python codes

```
from IPython import display
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import warnings
warnings.filterwarnings("ignore")
import numpy as np
import matplotlib.pyplot as plt


# ## **(b) i**


# define the matrix A and vector b
def get_A_b(z, U):
    '''
    params:
        z: the vector of the labels (N, 1)
        U: the extended features (N, d)
    return:
        A: shape (N + 1, 1)
        b: shape (N + 1, 1)
    '''
    zTz = z @ z.T
    UTU = U @ U.T
    A = np.concatenate((np.concatenate((zTz * UTU, -z), axis=1),
```

```
                        np.concatenate((z.T, np.zeros((1, 1))), axis=1)),
                        axis=0)
    b = np.concatenate((np.ones((z.shape[0], 1)), np.zeros((1, 1))), axis=0)
    return A, b


def get_lamda_mu(A, b):
    '''
    params:
        A: matrix shape (N + 1, N + 1)
        b: vector shape (N + 1, 1)
    '''
    sol = (np.linalg.inv(A) @ b).ravel()
    lamda, mu = sol[:A.shape[0]-1], sol[-1]
    ## check lamda is greater than zero
    while np.any(lamda < 0):
        idx1 = [i for i in range(lamda.shape[0]) if lamda[i] >= 0]
        idx = idx1 + [sol.shape[0] - 1]
        A = A[idx, :][:, idx]
        b = b[idx, :]
        sol = (np.linalg.inv(A) @ b).ravel()
        tmp, mu = sol[:A.shape[0]-1], sol[-1]
        lamda[lamda<0] = 0
        lamda[idx1] = tmp

    print("lamda is ", lamda)
    print("mu is ", mu)

    return lamda, mu


```

```python
# ## **(b) ii**


def check_KKT12(lamda, z):
    '''
    params:
        lamda: lagaurange multiplier. vector shape (1, N)
        z: the vector of the labels (N, 1)
    '''
    print("KKT conditions 1 is satisfied: ", np.allclose(lamda@z, 0))
    print("KKT conditions 2 is satisfied: ", np.all(lamda>=0) or np.allclose(lamda, 0))


# ## **(b) iii**


def get_w_w0(lamda, z, U):
    '''
    params:
        lamda: lagaurange multiplier. vector shape (1, N)
        z: the vector of the labels (N, 1)
        U: the extended features (N, d)
    returns:
        w: the weights, vector shape (1, N)
        w0: the bias, scalar
    '''
    w = lamda.reshape((1, -1)) @ (z * U)
    w0 = 1 / z[0, 0] - w @ U[0:1].T
    print("w is {}, w0 is {}".format(w, w0))

```

```
    return w, w0


# ## **(b) iv**


def check_KKT3(z, U, w, w0):
    '''
    params:
        z: the vector of the labels (N, 1)
        U: the extended features (N, d)
        w: the weights, vector shape (1, N)
        w0: the bias, scalar
    '''
    z_hat = U @ w.T + w0
    cond3 = z * z_hat - 1
    if np.all(cond3 >=0) or np.allclose(cond3, 0):
        res = True
    else:
        res = False
    print("KKT conditions 3 is satisfied: ", res)


# ## **(c)**


z = np.array([[1], [1], [-1]])
U = np.array([[1, 2], [2, 1], [0, 0]])
A, b = get_A_b(z, U)
lamda, mu = get_lamda_mu(A, b)
check_KKT12(lamda, z)
w, w0 = get_w_w0(lamda, z, U)
check_KKT3(z, U, w, w0)
```

```python

# ## **(d)**


def plot_decision_boundary(training, label_train, w, w0):
    # Total number of classes
    classes = np.unique(label_train)
    nclass = len(classes)

    class_names = []
    for c in classes:
        class_names.append('Class ' + str(int(c)))

    # Set the feature range for plotting
    max_x1 = np.ceil(np.max(training[:, 0])) + 1.0
    min_x1 = np.floor(np.min(training[:, 0])) - 1.0
    max_x2 = np.ceil(np.max(training[:, 1])) + 1.0
    min_x2 = np.floor(np.min(training[:, 1])) - 1.0

    xrange = (min_x1, max_x1)
    yrange = (min_x2, max_x2)

    # step size for how finely you want to visualize the decision boundary.
    inc = 0.005

    # generate grid coordinates. This will be the basis of the decision boundary visualization.
    (x1, x2) = np.meshgrid(np.arange(xrange[0], xrange[1] + inc / 100, inc),
                           np.arange(yrange[0], yrange[1] + inc / 100, inc))

```

```
    # size of the (x1, x2) image, which will also be the size of the
    # decision boundary image that is used as the plot background.
    image_size = x1.shape
    # make (x1, x2) pairs as a bunch of row vectors.
    grid_2d = np.hstack((x1.reshape(x1.shape[0] * x1.shape[1], 1, order='F'),
                         x2.reshape(x2.shape[0] * x2.shape[1], 1, order='F')))

    pred_label = np.where(grid_2d @ w.T + w0 > 0, 1, -1)
    # reshape the idx (which contains the class label) into an image.
    decision_map = pred_label.reshape(image_size, order='F')

    # create fig
    fig, ax = plt.subplots()
    ax.imshow(decision_map, vmin=np.min(classes), vmax=9, cmap='Pastel1',
              extent=[xrange[0], xrange[1], yrange[0], yrange[1]],
              origin='lower')

    # plot the class training data.
    data_point_styles = ['rx', 'bo', 'g*']
    for i in range(nclass):
        ax.plot(training[label_train == classes[i], 0],
                training[label_train == classes[i], 1],
```

```python
                data_point_styles[int(classes[i]) - 1],
                label=class_names[i])
    ax.legend()

    plt.tight_layout()
    plt.show()

    return fig


fig = plot_decision_boundary(U, z.ravel(), w, w0)
fig.savefig("./figs/2d.png", dpi=200)


# ## **(f)**


z = np.array([[1], [1], [-1]])
U = np.array([[1, 2], [2, 1], [1, 1]])
A, b = get_A_b(z, U)

# (i)
lamda, mu = get_lamda_mu(A, b)

# (ii)
check_KKT12(lamda, z)

# (iii)
w, w0 = get_w_w0(lamda, z, U)

# (iv)
check_KKT3(z, U, w, w0)

```

```python

fig = plot_decision_boundary(U, z.ravel(), w, w0)
fig.savefig("./figs/2f4.png", dpi=200)


# ## **(g)**


z = np.array([[1], [1], [-1]])
U = np.array([[1, 2], [2, 1], [0, 1.5]])
A, b = get_A_b(z, U)

# (i)
lamda, mu = get_lamda_mu(A, b)

# (ii)
check_KKT12(lamda, z)

# (iii)
w, w0 = get_w_w0(lamda, z, U)

# (iv)
check_KKT3(z, U, w, w0)


fig = plot_decision_boundary(U, z.ravel(), w, w0)
fig.savefig("./figs/2g4.png", dpi=200)
```