Used Car Price Prediction

November 23, 2020

1 Title: Analyzing effectiveness of Linear Regression and Lasso Regression ML Techniques for Used Car Price Prediction

2 Project Introduction

- The aim of this project is to utilize Machine learning techniques to optimally predict used car prices.
- This project utilizes the data fetched from Kaggle. The link for the data source is https://www.kaggle.com/adityadesai13/used-car-dataset-ford-and-mercedes?select=bmw.csv
- In specific, prices of BMW used cars at United Kingdom from the last 10 years was utilized to train the ML model, predict prices for used cars.

Key Steps and Analysis made in this Project * In this exercise, the features & sample count of the data were optimized to reduce the validation error. Also two different regression techniques were performed to compare the pros and cons of the regression methods * As the first step, the data was plotted and visualized to see the correlation between the features and label * Also, the categories (string values) in the data were label encoded and one-hot encoded, so that the ML methods leverage the data correctly * Linear Regression is performed with varying features & varying samples. From the optimal combination identified for the features and samples, the regression was repeated with optimal sample count and optimal feature count. As expected, this achieved the least mean squared error * Finally, Lasso regression was performed on this data. This regression was able to optimize the co-efficients but this optimization was achieved at the cost of a little increase in the error value

3 Problem Formulation

- As explained above, the datapoints are the used car prices data for the last 10 years

1. Features used in this exercise: 1. car model, 2. year of manufacture, 3. mileage, 4. engine size, 5. transmission type (automatic / manual) and 6. fuel type

2. Labels: The label is the price of the used car

3. Metrics and Method used 1. The Metrics used is the Mean Squared error of the training and validation data and optimized weight vectors. These errors were analyzed for different number of features and for different sample counts to find the optimal combination. 2. In addition to Linear Regression, regularization technique such as Lasso regression was performed to shrink the co-efficients and thus reduce the number of features used in this exercise. This is especially relevant

for this exercise considering that the one hot encoding significantly increased the number of features 3. The difference in Validation Error between Linear Regression and Lasso Regression is presented

4. Predictors used: 1. Linear Regression from sklearn library 2. Lasso Regression from sklearn library

4 Method

Pre-processing Techniques:

- One Hot Encoding
- Visualization by Plots

Machine Learning Algorithms used

- Linear Regression with Varying Features
- Linear Regression with Varying Sample Count (revealed overfitting in case of higher samples)
- Linear Regression with optimal combination of Features and Sample Count
- Lasso Regression to optimize the weight vectors

Loss Functions used

- Mean Squared Error
- Also complexity reduction by simpler weight vectors was also evaluated

Summary

- Pros and Cons of various techniques were summarized

[33]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

PRE-PROCESSING: Module to perform onehot encoding for string type of data elements

[34]:
```
def onehotencode(orig_df, feature_colname):
    dummies = pd.get_dummies(orig_df[[feature_colname]])
    updated_df = pd.concat([orig_df, dummies], axis=1)
    updated_df = updated_df.drop([feature_colname], axis=1)
    return(updated_df)
```

Module to parse the data to fetch configurable number of features.

- Also, one hot encoding is performed by invoking the module above.
- One hot encoding was performed for model category, transmission category and fuelType category

[35]:
```
# maximum n value is 29
def GetFeaturesLabels(sampleCount=10000, featureCount=29):
    if (featureCount > 29):
        featureCount = 29
    file = "../../../notebooks/
 ↪bmw_refined_fuel_transmission_ToBeUsedForEncoding.csv"
    df = pd.read_csv(file)
    df = onehotencode(df,'model')
    df = onehotencode(df,'transmission')
    df = onehotencode(df,'fuelType')
    X = df.values
    X1=X[:sampleCount,1:featureCount]
    y1=X[:sampleCount,0]
    return (X1,y1)
```

Generation of feature matrix X and label vector y

[36]:
```
X,y = GetFeaturesLabels(featureCount=29)
print(X)
print(y)
X.shape
```

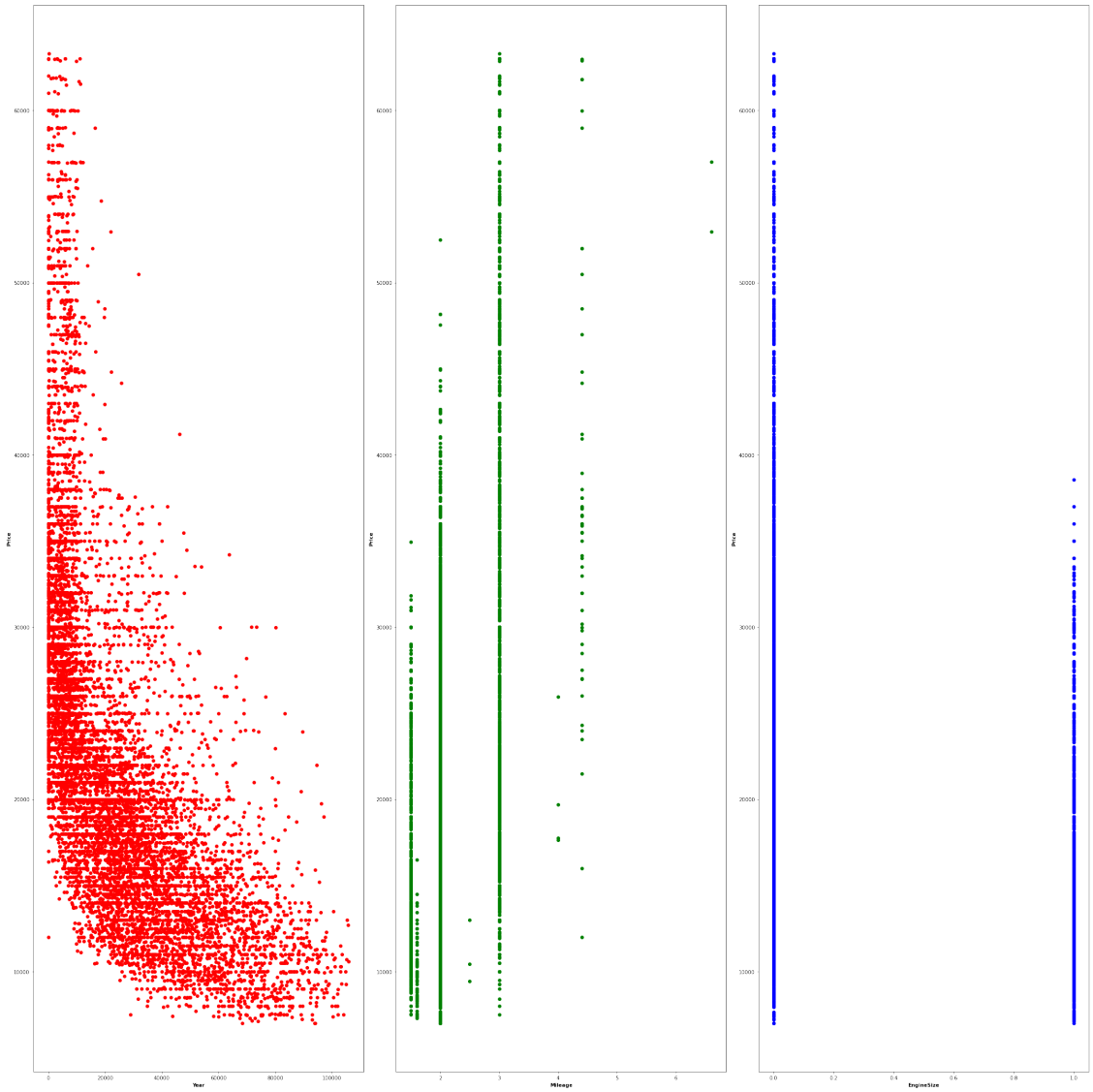
```
[[2.0140e+03 6.7068e+04 2.0000e+00 … 0.0000e+00 1.0000e+00 0.0000e+00]
 [2.0180e+03 1.4827e+04 2.0000e+00 … 0.0000e+00 0.0000e+00 1.0000e+00]
 [2.0160e+03 6.2794e+04 3.0000e+00 … 0.0000e+00 1.0000e+00 0.0000e+00]
 …
 [2.0160e+03 3.3500e+04 3.0000e+00 … 0.0000e+00 1.0000e+00 0.0000e+00]
 [2.0150e+03 3.9000e+04 2.0000e+00 … 0.0000e+00 1.0000e+00 0.0000e+00]
 [2.0160e+03 1.9500e+04 3.0000e+00 … 0.0000e+00 0.0000e+00 1.0000e+00]]
[11200. 27000. 16000. … 32000. 15000. 24995.]
```

[36]:
```
(10000, 28)
```

Visualization - First few features (Year, Mileage and Engine size) are plotted to see the correlation to the label (price)

[37]:
```
fig ,axes= plt.subplots(1, 3, figsize=(30, 30))
x1 = X[:,1]
x2 = X[:,2]
x3 = X[:,3]
y1 = y
axes[0].scatter(x1,y1,c='r')
axes[1].scatter(x2,y1,c='g')
axes[2].scatter(x3,y1,c='b')
axes[0].set_xlabel('Year',fontweight='bold')
axes[1].set_xlabel('Mileage',fontweight='bold')
axes[2].set_xlabel('EngineSize',fontweight='bold')
axes[0].set_ylabel('Price',fontweight='bold')
```

```
axes[1].set_ylabel('Price',fontweight='bold')
axes[2].set_ylabel('Price',fontweight='bold')
fig.tight_layout()
plt.show()
```



4.0.1 Training and Validation Error computed for VARYING NUMBER OF FEATURES

[38]:
```
#Error Calculation with variable number of features
# maximum number of features used
max_features = 29
# read in all data points using max_features
X, y = GetFeaturesLabels(max_features)
```

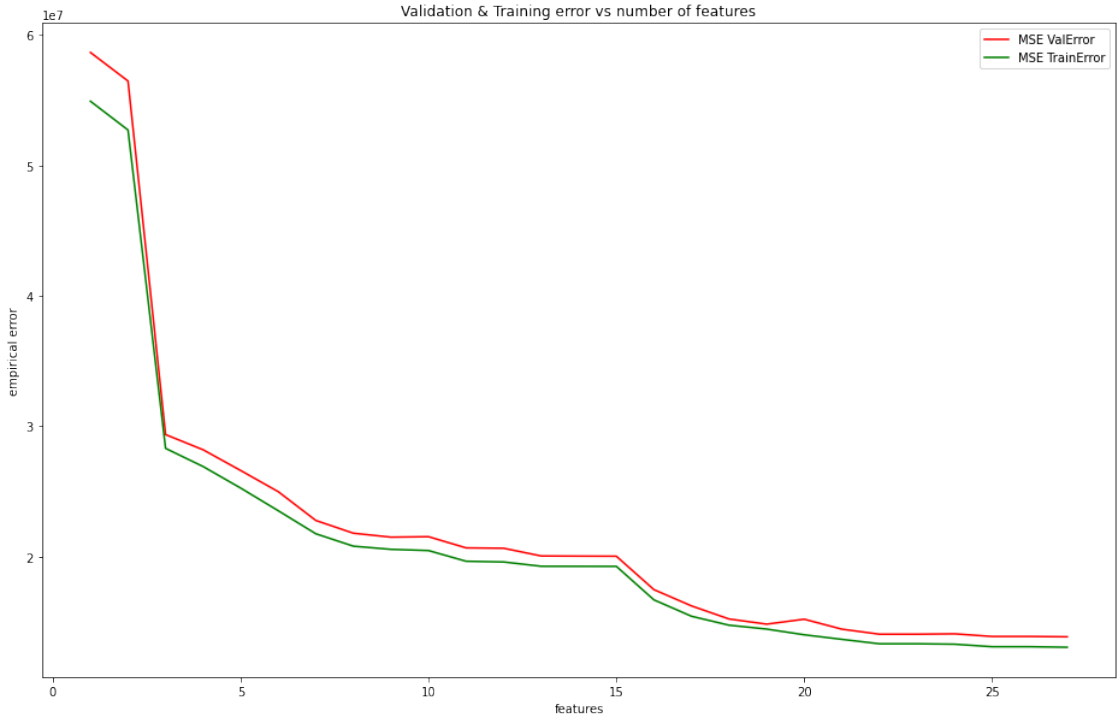
```
# vector for storing the training error of LinearRegresion.fit() for each␣
 ↪number of feature
linregVal_error = np.zeros(max_features-2)
linregTrain_error = np.zeros(max_features-2)

for i in range(2, max_features):
    X, y = GetFeaturesLabels(featureCount= i)
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
    reg = LinearRegression(fit_intercept=True)
    reg = reg.fit(X_train, y_train)
    y_pred = reg.predict(X_val)
    linregVal_error[i-2] = mean_squared_error(y_val, y_pred)
    y_pred1 = reg.predict(X_train)
    linregTrain_error[i-2] = mean_squared_error(y_train, y_pred1)


# create a numpy array "r_values" containing the values 1,2...,max_r
r_values = np.linspace(1, max_features-2, max_features-2, endpoint=True)
# create a plot object which can be accessed using variables "fig" and "axes"
fig, axes = plt.subplots(1,1, figsize=(16, 10))
# add a curve representing the average squared error for each choice of r
axes.plot(r_values, linregVal_error, label='MSE ValError', color='red')
axes.plot(r_values, linregTrain_error, label='MSE TrainError', color='green')
# add captions for the horizontal and vertical axes
axes.set_xlabel('features')
axes.set_ylabel('empirical error')
# add a title to the plot
axes.set_title('Validation & Training error vs number of features')
axes.legend()
print("---Validation Error-----")
print(linregVal_error)
print("---Training Error-----")
print(linregTrain_error)
```

```
---Validation Error-----
[58656459.69968098 56467544.63169807 29354203.80988351 28188999.29437372
 26598223.21913239 24975965.77102366 22768669.04758659 21791438.64475378
 21489226.55649963 21524227.3190611  20666272.52665377 20637623.83025271
 20045742.68113966 20033839.92570095 20025639.88557047 17464582.67188605
 16216452.17377208 15213036.34770644 14818596.55491561 15196053.94165227
 14435296.84484158 14048072.57915442 14048072.57915449 14075947.18444996
 13874228.65680484 13874228.65680483 13848564.06340784]
---Training Error-----
[54916584.23031607 52710709.17783149 28298417.75557736 26902611.09157137
 25251761.37095318 23514592.56525482 21748292.67782405 20796986.5919282
 20554199.26760072 20460612.59290744 19640846.78476578 19585666.28682847
```

```
19256495.97964295 19253238.63694704 19249290.94947645 16679478.43632634
15423089.73265694 14736670.21190295 14441649.00743674 14002018.84817875
13653822.95544583 13313051.94340781 13313051.9434078  13280414.03546568
13087397.82139455 13087397.82139456 13044503.46932707]
```



Analysis of Result

- It could be seen that the Training Error is lesser than the Validation Error.
- In this case, all the features count (i.e., are useful) and thus the error decreases as the number of features increase. This is expected because all the chosen features influence the car price.

4.0.2 Training and Validation Error computed for VARYING NUMBER OF SAMPLES

[39]:
```
#Error Calculation with variable number of features
# maximum number of features used
max_sampleset = 10
# read in all data points using max_r features
X, y = GetFeaturesLabels() #No arguments will get all samples and all features
# vector for storing the training error of LinearRegresion.fit() for each r
linregVal_error = np.zeros(max_sampleset)
linregTrain_error = np.zeros(max_sampleset)
samplesUsed = [1000,2000,3000,4000,5000,6000,7000,8000,9000,9918]
```

```
for i in range(1, max_sampleset+1):
    indexRange = samplesUsed[i-1]
    X1 = X[:indexRange]
    y1 = y[:indexRange]
    X_train, X_val, y_train, y_val = train_test_split(X1, y1, test_size=0.2,␣
 ↪random_state=2)
    reg = LinearRegression(fit_intercept=True)
    reg = reg.fit(X_train, y_train)
    y_pred = reg.predict(X_val)
    linregVal_error[i-1] = mean_squared_error(y_val, y_pred)
    #y_pred1 = reg.predict(X_train)
    #linregTrain_error[i-1] = mean_squared_error(y_train, pred1)


# Print the training errors
#print(f"Validation errors (rounded to 2 decimals): \n {np.
 ↪round(linregVal_error, 2)}")
#print(f"Training errors (rounded to 2 decimals): \n {np.
 ↪round(linregTrain_error, 2)}")


# create a numpy array "r_values" containing the values 1,2...,max_r
r_values = samplesUsed
# create a plot object which can be accessed using variables "fig" and "axes"
fig, axes = plt.subplots(1,1, figsize=(8, 5))
# add a curve representing the average squared error for each choice of r
axes.plot(r_values, linregVal_error, label='MSE', color='red')
# add captions for the horizontal and vertical axes
axes.set_xlabel('samples count')
axes.set_ylabel('empirical error')
# add a title to the plot
axes.set_title('Training error vs number of samples')
axes.legend()
plt.tight_layout()
plt.show()
```

Analysis of Result

- It could be seen that the Training Error is at its lowest with 4000 samples.
- The possible justification is that overfitting occurs when samples more than 4000 are included
- As a next step, the training error was recomputed with the best combination of above two exercises. In other words, the training error was computed with ALL Features which yielded the best result in the first exercise and with 4000 samples which again yielded the best result in the second exercise

[40]:
```
#Error Calculation with variable number of features
# maximum number of features used
maximum_features = 29
# read in all data points using max_r features
X, y = GetFeaturesLabels(maximum_features)
# vector for storing the training error of LinearRegresion.fit() for each r
linregVal_error = np.zeros(maximum_features-2)
linregTrain_error = np.zeros(maximum_features-2)

for index in range(2, maximum_features):
    X, y = GetFeaturesLabels(featureCount= index)
    X = X[:4000]
    y = y[:4000]
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
    reg = LinearRegression(fit_intercept=True)
```

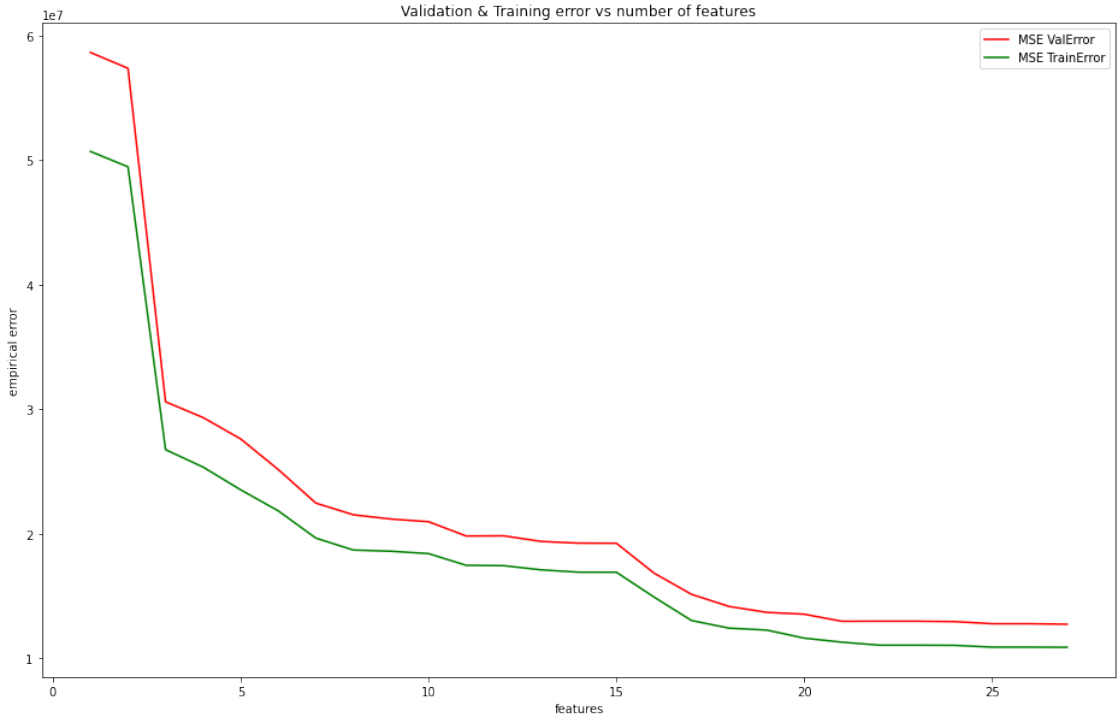
```
    reg = reg.fit(X_train, y_train)
    y_pred = reg.predict(X_val)
    linregVal_error[index-2] = mean_squared_error(y_val, y_pred)
    y_pred1 = reg.predict(X_train)
    linregTrain_error[index-2] = mean_squared_error(y_train, y_pred1)
print("---Weight Vector-----")
print(reg.coef_)


# Print the training errors
#print(f"Validation errors (rounded to 2 decimals): \n {np.
 ↪round(linregVal_error, 2)}")
#print(f"Training errors (rounded to 2 decimals): \n {np.
 ↪round(linregTrain_error, 2)}")


# create a numpy array "r_values" containing the values 1,2...,max_r
r_values = np.linspace(1, maximum_features-2, maximum_features-2, endpoint=True)
# create a plot object which can be accessed using variables "fig" and "axes"
fig, axes = plt.subplots(1,1, figsize=(16, 10))
# add a curve representing the average squared error for each choice of r
axes.plot(r_values, linregVal_error, label='MSE ValError', color='red')
axes.plot(r_values, linregTrain_error, label='MSE TrainError', color='green')
# add captions for the horizontal and vertical axes
axes.set_xlabel('features')
axes.set_ylabel('empirical error')
# add a title to the plot
axes.set_title('Validation & Training error vs number of features')
axes.legend()
print("---Validation Error-----")
print(linregVal_error)
print("---Training Error-----")
print(linregTrain_error)
```

```
---Weight Vector-----
[ 2.25127346e+03 -1.23695512e-01  4.33021145e+03 -1.13146447e+04
 -1.18017371e+04 -8.71398248e+03 -9.21207394e+03 -6.97285228e+03
 -4.91368659e+03  3.46221847e+03  1.74218550e+04  3.94514028e+03
  3.17561735e+03  1.61823447e+04 -3.96884880e+02 -8.34210971e+03
 -7.71297441e+03 -2.33656195e+03  3.21402650e+02  6.63660438e+03
  3.77627512e+03  2.26901164e+04 -5.89406635e+03  3.24169787e+00
 -5.45791464e+02  5.42549766e+02 -2.39099776e+02]
---Validation Error-----
[58628398.76216327 57357720.17077576 30588654.19527505 29321813.26712004
 27604671.50063821 25150660.1061774  22458519.93566503 21516027.94300891
 21176013.68640488 20962503.22698726 19817144.72667251 19829791.60711297
 19379161.03676664 19242140.37208818 19235280.04782359 16842491.85512713
```

```
 15132997.54840832 14160648.62451728 13687438.21134562 13538481.40707987
 12969213.26296465 12976040.55547994 12976040.55547989 12944326.21235884
 12774144.56109965 12774144.56109973 12732742.65744813]
---Training Error-----
[50685932.40734331 49446609.24330065 26744717.3235126  25345075.29407465
 23521288.6540581  21834290.1961676  19647282.74319436 18687469.44620726
 18593661.08652604 18403915.29429509 17463398.16991414 17434598.24646787
 17098471.90803213 16912039.16851474 16908747.61969123 14919263.08132117
 13031926.94990278 12419641.36288669 12266329.65878817 11617231.35858678
 11293639.65280075 11052651.304392   11052651.30439167 11038761.11096745
 10894553.34290899 10894553.34290923 10883940.11996005]
```



Analysis of Result Least Error with Linear Regression

1. It can be seen that the Validation and Training error in this case is the least when compared with the various exercises done earlier in this project

LASSO Regression

[41]:
```
from sklearn.linear_model import Lasso

X,y = GetFeaturesLabels()    # read in m data points using n features
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=2)    # 80% training and 20% test
```

```
def fit_lasso(X_train, y_train, alpha_val):
    lasso = Lasso(alpha=alpha_val, fit_intercept=True)    # Create Ridge␣
 ↪regression model
    lasso.fit(X_train, y_train)
    y_pred = lasso.predict(X_train)
    w_opt = lasso.coef_
    training_error = mean_squared_error(y_pred, y_train)
    return w_opt, training_error

# Set alpha value
alpha_val = 10

# Fit Lasso and calculate optimal weights and training error using the function␣
 ↪'fit_lasso'
w_opt, training_error = fit_lasso(X_train, y_train, alpha_val)

# Print optimal weights and the corresponding training error
print('Optimal weights: \n', w_opt)
print('Training error: \n', training_error)
```

```
Optimal weights:
 [ 2.29160239e+03 -1.06079553e-01  4.77784365e+03 -8.48811041e+03
 -8.78227255e+03 -6.02042512e+03 -6.22275455e+03 -4.25835533e+03
 -2.62488193e+03  3.56274250e+03  1.60219033e+04  0.00000000e+00
  5.64240850e+03  4.52591953e+02  0.00000000e+00 -5.79601022e+03
 -4.29315952e+03  0.00000000e+00  1.26895449e+03  8.01287835e+03
  6.09091582e+03  4.53736535e+03 -1.81014434e+03 -0.00000000e+00
 -5.07416021e+02  7.34213791e+02 -4.58696380e+02  0.00000000e+00]
Training error:
 13596051.727953652
```

Analysis of Result

- It can be seen that Lasso Regression eliminates 5 coefficients and thus simplifies the ML features
- However, it comes with a cost that the training error is slightly higher than the linear regression (which requires more weight vectors)
- Training Error in Linear Regression:10883940.11
- Training Error in Lasso Regression:13596051.72

CONCLUSION

1. The main findings could be seen in this note book with the title “Analysis of Result”
2. In conclusion, it can be seen that both Linear Regression and Lasso Regression prove to be effective
3. The choice of one technique vs the other is based on the accuracy needs vs the reduced complexity of ML algorithm. It can be seen that Linear Regression with all the features offers the best accuracy while the Lasso Regression reduces the complexity with a little loss

in the accuracy

4. Therefore the suitable ML technique is more a choice based on the accuracy needs of the end application. Concluding Remarks on this exercise

- Various data were analyzed prior to finalizing this data. This was also sanity checked and validated by visualization plots before use in the project
- One hot encoding techniques were used to convert the string categories into numerical values
- Applicable techniques were done and duly analyzed with varying feature count, sample count and their combinations
- The results from every step is analyzed and importantly leveraged in the subsequent step in this project

[ ]: