In [1]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from sklearn.metrics import confusion_matrix
```

In [422…
```python
df = pd.read_csv('logistic.csv')
df.head()
```

Out[422…

	Y	X1	X2	X3	X4	X5	X6	X7	X8	X9	...	X21	X22	X23	X24	X25	X26	
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	25.38	17.33	184.60	2019.0	0.1622	0.6656	0.7
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4

5 rows × 31 columns

In [5]:
```python
### Function to Split data into Training and Test set

def train_test_split(data):
    # Calculate the split index for 80% of the data
    split_idx = int(len(data) * 0.8)

    # Split the data into training and testing sets
    train_data = data[:split_idx]   # First 80% for training
    test_data = data[split_idx:]    # Last 20% for testing

    return train_data, test_data

# def splitDataSet(inputDataframe, trainSetSize):

#         trainSet = inputDataframe.sample(frac = trainSetSize)
#         testSet = inputDataframe.drop(trainSet.index)
#         return trainSet, testSet
```

One Hot encoding and Data Splitting

In [423…
```python
#One hot encoding
df['Y'] = df['Y'].map({'M':'0', 'B':'1'})
df['Y'].astype(float)

train_data, test_data = train_test_split(df)

Xtrain = train_data.drop('Y', axis=1)
Ytrain = train_data['Y'].astype(float)
Xtrainn = (Xtrain - Xtrain.mean())/Xtrain.std()


Xtests = test_data.drop('Y', axis=1)
Ytests = test_data['Y'].astype(float)
Xtestss = (Xtests - Xtests.mean())/Xtests.std()
```

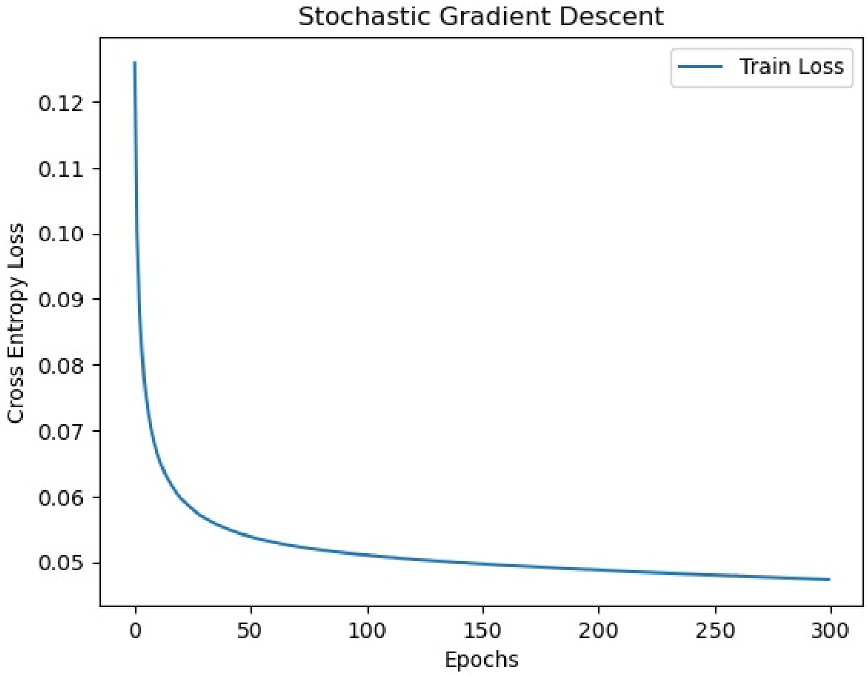
In [ ]:

1A. Extending the Logistic Regression Class to include Stochastic Gradient Descent Method

In [424…
```python
import numpy as np

class Loss:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mean_square_loss(self, theta):
        predictions = np.dot(self.x, theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
```

```
        return gradient

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    def cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        return gradient

    @staticmethod
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))


class Optimization:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loss_obj = Loss(x, y)

    def mean_square_hessian(self):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def cross_entropy_hessian(self, theta):
        p = self.sigmoid(np.dot(self.x, theta))
        W = np.diag(p * (1 - p))
        hess = np.dot(self.x.T, np.dot(W, self.x))
        return hess

    @staticmethod
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))

    def newtons_method(self, theta, lr=0.01, epochs=50, loss_trajectory=None):
        for epoch in range(epochs):
            gradient = self.loss_obj.cross_entropy_gradient(theta)
            hessian = self.cross_entropy_hessian(theta)
            inv_hessian = np.linalg.inv(hessian)
            theta -= lr * np.dot(inv_hessian, gradient)

            # Calculate loss for monitoring
            loss = self.loss_obj.cross_entropy_loss(theta)
            loss_trajectory.append(loss)
        return theta

    def sgd(self, theta, lr=0.01, epochs=50, batch_size=20, loss_trajectory=None):
        n_samples = self.x.shape[0]
        for epoch in range(epochs):

            indices = np.arange(n_samples)
            np.random.shuffle(indices)

            for i in range(0, n_samples, batch_size):
                batch_indices = indices[i:i + batch_size]
                x_batch = self.x[batch_indices]
                y_batch = self.y[batch_indices]

                predictions = Loss.sigmoid(np.dot(x_batch, theta))
                gradient = np.dot(x_batch.T, (predictions - y_batch)) / batch_size
                theta -= lr * gradient
                #print('Updated theta:', theta)

            # Calculate loss for monitoring
            loss = self.loss_obj.cross_entropy_loss(theta)
            loss_trajectory.append(loss)
        return theta


class LogisticRegression:
    def __init__(self):
        self.theta = None
        self.train_loss_trajectory = []

    def fit(self, x, y, lr=0.01, epochs=200, method="sgd", batch_size=1):
        self.theta = np.zeros(x.shape[1])  # Initialize coefficients (theta)
        optimizer = Optimization(x, y)

        if method == "sgd":
            self.theta = optimizer.sgd(self.theta, lr=lr, epochs=epochs, batch_size=batch_size, loss_trajectory=
        elif method == "newton":
            self.theta = optimizer.newtons_method(self.theta, lr=lr, epochs=epochs, loss_trajectory=self.train_
```

```
        return self.theta

    def predict_proba(self, x):
        z = np.dot(x, self.theta)
        return Loss.sigmoid(z)

    def predict(self, x, threshold=0.5):
        return (self.predict_proba(x) >= threshold).astype(int)



if __name__ == "__main__":

    model = LogisticRegression()

    Xtrainns = np.array(Xtrainn)  # Convert Xtrainn to a NumPy array 'cus it's a DataFrame
    Ytrains = np.array(Ytrain)

    model.fit(Xtrainns, Ytrains, lr=0.01, epochs=300, method="sgd", batch_size=1)
    predictions = model.predict(Xtrainns)
    plt.plot(model.train_loss_trajectory, label='Train Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Cross Entropy Loss')
    plt.legend()
    plt.title("Stochastic Gradient Descent")
    plt.show()
```



In [ ]:

B. Extending the Loss Class to include L1/L2 Regularization

In [437…

```
class Loss:
    def __init__(self, x, y, lambda_=0.01):
        self.x = x
        self.y = y
        self.lambda_ = lambda_  ## Adding lambda


    def mean_square_loss(self, theta):
        predictions = np.dot(self.x, theta)
        return np.mean((self.y - predictions) ** 2)

    def mean_square_gradient(self, theta):
        predictions = np.dot(self.x, theta)
        gradient = -2 * np.dot(self.x.T, (self.y - predictions)) / len(self.y)
        return gradient

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    def L2_regularized_cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        loss =  -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))
```

```
        loss -= (self.lambda_) * np.sum(theta ** 2)   ### Adding the L2 regularization
        return loss

    def L1_regularized_cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        loss =  -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))
        loss -= (self.lambda_) * np.sum(np.abs(theta))        ### Adding L1 Regularization
        return loss
    def cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        return gradient

    def L2_regularized_cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        gradient -= 2*(self.lambda_ * theta)


    def L1_regularized_cross_entropy_gradient(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        gradient = np.dot(self.x.T, (predictions - self.y)) / len(self.y)
        gradient -= self.lambda_ * np.sign(theta)

    @staticmethod
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))
```

In [ ]:

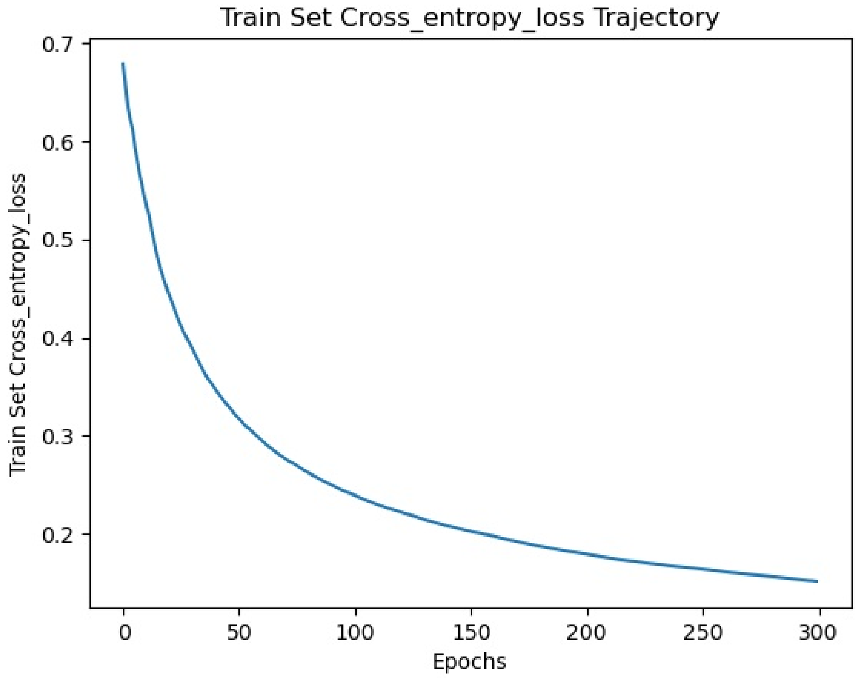
C and E. Fitting the Logistic Regression Models with the combinations as stated and Generating the trajectory, accuracy report

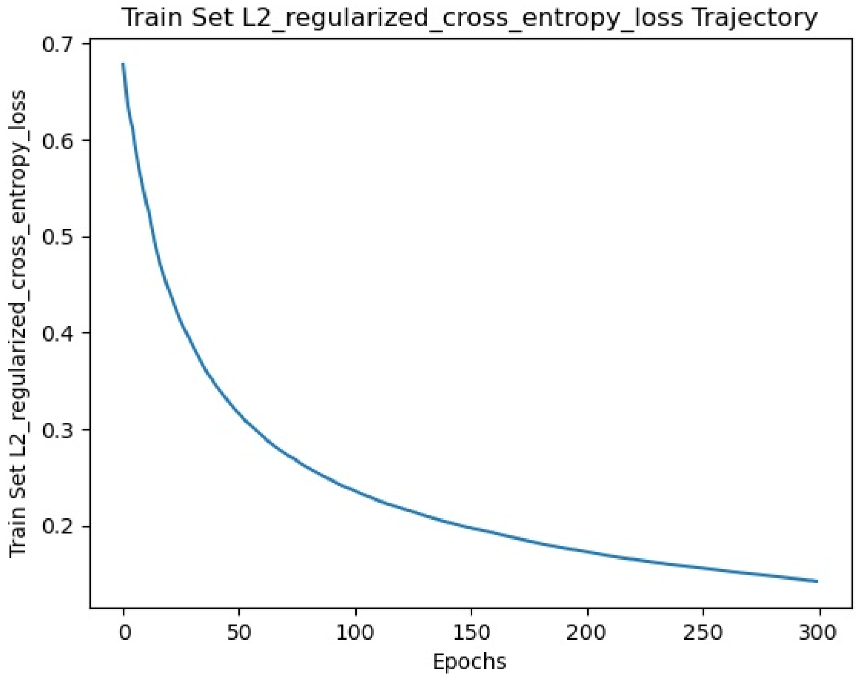
In [ ]:

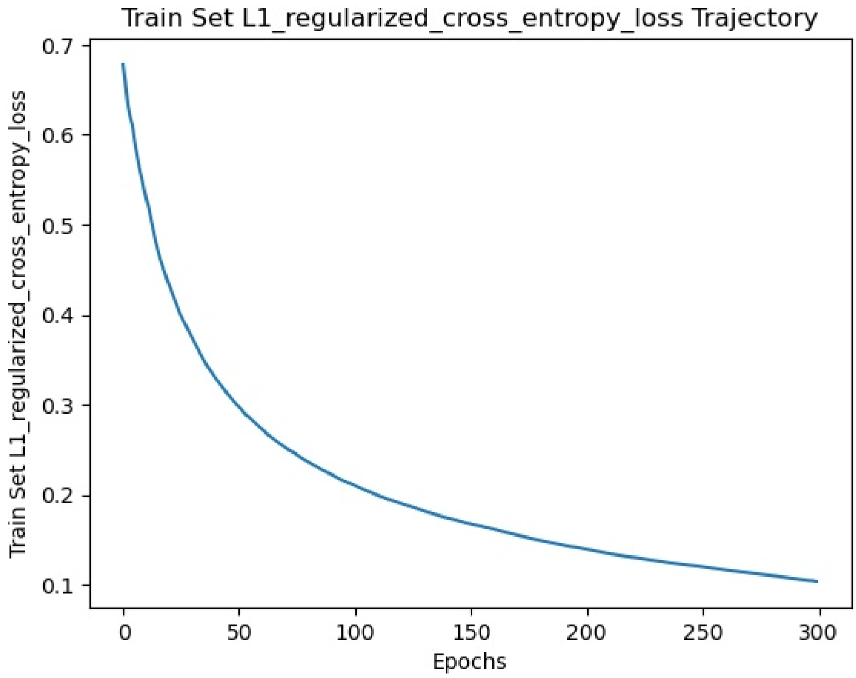
In [440…

```
class Optimization:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loss_obj = Loss(x, y)

    def mean_square_hessian(self):
        return 2 * np.dot(self.x.T, self.x) / len(self.x)

    def cross_entropy_hessian(self, theta):
        p = self.sigmoid(np.dot(self.x, theta))
        W = np.diag(p * (1 - p))
        hess = np.dot(self.x.T, np.dot(W, self.x))
        return hess

    @staticmethod
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))

    def newtons_method(self, theta, lr=0.01, epochs=50, loss_trajectory=None):
        for epoch in range(epochs):
            gradient = self.loss_obj.cross_entropy_gradient(theta)
            hessian = self.cross_entropy_hessian(theta)
            inv_hessian = np.linalg.inv(hessian)
            theta -= lr * np.dot(inv_hessian, gradient)

            # Calculate loss for monitoring
            loss = self.loss_obj.cross_entropy_loss(theta)
            loss_trajectory.append(loss)
        return theta

    def sgd(self, theta, lr=0.01, epochs=50, batch_size=20, loss_trajectory=None,loss_trajectory1=None,loss_tra
        n_samples = self.x.shape[0]
        for epoch in range(epochs):

            indices = np.arange(n_samples)
            np.random.shuffle(indices)

           # for i in range(0, n_samples, batch_size):
            batch_indices = indices[i:i + batch_size]
            x_batch = self.x[batch_indices]
            y_batch = self.y[batch_indices]

            predictions = Loss.sigmoid(np.dot(x_batch, theta))
            gradient = np.dot(x_batch.T, (predictions - y_batch)) / batch_size
            theta -= lr * gradient
```

```
                #print('Updated theta:', theta)

            # Calculate loss for monitoring
            loss = self.loss_obj.cross_entropy_loss(theta)
            loss1 = self.loss_obj.L2_regularized_cross_entropy_loss(theta)
            loss2 = self.loss_obj.L1_regularized_cross_entropy_loss(theta)

            loss_trajectory.append(loss)
            loss_trajectory1.append(loss1)
            loss_trajectory2.append(loss2)

        return theta


class LogisticRegression:
    def __init__(self):
        self.theta = None
        self.train_loss_trajectory = []
        self.train_loss_trajectory1 = []
        self.train_loss_trajectory2 = []

    def fit(self, x, y, lr=0.01, epochs=200, method="sgd", batch_size=1):
        self.theta = np.zeros(x.shape[1])  # Initialize coefficients (theta)
        optimizer = Optimization(x, y)

        if method == "sgd":
            self.theta = optimizer.sgd(self.theta, lr=lr, epochs=epochs, batch_size=batch_size, loss_trajectory=
                                       loss_trajectory1=self.train_loss_trajectory1, loss_trajectory2=self.trai
        elif method == "newton":
            self.theta = optimizer.newtons_method(self.theta, lr=lr, epochs=epochs, loss_trajectory=self.train_
        return self.theta

    def predict_proba(self, x):
        z = np.dot(x, self.theta)
        return Loss.sigmoid(z)

    def predict(self, x, threshold=0.5):
        return (self.predict_proba(x) >= threshold).astype(int)


if __name__ == "__main__":

    model = LogisticRegression()

    Xtrainns = np.array(Xtrainn)  # Convert Xtrainn to a NumPy array if it's a DataFrame
    Ytrains = np.array(Ytrain)

    model.fit(Xtrainns, Ytrains, lr=0.01, epochs=300, method="sgd", batch_size=20)
    predictions = model.predict(Xtrainns)
    loss_trajectories = [model.train_loss_trajectory, model.train_loss_trajectory1, model.train_loss_trajectory2
    labels = ['Train Set Cross_entropy_loss', 'Train Set L2_regularized_cross_entropy_loss', 'Train Set L1_regu

for i, trajectory in enumerate(loss_trajectories):
    plt.figure()
    plt.plot(loss_trajectories[i], label=labels[i])
    plt.xlabel('Epochs')
    plt.ylabel(labels[i])
    plt.title(f"{labels[i]} Trajectory")
   # plt.legend()
    plt.show()
```



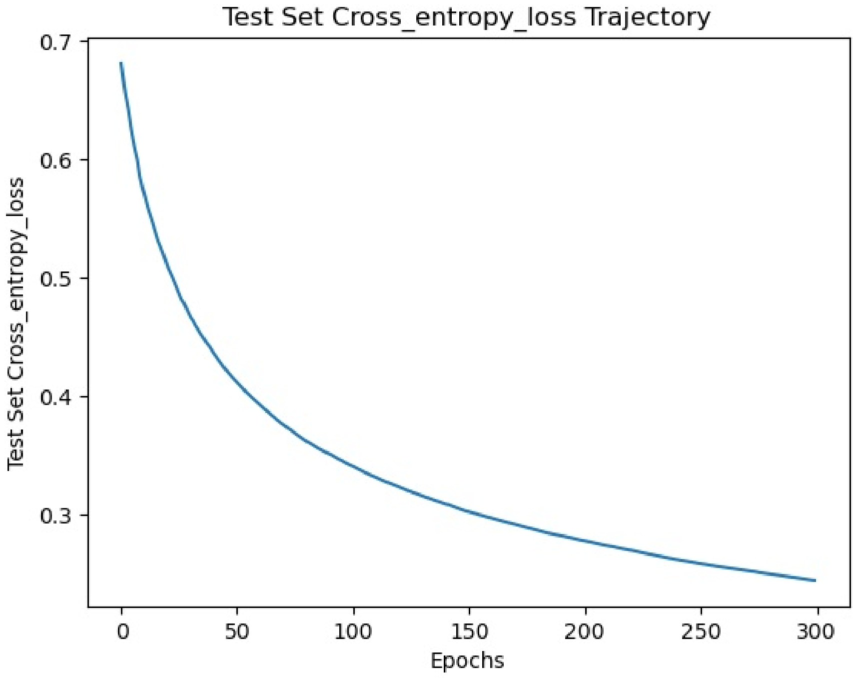


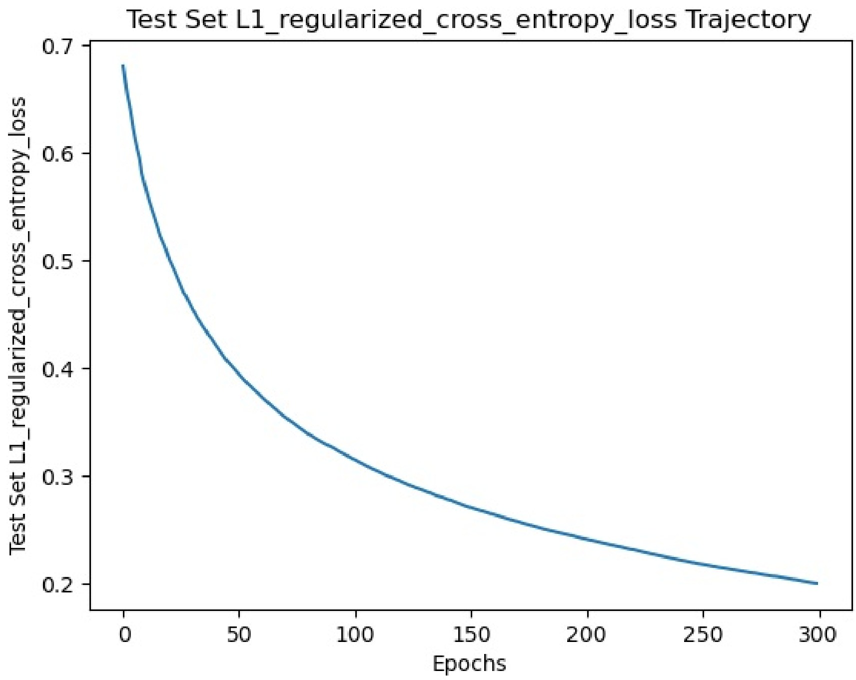


In [441…

```
if __name__ == "__main__":
```

```
    model2 = LogisticRegression()

    Xtestss = np.array(Xtestss)  # Convert Xtrainn to a NumPy array if it's a DataFrame
    Ytests = np.array(Ytests)

    model2.fit(Xtestss, Ytests, lr=0.01, epochs=300, method="sgd", batch_size=20)
    predictions2 = model2.predict(Xtestss)
    loss_trajectories2 = [model2.train_loss_trajectory, model2.train_loss_trajectory1, model2.train_loss_trajec
    labels = ['Test Set Cross_entropy_loss', 'Test Set L2_regularized_cross_entropy_loss', 'Test Set L1_regular

for i, trajectory in enumerate(loss_trajectories):
    plt.figure()
    plt.plot(loss_trajectories2[i], label=labels[i])
    plt.xlabel('Epochs')
    plt.ylabel(labels[i])
    plt.title(f"{labels[i]} Trajectory")
   # plt.legend()
    plt.show()
```





In [442…

```
accuracy = np.mean(predictions == Ytrains)
print("Accuracy for Training Dataset :", (accuracy*100))

accuracy2 = np.mean(predictions2 == Ytests)
print("Accuracy for Test Dataset:", (accuracy2*100))
```

```
Accuracy for Training Dataset : 97.14285714285714
Accuracy for Test Dataset: 95.6140350877193
```

In [ ]:

Confusion Matrix

In [446…

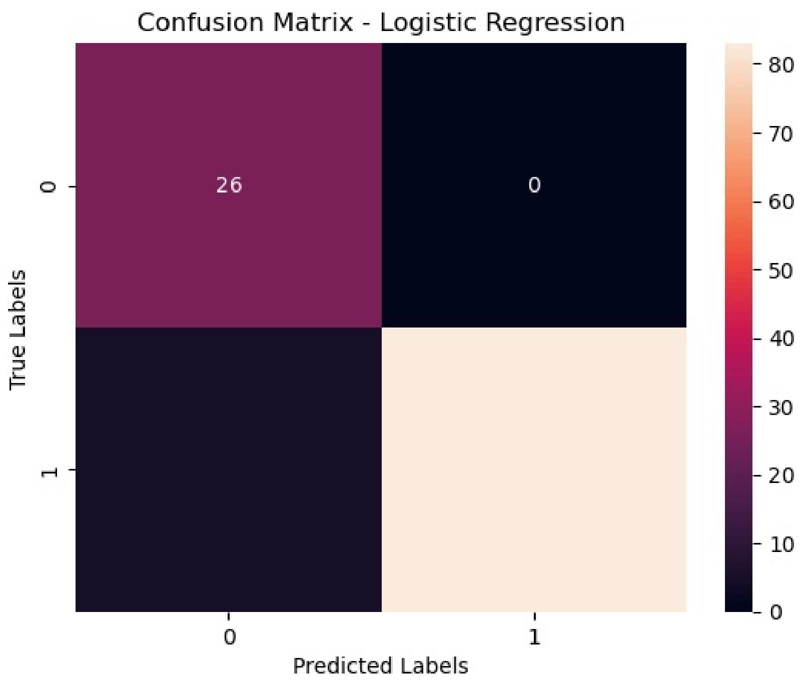
```
cm_lr = confusion_matrix(Ytests, predictions2)
cm_lr
```

Out[446…

```
array([[26,  0],
       [ 5, 83]], dtype=int64)
```

In [445…

```
sns.heatmap(cm_lr, annot=True, fmt='d')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Logistic Regression')
plt.show()
```



In [ ]:

True Positives (TP) = 26: The model correctly predicted 26 instances as positive. True Negatives (TN) = 83: The model correctly predicted 83 instances as negative. False Positives (FP) = 5: The model incorrectly predicted 5 instances as positive when they were actually negative. False Negatives (FN) = 0: The model did not miss any actual positive instances

Accuracy= TP+TN / FP+ FN + TP +TN = 95.6%

In [ ]:

In [ ]:

In [435…

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class Loss:
    def __init__(self, x, y, lambda_=0.01):
        self.x = x
        self.y = y
        self.lambda_ = lambda_  ## Adding lambda

    def cross_entropy_loss(self, theta):
        predictions = self.sigmoid(np.dot(self.x, theta))
        return -np.mean(self.y * np.log(predictions) + (1 - self.y) * np.log(1 - predictions))

    @staticmethod
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))

class Optimization:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loss_obj = Loss(x, y)

    def sgd(self, theta, lr=0.01, epochs=50, batch_size=20, loss_trajectory=None):
        n_samples = self.x.shape[0]
        for epoch in range(epochs):
            indices = np.arange(n_samples)
            np.random.shuffle(indices)

            for i in range(0, n_samples, batch_size):
                batch_indices = indices[i:i + batch_size]
                x_batch = self.x[batch_indices]
                y_batch = self.y[batch_indices]

                predictions = Loss.sigmoid(np.dot(x_batch, theta))
                gradient = np.dot(x_batch.T, (predictions - y_batch)) / batch_size
                theta -= lr * gradient

                # Calculate loss for monitoring
                loss = self.loss_obj.cross_entropy_loss(theta)
                loss_trajectory.append(loss)
```

```python
        return theta

class LogisticRegression:
    def __init__(self):
        self.theta = None
        self.train_loss_trajectory = []

    def fit(self, x, y, lr=0.01, epochs=200, batch_size=1):
        self.theta = np.zeros(x.shape[1])  # Initialize coefficients (theta)
        optimizer = Optimization(x, y)
        self.theta = optimizer.sgd(self.theta, lr=lr, epochs=epochs, batch_size=batch_size, loss_trajectory=sel
        return self.theta

    def predict_proba(self, x):
        z = np.dot(x, self.theta)
        return Loss.sigmoid(z)

    def predict(self, x, threshold=0.5):
        return (self.predict_proba(x) >= threshold).astype(int)

# AIC calculation function based on cross-entropy loss and number of parameters
def compute_aic(model, X, y):
    predictions = model.predict_proba(X)
    loss = Loss(X, y).cross_entropy_loss(model.theta)
    num_params = len(model.theta)
    return 2 * num_params + 2 * loss * len(y)
    #return -2 * loss + 2 * num_params

# Backward Feature Selection with AIC
def backward_feature_selection(X, y, min_features=1):
    selected_features = list(X.columns)
    best_aic = float('inf')  # Initialize with a very high AIC value
    best_features = selected_features.copy()

    while len(selected_features) >= min_features:
        # Train model with the current set of selected features
        X_selected = X[selected_features]
        model = LogisticRegression()
        model.fit(X_selected.values, y, lr=0.01, epochs=300, batch_size=20)

        # Compute AIC for the model with the current features
        current_aic = compute_aic(model, X_selected.values, y)

        # Initialize variables to track best feature to remove
        worst_feature = None
        best_aic_with_removal = current_aic

        # Try removing each feature and calculate the resulting AIC
        for feature in selected_features:
            temp_features = [f for f in selected_features if f != feature]
            X_temp = X[temp_features]

            # Train model without this feature
            temp_model = LogisticRegression()
            temp_model.fit(X_temp.values, y, lr=0.01, epochs=300, batch_size=20)
            temp_aic = compute_aic(temp_model, X_temp.values, y)

            # Check if this removal results in a lower AIC
            if temp_aic < best_aic_with_removal:
                best_aic_with_removal = temp_aic
                worst_feature = feature

        # If removing the worst feature improves AIC, update selected features
        if worst_feature is not None and best_aic_with_removal < current_aic:
            selected_features.remove(worst_feature)
            best_aic = best_aic_with_removal
            best_features = selected_features.copy()
        else:
            # Stop if no improvement is possible
            break

    return best_features, best_aic

# Example usage
if __name__ == "__main__":

    Xtrainnz = Xtrainn.copy()  # Convert Xtrainn to a NumPy array if it's a DataFrame
    Ytrainz = np.array(Ytrain.copy())

    # Perform backward feature selection
    best_features, best_aic = backward_feature_selection(Xtrainnz, Ytrainz, min_features=1)
    print("Selected Features:", best_features)
```

```
    print("Best AIC:", best_aic)
```

```
Selected Features: ['X2', 'X11', 'X20', 'X21', 'X23', 'X25', 'X28']
Best AIC: 80.33208544091798
```

In [ ]:

In [ ]:

In [ ]:

2. K-Fold Cross Validation

In [271…
```
Xtestss = np.array(Xtestss)  # Convert Xtest to a NumPy array if it's a DataFrame (Already did for training dat
Ytests = np.array(Ytests)
```

In [135…
```
Xtrain = train_data.drop('Y', axis=1)
Ytrain = train_data['Y'].astype(float)
Xtrainn = (Xtrain - Xtrain.mean())/Xtrain.std()


Xtests = test_data.drop('Y', axis=1)
Ytests = test_data['Y'].astype(float)
Xtestss = (Xtests - Xtests.mean())/Xtests.std()
```

In [136…
```
y_train_bank=pd.DataFrame(Ytrain.values.reshape(-1,1))
y_test_bank=pd.DataFrame(Ytests.values.reshape(-1,1))
x_train_bank=pd.DataFrame(Xtrainn.values)
x_test_bank=pd.DataFrame(Xtestss.values)
```

In [137…
```
print('x_train_bank :',x_train_bank.shape)
print('x_test_bank :',x_test_bank.shape)
print('y_train_bank :',y_train_bank.shape)
print('y_test_bank :',y_test_bank.shape)
```

```
x_train_bank : (455, 30)
x_test_bank : (114, 30)
y_train_bank : (455, 1)
y_test_bank : (114, 1)
```

In [ ]:

In [365…
```
def logistic_function(X, beta):
    z = np.dot(X,beta)
    return 1 / (1 + np.exp(-z))

def log_likelihood(x, y, beta):
    z = np.dot(x, beta)
    log = np.sum( y*z - np.log(1 + np.exp(z)) )
    return log

#L2-regularized cross-entropy loss

betas = lambda x,y,beta,alpha,lamda : beta-alpha*(-2*np.dot(x.T,y-logistic_function(x, beta))+(2*lamda)*beta)

def stochastic_gradient_descent(x_train,y_train,alpha,epochs,lamda,x_test,y_test):
    m_train,n_features = np.shape(x_train)
    ini_alpha          = alpha
    beta_hat           = np.random.random(n_features).reshape(-1,1)
    logtrain          = []
    logtest           = []
    y_hat              = logistic_function(x_train,beta_hat)

    chunk_size = 20

    for i in range(epochs):
        for chunk in range(len(x_train)//chunk_size):
            x_chunk  = x_train[chunk*chunk_size:min((chunk+1)*chunk_size,len(x_train))]
            y_chunk  = y_train[chunk*chunk_size:min((chunk+1)*chunk_size,len(y_train))]

            beta_hat = betas(x_chunk,y_chunk,beta_hat,alpha,lamda)

            y_hat=logistic_function(x_train,beta_hat)

            logtest.append(log_likelihood(x_test,y_test,beta_hat))
            logtrain.append(log_likelihood(x_train, y_train,beta_hat))

    return logtest,logtrain,beta_hat
```

In [379…
```
import math as Math
```

```python
def gridsearch(alpha,lamda):
    comb=[]
    for i in range(0,len(alpha)):
        for k in range(0,len(lamda)):
            comb.append(dict([('alpha',alpha[i]),('lamda',lamda[k])]))
    return comb

def data_k_divide(data,k):
    k_size=Math.floor(len(data)/k)
    k_data=[]
    c=0
    for i in range (0,k):
        data_set=pd.DataFrame(data.head(0))
        for j in range(i*k_size,(i*k_size)+k_size):
            #data_set=data_set.append(data.iloc[j])
            data_set=pd.concat([data_set, data.iloc[[j]]])
            c=c+1
        k_data.append(data_set)

    #adding datas which are remaining at the end of k division
    # for j in range(c,len(data)):
    #     k_data[k-1]=k_data[k-1].append(data.iloc[j])
    return k_data

def k_data_train_test(x,y,k):
    k_folded_data=[]
    for i in range(0,k):
        x_test=x[i]
        y_test=y[i]
        x_train=pd.DataFrame()
        y_train=pd.DataFrame()
        for j in range(0,k):
            if i!=j:

                x_train = pd.concat([x_train, x[j]])
                y_train = pd.concat([y_train, y[j]])


        final_data=dict([('x',x_train),('y',y_train),('xt',x_test),('yt',y_test)])
        k_folded_data.append(final_data)
    return k_folded_data

def kfold(x_train,y_train,k,x_test,y_test):
    x_train_k=data_k_divide(x_train,k)
    y_train_k=data_k_divide(y_train,k)
    data=k_data_train_test(x_train_k,y_train_k,k)
    return data
```

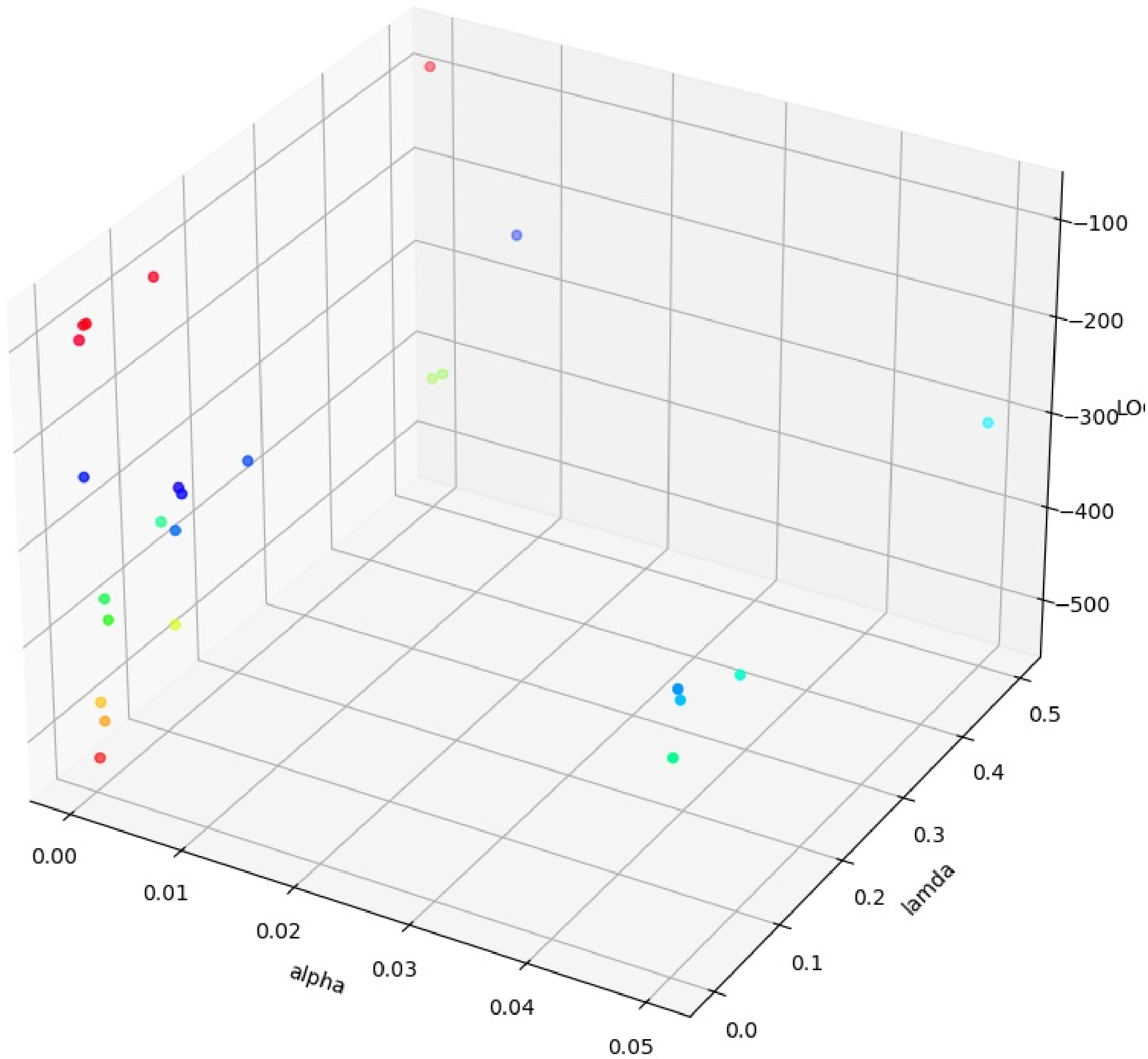
In [513…

```python
import warnings

# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)

alpha=[0.0001,0.00001,0.001,0.05,0.008]
lamda=[0.0000001,0.1,0.5,0.01,0.006]
epochs=20
k=3
parameter=gridsearch(alpha,lamda)
log_test=[]
log_last=[]
log_train=[]
alpha_com=[]
lamda_com=[]
avg_log=[]

## Just commenting to save space

for i in range (0,len(parameter)):
    k_folded_data=kfold(x_train_bank,y_train_bank,k,x_test_bank,y_test_bank)
    for j in range(0,k):
            logtest,logtrain,beta_hat =stochastic_gradient_descent(k_folded_data[j]['x'],k_folded_data[j]['y'],
            log_last.append(logtest[-1])
            log_test.append(logtest)
            log_train.append(logtrain)
    alpha_com.append(parameter[i]['alpha'])
    lamda_com.append(parameter[i]['lamda'])
    avg_log.append(np.mean(log_last))
```

In [384…

```python
from mpl_toolkits import mplot3d
fig = plt.figure()
fig.set_figheight(10)
fig.set_figwidth(10)
ax = plt.axes(projection= '3d')
```

```
ax.scatter3D(alpha_com, lamda_com, avg_log, c=avg_log, cmap='hsv')
ax.set_xlabel('alpha')
ax.set_ylabel('lamda')
ax.set_zlabel('LOG')
plt.show()
```



In [397…

```
combined_df = pd.DataFrame({
    'alpha': alpha_com,
    'lamda': lamda_com,
    'avg_log': avg_log
})
combined_df.sort_values('avg_log', ascending=False)
```

Out[397...

	alpha	lamda	avg_log
4	0.00010	6.000000e-03	-84.380376
2	0.00010	5.000000e-01	-84.741091
3	0.00010	1.000000e-02	-85.143950
1	0.00010	1.000000e-01	-88.523028
0	0.00010	1.000000e-07	-96.002594
24	0.00800	6.000000e-03	-220.392050
23	0.00800	1.000000e-02	-229.112417
5	0.00001	1.000000e-07	-233.649491
22	0.00800	5.000000e-01	-238.570482
21	0.00800	1.000000e-01	-248.757093
20	0.00800	1.000000e-07	-260.047036
19	0.05000	6.000000e-03	-272.495024
18	0.05000	1.000000e-02	-286.111628
17	0.05000	5.000000e-01	-301.280896
16	0.05000	1.000000e-01	-318.123547
15	0.05000	1.000000e-07	-337.281161
6	0.00001	1.000000e-01	-339.223252
14	0.00100	6.000000e-03	-358.811995
13	0.00100	1.000000e-02	-383.350229
12	0.00100	5.000000e-01	-411.505411
7	0.00001	5.000000e-01	-419.611547
11	0.00100	1.000000e-01	-444.288197
8	0.00001	1.000000e-02	-473.424552
10	0.00100	1.000000e-07	-483.293431
9	0.00001	6.000000e-03	-530.000710

Using the best parameters alpha,lamda

In [398...

```
## We alreadey generated the Optimal Betas in the codes above

#beta_hat3
def predict(x, threshold=0.5):
    return (x >= threshold).astype(int)

def predict_proba(x,beta):
    z = np.dot(x, beta)
    return 1 / (1 + np.exp(-z))

pre_values4 = predict_proba(x_test_bank,beta_hat)
pre_probality4 = predict(pre_values4)

### Now fitting it to our test data

accuracy4 = np.mean(pre_probality4 == y_test_bank)
print(" Final Accuracy for Test Dataset - Optimum hyper-parameters:", (accuracy4 * 100))
```

```
Final Accuracy for Test Dataset - Optimum hyper-parameters: 92.10526315789474
```

In [385...

```
# logtest1, logtrain1, beta_hat1 = stochastic_gradient_descent(
#     x_train_bank,
#     y_train_bank,
#     alpha=0.0001,
#     epochs=20,
#     lamda=0.0000001,
#     x_test=x_test_bank,
#     y_test=y_test_bank
# )
```

In [389...

In [ ]:

3 Coordinate Descent for L1-Regularized Linear Regression

A. L1-Regularized Linear Regression

In [360…

```
df1 = pd.read_csv('regression2.csv')
df1.head()
```

Out[360…

	x1	x2	x3	x4	x5	x6	x7	x8	x9	y
0	0.496714	-0.138264	0.647689	1.523030	-0.234153	-0.068678	0.419500	0.998859	0.210385	10.042597
1	-0.234137	1.579213	0.767435	-0.469474	0.542560	-0.369752	0.588956	-0.452418	0.433443	-6.907481
2	-0.463418	-0.465730	0.241962	-1.913280	-1.724918	0.215827	0.058546	-0.941923	1.002438	-11.705652
3	-0.562288	-1.012831	0.314247	-0.908024	-1.412304	0.569502	0.098751	-0.788289	0.880582	-4.990023
4	1.465649	-0.225776	0.067528	-1.424748	-0.544383	-0.330909	0.004560	-0.989354	0.434624	-2.982742

In [ ]:

In [399…

```
train_data2, test_data2 = train_test_split(df1)

Xtrain2 = train_data2.drop('y', axis=1)
Ytrain2 = train_data2['y']
Xtrainn2 = (Xtrain2 - Xtrain2.mean())/Xtrain2.std()
Ytrainn2 = (Ytrain2 - Ytrain2.mean())/Ytrain2.std()


Xtests2 = test_data2.drop('y', axis=1)
Ytests2 = test_data2['y'].astype(float)
Xtestss2 = (Xtests2 - Xtests2.mean())/Xtests2.std()
Ytestss2 = (Ytests2 - Ytests2.mean())/Ytests2.std()
```

In [453…

```
def mean_square_loss(x,y,theta,lambda_):
    predictions = np.dot(x, theta)
    loss = np.mean((y - predictions) ** 2)
    loss+= (lambda_) * np.sum(np.abs(theta))
    return loss

def mean_square_gradient(x,y,theta,lambda_):
    predictions = np.dot(x, theta)
    gradient = -2 * np.dot(x.T, (y - predictions)) / len(y)
    gradient+= lambda_ * np.sign(theta)
    return gradient


def sgd(x,y,theta, lr, epochs, batch_size, loss_trajectory):
    n_samples = x.shape[0]
    for epoch in range(epochs):

        indices = np.arange(n_samples)
        np.random.shuffle(indices)

        for i in range(0, n_samples, batch_size):
            batch_indices = indices[i:i + batch_size]
            x_batch = x[batch_indices]
            y_batch = y[batch_indices]

            predictions = np.dot(x_batch, theta)
            gradient = mean_square_gradient(x_batch,y_batch,theta,lambda_)
            theta -= lr * gradient
                #print('Updated theta:', theta)

            # Calculate loss for monitoring
        loss = mean_square_loss(x,y,theta,lambda_)

        loss_trajectory.append(loss)

    return theta,loss_trajectory

lambda_ = 0.5
Xtrainns2 = np.array(Xtrainn2)  # Convert Xtrainn to a NumPy array if it's a DataFrame
Ytrains2 = np.array(Ytrainn2)
theta = np.zeros(Xtrainns2.shape[1])
```

In [454…

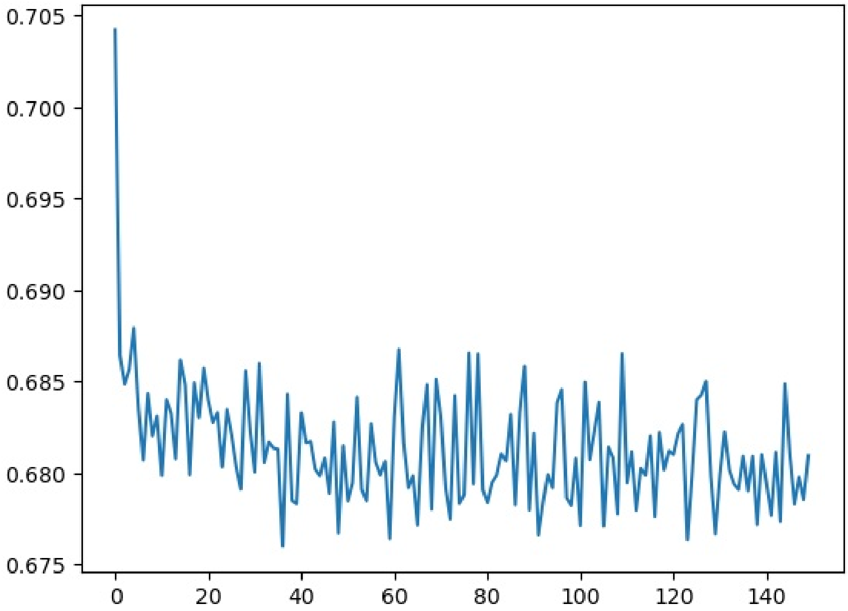
```
ttheta,traj = sgd(Xtrainns2,Ytrains2,theta, lr=0.01, epochs=150, batch_size=20, loss_trajectory=[])
predictionss = np.dot(Xtrain2, ttheta)
print('Optimum Theta after iterations are :', ttheta)
```

```
Optimum Theta after iterations are : [ 0.20284814 -0.05139114  0.01080433  0.48773802 -0.00337104  0.00531932
  0.00295894  0.03459513 -0.00196837]
```

In [455…

```
plt.plot(traj)
```

Out[455… [<matplotlib.lines.Line2D at 0x1f5a63df190>]



In [456…
```
from sklearn.metrics import mean_squared_error, r2_score
r2 = r2_score(Ytrains2, predictionss)
print("R-squared (R2):", r2)
```

R-squared (R2): 0.7005693847824108

In [ ]:

B. Coordinate Descent

In [ ]:

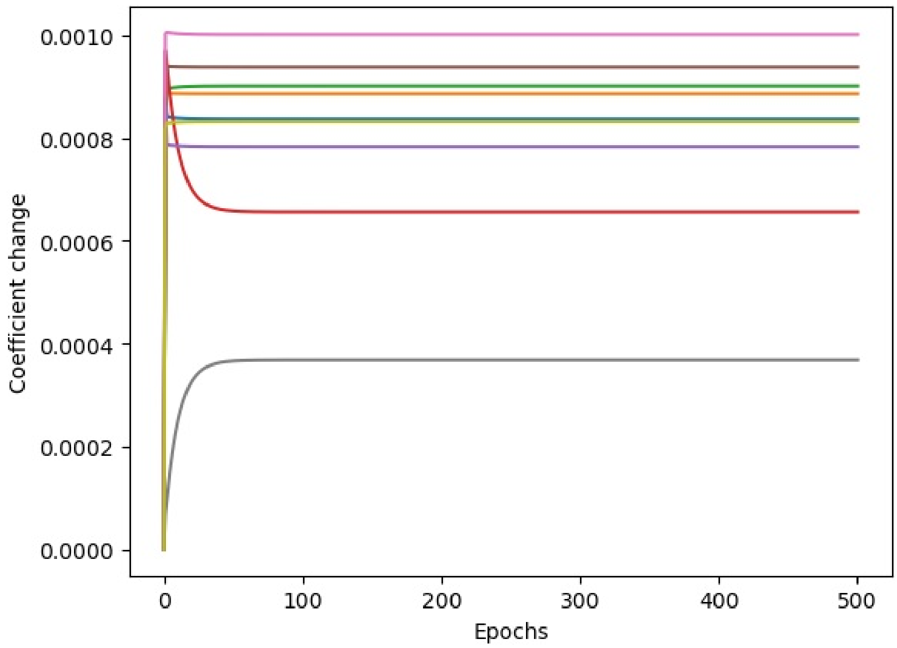
In [511…
```
def soft_threshold(beta, reg):

    if beta < reg:
        return beta + reg
    elif beta > reg:
        return beta - reg
    else:
        return 0

# Coordinate Descent algorithm for Lasso regression
def coordinate_descent(x, y, lambda_, epochs):
    m_train, n_features = np.shape(x)
    beta = np.zeros(n_features).reshape(-1, 1)
    beta_hist = np.zeros((epochs + 1, n_features))  # To track the evolution of coefficients

    # Coordinate descent loop
    for j in range(epochs):
        for i in range(n_features):

            # Choosing the feature (coordinate) to update
            _x = x[:, i]
            x_coor = np.delete(x, i, axis=1)  # All features except i-th
            beta_coor = np.delete(beta, i, axis=0)  # All coefficients except i-th

            # Compute the linear part of the update
            col = x_coor.dot(beta_coor)
            num = ((y - col).T).dot(_x)  # Numerator
            den = _x.T.dot(_x)  # Denominator
            update = num / den

            # Apply soft-thresholding for L1 regularization (Lasso)
            reg = lambda_ / (_x.T.dot(_x))  # Regularization term
            update = soft_threshold(np.mean(update), reg)  # Apply soft-thresholding

            # the update should be scalar(I think..), and then update the coefficient
            beta[i] = update

        # Store the coefficients after each epoch
        beta_hist[j + 1] = beta.ravel()

    return beta, beta_hist

lambda_ = 0.7
```

```
epochs = 500

beta, beta_hist = coordinate_descent(Xtrainns2, Ytrains2, lambda_, epochs)

# The final coefficients are in `beta` while `beta_hist` tracks the evolution of the coefficients
print("Final coefficients:", beta)
#rint(beta_hist)
```

```
Final coefficients: [[0.00083667]
 [0.00088599]
 [0.00090127]
 [0.00065628]
 [0.00078307]
 [0.00093803]
 [0.00100089]
 [0.00036851]
 [0.00083157]]
```

In [512…

```
plt.plot(beta_hist)
plt.xlabel('Epochs')
plt.ylabel('Coefficient change')
plt.show()
```



In [ ]:

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js