1. a) equation for θ between vectors: $\theta = \cos^{-1}\left(\frac{x^{(1)} \cdot x^{(2)}}{\|x^{(1)}\| \|x^{(2)}\|}\right)$

$x^{(1)} = \begin{bmatrix} \sqrt{2} \\ -\sqrt{2} \end{bmatrix} \rightarrow \|x^{(1)}\| = \sqrt{(\sqrt{2})^2 + (-\sqrt{2})^2} = \sqrt{2+2} = 2$

$x^{(2)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \|x^{(2)}\| = \sqrt{0^2 + 1^2} = \sqrt{1} = 1$

$\theta = \cos^{-1}\left(\frac{\sqrt{2}\cdot 0 + (-\sqrt{2})(1)}{2 \cdot 1}\right) = \cos^{-1}\left(\frac{-\sqrt{2}}{2}\right) = 135^\circ$

b) $V = \{\theta^T x = 0 : x \in \mathbb{R}^d, \theta \neq \bar{0}\}$

- any vector orthogonal to V is parallel to θ, because $\theta^T x = 0$ and 0 implies orthogonality
- So, our vectors, V, ...
  - $\hat{n} = \frac{\theta}{\|\theta\|}$ - - - - - -> $V = \pm \frac{\theta}{\|\theta\|} \cdot C$

c) $\bar{x} \in \mathbb{R}^d \quad \bar{\theta} = [\theta_1, \dots, \theta_d]^T$

i. $\hat{n} = \frac{\theta}{\|\theta\|}$ is our normal unit vector

so, $d(x) = \hat{n}^T x = \frac{\theta^T x}{\|\theta\|}$

ii. with offset $\bar{\theta} \cdot \bar{x} + b = 0 \rightarrow \theta^T x + b = 0$

$d(x) = \hat{n}^T x + b = \frac{\theta^T x + b}{\|\theta\|}$

iii. p: $-x_1 + \sqrt{3} x_2 + 5 = 0$

so, $\theta = [-1, \sqrt{3}]^T$, $b = 5$, $\|\theta\| = \sqrt{(-1)^2 + (\sqrt{3})^2} = \sqrt{4} = 2$

iii.i point $\bar{x} = [2,3]^T$

$d(x) = \frac{\theta^T x + b}{\|\theta\|} = \frac{(-1)(2) + (\sqrt{3})(3) + 5}{2} = \frac{-2 + 3\sqrt{3} + 5}{2} = \frac{3\sqrt{3} + 3}{2}$

iii.i point @ origin: $\bar{x} = [0,0]^T$

$d(x) = \frac{(0)(-1) + (0)(\sqrt{3}) + 5}{2} = \frac{5}{2}$

2.

a) $\{0,1\}^3$ because each word is either 0 or 1 if absent/present

b) $\{1,-1\}$ because output or label is either -1 for neg, +1 for pos

c) from our three "positive" reviews, we have...

$[0,1,0]$ $\theta_2 > 0$

$[1,0,0]$ $\theta_1 > 0$

$[0,0,1]$ $\theta_3 > 0$

but our "negative" review, we have...

$[1,1,0]$ $\theta_1 + \theta_2 < 0$

so, its <u>not possible</u> because if θ_1 and θ_2 are >0 and $\theta_1 + \theta_2$ needs to be <0, the data wouldn't be possible to learn.

d) lets update our bag of words vectors for each review in order...

(-1) $[1,1,1]$ $-2 + -2 + 3 = -1 < 0$

(+1) $[0,1,1]$ $0 + (-2) + 3 = 1 > 0$

(+1) $[1,0,1]$ $(-2) + 0 + 3 = 1 > 0$

(+1) $[0,0,1]$ $0 + 0 + 3 = 3 > 0$

if we make θ_1 and θ_2 worth -2, and make $\theta_3 = 3$, then...

So, learning $\overline{\theta}$ with no explicit offset is <u>possible</u> (yes)

3.

a) $\overline{x} = \{x_1, x_2, \cdots, x_6\}^T$ $\quad \overline{z} = \{z_1, z_2\}^T$

$\overline{z} = A\overline{x} = [z_1, z_2]^T$

- average: all 6 x's $\div 6$ $\quad [1/6, 1/6, \ldots, 1/6]$
- avg 1st 3 - avg last 3: first 3 $\div 3$, last 3 $\div(-3)$ $\quad [1/3, 1/3, 1/3, -1/3, -1/3, -1/3]$

So, $A = \begin{bmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 1/3 & 1/3 & 1/3 & -1/3 & -1/3 & -1/3 \end{bmatrix}$

b) $\overline{\theta}_x^T \overline{x} = \overline{\theta}_z^T (A\overline{x})$

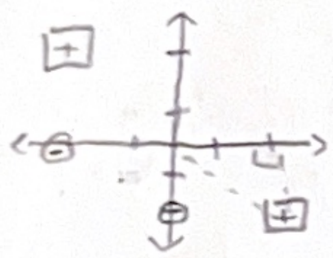
applying the identity

$(u^T A)\overline{x} = (A^T u)^T \overline{x}$ $\quad \rightarrow \quad \overline{\theta}_z^T (A\overline{x}) = (A^T \overline{\theta}_z)^T \overline{x}$

$\hookrightarrow \overline{\theta}_x^T \cancel{x} = (A^T \overline{\theta}_z)^T \cancel{x}$

$$\boxed{\overline{\theta}_x = A^T \overline{\theta}_z}$$

4.



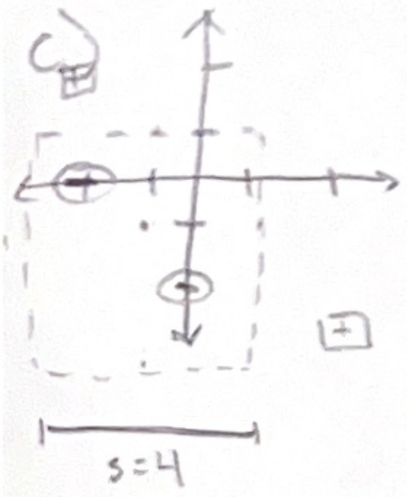
a) both negative points are a distance of 2 from (0,0)
both positive points are a distance of $2\sqrt{2}$ from (0,0)

- so, yes if $r = 2.2$, only points inside the circle will be negative and perfectly classified.

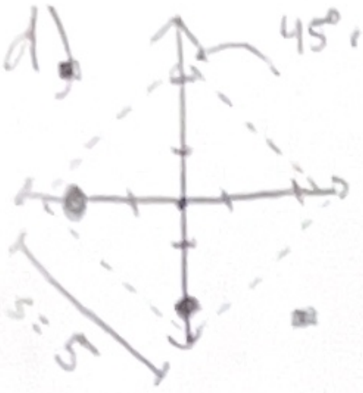
b) if prediction = $ax + by$, it is impossible to classify (-2,2) and (+2,-2) as both positive. $-2a + 2b = 2a - 2b$

$P_1 = -(P_2)$, because there is no (a,b) that will classify the data, it is impossible.

c)



a square centered at (-1,-1) with $s = 4$ will perfectly label the training set with negatives inside the square only.

d) 

a square centered at the origin with $s = 5$ and $\beta = 45°$ will <u>perfectly</u> classify the training set with points inside the square being negative.

5.

a) starting with $\bar{\theta} = \bar{0}$ and $b = 0$ and N incorrectly classified points

$$\bar{\theta}^{(k+1)} = \bar{\theta}^{(k)} + y^{(i)}\bar{x}^{(i)} \qquad b^{(k+1)} = b^{(k)} + y^{(i)}$$

↳ starting with $\bar{0}$ and 0, we can use summations

↳ also add in # times, a_i $\quad \bar{\theta} = \sum_{i=1}^{N} a_i y^{(i)} \bar{x}^{(i)}$ and $b = \sum_{i=1}^{N} a_i y^{(i)}$

so, $\boxed{\sum_{i=0}^{N} a_i y^{(i)} \bar{x}^{(i)} + \sum_{i=0}^{N} a_i y^{(i)} = 0}$ is the final boundary

b)

i	# misclassifications
0	6.0
1	1.0
2	5.0
3	4.0
4	0.0
5	1.0

$\bar{\theta} = [1.0, 3.0]$

$b = -7.0$

$\bar{\theta} = 6(-1)(-2,2) + (1)(-1)(0,1) + 5(-1)(2,1) + 4(1)(-1,4) + \cancel{0(1)(1,5)} + (1)(3,3)$

$= (12,-12) + (0,1) + (-10,-5) + (-4,16) + (3,3)$

$\boxed{= (1,3)}$ ✓

$b = 6(-1) + 1(-1) + 5(-1) + 4(1) + \cancel{0(1)} + 1(1)$

$\boxed{= -7}$ ✓

c) <u>No</u>, according to the lecture notes and piazza (@87), the algorithm will converge in a finite number of mistakes so long as the data is linearly seperable. Order would not affect this.

d) <u>Yes</u>, because we don't use a loss function to maximize the margin between points, the order could change $\bar{\theta}$ and $\bar{b}$ to be closer/further from a boundary that would seperate the points.

```python
"""
EECS 445 - Introduction to Machine Learning
HW1 Q5 Perceptron Algorithm with Offset
"""

import numpy as np
from helper import load_data

def all_correct(X, y, theta, b):
    """
    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        theta: np.array, shape (d,), normal vector of decision boundary
        b: float, offset

    Returns true if the linear classifier specified by theta and b correctly classifies all examples
    """

    for i in range(len(X)):
        for i in range(len(X)):
            pred = np.dot(theta, X[i]) + b
            if y[i] * pred <= 0:
                return False

    return True



def perceptron(X, y):
    """
    Implements the Perception algorithm for binary linear classification.
    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)

    Returns:
        theta: np.array, shape (d,)
        b: float
        alpha: np.array, shape (n,).
            Misclassification vector, in which the i-th element is has the number of times
            the i-th point has been misclassified)
    """
    n, d = X.shape
    theta = np.zeros((d,))
    b = 0.0
    alpha = np.zeros((n,))

    updated = True
    while updated:
        updated = False
        for i in range(n):
            margin = y[i] * (np.dot(theta, X[i]) + b)
            if margin <= 0:
                theta += y[i] * X[i]
                b += y[i]
                alpha[i] += 1
                updated = True

    return theta, b, alpha



def main(fname):
    X, y = load_data(fname)
    theta, b, alpha = perceptron(X, y)

    print("Done!")
    print("============== Classifier ==============")
    print("Theta: ", theta)
    print("b: ", b)

    print("\n")
    print("============== Alpha ==================")
    print("i \t Number of Misclassifications")
    print("=======================================")
    for i in range(len(alpha)):
        print(i, "\t\t", alpha[i])
    print("Total Number of Misclassifications: ", np.sum(alpha))


if __name__ == '__main__':
    main('dataset/q5.csv')
```

Q5

6.

a)

a/g	η	θ_0	θ_1	# itr	runtime
GD	10^{-4}	0.288	0.471	209417	9.867
GD	10^{-3}	0.287	0.474	26418	1.238
GD	10^{-2}	0.287	0.475	3188	0.148
GD	10^{-1}	0.287	0.475	371	0.0179
SGD	10^{-4}	0.287	0.474	279340	0.749
SGD	10^{-3}	0.286	0.475	29700	0.0808
SGD	10^{-2}	0.278	0.479	5840	0.016
SGD	10^{-1}	0.215	0.479	800	0.0023
closed	NA	0.287	0.475	NA	0.0002

b) SGD is much faster, uses slightly more iterations, and GD and SGD converge to similar coefficients, but SGD is likely "jittering" more under the hood where as GD has a smoother line.

c) Runtime of closed form is much faster than SGD. $\eta = 10^{-4}, 10^{-3}$ produce similar coefficients to the closed form solution

d) I used $lr = 0.25$ and then $lr /= (k+1)$ in my solution. This is fast (still slower than closed form) and takes 220 iterations. The resulting coefficients are not very close to closed form

---

7.1

a) $\nabla_{\bar{\theta}} J^{(i)}(\bar{\theta}) = \nabla_{\bar{\theta}} \left(\ln\left(1 + e^{-y^{(i)} \bar{\theta} \cdot \bar{x}^{(i)}}\right)\right)$

$z = \bar{\theta} \cdot \bar{x}^{(i)}$

$\frac{d}{dz}$ ↳ $= \frac{1}{1+e^{-y^{(i)}z}} \cdot e^{-y^{(i)}z} \cdot (-y^{(i)})$

$= (-y^{(i)}) \cdot \frac{e^{-y^{(i)}z}}{1+e^{-y^{(i)}z}}$

↳ now, for $\frac{dz}{d\theta} = \bar{x}^{(i)}$

$\nabla_{\bar{\theta}} J^{(i)}(\bar{\theta}) = (-y^{(i)}) \cdot \frac{e^{-y^{(i)}z}}{1+e^{-y^{(i)}z}} \cdot (\bar{x}^{(i)})$

$= -y^{(i)} \left(1 - \sigma(y^{(i)}z)\right) \bar{x}^{(i)}$

```python
import numpy as np
from helper import load_data
import time

def calculate_squared_loss(X, y, theta):
    """
    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        theta: np.array, shape (d,). Specifies an (d-1)^th degree polynomial

    Returns:
        The squared loss for the given data and parameters
    """
    n, _ = X.shape
    r = X @ theta - y
    return 0.5 * np.dot(r, r) / n

def ls_gradient_descent(X, y, learning_rate=0):
    """
    Implement the Gradient Descent (GD) algorithm for least squares regression.
    Note:
        - Please use the following stopping criteria together: number of iterations >= 1e6 or |new_loss - prev_loss| <= 1e-10

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)

    Returns:
        theta: np.array, shape (d,)
    """
    _, d = X.shape
    theta = np.zeros(d)

    eps = 1e-10
    max_iter = 1e6
    n_iter = 0

    step = learning_rate
    prev_loss = np.inf
    new_loss = calculate_squared_loss(X, y, theta)



    while (n_iter < max_iter) and (abs(new_loss - prev_loss) > eps):
        n_iter += 1
        grad = []

        for xt, yt in zip(X, y):
            pred = np.dot(theta, xt)
            g = (pred - yt) * xt
            grad.append(g)

        gradient = np.mean(grad, axis=0)
        theta -= gradient * step

        prev_loss = new_loss
        new_loss = calculate_squared_loss(X, y, theta)

    print("Learning rate:", learning_rate, "\t\t\tNum iterations:", n_iter)
    return theta

```

Q6

```python
def ls_stochastic_gradient_descent(X, y, learning_rate=0):
    """
    Implement the Stochastic Gradient Descent (SGD) algorithm for least squares regression.
    Note:
        - Please do not shuffle your data points.
        - Please use the following stopping criteria together: number of iterations >= 1e6 or |new_loss - prev_loss| <= 1e-10

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        learning_rate: the learning rate for the algorithm

    Returns:
        theta: np.array, shape (d,)
    """
    _, d = X.shape
    theta = np.zeros(d)
    adaptive = (learning_rate == 'adaptive')

    eps = 1e-10
    max_iter = 1e6
    n_iter = 0
    epochs = 0

    step = learning_rate
    prev_loss = np.inf
    new_loss = calculate_squared_loss(X, y, theta)

    if adaptive:
        step = 0.25    # best results
    else:
        step = 0.01 if learning_rate == 0 else float(learning_rate)

    while (n_iter < max_iter) and (abs(new_loss - prev_loss) > eps):
        epochs += 1

        # simply decrementing our learning rate as we go...gradually approaches 0
        if adaptive:
            step /= (epochs + 1)


        for xt, yt in zip(X, y):
            pred = np.dot(theta, xt)
            gradient = (pred - yt) * xt

            theta -= step * gradient
            n_iter += 1

            if n_iter >= max_iter:
                break

        prev_loss = new_loss
        new_loss = calculate_squared_loss(X, y, theta)

    print("Learning rate:", learning_rate, "\t\t\tNum iterations:", n_iter)
    return theta



def ls_closed_form_optimization(X, y):
    """
    Implement the closed form solution for least squares regression.

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)

    Returns:
        theta: np.array, shape (d,)
    """
    # piazza @85
    return np.linalg.pinv(X) @ y
```

Q6

Q6

```
def main(fname_train):
    # TODO: This function should contain all the code you implement to complete question 6.

    X_train, y_train = load_data(fname_train)

    # Appending a column of constant ones to the X_train matrix to make X_train the same dimensions as theta.
    # The term multiplied by theta_0 is x^0 = 1 (theta_0 is a constant), which is why the column contains only ones.
    X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))

    print("Closed-form:")
    t11 = time.process_time()
    theta_cf = ls_closed_form_optimization(X_train, y_train)
    t1 = t11 - time.process_time()
    loss_cf = calculate_squared_loss(X_train, y_train, theta_cf)
    print("Squared Loss:", loss_cf)
    print("Theta", np.linalg.norm(theta_cf))
    print("\n")


    print("Gradient Decent with LR=0.01:")
    t21 = time.process_time()
    theta_gd = ls_gradient_descent(X_train, y_train, learning_rate=0.01)
    t2 = t21 - time.process_time()
    loss_gd = calculate_squared_loss(X_train, y_train, theta_gd)
    print("Squared Loss:", loss_gd)
    print("Theta", np.linalg.norm(theta_gd))
    print("\n")

    print("Gradient Decent with LR=0.05:")
    t31 = time.process_time()
    theta_gd2 = ls_gradient_descent(X_train, y_train, learning_rate=0.05)
    t3 = t31 - time.process_time()
    loss_gd2 = calculate_squared_loss(X_train, y_train, theta_gd2)
    print("Squared Loss:", loss_gd2)
    print("Theta", np.linalg.norm(theta_gd2))
    print("\n")



    print("Stochastic Gradient Decent with LR=0.01:")
    t41 = time.process_time()
    theta_sgd = ls_stochastic_gradient_descent(X_train, y_train, learning_rate=0.01)
    t4 = t41 - time.process_time()
    loss_sgd = calculate_squared_loss(X_train, y_train, theta_sgd)
    print("Squared Loss:", loss_sgd)
    print("Theta", np.linalg.norm(theta_sgd))
    print("\n")

    print("Stochastic Gradient Decent with LR=0.05:")
    t51 = time.process_time()
    theta_sgd1 = ls_stochastic_gradient_descent(X_train, y_train, learning_rate=0.05)
    t5 = t51 - time.process_time()
    loss_sgd1 = calculate_squared_loss(X_train, y_train, theta_sgd1)
    print("Squared Loss:", loss_sgd1)
    print("Theta", np.linalg.norm(theta_sgd1))
    print("\n")

    print("Stochastic Gradient Decent with LR=\"adaptive\":")
    t61 = time.process_time()
    theta_sgd2 = ls_stochastic_gradient_descent(X_train, y_train, learning_rate='adaptive')
    t6 = t61 - time.process_time()
    loss_sgd2 = calculate_squared_loss(X_train, y_train, theta_sgd2)
    print("Squared Loss:", loss_sgd2)
    print("Theta", np.linalg.norm(theta_sgd2))
    print("\n")

    print("Done!")
```

b) $H^{(i)} = \nabla^2 J^{(i)}(\bar{\theta}) = s(1-s)\bar{x}^{(i)}\bar{x}^{(i)T}$, where $s = \sigma(y^{(i)}z$

$H^{(i)}\Delta = \nabla J^{(i)}$

$\Delta = \frac{-\nabla}{s\|x\|^2}\bar{x}^{(i)}$ $\rightarrow$ $\boxed{\bar{\theta}^{(k+1)} = \bar{\theta}^{(k)} + \eta\frac{y^{(i)}}{\sigma(y^{(i)}\bar{\theta}^{(k)T}\bar{x}^{(i)})\|\bar{x}^{(i)}\|^2}\bar{x}^{(i)}}$

c) according to my model,

$\bar{\theta} = [0.28465, -0.01356, 0.02133]$

d) in linear regression, the loss is quadratic, so we can solve for θ directly. However, in logistic regression, we use the sigmoid, $pred_i = \sigma(\bar{\theta}^T\bar{x}_i) = \frac{1}{1+e^{-\bar{\theta}^T\bar{x}_i}}$ and when we set the gradient of our loss we get non-linear equations because P depends on θ through sigmoid. So, we <u>cannot</u> find a closed form and must solve iteratively.

7.2.

a) for $J^{(i)}(\bar{\theta}) = \ln(1+e^{-y^{(i)}\bar{\theta}^T\bar{x}^{(i)}})$ and $\nabla_\theta J^{(i)}(\bar{\theta}) = -y^{(i)}\sigma(-y^{(i)}z)\bar{x}^{(i)}$ (from 7.1a)

let $z = y^{(i)}\bar{\theta}^T\bar{x}^{(i)}$

$\nabla_\theta^2 J^{(i)}(\bar{\theta}) = -y^{(i)}\bar{x}^{(i)}(\nabla_\theta\sigma(-z))$

where $\sigma'(t) = \sigma(t)(1-\sigma(t))$

$\nabla_\theta\sigma(-z) = \sigma'(-z)\nabla_\theta(-z) = \sigma(-z)(1-\sigma(-z))(-y^{(i)}\bar{x}^{(i)})$

↳ $\nabla_\theta^2 J^{(i)}(\theta) = -y^{(i)}\bar{x}^{(i)}(\sigma(-z)(1-\sigma(-z))(-y^{(i)}\bar{x}^{(i)}))^T$

apply $(y^{(i)})^2 = 1$ and $\sigma(-z) = 1-\sigma(z)$ $\longrightarrow$

$$\nabla_\theta^2 J^{(i)}(\bar{\theta}) = \sigma(z)(1-\sigma(z))\,\bar{x}^{(i)}(\bar{x}^{(i)})^T$$

$$\hookrightarrow \boxed{H_{jk}^{(i)} = \sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})\left(1-\sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})\right)\bar{x}_j^{(i)}\,\bar{x}_k^{(i)}}$$

$$\boxed{H^{(i)} = \sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})\left(1-\sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})\right)\bar{x}^{(i)}\,x^{(i)T}}$$

b) $\theta^{(k+1)} = \theta^{(k)} - \eta\,(H^{(i)})^{-1}\,\nabla_{\bar{\theta}} J^{(i)}(\bar{\theta})$

let $s = \sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})$ and $\Delta = c\bar{x}^{(i)}$

$\nabla J^{(i)}(\theta) = -y^{(i)}(1-s)\,\bar{x}^{(i)}$, $H^{(i)} = s(1-s)\,\bar{x}^{(i)}\bar{x}^{(i)T}$

$\hookrightarrow s(1-s)\,\bar{x}^{(i)}\bar{x}^{(i)T}(c\bar{x}^{(i)}) = c(s(1-s))\,\|\bar{x}^{(i)}\|\,\bar{x}^{(i)} = -y^{(i)}(1-s)x$

$\hookrightarrow c = -\dfrac{y^{(i)}}{s\|\bar{x}^{(i)}\|^2}$ and $\Delta = -\dfrac{y^{(i)}}{s\|\bar{x}^{(i)}\|^2}$ ($\Delta = cx$)

plugging back in ... $\bar{\theta}^{(k+1)} = \bar{\theta}^{(k)} + \eta\,\dfrac{y^{(i)}}{\sigma(y^{(i)}\bar{\theta}^T\bar{x}^{(i)})\,\|\bar{x}^{(i)}\|^2}\,\bar{x}^{(i)}$

c) according to my model...

$\bar{\theta} = [0.29492,\ 0.04274,\ 0.04487]$

d) the method still remains noisy with θ changing to.

$[0.50714,\ 0.04244,\ 0.04233]$, so it doesn't seem to have converged with 2,000 epochs. This isn't surprising beause we sample single points which can cause a "shake" or "jitter" in θ.

```python
import numpy as np
from helper import load_data

def sigmoid(z):
    """
    Implements the sigmoid function..
    Args:
        z: A scalar or numpy array of any size
    """
    # handles scalars
    z = np.asarray(z)
    return 1.0 / (1.0 + np.exp(-z))


def logistic_stochastic_gradient_descent(X, y, lr=0.0001):
    """
    Implements the Stochastic Gradient Descent (SGD) algorithm for logistic regression.
    Note:
        - Please do not shuffle your data points.
        - Please use the stopping criteria: number of epochs == 10,000

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        lr: the learning rate for the algorithm

    Returns:
        theta: np.array, shape (d+1,) including the offset term.
    """
    n, d = X.shape
    theta = np.zeros(d + 1)
    step = lr

    Xbar = np.hstack([np.ones((n, 1)), X])

    for _ in range(10000):

        # no shuffle
        for xi, yi in zip(Xbar, y):
            theta_dot_x = np.dot(theta, xi)
            gradient = -1 * yi * xi * sigmoid(-yi * theta_dot_x)
            theta -= step * gradient

    return theta


def stochastic_newtons_method(X, y, lr=0.0001):
    """
    Implements the Stochastic Newton's Method algorithm for logistic regression.
    Note:
        - Please do not shuffle your data points.
        - Please use the stopping criteria: number of epochs == 1,000

    Args:
        X: np.array, shape (n, d)
        y: np.array, shape (n,)
        lr: the learning rate for the algorithm

    Returns:
        theta: np.array, shape (d+1,) including the offset term.
    """
    n, d = X.shape
    theta = np.zeros(d + 1)
    step = lr

    Xbar = np.hstack([np.ones((n, 1)), X])
    eps = 1e-10

    #epochs = 1000
    epochs = 2000

    for _ in range(epochs):

        # no shuffle
        for xi, yi in zip(Xbar, y):
            margin = np.dot(theta, xi)
            sigmoid_signed = sigmoid(yi * margin)

            beta = -1 * yi * (1.0 - sigmoid_signed)
            alpha = sigmoid_signed * (1.0 - sigmoid_signed)

            # add eps to avoid 0 denom
            newton_step = (beta / (alpha * (xi @ xi) + eps) ) * xi
            theta -= step * newton_step


    return theta
```

Q7