```python
import numpy as np
from numpy import linalg
import cvxopt
import cvxopt.solvers
import sys
import pandas as pd
cvxopt.solvers.options['show_progress'] = False

# Reads the data from CSV files, converts it into Dataframe and returns x and y dataframes
def getDataframe(filePath):
    dataframe = pd.read_csv(filePath)
    y = dataframe['y']
    x = dataframe.drop('y', axis=1)
    y = y*2 -1.0
    return x.to_numpy(), y.to_numpy()

def compute_accuracy(predicted_y, y):
    acc = 100.0
    acc = np.sum(predicted_y == y)/predicted_y.shape[0]
    return acc

def gaussian_kernel_point(x, y, sigma=5.0):
    return np.exp(-linalg.norm(x-y)**2 / (2 * (sigma ** 2)))

def linear_kernel(X, Y=None):
    Y = X if Y is None else Y
    m = X.shape[0]
    n = Y.shape[0]
    assert X.shape[1] == Y.shape[1]
    kernel_matrix = np.zeros((m, n))
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    for i in range(m):
        for j in range(n):
            kernel_matrix[i,j] = np.dot(X[i], Y[j])
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    return kernel_matrix

def polynomial_kernel(X, Y=None, degree=3):
    Y = X if Y is None else Y
    m = X.shape[0]
    n = Y.shape[0]
    assert X.shape[1] == Y.shape[1]
    kernel_matrix = np.zeros((m, n))
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    for i in range(m):
        for j in range(n):
            kernel_matrix[i,j] = (np.dot(X[i], Y[j]) + 1) ** degree
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    return kernel_matrix

```

```python
    Y = X if Y is None else Y
    m = X.shape[0]
    n = Y.shape[0]
    assert X.shape[1] == Y.shape[1]
    kernel_matrix = np.zeros((m, n))
    #=======================#
    # STRART YOUR CODE HERE  #
    #=======================#
    for i in range(m):
        for j in range(n):
            kernel_matrix[i,j] = gaussian_kernel_point(X[i], Y[j])
    #=======================#
    #   END YOUR CODE HERE   #
    #=======================#
    return kernel_matrix


# Bonus question: vectorized implementation of Gaussian kernel
# If you decide to do the bonus question, comment the gaussian_kernel function above,
# then implement and uncomment this one.
# def gaussian_kernel(X, Y=None, sigma=5.0):
#     return

class SVM(object):
    def __init__(self):
        self.train_x = pd.DataFrame()
        self.train_y = pd.DataFrame()
        self.test_x = pd.DataFrame()
        self.test_y = pd.DataFrame()
        self.kernel_name = None
        self.kernel = None

    def load_data(self, train_file, test_file):
        self.train_x, self.train_y = getDataframe(train_file)
        self.test_x, self.test_y = getDataframe(test_file)


    def train(self, kernel_name='linear_kernel', C=None):
        self.kernel_name = kernel_name
        if(kernel_name == 'linear_kernel'):
            self.kernel = linear_kernel
        elif(kernel_name == 'polynomial_kernel'):
            self.kernel = polynomial_kernel
        elif(kernel_name == 'gaussian_kernel'):
            self.kernel = gaussian_kernel
        else:
            raise ValueError("kernel not recognized")

        self.C = C
        if self.C is not None:
            self.C = float(self.C)

        self.fit(self.train_x, self.train_y)

    # predict labels for test dataset
    def predict(self, X):
        if self.w is not None: ## linear case
            n = X.shape[0]
```

```
            #========================#
            # STRART YOUR CODE HERE  #
            #========================#
            predicted_y = np.dot(X, self.w) + self.b
            #========================#
            #   END YOUR CODE HERE   #
            #========================#
            return predicted_y

        else: ## non-linear case
            n = X.shape[0]
            predicted_y = np.zeros(n)
            #========================#
            # STRART YOUR CODE HERE  #
            #========================#
            kernel_matrix = self.kernel(X, self.sv)
            predicted_y = np.dot(kernel_matrix * self.sv_y, self.a)  + self.b

            #========================#
            #   END YOUR CODE HERE   #
            #========================#
            return predicted_y

    #================================================#
    # Please DON'T change any code below this line!  #
    #================================================#
    def fit(self, X, y):
        n_samples, n_features = X.shape
        # Kernel matrix
        K = self.kernel(X)

        # dealing with dual form quadratic optimization
        P = cvxopt.matrix(np.outer(y,y) * K)
        q = cvxopt.matrix(np.ones(n_samples) * -1)
        A = cvxopt.matrix(y, (1,n_samples),'d')
        b = cvxopt.matrix(0.0)

        if self.C is None:
            G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
            h = cvxopt.matrix(np.zeros(n_samples))
        else:
            tmp1 = np.diag(np.ones(n_samples) * -1)
            tmp2 = np.identity(n_samples)
            G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
            tmp1 = np.zeros(n_samples)
            tmp2 = np.ones(n_samples) * self.C
            h = cvxopt.matrix(np.hstack((tmp1, tmp2)))

        # solve QP problem
        solution = cvxopt.solvers.qp(P, q, G, h, A, b)
        # Lagrange multipliers
        a = np.ravel(solution['x'])

        # Support vectors have non zero lagrange multipliers
        sv = a > 1e-5
        ind = np.arange(len(a))[sv]
        self.a = a[sv]
        self.sv = X[sv]
        self.sv_y = y[sv]
```

```python
        print("%d support vectors out of %d points" % (len(self.a), n_samples))

        # Intercept via average calculating b over support vectors
        self.b = 0
        for n in range(len(self.a)):
            self.b += self.sv_y[n]
            self.b -= np.sum(self.a * self.sv_y * K[ind[n],sv])
        self.b /= len(self.a)

        # Weight vector
        if self.kernel_name == 'linear_kernel':
            self.w = np.zeros(n_features)
            for n in range(len(self.a)):
                self.w += self.a[n] * self.sv_y[n] * self.sv[n]
        else:
            self.w = None


    def test(self):
        accuracy = self.classify(self.test_x, self.test_y)
        return accuracy

    def classify(self, X, y):
        predicted_y = np.sign(self.predict(X))
        accuracy = compute_accuracy(predicted_y, y)
        return accuracy
```