[CS 376] Machine Learning

Assignment #3: Support Vector Machines

Joyce Jiyoung Whang

KAIST School of Computing

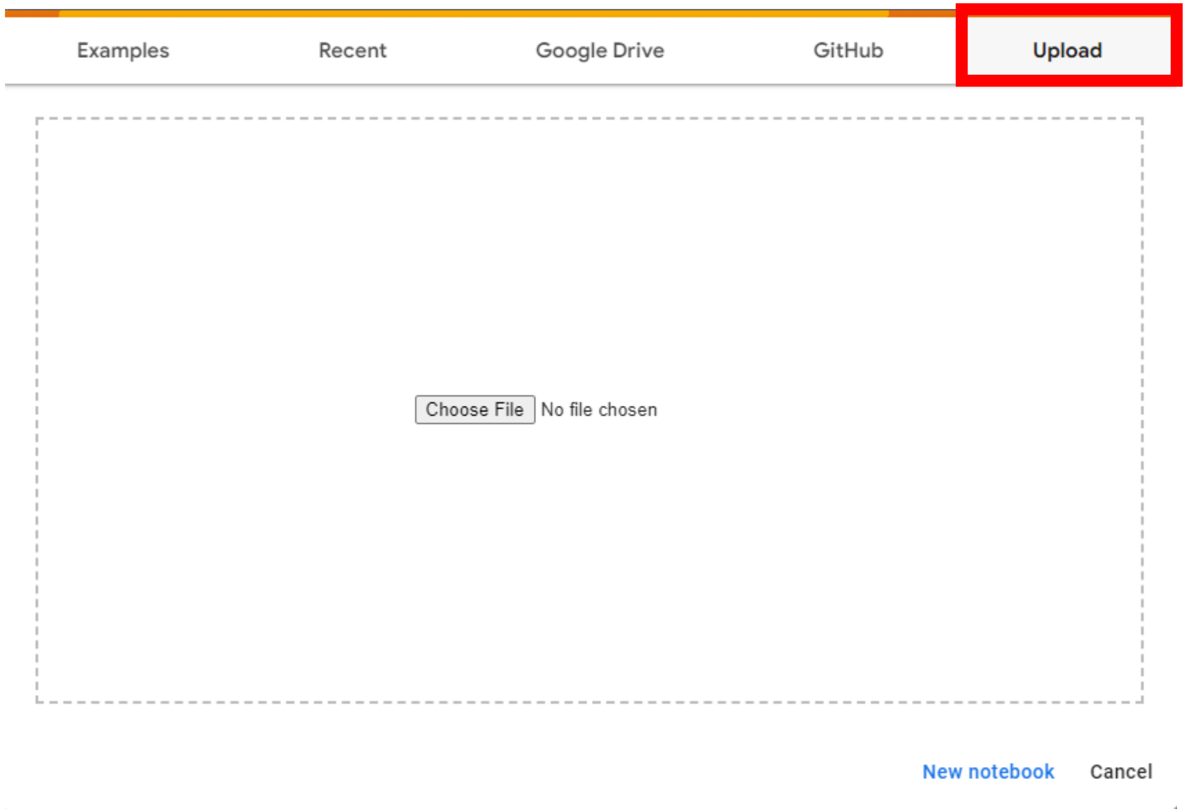
Preliminaries

Google Colab

In this assignment, we will provide you files in ipynb (Jupyter Notebook) format. You can open this file with Google Colab, which you can use freely with a Google account.

First, go to https://colab.research.google.com. Then, press the "Upload" tab and upload the provided ipynb file. Now, you can read and make changes to it.

A folder named Colab Notebooks will be created in your Google Drive, so you can access uploaded notebooks here.



Preliminaries

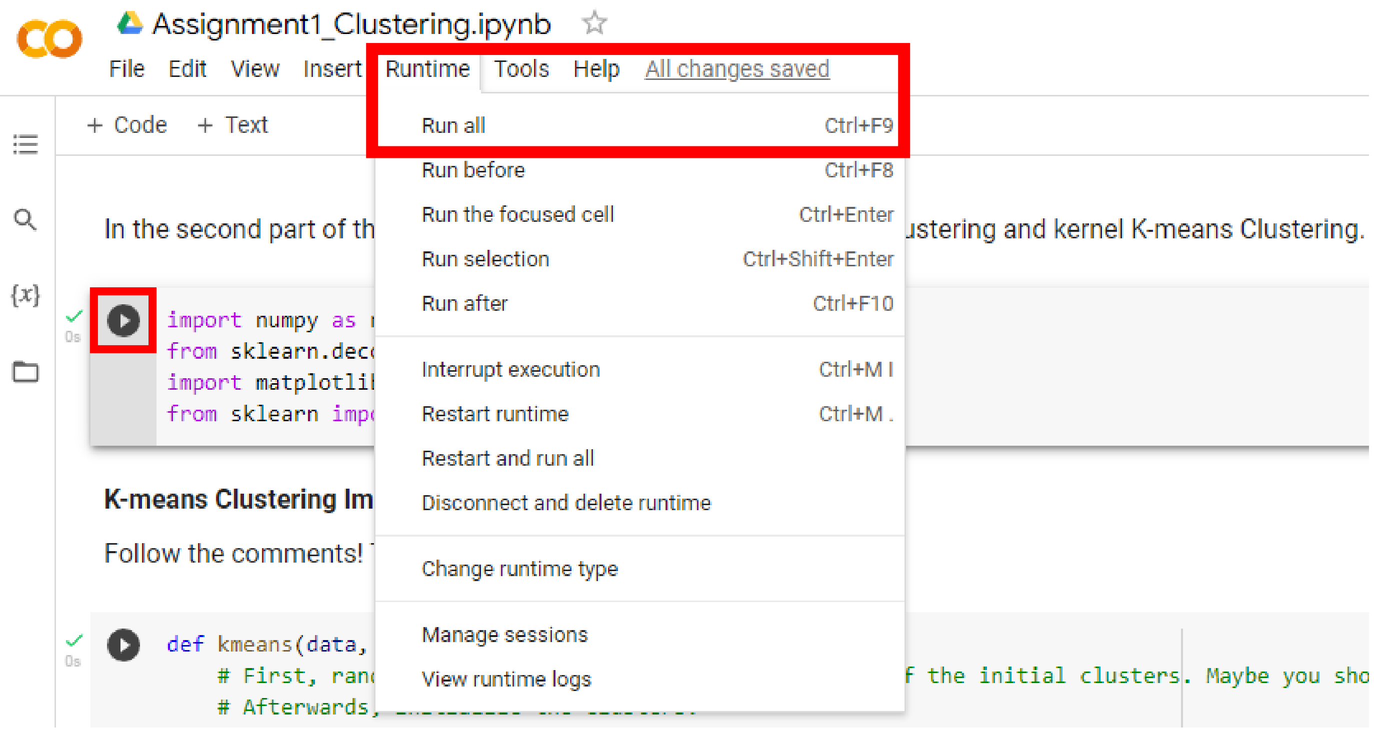
Google Colab

Here are some basic tips for running notebooks in Google Colab.

If you want to run the entire code in the notebook, you can either press Ctrl+F9, or go through the tabs “Runtime” -> “Run all”.

Blocks in notebooks are called cells. If you want to run a single cell, press Ctrl+Enter, or press the Play button, which is located at the top-left corner of each cell.

Variables in previous cells are preserved once you run them, so you do not have to rerun all cells every time you change and run a single cell.

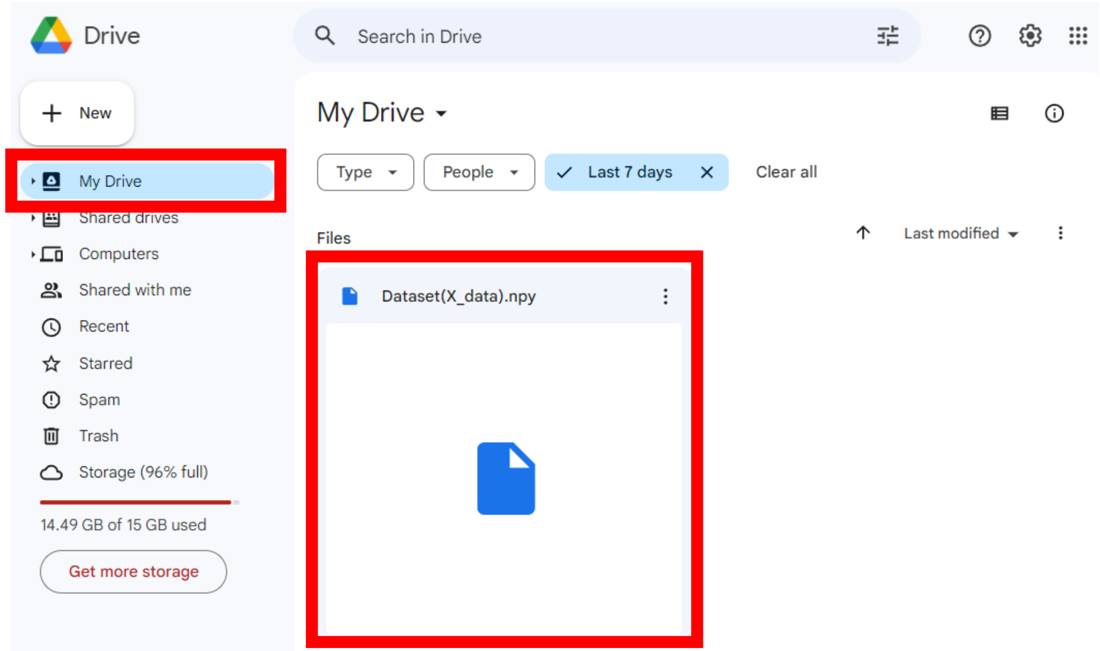


Preliminaries

Google Drive

In this assignment, you should upload the given dataset (e.g. Dataset0(X_data).npy file) to Google Drive.

You need to save the datasets in MyDrive folder.



Preliminaries

NumPy

In this assignment, you will need to handle multi-dimensional data. NumPy is a Python library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. Here are some basic functions in NumPy that you may find useful for this assignment. For more details on these functions, check the official NumPy documentation.

Notations

a, b: NumPy arrays. n, m, k: Integers. np: NumPy (after running the line "import numpy as np").

np.arange(n, m, k): It returns evenly spaced values within a given interval (n: start, m: stop, k: step).
np.identity(n): It returns the identity array which is a square array with ones on the main diagonal (n: number of rows in n x n output).
np.linalg.norm(a, axis=None): It returns the matrix/vector norm of a. If an axis is specified, the norm will be performed along that axis.
np.load(path): It loads arrays or pickled objects from .npy, .npz, or pickled files corresponding to the path.
np.meshgrid(a, b): It returns an n×m-dimensional rectangular matrix by taking two one-dimensional arrays a and b.
np.ones((n,m)): It returns a new array of given shape and type, filled with ones (n: row shape, m: column shape).
np.outer(a, b): It compute the outer product of two vectors a and b.
np.random.randn(k): It returns a sample (or samples) from the standard normal distribution (k: number of sample).
np.argwhere(a): It finds the indices of array elements that are non-zero, grouped by element.
np.sign(a): It returns an element-wise indication of the sign of a number.
np.sum(a, axis=None): It adds all elements of a. If the axis is specified, the sum is performed along that axis.
np.zeros((n,m)): It returns a new array of given shape and type, filled with zeros (n: row shape, m: column shape).

Preliminaries

CVXOPT

In this assignment, you will need to solve the convex optimization problem. CVXOPT is a Python library for convex optimization. Its main purpose is to make the development of software for convex optimization applications straightforward by building on Python's extensive standard library and on the strengths of Python as a high-level programming language. Check the official [CVXOPT documentation].

Notations

P, q, G, h, A, b: NumPy arrays. n, m, k: Integers. cvx: CVXOPT (after running the line "import cvxopt as cvx").

cvx.matrix(P, tc='d'): It converts the NumPy array to the CVXOPT type matrix object (tc='d': double data type).
cvx.matrix(k, (n,m)): It returns the n×m-dimensional rectangular matrix with the value of k.
cvx.solvers.qp(P, q, G, h, A, b)['x']: It provides the option of the quadratic programming solver. The standard format of quadratic programming supported by CVXOPT is as follows:

$$\min_{\boldsymbol{x}} \frac{1}{2}\boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{x}, \quad s.t.\ \boldsymbol{G}\boldsymbol{x} \le \boldsymbol{h}, \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

(here, $\le$ for vector means component-wise vector inequality).

Support Vector Machines

In this assignment, you should implement the soft margin Support Vector Machines (SVM) and the Kernel SVM by following the steps below:

Step 1. Build the soft margin SVM models based on the gradient descent.

Step 2. Load the dataset 1.

Step 3. Train and test the soft margin SVM models with different learning rates.

Step 4. Build the Kernel SVM model based on the dual form.

Step 5. Load the dataset 2.

Step 6. Train and test the Kernel SVM models with different types of kernels.

[Step 1] Build the soft margin SVM models based on the gradient descent

In this assignment, you should use only **NumPy** to build the soft margin SVM models. **Do not use other libraries.** The followings are the skeleton codes. Implement the functions inside the SVM class: objective_function, calculate_gradient, train, and predict.

```
class SVM():
    def __init__(self, X_data, reg=1):
        # Initialize the model parameters as the instance variable.
        print("-"*50, "\nInitialize the parameters of our SVM.\n")
        self.W = np.random.randn(X_data.shape[1])
        self.w_0 = np.random.randn(1)
        self.reg = reg

    # Function for calculating the loss.
    def objective_function(self, X_data, y_label):
        # Define the objective function for the SVM classifier.
        ######################## IMPLEMENT HERE ########################
        cost = NotImplemented
        raise NotImplementedError
        ################################################################
        return cost

    # Function for calculating the gradient
    def calculate_gradient(self, X_data, y_label):
        # Compute the gradient of W and w_0 from the entire epoch.
        ######################## IMPLEMENT HERE ########################
        gradient_W = NotImplemented
        gradient_w_0 = NotImplemented
        raise NotImplementedError
        ################################################################
        return gradient_W, gradient_w_0
```

Soft margin SVM in terms of the loss

- **Objective function**

$$f(\boldsymbol{w}) := \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{N}\sum_{i=1}^{N}\max\big(0, 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0)\big)$$

- **Gradient**

$$\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) = \lambda\boldsymbol{w} - \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}\{y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) < 1\}\cdot y_i\boldsymbol{x}_i$$

$$\nabla_{w_0} f(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N}\mathbb{1}\{y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) < 1\}\cdot y_i$$

[Step 1] Build the soft margin SVM models based on the gradient descent

In this assignment, you should use only **NumPy** to build the soft margin SVM models. **Do not use other libraries.** The followings are the skeleton codes. Implement the functions inside the SVM class: objective_function, calculate_gradient, train, and predict.

```
# Function for training
def train(self, x_input, y_label, lr=0.01, epochs=50):
    # Update the model parameters (W, w_0) using the above calculate_gradient function.
    print("- Total Epochs \t\t: ", epochs)
    print("- Learning rate \t: ", lr)
    print("-"*50)
    for epoch in range(epochs):
        ######################## IMPLEMENT HERE ########################
        cost = NotImplemented
        gradient_W, gradient_w_0 = NotImplemented
        self.W += NotImplemented
        self.w_0 += NotImplemented
        raise NotImplementedError
        ################################################################
        if epoch % 10 == 0:
            print("[Epoch : %d/%d]\t\t| Cost: %.4f"%(epoch+10, epochs, cost))
    print("-"*50, "\nLearning is complete.\n")
    print("- W \t:", self.W)
    print("- W_0 \t:", self.w_0)

# Function for prediction
def predict(self, x_input):
    # Compute our SVM prediction, and transform it into the binary label.
    ######################## IMPLEMENT HERE ########################
    result = NotImplemented
    raise NotImplementedError
    ################################################################
    return result

# Function for evaluation
def accuracy(self, predict, y_label):
    result = np.sum(predict == y_label) / len(predict)
    return result
```

[Step 2] Load the dataset 1

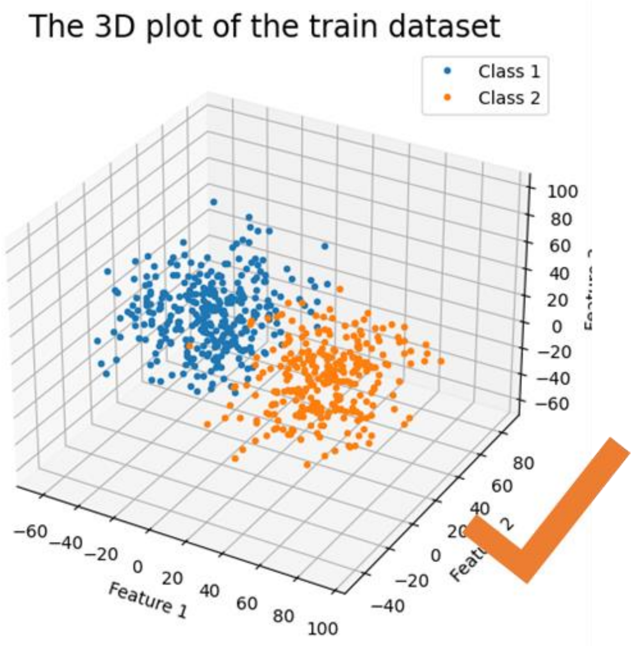
The dataset is the isotropic Gaussian blobs with some noise.

Load the train, validation, and test dataset by following the provided codes.

```
# Load the train, validation, test dataset.
train = np.load("./drive/MyDrive/Dataset1(Train).npy")
validation = np.load("./drive/MyDrive/Dataset1(Validation).npy")
test = np.load("./drive/MyDrive/Dataset1(Test).npy")
```

```
# Check the size of each dataset.
print("The shape of the train dataset₩t₩t: %s₩t | The shape of train label₩t₩t: %s"%(X_train.shape, y_train.shape))
print("The shape of the validation dataset₩t: %s₩t | The shape of validation label₩t: %s"%(X_val.shape, y_val.shape))
print("The shape of the test dataset₩t₩t: %s₩t | The shape of test label₩t₩t: %s"%(X_test.shape, y_test.shape))
```

```
The shape of the train dataset          : (600, 3)      | The shape of train label          : (600,)
The shape of the validation dataset     : (200, 3)      | The shape of validation label     : (200,)
The shape of the test dataset           : (200, 3)      | The shape of test label           : (200,)
```



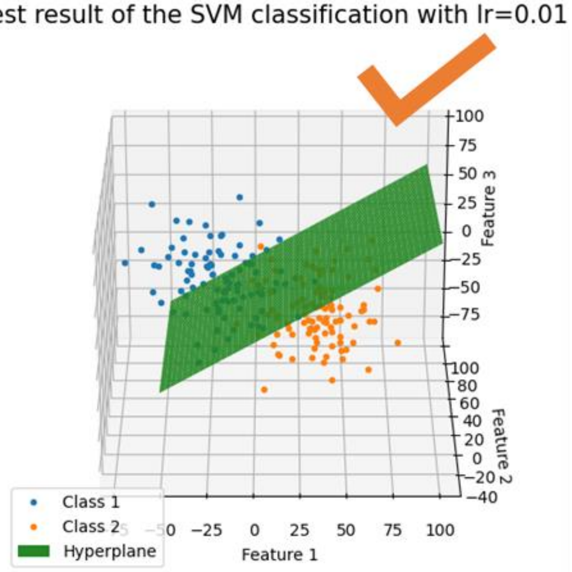
[Step 3] Train and test the soft margin SVM models with different learning rates.

Check the tendency of the objective function values by changing the hyperparameter.

Visualize the result of the SVM models with the different learning rates.

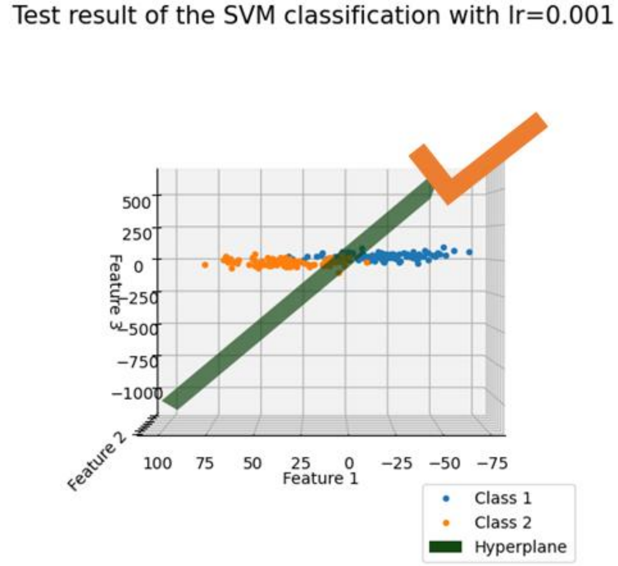
```
----------------------------------------------------------------
Initialize the parameters of our SVM.

- Total Epochs          :  100
- Learning rate         :  0.01
----------------------------------------------------------------
[Epoch : 10/100]                | Cost: 8.9862
[Epoch : 20/100]                | Cost: 0.6664
[Epoch : 30/100]                | Cost: 0.3693
[Epoch : 40/100]                | Cost: 0.3045
[Epoch : 50/100]                | Cost: 0.2633
[Epoch : 60/100]                | Cost: 0.2402
[Epoch : 70/100]                | Cost: 0.2237
[Epoch : 80/100]                | Cost: 0.2162
[Epoch : 90/100]                | Cost: 0.2136
[Epoch : 100/100]               | Cost: 0.2124
----------------------------------------------------------------
Learning is complete.

- W    : [ 0.12018249 -0.03484729 -0.14951882]
- W_0  : [1.45514652]

- The accuracy of the train set          : 0.925
- The accuracy of the validation set     : 0.970
----------------------------------------------------------------
```

- The accuracy of prediction for the test set   : 0.925




```
----------------------------------------------------------------
Initialize the parameters of our SVM.

- Total Epochs          :  100
- Learning rate         :  0.001
----------------------------------------------------------------
[Epoch : 10/100]                | Cost: 45.4743
[Epoch : 20/100]                | Cost: 33.3951
[Epoch : 30/100]                | Cost: 22.8464
[Epoch : 40/100]                | Cost: 14.3614
[Epoch : 50/100]                | Cost: 8.8270
[Epoch : 60/100]                | Cost: 5.6041
[Epoch : 70/100]                | Cost: 3.8263
[Epoch : 80/100]                | Cost: 2.7673
[Epoch : 90/100]                | Cost: 2.0620
[Epoch : 100/100]               | Cost: 1.5677
----------------------------------------------------------------
Learning is complete.

- W    : [ 0.53747534 -0.0628218   0.04898229]
- W_0  : [0.75268473]

- The accuracy of the train set          : 0.837
- The accuracy of the validation set     : 0.880
----------------------------------------------------------------
```

- The accuracy of prediction for the test set   : 0.85



[Step 4] Build the Kernel SVM models based on the dual form

In this assignment, you should use only **Numpy** and **CVXOPT** to build the Kernel SVM models. **Do not use other libraries.** The followings are the skeleton codes. Implement compute_inner_matrix function and the functions inside the Kernels class: polynomial_kernel, and gaussian_kernel.

```
class kernels:
    # Function for computing the kernel matrix
    def __init__(self, poly_c=10, poly_d=2, gamma=0.001):
        self.poly_c = poly_c
        self.poly_d = poly_d
        self.gamma = gamma

    # Function for the polynomial kernel.
    def polynomial_kernel(self, x, y, c=10, d=2):
        # Implement the polynomial kernel function
        ####################### IMPLEMENT HERE #######################
        result = NotImplemented
        raise NotImplementedError
        ##############################################################
        return result

    # Function for the gaussian kernel.
    def gaussian_kernel(self, x, y, gamma=0.001):
        # Implement the linear kernel function.
        # Gamma means 1/(2*sigma**2) where sigma is a free parameter.
        # It can be seen as the standard deviation of the gaussian distribution.
        ####################### IMPLEMENT HERE #######################
        result = NotImplemented
        raise NotImplementedError
        ##############################################################
        return result
```

```
# Function for computing the kernel matrix
def compute_inner_matrix(X_data, kernel):
    # Generate the kernel matrix using the kernel function.
    n = X_data.shape[0]
    inner_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            ####################### IMPLEMENT HERE #######################
            inner_matrix[i,j] = NotImplemented
            raise NotImplementedError
            ##############################################################
    return inner_matrix
```

[Step 4] Build the Kernel SVM models based on the dual form

The followings are the skeleton codes. Implement the functions inside the Kernel SVM class: compute_lagrange, predict, and train.

```python
class Kernel_SVM:
    def __init__(self, kernel, threshold_SV):
        # Set the parameters as the hinstance variables.
        self.kernel = kernel
        self.threshold_SV = threshold_SV

    # Function for computing the lagrangian multiplier vector.
    def compute_lagrange(self, X_data, y_label):
        # Here we will use the cvxopt python library. We will use its matrix objects cvx.matrix
        ####################### IMPLEMENT HERE #######################
        # Compute the kernel matrix using kernel function.
        kernel_matrix = NotImplemented

        # Compute P matrix using y_label vectors and kernel_matrix.
        P = NotImplemented
        # Set ones vector to compute the sum of the alpha vector.
        q = NotImplemented

        # Set the identity metrix for our inequality constraint
        # that all elements of the alpha vector should be greater then equal to zero.
        G = NotImplemented
        h = NotImplemented

        # Set the y_label vector for our equality constraint
        # that tha result of the multiplication of elements of the alpha and y_label vector should be zero.
        A = NotImplemented
        b = NotImplemented
        raise NotImplementedError
        ##############################################################

        # Compute the alpha vector. Refer to the instruction slides.
        alpha = np.array(cvx.solvers.qp(P, q, G, h, A, b)['x']).flatten()

        return alpha
```

```python
# Function for prediction.
def predict(self,x_data):
    # Compute our kernel SVM prediction, and transform it to the binary label.
    # And then append it to the list.
    y_pred = [] # It represents the predicted label of each data point.
    pred = [] # it represents the output value of each data point. (without sign function)
    for i in range(x_data.shape[0]):
        # Compute the bias vector to predict the label of each data point.
        self.bias = 0
        for (alpha, x_, y_) in zip(self.alpha_support, self.support_vectors, self.support_vectors_labels):
            ####################### IMPLEMENT HERE #######################
            # Compute the output value of kernel SVM model.
            self.bias += NotImplemented
            self.bias = NotImplemented
            raise NotImplementedError
            ##############################################################
        ####################### IMPLEMENT HERE #######################
        self.bias /= NotImplemented
        output = NotImplemented
        raise NotImplementedError
        ##############################################################
        for (alpha, x_, y_) in zip(self.alpha_support, self.support_vectors, self.support_vectors_labels):
            ####################### IMPLEMENT HERE #######################
            # Compute the output value of our prediction
            output += NotImplemented
            raise NotImplementedError
            ##############################################################
        # Using sign function, transform our prediction to the binary label.
        y_pred.extend(np.sign(output))
        pred.extend(output)

    return y_pred, pred
```

[Step 4] Build the Kernel SVM models based on the dual form

The followings are the skeleton codes. Implement the functions inside the Kernel SVM class: compute_lagrange, predict, and train.

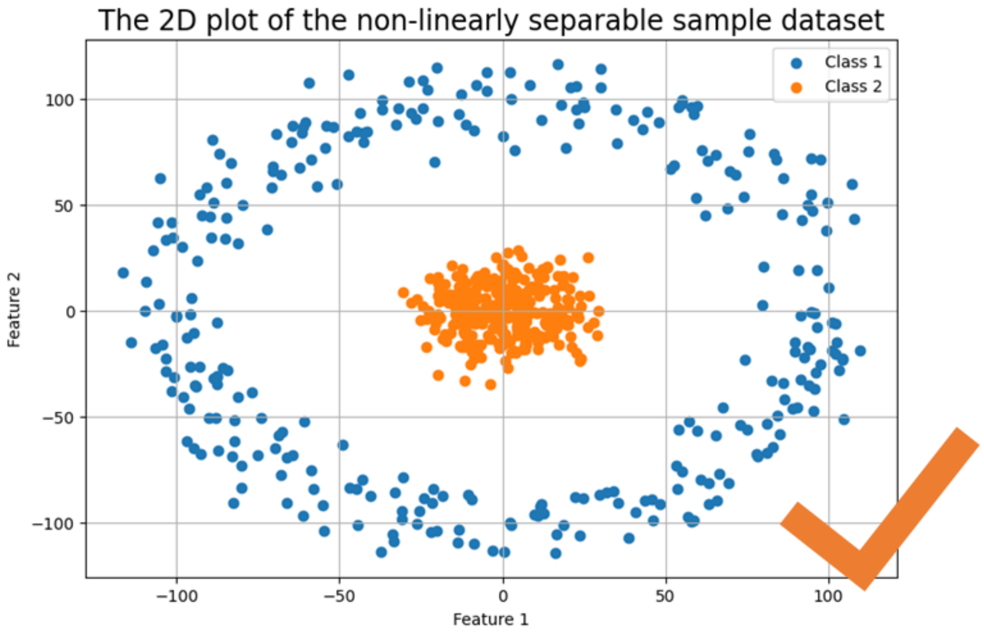
```
# Function for training.
def train(self,X_data, y_label):
    # Compute the alpha vector using compute_lagrange function.
    self.alpha = self.compute_lagrange(X_data, y_label)

    ######################## IMPLEMENT HERE ########################
    # Set the alpha values that are greater than the threshold.
    self.alpha_support = NotImplemented
    # Set the support vectors corresponding to the value of alpha that is greater than the threshold.
    self.support_vectors = NotImplemented
    # Set the labels of the support vectors corresponding to the alpha value that is greater than the threshold.
    self.support_vectors_labels = NotImplemented
    raise NotImplementedError
    ################################################################
```

```
# Function for evaluation.
def accuracy(self, predict, y_label):
    result = np.sum(predict == y_label) / len(predict)
    return result
```

[Step 5] Load the dataset 2

The dataset is the isotropic Gaussian blobs with some noise.

Load the train, validation, and test dataset by executing the provided code.




```
# Load the train, validation, test dataset.
train = np.load("./drive/MyDrive/Dataset2(Train).npy")
validation = np.load("./drive/MyDrive/Dataset2(Validation).npy")
test = np.load("./drive/MyDrive/Dataset2(Test).npy")
```

```
# Check the size of each dataset.
print("The shape of the train dataset₩t₩t: %s₩t | The shape of train label₩t₩t: %s"%(X_train.shape, y_train.shape))
print("The shape of the validation dataset₩t: %s₩t | The shape of validation label₩t: %s"%(X_val.shape, y_val.shape))
print("The shape of the test dataset₩t₩t: %s₩t | The shape of test label₩t₩t: %s"%(X_test.shape, y_test.shape))
```

```
The shape of the train dataset         : (600, 2)     | The shape of train label          : (600,)
The shape of the validation dataset    : (200, 2)     | The shape of validation label     : (200,)
The shape of the test dataset          : (200, 2)     | The shape of test label           : (200,)
```

[Step 6] Train and test the Kernel SVM models with different types of kernels

Check the accuracy of the Kernel SVM model with different types of kernels.

- The performance of the Polynomial Kernel SVM models varies depending on the kernel hyperparameters **c** and **d**.
- The performance of the Gaussian Kernel SVM models varies depending on the kernel hyperparameter **gamma**.

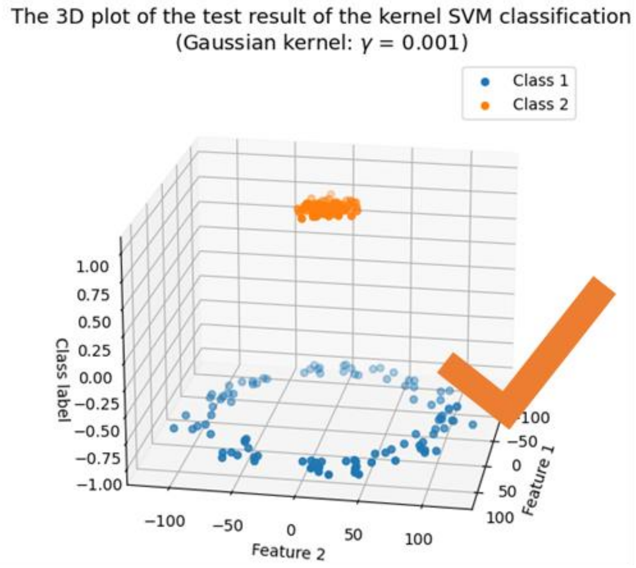
```
# Predict the results using our trained SVM model.
train_predicted, pred = Gaussian_kernel.predict(X_train)
# Compute the accuracy of prediction for the train set
acc = Gaussian_kernel.accuracy(train_predicted, y_train)
print("- The accuracy of prediction for the train set \t\t:",acc)

val_predicted, pred = Gaussian_kernel.predict(X_val)
acc = Gaussian_kernel.accuracy(val_predicted, y_val)
print("- The accuracy of prediction for the validation set \t:",acc)
```

```
- The accuracy of prediction for the train set          : 1.0
- The accuracy of prediction for the validation set     : 1.0
```

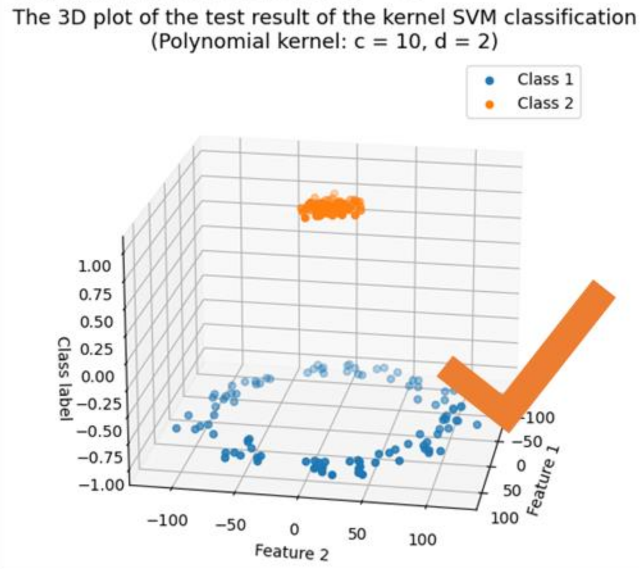
```
# Predict the results of the polynomial kernel SVM model.
train_predicted, pred = Polynomial_kernel.predict(X_train)
# Compute the accuracy of prediction for the train set
acc = Polynomial_kernel.accuracy(train_predicted, y_train)
print("- The accuracy of prediction for the train set \t\t:",acc)

val_predicted, pred = Polynomial_kernel.predict(X_val)
acc = Polynomial_kernel.accuracy(val_predicted, y_val)
print("- The accuracy of prediction for the validation set \t:",acc)
```

```
- The accuracy of prediction for the train set          : 1.0
- The accuracy of prediction for the validation set     : 1.0
```

```
- The accuracy of prediction for the test set     : 1.0
```




```
- The accuracy of prediction for the test set     : 1.0
```



Submission

- Take screenshots of all results for this assignment. You should submit the results with the check mark: ✓
- You should reproduce the result by yourself. You should not submit the pictures provided in this material.
- **Make one PDF document containing all the screenshots.**
- **Deadline: Nov. 10th (Friday) PM 11:59. We do not accept late submissions.**
- If you have any questions, please post them on the KLMS Q&A Board.
- Good luck and have fun!