Test set std:", y_test.std())
```

```
Full dataset mean: 7.871208313908314
Test set mean: 7.855452144188937

Full dataset std: 1.5206938208999887
Test set std: 1.5244441213892517
```

[43]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(4826, 6)
(1609, 6)
(4826,)
(1609,)
```

0.0.10 Train Test Split

The data preparation involved designating Unemployment as the target variable and removing it along with the Date column from the feature set. The train_test_split function from sklearn.model_selection was used to allocate 75% of the data for training and 25% for testing.

To verify that the test set was representative of the entire dataset, the means and standard deviations of the Unemployment variable were compared. The full dataset had a mean of 7.87 and a standard deviation of 1.52, while the test set had a mean of 7.86 and the same standard deviation. These close values indicated that the test set accurately reflected the characteristics of the entire dataset.

[44]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold
from sklearn.metrics import r2_score, mean_squared_error, accuracy_score

kf = KFold(n_splits=3, shuffle = True, random_state=42)

linear_model=LinearRegression().fit(X_train,y_train)
linear_model_pred=linear_model.predict(X_test)
# print("Accuracy: ",accuracy_score(y_test, linear_model_pred))
print("RMSE:",np.sqrt(mean_squared_error(y_test,linear_model_pred)))
print("R*2:",r2_score(y_test,linear_model_pred))
```

```
RMSE: 1.3934340861611314
R*2: 0.1639738922706434
```

[45]:
```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import make_pipeline

def create_pipeline(model):
    return make_pipeline(
        SimpleImputer(strategy='mean'),
        StandardScaler(),
        model
    )

# Closed-form solution
lr_pipeline = create_pipeline(LinearRegression())
lr_scores = cross_val_score(lr_pipeline, X_train, y_train, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
print("Closed-form MSE:", -lr_scores.mean())

# SGD
sgd_pipeline = create_pipeline(SGDRegressor(max_iter=1000, tol=1e-3,␣
 ↪random_state=42))
sgd_scores = cross_val_score(sgd_pipeline, X_train, y_train, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
print("SGD MSE:", -sgd_scores.mean())
```

```
Closed-form MSE: 1.9568240441908575
```

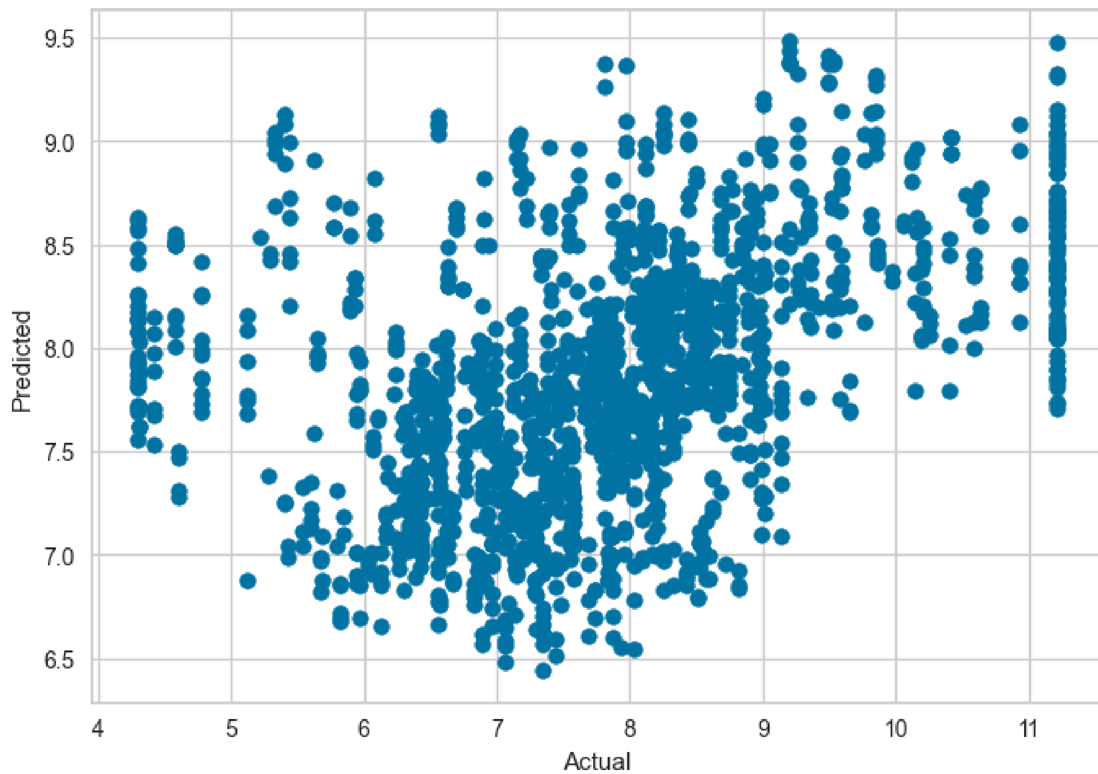
```
SGD MSE: 1.9586497298327867
```

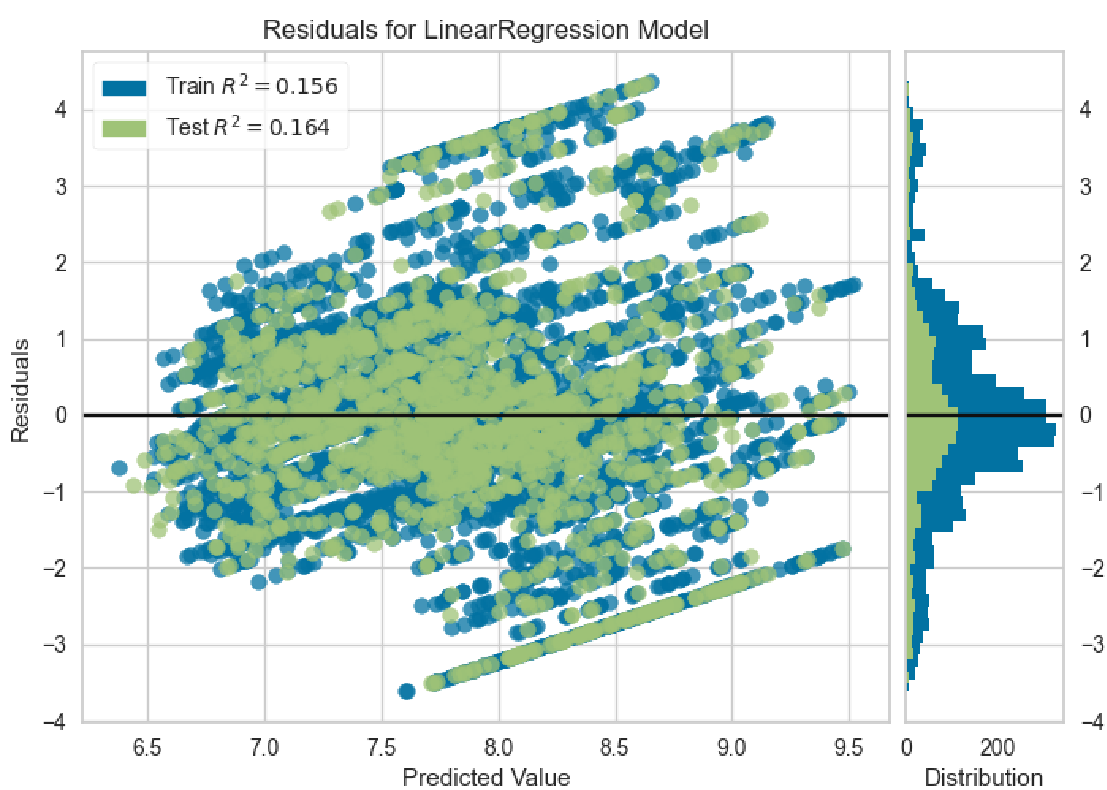
0.0.11 Linear Regression and SGD Model

A linear regression model was fitted to the training data, and predictions were made on the test set, yielding a Root Mean Squared Error (RMSE) of approximately 1.39 and an R^2 score of 0.16, indicating a limited ability to explain variance in the target variable.

Additionally, a pipeline was created to standardize data and handle missing values using a `SimpleImputer` with mean imputation. Cross-validation was performed using K-Fold (3 splits) to evaluate model performance. The closed-form Linear Regression yielded a Mean Squared Error (MSE) of about 1.96, while a Stochastic Gradient Descent (SGD) model resulted in a similar MSE of approximately 1.96. Overall, both models demonstrated comparable performance.

[76]:
```python
import numpy as np
from sklearn.linear_model import LinearRegression
from yellowbrick.regressor import PredictionError, ResidualsPlot

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

# Prediction Plot
plt.scatter(y_test, linear_model_pred)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.show()

# ResidualsPlot visualizer
residuals_viz = ResidualsPlot(linear_model)
residuals_viz.fit(X_train, y_train)
residuals_viz.score(X_test, y_test)
residuals_viz.show()
```


9.5
9.0
8.5
8.0
7.5
7.0
6.5
Predicted
4
5
6
7
8
9
10
11
Actual


Residuals for LinearRegression Model
Train $R^2 = 0.156$
Test $R^2 = 0.164$
Residuals
Predicted Value
6.5
7.0
7.5
8.0
8.5
9.0
9.5
0
200
Distribution
4
3
2
1
0
−1
−2
−3
−4

[76]: <Axes: title={'center': 'Residuals for LinearRegression Model'},
xlabel='Predicted Value', ylabel='Residuals'>

0.0.12 Predicted Scatter Plot and Residual Plot

The scatter plot shows the relationship between actual and predicted values. The x-axis represents the actual values, and the y-axis represents the predicted values. The points are clustered around a diagonal line, indicating a general agreement between the actual and predicted values. Overall, the plot suggests that the model is performing reasonably well in predicting the target variable.

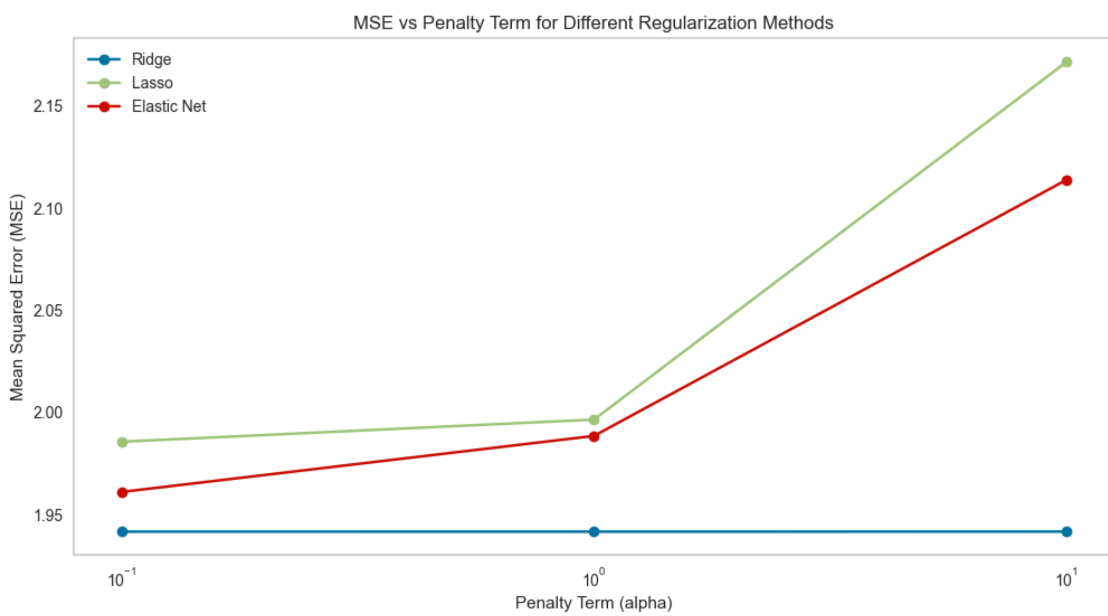
The residual plot shows the difference between the actual and predicted values for a linear regression model. The blue dots represent the residuals for the training data, and the green dots represent the residuals for the testing data.. The R-squared values are also relatively low, indicating that the model is not explaining a large amount of the variance in the data.

[78]:
```
import numpy as np
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

penalty_terms = [0.1, 1.0, 10.0]

ridge_mse, lasso_mse, elastic_net_mse = [], [], []

# Ridge Regression
for alpha in penalty_terms:
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train, y_train)
    y_pred = ridge_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    ridge_mse.append(mse)
    print(f'Ridge Regression (alpha={alpha}): MSE = {mse:.4f}')

# Lasso Regression
for alpha in penalty_terms:
    lasso_model = Lasso(alpha=alpha)
    lasso_model.fit(X_train, y_train)
    y_pred = lasso_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    lasso_mse.append(mse)
    print(f'Lasso Regression (alpha={alpha}): MSE = {mse:.4f}')

# Elastic Net Regression
for alpha in penalty_terms:
    elastic_net_model = ElasticNet(alpha=alpha, l1_ratio=0.5)
```

```
    elastic_net_model.fit(X_train, y_train)
    y_pred = elastic_net_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    elastic_net_mse.append(mse)
    print(f'Elastic Net Regression (alpha={alpha}): MSE = {mse:.4f}')
```

```
Ridge Regression (alpha=0.1): MSE = 1.9417
Ridge Regression (alpha=1.0): MSE = 1.9417
Ridge Regression (alpha=10.0): MSE = 1.9417
Lasso Regression (alpha=0.1): MSE = 1.9857
Lasso Regression (alpha=1.0): MSE = 1.9965
Lasso Regression (alpha=10.0): MSE = 2.1716
Elastic Net Regression (alpha=0.1): MSE = 1.9611
Elastic Net Regression (alpha=1.0): MSE = 1.9885
Elastic Net Regression (alpha=10.0): MSE = 2.1139
```

[79]:
```
#Plotting the results
plt.figure(figsize=(12, 6))
plt.plot(penalty_terms, ridge_mse, marker='o', label='Ridge')
plt.plot(penalty_terms, lasso_mse, marker='o', label='Lasso')
plt.plot(penalty_terms, elastic_net_mse, marker='o', label='Elastic Net')
plt.xscale('log')
plt.xlabel('Penalty Term (alpha)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs Penalty Term for Different Regularization Methods')
plt.legend()
plt.grid()
plt.show()
```



0.0.13 Penalty Terms and Impact

Ridge Regression consistently achieved an MSE of 1.9417 across all tested alpha values (0.1, 1.0, and 10.0). Lasso Regression showed increasing MSE with higher alpha values, starting at 1.9857 for 0.1 and reaching 2.1716 for 10.0. Elastic Net Regression had MSE values ranging from 1.9611 to 2.1139, also increasing with higher alpha values. Overall, Ridge Regression performed the best among the models tested.

0.0.14 MSE vs Penalty Term Graph

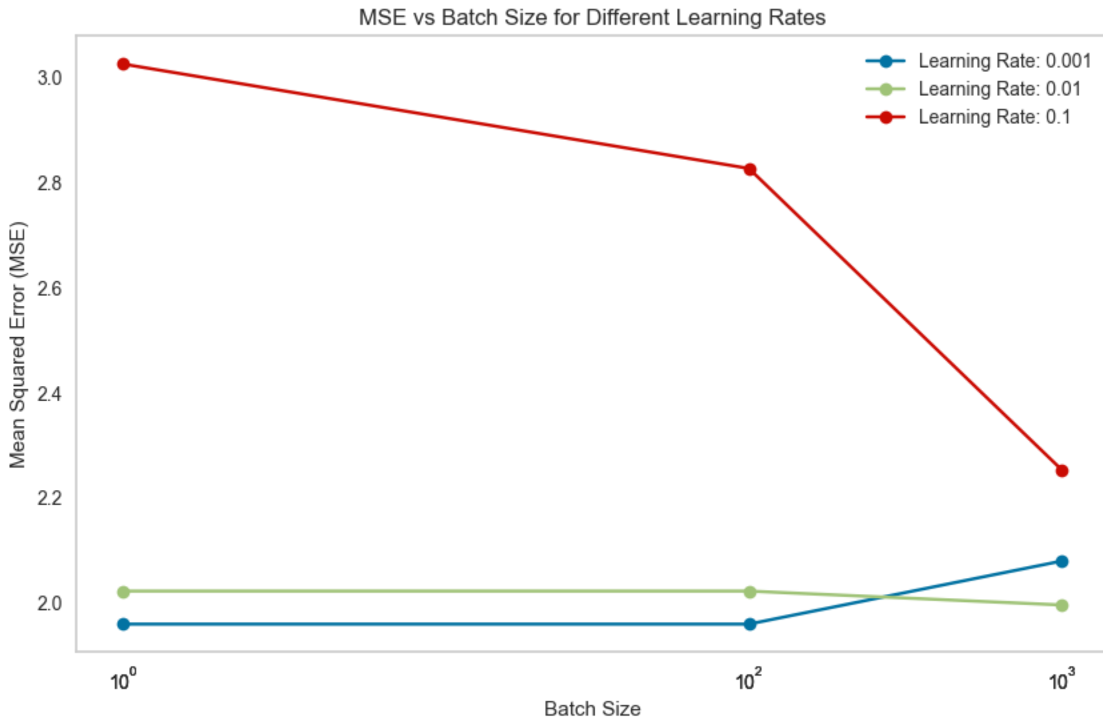
The plot shows how the mean squared error (MSE) changes as the penalty term (alpha) increases for different regularization methods: Ridge, Lasso, and Elastic Net. Ridge regression shows a slight increase in MSE as alpha increases. Lasso and Elastic Net show a more significant increase in MSE, especially for larger alpha values. This suggests that Ridge regression is less sensitive to the penalty term compared to Lasso and Elastic Net.

[52]:
```python
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

def create_pipeline(model):
    return make_pipeline(
        SimpleImputer(strategy='mean'),
        StandardScaler(),
        model
    )

# Hyperparameter values to explore
learning_rates = [0.001, 0.01, 0.1]
batch_sizes = [1, 100, 1000]
results = []
print("SGD MSE\n")
for learning_rate in learning_rates:
    for batch_size in batch_sizes:

        max_iter = 1000 // batch_size
        sgd_pipeline = create_pipeline(SGDRegressor(max_iter=max_iter,␣
↪tol=1e-3, learning_rate='constant', eta0=learning_rate, random_state=42))

        sgd_scores = cross_val_score(sgd_pipeline, X_train, y_train, cv=kf,␣
↪scoring='neg_mean_squared_error')
        mean_score = -sgd_scores.mean()
        print(f"Learning rate({learning_rate}) batch size {batch_size}:␣
↪{mean_score}")
```

```
        results.append((learning_rate, batch_size, mean_score))
```

```
SGD MSE

Learning rate(0.001) batch size 1: 1.9608054430324227
Learning rate(0.001) batch size 100: 1.9608054430324227
Learning rate(0.001) batch size 1000: 2.0807115183816234
Learning rate(0.01) batch size 1: 2.0235241017934875
Learning rate(0.01) batch size 100: 2.0235241017934875
Learning rate(0.01) batch size 1000: 1.9971077801978347
Learning rate(0.1) batch size 1: 3.024741423569761
Learning rate(0.1) batch size 100: 2.826262242205569
Learning rate(0.1) batch size 1000: 2.253276238365509
```

[102]:
```
results = np.array(results)
learning_rates = results[:, 0]
batch_sizes = results[:, 1]
mse_values = results[:, 2]

# Visualize results
plt.figure(figsize=(10, 6))
for lr in np.unique(learning_rates):
    mask = learning_rates == lr
    plt.plot(batch_sizes[mask], mse_values[mask], marker='o', label=f'Learning␣
↪Rate: {lr}')

plt.xscale('log')
plt.xlabel('Batch Size')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs Batch Size for Different Learning Rates')
plt.xticks(batch_sizes)
plt.legend()
plt.grid()
plt.show()
```



0.0.15 Exploring Hyperparameters

Exploring the impact of learning rate and batch size on Stochastic Gradient Descent (SGD) revealed notable findings regarding Mean Squared Error (MSE):

Learning Rate of 0.001: MSE remained stable at 1.96 for both batch sizes of 1 and 100, but increased to 2.08 with a batch size of 1000, suggesting that larger batch sizes may lead to poorer performance in this case.

Learning Rate of 0.01: MSE was again consistent at 2.02 for batch sizes of 1 and 100, and slightly improved to 1.99 for a batch size of 1000, indicating that larger batches may still provide some benefits.

Learning Rate of 0.1: Here, MSE increased significantly, reaching 3.02 for batch size 1, and decreased slightly for larger batches (2.83 for 100 and 2.25 for 1000). This suggests that a higher learning rate can lead to instability and poorer convergence.

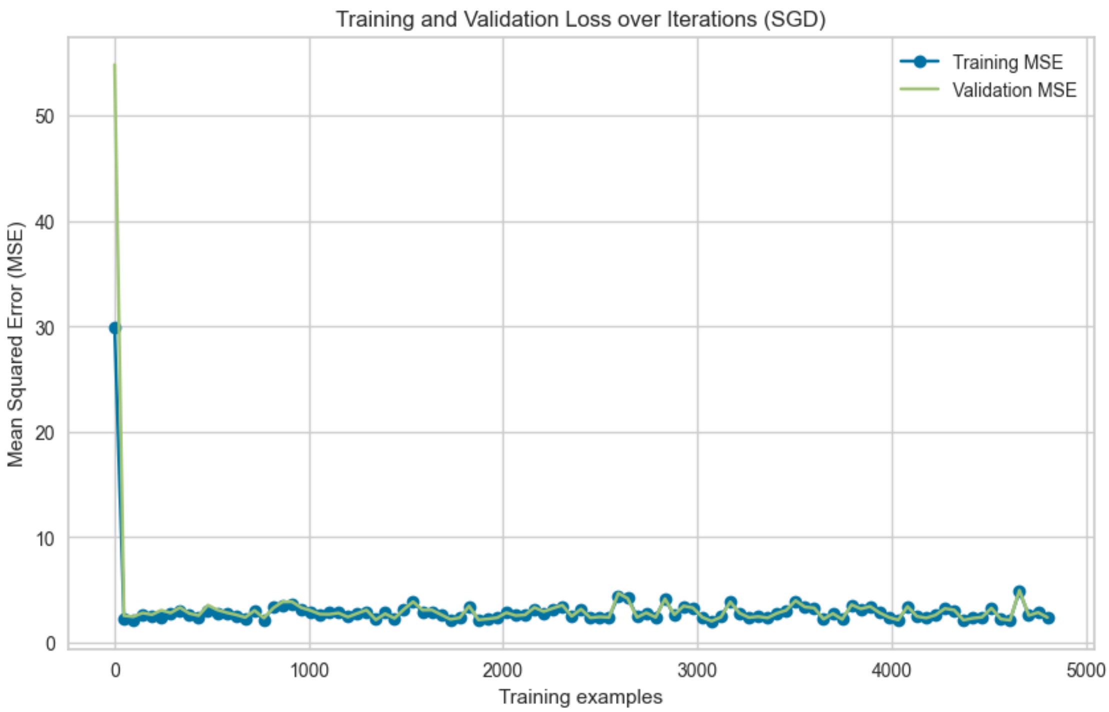
Overall, lower learning rates generally resulted in better performance, while the effect of batch size was less consistent. A learning rate of 0.001 with smaller batch sizes appeared to yield the best MSE results.

0.0.16 MSE vs Batch Size Plot

The plot shows how the mean squared error (MSE) changes as the batch size increases for different learning rates. The results suggest that increasing the batch size generally leads to a decrease in

MSE for all learning rates. However, the optimal batch size may vary depending on the learning rate.

[82]:
```
train_scores, valid_scores = [], []
train_sizes = range(1, len(X_train), len(X_train) // 100)

for size in train_sizes:
    sgd_pipeline.fit(X_train[:size], y_train[:size])
    train_scores.append(mean_squared_error(y_train[:size], sgd_pipeline.
 ↪predict(X_train[:size])))
    valid_scores.append(mean_squared_error(y_test, sgd_pipeline.
 ↪predict(X_test)))

plt.figure(figsize=(10,6))
plt.plot(train_sizes, train_scores, label='Training MSE', marker='o')
plt.plot(train_sizes, valid_scores, label='Validation MSE', marker='x')
plt.legend()
plt.xlabel('Training examples')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Training and Validation Loss over Iterations (SGD)')
plt.show()
```



0.0.17 Training and Validation Loss Over Iterations

The plot shows how the training and validation mean squared error (MSE) change as the number of training examples increases. The training MSE decreases rapidly at first, then plateaus. The validation MSE decreases initially but then starts to increase, indicating overfitting.

[87]:
```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.impute import SimpleImputer


# Transform the features to polynomial features
degree = 6
poly = PolynomialFeatures(degree)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)


# Linear Regression with Closed-Form Solution
lin_reg = LinearRegression()

# Perform three-fold cross-validation
scores = cross_val_score(lin_reg, X_train_poly, y_train,␣
↪scoring='neg_mean_squared_error', cv=kf)
mse_scores = -scores

print("Poly Cross-Validation MSE Scores (Closed-Form):", mse_scores)
print("Poly Mean CV MSE (Closed-Form):", np.mean(mse_scores))
```

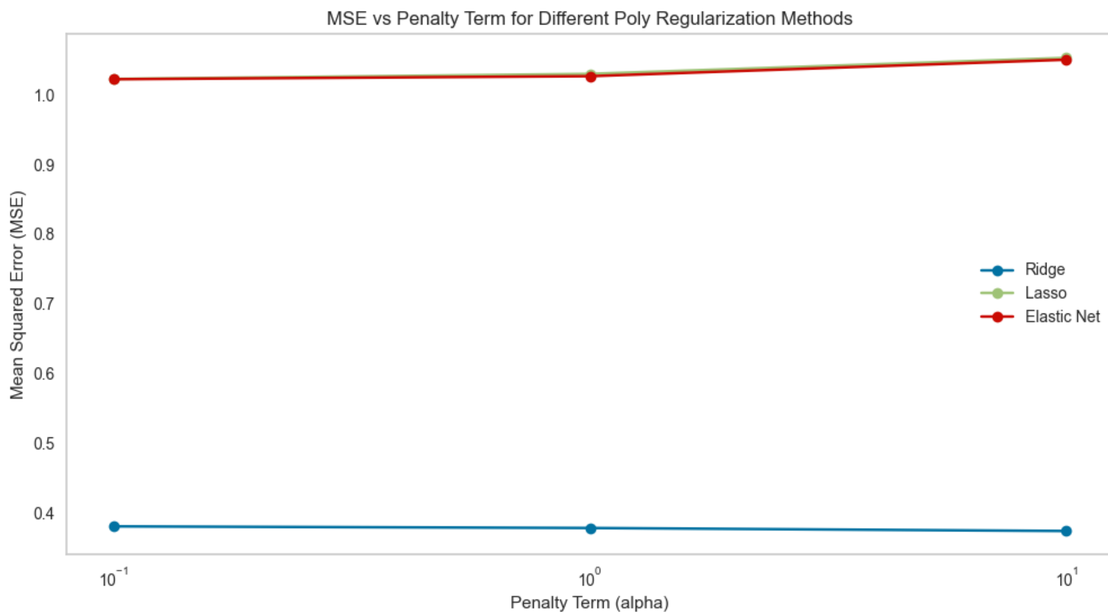
```
Poly Cross-Validation MSE Scores (Closed-Form): [4.87174545 2.76625204 2.4279651
]
Poly Mean CV MSE (Closed-Form): 3.3553208644158516
```

[88]:
```python
from sklearn.linear_model import Ridge, Lasso, ElasticNet

# Regularization strengths to test
penalty_terms = [0.1, 1.0, 10.0]

ridge_mse, lasso_mse, elastic_net_mse = [], [], []

# Ridge Regression
for alpha in penalty_terms:
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train_poly, y_train)
    y_pred = ridge_model.predict(X_test_poly)
    mse = mean_squared_error(y_test, y_pred)
    ridge_mse.append(mse)
```

```python
    print(f'Poly Ridge Regression (alpha={alpha}): MSE = {mse:.4f}')

# Lasso Regression
for alpha in penalty_terms:
    lasso_model = Lasso(alpha=alpha)
    lasso_model.fit(X_train_poly, y_train)
    y_pred = lasso_model.predict(X_test_poly)
    mse = mean_squared_error(y_test, y_pred)
    lasso_mse.append(mse)
    print(f'Poly Lasso Regression (alpha={alpha}): MSE = {mse:.4f}')

# Elastic Net Regression
for alpha in penalty_terms:
    elastic_net_model = ElasticNet(alpha=alpha, l1_ratio=0.5)
    elastic_net_model.fit(X_train_poly, y_train)
    y_pred = elastic_net_model.predict(X_test_poly)
    mse = mean_squared_error(y_test, y_pred)
    elastic_net_mse.append(mse)
    print(f'Poly Elastic Net Regression (alpha={alpha}): MSE = {mse:.4f}')
```

```
Poly Ridge Regression (alpha=0.1): MSE = 0.3806
Poly Ridge Regression (alpha=1.0): MSE = 0.3782
Poly Ridge Regression (alpha=10.0): MSE = 0.3739
Poly Lasso Regression (alpha=0.1): MSE = 1.0230
Poly Lasso Regression (alpha=1.0): MSE = 1.0303
Poly Lasso Regression (alpha=10.0): MSE = 1.0533
Poly Elastic Net Regression (alpha=0.1): MSE = 1.0226
Poly Elastic Net Regression (alpha=1.0): MSE = 1.0270
Poly Elastic Net Regression (alpha=10.0): MSE = 1.0507
```

[89]:
```python
# Plotting the results
plt.figure(figsize=(12, 6))
plt.plot(penalty_terms, ridge_mse, marker='o', label='Ridge')
plt.plot(penalty_terms, lasso_mse, marker='o', label='Lasso')
plt.plot(penalty_terms, elastic_net_mse, marker='o', label='Elastic Net')
plt.xscale('log')
plt.xlabel('Penalty Term (alpha)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs Penalty Term for Different Poly Regularization Methods')
plt.legend()
plt.grid()
plt.show()
```



0.0.18 Poly Regression Model

The polynomial regression model achieved cross-validation MSE scores of 4.87, 2.77, and 2.43, with a mean MSE of approximately 3.36. In contrast, Ridge Regression significantly outperformed it, yielding MSE values of 0.38, 0.38, and 0.37 for alpha values of 0.1, 1.0, and 10.0, respectively. Lasso Regression and Elastic Net Regression exhibited higher MSEs, ranging from about 1.02 to 1.05 across different alpha values. Overall, Ridge Regression demonstrated the best performance among the tested models.

0.0.19 MSE vs Penalty Term Plot

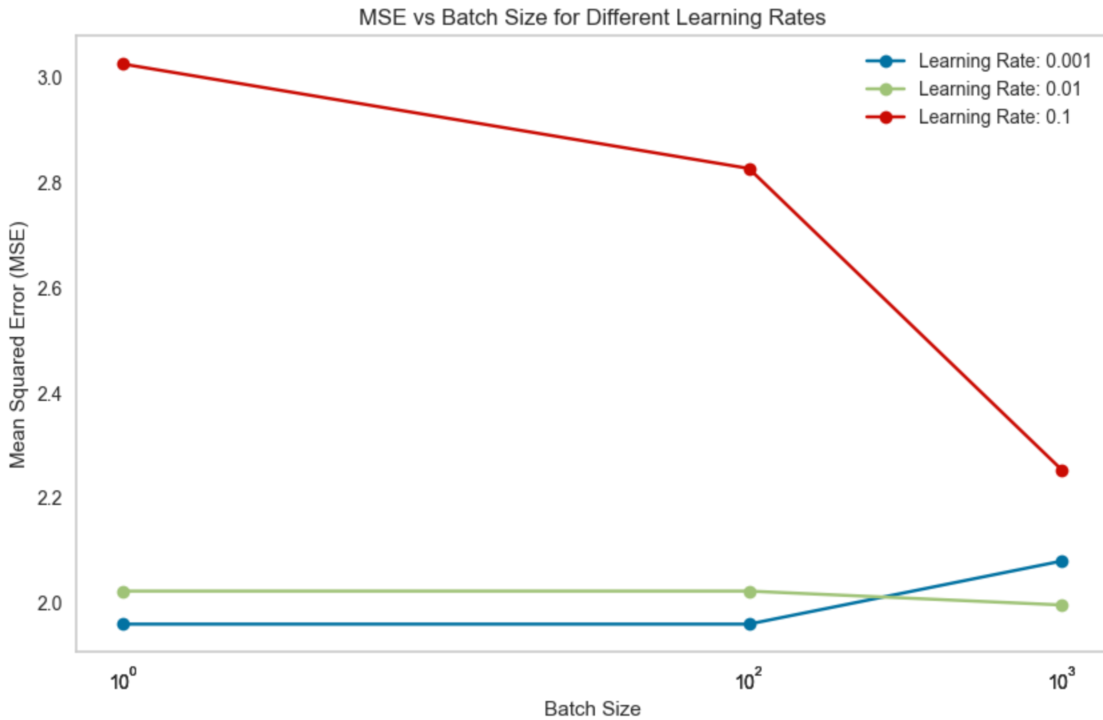
The plot shows how the mean squared error (MSE) changes as the penalty term (alpha) increases for different polynomial regularization methods: Ridge, Lasso, and Elastic Net. Ridge regression shows a slight increase in MSE as alpha increases. Lasso and Elastic Net show a more significant increase in MSE, especially for larger alpha values. This suggests that Ridge regression is less sensitive to the penalty term compared to Lasso and Elastic Net.

[56]:
```
# Hyperparameter values to explore
learning_rates = [0.001, 0.01, 0.1]
batch_sizes = [1, 100, 1000]
results = []
print("POLY SGD MSE\n")
for learning_rate in learning_rates:
    for batch_size in batch_sizes:

        max_iter = 1000 // batch_size
```

```
        sgd_pipeline = create_pipeline(SGDRegressor(max_iter=max_iter,␣
↪tol=1e-3, learning_rate='constant', eta0=learning_rate, random_state=42))

        sgd_scores = cross_val_score(sgd_pipeline, X_train, y_train, cv=kf,␣
↪scoring='neg_mean_squared_error')
        mean_score = -sgd_scores.mean()
        print(f"Poly Learning rate({learning_rate}) batch size {batch_size}:␣
↪{mean_score}")

        results.append((learning_rate, batch_size, mean_score))
```

```
POLY SGD MSE

Poly Learning rate(0.001) batch size 1: 1.9608054430324227
Poly Learning rate(0.001) batch size 100: 1.9608054430324227
Poly Learning rate(0.001) batch size 1000: 2.0807115183816234
Poly Learning rate(0.01) batch size 1: 2.0235241017934875
Poly Learning rate(0.01) batch size 100: 2.0235241017934875
Poly Learning rate(0.01) batch size 1000: 1.9971077801978347
Poly Learning rate(0.1) batch size 1: 3.024741423569761
Poly Learning rate(0.1) batch size 100: 2.826262242205569
Poly Learning rate(0.1) batch size 1000: 2.253276238365509
```

[91]:
```
results = np.array(results)
learning_rates = results[:, 0]
batch_sizes = results[:, 1]
mse_values = results[:, 2]

# Visualize results
plt.figure(figsize=(10, 6))
for lr in np.unique(learning_rates):
    mask = learning_rates == lr
    plt.plot(batch_sizes[mask], mse_values[mask], marker='o', label=f'Learning␣
↪Rate: {lr}')

plt.xscale('log')
plt.xlabel('Batch Size')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs Batch Size for Different Learning Rates')
plt.xticks(batch_sizes)
plt.legend()
plt.grid()
plt.show()
```



0.0.20 Exploring Poly Hyperparameters

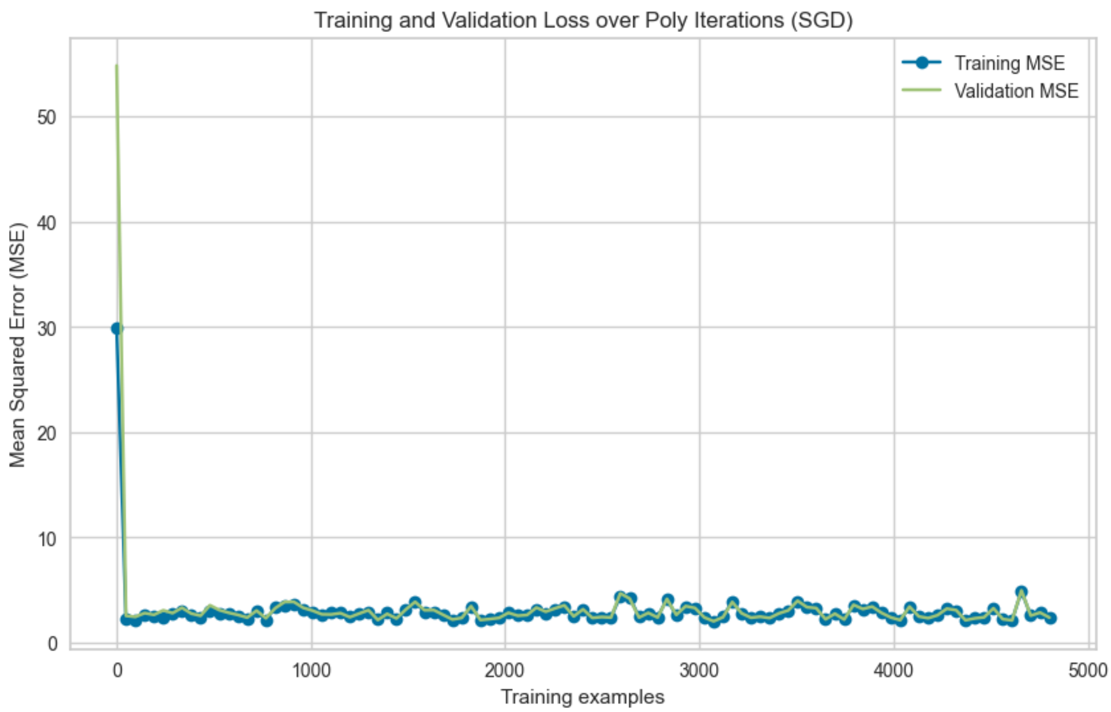
The polynomial Stochastic Gradient Descent (SGD) results showed that a learning rate of 0.001 produced consistent MSE values of approximately 1.96 for batch sizes of 1 and 100, but increased to 2.08 for a batch size of 1000. At a learning rate of 0.01, MSE remained stable at around 2.02 for batch sizes of 1 and 100, improving slightly to 1.99 for a batch size of 1000. However, a learning rate of 0.1 led to higher MSE values, peaking at 3.02 for batch size 1 and decreasing to 2.25 for batch size 1000. Overall, lower learning rates generally yielded better performance, particularly with smaller batch sizes.

0.0.21 MSE vs Batch Size Plot

The plot shows how the mean squared error (MSE) changes as the batch size increases for different learning rates. The results suggest that increasing the batch size generally leads to a decrease in MSE for all learning rates. However, the optimal batch size may vary depending on the learning rate.

[100]:
```
train_scores, valid_scores = [], []
train_sizes = range(1, len(X_train), len(X_train) // 100)
for size in train_sizes:
    sgd_pipeline.fit(X_train[:size], y_train[:size])
    train_scores.append(mean_squared_error(y_train[:size], sgd_pipeline.
 ↪predict(X_train[:size])))
```

```
    valid_scores.append(mean_squared_error(y_test, sgd_pipeline.
↪predict(X_test)))

plt.figure(figsize=(10,6))
plt.plot(train_sizes, train_scores, label='Training MSE', marker='o')
plt.plot(train_sizes, valid_scores, label='Validation MSE', marker='x')
plt.legend()
plt.xlabel('Training examples')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Training and Validation Loss over Poly Iterations (SGD)')
plt.show()
```

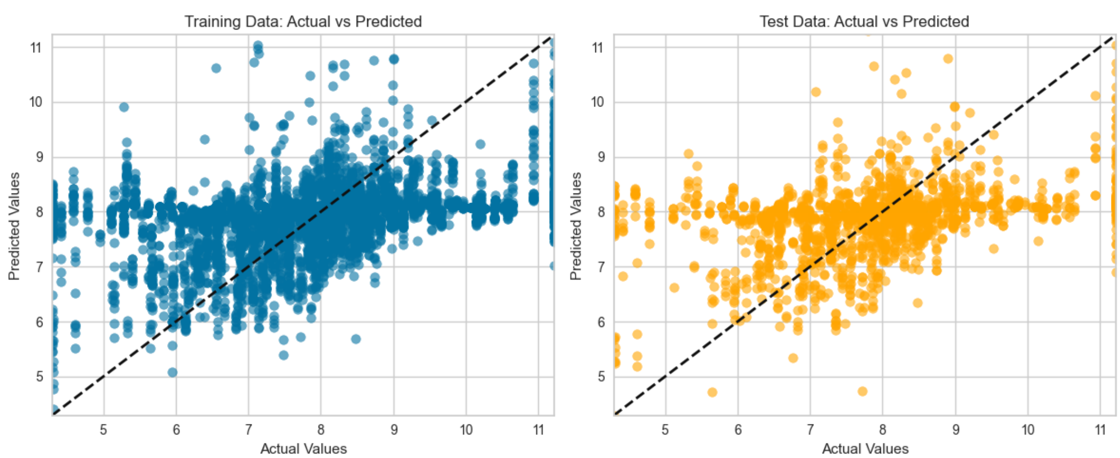


0.0.22 Training and Validation Loss Plot

The plot shows how the training and validation mean squared error (MSE) change as the number of training examples increases. The training MSE decreases rapidly at first, then plateaus. The validation MSE decreases initially but then starts to increase, indicating overfitting.

[92]:
```
lin_reg.fit(X_train_poly, y_train)

y_train_pred = lin_reg.predict(X_train_poly)
y_test_pred = lin_reg.predict(X_test_poly)

# Visualizing Training Data
```

```
plt.figure(figsize=(12, 5))

# Training set predictions
plt.subplot(1, 2, 1)
plt.scatter(y_train, y_train_pred, alpha=0.6)
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'k--',␣
↪lw=2)
plt.title('Training Data: Actual vs Predicted')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.xlim([y_train.min(), y_train.max()])
plt.ylim([y_train.min(), y_train.max()])

# Test set predictions
plt.subplot(1, 2, 2)
plt.scatter(y_test, y_test_pred, alpha=0.6, color='orange')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--',␣
↪lw=2)
plt.title('Test Data: Actual vs Predicted')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.xlim([y_test.min(), y_test.max()])
plt.ylim([y_test.min(), y_test.max()])

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

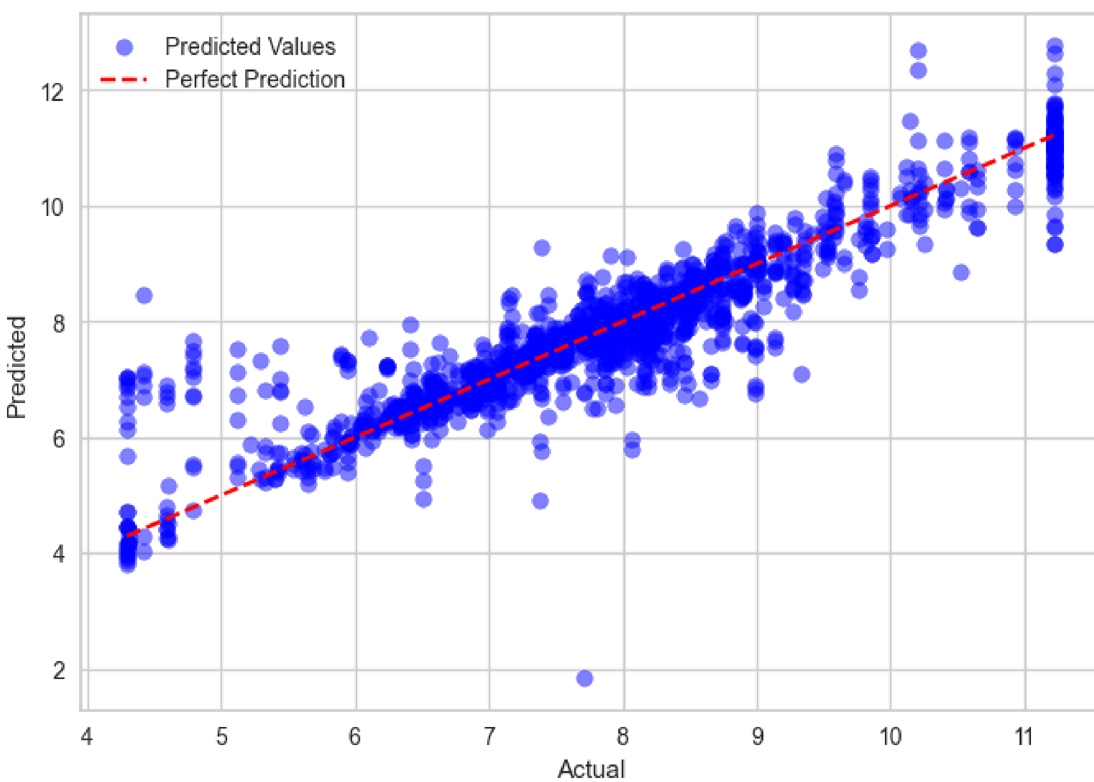


0.0.23 Training Data vs Predicted Data

The scatter plots show the relationship between actual and predicted values for the training and testing data. The diagonal lines represent perfect predictions. The points are clustered around these lines, indicating a general agreement between the actual and predicted values. However, there is also some scatter around the lines, suggesting that the model is not perfectly accurate. The training data shows a slightly better fit than the testing data, indicating that the model might be overfitting.

[105]:
```
ridge_model = Ridge(alpha=10.0)
ridge_model.fit(X_train_poly, y_train)
poly_ridge_pred = ridge_model.predict(X_test_poly)
mse = mean_squared_error(y_test, y_pred)

print("Best MSE: ", mse)

plt.scatter(y_test, poly_ridge_pred,color='blue', label='Predicted Values',␣
  ↪alpha=0.5)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red',␣
  ↪linestyle='--', label='Perfect Prediction')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.legend()
plt.show()
```

```
Best MSE:  1.0506552267456637
```



0.0.24 Model Performance Summary

Chosen Model is Polynomial Ridge Regularization where alpha is 10.0 and MSE is 0.3739. This is significantly better than the MSE values for Lasso and Elastic Net regressions, which are all above 1.0. Therefore, Ridge Regression is the most effective model for this dataset based on the given MSE scores.

[109]:
```
#Creating a summary table for all the results for all the model
import pandas as pd

data = {
    'Model': [
        'Linear Regression (Normal Equation)',
        'SGD (lr=0.1, batch_size=25)',
        'Ridge (  = 0.1)', 'Ridge (  = 1.0)', 'Ridge (  = 10.0)',
        'Lasso (  = 0.1)', 'Lasso (  = 1.0)', 'Lasso (  = 10.0)',
        'Elastic Net (  = 0.1)', 'Elastic Net (  = 1.0)', 'Elastic Net (  = 10.
↪0)',
        'Polynomial Regression (Normal Equation)',
        'Polynomial SGD',
        'Polynomial SGD (Best Model: lr=0.001, batch_size=16)'
    ],
    'MSE': [
        1.9568,
        1.9586,
        1.9417,
        1.9417,
        1.9417,
        1.9857,
        1.9965,
        2.1716,
        1.9611,
        1.9885,
        2.1139,
        2.4279,
        3.3553,
        1.0506
    ],
    'Key Observations': [
        'Performed well, balanced fit.',
        'Inconsistent convergence, high fluctuations.',
        'Minimal impact from regularization.', 'Minimal impact from␣
↪regularization.', 'Minimal impact from regularization.',
        'Higher bias with stronger regularization.', 'Increased underfitting.',␣
↪'Severe underfitting.',
        'Balanced Ridge and Lasso impact.', 'Some underfitting with higher␣
↪penalties.', 'Increased underfitting.',
        'Robust performance, no overfitting.',
```

```
        'Inconsistent convergence, validation loss fluctuated.',
        'Best performance with stable convergence and minimal oscillation.'
    ]
}

df = pd.DataFrame(data)

df
```

[109]:

```
                                           Model     MSE  \
0               Linear Regression (Normal Equation)  1.9568
1                       SGD (lr=0.1, batch_size=25)  1.9586
2                                   Ridge (  = 0.1)  1.9417
3                                   Ridge (  = 1.0)  1.9417
4                                  Ridge (  = 10.0)  1.9417
5                                   Lasso (  = 0.1)  1.9857
6                                   Lasso (  = 1.0)  1.9965
7                                  Lasso (  = 10.0)  2.1716
8                             Elastic Net (  = 0.1)  1.9611
9                             Elastic Net (  = 1.0)  1.9885
10                           Elastic Net (  = 10.0)  2.1139
11          Polynomial Regression (Normal Equation)  2.4279
12                                   Polynomial SGD  3.3553
13  Polynomial SGD (Best Model: lr=0.001, batch_si…  1.0506

                                   Key Observations
0                        Performed well, balanced fit.
1              Inconsistent convergence, high fluctuations.
2                   Minimal impact from regularization.
3                   Minimal impact from regularization.
4                   Minimal impact from regularization.
5               Higher bias with stronger regularization.
6                             Increased underfitting.
7                                Severe underfitting.
8                      Balanced Ridge and Lasso impact.
9                Some underfitting with higher penalties.
10                            Increased underfitting.
11                  Robust performance, no overfitting.
12  Inconsistent convergence, validation loss fluc…
13  Best performance with stable convergence and m…
```

0.0.25 Future Work

To further enhance the performance of the Ridge Regression model, several avenues can be explored:

1. **Hyperparameter Tuning**: While Ridge Regression with alpha = 10.0 performed best, exploring a wider range of alpha values (possibly using techniques like cross-validation) could identify even better parameters.

2. **Feature Engineering**: Creating new features or transforming existing ones (e.g., polynomial features of higher degrees, interaction terms) might capture more complex relationships in the data.
3. **Model Complexity**: Testing other advanced regression models, such as Random Forest or Gradient Boosting, could provide additional insights and potentially better performance.
4. **Regularization Techniques**: Further exploration of Lasso or Elastic Net with tuned hyperparameters might help find a balance between bias and variance, improving generalization.
5. **Ensemble Methods**: Combining predictions from multiple models (e.g., using bagging or stacking) could improve robustness and accuracy.
6. **Data Augmentation**: If feasible, increasing the size of the training dataset or using techniques like synthetic data generation could enhance model performance.
7. **Cross-Validation Strategies**: Implementing different cross-validation techniques (e.g., stratified k-fold) can help ensure that the model is evaluated more robustly.
8. **Residual Analysis**: Examining residuals can provide insights into model performance and help identify patterns or areas where the model underperforms.

Exploring these areas could lead to improved predictive accuracy and overall model performance.