HW2

Graded

Student

Ahamada Abdoul-Hakim

Total Points

5.75 / 7 pts

Question 1

Exercise 1 — **1.75** / 2 pts

1.1 1 — **1** / 1 pt

✔ **– 0 pts** Correct

1.2 2 — **0.75** / 1 pt

– 0 pts Correct

✔ **– 0.25 pts** Incorrect separating line for gaussians or hyper-parameter choice

– 0.5 pts Incorrect implementation

Question 2

Exercise 2 — **2** / 2 pts

2.1 1 — **1** / 1 pt

✔ **– 0 pts** Correct

– 1 pt Wrong or incomplete

– 0.5 pts No closed-form expression for alpha

– 0 pts Click here to replace this description.

2.2 2 — **1** / 1 pt

✔ **– 0 pts** Correct

– 1 pt wrong/no answer

– 0.5 pts error in the implementation or hyper-parameter choice

Question 3

Exercise 3 — **2** / 3 pts

3.1 1 — **1** / 1 pt

✔ **– 0 pts** Correct

– 1 pt wrong/no answer

– 0.5 pts incomplete

3.2 2 — **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Wrong/incomplete

– 1 pt No answer

3.3 3 — **0** / 1 pt

– 0 pts Correct

✔ **– 1 pt** Incorrect

Question assigned to the following page: 1.1

Homework 2
Machine Learning with Kernel Methods

Abdoul-Hakim Ahamada

March 6, 2024

Contents

1 Support Vector Classifier

Consider the convex optimization problem:

$$\min_{f,b,\xi_i} \frac{1}{2}\|f\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{s.t.} \quad y_i(f(x_i)+b) \geq 1-\xi_i \quad \text{and} \quad \xi_i \geq 0$$

The Lagrangian of this problem is given by:

$$L(f,b,\xi,\alpha,\mu) = \frac{1}{2}\|f\|^2 + C\sum_{i=1}^{n}\xi_i + \sum_{i=1}^{n}\alpha_i(1-\xi_i - y_i(f(x_i)+b)) - \sum_{i=1}^{n}\mu_i\xi_i$$

By applying the optimality condition to the Lagrangian, we derive the dual problem:

$$\min_{\alpha} \frac{1}{2}\left\|\sum_{i=1}^{n}\alpha_i y_i k_{x_i}\right\|^2 - \sum_{i=1}^{n}\alpha_i \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C \quad \text{and} \quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

Solving this problem yields the expression for f:

$$f = \sum_{i=1}^{n}\alpha_i y_i k_{x_i}$$

Using the Karush-Kuhn-Tucker (KKT) conditions, we can identify a condition characterizing the support vector points x_i lying on the margin of the separating hyperplane:

$$0 < \alpha_i < C$$

Question assigned to the following page: 1.2

```python
class RBF:
    def __init__(self, sigma=1.):
        self.sigma = sigma  ## the variance of the kernel

    def kernel(self,X,Y):
        ## Input vectors X and Y of shape Nxd and Mxd
        X2 = np.sum(X**2, axis=1)[:, np.newaxis]
        Y2 = np.sum(Y**2, axis=1)
        XY = np.dot(X, Y.T)
        squared_distances = X2 + Y2 - 2*XY
        return np.exp(-squared_distances / (2 * self.sigma**2)) ## Matrix of shape NxM
```

```python
class Linear:
    def kernel(self,X,Y):
        ## Input vectors X and Y of shape Nxd and Mxd
        return np.dot(X, Y.T) ## Matrix of shape NxM
```

Figure 1: RBF and Linear Kernels

```python
class KernelSVC:
    def __init__(self, C, kernel, epsilon=1e-3):
        self.type = 'non-linear'
        self.C = C
        self.kernel = kernel
        self.alpha = None
        self.support = None
        self.epsilon = epsilon
        self.norm_f = None

    def fit(self, X, y):
        N = len(y)  # Number of samples
        K = self.kernel(X, X)  # Compute kernel matrix

        # Lagrange dual problem
        def loss(alpha):
            return 0.5 * np.dot(alpha, np.dot(alpha * y, K * y)) - np.sum(alpha)

        # Partial derivative of ld on alpha
        def grad_loss(alpha):
            return np.dot(K * y, alpha * y) - np.ones(N)

        # Constraints on alpha
        fun_eq = lambda alpha: np.dot(alpha, y)  # Equality constraint
        jac_eq = lambda alpha: y  # Jacobian w.r.t alpha of the equality constraint
        fun_ineq = lambda alpha: np.hstack((alpha, self.C - alpha))  # Inequality constraint
        jac_ineq = lambda alpha: np.vstack((np.eye(N), -np.eye(N)))  # Jacobian w.r.t alpha of the inequality constraint

        constraints = ({'type': 'eq', 'fun': fun_eq, 'jac': jac_eq},
                       {'type': 'ineq', 'fun': fun_ineq, 'jac': jac_ineq})

        # Minimize the loss function to find optimal alpha
        optRes = optimize.minimize(fun=lambda alpha: loss(alpha),
                                   x0=np.ones(N),
                                   method='SLSQP',
                                   jac=lambda alpha: grad_loss(alpha),
                                   constraints=constraints)
        self.alpha = optRes.x  # Optimal Lagrange multipliers

        # Calculate support vectors and bias
        self.support_indices = np.where((self.alpha > self.epsilon) & (self.alpha < self.C - self.epsilon))[0]
        self.margin_points = X[self.support_indices]  # Support vectors
        self.support_labels = y[self.support_indices]  # Labels of support vectors

        alpha_y = self.alpha * y
        self.b = np.mean(y[self.support_indices] - np.dot(K[self.support_indices], alpha_y))  # Bias term

        # Compute the RKHS norm of the function f
        self.norm_f = np.sqrt(np.dot(alpha_y.T, np.dot(K, alpha_y)))

    def separating_function(self, x):
        Kmp = self.kernel(self.margin_points, x)  # Compute kernel matrix
        alpha_y = self.alpha[self.support_indices] * self.support_labels  # Lagrange multipliers multiplied by labels
        return np.dot(Kmp.T, alpha_y)  # Return the separating function value

    def predict(self, X):
        d = self.separating_function(X)  # Compute separating function
        return 2 * (d + self.b > 0) - 1  # Apply threshold to predict labels (-1 or 1)

    def score(self, X, y_true):
        y_pred = self.predict(X)  # Predict labels
        correct_predictions = (y_pred == y_true).sum()  # Count correct predictions
        accuracy = correct_predictions / len(y_true)  # Calculate accuracy
        return accuracy  # Return accuracy
```

Figure 2: KernelSVC

Question assigned to the following page: 1.2

```python
from tqdm.notebook import tqdm

def best_parameters(train_dataset, kernel=None, C_values=np.logspace(-5, 1, 50), sigma_values=np.logspace(-2, 0, 5)):
    best_C = None
    best_accuracy = 0

    if kernel is None:
        parameter_values = [(sigma, C) for sigma in sigma_values for C in C_values]
        parameter_desc = 'Sigma and C'
    else:
        parameter_values = C_values
        parameter_desc = 'C'

    for parameters in tqdm(parameter_values, desc=parameter_desc):
        if kernel is None:
            sigma, C = parameters
            kernel_func = RBF(sigma).kernel
        else:
            C = parameters
            kernel_func = kernel

        model = KernelSVC(C=C, kernel=kernel_func, epsilon=1e-14)
        model.fit(train_dataset['x'], train_dataset['y'])
        predictions = model.predict(train_dataset['x'])
        correct_predictions = np.sum(predictions == train_dataset['y'])
        total_samples = len(train_dataset['y'])
        accuracy = correct_predictions / total_samples

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_C = C
            if kernel is None:
                best_sigma = sigma

    if kernel is None:
        print("Best value of sigma:", best_sigma)
    print("Best value of C:", best_C)
    print("Best accuracy:", best_accuracy)

    if kernel is None:
        return best_sigma, best_C
    else:
        return best_C
```

Figure 3: Best parameters

Dataset	C	Accuracy
1	0.020235896477251575	1.0
2	0.015264179671752334	0.91
3	1.0481131341546852	0.59

Table 1: Linear Kernel

Dataset	C	Accuracy
1	1.389495494373136	1.0
2	1.389495494373136	1.0
3	0.04714866363457394	1.0

Table 2: RBF Kernel ($\sigma = 0.01$)

Question assigned to the following page: 1.2

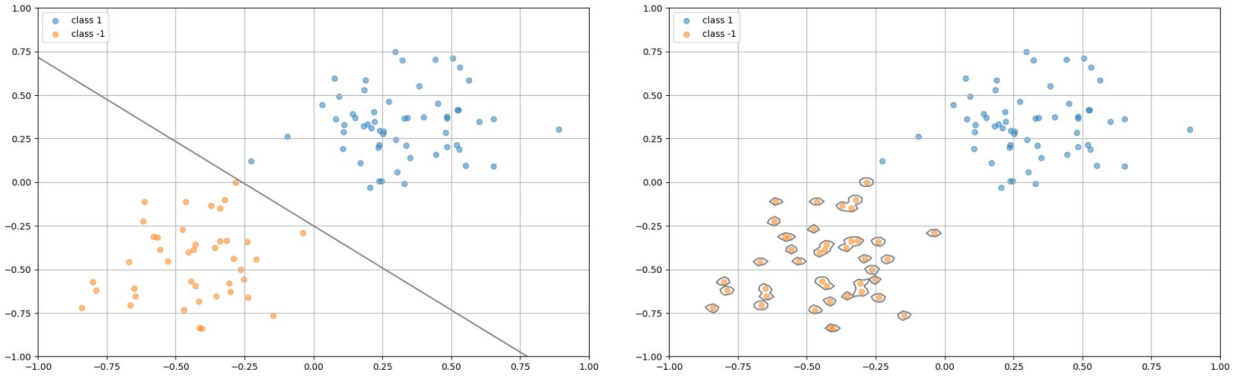

Figure 4: Dataset 1 - Linear/RBF

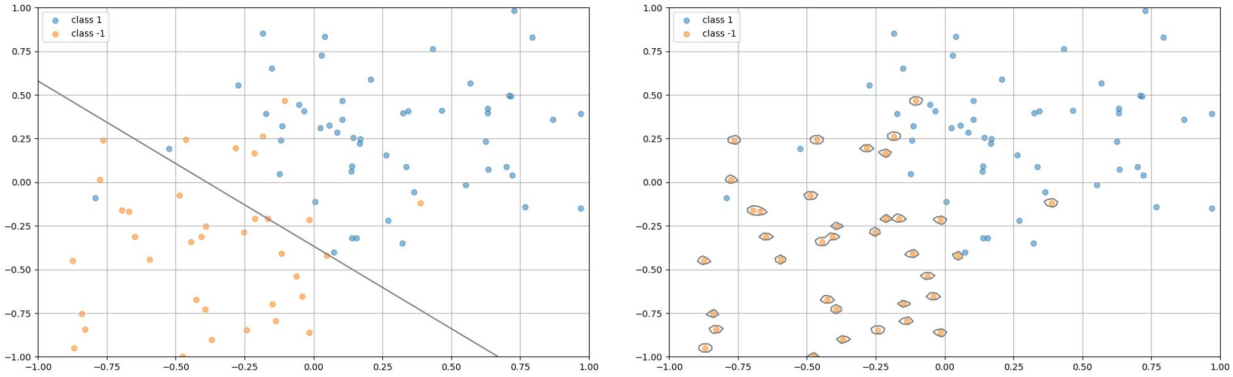

Figure 5: Dataset 2 - Linear/RBF

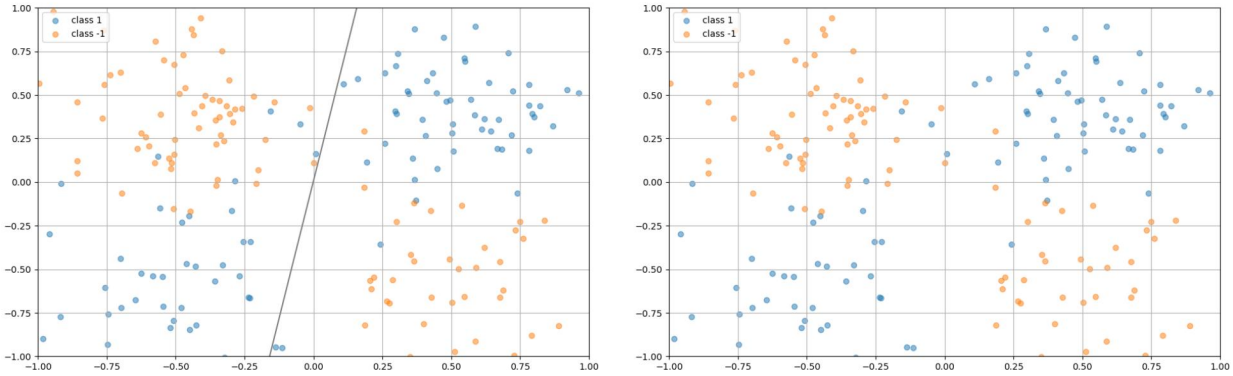

Figure 6: Dataset 3 - Linear/RBF

These results indicate that the linear kernel performs well on datasets with linearly separable classes (such as Dataset 1), but its performance decreases on datasets with more complex and overlapping classes (such as Dataset 3). On the other hand, the Gaussian kernel (RBF kernel) performs consistently well on all datasets, suggesting its ability to capture non-linear relationships between features and labels effectively.

Therefore, we can conclude that for these datasets, the Gaussian kernel is more suitable for classification tasks, especially when dealing with complex or non-linearly separable data.

Question assigned to the following page: 2.1

2 Kernel Ridge Regression

Consider the optimization problem expressed as follows:

$$\min_{f,b} \frac{1}{N} \sum_{i=1}^{N} \|f(x_i) + b - y_i\|^2 + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2$$

Utilizing the Representer theorem, we can transform it into an equivalent finite-dimensional optimization problem:

$$\min_{\alpha,b} \frac{1}{N} \|\mathbf{K}\alpha + b\mathbf{1} - y\|^2 + \frac{\lambda}{2} \alpha^\top \mathbf{K} \alpha, \quad \text{where} \quad \mathbf{K}_{ij} = k(x_i, x_j) \quad \text{and} \quad \mathbf{1} = (1, ..., 1)^\top$$

With this alternative formulation, we can derive closed-form expressions for f and b:

$$f = \sum_{i=1}^{N} \alpha_i k_{x_i}, \quad \text{where} \quad \alpha = (A\mathbf{K} + B)^{-1} A y \quad \text{and} \quad b = \frac{\mathbf{1}^\top C y}{\mathbf{1}^\top C \mathbf{1}}$$

with

$$A = \mathbf{I} - \frac{1}{N} \mathbf{1}\mathbf{1}^\top, \quad B = \frac{\lambda N}{2} \mathbf{I}, \quad \text{and} \quad C = \mathbf{I} - \mathbf{K}(\mathbf{K} + B)^{-1}$$

In a q-dimensional setting, we generalize the problem as follows:

$$\min_{f_j, b_j} \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{q} \|f_j(x_i) + b_j - (y_i)_j\|^2 + \frac{\lambda}{2} \sum_{j=1}^{q} \|f_j\|_{\mathcal{H}}^2 \quad \Longleftrightarrow \quad \min_{\alpha_j, b_j} \frac{1}{N} \sum_{j=1}^{q} \|\mathbf{K}\alpha_j + b_j \mathbf{1} - y_j\|^2 + \frac{\lambda}{2} \sum_{j=1}^{q} \alpha_j^\top \mathbf{K} \alpha_j$$

We can derive expressions for f_j and b_j as:

$$f_j = \sum_{i=1}^{N} \alpha_{ji} k_{x_i}, \quad \text{where} \quad \alpha_j = (A\mathbf{K} + B)^{-1} A y_j \quad \text{and} \quad b_j = \frac{\mathbf{1}^\top C y_j}{\mathbf{1}^\top C \mathbf{1}}$$

Question assigned to the following page: 2.2

```python
class KernelRR:

    def __init__(self,kernel,lmbda):
        self.lmbda = lmbda
        self.kernel = kernel
        self.alpha = None
        self.b = None
        self.support = None
        self.type='ridge'

    def fit(self, X, y):
        self.support = X
        N = X.shape[0]
        K = self.kernel(X, X)
        A = np.eye(N) - np.ones((N,N))/N
        B = self.lmbda*N*np.eye(N)/2
        C = np.eye(N) - K.dot(np.linalg.inv(K+B))
        D = np.ones(N)
        self.b = np.dot(D,np.dot(C,y)) / np.dot(D,np.dot(C,D))
        self.alpha = np.linalg.inv(np.dot(A,K)+B).dot(np.dot(A,y))

    ### Implementation of the separting function $f$
    def regression_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        K_new = self.kernel(self.support, x)
        return np.dot(K_new.T, self.alpha)

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        return self.regression_function(X)+self.b
```

```python
class MultivariateKernelRR:
    def __init__(self,kernel,lmbda):
        self.lmbda = lmbda
        self.kernel = kernel
        self.support = None
        self.alpha = None
        self.b = None
        self.type='ridge'

    def fit(self, X, y):
        self.support = X
        if len(y.shape) == 1:  # Si y est unidimensionnel, le redimensionner
            y = y.reshape(-1, 1)
        N, q = y.shape
        K = self.kernel(X, X)

        self.alpha = np.zeros((N, q))
        self.b = np.zeros(q)

        A = np.eye(N) - np.ones((N,N))/N
        B = self.lmbda*N*np.eye(N)/2
        C = np.eye(N) - K.dot(np.linalg.inv(K+B))
        D = np.ones(N)

        for j in range(q):
            self.b[j] = np.dot(D,np.dot(C,y[:, j])) / np.dot(D,np.dot(C,D))
            self.alpha[:, j] = np.linalg.inv(np.dot(A,K)+B).dot(np.dot(A,y[:, j]))

    ### Implementation of the separting function $f$
    def regression_function(self,x):
        # Input : matrix x of shape N data points times d dimension
        # Output: vector of size N
        K_new = self.kernel(self.support, x)
        return np.dot(K_new.T, self.alpha)

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        return self.regression_function(X)+np.expand_dims(self.b,axis=0)
```

Figure 7: KernelKRR and MultivariateKernelKRR

```python
sigma_values = np.logspace(-3, 3, 100)
lmbda_values = np.logspace(-10, 10, 100)

best_sigma = None
best_lmbda = None
best_mse = np.inf

for sigma in tqdm(sigma_values, desc='Sigma'):
    for lmbda in tqdm(lmbda_values, desc='Lambda', leave=False):
        kernel = RBF(sigma).kernel
        model = KernelRR(kernel, lmbda=lmbda)
        model.fit(train_set['x'].reshape(-1,1), train_set['y'])
        predictions = model.predict(train_set['x'].reshape(-1,1))

        mse = np.mean((predictions - train_set['y_clean'])**2)
        if mse < best_mse:
            best_mse = mse
            best_sigma = sigma
            best_lmbda = lmbda

print(f"Best value of sigma: {best_sigma}")
print(f"Best value of lambda: {best_lmbda}")
print(f"Best MSE: {best_mse}")
```

Figure 8: Best parameters

σ	λ	**MSE**
0.2009233002565048	0.0011768119524349992	0.0018553430783016728

Table 3: Best parameters

Question assigned to the following page: 2.2

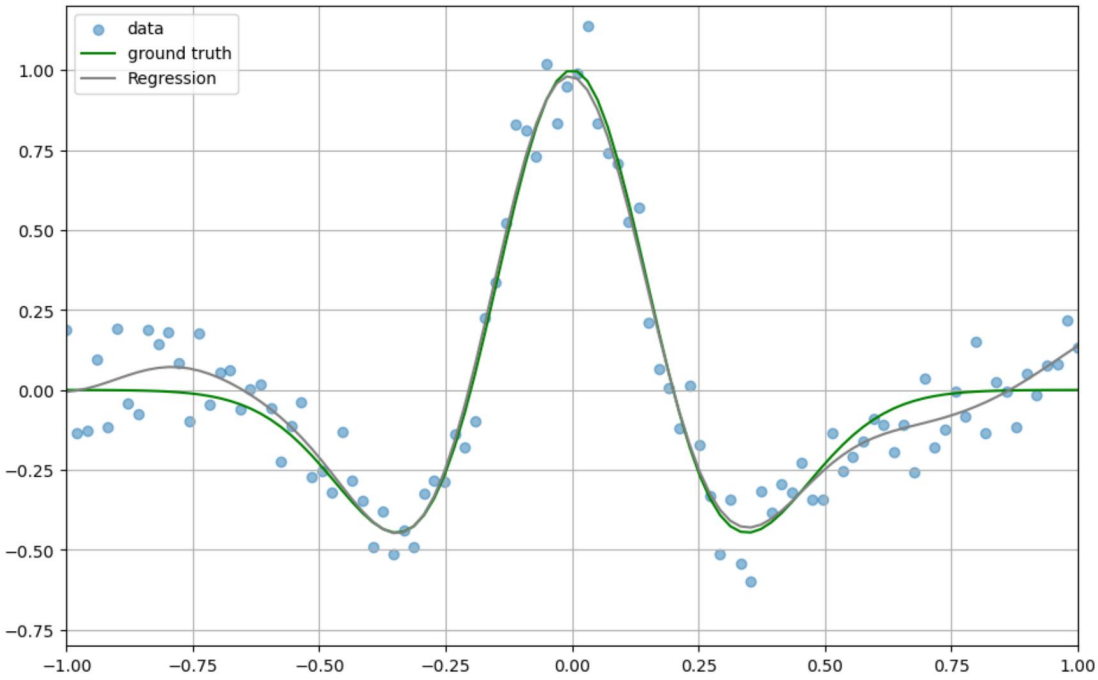

Figure 9: Regression

The kernel ridge regression model effectively traces the general pattern of the data, showing its strength in non-linear fitting. Although it aligns closely with the actual data trend, discrepancies suggest potential refinement opportunities, possibly through more sophisticated modeling techniques to achieve an even closer fit to the ground truth.

Question assigned to the following page: 3.1

3 Kernel PCA

Consider the operator defined as:

$$C = \frac{1}{N}\sum_{i=1}^{N} \widetilde{\varphi}(X_i) \otimes \widetilde{\varphi}(X_i), \quad \text{where} \quad \widetilde{\varphi}(X_i) = \varphi(X_i) - \frac{1}{N}\sum_{j=1}^{N} \varphi(X_j)$$

For a non-trivial eigenvector v of C associated with the eigenvalue λ, we have:

$$Cv = \frac{1}{N}\sum_{i=1}^{N} \langle \widetilde{\varphi}(X_i), v\rangle \, \widetilde{\varphi}(X_i) = \lambda v \quad \iff \quad v = \sum_{i=1}^{N} \left(\frac{\langle \widetilde{\varphi}(X_i), v\rangle}{\lambda N} \right) \widetilde{\varphi}(X_i) = \sum_{i=1}^{N} \alpha_i \widetilde{\varphi}(X_i)$$

Let G be a matrix constructed from inner products of the centered features $\widetilde{\varphi}(X_i)$, defined as:

$$G = \left(\langle \widetilde{\varphi}(X_i), \widetilde{\varphi}(X_j) \rangle \right)_{1 \leq i,j \leq N}$$

Then, we can observe that:

$$(G\alpha)_i = \sum_{j=1}^{N} G_{ij}\alpha_j = \sum_{j=1}^{N} \langle \widetilde{\varphi}(X_i), \widetilde{\varphi}(X_j) \rangle \, \frac{\langle \widetilde{\varphi}(X_j), v\rangle}{\lambda N} = \frac{1}{\lambda} \langle Cv, \widetilde{\varphi}(X_i) \rangle = \langle v, \widetilde{\varphi}(X_i) \rangle = \lambda N \alpha_i$$

Thus, the previously defined vector α is actually an eigenvector of G, associated with the eigenvalue λN. Therefore, every eigenvector v of C associated with an eigenvalue λ is a linear combination of the features $\widetilde{\varphi}(X_i)$ with a vector of coefficients being an eigenvector of some matrix G.

Question assigned to the following page: 3.2

```python
class KernelPCA:

    def __init__(self,kernel, r=2):
        self.kernel = kernel          # <---
        self.alpha = None # Matrix of shape N times d representing the d eingenvectors alpha corresp
        self.lmbda = None # Vector of size d representing the top d eingenvalues
        self.support = None # Data points where the features are evaluated
        self.r = r ## Number of principal components

    def compute_PCA(self, X):
        # assigns the vectors
        self.support = X
        N = X.shape[0]
        K = self.kernel(X,X)

        one_n = np.ones((N, N)) / N
        #G = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)
        G = K

        values, vectors = np.linalg.eigh(G)
        self.lmbda = np.flip(values[-self.r:])
        self.alpha = np.flip(vectors[:, -self.r:])

        # Normalize the eigenvectors
        self.alpha = self.alpha / np.sqrt(self.lmbda*N)

        #constraints = ({})
        # Maximize by minimizing the opposite

    def transform(self, x):
        # Input: matrix x of shape N data points times d dimension
        # Output: vector of size N
        if self.alpha is None:
            raise ValueError("PCA has not been computed. Please run 'compute_PCA' first.")
        k_x_support = self.kernel(x, self.support)
        # Project the input data onto the eigenvectors
        projection = k_x_support @ self.alpha
        return projection
```

Figure 10: KernelPCA

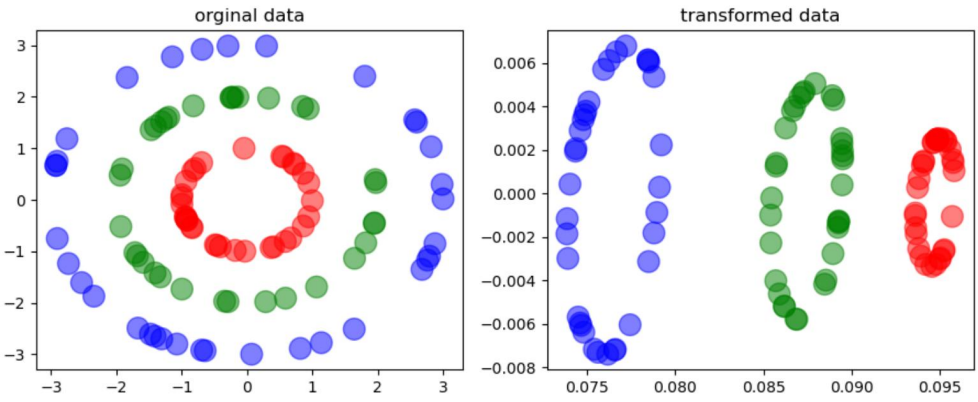

Figure 11: Application

The Kernel PCA transformation has successfully unfolded the complex structure of the original dataset into a new feature space where the concentric circles are now more spread out. This indicates that Kernel PCA can effectively capture the non-linear relationships in the data. It reveals the potential for better data separability in this transformed space, which could be beneficial for further analysis or classification tasks.

Question assigned to the following page: 3.3

```python
class Denoiser:
    def __init__(self, kernel_encoder, kernel_decoder, dim_pca, lmbda):
        self.pca = KernelPCA(kernel_encoder, r=dim_pca)
        self.ridge_reg = MultivariateKernelRR(kernel_decoder, lmbda=lmbda)

    def fit(self, train_noisy, train_clean):
        # Apply PCA on the noisy training data
        self.pca.compute_PCA(train_noisy)
        encoded_train = self.pca.transform(train_noisy)
        # Train ridge regression to map from encoded space to clean data
        self.ridge_reg.fit(encoded_train, train_clean)

    def denoise(self, test_noisy):
        # Transform the test data to the PCA space
        encoded_test = self.pca.transform(test_noisy)
        # Use the trained regression model to predict the clean data
        denoised_test = self.ridge_reg.predict(encoded_test)
        return denoised_test
```

Figure 12: Denoiser

MemoryError: Unable to allocate 23.4 GiB for an array with shape (2000, 28, 28, 2000) and data type float64