```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

```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

```python
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!")
```

```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