HW6_P2

April 2, 2022

1 Problem 2

[5]:
```
#!/usr/bin/env python
# coding: utf-8

##################################################
# Author:   Sarthak Kumar Maharana
# Email:    maharana@usc.edu
# Date:     03/30/2022
# Course:   EE 559
# Project:  Homework 6
# Instructor: Prof. B Keith Jenkins
##################################################


#import libraries

import sys
import numpy as np
import matplotlib.pyplot as plt
```

[12]:
```
class SVM:
    def __init__(self, u, z, idx):
        self.u = u
        self.z = z
        self.data_no = idx # dataset number

    def solve_lambda_mu(self):
        """
        To calculate lambda vector and mu. If any lambda < 0, re-optimise w/␣
↪the remaining
        lambdas. (Derivation in the answer sheet, attached later).

        Parameters
        -----------
        None

```

```
        Returns
        -------
            : ndarray
            lambda vector
            : float
            mu
        """
        z_u = np.zeros(self.u.shape[0]) # N x D
        z_u = np.array([
            np.dot(self.z[idx], self.u[idx])
            for idx in range(len(self.u))
        ]) # dot product of z * u^T

        A = np.concatenate(
            (np.concatenate(
            (z_u @ z_u.transpose(), -self.z.reshape(-1 ,1)),
            axis = 1), [np.append(self.z, 0)]),
            axis = 0
        ) # (z * u^T)^T * (z * u^T) and the append a -z column, hstack z and 0.
        b = np.append(np.ones(self.z.shape[0]).reshape(-1, 1), 0)
        rho = np.dot(np.linalg.pinv(A), b)

        # if any lambda is < 0 (re-optimise)
        if np.any(rho[:2] < 0):
            #index where it is -ve
            index = np.where(rho[:2] < 0)[0][0]
            # set lambda at that index = 0.0
            rho[index] = 0.0
            z, u = self.z.copy(), self.u.copy()
            # re-compute lambdas for the remaining
            z, u = np.delete(z, index), np.delete(u, (index), axis = 0)
            z_u_spec = np.array([
                np.dot(z[idx], u[idx])
                for idx in range(len(u))
            ])
            A_star = np.concatenate(
            (np.concatenate(
            (z_u_spec @ z_u_spec.transpose(), -z.reshape(-1 ,1)),
            axis = 1), [np.append(z, 0)]),
            axis = 0
            )
            b_star = np.append(np.ones(z.shape[0]).reshape(-1, 1), 0)
            rho_star = np.dot(np.linalg.pinv(A_star), b_star)
            rho  = np.concatenate((rho_star[:index], [0.0], rho_star[index:]))
            return np.round(rho[:-1], 4), np.round(rho[-1], 4)
        return np.round(rho[:-1], 4), np.round(rho[-1], 4)
```

```
    def KKT_check_lambda(self, lambda_vec):
        """
        To check if the obtained lambda values satisfy the KKT conditions.

        Parameters
        ----------
        lambda_vec: ndarray
            obtained lambdas

        Returns
        -------
        lambda_val_flag: bool
            True if condition all >=0.s
        lambda_z_flag: bool
            True if the sum(z_i * lambda_i) == 0
        """
        # check if lambda is >- 0 (boolean)
        lambda_val_flag = np.all((lambda_vec >= 0.0))
        lambda_z_sum = 0.0
        for idx in range(len(lambda_vec)):
            lambda_z_sum += self.z[idx] * lambda_vec[idx]
        # check if sum(z_i * lambda_i) == 0.0
        lambda_z_flag = True if lambda_z_sum == 0.0 else False
        return lambda_val_flag, lambda_z_flag

    def optimal_weights(self, lambda_vec):
        """
        To calculate optimal weight vector and bias.

        Parameters
        ----------
        lambda_vec: ndarray
            obtained lambdas

        Returns
        -------
        None
        """
        self.weights = np.zeros((1, self.u.shape[1]))
        # calculate optimal weights (sum(z_i * lambda_i * u_i))
        for idx in range(len(self.u)):
            self.weights += lambda_vec[idx]*self.z[idx]*self.u[idx]
        # bias calculation (for i = 0)
        self.bias = (1/self.z[0]) - (np.dot(self.weights, self.u[0])).
↪tolist()[0]

    def KKT_check_weights_bias(self):
```

```
        """
        To check if the optimal weights and bias satisfy the KKT conditions.

        Parameters
        ----------
        None

        Returns
        -------
        w_flag: bool
            True, if yes else False.
        """

        #check if z[i]*[(w*u[i] + w_0) - 1] >= 0
        w_flag = True if (self.z[0]*(np.dot(self.weights, self.u[0]) + self.
↪bias) - 1) >= 0 else False
        return w_flag

    def _plot(self):
        """
        To plot the data points, class labels, decision boundary of the SVM and␣
↪support vectors.

        Parameters
        ----------
        None

        Returns
        -------
        None
        """
        classes = np.unique(self.z)
        class_names = ['Class 1' if int(cl) == 1 else 'Class 2' for cl in␣
↪classes]
        data_point_styles = ['rx', 'bo']

        _, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (8, 6), dpi = 200)

        for idx in range(len(classes)):
            plt.plot(
                self.u[self.z == classes[idx]][:, 0],
                self.u[self.z == classes[idx]][:, 1],
                data_point_styles[idx],
                label = class_names[idx]
            )
        ax.legend()
```

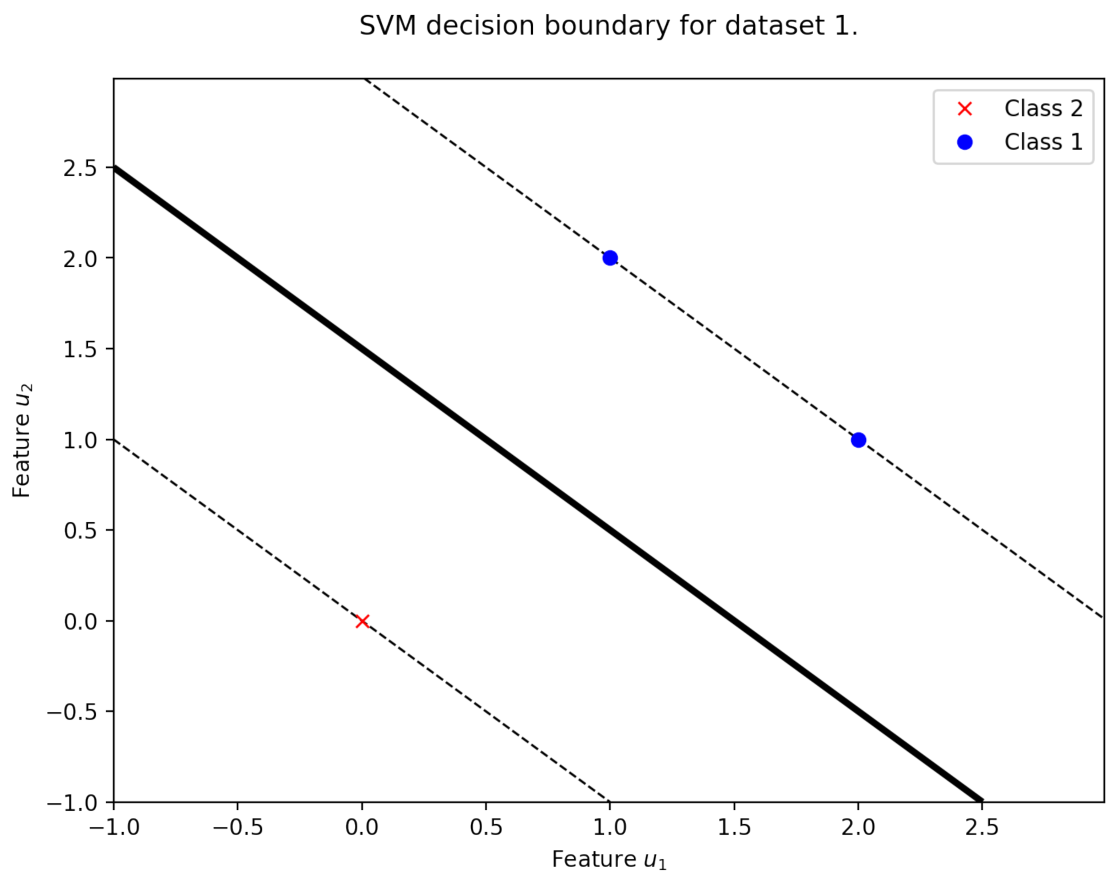
```
        x_min, x_max = np.ceil(self.u[:, 0].min()) - 1, np.ceil(self.u[:, 0].
↪max()) + 1
        y_min, y_max = np.ceil(self.u[:, 1].min()) - 1, np.ceil(self.u[:, 1].
↪max()) + 1

        xx, yy = np.meshgrid(
            np.arange(x_min, x_max, 0.01),
            np.arange(y_min, y_max, 0.01)
        )

        weight_vec = np.dot(self.weights, np.array([xx.ravel(), yy.ravel()])) +␣
↪self.bias
        support_vec1 = weight_vec - 1
        support_vec2 = weight_vec + 1

        # SVM boundary
        ax.contour(
            xx,
            yy,
            weight_vec.reshape(xx.shape),
            levels = [0],
            colors = 'k',
            linestyles = 'solid',
            linewidths = 3,
        )
        # Support vector
        ax.contour(
            xx,
            yy,
            support_vec1.reshape(xx.shape),
            levels = [0],
            colors = 'k',
            linestyles = '--',
            linewidths = 1,
        )
        # Support vector
        ax.contour(
            xx,
            yy,
            support_vec2.reshape(xx.shape),
            levels = [0],
            colors = 'k',
            linestyles = '--',
            linewidths = 1,
        )
        ax.set_xlabel('Feature $u_{1}$')
        ax.set_ylabel('Feature $u_{2}$')
```

```
        ax.set_title(f'SVM decision boundary for dataset {self.data_no}.\n')
        plt.show()

    def _runner(self):
        """
        Method to run scripts and answer all questions for all the datasets.

        Parameters
        ----------
        None

        Returns
        -------
        None
        """
        print(f"--- Running SVM for dataset {self.data_no} i.e \n {self.u} ---")
        lambda_vec, mu = self.solve_lambda_mu()
        print(f"Lambda vector: {lambda_vec}, mu: {mu}")
        lam_all_vals, lam_sum = self.KKT_check_lambda(lambda_vec)
        if lam_all_vals and lam_sum:
            print('KKT conditions are satisfied, involving lambda.')
            self.optimal_weights(lambda_vec)
            print(f"Optimal weights: {self.weights}, bias: {self.bias}")
            print(f"KKT conditions on the optimal weights and bias: {self.
↪KKT_check_weights_bias()}")
            self._plot()
        else:
            sys.exit('KKT conditions are not satisfied, involving lambda.')
```

[13]:
```
if __name__ == '__main__':
    u_1 = np.array([
        [1, 2],
        [2, 1],
        [0, 0],
    ])
    u_2 = np.array([
        [1, 2],
        [2, 1],
        [1, 1],
    ])
    u_3 = np.array([
        [1, 2],
        [2, 1],
        [0, 1.5],
    ])
    z = np.array([1, 1, -1])
    for idx in range(1, 4):
```

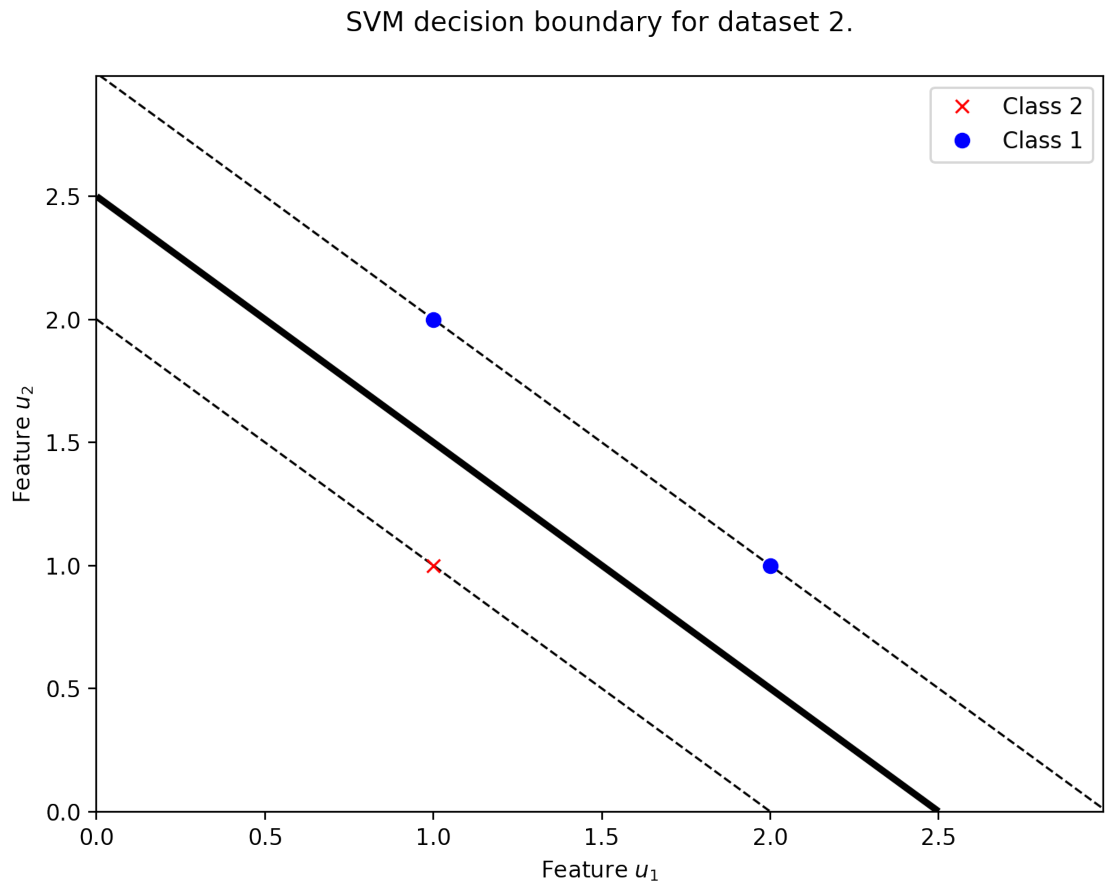
```python
        hw6 = SVM(eval(f"u_{idx}"), z, idx)
        print(hw6._runner())
```

```
--- Running SVM for dataset 1 i.e
 [[1 2]
 [2 1]
 [0 0]] ---
Lambda vector: [0.2222 0.2222 0.4444], mu: 1.0
KKT conditions are satisfied, involving lambda.
Optimal weights: [[0.6666 0.6666]], bias: -0.9998000000000002
KKT conditions on the optimal weights and bias: True
```

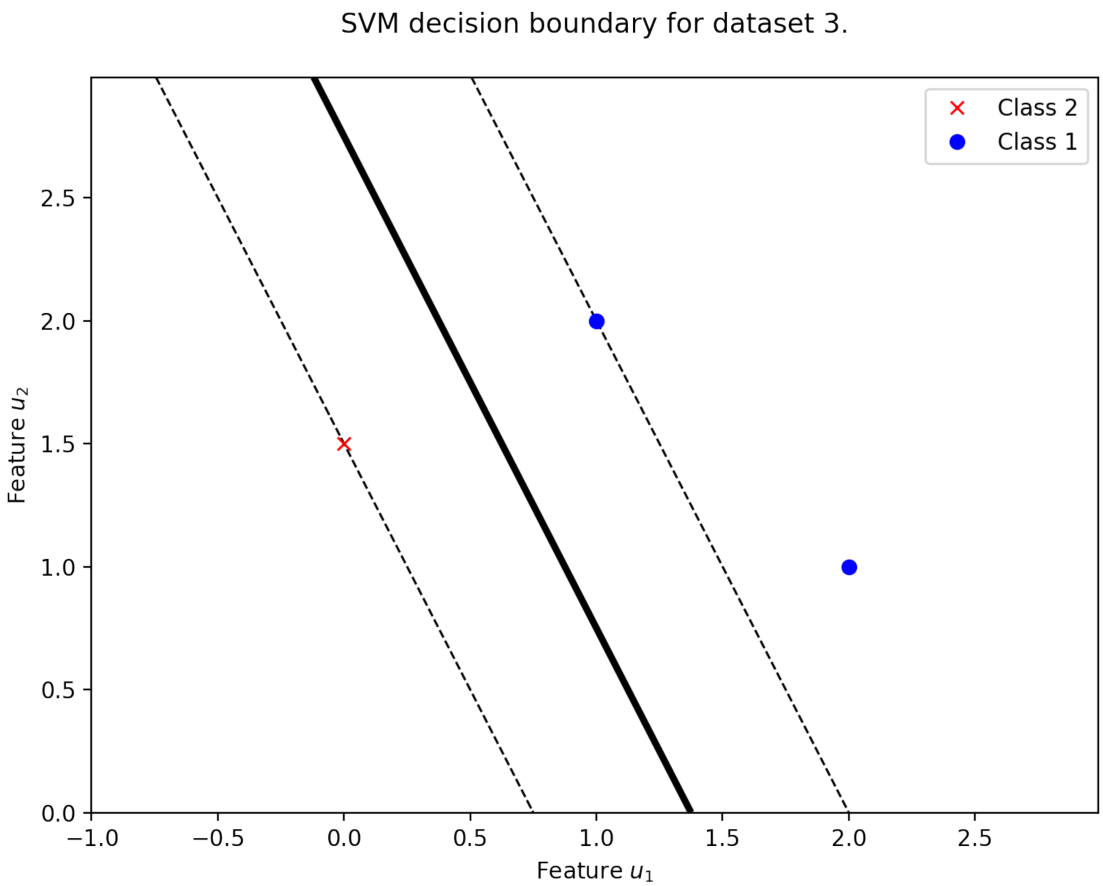



```
None
--- Running SVM for dataset 2 i.e
 [[1 2]
 [2 1]
 [1 1]] ---
Lambda vector: [2. 2. 4.], mu: 5.0
KKT conditions are satisfied, involving lambda.
Optimal weights: [[2. 2.]], bias: -5.0
```

```
KKT conditions on the optimal weights and bias: True
```




```
None
--- Running SVM for dataset 3 i.e
 [[1.  2. ]
 [2.  1. ]
 [0.  1.5]] ---
Lambda vector: [1.6 0.  1.6], mu: 2.2
KKT conditions are satisfied, involving lambda.
Optimal weights: [[1.6 0.8]], bias: -2.1999999999999997
KKT conditions on the optimal weights and bias: True
```


SVM decision boundary for dataset 3.
Class 2
Class 1
Feature u_2
Feature u_1
0.0
0.5
1.0
1.5
2.0
2.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

```
None
```

[ ]: