

# Assignment Module 2: Quantum-Kernel SVM (QSVM)

This assignment focuses on **quantum-kernel SVM (QSVM)** using a *PennyLane* implementation of the **inverse-circuit overlap test**.

To complete the assignment, you will implement missing code (marked **YOUR CODE HERE**) and answer a set of short theoretical questions.

## Preparation

- Look at the [notebook](#) on Quantum SVM.

## Use of generative AI tools

You may use AI-based tools (e.g., ChatGPT, GitHub Copilot, Claude, Gemini, DeepSeek, ...) for brainstorming, refactoring, coding assistance, plotting, or editing.

This is allowed with disclosure. LLMs are a great tool, but you have to make sure to grasp the contents of the course!

**Make sure to fill in the mandatory AI-disclosure in the notebook before submitting!**

## Preparatory code

Run this to import the modules we need

```
In [1]: # Reproducibility
SEED = 123

# Imports
import math, sys, os, json, pathlib
import numpy as np
np.random.seed(SEED)

import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay,
from sklearn.svm import SVC

import pennylane as qml
from pennylane import numpy as pnp # optional

print("pennylane versino: ", qml.__version__)

print("Imports OK")

pennylane versino: 0.43.1
Imports OK
```

# Task 0: Loading the dataset

In this exercise, we will use the breast cancer dataset. Before designing the QSVM, we must first load the dataset, and split the datapoints into a training and test set. Furthermore, we will apply a scaling to the input features using the `StandardScaler()`, which ensures that the features have zero mean and unity variance. Remember to fit the scaler on the training data, and using the same fitted scaler on both the training and test datasets!

```
In [2]: # Data loading and preprocessing
# TODO: Load Breast Cancer dataset, map labels to {-1,+1}, train/test split, and scal

data = load_breast_cancer()
# -----YOUR CODE HERE-----
# Load dataset
X = data['data']
y01 = data['target']

# Remap binary labels from 0/1 to -1/+1
y = np.where(y01==1, +1, -1)

# Split in train and test datasets
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=42)

# Make mean 0 and standard deviation 1
scaler = StandardScaler()
X_tr = scaler.fit_transform(X_tr)
X_te = scaler.transform(X_te)
# ---YOUR CODE ENDS HERE---

print("Shapes:", X_tr.shape, X_te.shape, " (+1 count):", (y_tr==+1).sum(), " (-1 count):", (y_tr==-1).sum())

Shapes: (455, 30) (114, 30) (+1 count): 286 (-1 count): 169
```

## Question 0.1 — Short answer

Why is feature scaling important for angle-embedding feature maps? Write a concise justification (2–4 sentences).

**Answer:** According to [IBM](#), we scale the data to be such that  $\vec{x}_k \in (0, 2\pi]$  to avoid any potential unwanted effect from the periodicity of rotation gates  $\hat{R}_x(\theta)$ ,  $\hat{R}_y(\theta)$ ,  $\hat{R}_z(\theta)$ .

# Task 1: Dimensionality reduction using PCA

For angle-based maps we set the number of qubits equal to the feature dimension. Unfortunately, the available qubit count is often smaller than the dimensionality of the data. Here, we use PCA to reduce the data dimensionality to `n_qubits` components. Remember to fit the PCA transform on the training data, and using the same fitted scaler on both the training and test datasets!

## Instructions:

1. Choose `n_qubits` (e.g., 8).
2. Fit `PCA(n_components=n_qubits)` on the training data and transform train/test.

3. Print shapes to confirm.

```
In [3]: # PCA reduction to match qubits
# -----YOUR CODE HERE-----
n_qubits = 8
pca = PCA(n_components=n_qubits)
Xtr_red = pca.fit_transform(X_tr)
Xte_red = pca.transform(X_te)
# ---YOUR CODE ENDS HERE---
print("Reduced shapes:", Xtr_red.shape, Xte_red.shape)
```

Reduced shapes: (455, 8) (114, 8)

## Question 1.1 — Short answer

What are the trade-offs in choosing a small vs a large `n_qubits` for angle-embedding maps (consider noise/depth, and dataset size,...)?

**Answer:** A large number of qubits implies a greater capacity to identify patterns from more complex data. At the same time, it introduces more noise and requires deeper architectures. A small number of qubits introduces less noise; however, it may not be enough to represent complex data and/or larger datasets.

## Task 2: Implement an angle-embedding feature map

Implement a simple angle-embedding feature map followed by a ring of  $CZ$  entanglers.

### Instructions:

1. Define a device `default.qubit` with `n_qubits` wires and analytic mode (`shots=None`).
2. Implement `feature_map_angle(x, scale=1.0, entangle=True)` that applies `AngleEmbedding(..., rotation="Y")` and a CZ ring when `entangle=True`.
3. Do **not** return anything; this is a template used inside a QNode.

```
In [5]: # Device and feature map

dev = qml.device("default.qubit", wires=n_qubits, shots=None, seed=SEED)
def feature_map_angle(x, wires=None, scale=1.0, entangle=True):
    # -----YOUR CODE HERE-----
    if wires is None:
        wires = range(n_qubits)
    qml.AngleEmbedding(features=x*scale, wires=wires, rotation='Y')
    if entangle:
        for index in range(len(wires)):
            qml.CZ(wires=[wires[index], wires[(index+1) % len(wires)]])
    # ---YOUR CODE ENDS HERE---
    return None
```

## Question 2.1: Short answer

Explain (2–4 sentences) how the entangling layer can change the induced kernel compared to a no-entanglement feature map.

**Answer:** The entangling layer changes the induced kernel by it to depend on interactions between multiple features. With a no-entanglement feature map the kernel can only capture single feature behavior. An entangling layer is best when there are relationships between features.

## Task 3: Inverse-circuit (overlap) kernel QNode

We estimate  $k(x, z) = |\langle \phi(z) | \phi(x) \rangle|^2$  by the inverse-circuit method.

**Instructions:** Implement the function `all_bitstring_probabilities(x, z, scale=1.0, entangle=True)` as a QNode that

1. Encodes `x` with `feature_map_angle`
2. Applies the adjoint encoding of `z`,
3. Returns the output probabilities corresponding to all possible bitstring outcomes using `qml.probs()`.

The kernel  $k(x, z)$  equals the probability of measuring zero on all qubits, which we denote as  $\Pr(0^n)$ . We must therefore extract the probability of this specific outcome. We provide a helper function `overlap_probability()` that extracts  $\Pr(0^n)$ .

Call `overlap_probability()` on a small subset of your training data points to confirm outputs lie in the range  $[0, 1]$ !

```
In [6]: # TODO: QNode returning Pr(0^n)
@qml.qnode(dev)
def all_bitstring_probabilities(x, z, scale=1.0, entangle=True):
    # -----YOUR CODE HERE-----
    feature_map_angle(x=x, wires=range(n_qubits), scale=scale, entangle=entangle)
    qml.adjoint(feature_map_angle)(x=z, scale=scale, entangle=entangle)
    probs = qml.probs()

    # ---YOUR CODE ENDS HERE---
    return probs

def overlap_probability(x, z, scale=1.0, entangle=True):
    return all_bitstring_probabilities(x, z, scale=scale, entangle=entangle)[0]

# Quick sanity check for x_5
print("Overlap:", overlap_probability(Xtr_red[5], Xtr_red[5]))
```

Overlap: 1.0000000000000000

### Question 3.1 — Derivation

Show that  $\Pr(0^n)$  equals the fidelity kernel value  $k(x, z) = |\langle \phi(z) | \phi(x) \rangle|^2$ , where  $\Pr(0^n)$  denotes the probability of measuring 0 on all qubits in the circuit above. Provide a short derivation using Dirac notation (2–8 lines).

**Answer:** Here I will show that  $\Pr(0^n)$  is equal to the fidelity kernel value  $k(x, z) = |\langle \phi(z) | \phi(x) \rangle|^2$ . Let's start with the general probability of detecting a quantum state  $|a\rangle$ . This can be written as

$$\Pr(a) = |\langle a | \psi \rangle|^2,$$

where  $\psi$  is the current quantum state. Applying this to our circuit, which applies the unitary matrices  $U_\phi(x)$  followed by  $U_\phi(z)^\dagger$  to the state  $|0\rangle$ , we are obtaining the quantum state  $|\psi\rangle = U_\phi(z)^\dagger U_\phi(x) |0\rangle$ . Then, the probability of measuring the state  $|0\rangle$  is given by

$$\Pr(0^n) = |\langle 0 | U_\phi(z)^\dagger U_\phi(x) | 0 \rangle|^2.$$

We also know that the encoded state  $|\phi(x)\rangle$  is given by  $|\phi(x)\rangle = U_\phi(x) |0\rangle$ . By the unitary properties of  $U$ , we can also write that  $\langle \phi(z) | = \langle 0 | U_\phi(z)^\dagger$ . Plugging the expressions for  $|\phi(x)\rangle$  and  $\langle \phi(z) |$  into our expression for  $\Pr(0^n)$  yields

$$\begin{aligned} \Pr(0^n) &= |\langle \phi(z) | \phi(x) \rangle|^2 \\ &= k(x, z). \end{aligned}$$

## Task 4: Build a Gram matrix

Implement `quantum_kernel(A, B, ...)` that returns a matrix  $K$  with entries  $K_{ij} = k(A_i, B_j)$ .

### Instructions:

1. Loop over rows of `A` and `B` and call `overlap_probability` in order to construct the Gram matrix.
2. Time your function on a small subset (e.g., 40×40) and comment on the ( $O(n^2)$ ) cost.

```
In [7]: # Gram matrix builder
def quantum_kernel(A, B, scale=1.0, entangle=True):
    # -----YOUR CODE HERE-----
    K = np.empty(shape=(len(A), len(B)), dtype=float)
    for index_0 in range(len(A)):
        for index_1 in range(len(B)):
            K[index_0, index_1] = overlap_probability(x=A[index_0], z=B[index_1], sca
    # ---YOUR CODE ENDS HERE---
    return K

# mini benchmark
Amini = Xtr_red[:40]; Bmini = Xtr_red[:40]
Kmini = quantum_kernel(Amini, Bmini)
print("Kmini shape:", Kmini.shape, " min/max:", Kmini.min(), Kmini.max())
```

Kmini shape: (40, 40) min/max: 5.391015146694988e-17 1.0000000000000000

### Question 4.1: Symmetry and jitter

Why do we not need to symmetrize and add jitter now that we run on a simulator?

**Answer:** We build the gram matrix  $K$  by evaluating  $k(x_i, x_j)$  for all training pairs of  $(i, j)$ . However, when we build  $K$  using quantum hardware, there will be some noise and gate errors

that can negatively impact this matrix construction. To mitigate these effects we often symmetrize to get a more consistent estimate of  $K$ , and add a jitter to ensure that  $K$  is positive semidefinite (PSD). While using a simulator, the Gram matrix will be perfectly symmetrical and PSD, since there is no noise or gate errors to break the symmetry and PSD.

## Task 5: Train a QSVM with a precomputed kernel

Train an SVM on the **precomputed** Gram matrix.

### Instructions.

1. Compute the Gram matrix of `Xtr_red` and `Xtr_red`.
2. Fit `SVC(kernel="precomputed", C=...)` on `(K_tr, y_tr)`.
3. Compute the Gram matrix of the `Xtr_red` and `Xtr_red`. Predict on test data, report accuracy and confusion matrix.
4. Briefly comment on your results (1–3 sentences).

```
In [10]: # -----YOUR CODE HERE-----
# Create the Gram matrix for the training data.
# Note that this might take around 20-40 min to run.
K_tr = quantum_kernel(A=Xtr_red, B=Xtr_red, scale=1.0, entangle=True)
# ---YOUR CODE ENDS HERE---

# We save the matrix for further use
np.savetxt("K_tr.npy", K_tr)
```

Use the Gram matrix to train the data.

```
In [8]: K_tr = np.loadtxt("K_tr.npy")

C = 0.9 # Hyper parameter you may tune
clf = SVC(kernel="precomputed", C=C)

# -----YOUR CODE HERE-----
# Fit the classifier to the training data
clf.fit(X=K_tr, y=y_tr)
# ---YOUR CODE ENDS HERE---
```

```
Out[8]: SVC ⓘ ?
        ► Parameters
```

Compute the Gram matrix for all pairs of training and test data.

```
In [8]: # -----YOUR CODE HERE-----
K_te_tr = quantum_kernel(A=Xte_red, B=Xtr_red)
# ---YOUR CODE ENDS HERE---
np.savetxt("K_te_tr.npy", K_te_tr)
```

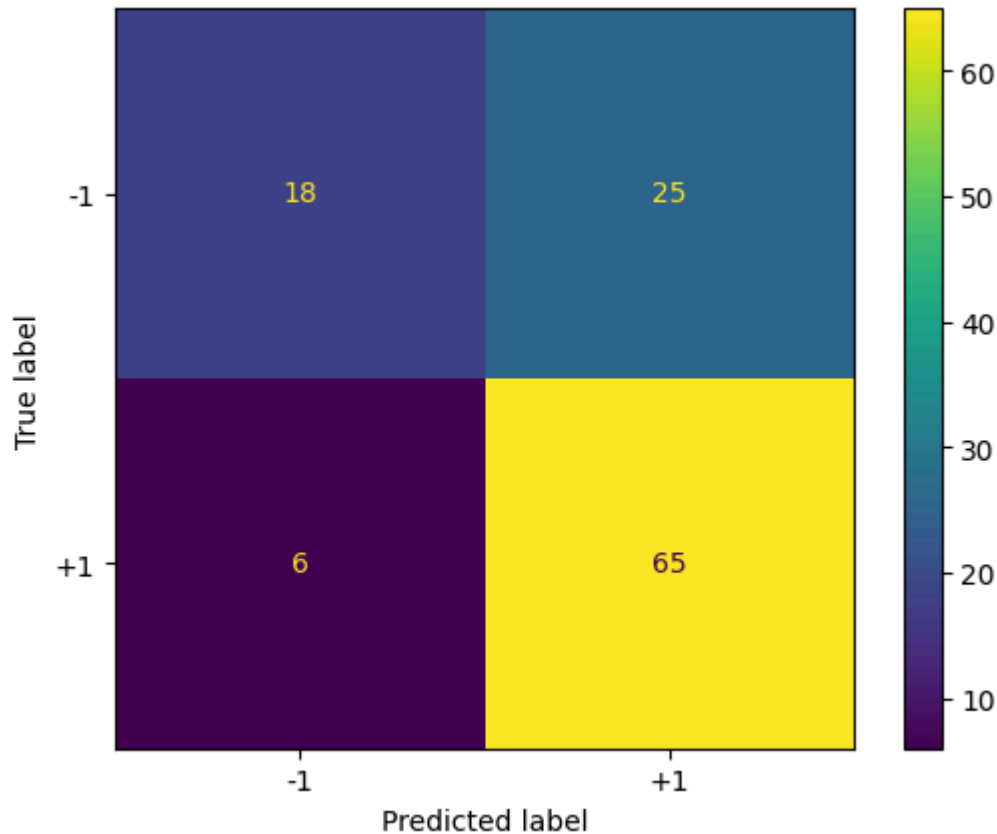
Use the classifier and the Gram matrix of the training and test data to predict labels for the test data.

```
In [8]: K_te_tr = np.loadtxt("K_te_tr.npy")

# -----YOUR CODE HERE-----
y_pred = clf.predict(X=K_te_tr)
acc = accuracy_score(y_true=y_te, y_pred=y_pred)
# ---YOUR CODE ENDS HERE---

print(f"QSVM test accuracy: {acc:.4f}")
cm = confusion_matrix(y_te, y_pred, labels=[-1,+1])
ConfusionMatrixDisplay(cm, display_labels=["-1", "+1"]).plot(values_format="d"); plt.s
print(classification_report(y_te, y_pred, target_names=["neg(-1)", "pos(+1)"]))
```

QSVM test accuracy: 0.7281



	precision	recall	f1-score	support
neg(-1)	0.75	0.42	0.54	43
pos(+1)	0.72	0.92	0.81	71
accuracy			0.73	114
macro avg	0.74	0.67	0.67	114
weighted avg	0.73	0.73	0.71	114

## Question 5.1: Short answer

Where do labels ( $\pm 1$ ) enter the SVM training when using a precomputed kernel? Answer in 2–4 sentences.

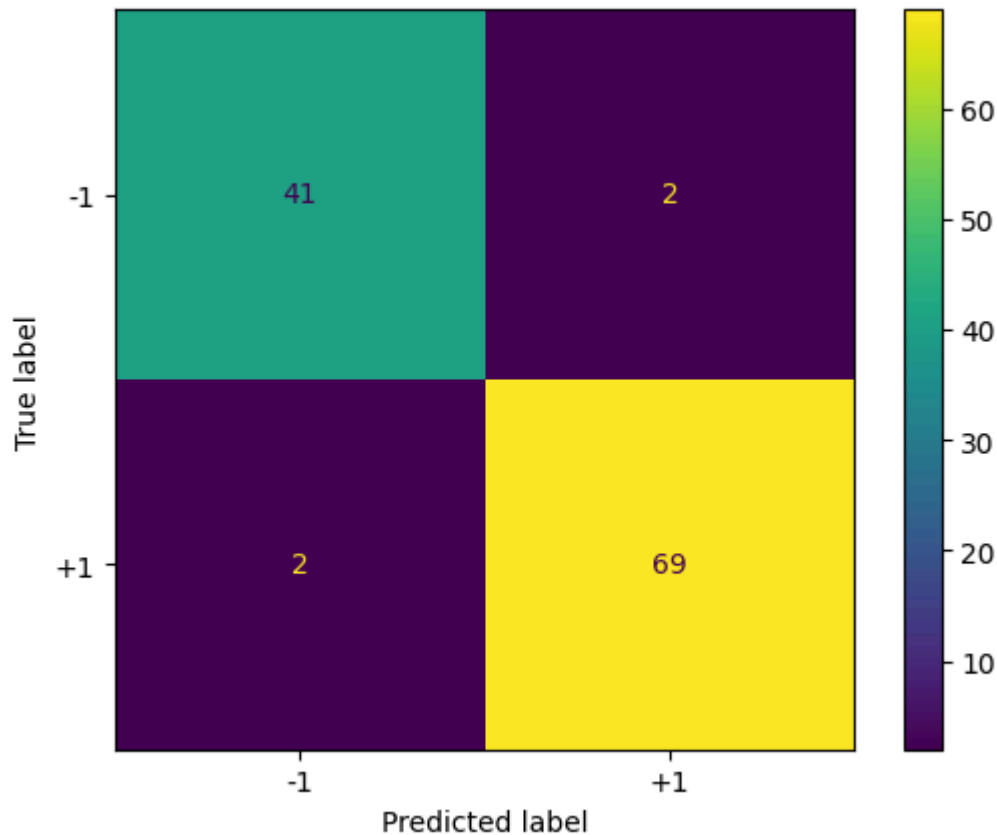
**Answer:** The labels ( $\pm 1$ ) enter the SVM training when the data, namely when calling `clf.fit(X=K_tr, y=y_tr)`. Now, the SVM can correlate the labels to the precomputed Gram matrix, `K_tr`.

## Task 6: Classical baseline (RBF SVM)

The code below trains a conventional SVM with an RBF kernel on the same PCA-reduced features. Compare accuracy and qualitative behavior.

```
In [9]: rbf = SVC(kernel="rbf", C=1.0, gamma="scale", random_state=SEED)
rbf.fit(Xtr_red, y_tr)
rbf_pred = rbf.predict(Xte_red)
print("RBF SVM accuracy:", accuracy_score(y_te, rbf_pred))
cm = confusion_matrix(y_te, rbf_pred, labels=[-1,+1])
ConfusionMatrixDisplay(cm, display_labels=["-1", "+1"]).plot(values_format="d"); plt.s
```

RBF SVM accuracy: 0.9649122807017544



### Question 6.1: Short answer

How does the result differ between the QSVM and the classical SVM? Answer in 2–4 sentences.

**Answer:** The classical SVM performed much better than the QSVM. The QSVM struggled to correctly identify the label -1 and achieved an accuracy of only 0.73, whereas the classical SVM classified each label quite well, with an accuracy of 0.96. The QSVM also often made the mistake of classifying class -1 as +1.

## Bonus A: Gram-matrix spectral analysis

Perform a basic spectral analysis to diagnose kernel quality and potential concentration.

**Instructions:**

1. Compute eigenvalues of  $K$  (and of the centered kernel  $K_c = HKH$  with  $H = I - \frac{1}{n}\mathbf{1}\mathbf{1}^\top$ ).
2. Plot scree diagrams (eigenvalues in descending order).
3. Compute effective rank  $r_{\text{eff}} = (\sum \lambda_i)^2 / \sum \lambda_i^2$ .
4. Plot histograms of same-class vs different-class similarities.

```
In [10]: # Spectral analysis of K
# -----YOUR CODE HERE-----
# These are the functions from QSVM_Spectral_Analysis_DD2368.ipynb
def symmetrize(K):
    return 0.5 * (K + K.T)

def center_kernel(K):
    n = K.shape[0]
    H = np.eye(n) - np.ones((n, n)) / n
    return H @ K @ H

def spectrum(Kc):
    w, _ = np.linalg.eigh(symmetrize(Kc))
    w = w[:, :-1]
    return w

def effective_rank(eigs, clip_at=0.0):
    # Clip tiny negatives from numerical issues before r_eff
    lam = np.copy(eigs)
    lam[lam < clip_at] = 0.0
    s1 = lam.sum()
    s2 = (lam**2).sum()
    return (s1**2 / s2) if s2 > 0 else 0.0

def scree_plot(eigs, title="Scree plot"):
    x = np.arange(1, len(eigs)+1)
    plt.figure()
    plt.plot(x, eigs, marker='o')
    plt.xlabel("Component index (desc)")
    plt.ylabel("Eigenvalue of $K_c$")
    plt.title(title)
    plt.tight_layout()
    plt.show()

# ---YOUR CODE ENDS HERE---

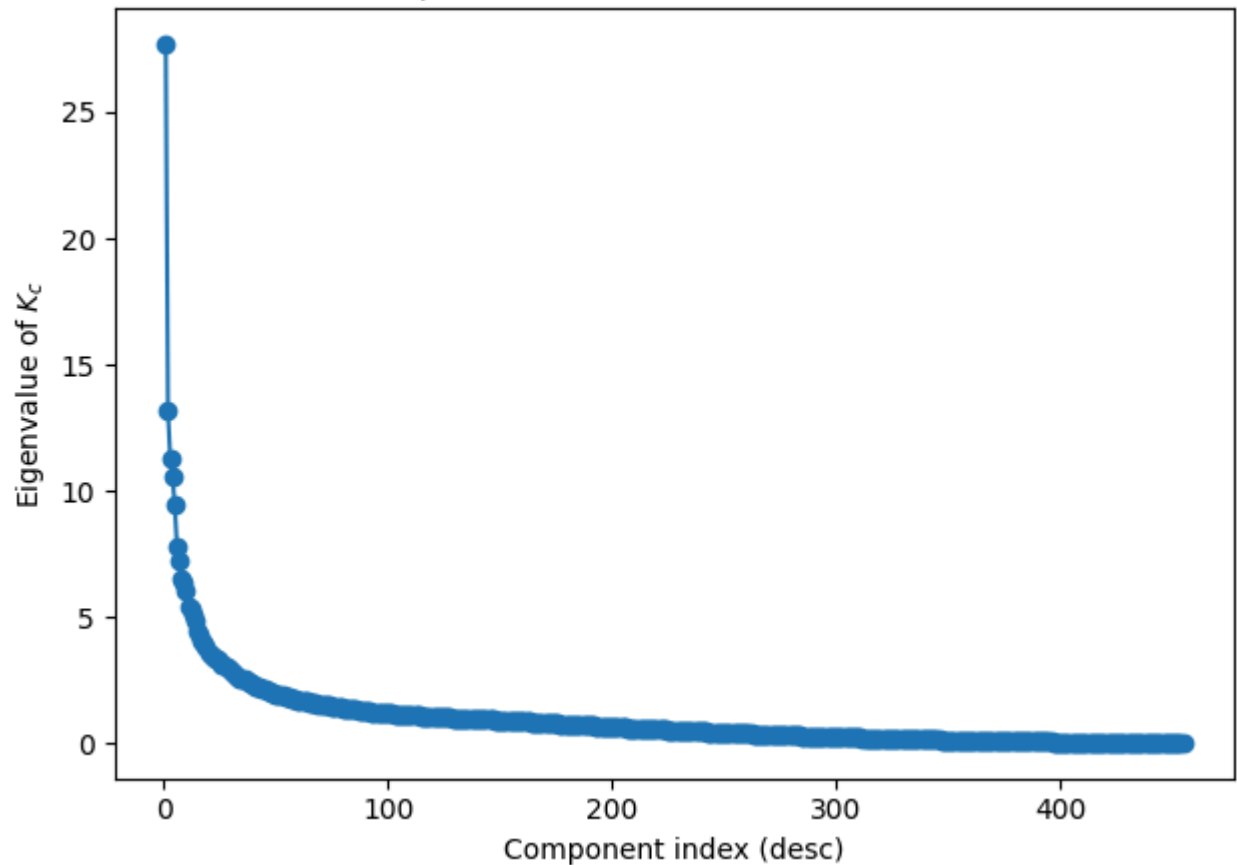
# Apply the functions to analyze the quantum kernel
K_tr_sym = symmetrize(K_tr)
K_tr_centered = center_kernel(K_tr_sym)

# Compute eigenvalues
eigs = spectrum(K_tr_sym)
eigs_c = spectrum(K_tr_centered)

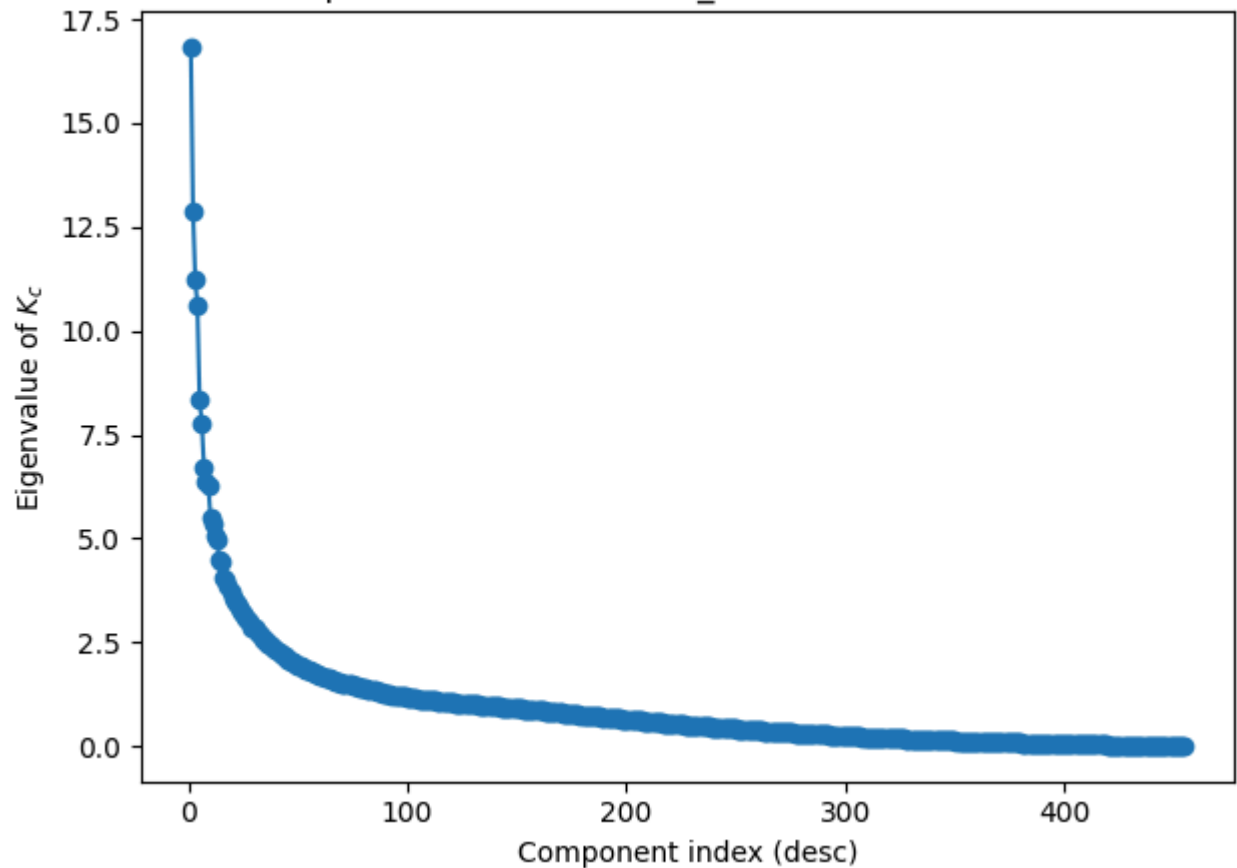
# Compute effective ranks
eff_rank = effective_rank(eigs)
eff_rank_c = effective_rank(eigs_c)

# Plot scree diagrams
scree_plot(eigs, title=f"Spectrum of K (eff. rank ≈ {eff_rank:.2f})")
scree_plot(eigs_c, title=f"Spectrum of centered K_c (eff. rank ≈ {eff_rank_c:.2f})")
```

Spectrum of K (eff. rank  $\approx 96.62$ )



Spectrum of centered  $K_c$  (eff. rank  $\approx 120.21$ )



```
In [13]: # -----YOUR CODE HERE-----
# Similarity histogram: same vs diff classes
def similarity_distributions(K, y, title="Within vs Between similarities"):
    # Use upper triangle i<j to avoid duplicates/self
    n = K.shape[0]
    within, between = [], []
    for i in range(n):
```

```

        for j in range(i+1, n):
            if y[i] == y[j]:
                within.append(K[i,j])
            else:
                between.append(K[i,j])

plt.figure()
bins = 20
plt.hist(within, bins=bins, alpha=0.7, label="Within-class")
plt.hist(between, bins=bins, alpha=0.7, label="Between-class")
plt.xlabel("Similarity K[i,j]")
plt.ylabel("Count")
plt.title(title)
plt.legend()
plt.tight_layout()
plt.show()

return within, between

def diagnostic_report(K, y, label):
    # 1) Symmetrize
    K_sym = symmetrize(K)
    # 2) Center
    Kc = center_kernel(K_sym)
    # 3) Spectrum
    eigs = spectrum(Kc)
    r_eff = effective_rank(eigs, clip_at=0.0)

    print(f"[{label}] n={K.shape[0]}")
    print(f" Sum eigs: {eigs.sum():.4f} | r_eff: {r_eff:.2f}")
    print(f" Top-5 eigs: {np.round(eigs[:5], 6)}")

    # 4) Plots
    # scree_plot(eigs, title=f"Scree - {label}")
    same, diff = similarity_distributions(K_sym, y, title=f"Similarities - {label}")

    return {"label": label, "eigs": eigs, "r_eff": r_eff, "K_sym": K_sym, "Kc": Kc, '

quantum_diagnostics = diagnostic_report(K_tr, y_tr, "Quantum Kernel")
same = quantum_diagnostics['same']
diff = quantum_diagnostics['diff']
# ---YOUR CODE ENDS HERE---

plt.figure()
plt.hist(same, bins=40, alpha=0.6, density=False, label="same-class")
plt.hist(diff, bins=40, alpha=0.6, density=False, label="diff-class")
plt.title("Kernel similarities: same vs different class"); plt.legend(); plt.show()

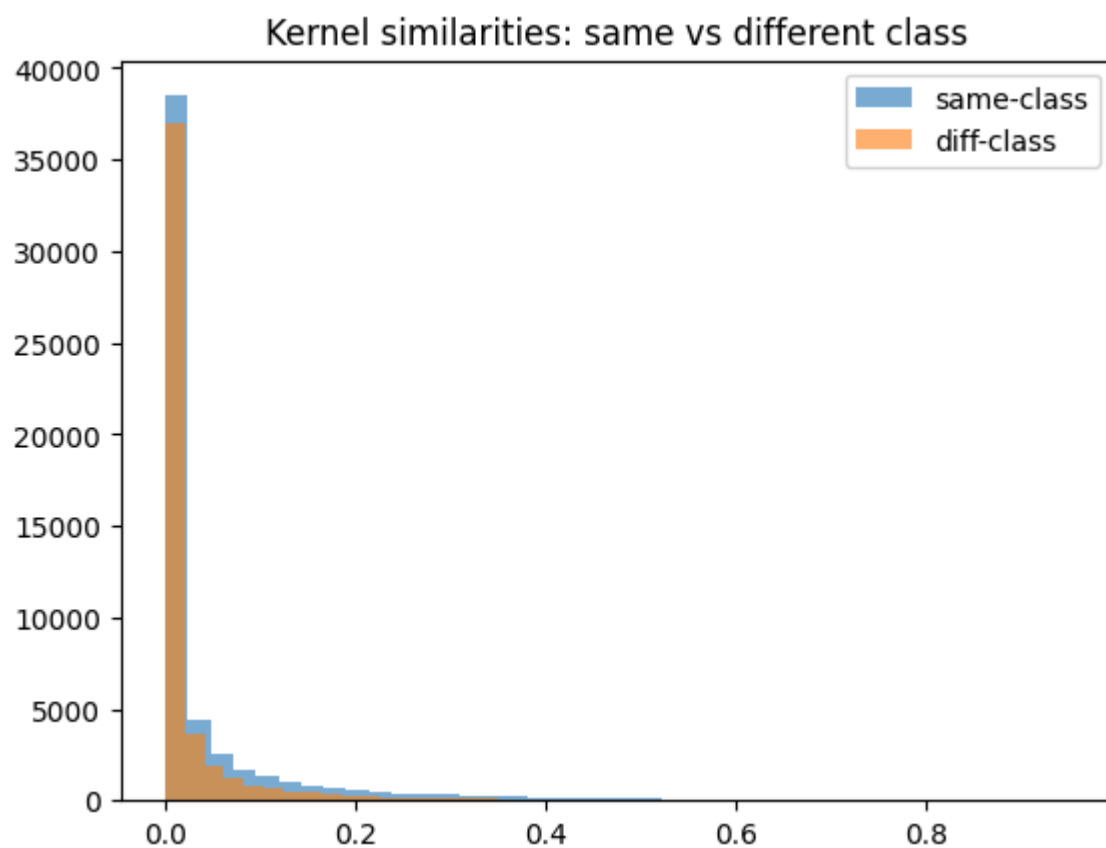
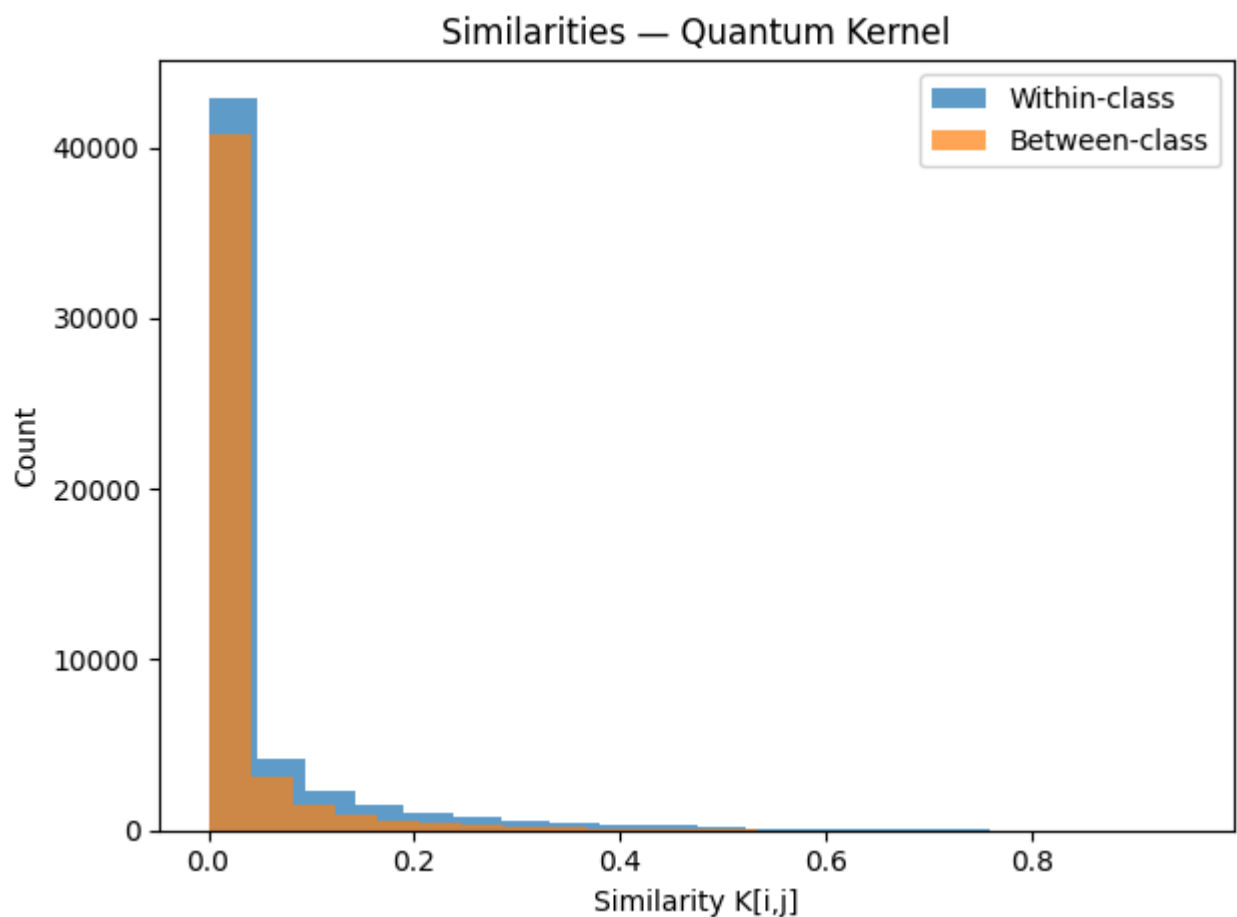
print("Top-5 eigenvalues:", eigs[:5])

```

```

[Quantum Kernel] n=455
Sum eigs: 437.8105 | r_eff: 120.21
Top-5 eigs: [16.835218 12.855159 11.254304 10.594973 8.323904]

```



Top-5 eigenvalues: [27.70782571 13.15646059 11.27999754 10.5959794 9.43850635]

## Question Bonus A.1: Interpretation

Comment on the spectra you obtained. Do you see signs of kernel concentration (near-identity or rank-1 structure)? How does the centered spectrum differ?

**Answer:** From the previous plots, the effective ranks for both kernels (regular and centered) were not near 1, and the eigenvalues gradually decayed. The difference between the regular and the centered version is that, for the first, we got larger eigenvalues, while we noticed that the eigenvalues of the latter were smaller. In summary, we didn't see any significant kernel concentration.

## Bonus B: Feature-map variations

Evaluate the impact of modifying the feature map (e.g., adding depth or changing rotation axis).

### Instructions:

1. Implement an alternative map (e.g., two angle+CZ layers or `rotation="X"`).
2. Rebuild  $K_{tr}$ ,  $K_{te}$ , retrain the SVM, and report test accuracy.
3. Compare spectra as in Bonus A. Summarize observations (3-6 sentences).

```
In [ ]: # Alternative feature map (Y-embedding with two layers)
# -----YOUR CODE HERE-----
def feature_map_angle_alt(x, wires = None, scale = 1.0, entangle = True):
    if wires is None:
        wires = range(n_qubits)

    # Layer 1:
    qml.AngleEmbedding(features = x * scale, wires = wires, rotation = 'Y')
    if entangle:
        for index in range(len(wires)):
            qml.CZ(wires=[wires[index], wires[(index + 1) % len(wires)]])

    # Layer 2:
    qml.AngleEmbedding(features = x * scale, wires = wires, rotation = 'Y')
    if entangle:
        for index in range(len(wires)):
            qml.CZ(wires=[wires[index], wires[(index + 1) % len(wires)]])
    return None

@qml.qnode(dev)
def all_bitstring_probabilities_alt(x, z, scale = 1.0, entangle = True):
    feature_map_angle_alt(x = x, wires = range(n_qubits), scale = scale, entangle = entangle)
    qml.adjoint(feature_map_angle_alt)(x = z, scale = scale, entangle = entangle)
    probs = qml.probs()
    return probs

def overlap_probability_alt(x, z, scale = 1.0, entangle = True):
    return all_bitstring_probabilities_alt(x, z, scale = scale, entangle = entangle)

def quantum_kernel_alt(A, B, scale = 1.0, entangle = True):
    K = np.empty(shape = (len(A), len(B)), dtype = float)
    for index_0 in range(len(A)):
        for index_1 in range(len(B)):
            K[index_0, index_1] = overlap_probability_alt(x = A[index_0], z = B[index_1])
    return K
```

```
In [ ]: K_tr_alt = np.loadtxt("K_tr_alt.npy")

C_alt = 0.9
clf_alt = SVC(kernel="precomputed", C=C_alt)
```

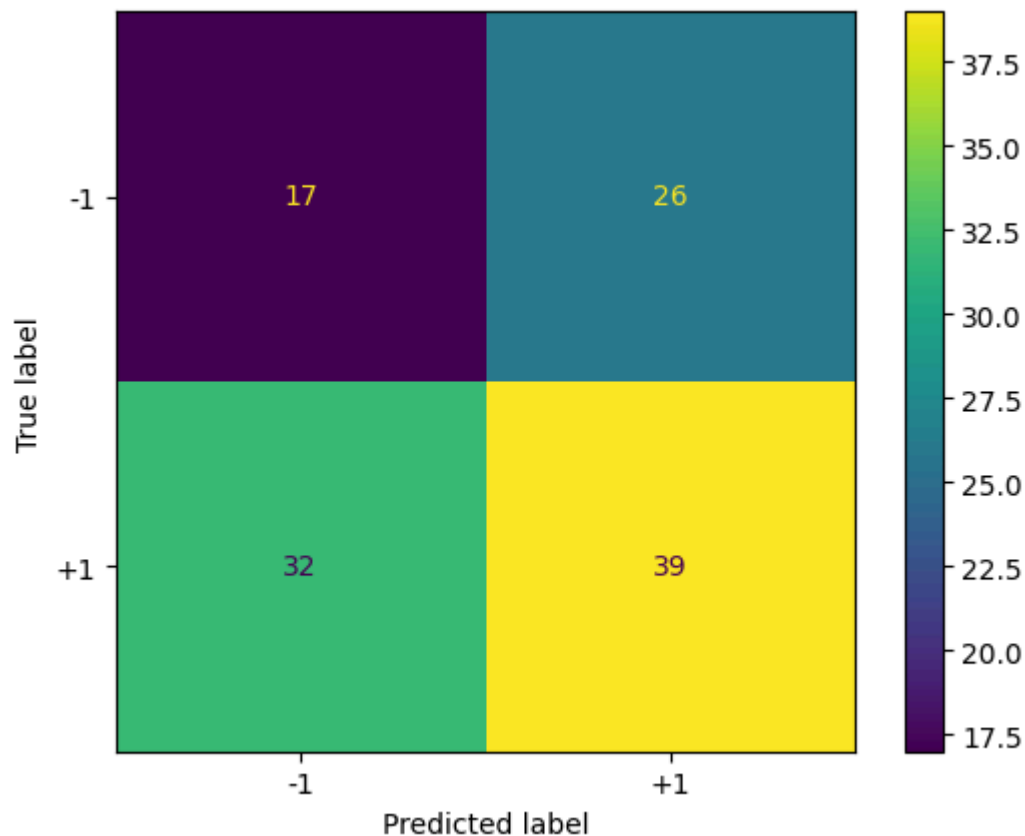
```

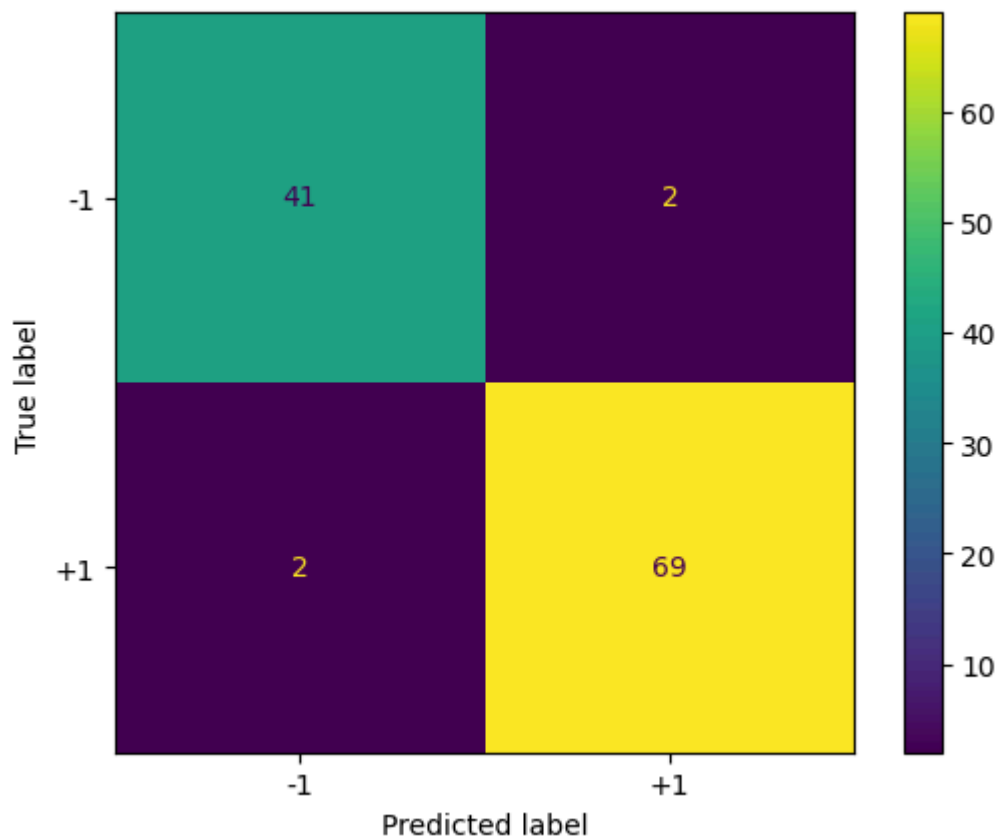
clf_alt.fit(X=K_tr_alt, y=y_tr)

K_te_tr_alt = np.loadtxt("K_te_tr_alt.mpy")
y_pred_alt = clf.predict(X=K_te_tr_alt)
acc_alt = accuracy_score(y_true=y_te, y_pred=y_pred_alt)
cm_alt = confusion_matrix(y_te, y_pred_alt, labels=[-1,+1])
ConfusionMatrixDisplay(cm_alt, display_labels=["-1", "+1"]).plot(values_format="d"); p

# TASK 6
rbf_alt = SVC(kernel="rbf", C=1.0, gamma="scale", random_state=SEED)
rbf_alt.fit(Xtr_red, y_tr)
rbf_pred_alt = rbf_alt.predict(Xte_red)
cm_alt = confusion_matrix(y_te, rbf_pred_alt, labels=[-1,+1])
ConfusionMatrixDisplay(cm_alt, display_labels=["-1", "+1"]).plot(values_format="d"); p

```





```
In [16]: print(f"QSV test accuracy: {acc_alt:.4f}")
print(classification_report(y_te, y_pred_alt, target_names=["neg(-1)", "pos(+1)"]))
print("RBF SVM accuracy:", accuracy_score(y_te, rbf_pred_alt))
```

QSV test accuracy: 0.4912

	precision	recall	f1-score	support
neg(-1)	0.35	0.40	0.37	43
pos(+1)	0.60	0.55	0.57	71
accuracy			0.49	114
macro avg	0.47	0.47	0.47	114
weighted avg	0.50	0.49	0.50	114

RBF SVM accuracy: 0.9649122807017544

```
In [ ]: K_tr_sym_alt = symmetrize(K_tr_alt)
K_tr_centered_alt = center_kernel(K_tr_sym_alt)

eigs_alt = spectrum(K_tr_sym_alt)
eigs_c_alt = spectrum(K_tr_centered_alt)

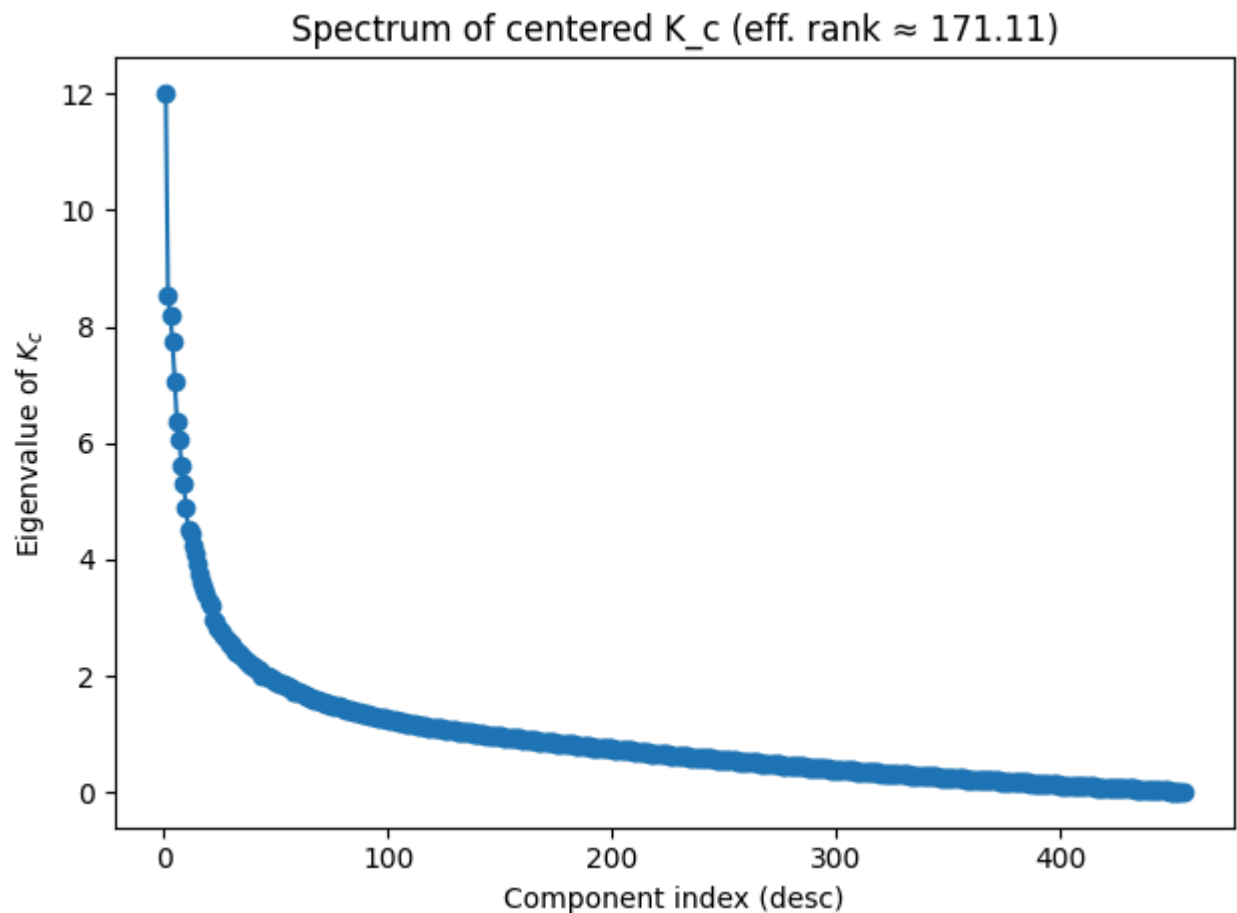
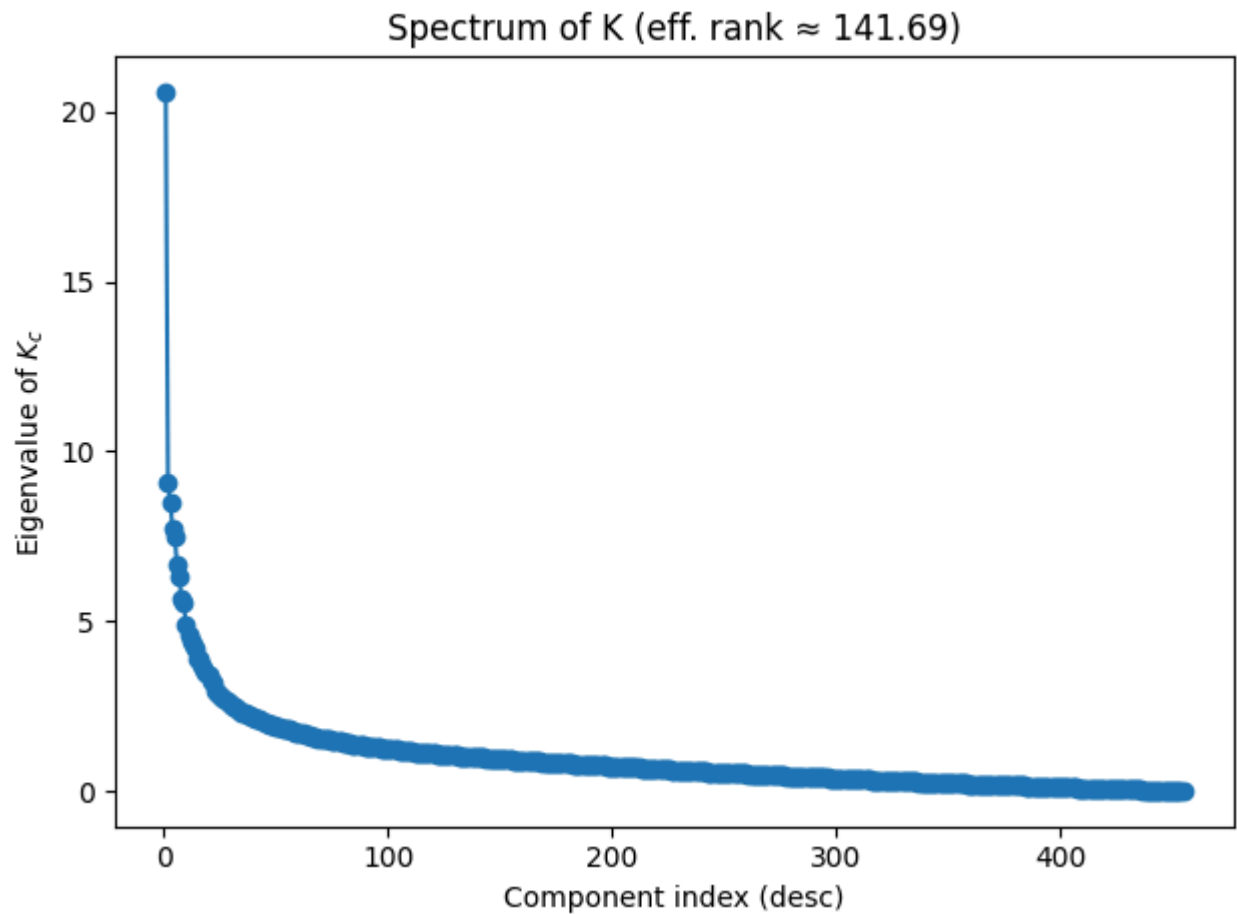
eff_rank_alt = effective_rank(eigs_alt)
eff_rank_c_alt = effective_rank(eigs_c_alt)

scree_plot(eigs_alt, title = f"Spectrum of K (eff. rank ≈ {eff_rank_alt:.2f})")
scree_plot(eigs_c_alt, title = f"Spectrum of centered K_c (eff. rank ≈ {eff_rank_c_alt:.2f})")

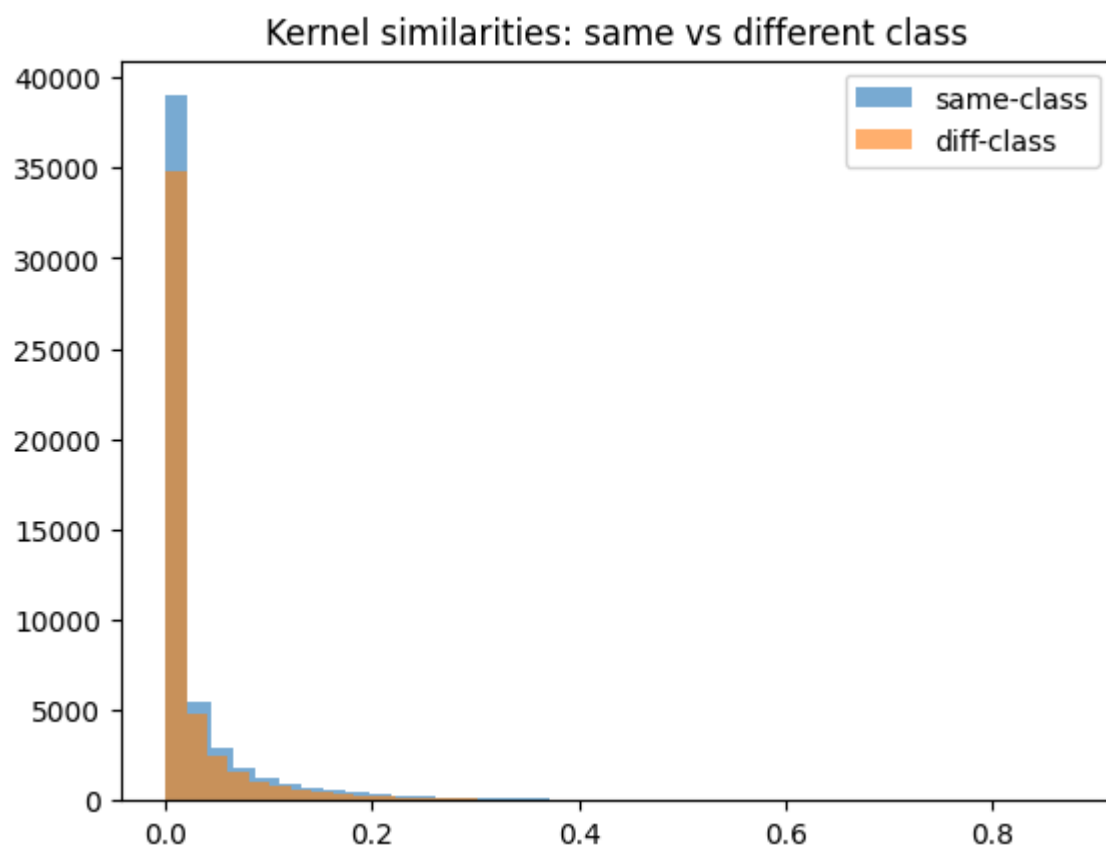
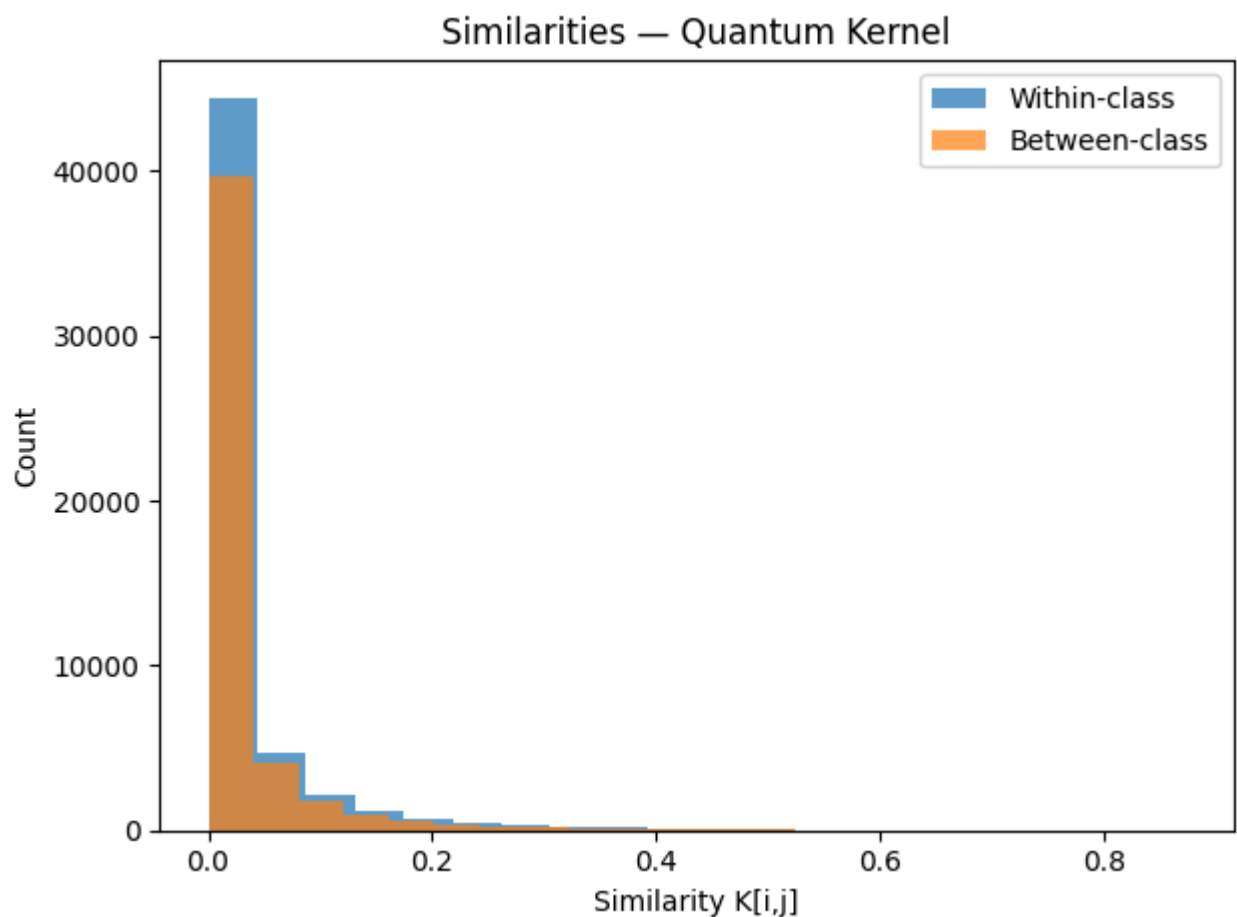
quantum_diagnostics_alt = diagnostic_report(K_tr_alt, y_tr, "Quantum Kernel")
same_alt = quantum_diagnostics_alt['same']
diff_alt = quantum_diagnostics_alt['diff']

plt.figure()
plt.hist(same_alt, bins=40, alpha=0.6, density=False, label="same-class")
plt.hist(diff_alt, bins=40, alpha=0.6, density=False, label="diff-class")
plt.title("Kernel similarities: same vs different class"); plt.legend(); plt.show()
```

```
print("Top-5 eigenvalues:", eigs_alt[:5])
print("Top-5 (centered) eigenvalues:", eigs_c_alt[:5])
# -----YOUR CODE HERE-----
```



```
[Quantum Kernel]  n=455
Sum eigs: 440.6395 |  r_eff: 171.11
Top-5 eigs: [12.018343  8.530734  8.196106  7.741219  7.07393 ]
```



Top-5 eigenvalues: [20.57285657 9.06841855 8.52742123 7.74857686 7.51799458]

Top-5 (centered) eigenvalues: [12.01834269 8.5307336 8.19610557 7.74121859 7.07393033]

## Question Bonus B.1: Wrap-up discussion

Write 1–2 short paragraphs summarizing your results: performance, any signs of concentration, and whether feature-map changes improved or degraded the kernel and

accuracy.

**Answer:** Compared to the first implementation, the two-layer implementation took almost twice as long to run, which was to be expected due to the complexity increment of the circuit. Also, the accuracy of the second implementation was lower than that obtained using the original feature map. The precision of the first implementation was 75% for the negative class and 72% for the positive class; Meanwhile, the accuracy was 35% for the negative class and 60% for the positive class using the second feature map. Another observation is that the eigenvalues from the second implementation were more concentrated than those from the first implementation.

## Feedback to us

### Optional question

Was there any part of the tasks where you struggled for some "unnecessary" reason? (Errors in the notebook, bad instructions etc.)

**Answer:** [Your optional answer here]

## Disclosure of AI Usage (Mandatory)

Fill in this part disclosing any AI usage before submitting the assignment by describing your use of LLMs or other AI-based tools in this assignment.

For each task, we ask you to provide information about:

- **Tools/models used.**
- **Per-task usage:** for each task, a brief summary of what the tool was used for.
- **Prompts/transcripts:** main prompts or a summary of interactions (a link or screenshot is acceptable if long).
- **Validation:** how you checked and verified the correctness of AI-generated outputs (tests run, docs consulted, comparisons, plots etc.).

Disclosure:

- **Task 0:** None
- **Task 1:** None
- **Task 2:** None
- **Task 3:** We used Chat-GPT to obtain the general probability formula of measuring the state  $|a\rangle$ , given as  $\Pr(a) = |\langle a|\psi\rangle|^2$ .
- **Task 4:** We used Chat-GPT to explain the formula for symmetrization of the Gram matrix.
- **Task 5:** We used Chat-GPT to help tune the hyperparameter C.

- **Task 6:** None
- **Bonus A:** We used Chat-GPT to interpret the results.
- **Bonus B:** None

If you did not use any AI tools for a given exercise, specify this by writing "None". If you did not complete the bonus exercises, you can leave those fields empty.