1)# Question1.py

```python
# import random

import numpy as np


def logisticregr(theta, n, m):
    """Function to generate n*(m+1) independent variables, n*1 dependent 
variable, and coefficient matrix (m+1*1)

    Args:
        theta (float): spread of noise in the output variable
        n (int): size of the data set
        m (int): number of indepedent variables

    Returns:
        array: independent, dependent variables matrix and coefficient 
matrix
    """
    x_ones = np.ones((n, 1))  # first column of ones
    # np.random.seed(1998)
    x_random = np.random.randn(n, m)  # (n*m) matrix of random Real numbers
    X = np.concatenate((x_ones, x_random), axis=1)  # n rows and m+1 
columns where 1st column is of ones
    beta = np.random.rand(m + 1, 1)  # Coefficients
    Y = (
        1 / (1 + np.exp(-np.dot(X, beta))) >= 0.5
    )  # X*beta i.e. matrix multiplication of (n*m) and (m * 1) shape= Y 
shape (n*1)

    Y = Y.astype(int)
    noise = np.random.binomial(n=1, p=theta, size=(n, 1))
    Y = Y + noise
    Y %= 2

    ##### ------------------------------- ####
    # Alternatively, we can use Maximum likelihood estimation to add noise 
to Y

    # probflip = theta  # probability that the output would be flipped
```

```python
    # # As it is a bernoulli problem, by Maximum likelihood estimation, we
know (theta = # of flips/ n)
    # # hence, total number of flips =  n*theta)...taking this number of
random sets of inputs whose output is reversed

    # # flipping outputs of Y (adding noise)
    # for i in random.sample(range(0, n), int(n * probflip)):  # n unique
random intergers
    #     if Y[i] == 1:
    #         Y[i] = 0
    #     else:
    #         Y[i] = 1

    #### ---------------------------------- ######

    return X, Y, beta


if __name__ == "__main__":
    theta = 0.3
    n = 10
    m = 5
    X, Y_true, beta_true = logisticregr(theta, n, m)
    print("shape of X is ", X.shape)
    print("shape of beta is ", beta_true.shape)
    print("shape of Y is ", Y_true.shape)
    print("X matrix is \n", X)
    print("beta vector is \n", beta_true)
    print("Y dependent vector is \n ", Y_true)
```

2 and 4)# Question2 and Question 4.py

```python
import argparse  # Commandline input

import numpy as np
from numpy.linalg import norm
```

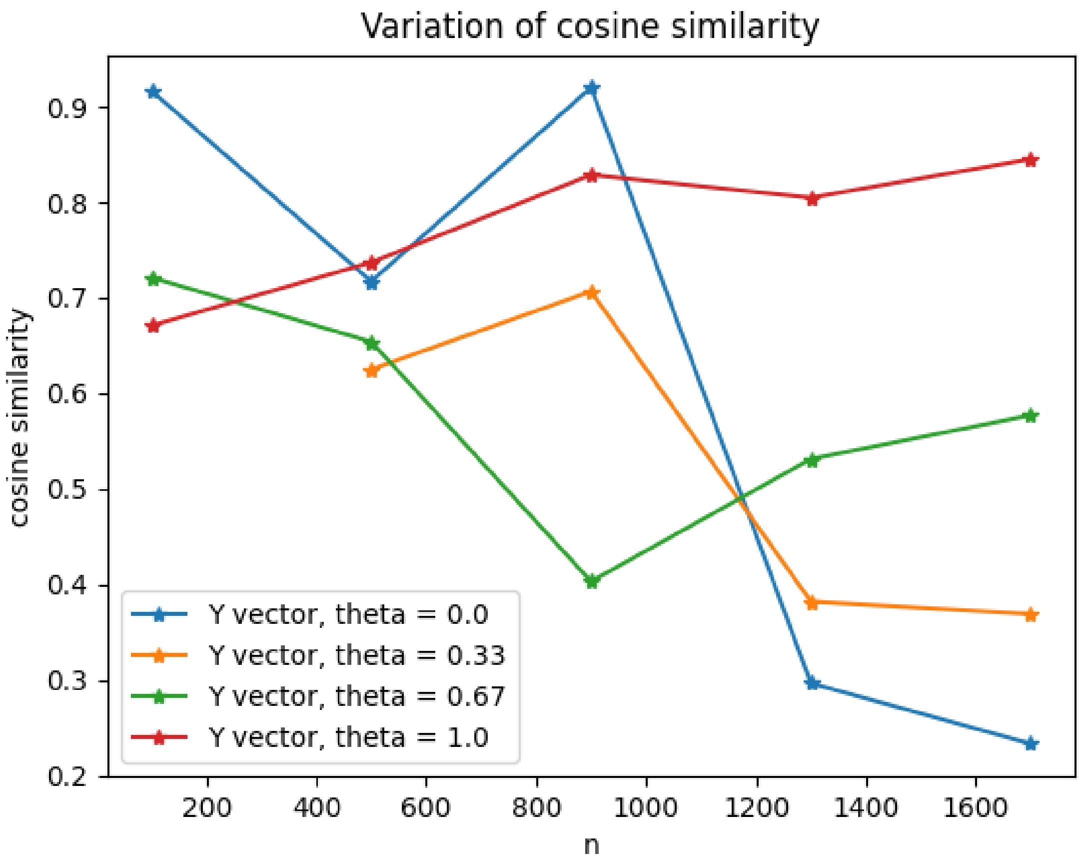
```python
import Q1 as q1  # importing First Question solution

parser = argparse.ArgumentParser()
parser.add_argument(
   "-regularization",
   type=str,
   default="0",
   help="Put l1 for lasso or l2 for Ridge or 0 for no regularization",
)
parser.add_argument(
   "-regconstant",
   type=float,
   help="Regularization constant (float)",
)
args = parser.parse_args()


def cost(Y_pred, Y_true, beta):
   """Calculate the cost

   Args:
       Y_pred (array): predicted value of Y output
       beta (array): value of beta array at current iteration including
initial initialization

   Returns:
       float: cost at current iteration
   """

   # remove 0 and 1 from Y_pred and convert it to close to 0 and 1
respectively
   Y_pred = np.where(Y_pred == 0, 0.01, Y_pred)
   Y_pred = np.where(Y_pred == 1, 0.99, Y_pred)

   # removing any 0 component in beta such that 0 doesn't come in
denomenator in regularization term
   beta = np.where(beta == 0, 0.01, beta)

   if args.regularization == "0":
```

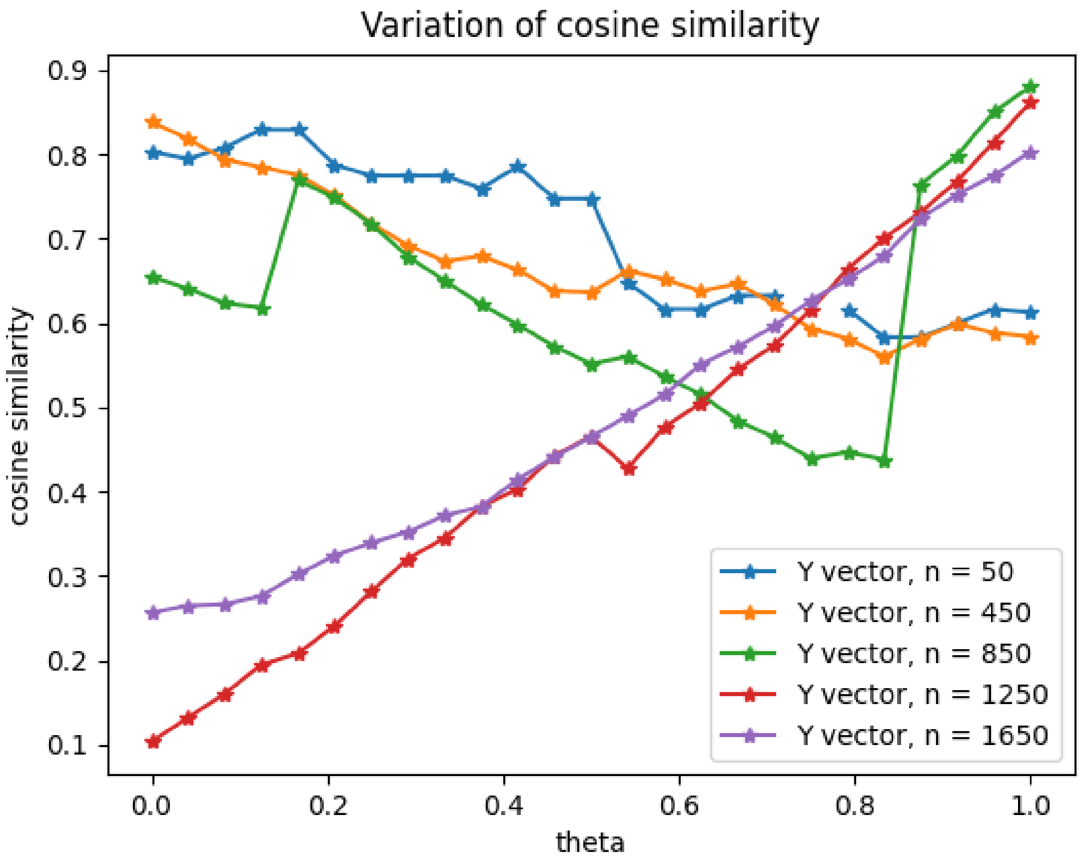
```python
        cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) * (np.log(1
- Y_pred)))  # initial cost function

    elif args.regularization == "l1":
        cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) * (np.log(1
- Y_pred))) + args.regconstant * np.sum(
            np.abs(beta)
        )  # initial cost function
    elif args.regularization == "l2":
        cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) * (np.log(1
- Y_pred))) + args.regconstant * np.sum(
            beta**2
        )

    return cost


def gradientdescent(X, Y_true, epochs, threshold, LR):
    np.random.seed(1998)
    beta = np.random.rand(X.shape[1], 1)  # initial values of beta
coefficients
    # print("Initial beta ", beta)

    Y_pred = 1 / (1 + np.exp(-np.dot(X, beta)))  # initial Y prediction

    init_cost = cost(Y_pred, Y_true, beta)  # initial cost function

    # print("init cost", init_cost)
    for j in range(epochs):
        if args.regularization == "0":
            beta[0] = beta[0] + LR * sum(Y_true - Y_pred)  # intercept

            for i in range(1, X.shape[1]):
                residualderivative = -(Y_true - Y_pred) * np.array(X[:,
i]).reshape(Y_true.shape[0], 1)
                beta[i] = beta[i] - LR * sum(residualderivative)

        if args.regularization == "l1":
            beta[0] = beta[0] + LR * sum(Y_true - Y_pred) -
args.regconstant * beta[0] / np.abs(beta[0])  # intercept
```

```
            for i in range(1, X.shape[1]):
                residualderivative = -(Y_true - Y_pred) * np.array(X[:,
i]).reshape(
                    Y_true.shape[0], 1
                ) + args.regconstant * beta[i] / np.abs(beta[i])
                beta[i] = beta[i] - LR * sum(residualderivative)

        if args.regularization == "l2":
            beta[0] = beta[0] + LR * sum(Y_true - Y_pred) -
args.regconstant * 2 * beta[0]  # intercept

            for i in range(1, X.shape[1]):
                residualderivative = (
                    -2 * (Y_true - Y_pred) * np.array(X[:,
i]).reshape(Y_true.shape[0], 1)
                    + args.regconstant * 2 * beta[i]
                )
                beta[i] = beta[i] - LR * sum(residualderivative)

        Y_pred = 1 / (1 + np.exp(-np.dot(X, beta)))  # Y prediction for the
new betas

        new_cost = cost(Y_pred, Y_true, beta)  # Cost function for the new
betas
        # print(new_cost)
        if init_cost - new_cost > threshold:  # condition
            init_cost = new_cost
            if j == epochs - 1:
                print(f"Loop ran till last epoch {j}")
        else:
            print(f"Loop break at {j}th iteration")
            break
    # cosinesim_beta = np.dot(beta_true.flatten(), beta.flatten()) /
(norm(beta_true.flatten()) * norm(beta.flatten()))
    Y_pred = Y_pred >= 0.5
    Y_pred = Y_pred.astype(int)

    cosinesim_Y = np.dot(Y_true.flatten(), Y_pred.flatten()) /
(norm(Y_true.flatten()) * norm(Y_pred.flatten()))
```

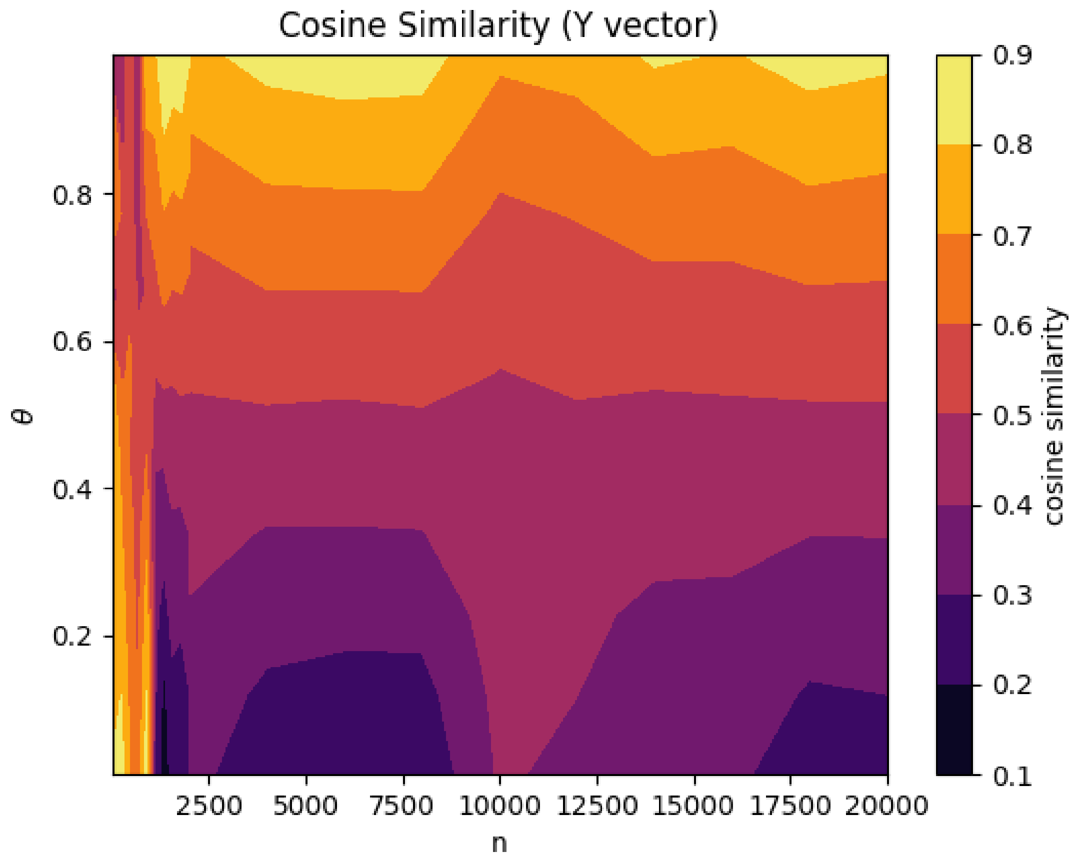
```python
    return beta, new_cost, j, cosinesim_Y


if __name__ == "__main__":

    theta = 0.8
    n = 1000
    m = 50
    X, Y_true, beta_true = q1.logisticregr(theta, n, m)
    print(f"True (Original) beta vector: {beta_true}")
    print(f"True (Original) output vector: {Y_true}")
    beta_new, new_cost, iteration, cosinesim_Y = gradientdescent(X, Y_true,
epochs=10000, threshold=0.00001, LR=0.001)
    print(f"Obtained beta vector after gradient descent: {beta_new}")
    print(f"cost function value is {new_cost}")
    # print(f"Cosine similarity between True beta vector and beta vector
after gradient descent is {cosinesim_beta}")
    print(f"Cosine similarity between True Y vector and Y vector after
gradient descent is {cosinesim_Y}")
```

Variation of cosine similarity between Y_true and Y_pred considering L1 regularization:

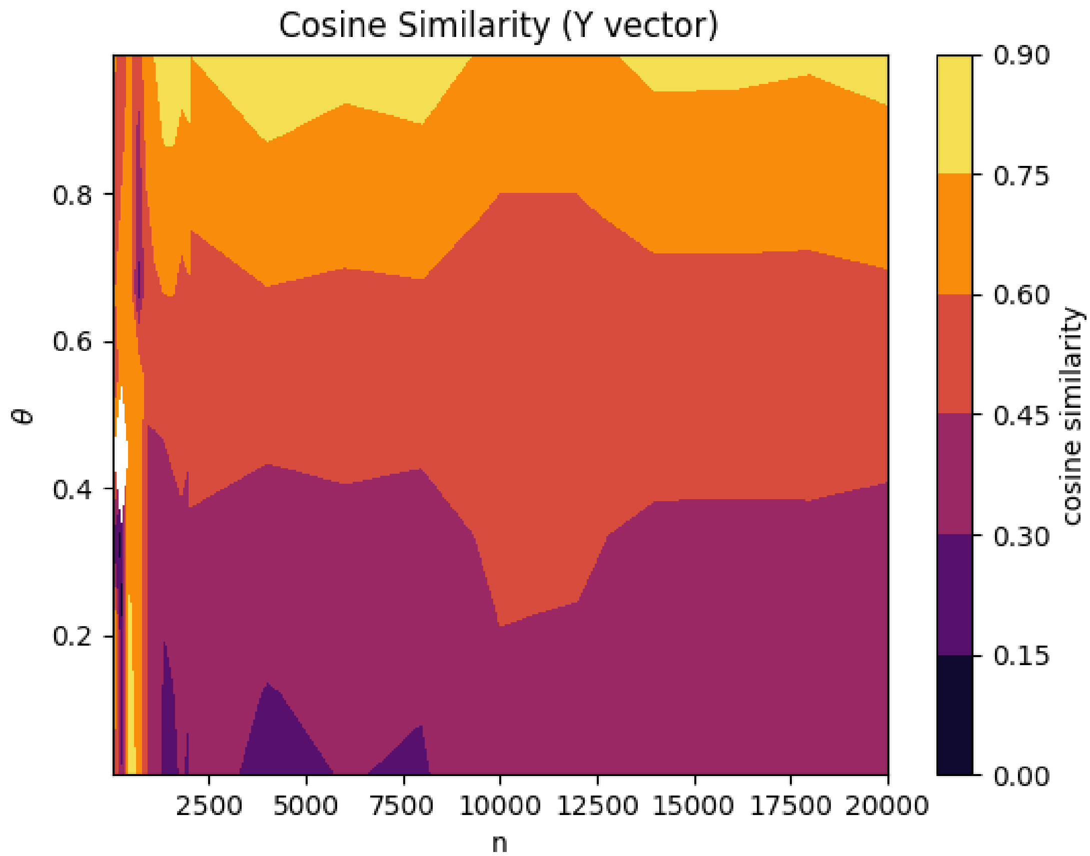




Variation of cosine similarity
cosine similarity
theta
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.0
0.2
0.4
0.6
0.8
1.0
Y vector, n = 50
Y vector, n = 450
Y vector, n = 850
Y vector, n = 1250
Y vector, n = 1650




Variation of Cosine similarity using L2 regularization:




3)

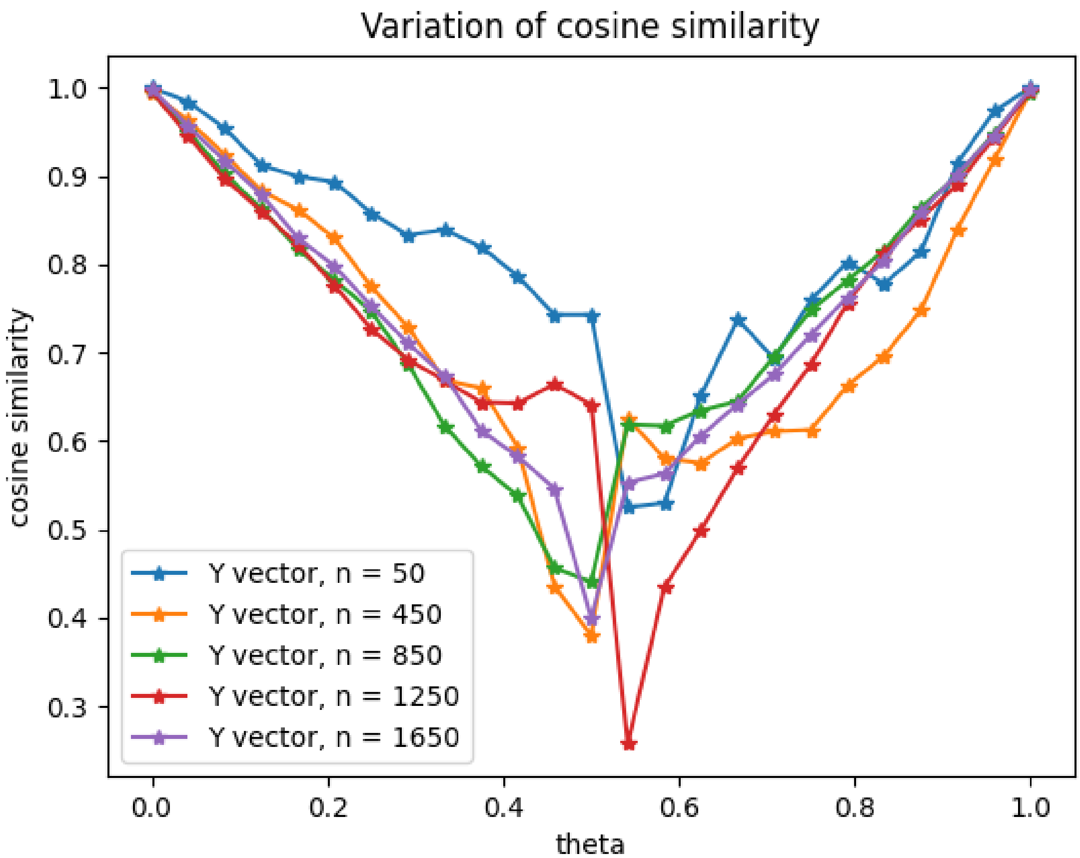
```
# Question3.py
import matplotlib.pyplot as plt
import numpy as np

import Q1 as q1   # importing First Question solution
import Q2_Q4 as q2

# ---------Caveat----------------    #

# 1) I have divided the script into three parts, please run one by one and comment others which is not needed for the moment
# 2) section 1 and section 2 took 5-10 mins to run when theta but this may vary depending on n and theta values you put
# 3) section 3 will take the most amount of time, on my system it took around ~25 mins again depending on the parameters you put
```

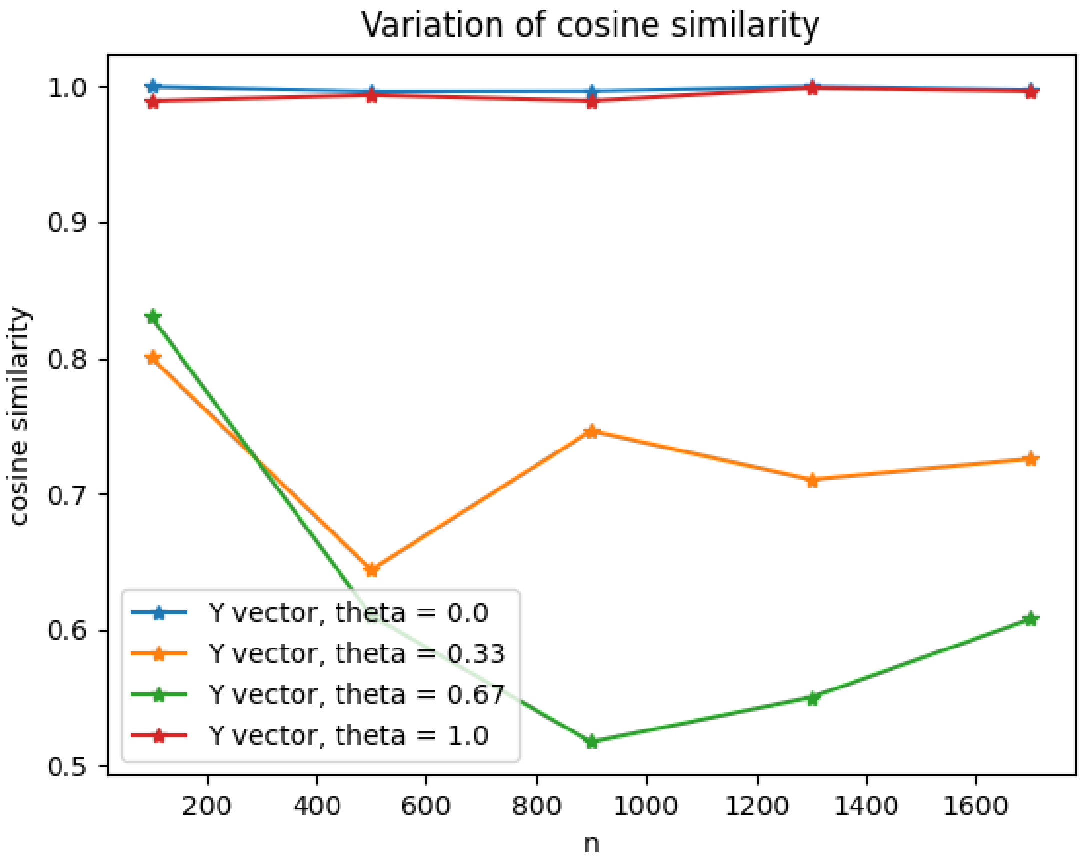
```
# -------------------------------   #


if __name__ == "__main__":

    # --------------------------------------------------#

    #                   VARIATION WITH theta

    # ---------------------------------------------------#

    # m = 5
    # iter = []  # list to store at which epoch , we break the loop to get
final new beta
    # costf = []  # final cost functions list

    # for n in range(50, 2000, 400):
    #     betacos_similarity = []
    #     Ycos_similarity = []
    #     for theta in np.linspace(0, 1, 25):  # Variation with theta,
0<=theta<=1
    #         print(f"For n {n} and theta {theta}")
    #         X, Y_true, beta_true = q1.logisticregr(theta, n, m)
    #         beta_new, new_cost, iteration, cosinesim_Y =
q2.gradientdescent(X, Y_true, 10000, 0.000001, 0.001)
    #         Ycos_similarity.append(cosinesim_Y)

    #     plt.plot(np.linspace(0, 1, 25), Ycos_similarity, marker="*",
label="Y vector, n = " + str(n))
    # plt.ylabel("cosine similarity")
    # plt.xlabel("theta")
    # plt.title("Variation of cosine similarity")
    # plt.legend()
    # plt.show()

    # --------------------------------------------------#

    #                   VARIATION WITH N

    # ---------------------------------------------------#
```

```
    # m = 5

    # for theta in np.linspace(0, 1, 4):
    #     betacos_similarity = []
    #     Ycos_similarity = []
    #     for n in range(100, 2000, 400):  # Variation with n
    #         print(f"For theta {theta} and n {n}")
    #         X, Y_true, beta_true = q1.logisticregr(theta, n, m)
    #         beta_new, new_cost, iteration, cosinesim_Y =
q2.gradientdescent(X, Y_true, 10000, 0.000001, 0.001)
    #         # betacos_similarity.append(cosinesim_beta)
    #         Ycos_similarity.append(cosinesim_Y)

    #     # plt.plot(range(10, 2000, 100), betacos_similarity, marker="*",
label="Beta vector, theta = " + str(theta))
    #     plt.plot(range(100, 2000, 400), Ycos_similarity, marker="*",
label="Y vector, theta = " + str(round(theta, 2)))
    # plt.xlabel("n")
    # plt.ylabel("cosine similarity")
    # plt.title("Variation of cosine similarity")
    # plt.legend()
    # plt.show()

    # --------------------------------------------------#

    #         VARIATION WITH BOTH theta and N

    # --------------------------------------------------#

    ninputs = np.concatenate((np.linspace(10, 1999, 10, dtype=int),
np.linspace(2000, 20000, 10, dtype=int))) # non-linear spacing
    theta = np.linspace(0.01, 0.99, 10)
    cosinesim_Y = np.zeros((10, 20))
    cosinesim_beta = np.zeros((10, 20))
    m = 10
    for t in theta:
        for n in ninputs:
            print(f"For n {n} and theta {t}")
            X, Y_true, beta_true = q1.logisticregr(t, n, m)
            (
```

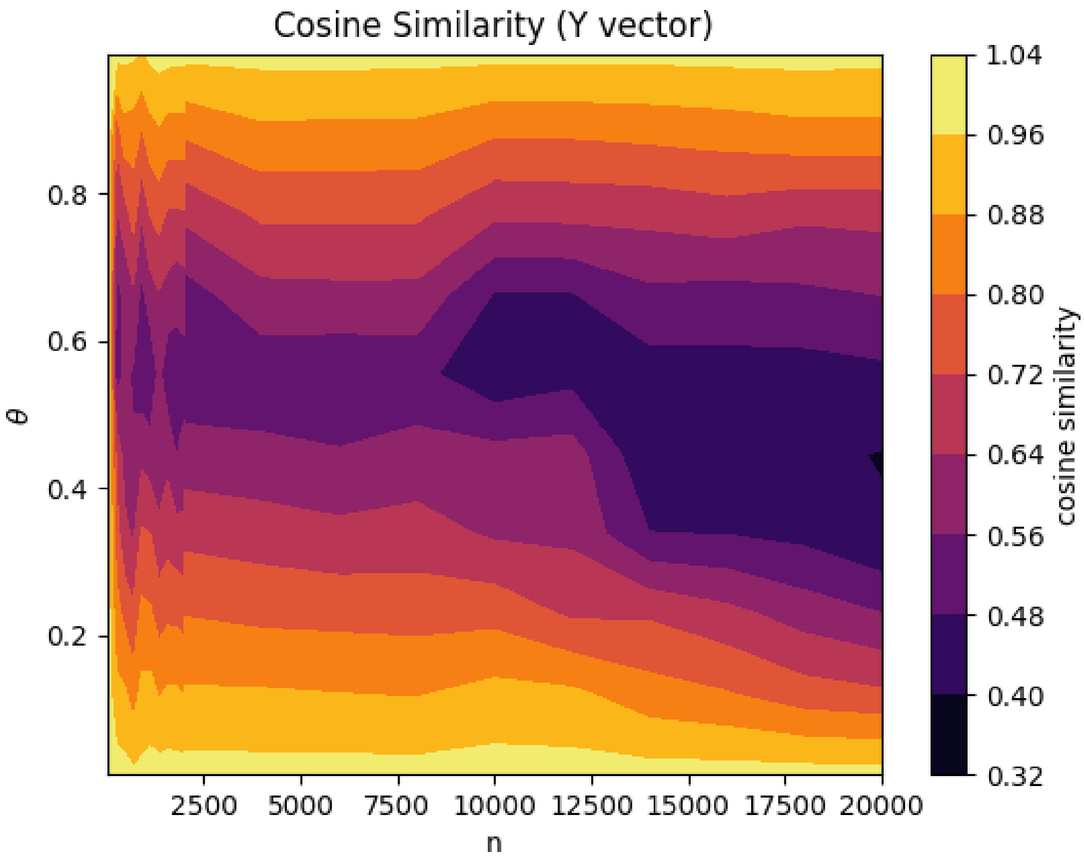
```
                beta_new,
                new_cost,
                iteration,
                cosinesim_Y[np.where(theta == t)[0], np.where(ninputs ==
n)[0]],
            ) = q2.gradientdescent(X, Y_true, 10000, 0.000001, 0.001)

   plt.title("Cosine Similarity (Y vector)")
   plt.contourf(ninputs, theta, cosinesim_Y, cmap="inferno")
   c = plt.colorbar()
   c.set_label("cosine similarity")
   plt.xlabel("n")
   plt.ylabel(r"$\theta$")
   plt.show()
```

Cosine similarity between Y_true and Y_prediction:





Variation of cosine similarity
cosine similarity
n
1.0
0.9
0.8
0.7
0.6
0.5
200
400
600
800
1000
1200
1400
1600
Y vector, theta = 0.0
Y vector, theta = 0.33
Y vector, theta = 0.67
Y vector, theta = 1.0




5)

```
# Question5.py
import argparse  # Commandline input

import numpy as np
from numpy.linalg import norm

parser = argparse.ArgumentParser()
parser.add_argument(
    "-regression",
    type=str,
    default="linear",
```

```
    help="Put linear for linear regression and logisitic for logisitic
regression",
)
parser.add_argument(
    "-regularization",
    type=str,
    default="0",
    help="Put l1 for lasso or l2 for Ridge or 0 for no regularization",
)
parser.add_argument(
    "-regconstant",
    type=float,
    help="Regularization constant (float)",
)
args = parser.parse_args()


class linear_logistic_reg:
    def __init__(self, std_dev, theta, n, m, epochs, threshold, LR):
        self.std_dev = std_dev
        self.theta = theta
        self.n = n
        self.m = m
        self.epochs = epochs
        self.threshold = threshold
        self.LR = LR

    def Regression(self):
        """Function to generate m*n independent variables, n*1 dependent
variable, and coefficient matrix

        Args:
            stddev (float): spread of noise in the output variable (linear
regression)
            theta (float): spread of noise in the output variable
(logisitic regression)
            n (int): size of the data set
            m (int): number of indepedent variables

        Returns:
```

```
            array: independent, dependent variables matrix and coefficient
matrix
        """
        x_ones = np.ones((self.n, 1))  # first column of ones
        # np.random.seed(1998)
        x_random = np.random.randn(self.n, self.m)  # (n*m) matrix of
random Real numbers
        X = np.concatenate((x_ones, x_random), axis=1)  # n rows and m+1
columns where 1st column is of ones
        beta = np.random.rand(self.m + 1, 1)  # Coefficients

        if args.regression == "linear":
            error = np.random.normal(loc=0, scale=self.std_dev,
size=(self.n, 1))
            Y = (
                np.matmul(X, beta) + error
            )  # X*beta i.e. matrix multiplication of (n*m) and (m * 1)
shape= Y shape (n*1)

        if args.regression == "logistic":
            Y = (
                1 / (1 + np.exp(-np.dot(X, beta))) >= 0.5
            )  # X*beta i.e. matrix multiplication of (n*m) and (m * 1)
shape= Y shape (n*1)

            Y = Y.astype(int)
            noise = np.random.binomial(n=1, p=self.theta, size=(self.n, 1))
            Y = Y + noise
            Y %= 2

        return X, Y, beta


    def cost(self, Y_pred, Y_true, beta):
        """Calculate the cost

        Args:
            Y_pred (array): predicted value of Y output
            beta (array): value of beta array at current iteration
including initial initialization
```

```
        Returns:
            float: cost at current iteration
        """

        # remove 0 and 1 from Y_pred and convert it to close to 0 and 1
respectively
        Y_pred = np.where(Y_pred == 0, 1e-4, Y_pred)
        Y_pred = np.where(Y_pred == 1, 1 - 1e-4, Y_pred)

        # removing any 0 component in beta such that 0 doesn't come in
denomenator in regularization term
        beta = np.where(beta == 0, 0.01, beta)

        if args.regularization == "0":
            cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) *
(np.log(1 - Y_pred)))  # initial cost function
        elif args.regularization == "l1":
            cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) *
(np.log(1 - Y_pred))) + args.regconstant * np.sum(
                np.abs(beta)
            )  # initial cost function
        elif args.regularization == "l2":
            cost = -np.mean(Y_true * np.log(Y_pred) + (1 - Y_true) *
(np.log(1 - Y_pred))) + args.regconstant * np.sum(
                beta**2
            )

        return cost

    def gradientdescent(self, X, Y_true):
        np.random.seed(1998)
        beta = np.random.rand(X.shape[1], 1)  # initial values of beta
coefficients
        print("Initial beta ", beta)

        if args.regression == "linear":
            Ypredict = np.matmul(X, beta)  # initial Y prediction
            init_cost = sum((Y_true - Ypredict) ** 2)  # initial cost
function
```

```
            print("init cost", init_cost)
            for j in range(self.epochs):
                beta[0] = beta[0] + 2 * self.LR * sum(Y_true - np.matmul(X,
beta))  # intercept
                for i in range(1, X.shape[1]):
                    residualderivative = (
                        -2 * (Y_true - np.matmul(X, beta)) * np.array(X[:,
i]).reshape(Y_true.shape[0], 1)
                    )
                    beta[i] = beta[i] - self.LR * sum(residualderivative)

                Ypred = np.matmul(X, beta)  # Y prediction or the new betas
                new_cost = sum((Y_true - Ypred) ** 2)  # Cost function for
the new betas
                print(new_cost)
                if init_cost - new_cost > self.threshold:  # condition
                    init_cost = new_cost
                else:
                    print(f"Loop break at {j}th iteration")
                    break
            cosinesim_Y = np.dot(Y_true.flatten(), Ypred.flatten()) /
(norm(Y_true.flatten()) * norm(Ypred.flatten()))

            return beta, new_cost, j, cosinesim_Y

        if args.regression == "logistic":

            Y_pred = 1 / (1 + np.exp(-np.dot(X, beta)))  # initial Y
prediction

            init_cost = test.cost(Y_pred, Y_true, beta)  # initial cost
function

            # print("init cost", init_cost)
            for j in range(self.epochs):
                if args.regularization == "0":
                    beta[0] = beta[0] + self.LR * sum(Y_true - Y_pred)  #
intercept
```

```
                   for i in range(1, X.shape[1]):
                       residualderivative = -(Y_true - Y_pred) *
np.array(X[:, i]).reshape(Y_true.shape[0], 1)
                       beta[i] = beta[i] - self.LR *
sum(residualderivative)

               if args.regularization == "l1":
                   beta[0] = (
                       beta[0] + self.LR * sum(Y_true - Y_pred) -
args.regconstant * beta[0] / np.abs(beta[0])
                   )  # intercept

                   for i in range(1, X.shape[1]):
                       residualderivative = -(Y_true - Y_pred) *
np.array(X[:, i]).reshape(
                           Y_true.shape[0], 1
                       ) + args.regconstant * beta[i] / np.abs(beta[i])
                       beta[i] = beta[i] - self.LR *
sum(residualderivative)

               if args.regularization == "l2":
                   beta[0] = beta[0] + self.LR * sum(Y_true - Y_pred) -
args.regconstant * 2 * beta[0]  # intercept

                   for i in range(1, X.shape[1]):
                       residualderivative = (
                           -2 * (Y_true - Y_pred) * np.array(X[:,
i]).reshape(Y_true.shape[0], 1)
                           + args.regconstant * 2 * beta[i]
                       )
                       beta[i] = beta[i] - self.LR *
sum(residualderivative)

               Y_pred = 1 / (1 + np.exp(-np.dot(X, beta)))  # Y prediction
for the new betas

               new_cost = test.cost(Y_pred, Y_true, beta)  # Cost function
for the new betas
               # print(new_cost)
               if init_cost - new_cost > self.threshold:  # condition
```

```
                init_cost = new_cost
                if j == self.epochs - 1:
                    print(f"Loop ran till last epoch {j}")
            else:
                print(f"Loop break at {j}th iteration")
                break
        # cosinesim_beta = np.dot(beta_true.flatten(), beta.flatten())
/ (norm(beta_true.flatten()) * norm(beta.flatten()))
        Y_pred = Y_pred >= 0.5
        Y_pred = Y_pred.astype(int)

        cosinesim_Y = np.dot(Y_true.flatten(), Y_pred.flatten()) /
(norm(Y_true.flatten()) * norm(Y_pred.flatten()))
        return beta, new_cost, j, cosinesim_Y


if __name__ == "__main__":

    test = linear_logistic_reg(std_dev=0.3, theta=0.8, n=1000, m=50,
epochs=10000, threshold=0.00001, LR=0.001)
    X, Y_true, beta_true = test.Regression()
    print(f"True (Original) beta vector: {beta_true}")
    print(f"True (Original) output vector: {Y_true}")
    beta_new, new_cost, iteration, cosinesim_Y = test.gradientdescent(
        X, Y_true
    )
    print(f"Obtained beta vector after gradient descent: {beta_new}")
    print(f"cost function value is {new_cost}")
    print(f"Cosine similarity between True Y vector and Y vector after
gradient descent is {cosinesim_Y}")
```

③ Loss function $(L) = -\left[y \log \hat{y} + (1-y)\log(1-\hat{y})\right]$

$y \to$ data

$\hat{y} = \sigma(z) = \dfrac{1}{1+e^{-z}}$ where $z = \vec{x}\cdot\vec{\beta} = \sum_i \beta_i x_i$

$$\frac{\partial L}{\partial \beta_i} = \frac{\partial L}{\partial \hat{y}}\,\frac{\partial \hat{y}}{\partial z}\,\frac{\partial z}{\partial \beta_i}$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial -\left(y \log \hat{y} + (1-y)\log(1-\hat{y})\right)}{\partial \hat{y}}$$

$$= -\left[\frac{y}{\hat{y}} + \frac{(1-y)}{1-\hat{y}}(-1)\right]$$

$$= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial\, 1/(1+e^{-z})}{\partial z} = -(1+e^{-z})^{-2}\,\frac{\partial}{\partial z}(1+e^{-z})$$

$$= -(1+e^{-z})^{-2}\, e^{-z}\,\frac{\partial(-z)}{\partial z}$$

$$= (1+e^{-z})^{-2}\, e^{-z} = \frac{1}{1+e^{-z}}\left(\frac{e^{-z}}{1+e^{-z}}\right)$$

$$= \hat{y}(1-\hat{y})$$

$$\therefore \frac{\partial L}{\partial \beta_i} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\left(\hat{y}(1-\hat{y})\right)\frac{\partial z}{\partial \beta_i}$$

As $x_0 = 1$ $\therefore \frac{\partial z}{\partial \beta_0} = \frac{\partial \beta_0}{\partial \beta_0} = 1$

$\forall$ other $x_n$, where $n \neq 0$

$$\frac{\partial z}{\partial \beta_i} = x_i$$

$$\therefore \frac{\partial L}{\partial \beta_i} = \begin{cases} (\hat{y} - y) & \text{where } i = 0 \\ (\hat{y} - y)x_i; & \forall\ i \neq 0 \end{cases}$$

(ii) Cosine Similarity ( $\frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|} = \cos\theta$; where $\theta$ is the Angle B/w 2 vectors), is minima at $\theta = 0.5$

here $\vec{A} = Y_{true}$

$\vec{B} = Y_{predicted}$