Lab 4: Support Vector Machine

NOTE: This is a lab project accompanying the following book [MLF] and it should be used together with the book.

[MLF] *H. Jiang*, "Machine Learning Fundamentals: A Concise Introduction", Cambridge University Press, 2021. (bibtex)

The purpose of this lab is to practise and implement an important discriminative model in machine learning, namely support vector machines (SVMs). We first introduce the SVM implementation in *scikit-learn* and show how to use it to fine-tune hyper-parameters for several popular kernel functions in a binary classification task. Next, we will demonstrate how to use a projected gradient descent method described in [MLF] to implement SVMs from scratch, including both linear and nonlinear SVMs. We will compare our own SVM implementation with that of *scikit-learn* in terms of classification performance and running speed in a binary classification task. As a project-end exercise, we will consider how to extend binary SVMs for multiple-class classification tasks based on the *one-vs-one* strategy.

Prerequisites: N/A

I. SVMs from scikit-learn

Example 4.1:

Use the SVM functions from scikit-learn to build a binary classifier to classify two digits ('3' and '8') in the MNIST data set. Fine-tune the hyper-parameters for three important kernel funcions, i.e. linear, polynomial and Gaussian RBF kernels, towards the best possible performance.

In [1]:

```
# download MNIST data from Google drive

!gdown --folder https://drive.google.com/drive/folders/1r20aRjc2iu9O3kN3Xj9jNYY2uMgcERY1 2> /dev/null

# install python_mnist

!pip install python_mnist
```

```
Processing file 1Jf2XqGR7y1fzOZNKLJiom7GmZZUzXhfs t10k-images-idx3-ubyte
Processing file 1qiYu9dW3ZNrlvTFO5fI4qf8Wtr8K-pCu t10k-labels-idx1-ubyte
Processing file 1SnWvBcUETRJ53rEJozFUUo-hOQFPKxjp train-images-idx3-ubyte
Processing file 1kKEIi_pwVHmabByAnwZQsaMgro9XiBFE train-labels-idx1-ubyte
Building directory structure completed
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting python_mnist
  Downloading python_mnist-0.7-py2.py3-none-any.whl (9.6 kB)
Installing collected packages: python_mnist
Successfully installed python_mnist-0.7
```

In [2]:

```
#load MINST images

from mnist import MNIST
import numpy as np

mnist_loader = MNIST('MNIST')
train_data, train_label = mnist_loader.load_training()
test_data, test_label = mnist_loader.load_testing()
train_data = np.array(train_data, dtype='float')/255 # norm to [0,1]
train_label = np.array(train_label, dtype='short')
test_data = np.array(test_data, dtype='float')/255 # norm to [0,1]
test_label = np.array(test_label, dtype='short')

print(train_data.shape, train_label.shape, test_data.shape, test_label.shape)
```

```
(60000, 784) (60000,) (10000, 784) (10000,)
```

In [ ]:

```
# prepare digits '3' and '8' for binary SVMs

digit_train_index = np.logical_or(train_label == 3, train_label == 8)
X_train = train_data[digit_train_index]
y_train = train_label[digit_train_index]
digit_test_index = np.logical_or(test_label == 3, test_label == 8)
X_test = test_data[digit_test_index]
y_test = test_label[digit_test_index]

# normalize all feature vectors to unit-length
X_train = np.transpose (X_train.T / np.sqrt(np.sum(X_train*X_train, axis=1)))
X_test =  np.transpose (X_test.T  / np.sqrt(np.sum(X_test*X_test, axis=1)))

# convert labels: '3' => -1, '8' => +1
CUTOFF = 5 # any number between '3' and '8'
y_train = np.sign(y_train-CUTOFF)
y_test = np.sign(y_test-CUTOFF)
```

In [ ]:

```
# linear SVM: use sk-learn SVC functions
import numpy as np
from sklearn.svm import SVC

for c in [0.01, 0.1, 1, 2, 4, 10]:
  linearSVM = SVC(kernel='linear', C=c)
  linearSVM.fit(X_train,y_train)
  predict = linearSVM.predict(X_train)
  train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
  predict = linearSVM.predict(X_test)
  test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
  print(f'linear SVM (C={c}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
linear SVM (C=0.01): training accuracy=94.54%  test accuracy=95.31%
linear SVM (C=0.1): training accuracy=96.50%  test accuracy=96.98%
linear SVM (C=1): training accuracy=97.39%  test accuracy=96.82%
linear SVM (C=2): training accuracy=97.48%  test accuracy=96.93%
linear SVM (C=4): training accuracy=97.60%  test accuracy=96.93%
linear SVM (C=10): training accuracy=97.79%  test accuracy=97.08%
```

In [ ]:

```
# nonlinear SVM (polynomial kernel): use sk-learn SVC functions
import numpy as np
from sklearn.svm import SVC

for c in [1, 2, 4]:
  for d in [2, 3]:
    polySVM = SVC(kernel='poly', C=c, degree=d)
    polySVM.fit(X_train,y_train)
    predict = polySVM.predict(X_train)
    train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
    predict = polySVM.predict(X_test)
    test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
    print(f'nonlinear polynomial SVM (C={c},d={d}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
nonlinear polynomial SVM (C=1,d=2): training accuracy=99.57%  test accuracy=99.45%
nonlinear polynomial SVM (C=1,d=3): training accuracy=99.84%  test accuracy=99.55%
nonlinear polynomial SVM (C=2,d=2): training accuracy=99.73%  test accuracy=99.50%
nonlinear polynomial SVM (C=2,d=3): training accuracy=99.96%  test accuracy=99.50%
nonlinear polynomial SVM (C=4,d=2): training accuracy=99.92%  test accuracy=99.45%
nonlinear polynomial SVM (C=4,d=3): training accuracy=99.99%  test accuracy=99.50%
```

In [ ]:

```
# nonlinear SVM (Gaussian RBF kernel): use sk-learn SVC functions
import numpy as np
from sklearn.svm import SVC

for c in [1, 2, 10]:
  for g in ['scale', 1, 2]:
    rbfSVM = SVC(kernel='rbf', C=c, gamma=g)
    rbfSVM.fit(X_train,y_train)
    predict = rbfSVM.predict(X_train)
    train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
    predict = rbfSVM.predict(X_test)
    test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
    print(f'nonlinear RBF SVM (C={c},gamma={g}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
nonlinear RBF SVM (C=1,gamma=scale): training accuracy=99.56%  test accuracy=99.40%
nonlinear RBF SVM (C=1,gamma=1): training accuracy=99.41%  test accuracy=99.24%
nonlinear RBF SVM (C=1,gamma=2): training accuracy=99.79%  test accuracy=99.55%
nonlinear RBF SVM (C=2,gamma=scale): training accuracy=99.78%  test accuracy=99.55%
nonlinear RBF SVM (C=2,gamma=1): training accuracy=99.70%  test accuracy=99.55%
nonlinear RBF SVM (C=2,gamma=2): training accuracy=99.96%  test accuracy=99.60%
nonlinear RBF SVM (C=10,gamma=scale): training accuracy=100.00%  test accuracy=99.50%
nonlinear RBF SVM (C=10,gamma=1): training accuracy=99.98%  test accuracy=99.50%
nonlinear RBF SVM (C=10,gamma=2): training accuracy=100.00%  test accuracy=99.60%
```

II. Linear SVMs

Example 4.2:

Implement your own linear SVM function from scratch. Use the projected gradient descent (PGD), i.e. Algorithm 6.5 on page 127, to solve quadratic programming arising from the SVM dual problem. Use your implementation to build a binary classifier to classify two digits ('3' and '8') in the MNIST data set. Compare your own implementation with that of scikit-learn in terms of accuracy and running speed.

The challenge to implement SVMs from scratch is how to implement an efficient optimization method to solve the quadratic programming in the SVM dual problem. The strict projected gradient decent (PGD) algorithm typically converges very slowly in practice. Here we will implement a mini-batch version of PGD. At each step, instead of updating all variables in $\boldsymbol{\alpha}$ as in Algorithm 6.5, we first randomly select a subset of varaibles in $\boldsymbol{\alpha}$ (like a mini-batch in SGD), denoted as $\boldsymbol{\alpha}_s$, and only update all variables in $\boldsymbol{\alpha}_s$ using the same idea of PGD while keeping the other variables in $\boldsymbol{\alpha}$ unchanged. We first compute the gradient w.r.t. $\boldsymbol{\alpha}_s$ as follows:

$$\nabla L(\boldsymbol{\alpha}_s^{(n)}) = \mathbf{Q}_s\, \boldsymbol{\alpha}^{(n)} - \mathbf{1}$$

where $\mathbf{Q}_s$ denotes a smaller matrix where we only keep the rows in $\mathbf{Q}$ corresponding to the selected variables in $\boldsymbol{\alpha}_s$.

Then, we project the above gradient to the subspace $\mathbf{y}_s^\intercal \boldsymbol{\alpha}_s = 0$, where $\mathbf{y}_s$ denotes the targets corresponding to the selected subset $\boldsymbol{\alpha}_s$, in the same way as in Algorithm 6.5:

$$\tilde{\nabla} L(\boldsymbol{\alpha}_s^{(n)}) = \nabla L(\boldsymbol{\alpha}_s^{(n)}) - \frac{\mathbf{y}_s^\intercal \nabla L(\boldsymbol{\alpha}_s^{(n)})}{||\mathbf{y}_s||^2}\mathbf{y}_s$$

If we update $\boldsymbol{\alpha}_s$ using the projected gradient $\tilde{\nabla} L(\boldsymbol{\alpha}_s^{(n)})$ (while keeping the other variables unchanged), we can easily verify that the updated variables $\boldsymbol{\alpha}^{(n+1)}$ remains in the hyper-plane $\mathbf{y}^\intercal \boldsymbol{\alpha} = 0$.

Next, we will compute the maximum step size η_n that is allowed along the projected gradient $\tilde{\nabla} L(\boldsymbol{\alpha}_s^{(n)})$ to ensure the updated $\boldsymbol{\alpha}^{(n+1)}$ still satisfies the box constraint $[0, 1]$. For each variable $\alpha_k \in \boldsymbol{\alpha}_s$, the box bound for the current update depends on the sign of its corresponding projected gradient $\tilde{\nabla} L(\alpha_k^{(n)})$:

$$b_k = \begin{cases} 0 & \text{if } \tilde{\nabla} L(\alpha_k^{(n)}) > 0 \\ C & \text{if } \tilde{\nabla} L(\alpha_k^{(n)}) < 0 \end{cases}$$

Thus, the maximum allowed step size is computed as:

$$\eta_n = \min_{\alpha_k \in \boldsymbol{\alpha}_s} \frac{\left|\alpha_k^{(n)} - b_k\right|}{\left|\tilde{\nabla} L(\alpha_k^{(n)})\right| + \epsilon}$$

where $\epsilon = 10^{-3}$ is added for numerical stability.

In [ ]:

```
# solve linear SVMs using projected gradient descent (PGD)
import numpy as np

class mySVM1():
  def __init__(self, kernel='linear', optimizer='pgd', debug=0, threshold=0.001, \
               lr=1.0, max_epochs=10, batch_size=2, C=1):
    self.kernel = kernel         # kernel type
    self.optimizer = optimizer  # which optimizer is used to solve quadratic programming
    self.lr = lr                 # max learning rate in PGD
    self.max_epochs = max_epochs   # max epochs in PGD
    self.batch_size = batch_size   # size of each subset in PGD
    self.debug = debug             # whether print debugging info
    self.threshold = threshold     # threshold to filter out support vectors

    self.C = C     # C for the soft-margin term

  # Linear Kernel Function
  # X[N,d]: training samples;  Y[M,d]: other training samples
  # return Q[N,N]: linear kernel matrix between X and Y
  def Kernel(self, X, Y):
    if (self.kernel == 'linear'):
      K = X @ Y.T

    return K

  # construct matrix Q from any kernel function for dual SVM optimization
  def QuadraticMatrix(self, X, y):
    Q = np.outer(y, y) * self.Kernel(X, X)
    return Q

  # use projected gradient descent to solve quadratic program
  # refer to Algorithm 6.5 on page 127
  # Q[N,N]: quadratic matrix;  y[N]: training labels (+1 or -1)
  def PGD(self, Q, y):
    N = Q.shape[0]   # num of training samples
    alpha = np.zeros(N)
    prev_L = 0.0

    for epoch in range(self.max_epochs):
      indices = np.random.permutation(N)  #randomly shuffle data indices
      for batch_start in range(0, N, self.batch_size):
        idx = indices[batch_start:batch_start + self.batch_size] # indices of the selected subset
        alpha_s = alpha[idx]
        y_s = y[idx]

        grad_s = Q[idx,:] @ alpha - np.ones(idx.shape[0])
        proj_grad_s = grad_s - np.dot(y_s,grad_s)/np.dot(y_s, y_s)*y_s

        bound = np.zeros(idx.shape[0])
        bound[proj_grad_s < 0] = self.C

        eta = np.min(np.abs(alpha_s-bound)/(np.abs(proj_grad_s)+0.001))

        alpha[idx] -= min(eta, self.lr) * proj_grad_s

      L = 0.5 * alpha.T @ Q @ alpha - np.sum(alpha) # objectibve function
      if (L > prev_L):
        if (self.debug>0):
          print('Early stopping at epoch={epoch}!')
        break

      if (self.debug>1):
        print(f'[PGD optimizer] epoch = {epoch}: L = {L:.5f}  (# of support vectors = {(alpha>self.threshold).sum()})')
        print(f'                 alpha: max={np.max(alpha)} min={np.min(alpha)} orthogonal constraint={np.dot(alpha,y):.2f}')

      prev_L = L

    return alpha

  # train SVM from training samples
  # X[N,d]: input features;  y[N]: output labels (+1 or -1)
  def fit(self, X, y):
    if(self.kernel != 'linear'):
      print("Error: only linear kernel is supported!")
      return

    Q = self.QuadraticMatrix(X, y)

    alpha = self.PGD(Q, y)

    #save support vectors (pruning all data with alpha==0)
    self.X_SVs = X[alpha>self.threshold]
    self.y_SVs = y[alpha>self.threshold]
    self.alpha_SVs = alpha[alpha>self.threshold]

    # compute weight vector for linear SVMs (refer to the formula on page 120)
    if(self.kernel == 'linear'):
      self.w = (self.y_SVs * self.alpha_SVs) @ self.X_SVs

    # estimate b
    idx = np.nonzero(np.logical_and(self.alpha_SVs>self.threshold,self.alpha_SVs<self.C-self.threshold))
    if(len(idx) == 0):
      idx = np.nonzero(self.alpha_SVs>self.threshold)
    # refer to the formula on page 125 (above Figure 6.11)
    b = self.y_SVs[idx] - (self.y_SVs * self.alpha_SVs) @ self.Kernel(self.X_SVs, self.X_SVs[idx])
    self.b = np.median(b)

    return

  # use SVM from prediction
  # X[N,d]: input features
  def predict(self, X):
    if(self.kernel != 'linear'):
      print("Error: only linear kernel is supported!")
      return

    y = X @ self.w + self.b

    return np.sign(y)
```

In [ ]:

```
for c in [0.1, 1, 2, 4, 10]:
  svm = mySVM1(max_epochs=10, lr=2.0, C=c, kernel='linear')
  svm.fit(X_train,y_train)

  predict = svm.predict(X_train)
  train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
  predict = svm.predict(X_test)
  test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
  print(f'MY linear SVM (C={c}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
MY linear SVM (C=0.1): training accuracy=96.53%  test accuracy=96.82%
MY linear SVM (C=1): training accuracy=97.10%  test accuracy=96.67%
MY linear SVM (C=2): training accuracy=96.96%  test accuracy=96.98%
MY linear SVM (C=4): training accuracy=96.85%  test accuracy=96.67%
MY linear SVM (C=10): training accuracy=96.35%  test accuracy=96.47%
```

When we compare the above results with those generated by SVC from *sciki-learn*, we can see that our linear SVM implementation using PGD has achieved comparable performance to *sciki-learn*'s. For example, when we compare test classification accuracy, we can see that our implementation obtains 96.98% at C=2 while scikit-learn's SVC gets 97.08% at C=10. (Note: the results vary slightly between different runs due to the randomness in the PGD optimization algorithm.)

Next, we compare the training time between our PGD implementation and sciki-learn's SVC. From the following results, we can see that their running times are pretty close as well.

In [ ]:

```
from sklearn.svm import SVC

c=1

linearSVM = SVC(kernel='linear', C=c)
%timeit linearSVM.fit(X_train,y_train)

svm = mySVM1(max_epochs=10, lr=2.0, C=c, kernel='linear')
%timeit svm.fit(X_train,y_train)
```

```
CPU times: user 10.5 s, sys: 21.6 ms, total: 10.5 s
Wall time: 10.5 s
CPU times: user 12.2 s, sys: 1.12 s, total: 13.3 s
Wall time: 8.47 s
```

III. Nonlinear SVMs

Example 4.3:

Add two more kernel functions (i.e. polynomial and Gaussian RBF kernels) to extend the above SVM implementation in Example 4.2 for nonlinear SVMs. Use your nonlinear SVM implementation to build a binary classifier to classify two digits ('3' and '8') in the MNIST data set. Compare your own implementation with that of scikit-learn for these two nonlinear kernel functions in terms of classification accuracy and running speed.

First of all, let us consider how to use vectorization to compute various kernel matrices for two sets of feature vectors, i.e. $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and $\{\mathbf{y}_1, \cdots, \mathbf{y}_M\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{y}_j \in \mathbb{R}^d$.

As the way on page 112, if we pack all feature vectors $\mathbf{x}_i$ from the first set row by row as a matrix $\mathbf{X} \in \mathbb{R}^{N\times d}$, and all feature vectors $\mathbf{y}_j$ from the second set row by row as a matrix $\mathbf{Y} \in \mathbb{R}^{M\times d}$, we can conveniently compute the kernel matrices for different kernel functions as follows:

(1) Linear kernel $\Phi(\mathbf{x}_i, \mathbf{y}_j) = \mathbf{x}_i^\intercal \mathbf{y}_j$:

$$\mathbf{K} = \Big[\Phi(\mathbf{x}_i, \mathbf{y}_j)\Big]_{N\times M} = \begin{bmatrix} \mathbf{x}_1^\intercal \mathbf{y}_1 & \cdots & \mathbf{x}_1^\intercal \mathbf{y}_M \\ \vdots & \mathbf{x}_i^\intercal \mathbf{y}_j & \vdots \\ \mathbf{x}_N^\intercal \mathbf{y}_1 & \cdots & \mathbf{x}_N^\intercal \mathbf{y}_M \end{bmatrix}_{N\times M} = \mathbf{X}\mathbf{Y}^\intercal$$

(2) Polynomial kernel $\Phi(\mathbf{x}_i, \mathbf{y}_j) = (\mathbf{x}_i^\intercal \mathbf{y}_j + 1)^p$:

$$\mathbf{K} = \Big[\Phi(\mathbf{x}_i, \mathbf{y}_j)\Big]_{N\times M} = \begin{bmatrix} (\mathbf{x}_1^\intercal \mathbf{y}_1+1)^p & \cdots & (\mathbf{x}_1^\intercal \mathbf{y}_M+1)^p \\ \vdots & (\mathbf{x}_i^\intercal \mathbf{y}_j+1)^p & \vdots \\ (\mathbf{x}_N^\intercal \mathbf{y}_1+1)^p & \cdots & (\mathbf{x}_N^\intercal \mathbf{y}_M+1)^p \end{bmatrix}_{N\times M} = \text{power}\Big(\mathbf{X}\mathbf{Y}^\intercal + 1, p\Big)$$

(3) Gaussian RBF kernel $\Phi(\mathbf{x}_i, \mathbf{y}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{y}_j||^2)$:

We can show that

$$||\mathbf{x}_i - \mathbf{y}_j||^2 = \big(\mathbf{x}_i - \mathbf{y}_j\big)^\intercal\big(\mathbf{x}_i - \mathbf{y}_j\big) = \mathbf{x}_i^\intercal\mathbf{x}_i + \mathbf{y}_j^\intercal\mathbf{y}_j - 2\mathbf{x}_i^\intercal\mathbf{y}_j$$

We first compute two diagonal vectors as follows:

$$\mathbf{a} = \begin{bmatrix} \mathbf{x}_1^\intercal\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N^\intercal\mathbf{x}_N \end{bmatrix}_{N\times 1} = \text{diag}\big(\mathbf{X}\mathbf{X}^\intercal\big)$$

$$\mathbf{b} = \begin{bmatrix} \mathbf{y}_1^\intercal\mathbf{y}_1 \\ \vdots \\ \mathbf{y}_N^\intercal\mathbf{y}_N \end{bmatrix}_{M\times 1} = \text{diag}\big(\mathbf{Y}\mathbf{Y}^\intercal\big)$$

Finally, we can verify that

$$\mathbf{K} = \Big[\Phi(\mathbf{x}_i, \mathbf{y}_j)\Big]_{N\times M} = \begin{bmatrix} \exp(-\gamma||\mathbf{x}_1 - \mathbf{y}_1||^2) & \cdots & \exp(-\gamma||\mathbf{x}_1 - \mathbf{y}_M||^2) \\ \vdots & \exp(-\gamma||\mathbf{x}_i - \mathbf{y}_i||^2) & \vdots \\ \exp(-\gamma||\mathbf{x}_N - \mathbf{y}_1||^2) & \cdots & \exp(-\gamma||\mathbf{x}_N - \mathbf{y}_M||^2) \end{bmatrix}_{N\times M}$$

$$= \exp\Big(-\gamma\big(\mathbf{a}\,\mathbf{1}_M^\intercal + \mathbf{1}_N\,\mathbf{b}^\intercal - 2\,\mathbf{X}\mathbf{Y}^\intercal\big)\Big)$$

where $\mathbf{1}_m$ denotes an m-dimension vector consisting of all 1's.

In [ ]:

```
# extend for nonlinear SVMs by adding polynomial and RBF kernel functions
#
import numpy as np

class mySVM2():
  def __init__(self, kernel='linear', optimizer='pgd', debug=0, threshold=0.001, \
               lr=1.0, max_epochs=20, batch_size=2, C=1, order=3, gamma=1.0):
    self.kernel = kernel           # kernel type
    self.optimizer = optimizer     # which optimizer is used to solve quadratic programming
    self.lr = lr                   # max learning rate in PGD
    self.max_epochs = max_epochs   # max epochs in PGD
    self.batch_size = batch_size   # size of each subset in PGD
    self.debug = debug             # whether print debugging info
    self.threshold = threshold     # threshold to filter out support vectors

    self.C = C                     # C for the soft-margin term
    self.order = order             # power order for polynomial kernel
    self.gamma = gamma             # gamma for Gaussian RBF kernel

  # Kernel Function
  # X[N,d]: training samples;  Y[M,d]: other training samples
  # return Q[N,N]: linear kernel matrix between X and Y
  def Kernel(self, X, Y):
    if (self.kernel == 'linear'):
      K = X @ Y.T
    elif (self.kernel == 'poly'):
      K = np.power(X @ Y.T +1, self.order)
    elif (self.kernel == 'rbf'):
      d1 = np.sum(X*X, axis=1)
      d2 = np.sum(Y*Y, axis=1)
      K = np.outer(d1, np.ones(Y.shape[0])) + np.outer(np.ones(X.shape[0]), d2) \
          - 2 * X @ Y.T
      K = np.exp(-self.gamma * K)

    return K

  # construct matrix Q from any kernel function for dual SVM optimization
  def QuadraticMatrix(self, X, y):
    Q = np.outer(y, y) * self.Kernel(X, X)
    return Q

  # use projected gradient descent to solve quadratic program
  # refer to Algorithm 6.5 on page 127
  # Q[N,N]: quadratic matrix;  y[N]: training labels (+1 or -1)
  def PGD(self, Q, y):
    N = Q.shape[0]   # num of training samples
    alpha = np.zeros(N)
    prev_L = 0.0

    for epoch in range(self.max_epochs):
      indices = np.random.permutation(N)  #randomly shuffle data indices
      for batch_start in range(0, N, self.batch_size):
        idx = indices[batch_start:batch_start + self.batch_size] # indices of the current subset
        alpha_s = alpha[idx]
        y_s = y[idx]

        grad_s = Q[idx,:] @ alpha - np.ones(idx.shape[0])
        proj_grad_s = grad_s - np.dot(y_s,grad_s)/np.dot(y_s, y_s)*y_s

        bound = np.zeros(idx.shape[0])
        bound[proj_grad_s < 0] = self.C

        eta = np.min(np.abs(alpha_s-bound)/(np.abs(proj_grad_s)+0.001))

        alpha[idx] -= min(eta, self.lr) * proj_grad_s

      L = 0.5 * alpha.T @ Q @ alpha - np.sum(alpha) # objectibve function
      if (L > prev_L):
        if (self.debug>0):
          print(f'Early stopping at epoch={epoch}! (reduce learning rate lr)')
        break

      if (self.debug>1):
        print(f'[PGD optimizer] epoch = {epoch}: L = {L:.5f}  (# of support vectors = {(alpha>self.threshold).sum()})')
        print(f'                 alpha: max={np.max(alpha)} min={np.min(alpha)} orthogonal constraint={np.dot(alpha,y):.2f}')

      prev_L = L

    return alpha

  # train SVM from training samples
  # X[N,d]: input features;  y[N]: output labels (+1 or -1)
  def fit(self, X, y):
    if(self.kernel != 'linear' and self.kernel != 'poly' and self.kernel != 'rbf'):
      print("Error: only linear/poly/rbf kernel is supported!")
      return

    Q = self.QuadraticMatrix(X, y)

    alpha = self.PGD(Q, y)

    #save support vectors (pruning all data with alpha==0)
    self.X_SVs = X[alpha>self.threshold]
    self.y_SVs = y[alpha>self.threshold]
    self.alpha_SVs = alpha[alpha>self.threshold]

    if(self.kernel == 'linear'):
      self.w = (self.y_SVs * self.alpha_SVs) @ self.X_SVs

    # estimate b
    idx = np.nonzero(np.logical_and(self.alpha_SVs>self.threshold,self.alpha_SVs<self.C-self.threshold))
    if(len(idx) == 0):
      idx = np.nonzero(self.alpha_SVs>self.threshold)
    # refer to the formula on page 125 (above Figure 6.11)
    b = self.y_SVs[idx] - (self.y_SVs * self.alpha_SVs) @ self.Kernel(self.X_SVs, self.X_SVs[idx])
    self.b = np.median(b)

    return

  # use SVM from prediction
  # X[N,d]: input features
  def predict(self, X):
    if(self.kernel != 'linear' and self.kernel != 'poly' and self.kernel != 'rbf'):
      print("Error: only linear/poly/rbf kernel is supported!")
      return

    if(self.kernel == 'linear'):
      y = X @ self.w + self.b
    else:
      y = (self.y_SVs * self.alpha_SVs) @ self.Kernel(self.X_SVs, X) + self.b

    return np.sign(y)
```

In [ ]:

```
c = 2
svm = mySVM2(max_epochs=20, lr=1.0, C=c, kernel='linear', debug=0)
svm.fit(X_train,y_train)

predict = svm.predict(X_train)
train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
predict = svm.predict(X_test)
test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
print(f'MY linear SVM (C={c}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
MY linear SVM (C=2): training accuracy=97.31%  test accuracy=96.98%
```

In [ ]:

```
c = 2
d = 3
svm = mySVM2(max_epochs=20, lr=0.1, C=c, kernel='poly', order=d, debug=0)
svm.fit(X_train,y_train)

predict = svm.predict(X_train)
train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
predict = svm.predict(X_test)
test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
print(f'MY poly SVM (C={c}, d={d}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
MY poly SVM (C=2, d=3): training accuracy=99.77%  test accuracy=99.50%
```

In [ ]:

```
c = 2
g = 2.0
svm = mySVM2(max_epochs=20, lr=1.0, C=c, kernel='rbf', gamma=g, debug=0)
svm.fit(X_train,y_train)

predict = svm.predict(X_train)
train_acc = np.count_nonzero(np.equal(predict,y_train))/y_train.size
predict = svm.predict(X_test)
test_acc = np.count_nonzero(np.equal(predict,y_test))/y_test.size
print(f'MY RBF SVM (C={c}, gamma={g}): training accuracy={100*train_acc:.2f}%  test accuracy={100*test_acc:.2f}%')
```

```
MY RBF SVM (C=2, gamma=2.0): training accuracy=99.97%  test accuracy=99.60%
```

From the above results, we can see that our SVM implmentation delivers comparable classification accuracies with the scikit-learn SVC functions for all three kernel functions.

Exercises

Problem 4.1:

Use the *one-versus-one* strategy discussed in Section 6.5.5 (page 127) to extend the above binary SVMs to deal with a pattern classification task involving any number of classes. Use your extension to build a 10-class classifier to recognize all 10 digits in the MNIST data set. Compare three different kernels and fine-tune their hyper-parameters towards the best possible accuracy in each case.

Problem 4.2:

Refer to Q6.12 (page 131), derive the closed-form solution to update any two varaibles in $\boldsymbol{\alpha}$ if we keep all other variables in $\boldsymbol{\alpha}$ constant in the quadratic programming problem (SVM4 on page 122). Based on this result, implement the famous *sequential minimization optimization* (SMO) method as another optimizer option for our SVM implementation (besides PGD). Compare SMO with PGD in terms of their convergence speeds and final classification accuracies.