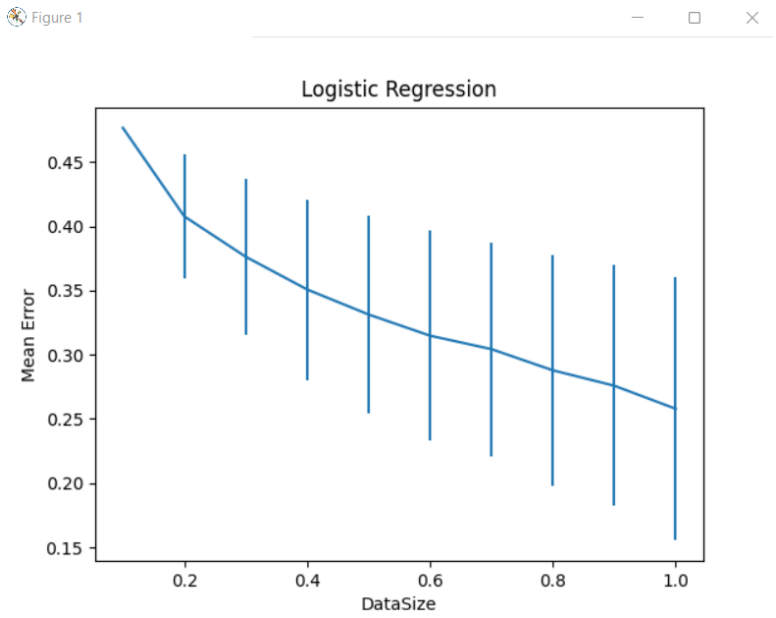
ML PP3 Report

1. Logistic Regression

Dataset: A.csv

Plot:



Runtime and Iterations:

```
"C:\Users\Abhishek Sharma\PycharmProjects\GLM\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/GLM/main.py"
Average No of Iterations 85
Runtime: 7.753549337387085 seconds
```

Discussion of the result:

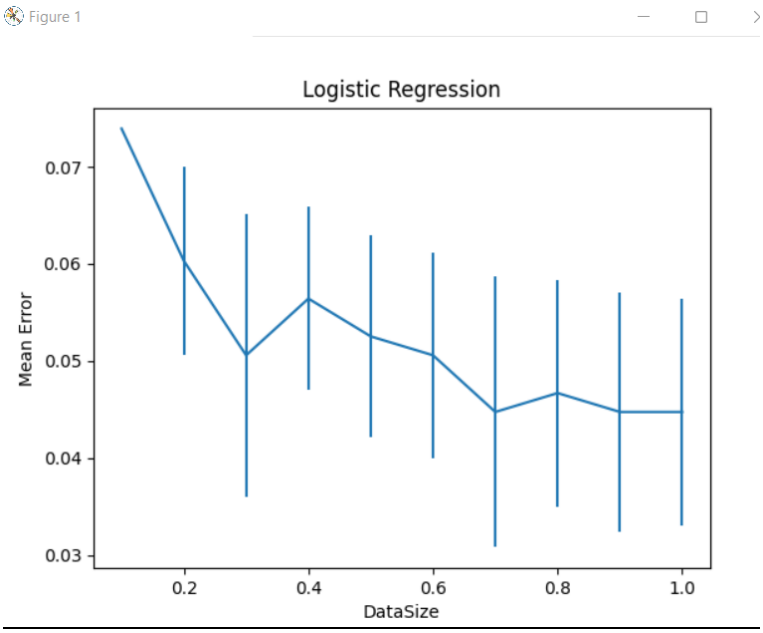
It can be seen from the plot that as the train dataset size increases, the overall error rate decreases. Each fragment of the dataset takes about 85 iterations before W converges and the avg runtime for the whole process is around 7.7 seconds.

The highest mean of the error goes upto 0.45 and the error goes to a low of around 0.28 at full fraction of the train Dataset.

P.S: I acknowledge the unusually high standard deviation but I couldn't resolve the issue as I ran out of the time. I acknowledge that the std should not be as high as it shows in the plots.

Dataset: USPS

Plot:



Runtime and Iterations:

```
"C:\Users\Abhishek Sharma\PycharmProjects\GLM\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/GLM/main.py"
Average No of Iterations100
Runtime: 16.506453275680542 seconds
```

Discussion of the result:

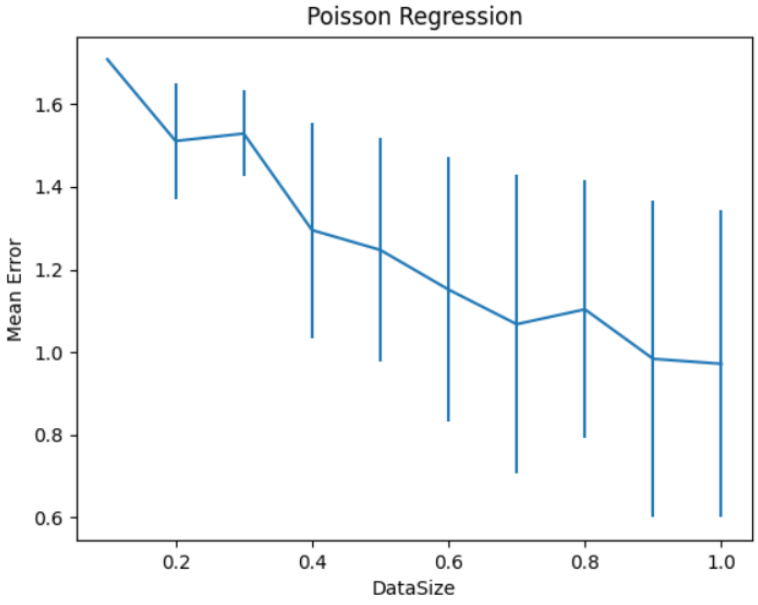
It can be seen from the plot that as the train dataset size increases, the overall error rate decreases. Each fragment of the dataset takes about 100 iterations before W converges and the avg runtime for the whole process is around 16.5 seconds.

The highest mean of the error goes up to 0.07 and the error goes to a low of around 0.05 at full fraction of the train Dataset.

P.S: I acknowledge the unusually high standard deviation but I couldn't resolve the issue as I ran out of the time. I acknowledge that the std should not be as high as it shows in the plots.

2. Poisson Regression

Plot:



Runtime and Iterations:

```
"C:\Users\Abhishek Sharma\PycharmProjects\GLM\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/GLM/main.py"
Runtime: 4.8341827392578125 seconds
Average No of Iterations 96
```

Discussion of the result:

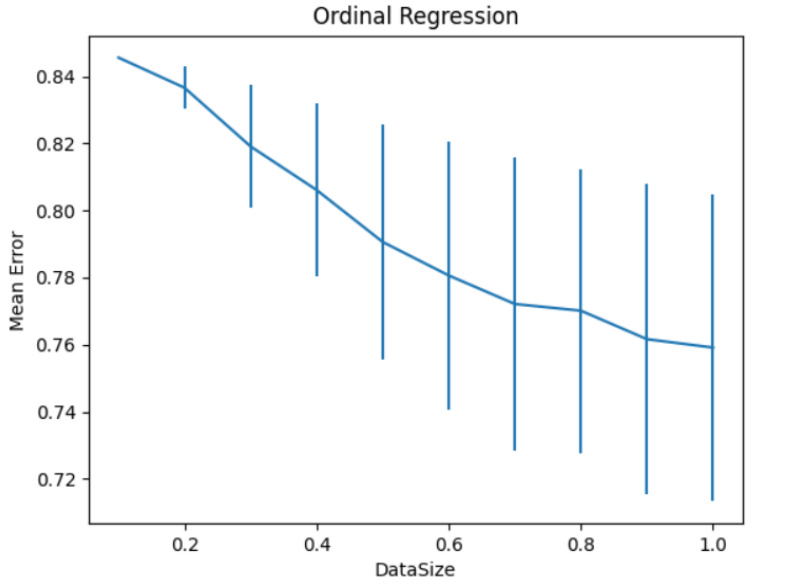
It can be seen from the plot that as the train dataset size increases, the overall error rate decreases. Each fragment of the dataset takes about 96 iterations before W converges and the avg runtime for the whole process is around 4.9 seconds.

The highest mean of the error goes up to 1.7 and the error goes to a low of around 1 at full fraction of the train Dataset.

P.S: I acknowledge the unusually high standard deviation but I couldn't resolve the issue as I ran out of the time. I acknowledge that the std should not be as high as it shows in the plots.

3. **Oridnal Regression:**

Plot:



Runtime and Iterations:

```
"C:\Users\Abhishek Sharma\PycharmProjects\GLM\venv\Scripts\python.exe" "C:/Users/Abhishek Sharma/PycharmProjects/GLM/main.py"
Runtime: 57.68153405189514 seconds
Average No of Iterations 99
```

Discussion of the result:

It can be seen from the plot that as the train dataset size increases, the overall error rate decreases. Each fragment of the dataset takes about 99 iterations before W converges and the avg runtime for the whole process is around 57.6 seconds.

The highest mean of the error goes up to 0.84 and the error goes to a low of around 0.76 at full fraction of the train Dataset.

P.S: I acknowledge the unusually high standard deviation but I couldn't resolve the issue as I ran out of the time. I acknowledge that the std should not be as high as it shows in the plots.

Q. Are the learning curves as expected?

The error rate is expected to go down as the learning process progresses further. Also, a bigger train data size is expected to enhance the learning which would result in lower error rates.
As is evident from the plots, the learning curves follow the expected behavior.

Q. How does learning time vary across datasets for classification?

The time taken by the USPS dataset of the Logistic Likelihood model was of about 16.7 seconds, whereas the same model takes only about 7 seconds on the A.csv dataset. This shows that the learning time depends on the size of the datasets.

The dataset used in the Poisson takes the least amount of time out of all the other datasets.

Q. How does learning time vary across the likelihood models?

The learning time varies widely across different likelihood models. For example, the shortest time was taken by the Poisson Likelihood model of about 4.9 seconds, whereas the highest time was taken by the Ordinal Likelihood Model, around 57.6 seconds.
The logistic Likelihood model takes around 7.6 seconds for the A.csv dataset and around 16.7 seconds for the USPS dataset.

Since the steps involved in the Ordinal Model are more complex than the other two models, the effect of this can be easily seen in the runtime of this model, which is the highest of all.

Q. What are the main costs affecting these (time per iteration, number of iterations)?

The main costs that affect the number of iterations can be though to be as follows:

1. **The complexity of the Model:**
   It can be observed that the more complexly designed model results in higher amount of time taken as compared to those models that are simpler in comparison.
   For eg: The Ordinal model involves some complex steps in creating the y matrix and in calculating the d and R matrices and these complex steps result in higher execution time.
2. **The learning labels provided:**
   The learning labels provided with the datasets can also affect the number of iterations and the time per iteration. This is because the labels are used in the calculation of the d and R matrices which affect the convergence of W.

ReadMe:

Info about the code:

1. **The code is divided into three sections: Part 1: Logistic Regression, Part 2: Poisson Regression and Part 3: Ordinal Regression.**
2. **In each part, the code divides the datasets into Train and Test Dataset and pass them as arguments along with other required values to the common functions created to perform the predictions.**
3. **Out of these 3 sections of code, one is left uncommented and the remaining three are commented.**

Steps to run the code:

1. **Change the path values in the path variables according to the new location where the code will be run. The variables are divided into the dataset and the respective label file names of the datasets.**

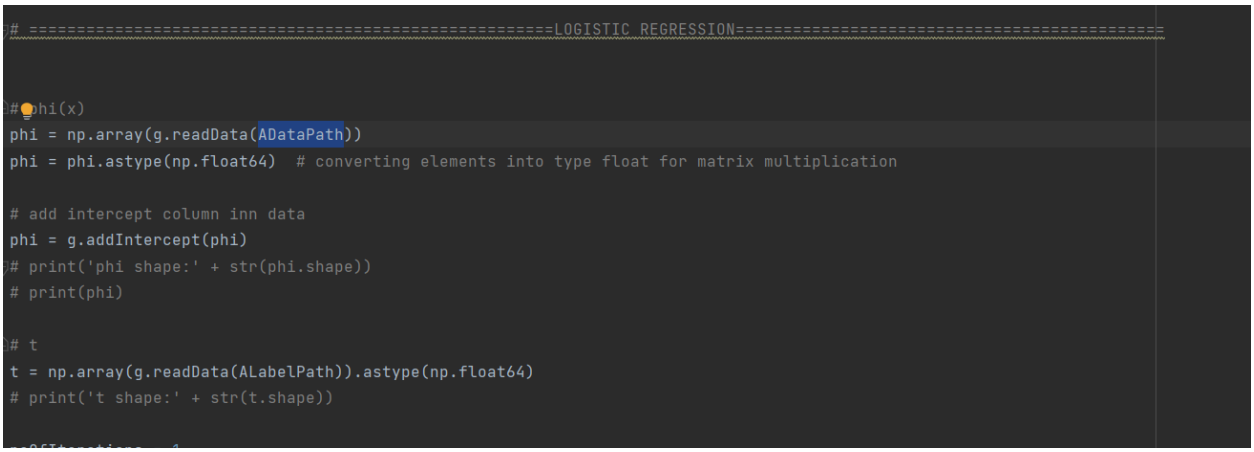
```
ADataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/A.csv"
APDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AP.csv"
AODataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AO.csv"
USPSDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/usps.csv"
ALabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-A.csv"
APLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-AP.csv"
AOLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-AO.csv"
USPSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-usps.csv"
IRLSDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/irlstest.csv"
IRLSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-irlstest.csv"
```

2. **Move to the part which you want to execute and uncomment the code for that particular model that you want to test the code on.**

```
        mean = sum(errorList[i]) / len(errorList[i])
        MeanErrorList_innerLoop.append(mean)
        variance = sum([((x - MeanErrorList_innerLoop[i]) ** 2) for x in MeanErrorList_innerLoop]) / len(
            MeanErrorList_innerLoop)
        std = variance ** 0.5
        # std = np.std(errorList[i])
        StdList.append(std)

end = time.time()
print('Average No of Iterations ' + str(noOfIterations * 85))
print("Runtime: " + str(end - start) + " seconds")

plt.xlabel("DataSize")
plt.ylabel("Mean Error")
plt.title("Logistic Regression")
plt.errorbar(train_data_size, MeanErrorList_innerLoop, StdList)
plt.show()

# ==================================================POISSON REGRESSION=============================================
#
# phi(x)
# phi = np.array(g.readData(APDataPath))
# phi = phi.astype(np.float64)  # converting elements into type float for matrix multiplication
#
# # add intercept column inn data
# phi = g.addIntercept(phi)
# # print('phi shape:' + str(phi.shape))
#
#
# # t
```

3. **For Logistic Model:**
   This model runs on two datasets: A.csv and USPS.csv

Initially, A.csv is being used by default. In order to use the USPS dataset, just replace the dataset path variable accordingly to USPS path variable.



```
# ========================================================LOGISTIC REGRESSION=============================================

#phi(x)
phi = np.array(g.readData(ADataPath))
phi = phi.astype(np.float64)  # converting elements into type float for matrix multiplication

# add intercept column inn data
phi = g.addIntercept(phi)
# print('phi shape:' + str(phi.shape))
# print(phi)

# t
t = np.array(g.readData(ALabelPath)).astype(np.float64)
# print('t shape:' + str(t.shape))
```

4. **Run the code.**
5. **The plots will be automatically shown at the end of the execution.**

CODE:

```
# import necessary libraries
import csv
import time

# import time
import numpy as np

# import statistics
#
import math

# import operator
import matplotlib.pyplot as plt

ADataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/A.csv"
APDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AP.csv"
AODataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/AO.csv"
USPSDataPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/usps.csv"
ALabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-
A.csv"
APLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-
AP.csv"
AOLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-
AO.csv"
USPSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-
usps.csv"
IRLSDataPath = "C:/Users/Abhishek
```

```
Sharma/Downloads/pp3data/pp3data/irlstest.csv"
IRLSLabelPath = "C:/Users/Abhishek Sharma/Downloads/pp3data/pp3data/labels-
irlstest.csv"


class GLM:
    # read data
    def readData(self, path):
        file = open(path)
        csvReader = csv.reader(file)
        data = []
        for row in csvReader:
            data.append(row)
        file.close()
        return data

    # add intercept column in the beginning of the matrix
    def addIntercept(self, data):
        newData = np.concatenate((np.ones((len(data), 1)), data), axis=1)
        return newData

    # calculate first derivative vector g
    def calculateG(self, phiT, d, alpha, w):
        g = np.array(phiT @ d - alpha * w)
        return g

    # calculate second derivative matrix H
    def calculateHInverse(self, phiT, phi, R, alpha):
        # I for H matrix
        I = np.eye(len(((-phiT @ R) @ phi)))
        H = -((phiT @ R) @ phi) - (alpha * I)
        HInverse = np.linalg.inv(H)
        return HInverse

    # sigmoid
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    # Logistic regression function to calculate d and R matrices
    def logisticReg(self, phi, t, w):
        # print('wT shape:' + str(wT.shape))

        wT_phi = phi @ w

        # print('wT_phi shape' + str(wT_phi.shape))

        y = np.array(self.sigmoid(wT_phi))
        # print('y shape' + str(y.shape))

        # d vector
        # print(y)
        d = np.array(t.reshape(-1, ) - y)
        # print('d shape' + str(d.shape))

        # R matrix
        R = y * (1 - y)
        # print('R shape before diagonal' + str(R.shape))
```

```
        R = np.diag(R)
        # print('R shape after diagonal' + str(R.shape))

        return [d, R]

    # poisson data function to calculate d and R matrices
    def poissonReg(self, phi, t, w):

        # print('t shape: '+str(t.shape))
        wT_phi = phi @ w
        # print('wT_phi shape' + str(wT_phi.shape))

        y = np.array(np.exp(wT_phi))
        # print('y shape' + str(y.shape))

        # d vector
        d = t.reshape(-1, ) - y
        # print('d shape' + str(d.shape))

        # R matrix
        R = y
        # print('R shape before diagonal' + str(R.shape))
        R = np.diag(R)
        # print('R shape after diagonal' + str(R.shape))

        return [d, R]

    # ordinal data function to calculate d and R matrices
    def ordinalReg(self, phi, t, w):
        s = 1
        k = 5
        a = phi @ w
        y = np.empty((len(t), k + 1))
        ordinals_phi = [-np.inf, -2, -1, 0, 1, np.inf]

        # creating y matrix
        for i in range(len(a)):
            for j in range(6):
                X = s * (ordinals_phi[j] - a[i])
                yij = self.sigmoid(X)
                y[i, j] = yij

        d = np.empty(len(t), )
        R = np.empty(len(t), )

        # filling up d and R
        for i in range(y.shape[0]):
            yi_j = y[i, int(t[i])]
            yi_jMinus1 = y[i, int(t[i]) - 1]
            d[i] = yi_j + yi_jMinus1 - 1
            R[i] = s ** 2 * ((yi_j * (1 - yi_j)) + (yi_jMinus1 * (1 -
yi_jMinus1)))

        # # print('R shape before diagonal' + str(R.shape))
        R = np.diag(R)
        # print('R shape after diagonal' + str(R.shape))
```

```
        return [d, R]

    # implement Newton-Raphson method and update weight vectors
    def MyGLM(self, phi, phiT, t, alpha, w_old, counter, distribution):

        if distribution == "Logistic":
            dRList = self.logisticReg(phi, t, w_old)
            d = dRList[0]
            R = dRList[1]

            g = self.calculateG(phiT, d, alpha, w_old)
            # print('g vector shape: ' + str(g.shape))
            HInverse = self.calculateHInverse(phiT, phi, R, alpha)
            # print('h vector shape: ' + str(HInverse.shape))
            w_new = w_old - (HInverse @ g)

            while (np.linalg.norm((w_new - w_old), 2)) /
((np.linalg.norm(w_old, 2) + 1e-10)) > 0.001 or counter <100:
                dRList = self.logisticReg(phi, t, w_old)
                d = dRList[0]
                R = dRList[1]
                g = self.calculateG(phiT, d, alpha, w_old)
                HInverse = self.calculateHInverse(phiT, phi, R, alpha)
                w_new = w_old - (HInverse @ g)
                w_old = w_new
                counter += 1
            return w_new



        elif distribution == "Poisson":

            dRList = self.poissonReg(phi, t, w_old)

            d = dRList[0]

            R = dRList[1]

            g = self.calculateG(phiT, d, alpha, w_old)

            # print('g vector shape: ' + str(g.shape))

            HInverse = self.calculateHInverse(phiT, phi, R, alpha)

            w_new = w_old - (HInverse @ g)

            while (np.linalg.norm((w_new - w_old), 2)) /
((np.linalg.norm(w_old, 2) + 1e-10)) > 0.001 or counter <100:
                dRList = self.poissonReg(phi, t, w_old)

                d = dRList[0]

                R = dRList[1]

                g = self.calculateG(phiT, d, alpha, w_old)

                HInverse = self.calculateHInverse(phiT, phi, R, alpha)
```

```
                w_new = w_old - (HInverse @ g)

                w_old = w_new

                counter += 1

            return w_new



        elif distribution == "Ordinal":
            dRList = self.ordinalReg(phi, t, w_old)
            d = dRList[0]
            R = dRList[1]

            g = self.calculateG(phiT, d, alpha, w_old)
            # print('g vector shape: ' + str(g.shape))
            HInverse = self.calculateHInverse(phiT, phi, R, alpha)
            w_new = w_old - (HInverse @ g)

            while (np.linalg.norm((w_new - w_old), 2)) /
((np.linalg.norm(w_old, 2) + 1e-10)) > 0.001 or counter <100:
                dRList = self.ordinalReg(phi, t, w_old)
                d = dRList[0]
                R = dRList[1]
                g = self.calculateG(phiT, d, alpha, w_old)
                HInverse = self.calculateHInverse(phiT, phi, R, alpha)
                w_new = w_old - (HInverse @ g)
                # print(w_new[1])
                w_old = w_new
                counter += 1
            return w_new

    def calculateError_Logistic(self, Wmap, phi, t):
        Wmap_phi = phi @ Wmap
        probOfT1 = np.array(self.sigmoid(Wmap_phi))
        # print(probOfT1)
        t_hat = []
        for i in range(len(probOfT1)):
            if probOfT1[i] >= 0.5:
                t_hat.append(1)
            else:
                t_hat.append(0)
        t_hat = np.array(t_hat).astype(int)
        t = t.astype(int)

        # print(t_hat)
        # print(t.reshape(-1,))

        error = []
        for i in range(len(t_hat)):
            if t_hat[i] == int(t[i]):
                error.append(0)
            else:
                error.append(1)
        # print(np.mean(error))
```

```
        return error

    def calculateError_Poisson(self, Wmap, phi, t):
        a = phi @ Wmap
        lambdaa = np.exp(a)

        t_hat = []
        for i in range(len(lambdaa)):
            t_hat.append(np.floor(lambdaa[i]))

        t_hat = np.array(t_hat)
        # print(t_hat)
        t = t.astype(int)

        error = []
        for i in range(len(t_hat)):
            error.append(int(abs(t_hat[i] - t[i])))

        return error

    def calculateError_Ordinal(self, Wmap, phi, t):
        a = phi @ Wmap

        s = 1
        k = 5
        y = np.empty((len(t), k + 1))
        p = np.empty((len(t), k))
        ordinals_phi = [-np.inf, -2, -1, 0, 1, np.inf]

        for i in range(len(a)):
            for j in range(6):
                X = s * (ordinals_phi[j] - a[i])
                yij = self.sigmoid(X)
                y[i, j] = yij

            for k in range(1, len(y[i])):
                p[i, k - 1] = y[i, k] - y[i, (k - 1)]
        p_max = p.argmax(axis=1)
        t_hat = []
        t_hat.append(p_max)
        # print(t_hat)
        t = t.astype(int)

        error = []
        for i in range(len(t_hat)):
            error.append(abs(t_hat[i] - t[i]))
        return error


g = GLM()

# alpha
alpha = 10

# counter
counter = 0
```

```
start = time.time()

# ======================================================LOGISTIC
REGRESSION=============================================


# phi(x)
phi = np.array(g.readData(ADataPath))
phi = phi.astype(np.float64)  # converting elements into type float for
matrix multiplication

# add intercept column inn data
phi = g.addIntercept(phi)
# print('phi shape:' + str(phi.shape))
# print(phi)

# t
t = np.array(g.readData(ALabelPath)).astype(np.float64)
# print('t shape:' + str(t.shape))

noOfIterations = 1

TotalMeanErrors_30 = []

for i in range(30):
    dummy = np.c_[t, phi]
    # print(dummy.shape)
    np.random.shuffle(dummy)
    x = dummy[:, 1:dummy.shape[1]]
    # print(x.shape)
    y = dummy[:, 0:1]
    # print(y.shape)

    train_phi = x[:int(2 * len(x) / 3)]
    train_t = y[:int(2 * len(y) / 3)]
    test_phi = x[int(2 * len(y) / 3):]
    test_t = y[int(2 * len(y) / 3):]
    # print(train_phi.shape)
    # print(train_t.shape)
    # print(test_phi.shape)
    # print(test_t.shape)

    train_data_size = np.arange(0.1, 1.1, 0.1)
    error = []
    errorList = []
    for s in train_data_size:
        segmented_Train_Phi = train_phi[:int(s * len(train_phi))]
        segmented_Train_t = train_t[:int(s * len(train_t))]
        segmented_Train_Phi_Transpose = segmented_Train_Phi.T
        w = np.zeros([len(segmented_Train_Phi_Transpose)])
        Wmap = g.MyGLM(segmented_Train_Phi, segmented_Train_Phi_Transpose,
segmented_Train_t, alpha, w, counter,
                       "Logistic")

        # calculation of error
        error = g.calculateError_Logistic(Wmap, test_phi, test_t)
```

```
        errorList.append(error)

    MeanErrorList_innerLoop = []
    StdList = []
    for i in range(len(errorList)):
        mean = sum(errorList[i]) / len(errorList[i])
        MeanErrorList_innerLoop.append(mean)
        variance = sum([((x - MeanErrorList_innerLoop[i]) ** 2) for x in
MeanErrorList_innerLoop]) / len(
            MeanErrorList_innerLoop)
        std = variance ** 0.5
        # std = np.std(errorList[i])
        StdList.append(std)

end = time.time()
print('Average No of Iterations ' + str(noOfIterations * 85))
print("Runtime: " + str(end - start) + " seconds")

plt.xlabel("DataSize")
plt.ylabel("Mean Error")
plt.title("Logistic Regression")
plt.errorbar(train_data_size, MeanErrorList_innerLoop, StdList)
plt.show()

# ==================================================POISSON
REGRESSION==============================================
#
# phi(x)
# phi = np.array(g.readData(APDataPath))
# phi = phi.astype(np.float64)  # converting elements into type float for
matrix multiplication
#
# # add intercept column inn data
# phi = g.addIntercept(phi)
# # print('phi shape:' + str(phi.shape))
#
#
# # t
# t = np.array(g.readData(APLabelPath)).astype(int).reshape(-1, )
# # print('t shape:' + str(t.shape))
# # print(t)
#
# for i in range(30):
#     dummy = np.c_[t, phi]
#     # print(dummy.shape)
#     np.random.shuffle(dummy)
#     x = dummy[:, 1:dummy.shape[1]]
#     # print(x.shape)
#     y = dummy[:, 0:1]
#     # print(y.shape)
#
#     train_phi = x[:int(2 * len(x) / 3)]
#     train_t = y[:int(2 * len(y) / 3)]
#     test_phi = x[int(2 * len(y) / 3):]
#     test_t = y[int(2 * len(y) / 3):]
#     # print(train_phi.shape)
#     # print(train_t.shape)
```

```
#     # print(test_phi.shape)
#     # print(test_t.shape)
#
#     train_data_size = np.arange(0.1, 1.1, 0.1)
#     # print(train_data_size)
#     errorList = []
#     for s in train_data_size:
#         segmented_Train_Phi = train_phi[:int(s * len(train_phi))]
#         segmented_Train_t = train_t[:int(s * len(train_t))]
#         segmented_Train_Phi_Transpose = segmented_Train_Phi.T
#         w = np.zeros([len(segmented_Train_Phi_Transpose)])
#         Wmap = g.MyGLM(segmented_Train_Phi, segmented_Train_Phi_Transpose,
segmented_Train_t, alpha, w, counter,
#                        "Poisson")
#
#         # calculation of error
#         error = g.calculateError_Poisson(Wmap, test_phi, test_t)
#         errorList.append(error)
#
#         MeanErrorList_innerLoop = []
#         StdList = []
#         for i in range(len(errorList)):
#             mean = sum(errorList[i]) / len(errorList[i])
#             MeanErrorList_innerLoop.append(mean)
#             variance = sum([((x - MeanErrorList_innerLoop[i]) ** 2) for x
in MeanErrorList_innerLoop]) / len(
#                 MeanErrorList_innerLoop)
#             res = variance ** 0.5
#             StdList.append(res)
#
# end = time.time()
# print("Runtime: " + str(end - start) + " seconds")
#
# plt.xlabel("DataSize")
# plt.ylabel("Mean Error")
# plt.title("Poisson Regression")
# plt.errorbar(train_data_size, MeanErrorList_innerLoop, StdList)
# plt.show()


# ================================================ORDINAL
REGRESSION================================================
#
# #phi(x)
# phi = np.array(g.readData(AODataPath))
# phi = phi.astype(np.float64)  # converting elements into type float for
matrix multiplication
#
# # add intercept column inn data
# phi = g.addIntercept(phi)
# # print('phi shape:' + str(phi.shape))
#
# phiT = phi.T
# w = np.zeros([len(phiT)])
# # print('w shape ' + str(w.shape))
#
# # t
```

```
# t = np.array(g.readData(AOLabelPath)).astype(int).reshape(-1, )
# # print('t shape:' + str(t.shape))
# # print(t)
#
# noOfIterations = 1
#
# for i in range(30):
#     dummy = np.c_[t, phi]
#     # print(dummy.shape)
#     np.random.shuffle(dummy)
#     x = dummy[:, 1:dummy.shape[1]]
#     # print(x.shape)
#     y = dummy[:, 0:1]
#     # print(y.shape)
#
#     train_phi = x[:int(2 * len(x) / 3)]
#     train_t = y[:int(2 * len(y) / 3)]
#     test_phi = x[int(2 * len(y) / 3):]
#     test_t = y[int(2 * len(y) / 3):]
#     # print(train_phi.shape)
#     # print(train_t.shape)
#     # print(test_phi.shape)
#     # print(test_t.shape)
#
#     train_data_size = np.arange(0.1, 1.1, 0.1)
#     # print(train_data_size)
#     errorList = []
#     for s in train_data_size:
#         segmented_Train_Phi = train_phi[:int(s * len(train_phi))]
#         segmented_Train_t = train_t[:int(s * len(train_t))]
#         segmented_Train_Phi_Transpose = segmented_Train_Phi.T
#         w = np.zeros([len(segmented_Train_Phi_Transpose)])
#         Wmap = g.MyGLM(segmented_Train_Phi, segmented_Train_Phi_Transpose,
segmented_Train_t, alpha, w, counter,
#                        "Ordinal")
#         # print(Wmap[i])
#         # calculation of error
#         error = g.calculateError_Poisson(Wmap, test_phi, test_t)
#         errorList.append(error)
#
#         MeanErrorList_innerLoop = []
#         StdList = []
#         for i in range(len(errorList)):
#             mean = sum(errorList[i]) / len(errorList[i])
#             MeanErrorList_innerLoop.append(mean)
#             variance = sum([((x - MeanErrorList_innerLoop[i]) ** 2) for x
in MeanErrorList_innerLoop]) / len(
#                 MeanErrorList_innerLoop)
#             res = variance ** 0.5
#             StdList.append(res)
#
# end = time.time()
# print("Runtime: " + str(end - start) + " seconds")
# print('Average No of Iterations ' + str(int(noOfIterations*99)))
# plt.xlabel("DataSize")
# plt.ylabel("Mean Error")
# plt.title("Ordinal Regression")
```

```
# plt.errorbar(train_data_size,MeanErrorList_innerLoop, StdList)
# plt.show()
```