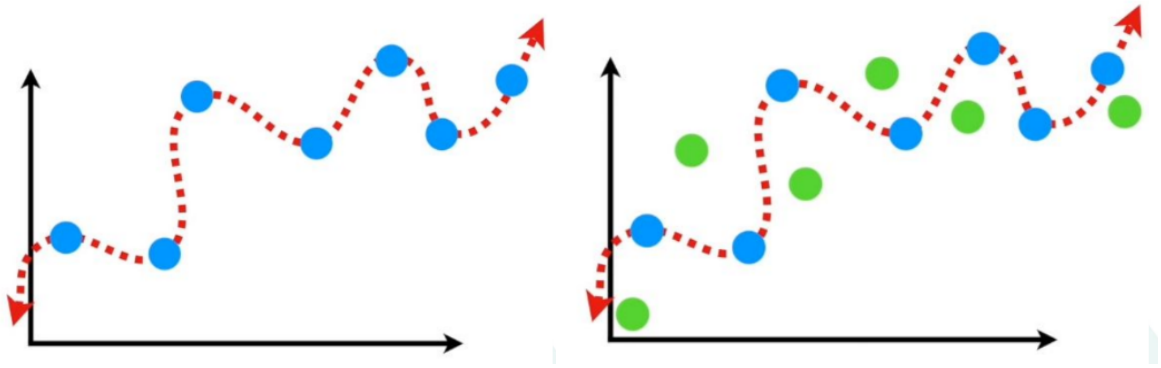
Machine Learning

Section A (Theoretical) Solutions

A) Reference : Lecture Slides of Logistic Regression



Part A Part B

If the model has high complexity , for example a high-degree polynomial as illustrated above.(Part A) Then model would have **low bias** because of almost 0 training error and **high variance** because of very high testing error , thus , high (testing - training) error. (As illustrated by part B , where green dots represent testing data and blue dots represent training data). Thus, the model is likely to **overfit** in such cases.

B) Reference : https://www.geeksforgeeks.org/f1-score-in-machine-learning/

B) Spam → 1, Legitimate → 0

True Positive (TP) = 200
True Negative (TN) = 730
False Positive (FP) = 20
False Negative (FN) = 50

(i) Precision $= \frac{TP}{TP+FP} = \frac{200}{200+20} = \frac{200}{220} = \frac{10}{11}$
$= 0.909$

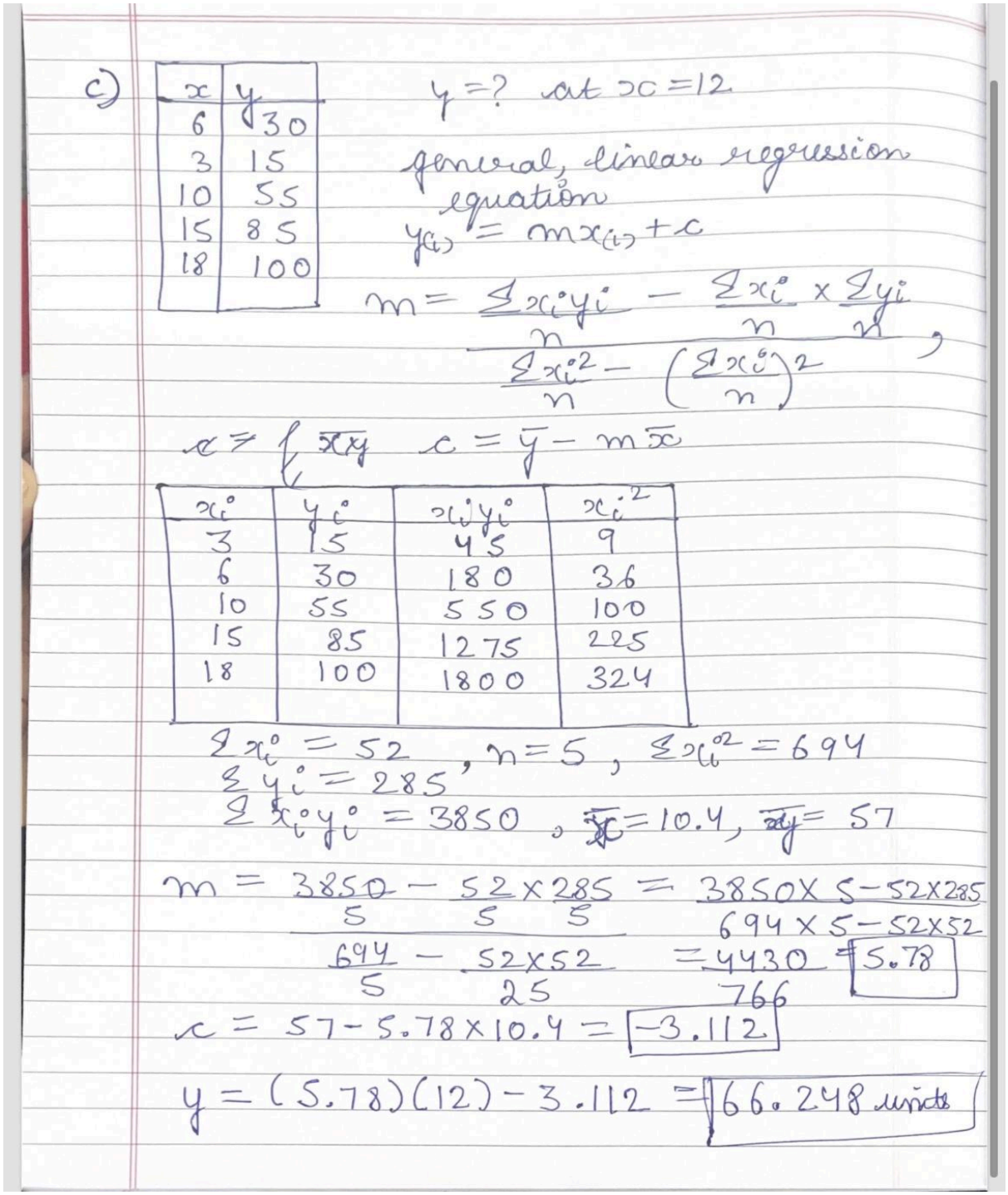
(ii) Accuracy $= \frac{TP+TN}{(TP+TN+FP+FN)} = \frac{200+730}{(200+730+20+50)}$
$= \frac{930}{1000} = 0.93 \quad (93\%)$

(iii) Recall $= \frac{TP}{TP+FN} = \frac{200}{200+50} = \frac{200}{250} = \frac{4}{5}$
$= 0.8$

(iv) F1 score $= 2 \times \frac{(Precision \times Recall)}{(Precision + Recall)}$
$= 2 \frac{\left(\frac{10}{11} \times \frac{4}{5}\right)}{\left(\frac{10}{11} + \frac{4}{5}\right)} = \frac{16}{\frac{94}{55}} = \frac{16 \times 5}{94}$
$= \frac{80}{94} = 0.851$

∵ F1 score measures the performance of model which is 0.851, and the model too has a good accuracy of 93%.

C) Reference : Prev year tutorial



c)

x	y
6	30
3	15
10	55
15	85
18	100

$y = ?$ at $x = 12$

general, linear regression equation

$y_{(i)} = mx_{(i)} + c$

$$m = \frac{\frac{\Sigma x_i y_i}{n} - \frac{\Sigma x_i}{n} \times \frac{\Sigma y_i}{n}}{\frac{\Sigma x_i^2}{n} - \left(\frac{\Sigma x_i}{n}\right)^2},$$

$c = \bar{y} - m\bar{x}$

x_i	y_i	$x_i y_i$	x_i^2
3	15	45	9
6	30	180	36
10	55	550	100
15	85	1275	225
18	100	1800	324

$\Sigma x_i = 52$, $n = 5$, $\Sigma x_i^2 = 694$

$\Sigma y_i = 285$

$\Sigma x_i y_i = 3850$, $\bar{x} = 10.4$, $\bar{xy} = 57$

$$m = \frac{3850}{5} - \frac{52 \times 285}{5 \quad 5} = \frac{3850 \times 5 - 52 \times 285}{694 \times 5 - 52 \times 52}$$

$$\frac{694}{5} - \frac{52 \times 52}{25} = \frac{4430}{766} = \boxed{5.78}$$

$c = 57 - 5.78 \times 10.4 = \boxed{-3.112}$

$y = (5.78)(12) - 3.112 = \boxed{66.248 \text{ units}}$

D) Given such a scenario where given two models f1 and f2 , where empirical risk for f1 is lower than that for f2, such case where model f_1 has a lower empirical risk on the training set but may not necessarily generalize better than model f_2. This case can occur when the model f1 overfits the data, is a complex equation resulting in lower empirical risk (training error) than f2 but doesn't perform good on the testing data. For example , given toy example between height and weight .

Let f1 be a random curve (high degree polynomial equation) which satisfies f1(150) = 50 , f1(160) = 0 , f1(180) = 80 etc.

and f2 be simple linear equation (y = mx + c) : (y = x - 50 in below example)

Here , empirical risk of f1 is 0 whereas that of f2 (MSE is 12.8) (predicted weight 80 for height 180 rather than 88) . However , f2 would perform better on testing data . For example, given height 210 , f2 would predict better like 110 whereas f1 may or may not predict right. Thus , f2 generalizes the data more than f1.

X (Height)	Y (Weight)
150	50
160	60
180	88
190	90
200	100

Section B (Scratch Implementation) Solutions

A) For implementing Batch Gradient Descent from scratch first the data is processing , following are the steps followed for it :

```
'''
There is not categorical data to encode
'''

df.info()
```

```
'''
Dropping NULL values
'''

df = df.dropna()
```

```
'''
A function for train , test and valuation split : 70:15:15 (train: test: validation)
'''

def train_test_val_split(X,y):
    index = np.random.permutation(X.shape[0])
    X = X[index]
    y = y[index]

    X_train = X[:int(0.7 * X.shape[0])]
    y_train = y[:int(0.7 * y.shape[0])]
    X_val = X[int(0.7 * X.shape[0]):int((0.85) * X.shape[0])]
    y_val = y[int(0.7 * y.shape[0]):int((0.85) * y.shape[0])]
    X_test = X[int((0.85) * X.shape[0]):]
    y_test = y[int((0.85) * y.shape[0]):]

    return (X_train,y_train),(X_val,y_val),(X_test,y_test)
```

```
(X_train, y_train), (X_val, y_val),(X_test, y_test) = train_test_val_split(X, y)  # For part a
```

```
'''
However, used inbuilt train-test split function because it gave better results
'''

from sklearn.model_selection import train_test_split

X_train4, X_temp4, y_train4, y_temp4 = train_test_split(X, y, test_size=0.3, random_state=42)
X_val4, X_test4, y_val4, y_test4 = train_test_split(X_temp4, y_temp4, test_size=0.5, random_state=42)

X_train5, X_temp5, y_train5, y_temp5 = train_test_split(X, y, test_size=0.3, random_state=42)
X_val5, X_test5, y_val5, y_test5 = train_test_split(X_temp5, y_temp5, test_size=0.5, random_state=42)

```

Implemented both StandardScaler() and also implemented it from Scratch,

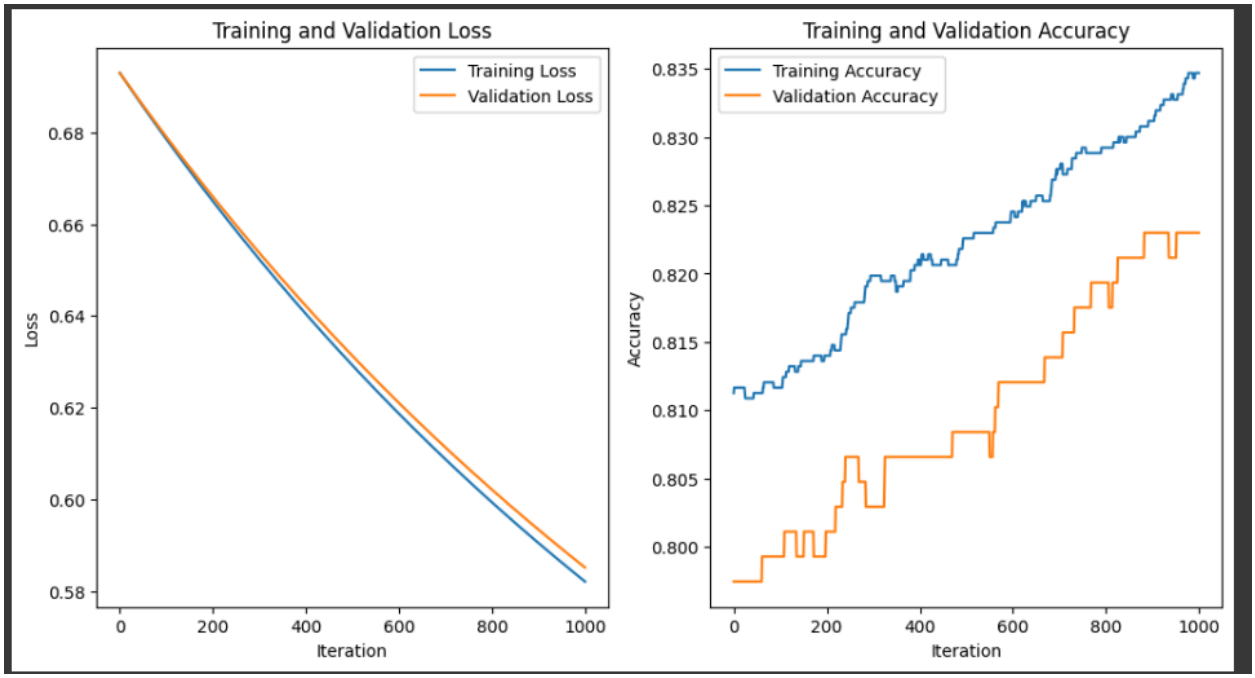
By calculating the standard deviation and mean of the training data followed by fit transforming on training data and transforming on testing data.

```
'''
Using an inbuilt scalar also gave better results than implemented scalar
'''

scaler = StandardScaler()

X_train4 = scaler.fit_transform(X_train4)
X_val4 = scaler.transform(X_val4)
X_test4 = scaler.transform(X_test4)
```

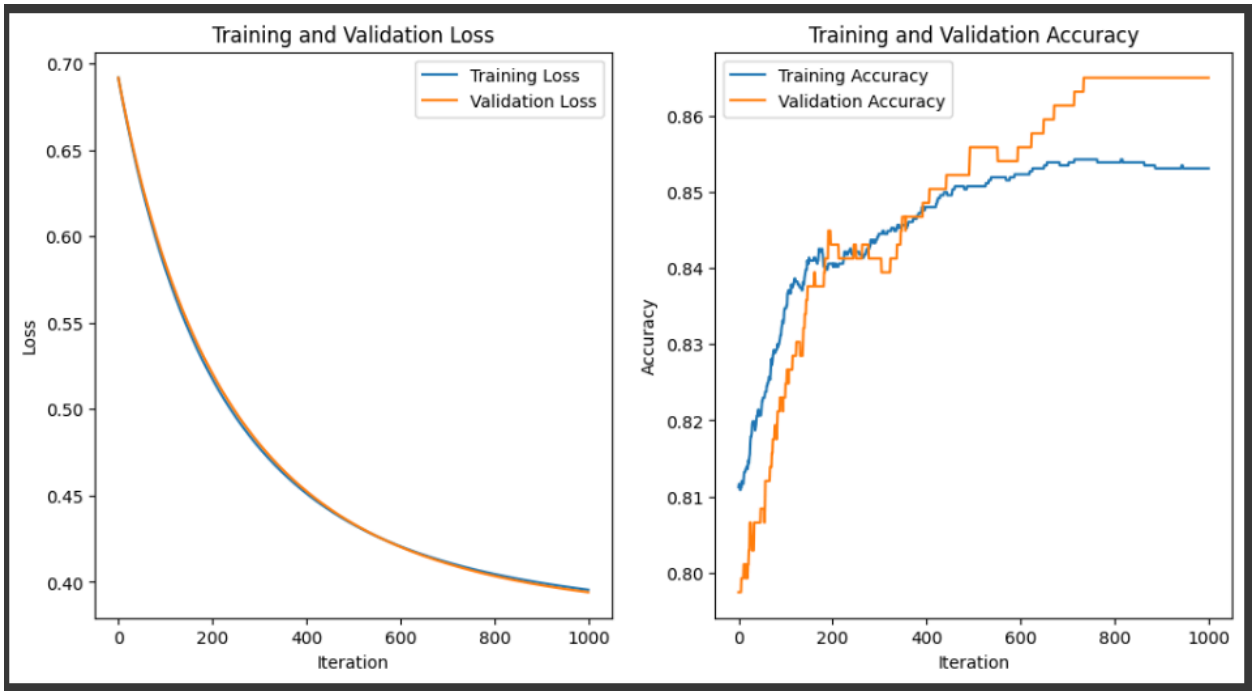
```
class StandardScaler:
    def __init__(self):
        self.mean = None
        self.std = None

    def fit(self, X):
        """
        Computing the mean and standard deviation for each feature in the training data.
        """
        self.mean = np.mean(X, axis=0)
        self.std = np.std(X, axis=0)

    def transform(self, X):
        """
        Transforming the data by subtracting the mean and dividing by the standard deviation.
        """
        if self.mean is None or self.std is None:
            raise ValueError("The scaler is not fitted ")
        return (X - self.mean) / self.std

    def fit_transform(self, X):
        """
        Fitting to the data and then transform it.
        """
        self.fit(X)
        return self.transform(X)
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

Batch Gradient Descent is implemented by calculating the cross entropy loss and sigmoid function , followed by appending the losses and accuracies in a list. Following are the results obtained :



For alpha = 0.001 and epochs = 1000



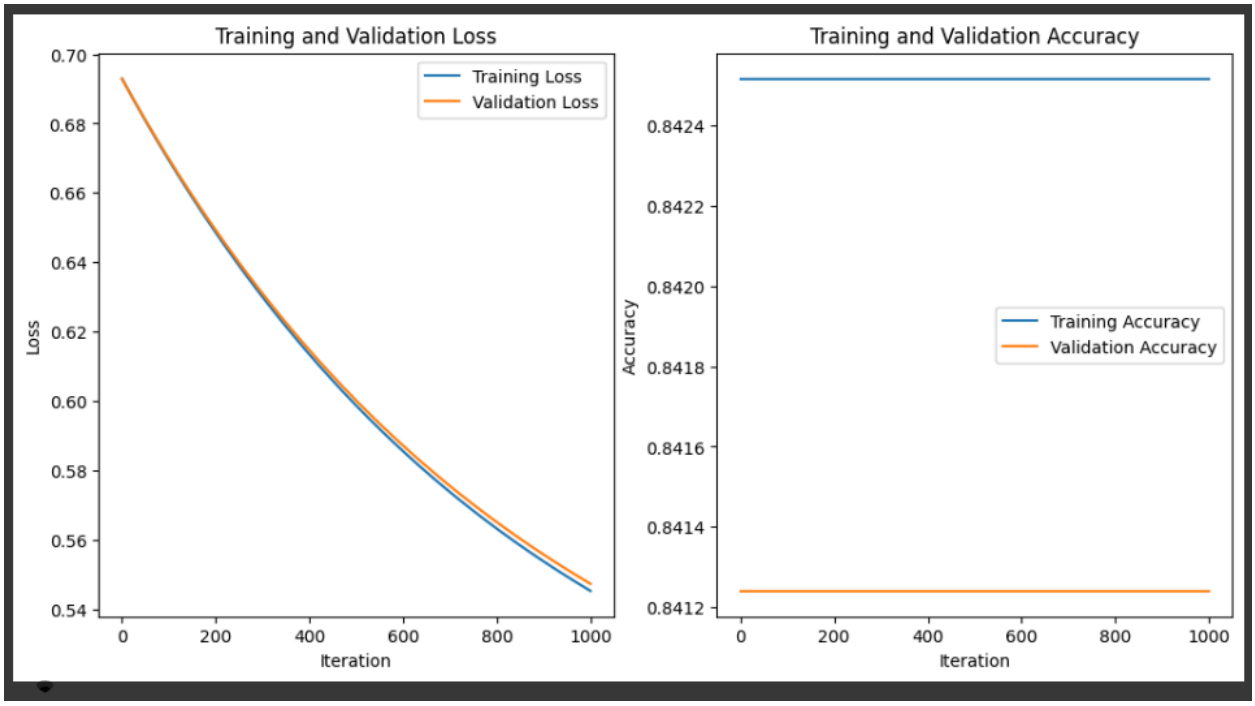
For alpha = 0.01 and epochs = 1000

Key Observations : the model performed better with moderate alpha like = 0.001 and as the number of epochs increase the training and validation losses decreases , followed by an increase in the training and validation accuracies. Leading the losses and accurcaies convergence

B)

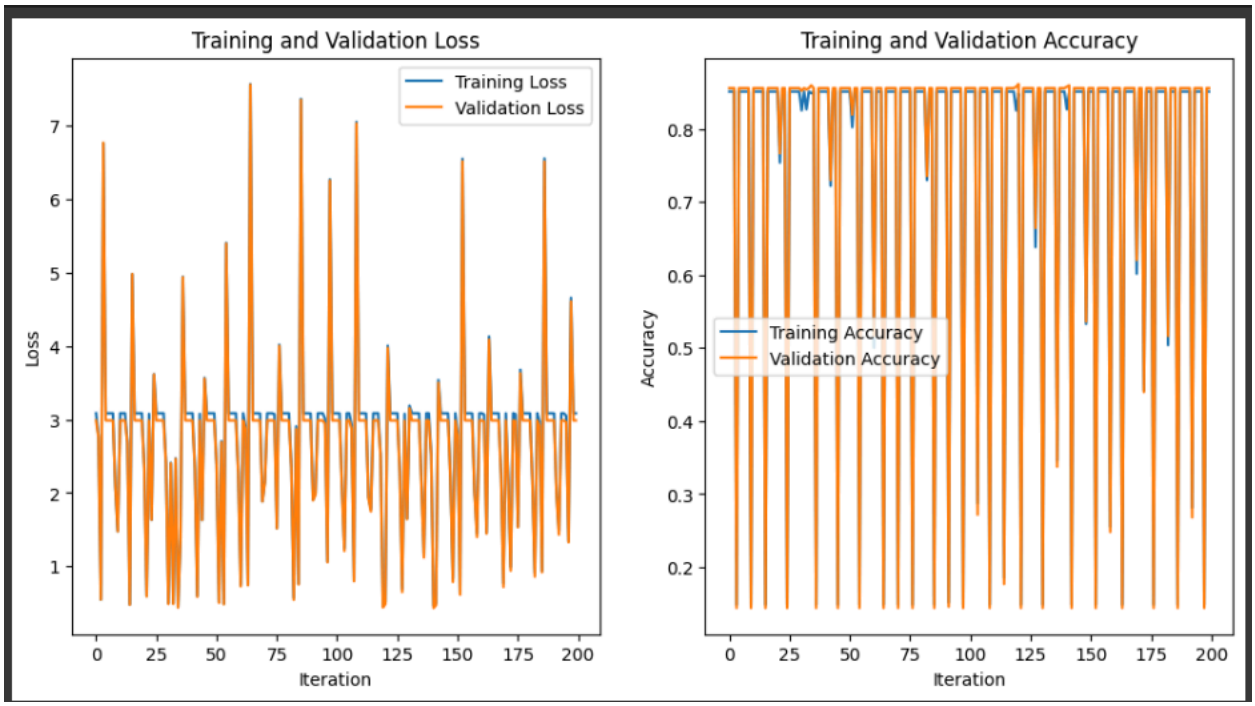
```
'''
Implementing Min-Max Scaler , here the min and max are evaluated for the training data and the tested data is fitter over it.
'''

X_min = X_train2.min(axis = 0)
X_max = X_train2.max(axis = 0)

X_train2 = (X_train2 - X_min)/(X_max-X_min)
X_val2 = (X_val2 - X_min)/(X_max-X_min)
X_test2 = (X_test2 - X_min)/(X_max-X_min)
```

Min - Max Scaling



For alpha = 0.001 and epochs = 1000 and similar graph for 0.01

No Scaling

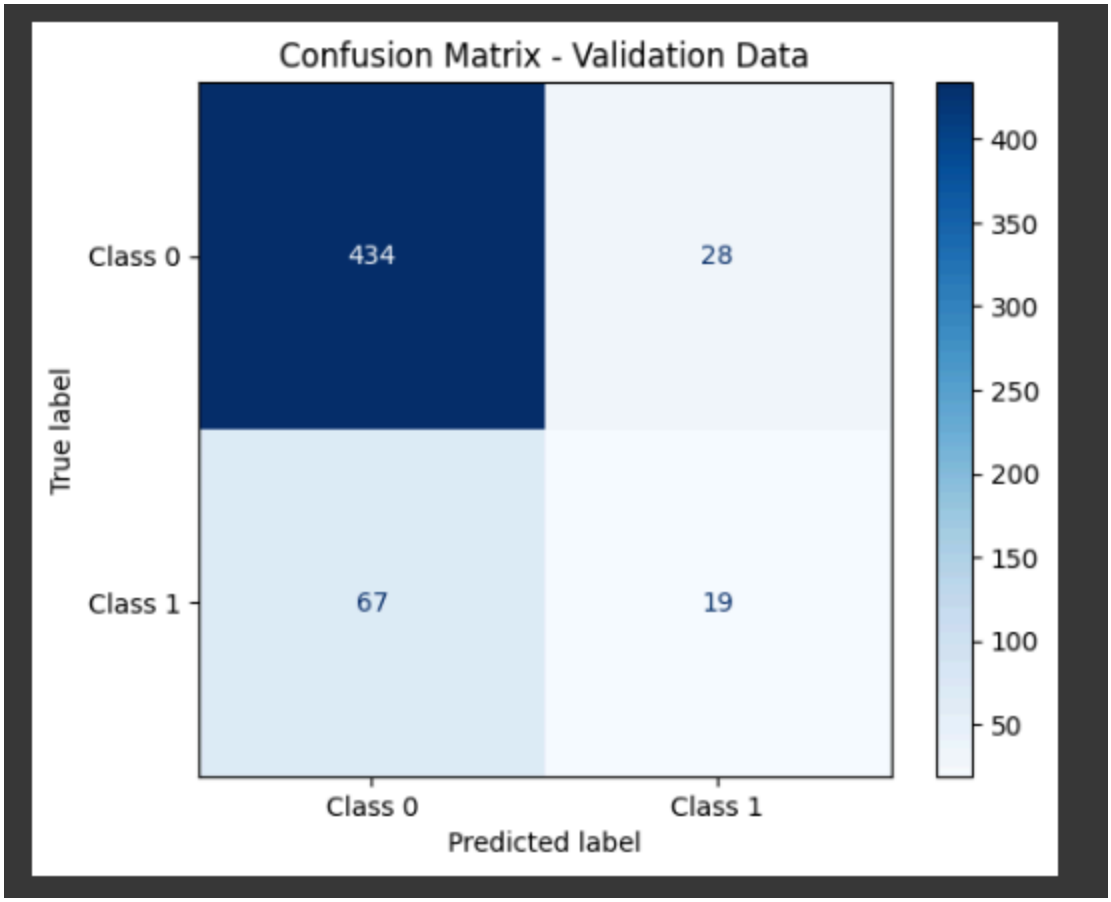


Key Observations : No scaling was not found t be a good approach with high training and testing loss , whereas min-max scaling was found to be a good scaling technique with minimum losses and better testing accuracy, The following can also be proved from the fact that the features followed a skew distribution and had high outliers , thus making min-max scaling better approach.

C)

```python
def confusion_matrix(y, y_pred):
    TP, TN, FP, FN = 0, 0, 0, 0
    for i in range(len(y)):
        if y[i] == 1 and y_pred[i] == 1:
            TP += 1
        elif y[i] == 0 and y_pred[i] == 0:
            TN += 1
        elif y[i] == 0 and y_pred[i] == 1:
            FP += 1
        else:
            FN += 1
    return TP, TN, FP, FN

# Accuracy function
def accuracy2(TP, TN, FP, FN):
    return (TP + TN) / (TP + TN + FP + FN)

# Precision function
def precision(TP, FP):
    return TP / (TP + FP) if (TP + FP) != 0 else 0

# Recall function
def recall(TP, FN):
    return TP / (TP + FN) if (TP + FN) != 0 else 0

# F1 score function
def f1_score(prec, rec):
    return 2 * prec * rec / (prec + rec) if (prec + rec) != 0 else 0
```

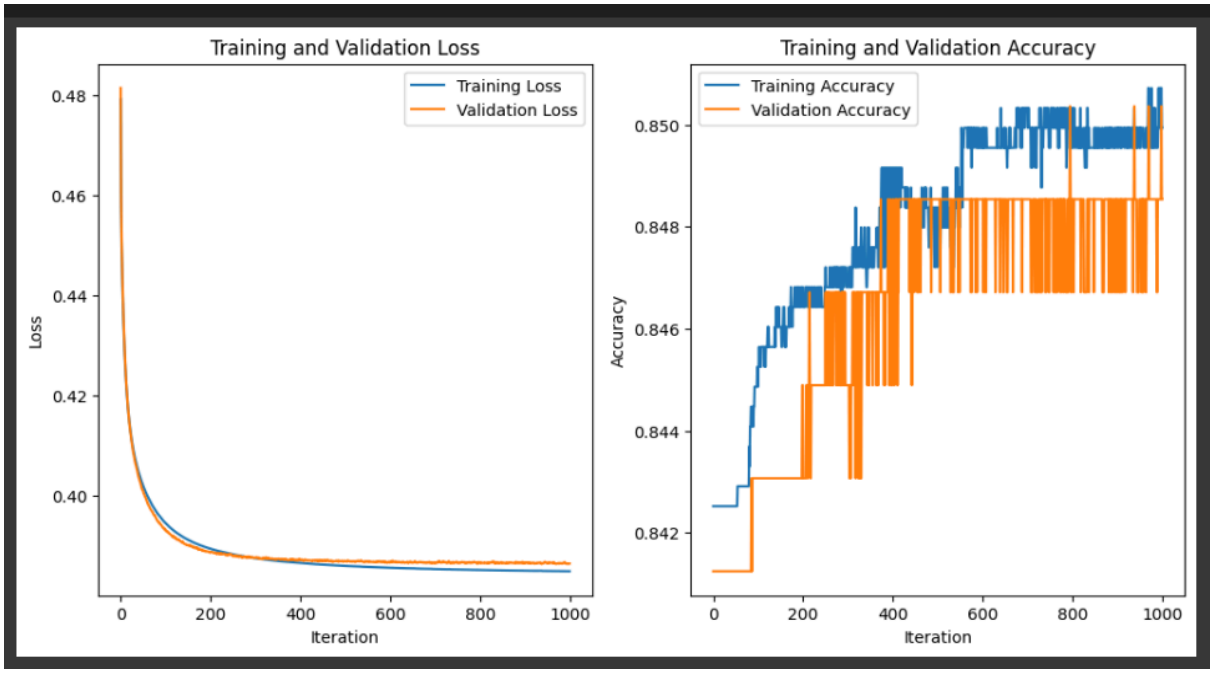
```
Test Results:
Validation Set Accuracy: 0.8266423357664233
Validation Set Precision: 0.40425531914893614
Validation Set Recall: 0.220930232558139 54
Validation Set F1 Score: 0.2857142857142857
Validation Set ROC-AUC Score: 0.7255360918151615
```



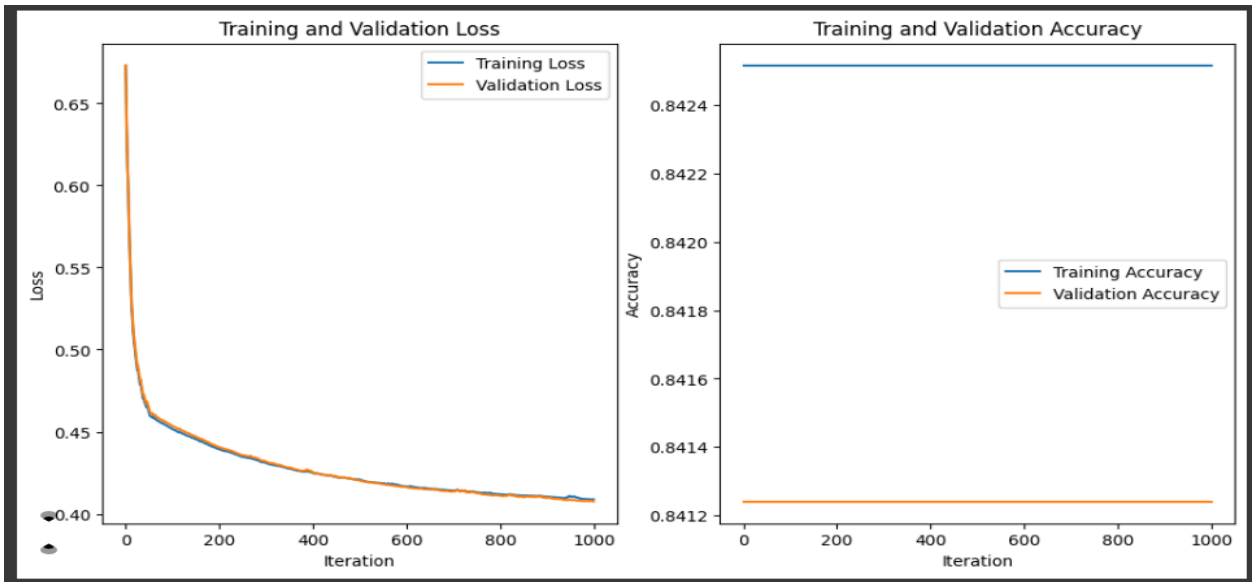
The results very verified using in built functions as well , and both give almost same result.

Key Observations : Lower recall and precision explain the fact that the model is not able to predict class 1 well , the result is correct since the data is not so good because high rows of class 0 and very less rowes of class 1.

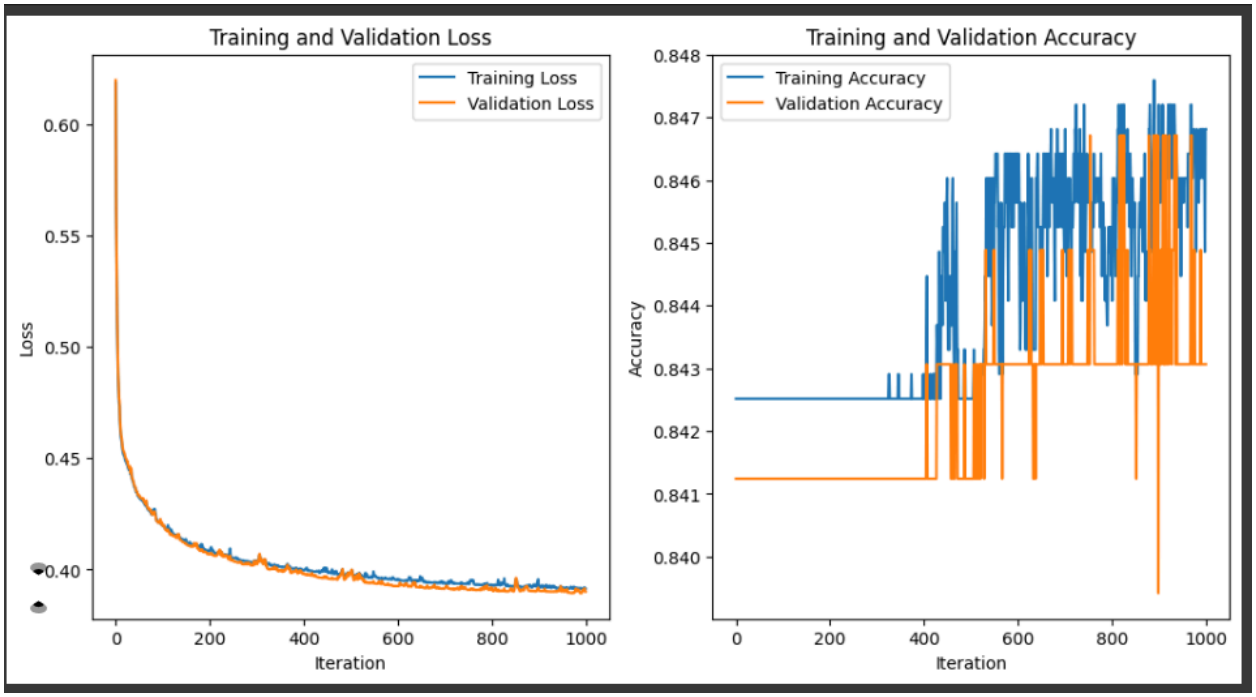
D) Stochastic gradient descent



Mini-Batch with Batch size 32 :



Mini-Batch with batch size 64 :



Key observations : both these performed better than batch gradient descent , however mini-batch performed the best , with less noisy convergence than stochastic gradient descent and very less loss.

However , the batch size 64 outperformed batch size 32 with better testing accuracy , however had very noisy convergence.

E) Performing the Kfold cross validation following results were obtained :

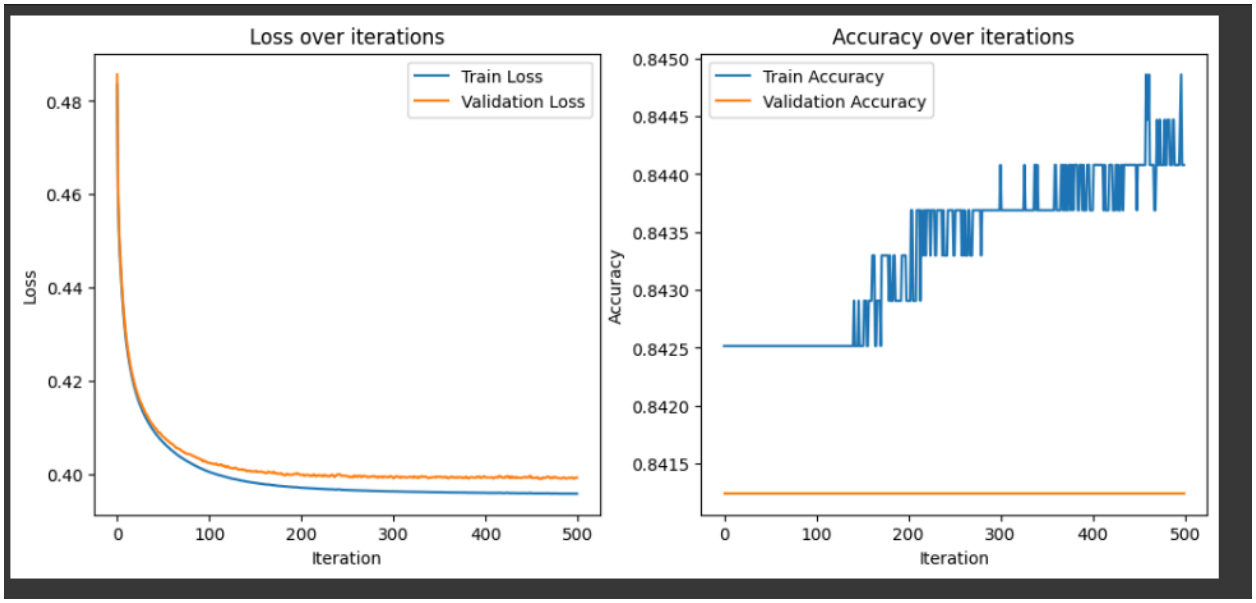
```
Fold 1: Validation Accuracy: 0.8235294117647058
Fold 2: Validation Accuracy: 0.8467852257181943
Fold 3: Validation Accuracy: 0.8290013679890561
Fold 4: Validation Accuracy: 0.853625170998632
Fold 5: Validation Accuracy: 0.813953488372093

Cross-Validation Results (Accuracy per fold): [0.8235294117647058, 0.8467852257181943, 0.8290013679890561, 0.853625170998632, 0.813953488372093]
Mean Accuracy across folds: 0.8333789329685362
  Mean Precision: 0.4077249415072822
  Mean Recall: 0.1920311584936441
  Mean F1 Score: 0.2586813218961737
  Standard Deviation of Accuracy: 0.014718443399521495
  Standard Deviation of Precision: 0.049403472686385265
  Standard Deviation of Recall: 0.029427420283132732
  Standard Deviation of F1 Score: 0.027995704664269566
```

Reference : https://medium.com/@avijit.bhattacharjee1996/implementing-k-fold-cross-validation-from-scratch-in-python-ae413b41c80d

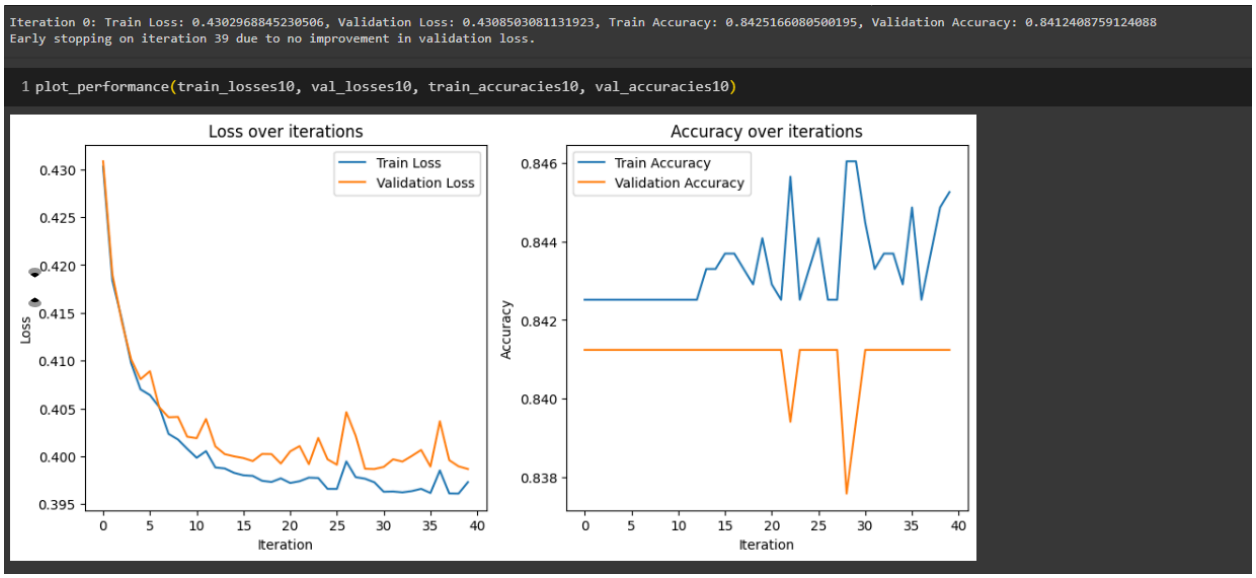
Key Observations : Each fold have the accuracy around the mean accuracy of the model, the model has close results with that with k-fold , and even kfold performed better than that without kfold, However this dataset was not really good to present the importance of kfold cross validatiion.

F) For L2 :

Without early stopping :



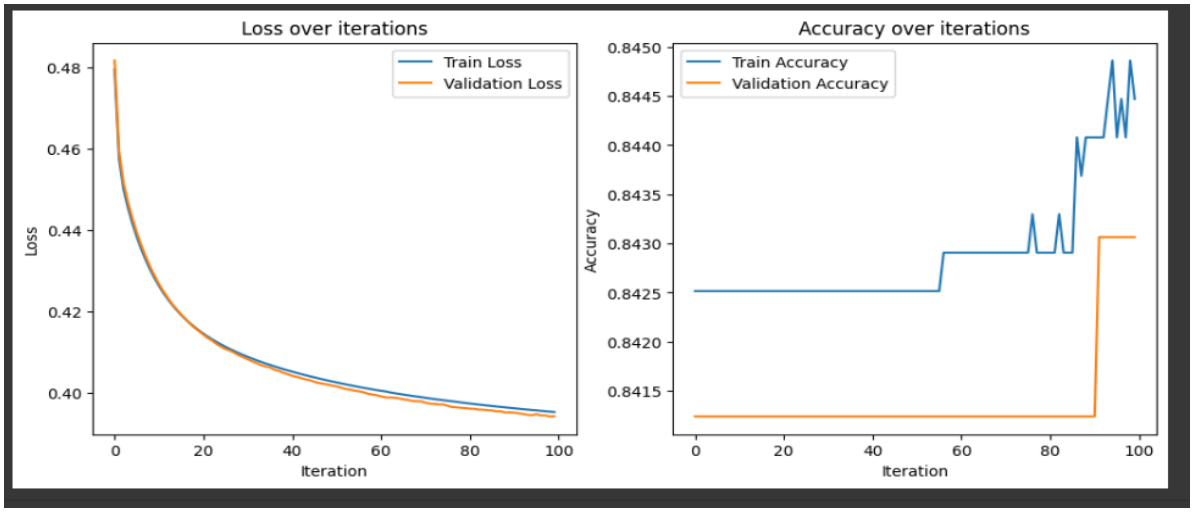
With early stopping :



Early stopping after 39 iterations :

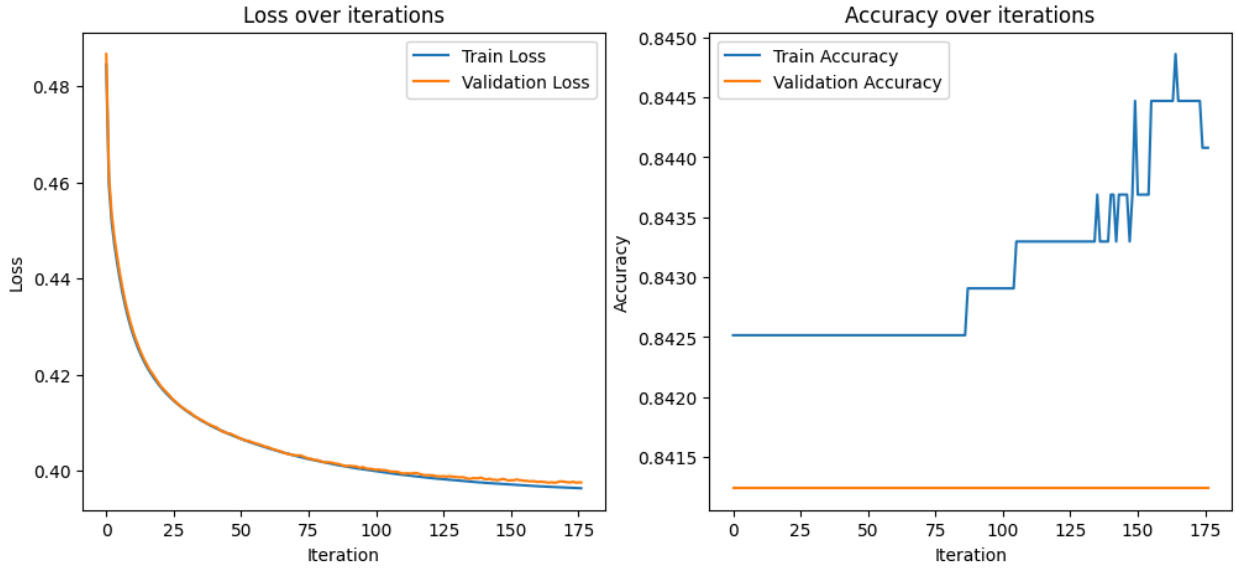
For L1 :

Without early stopping :



Experimented with different values of alpha , l1 regularization parameter

With early stopping :

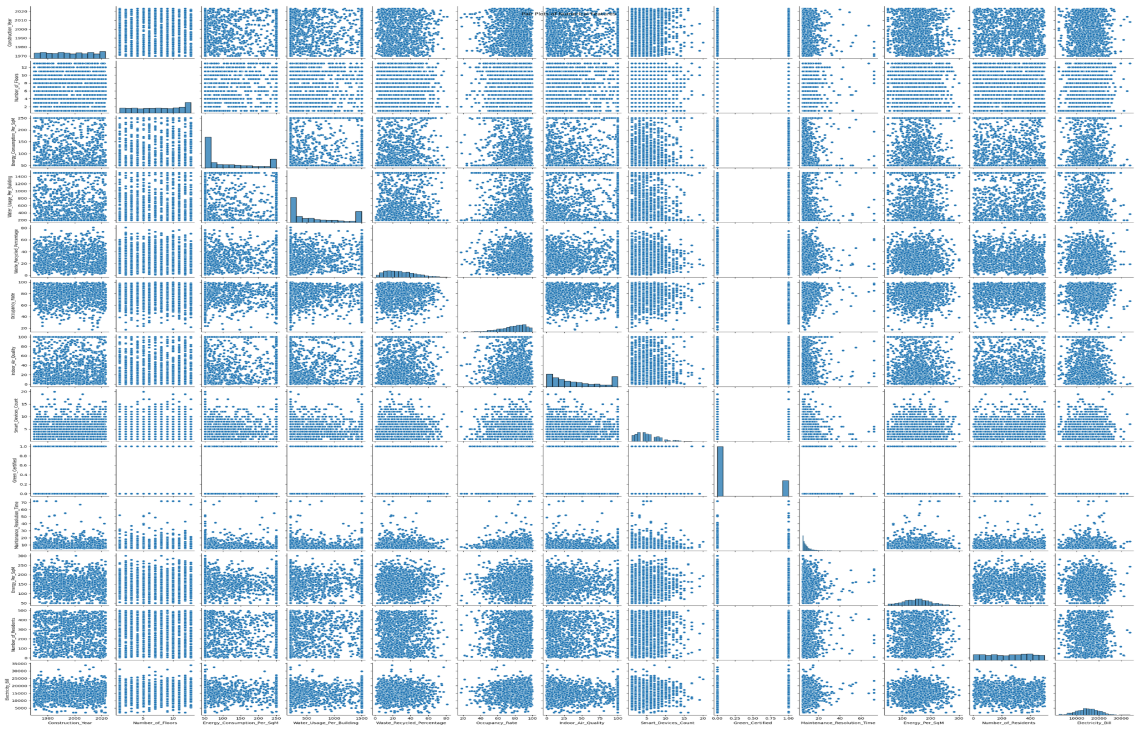


Early stopping occurred after 176 iterations because of no improvement in the loss

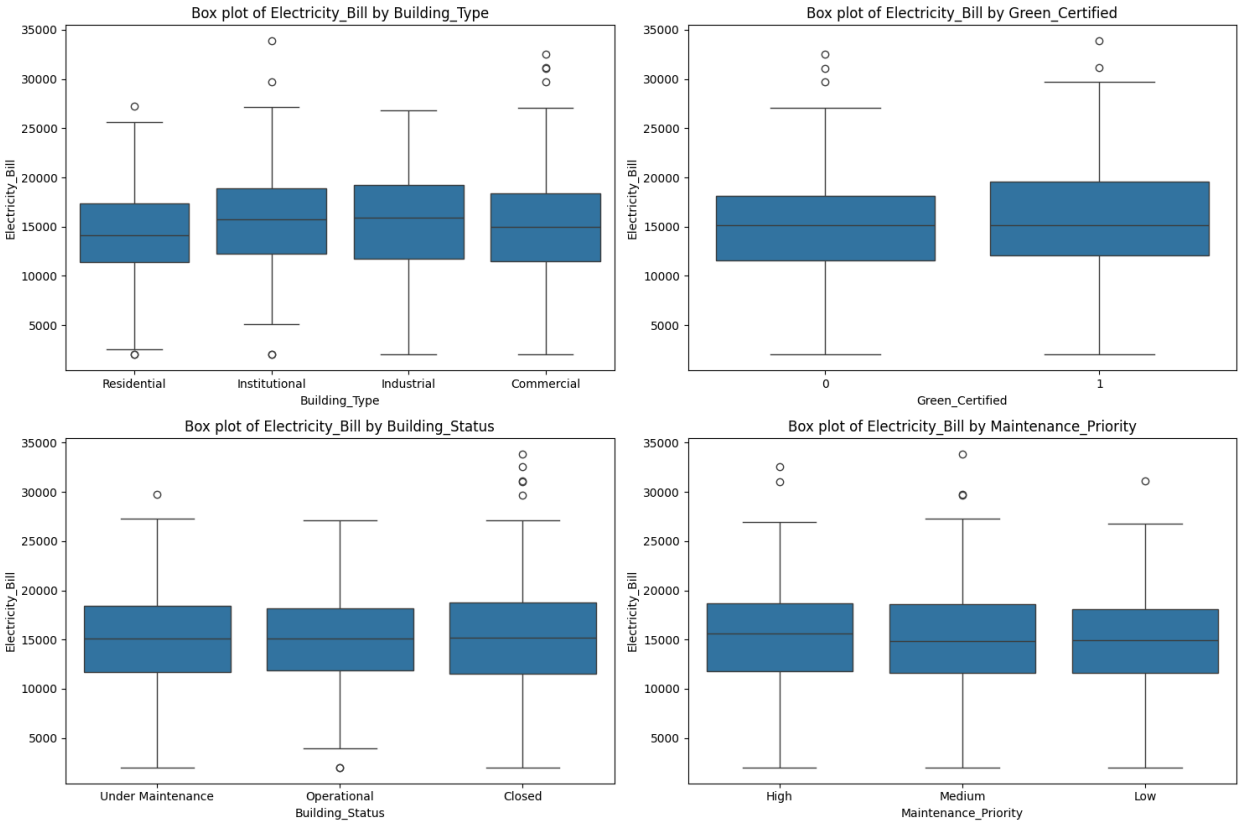
Key Observations : Early stopping prevents the model to overfit , when there is no significant change observed in the validation which can easily be seen from the graph above. With lower alpha the model performed better rather than high alpha , which resulted in more precise convergence. Also with moderate L1 and L2 regularization parameters the model gave a better performance

Section C (Algorithm implementation using packages) Solutions

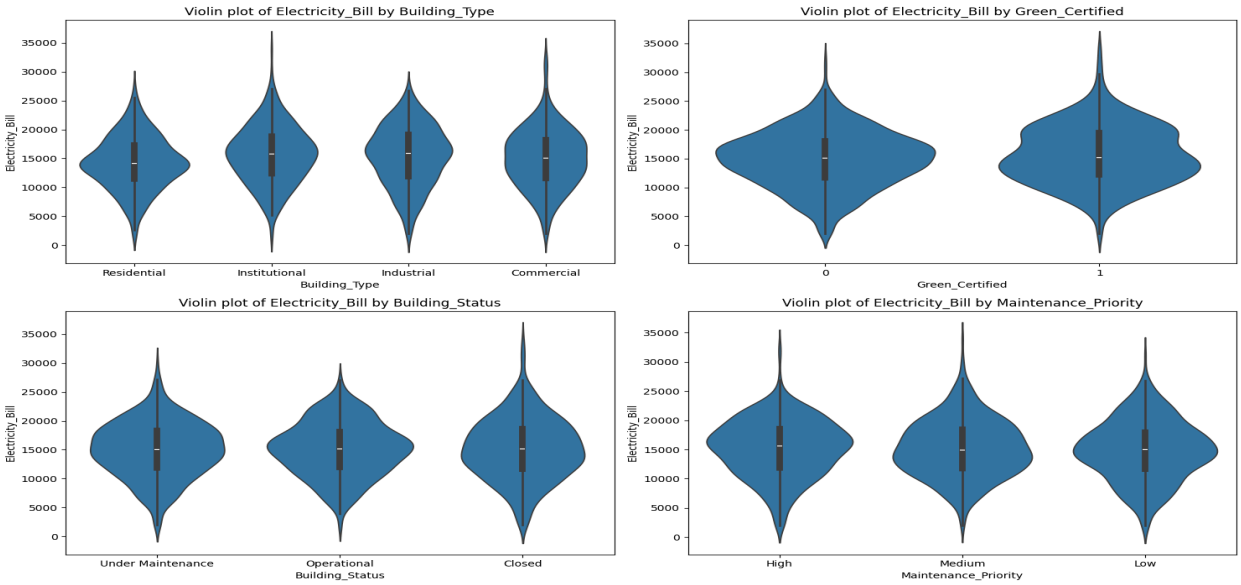
A) Performed EDA on the , following are the pair plots , violin plots , box plots and count plots obtained :



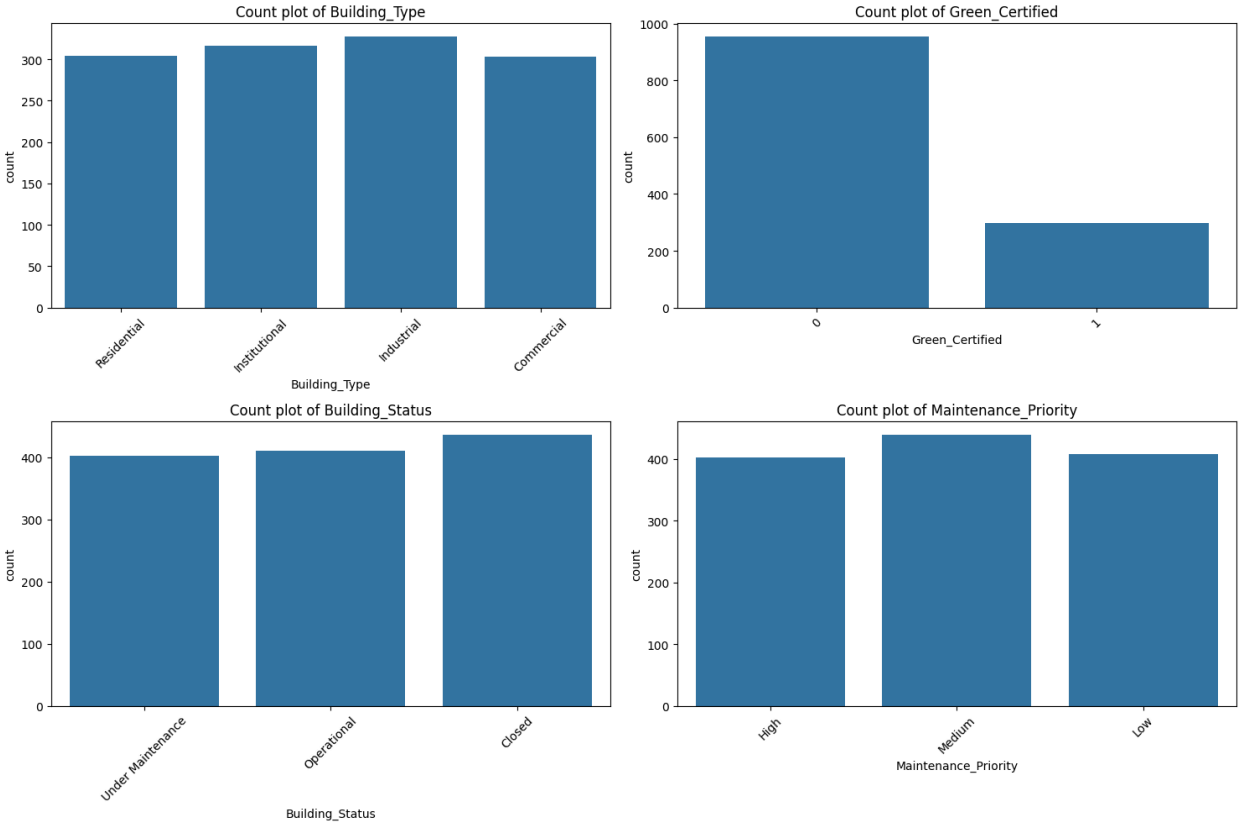
Pair plots


Box plot of Electricity_Bill by Building_Type
Electricity_Bill
Residential
Institutional
Industrial
Commercial
Building_Type
Box plot of Electricity_Bill by Green_Certified
0
1
Green_Certified
Box plot of Electricity_Bill by Building_Status
Under Maintenance
Operational
Closed
Building_Status
Box plot of Electricity_Bill by Maintenance_Priority
High
Medium
Low
Maintenance_Priority

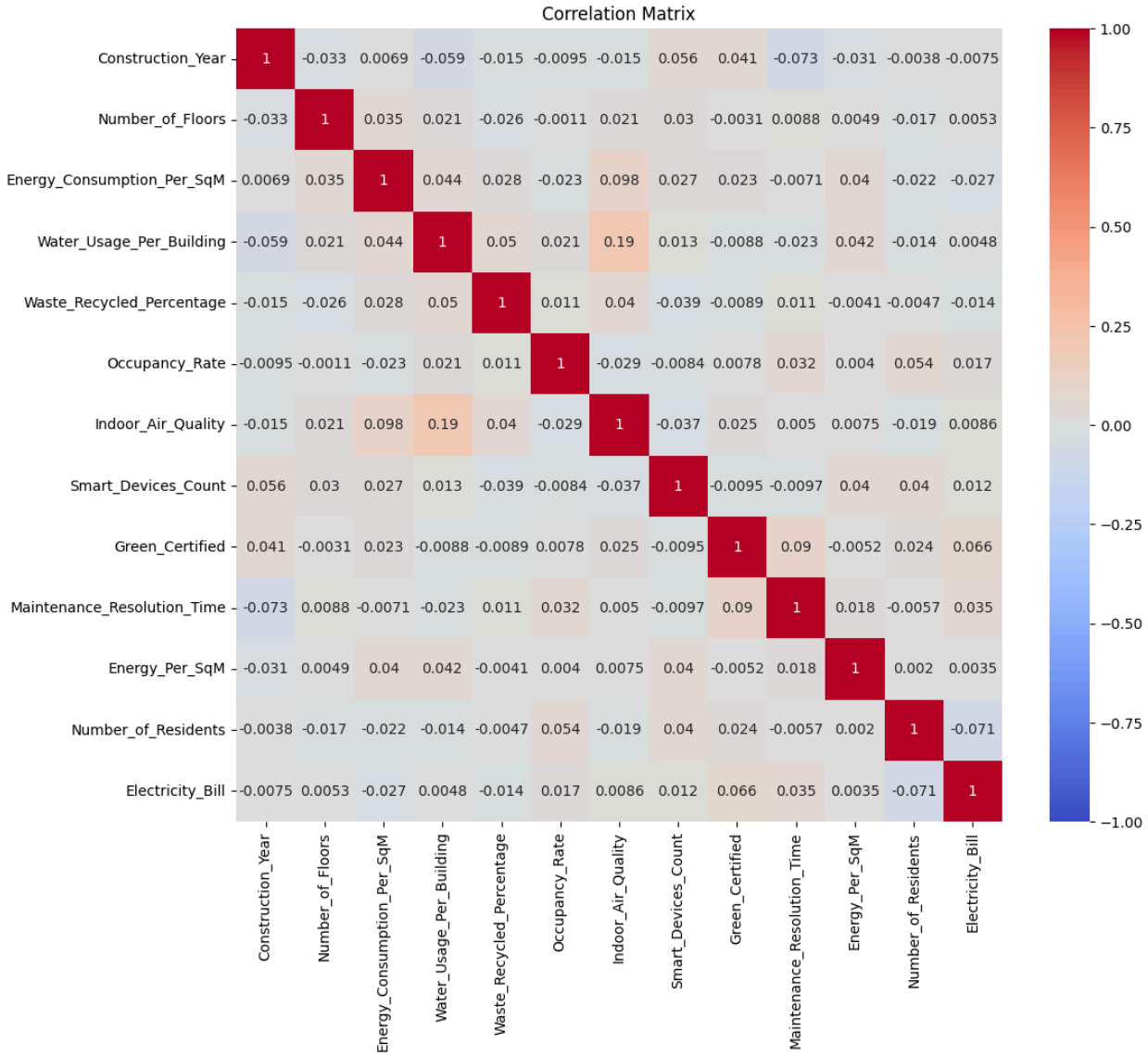
Box plots


Violin plot of Electricity_Bill by Building_Type
Electricity_Bill
Residential
Institutional
Industrial
Commercial
Building_Type
Violin plot of Electricity_Bill by Green_Certified
0
1
Green_Certified
Violin plot of Electricity_Bill by Building_Status
Under Maintenance
Operational
Closed
Building_Status
Violin plot of Electricity_Bill by Maintenance_Priority
High
Medium
Low
Maintenance_Priority

Violin plots



Count plots



Correlation Matrix

```
Insights gained from the dataset :
```

1. All building types have almost similar median electricity bill as seen from the box plots.

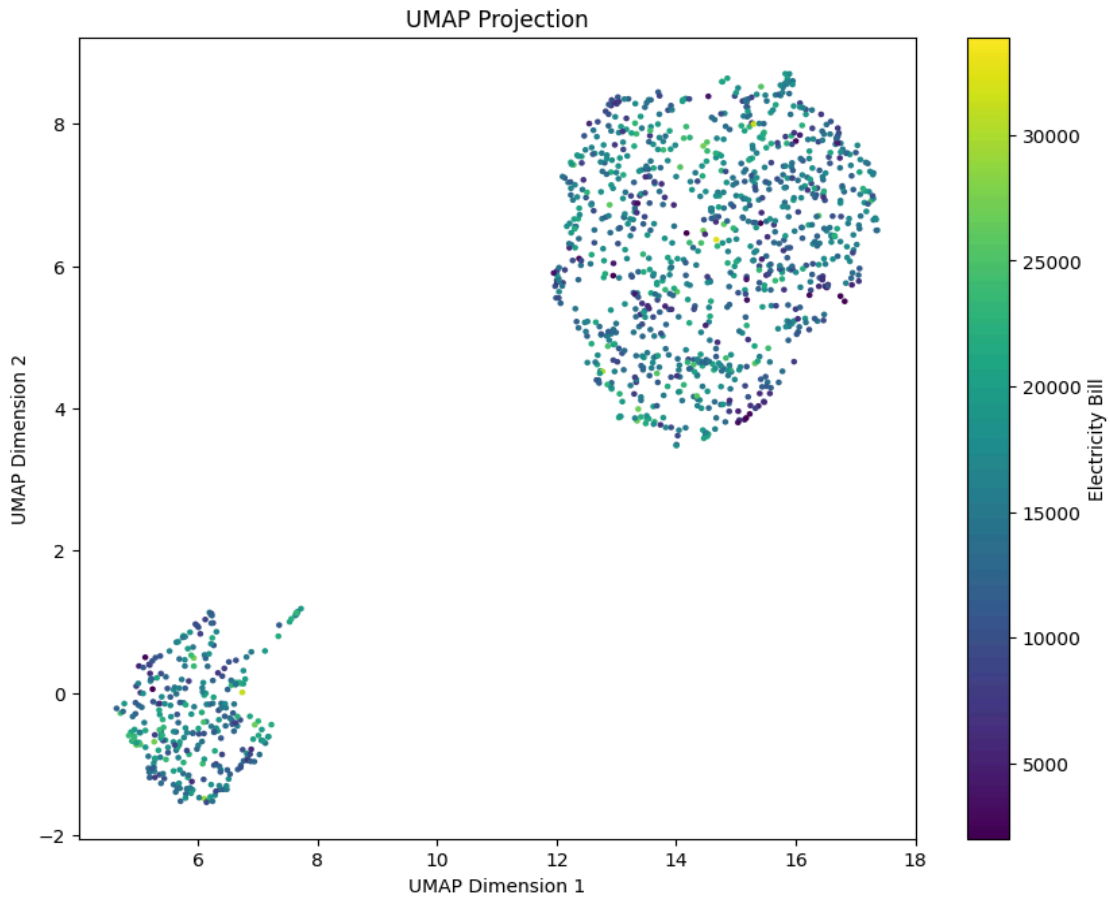
2. Significant outliers can be seen in each data , for example : high number of outliers in closed building status.

3. Violin plots of the green certifications suggest that mostly the bills are concentarted for both towards the median.

4. Closed buildings generally have the smallest spread in electricity bills across the median, likely due to reduced energy usage but its unevenly spread across the electricity bills

5. Indoor air quality and water usage per building are 2 features more correlated than others according to the confusion matrix.

6. The correlation matrix represents that electricity_bill is highly correlated with features such as Green Certified , Number of residents.



B)

UMAP

Insights gained : The UMAP projection is separated into 2 distinct parts as seen from the UMAP.It represents that the densities are not evenly distributed , there may be a bunch of outliers in the data which are located far from the density of the data.The colors on the colorbar, represents the electricity bill and how it is distributed across the data.

C) Necessary preprocessing steps include : checking and removing the null values , encoding the categorical data using LabelEncoder() , splitting the data into train test split (80 : 20)normalizing the numerical features which is done by using StandardScaler() , below code snippets illustrate the necessary preprocessing steps . further inbuilt linear regression model is applied and 5 mean square errors for the model are evaluated for training and testing data.

```
'''

Check for null values in the data
Insights : there are no null values obtained in the data

'''

df.isnull().sum()
```

```
'''
First encoding the categorical features , the checking of the null values have been done above
'''

cat_features = ["Building_Type","Building_Status","Maintenance_Priority"]
label_encoder = LabelEncoder()

for col in cat_features:
    features[col] = label_encoder.fit_transform(features[col])
    temp[col] = label_encoder.fit_transform(temp[col])

```

```
'''
Split the given dataset into 80:20 (train: test)
'''
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)
```

```
'''
Normalizing the data using the Standard Scaler
'''

scaler = StandardScaler()
numerical_cols = features.select_dtypes(include=[float, int]).columns
X_train[numerical_cols] = scaler.fit_transform(X_train[numerical_cols])
X_test[numerical_cols] = scaler.transform(X_test[numerical_cols])
X_train1[numerical_cols] = scaler.fit_transform(X_train1[numerical_cols])
X_test1[numerical_cols] = scaler.transform(X_test1[numerical_cols])
X_trainh[numerical_cols] = scaler.fit_transform(X_trainh[numerical_cols])
X_testh[numerical_cols] = scaler.transform(X_testh[numerical_cols])
```

Results Obtained from the model :

```
Training Data Accuracy:
Mean Squared Error (MSE):  24475013.16847547
Root Mean Squared Error (RMSE):  4947.222773281538
R2 Score:  0.013922520844610209
Adjusted R2 Score:  -0.0011091480449536562
Mean Absolute Error (MAE):  4006.32846932936

Testing Data Accuracy:
Mean Squared Error (MSE):  24278016.155742623
Root Mean Squared Error (RMSE):  4927.272689403604
R2 Score:  3.7344733075372893e-05
Adjusted R2 Score:  -0.0640628254763429
Mean Absolute Error (MAE):  3842.4093125585155
```

The model has a higher MSE whereas a lower R2 score , it has good testing accuracy as well because of lower MSE , MAE as well

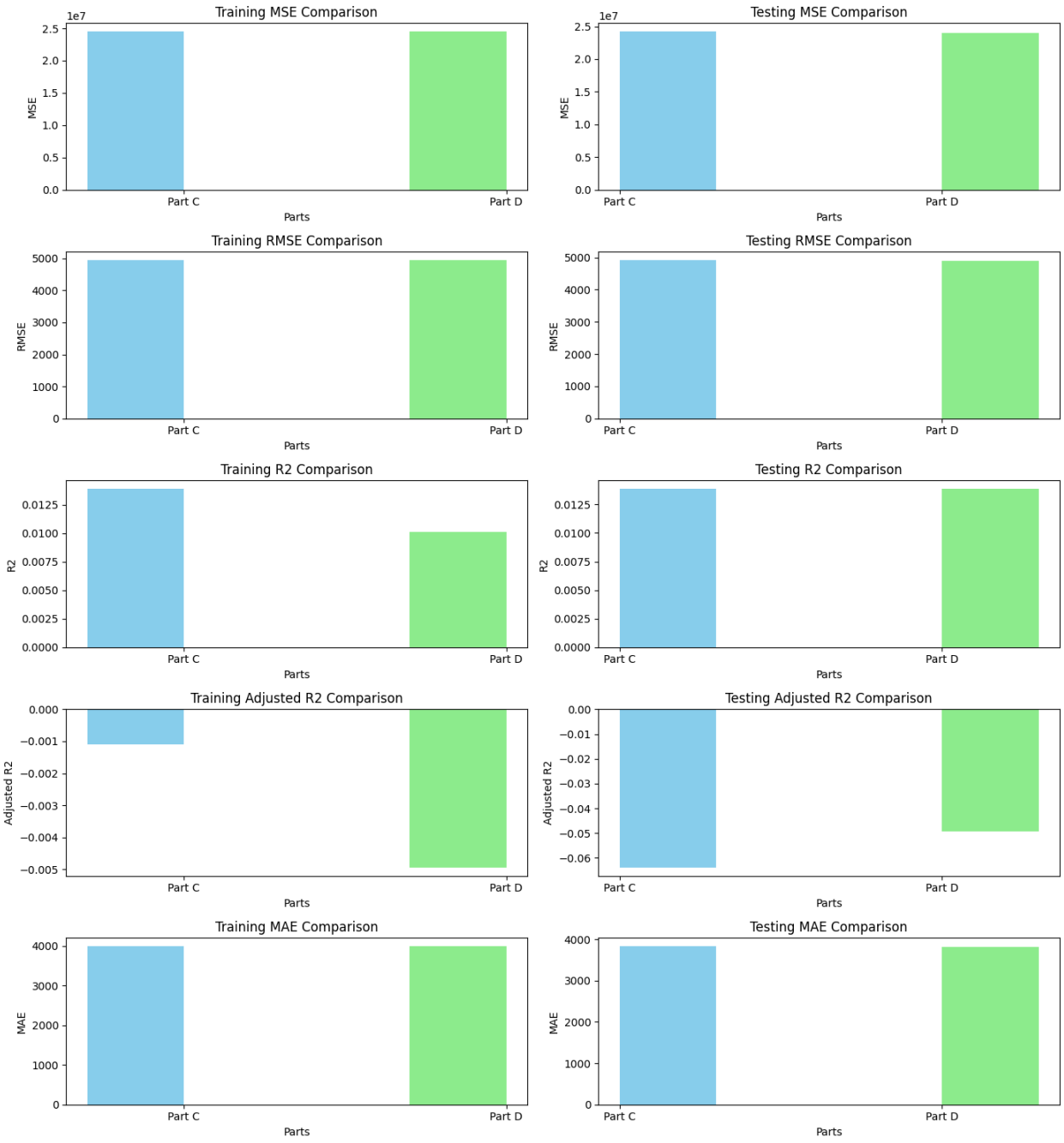
D) Performing RFE and even performing correlation analysis and finding the top 3 features : the top 3 features obtained by both are given below :

```
Selected Features: Index(['Building_Type', 'Green_Certified', 'Number_of_Residents'], dtype='object')
```

The testin and training data accuracy obtained by this is illustrated below :

```
Training Data Accuracy:
Mean Squared Error (MSE):  24569032.90689799
Root Mean Squared Error (RMSE):  4956.715939702212
R2 Score:  0.010134545491283897
Adjusted R2 Score:  -0.004954866925007462
Mean Absolute Error (MAE):  4006.4733775147365

Testing Data Accuracy:
Mean Squared Error (MSE):  23941409.062998377
Root Mean Squared Error (RMSE):  4892.995918964002
R2 Score:  0.01390151386794114
Adjusted R2 Score:  -0.049309927550780674
Mean Absolute Error (MAE):  3813.948128176773
```

By comparing these errors with that obtained with part (c) : following comparisons and insights can be gained :



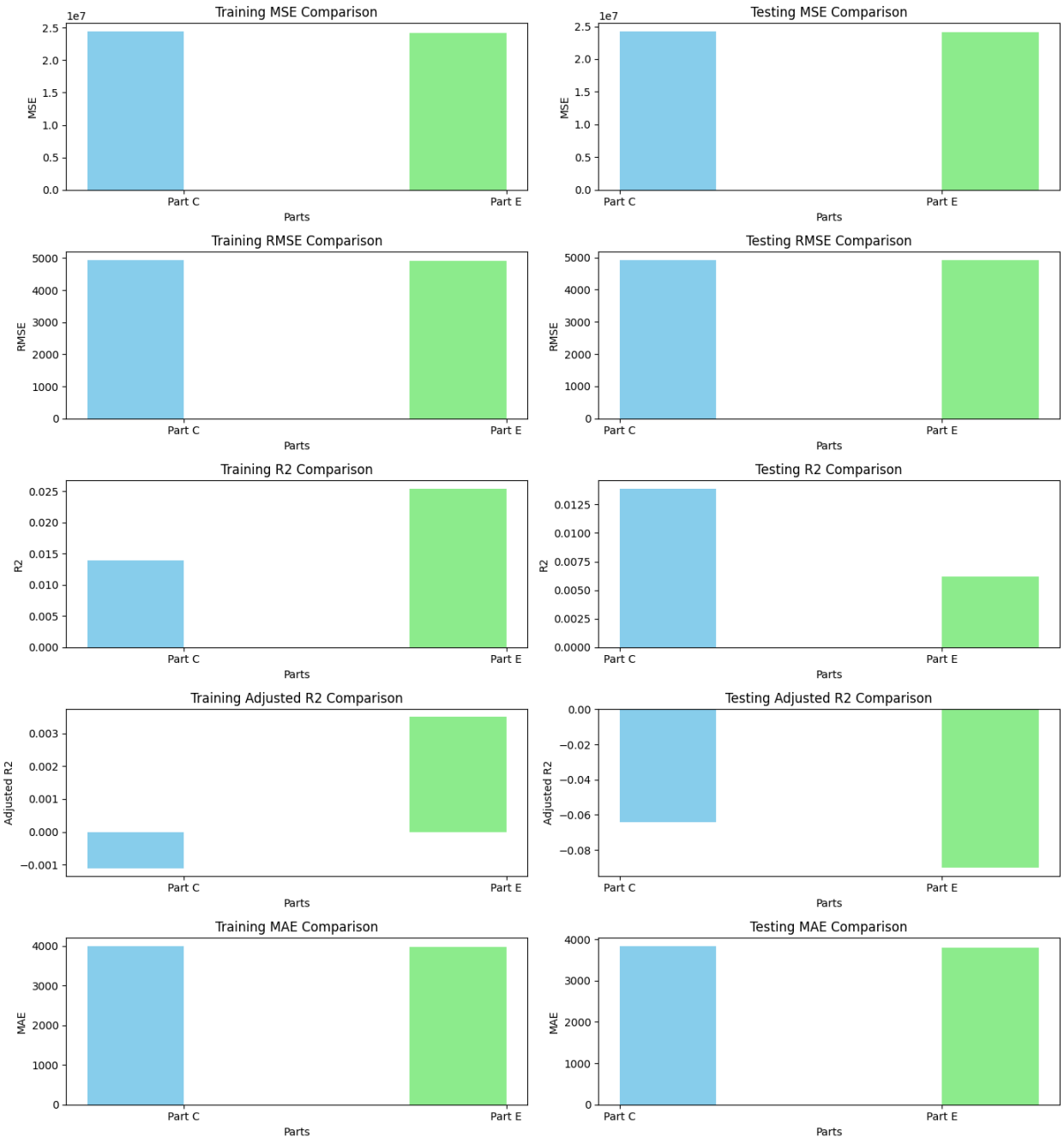
The comparison shows that there is not much significant difference in the error of the 2 models , which can be proved from the fact as most features are very less correlated with the target , However , the MSE of part (c) is less than that of part (d) and even R2 score of part (c) is greater than that of part (d) showing that taking all features is better than taking top 3 features.

E) Performin One Hot Encoding and then performing Ridge Regression on the dataset , following steps were followed :

First , the train-test split was done and then the training and testing features were encoding separately , followed by scaling of the features . Following are the erorrs obtained , the code used and the comparison between part c and e

```
'''
First the data is split into training and testing data and then training and testing data are encoded separately , further scaling of the features is done
'''

cat_features = ["Building_Type","Building_Status","Maintenance_Priority"]
num_features = temp2.drop(columns=cat_features + ['Electricity_Bill']).columns

onehotencoder=OneHotEncoder()

enc_data=onehotencoder.fit(features2[cat_features])

X_traind, X_testd, y_traind, y_testd = train_test_split(features2, target2, test_size=0.2, random_state=42)

enc_data_train= pd.DataFrame(onehotencoder.transform(X_traind[cat_features]).toarray())

X_traind = X_traind[num_features].reset_index(drop=True).join(enc_data_train) # Reseting index of X_train3

enc_data_test=pd.DataFrame(onehotencoder.transform(X_testd[cat_features]).toarray())

X_testd = X_testd[num_features].reset_index(drop=True).join(enc_data_test) # Reseting index of X_test3
```

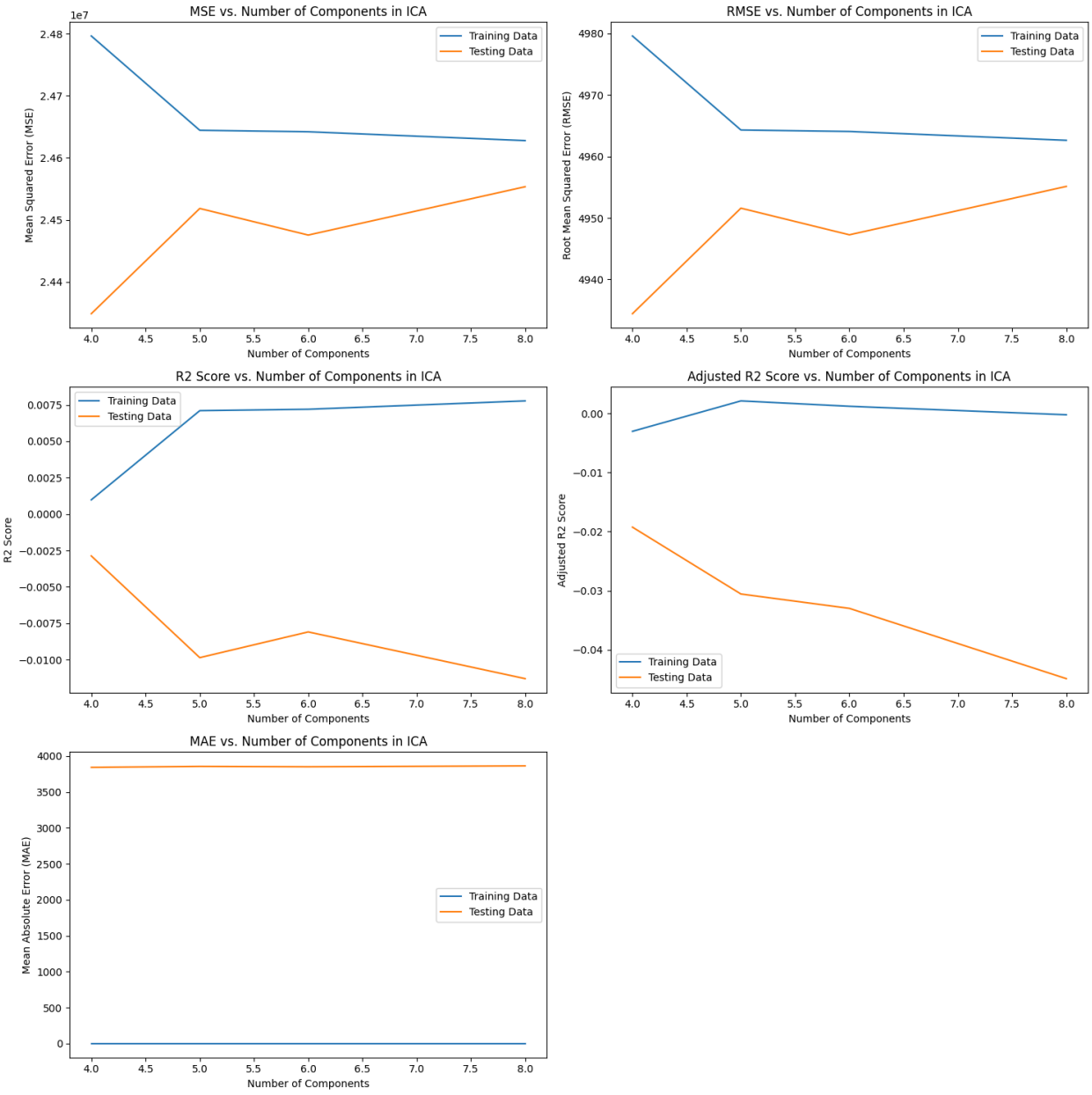
```
Training Data Accuracy:
Mean Squared Error (MSE):  24188931.12871915
Root Mean Squared Error (RMSE):  4918.224387796794
R2 Score:  0.025448523084228958
Adjusted R2 Score:  0.003503658711509483
Mean Absolute Error (MAE):  3976.718619750425

Testing Data Accuracy:
Mean Squared Error (MSE):  24129092.811629098
Root Mean Squared Error (RMSE):  4912.1372956819005
R2 Score:  0.006171197748729096
Adjusted R2 Score:  -0.09014701216108567
Mean Absolute Error (MAE):  3797.50796864784
```



Important observations obtained is that Ridge regression performs better than Linear regression because of the regularisation involved in it which prevents overfitting of the model , the following can also be proved by lower MSE of part e than part c and higher R2 score of part e than part c. Also, OneHot encoding performs better than Label Encoding since it reduces bias and helps better interpret categorical values.

```
Testing Data Accuracy:
Mean Squared Error (MSE):  24348689.73769163
Root Mean Squared Error (RMSE):  4934.439151280683
R2 Score:  -0.002873557962169704
Adjusted R2 Score:  -0.019247003806450058
Mean Absolute Error (MAE):  3841.852647293817

The number of components  5

Training Data Accuracy:
Mean Squared Error (MSE):  24644294.946417
Root Mean Squared Error (RMSE):  4964.30206035219
R2 Score:  0.007102301884543305
Adjusted R2 Score:  0.002107846662634638
Mean Absolute Error (MAE):  518.3014272072571

Testing Data Accuracy:
Mean Squared Error (MSE):  24518301.678271525
Root Mean Squared Error (RMSE):  4951.595871865103
R2 Score:  -0.00985953265545736
Adjusted R2 Score:  -0.0305533755377414
Mean Absolute Error (MAE):  3855.26485007387

The number of components  6

Training Data Accuracy:
Mean Squared Error (MSE):  24641905.07449755
Root Mean Squared Error (RMSE):  4964.061348784637
R2 Score:  0.007198587793017075
Adjusted R2 Score:  0.0011997877192588824
Mean Absolute Error (MAE):  518.3014272072571

Testing Data Accuracy:
Mean Squared Error (MSE):  24475415.142301746
Root Mean Squared Error (RMSE):  4947.2633993251
R2 Score:  -0.008093122496223737
Adjusted R2 Score:  -0.03298431070600705
Mean Absolute Error (MAE):  3850.2843449476304

The number of components  8

Training Data Accuracy:
Mean Squared Error (MSE):  24627663.200284276
Root Mean Squared Error (RMSE):  4962.626643248944
R2 Score:  0.00777238080084941
Adjusted R2 Score:  -0.00023752934404774884
Mean Absolute Error (MAE):  518.3014272072571

Testing Data Accuracy:
Mean Squared Error (MSE):  24553333.73712212
Root Mean Squared Error (RMSE):  4955.132060512829
R2 Score:  -0.011302432699801468
Adjusted R2 Score:  -0.04487263793464957
Mean Absolute Error (MAE):  3862.54118354153
```

F)



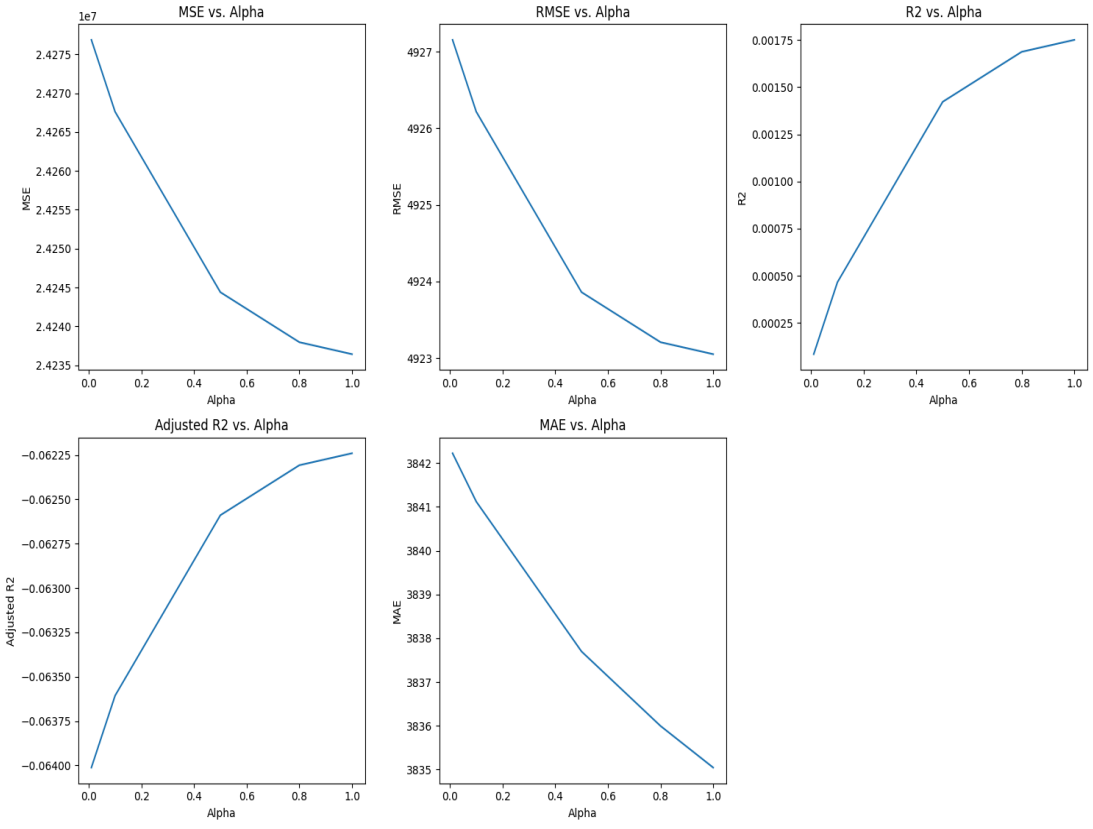
Above are the results obtained for part F with different number of components chosen ,

Insights gained are as the number of components increases MSE on training data decrease , however on testing data it may increase, whereas R2 score increases depicting a better performance of the model . Though there is not much significant difference obtained with MAE error.

Following alphas are used for performing elasticnet regularization on the model :

```
alphas = [0.01, 0.1, 0.5 , 0.8 , 1]
```

G)



Observations obtained :

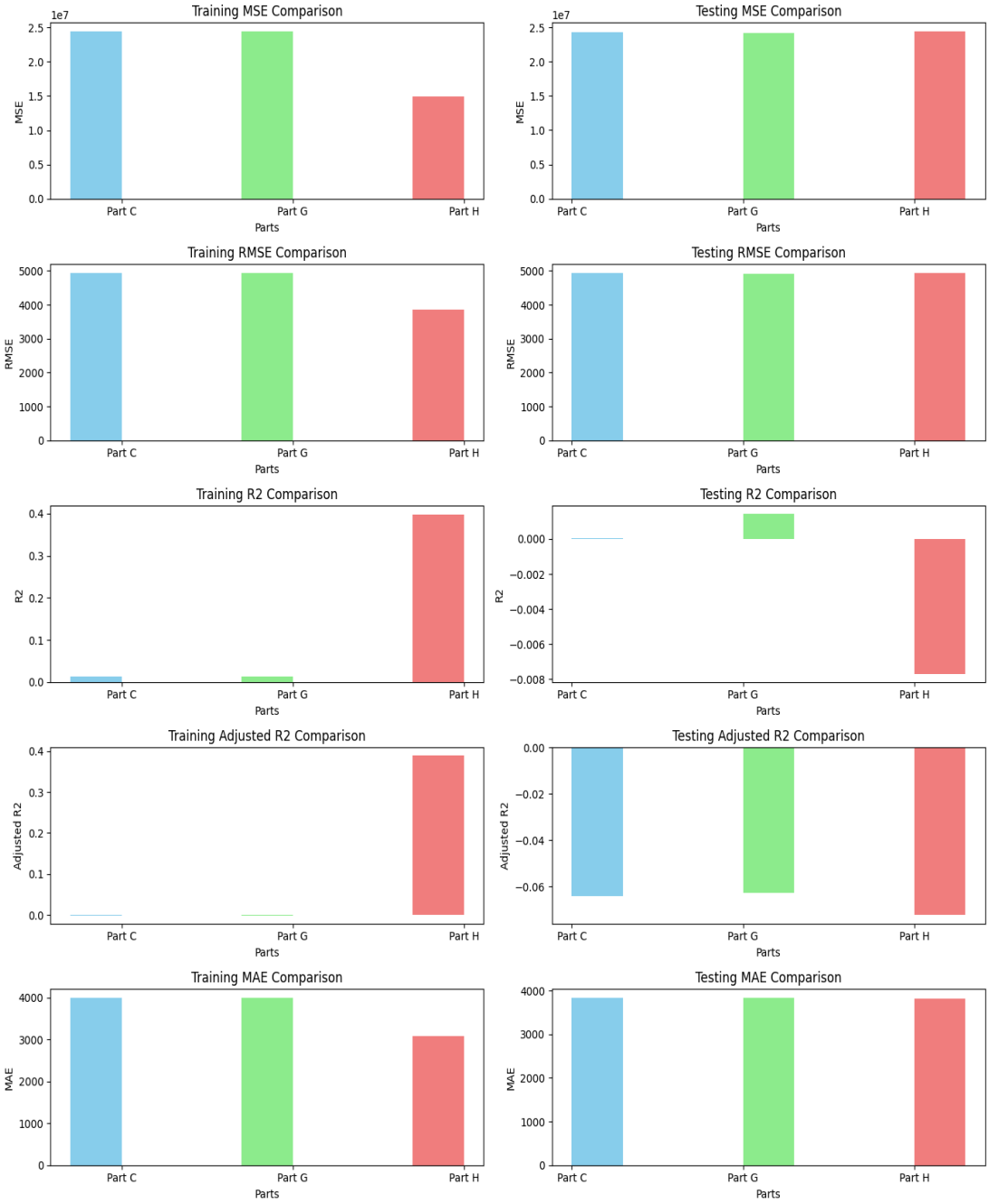
When alpha increases, the regularization term becomes more dominant which results is lower MSE and high R2 score , resulting in better performance of the model. This means:

- Larger alpha reduces the magnitude of the coefficients, shrinking them towards zero.
- If alpha is too small, the model may overfit the training data by learning a more complex function that captures noise in the data.

- As alpha increases, the model simplifies by reducing or eliminating some of the coefficients, which leads to a model that is less complex and less likely to overfit

H) Performing Gradient Boosting Regressor on the model :

```
Training Data Results
Mean Squared Error (MSE):  14926446.25730777
Root Mean Squared Error (RMSE):  3863.4759294329465
R2 score:  0.398626166333897
Adjusted R2 score:  0.38945888228410885
Mean Absolute error:  3092.7481886865007

Testing Data Results
Mean Squared Error (MSE):  24465723.08637143
Root Mean Squared Error (RMSE):  4946.283765249566
R2 score:  -0.007693926206012058
Adjusted R2 score  -0.07228969070639746
Mean Absolute error:  3816.6107652999817
```



Key Observations obtained are :

Gradient Boosting Regressor outforms each model with significantly very lower MSE and R2 score depicting the best performance . Following can be the possible reasons that since Gradient Boosting can use a wide range

of base learners, such as decision trees, and linear models. Gradient Boosting is generally more robust, as it updates the weights based on the gradients, which are less sensitive to outliers and also capture non-linear relationships which models in part c and g could not capture.

Aarzoo , 2022008