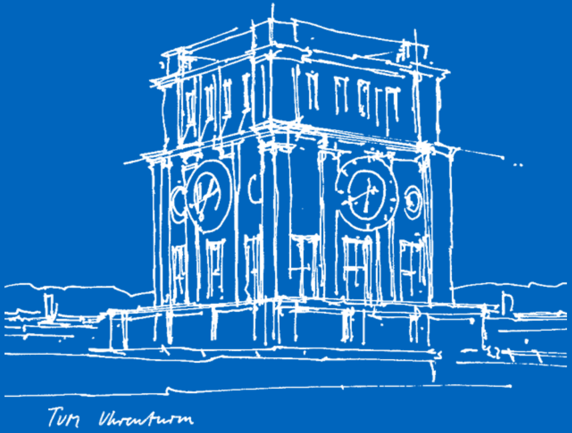
Chair of Scientific Computing in Computer Science
TUM Department of Informatics
Technical University of Munich

Quantum Machine Learning

Quantum Support Vector Machines



Maximilian Forstenhäusler

Chair of Scientific Computing in Computer Science
TUM Department of Informatics
Technical University of Munich

December 10^{th}, 2021

Outline

Introduction to Quantum Machine Learning
Machine Learning

- Training machines to learn from the algorithms implemented to handle the data [1]
- Classical machine learning helps to classify images, recognize patterns and speech, handles big data and many more [1]
- Nowadays, more and more data is being generated
  $\rightarrow$ classical algorithms less efficient [1]

Introduction to Quantum Machine Learning

Machine Learning

- Training machines to learn from the algorithms implemented to handle the data [1]
- Classical machine learning helps to classify images, recognize patterns and speech, handles big data and many more [1]
- Nowadays, more and more data is being generated
  $\rightarrow$ classical algorithms less efficient [1]

$\Longrightarrow$ **need to find alternative methods**

Introduction to Quantum Machine Learning
Quantum Machine Learning

- Quantum Machine Learning (QML) is the intersection between quantum computing and machine learning
- **Motivation**:
  - QML is expected to speed-up the performance of ML programs though exploitation of quantum mechanical properties, i.e. entanglement, superposition,
  - Quantum speed-up in supervised machine learning has recently been shown by researchers of *IBM Quantum* and *University of California, Berkley* [2]
- **How does QML work in general ?**
  - QML integrates quantum algorithms within machine learning programs
    $\Longrightarrow$ data can be classified, sorted and analyzed using quantum algorithms on a quantum computer

Outline

Linear Classification

Intro

- Linear Classification, i.e. Perceptron, SVM, seeks to find an optimal separating hyperplane between two classes of data in a dataset such that, with high probability, all training examples of one class are found only on one side of the hyperplane [3].

Linear Classification
Intro

- Linear Classification, i.e. Perceptron, SVM, seeks to find an optimal separating hyperplane between two classes of data in a dataset such that, with high probability, all training examples of one class are found only on one side of the hyperplane [3].

Setup:

- Input vector $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}\ x_i \in \mathbb{R}^D$
- Labels: $\mathbf{y} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}\ y_i \in \mathcal{C}$
- Classes: $C = \{1, ..., C\}$

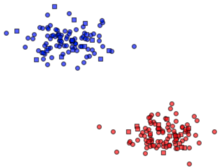


Figure 1 Linearly Separable Data

Linear Classification
Intro

Setup:

- Input vector $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}\ x_i \in \mathbb{R}^D$
- Labels: $\mathbf{y} = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}\ y_i \in \mathcal{C}$
- Classes: $C = \{1, ..., C\}$

Find

- $f(\cdot) : \mathbb{R}^D \rightarrow \mathcal{C}$
- in the case of a linear decision function:
  $y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + \mathbf{w}_0$
- famous perceptron algorithm
  $y_i(\mathbf{w}^T \phi(x_i) + \mathbf{w}_0) > 0$

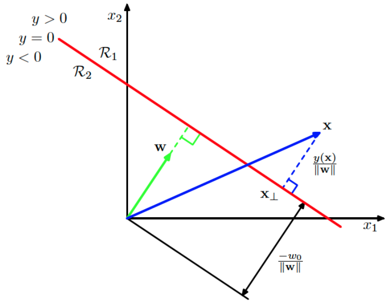


Figure 1 Illustration of Hyperplane Decision Function

Linear Classification
Limitations of LDA

1. no uncertainty measure
2. hard to optimize
3. poor generalization
4. can't handle noisy data

Linear Classification
Limitations of LDA

1. no uncertainty measure
2. hard to optimize
3. poor generalization
4. can’t handle noisy data

Extension: Introduce variables that maximize the margin of the hyperplane between the classes.

Support Vectors
Hard Margin - Introduction

- Add two hyperplanes to the decision function that are parallel to the decision function.
- Extend the additional hyperplanes until the first datapoint of a data cluster is reached.
- The distance between the two the support hyperplanes is called the margin.

$$\mathcal{M} = \frac{2s}{||w||}$$

- A datapoint that is on the hyperplane is called a Support Vector.

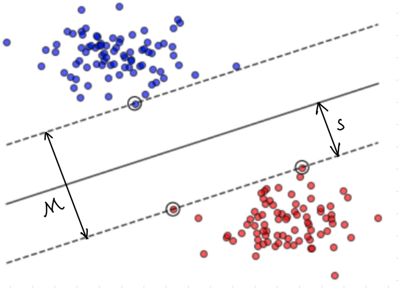


Figure 2 Illustration of Margin and Support Vectors

Support Vectors

Hard Margin - Optimization Constraints

This setup leads to the following constraints, ($s = 1$):

$$w^T x + b \geq +1 \text{ if } y = 1 \tag{1}$$

$$w^T x + b \geq -1 \text{ if } y = -1 \tag{2}$$

$\implies$ maximize the margin $\mathcal{M}$ (for mathematical convenience we minimize $\mathcal{M} = \frac{1}{2}||w||_2^2$) given the above constraints

$$\begin{aligned} &\min f_0(\theta) \\ &s.t.\ \ f_i(\theta) \geq 0 \text{ for } i = 1, ..., N \end{aligned} \tag{3}$$

Support Vectors

Hard Margin - Optimization Constraints

This setup leads to the following constraints, $(s = 1)$:

$$w^T x + b \geq +1 \text{ if } y = 1 \quad (1)$$

$$w^T x + b \geq -1 \text{ if } y = -1 \quad (2)$$

$\implies$ maximize the margin $\mathcal{M}$ (for mathematical convenience we minimize $\mathcal{M} = \frac{1}{2}||w||_2^2$) given the above constraints

$$\begin{aligned} &\min \frac{1}{2}||w||_2^2 \\ &s.t.\ y_i(w^T x_i + b - 1) \geq 0 \text{ for } i = 1, ..., N \end{aligned} \quad (3)$$

Support Vectors
Hard Margin - Quadratic Programming

This formulated optimization problem is a *quadratic programming* problem which in Python can be solved with the CVXOPT library.

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

$\mathbf{q} \rightarrow N \times 1$ vector

$\mathbf{P} \rightarrow N \times N$ symmetric matrix

$\mathbf{G} \rightarrow N \times N$ diagonal matrix

$\mathbf{h} \rightarrow N \times 1$ vector

$\mathbf{A} \rightarrow M \times M$ matrix

$\mathbf{b} \rightarrow M \times 1$ vector

Support Vectors

Hard Margin - Lagrangian

In order to solve the optimization problem, eq. (3), we introduce the Lagrangian

$$\mathcal{L}(\theta, \alpha) = f_0(\theta) - \alpha_i f_i(\theta) \tag{4}$$

and the Duality perspective. The Duality principle in optimization theory states that there are two perspectives of approaching an optimization problem. The two approaches are referred to as **Primal** and **Dual problem** [4].

Support Vectors

Hard Margin - Lagrangian

In order to solve the optimization problem, eq. (3), we introduce the Lagrangian and the Duality perspective.

Primal Problem: solves the lower bound of the constrained optimization ($f_0(\theta^*) = p^*$)

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}w^T w - \sum_{i=1}^{N} \alpha_i (y_i(w^T x_i + b) - 1) \tag{5}$$

$$g(\alpha) = \min_{w,b} \mathcal{L}(w, b, \alpha) \tag{6}$$

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{N} \alpha_i y_i x_i \stackrel{!}{=} 0 \tag{7}$$

$$\nabla_b \mathcal{L}(w, b, \alpha) = -\sum_{i=1}^{N} \alpha_i y_i \stackrel{!}{=} 0 \tag{8}$$

Support Vectors
Hard Margin - Lagrangian

Dual Problem: the best lowest bound to the solution of the primal problem ($g(\alpha^*) = d^*$)
$\rightarrow$ substitute eq. (7) and eq. (8) back into the Langrangian, $\mathcal{L}(w^*, b^*, \alpha)$.

$$g(\alpha) = \mathcal{L}(w^*, b^*, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j \tag{9}$$

From here we can set up a new constraint optimization problem:

$$\begin{aligned} &\max_{\alpha} \; g(\alpha) \\ s.t. \; &\sum_{i=1}^{N} \alpha_i y_i = 0, \;\; \alpha_i \geq 0 \text{ for } i = 1, ..., N \end{aligned} \tag{10}$$

Support Vectors
Hard Margin - Implementation

- With the constraint optimization problem eq. (10) we can formulate a quadratic *programming problem* that can be implemented in Python with the CVXOPT library

$$\begin{aligned} &\max_{\alpha}\ \boldsymbol{\alpha}\mathbf{1}_N - \frac{1}{2}\alpha^T \mathbf{Q}\alpha \quad \| \text{ where } \mathbf{Q} = \mathbf{y}\mathbf{y}^T \circ \mathbf{x}^T\mathbf{x} \\ &s.t.\ \sum_{i=1}^{N} \alpha_i y_i = 0,\ \ \alpha_i \geq 0 \text{ for } i = 1, ..., N \end{aligned} \tag{11}$$

Support Vectors
Hard Margin - Implementation

- With the constraint optimization problem eq. (10) we can formulate a quadratic *programming problem* that can be implemented in Python with the CVXOPT library

$$\min_{\alpha} \frac{1}{2}\alpha^T \mathbf{P}\alpha - \mathbf{1}_N^T \boldsymbol{\alpha}$$
$$s.t.\ y^T\alpha = 0,\ \ -\alpha_i \leq 0 \text{ for } i = 1, ..., N \tag{11}$$

$$\mathbf{q} := -\mathbf{1} \in \mathbb{R}^{Nx1}$$
$$\mathbf{P} := \mathbf{Q} \in \mathbb{R}^{NxN}$$
$$\mathbf{G} := -\mathbf{diag}(\mathbf{1}) \in \mathbb{R}^{NxN}$$
$$\mathbf{h} := \mathbf{0} \in \mathbb{R}^{Nx1}$$
$$\mathbf{A} := \mathbf{y} \in \mathbb{R}^{Nx1}$$
$$\mathbf{b} := \mathbf{0} \in \mathbb{R}$$

Support Vectors
Hard Margin - Support Vectors

Duality: We use the duality gap between the primal and dual solution, $p^* - d^* = 0$, to find the Support Vector
$\rightarrow$ strong duality holds if

$$\alpha_i(y_i(w^T x_i + b) - 1) = 0$$

implying that given $\alpha_i > 0$, if

$$y_i(w^T x_i + b) = 1$$

$\implies$ Support Vector

Support Vectors

Hard Margin - Classification

Parameters:

$$w^* = \sum_{i=1}^{N} \alpha_i y_i x_i = (\boldsymbol{\alpha} \cdot \mathbf{y})^T \mathbf{X} \tag{12}$$

$$b = y_i - w^T x_i \tag{13}$$

Classification:

$$\mathbf{y}_{pred} = \text{sign}((w^*)^T \mathbf{x} + b) \tag{14}$$

Support Vectors
Hard Margin - Code

```python
N, D = X.shape

yy = y[:, None] @ y[:, None].T
XX = X @ X.T

# - - - QP solver - - -
P = matrix(yy * XX)
q = matrix(-np.ones((N, 1)))

if self.C is None: # hard margin SVM
    G = matrix((-np.eye(N)))
    h = matrix(np.zeros_like(y))
else:  # soft margin SVM
    G = matrix(np.vstack((-np.eye(N), np.eye(N))))
    h = matrix(np.hstack((np.zeros_like(y), self.C*np.ones(N))))

A = matrix(y.reshape(1,-1))
b = matrix(np.zeros(1))

solvers.options['show_progress'] = False
solution = solvers.qp(P, q, G, h, A, b)
# - - - QP solver - - -
```

Figure 3 Code snip-it of the quadratic programming implementation of SVM using CVXOPT

```python
# lagrangian multipliers
self.alphas = np.ravel(solution['x'])

# find the instances where the langrangian multipliers are non-zero
is_sv = (self.alphas > self.alpha_tol).flatten()

self.sv_alphas = self.alphas[is_sv]
self.sv_X = X[is_sv]
self.sv_y = y[is_sv]

# weights
self.w = np.einsum('i,i,ij', self.sv_alphas.flatten(), self.sv_y, self.sv_X)

# bias
biases = y[is_sv] - np.dot(X[is_sv, :], self.w)
self.b = np.sum(self.sv_alphas*biases) / np.sum(self.sv_alphas)
```

Figure 4 Code snip-it of the parameter calculations and support vector determination

Support Vectors

Hard Margin - Example Plot

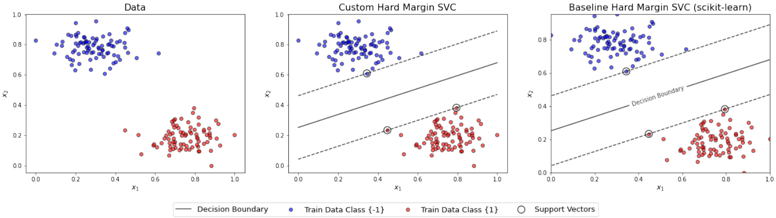


Figure 5 Custom Hard Margin SVC versus scikit-learn linear SVC

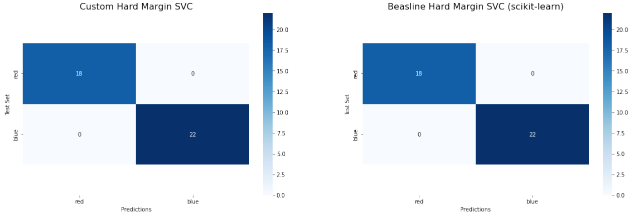


Figure 6 Confusion Matrices for custom hard margin SVC and SCIKIT baseline gard margin SVC

Support Vectors
Soft Margin

- Hard Margin classification assumes that the data is linearly separable, i.e. does not overlap.
- Unrealistic in real world problems
- **Extend** the previous Support Vector method to allow misclassification

Support Vectors
Soft Margin

- Hard Margin classification assumes that the data is linearly separable, i.e. does not overlap.
- Unrealistic in real world problems
- Extend the previous Support Vector method to allow misclassification

- Introduce *slack variables* $\xi_i \geq 0$ which measure the violation of the margin (in units of $||w||$)

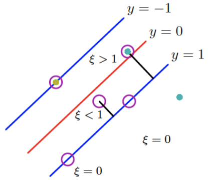


Figure 7 Depiction of Slack Variables ξ

Support Vectors
Soft Margin

$$\begin{aligned} &\min f_0(\theta) \\ &s.t.\ \ f_i(\theta) \geq 0 \text{ for } i = 1, ..., N \end{aligned} \tag{15}$$

Support Vectors
Soft Margin

$$\begin{aligned} &\min \ \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\xi_i \\ &s.t. \ y_i(w^T x_i + b - 1) \geq 1 - \xi_i \ \forall_i \end{aligned} \tag{15}$$

Support Vectors
Soft Margin

$$
\begin{aligned}
&\min \ \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\xi_i \\
&s.t.\ y_i(w^T x_i + b - 1) \geq 1 - \xi_i \ \forall_i
\end{aligned}
\tag{15}
$$

Repeat the procedure:

1. Calculate the primal
2. Calculate the dual
3. Use duality to determine support vectors and calculate decision function to perform the classification

Support Vectors
Soft Margin - Primal

$$\mathcal{L}(w, b, \xi, \alpha, \mu) = \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}(\alpha_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{N}\mu_i\xi_i \tag{16}$$

$$g(\alpha) = \min_{w,b,\xi} \mathcal{L}(w, b, \xi, \alpha, \mu) \tag{17}$$

- KKT conditions:

$$\alpha_i \geq 0 \tag{18}$$

$$y_i(w^T x_i + b) - 1 + \xi_i \geq 0 \tag{19}$$

$$\alpha_i(y_i(w^T x_i + b) - 1 + \xi_i) = 0 \tag{20}$$

$$\mu_i \geq 0 \tag{21}$$

$$\xi_i \geq 0 \tag{22}$$

$$\mu_i\xi_i = 0 \tag{23}$$

- determine w^*, b^* and ξ^*

Support Vectors
Soft Margin - Dual

$$g(\alpha) = \mathcal{L}(w^*, b^*, \xi^*, \alpha) = \sum_{i=0}^{N} \alpha_i - \frac{1}{2} \sum_{i=0}^{N} \sum_{j=0}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j \tag{24}$$

$$\max \;\; g(\alpha) \tag{25}$$

$$s.t. \;\; 0 \leq \alpha_i \leq C \tag{26}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \quad \text{for } i = 1, .., N \tag{27}$$

- similarly as we have seen in eq. (11), we reformulate eq. (24) as a *quadratic programming problem*

Support Vectors

Soft Margin - Quadratic Programming

$$\min_{\alpha} \frac{1}{2}\alpha^T \mathbf{P} \alpha - \mathbf{1}_N^T \boldsymbol{\alpha} \tag{28}$$

$$s.t.\ y^T \alpha = 0 \tag{29}$$

$$-\alpha_i \leq 0 \text{ for } i = 1, ..., N \tag{30}$$

$$\alpha_i \leq C \text{ for } i = 1, ..., N \tag{31}$$

$$\mathbf{q} := -\mathbf{1_N} \in \mathbb{R}^{Nx1}$$

$$\mathbf{P} := \mathbf{Q} \in \mathbb{R}^{NxN}$$

$$\mathbf{G} := -\begin{pmatrix} -\mathbf{diag}(\mathbf{1_N}) \\ \mathbf{diag}(\mathbf{1_N}) \end{pmatrix} \in \mathbb{R}^{2NxN}$$

$$\mathbf{h} := \begin{pmatrix} \mathbf{0_N} & C \cdot \mathbf{1_N} \end{pmatrix} \in \mathbb{R}^{2N \times 1}$$

$$\mathbf{A} := \mathbf{y} \in \mathbb{R}^{Nx1}$$

$$\mathbf{b} := \mathbf{0} \in \mathbb{R}$$

Support Vectors
Soft Margin - Code Snipit

```python
N, D = X.shape

yy = y[:, None] @ y[:, None].T
XX = X @ X.T

# - - - QP solver - - -
P = matrix(yy * XX)
q = matrix(-np.ones((N, 1)))

if self.C is None: # hard margin SVM
    G = matrix((-np.eye(N)))
    h = matrix(np.zeros_like(y))
else:  # soft margin SVM
    G = matrix(np.vstack((-np.eye(N), np.eye(N))))
    h = matrix(np.hstack((np.zeros_like(y), self.C*np.ones(N))))

A = matrix(y.reshape(1,-1))
b = matrix(np.zeros(1))

solvers.options['show_progress'] = False
solution = solvers.qp(P, q, G, h, A, b)
# - - - QP solver - - -
```

Figure 8 Code snip-it of the quadratic programming implementation of SVM using CVXOPT

```python
# lagrangian multipliers
self.alphas = np.ravel(solution['x'])

# find the instances where the langrangian multipliers are non-zero
is_sv = (self.alphas > self.alpha_tol).flatten()

self.sv_alphas = self.alphas[is_sv]
self.sv_X = X[is_sv]
self.sv_y = y[is_sv]

# weights
self.w = np.einsum('i,i,ij', self.sv_alphas.flatten(), self.sv_y, self.sv_X)

# bias
biases = y[is_sv] - np.dot(X[is_sv, :], self.w)
self.b = np.sum(self.sv_alphas*biases) / np.sum(self.sv_alphas)
```

Figure 9 Code snip-it of the parameter calculations and support vector determination

Support Vectors

Soft Margin - Example Plot

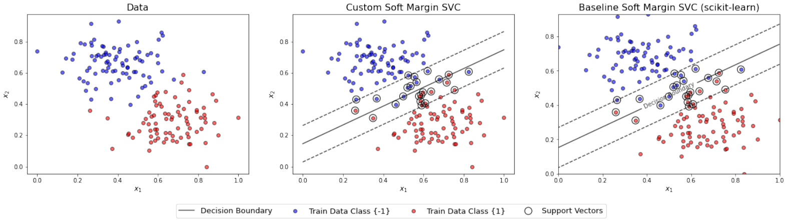


Figure 10 Classification result of the custom soft margin SVC and SCIKIT baseline soft SVC

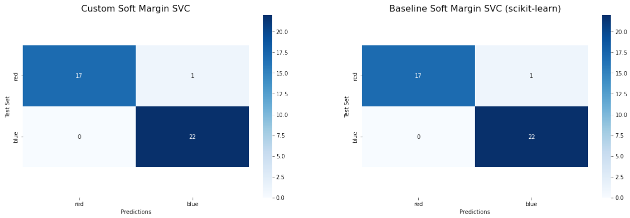


Figure 11 Confusion Matrices for custom soft margin SVC and SCIKIT baseline soft margin SVC

Kernel Methods

Hard Margin - Example Plot

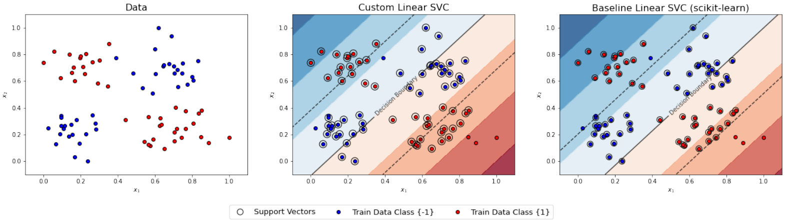


Figure 12 Displays the classification result of the custom soft margin SVC and SCIKIT baseline soft SVC

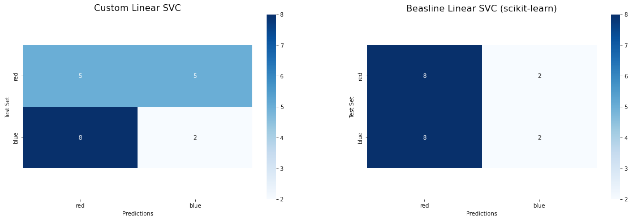


Figure 13 Confusion Matrices for custom soft margin SVC and SCIKIT baseline soft margin SVC

Kernel Methods

- So far, we still considered the data to be linearly separable in a feature space $\mathbf{X}$.
- **Extend** the Support Vector Machines for higher-dimensional data or non-linear data, we can further generalize the previous approaches by mapping d-dimensional data vectors into an n-dimensional feature space:

$$\phi : \mathbf{X} \rightarrow \mathcal{R}^n$$

- Further, we make us of the *kernel trick*:

$$-\frac{1}{2}\sum_{i=0}^{N}\sum_{j=0}^{N}\alpha_i\alpha_j y_i y_j \phi(x_i)^T\phi(x_j)$$

$$-\frac{1}{2}\sum_{i=0}^{N}\sum_{j=0}^{N}\alpha_i\alpha_j y_i y_j k(x_i, x_j), \quad \text{where } k(x_i, x_j) = \phi(x_i)^T\phi(x_j)$$

$\rightarrow$ Increase computational speed for higher dimensional finite data

Kernel Methods

- So far, we still considered the data to be linearly separable in a feature space $\mathbf{X}$.
- **Extend** the Support Vector Machines for higher-dimensional data or non-linear data, we can further generalize the previous approaches by mapping d-dimensional data vectors into an n-dimensional feature space:

$$\phi : \mathbf{X} \rightarrow \mathcal{R}^n$$

- Further, we make us of the *kernel trick*: therefore eq. (24) can be written as

$$g(\alpha) = \sum_{i=0}^{N} \alpha_i - \frac{1}{2} \sum_{i=0}^{N} \sum_{j=0}^{N} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \tag{32}$$

- Kernels can be seen as a measure of similarity

Kernel Methods

Linear:

$$k(x_1, x_2) = x_1^T x_2$$

Polynominal:

$$k(x_1, x_2) = (c + x_1^T x_2)^d$$

Radial Basis Function:

$$k(x_1, x_2) = \exp(-\gamma ||x_1 - x_2||^2)$$

Sigmoid:

$$k(x_1, x_2) = \tanh(\gamma x_1^T x_2 + c)$$

classification: if the kernel is not linear, the classification function eq. (14) has to be adopted

$$y_{pred} = \sum_{i \in SV} \alpha_i y_i k(x, x_i) \tag{33}$$

Kernel Methods

Code - Snipit

```python
N, D = X.shape

yy = y[:, None] @ y[:, None].T

# Gram Matrix
if self.kernel == 'quantum':
    if self.verbose: print('Computing Quantum Kernel ...')
    K = self.qk(X)
    if self.verbose:  print('Quantum Kernel computed!')
else:
    K = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
            K[i,j] = self.kernel_func(X[i], X[j])

self.gramMatrix = K

# QP solver
P = matrix(yy * K)
q = matrix(-np.ones((N, 1)))

if self.C is None: # hard SVM
    G = matrix((-np.eye(N)))
    h = matrix(np.zeros_like(y))
else:  # soft SVM
    G = matrix(np.vstack((-np.eye(N), np.eye(N))))
    h = matrix(np.hstack((np.zeros_like(y), self.C*np.ones(N))))

A = matrix(y.reshape(1,-1))
b = matrix(np.zeros(1))

# QP solver
solvers.options['show_progress'] = False
solution = solvers.qp(P, q, G, h, A, b)
```

Figure 14 Code snip-it of the quadratic programming implementation of SVM with the kernel trick using CVXOPT

```python
# lagrangian multipliers
alphas = np.ravel(solution['x'])

# find the instances where the langrangian multipliers are non-zero
is_sv = alphas > self.alpha_tol
sv_ind = np.arange(len(alphas))[is_sv]
self.alphas = alphas[is_sv]
self.sv_X = X[is_sv]
self.sv_y = y[is_sv]

# bias
self.b = 0
for i in range(len(self.alphas)):
    self.b += self.sv_y[i] # sum of all alphas
    self.b -= np.sum(self.alphas * self.sv_y * self.gramMatrix[sv_ind[i], is_sv]) # sum per row
self.b /= len(self.alphas) # divided by alphas

# Compute w only if the kernel is linear
if self.kernel == linear_kernel:
    self.w = np.einsum('i,i,ij', self.alphas, self.sv_y, self.sv_X)
else:
    self.w = None
```

Figure 15 Code snip-it of the parameter calculations and support vector determination using the kernel trick

Github-nonlinear- classifier

Kernel Methods

Kernel Method - Example Plot

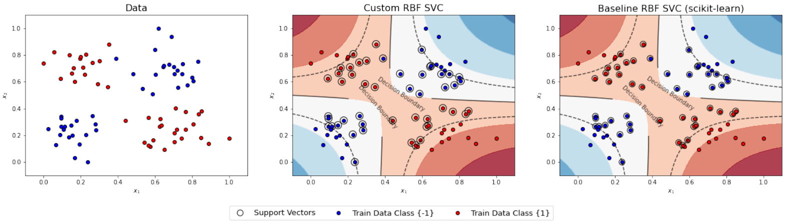


Figure 16 Displays the classification result of the custom RBF SVC and SCIKIT baseline RBF SVC

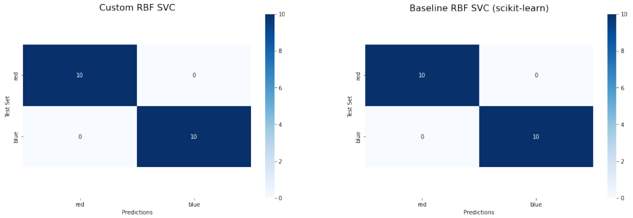


Figure 17 Confusion Matrices for custom RBF SVC and SCIKIT baseline RBF SVC

Limitations to SVM

- Feature space becomes large $\implies$ kernel functions becomes computationally expensive
- No probabilistic interpretation of the classification
- Not suitable for large datasets

Outline

Quantum Kernel Estimation

Introduction

- Idea: use the quantum advantage to speed-up the computational speed of support vector classifiers (potential use of the exponentially large quantum state space)
- two methods:
  1. Quantum Variational Classifier
  2. Quantum Kernel Estimation

Quantum Kernel Estimation

Introduction

- Idea: use the quantum advantage to speed-up the computational speed of support vector classifiers (potential use of the exponentially large quantum state space)
- two methods:
  1. Quantum Variational Classifier
  2. Quantum Kernel Estimation
- In order to make use of the quantum advantage, the classical data needs to be transformed into the quantum state space
  - requires a data map (encoding function)
  - requires a quantum feature map as a parameterized circuit
- From the quantum state space we can estimate a kernel matrix that can be used with a classical Support Vector Classifier

Quantum Kernel Estimation

Quantum Feature Map

- transforms low dimensional real space onto high dimensional quantum state space [5]

$$\Phi : \mathbf{x} \in \Omega \rightarrow |\Phi(x)\rangle \langle \Phi(x)| \; [6] \tag{34}$$

- This is facilitated by a unitary operator $\mathcal{U}_{\Phi(x)}$ on a initial state $|0\rangle^n$ with n=number of qubits [7]

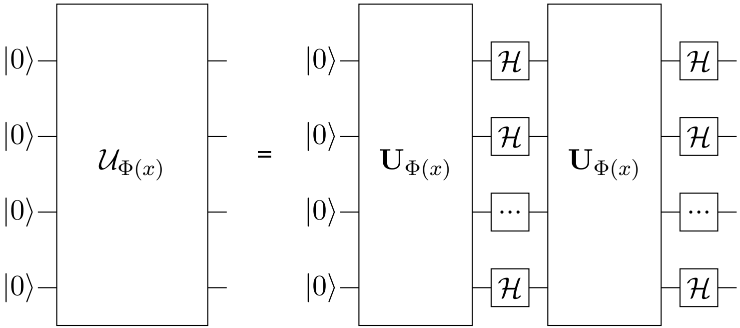
$$\Phi(x) = \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n} \tag{35}$$

$$\mathcal{U}_{\Phi(x)} = \prod_d \mathbf{U}_{\Phi(x)} \mathcal{H}^{\otimes n} \tag{36}$$

- Dimensions of the Feature Space have to align with the number of qubits
- Feature Maps have a high influence on the classification accuracy, as they are the basis for the kernel estimation $\implies$ careful analysis of the feature space is necessary [5]

Quantum Kernel Estimation

Quantum Feature Map - Quantum Circuit



Quantum Kernel Estimation
Pauli Feature Map

- The Pauli-Feature Map is a customizable example of a Quantum Feature Map where the unitary matrix, $\mathbf{U}_{\Phi(x)}$ from eq. (36)

$$\mathbf{U}_{\Phi(x)} = \exp\left\{ i \sum_{S \subseteq [n]} \phi_S(x) \prod_{k \in S} P_i \right\} \tag{37}$$

  where
  - $P_i \in$ {I, Pauli-X, Pauli-Y, Pauli-Z}
  - $S \in \{ \binom{n}{k}$ combinations, $k = 1, ..., n \}$
  - $\phi_S \rightarrow$ data mapping function (encoding function)
- Gives rise the the Z-Feature-Map and the ZZ-Feature Map

Quantum Kernel Estimation
Data Mapping functions

- Standard Function used in QISKIT

$$\phi_S : x \to \begin{cases} x_i & \text{if } S = \{i\} \\ (x_i - \pi)(x_j - \pi) & \text{if } S = \{i, j\} \end{cases}$$

- further examples [5]

$$\phi_S : x \to \begin{cases} x_i & \text{if } S = \{i\} \\ \exp\left(\frac{|x_i - x_j|^2}{8/\ln(\pi)}\right) & \text{if } S = \{i, j\} \end{cases} \tag{38}$$

$$\phi_S : x \to \begin{cases} x_i & \text{if } S = \{i\} \\ \frac{\pi}{3 \cos x_i \cos x_j} & \text{if } S = \{i, j\} \end{cases} \tag{39}$$

Quantum Kernel Estimation
Quantum Feature Map - Example

- Given eq. (37), if $k = 2, P_0 = Z$ and $P_1 = ZZ \implies$ ZZ-Feature Map

$$\mathbf{U}_{\Phi(x)} = \exp\left\{\left(i\sum_{jk}\phi_S(j,k)Z_j \otimes Z_k\right)\left(i\sum_{j}\phi_S(j)Z_j\right)\right\} \quad (40)$$

$$\mathcal{U}_{\Phi(x)} = \left(\exp\left\{\left(i\sum_{jk}\phi_S(j,k)Z_j \otimes Z_k\right)\left(i\sum_{j}\phi_S(j)Z_j\right)\right\}\mathcal{H}^{\otimes n}\right)^d$$

$$\mathcal{U}_{\Phi(x)} = \left(\exp\left(ix_0Z_0 + ix_1Z_1 + i(x_0-\pi)(x_1-\pi)Z_0Z_1\right)\mathcal{H}^{\otimes n}\right)^d \quad (41)$$

Quantum Kernel Estimation
Quantum Kernel

- Quantum Feature Maps $\Phi(x)$ naturally give rise to quantum kernels

$$k(x_i, x_j) = \phi(x_i)^T \phi(x_j) \tag{42}$$

- As the kernel entries are the fidelities between two feature vectors, we need to establish a way to estimate the fidelities of a quantum state
- For finite data, this scan be achieved by estimating the transition amplitude [6]:

$$K_{ij} = |\langle \Phi(x_i) | \Phi(x_j) \rangle|^2 \tag{43}$$

- plugging in eq. (35) into eq. (43)

$$K_{ij} = |\langle 0|^{\otimes n} \mathcal{U}^T_{\Phi(x)} \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n}|^2 \tag{44}$$

Quantum Kernel Estimation
Quantum Kernel

- Quantum Feature Maps $\Phi(x)$ naturally give rise to quantum kernels

$$k(x_i, x_j) = \phi(x_i)^T \phi(x_j) \tag{42}$$

- As the kernel entries are the fidelities between two feature vectors, we need to establish a way to estimate the fidelities of a quantum state
- For finite data, this scan be achieved by estimating the transition amplitude [6]:

$$K_{ij} = |\langle \Phi(x_i) | \Phi(x_j) \rangle|^2 \tag{43}$$

- plugging in eq. (35) into eq. (43)

$$K_{ij} = |\langle 0|^{\otimes n} \mathcal{U}_{\Phi(x)}^T \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n}|^2 \tag{44}$$

$\implies$ Quantum Kernel Matrix Estimate

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation

Implementation of Quantum Support Vector Machines
Quantum Kernel Estimation

1. **Built a parameterized quantum circuit that emulates a Quantum Feature Map for each data pair**
   example: ZZ-Feature Map for 2-dimensional input
   recall: eq. (41)

$$\mathcal{U}_{\Phi(x)} = \left(\exp\left(ix_0 Z_0 + ix_1 Z_1 + i(x_0 - \pi)(x_1 - \pi) Z_0 Z_1\right) \mathcal{H}^{\otimes n}\right)^d$$

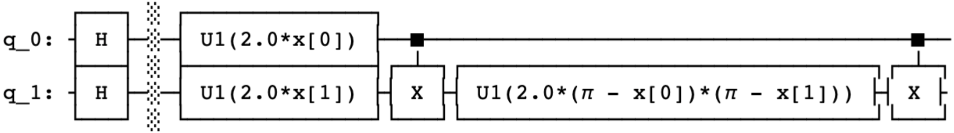


Figure 18 ZZ-Feature Map Circuit

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation

1. Built a parameterized quantum circuit that emulates a Quantum Feature Map for each data pair
2. **Construct the Quantum Kernel circuits for each data pair**
   recall: eq. (44) and eq. (41)

$$K_{ij} = |\langle 0|^{\otimes n} \mathcal{U}^T_{\Phi(x)} \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n}|^2$$

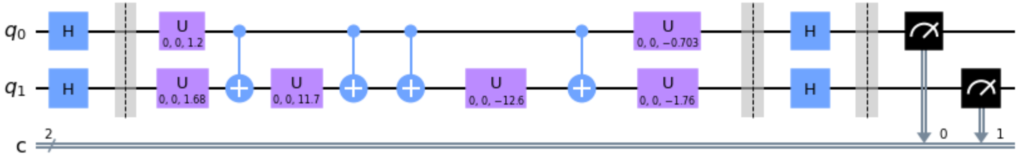


Figure 18 Parameterized Quantum Kernel Circuit

Implementation of Quantum Support Vector Machines
Quantum Kernel Estimation

1. Built a parameterized quantum circuit that emulates a Quantum Feature Map for each data pair
2. Construct the Quantum Kernel circuits for each data pair
3. **Measure the number of all zero strings** 0^n

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation

1. Built a parameterized quantum circuit that emulates a Quantum Feature Map each data pair
2. Construct the Quantum Kernel circuits for each data pair
3. Measure the number of all zero strings 0^n
4. **Calculate the frequency of the zero strings to find the transition probability $\Longrightarrow$ kernel entry of the the Quantum Kernel**

Note:** Step 3. and 4. computes the kernel values from the results of the inner products based of the measurements of the quantum circuits created for the quantum kernel estimate.

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation

1. Built a parameterized quantum circuit that emulates a Quantum Feature Map each data pair
2. Construct the Quantum Kernel circuits for each data pair
3. Measure the number of all zero strings 0^n
4. Calculate the frequency of the zero strings to find the transition probability $\implies$ Kernel entry of the the quantum kernel

example:

- ☐ given a dataset $\mathbf{X} \in \mathbb{R}^{10\times 2}$
- ☐ build parameterized kernel circuit from parameterized feature map circuits for each data pair $\implies$ 100 circuits

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation - Code Snipit

```python
class ZZFeatureMap:
    def __init__(self, n_qubits, reps, data_map, insert_barriers=False) -> None:
        self.n_qubits = n_qubits
        self.data_map = data_map
        self.reps = reps
        self.insert_barriers = insert_barriers
        self._circuit = None

    def map(self, data, reverse=False):
        circuit = QuantumCircuit(self.n_qubits)
        for i in range(self.reps):
            if i > 0:
                if self.insert_barriers: circuit.barrier()
            circuit.h(0)
            circuit.h(1)
            if self.insert_barriers: circuit.barrier()
            if not reverse:
                circuit.u(0,0,2*self.data_map.map(data[:1]),0)
                circuit.u(0,0,2*self.data_map.map(data[1:]),1)
            else:
                circuit.u(0,0,-2*self.data_map.map(data[:1]),0)
                circuit.u(0,0,-2*self.data_map.map(data[1:]),1)
            circuit.cx(0,1)
            if not reverse:
                circuit.u(0,0,2*self.data_map.map(data),1)
            else:
                circuit.u(0,0,-2*self.data_map.map(data),1)
            circuit.cx(0,1)

        if not reverse:
            return circuit.to_instruction()
        else:
            return circuit.to_instruction().reverse_ops()

    def __repr__(self) -> str:
        return f"ZZFeatureMap(feature_dimensions={self.n_qubits}, reps={self.reps})"
```

Figure 18 ZZ-Feature Map as parameterized circuit

github-quantum-feature-map

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation - Code Snipit

```python
def construct_circuit(self, X1, X2):
    circuit = QuantumCircuit(self.n_qubits, self.n_qubits)
    if self._statevector_sim: # statevector simulator
        raise BackendError
    else:
        instruction= self._feature_map.map(X1, reverse=False)
        instruction_re = self._feature_map.map(X2, reverse=True)
        circuit.append(instruction, [0,1])
        circuit.append(instruction_re, [0,1])
    circuit.barrier()
    circuit.measure([i for i in range(self.n_qubits)], [i for i in range(self.n_qubits)])

    return circuit
```

(a) Function that constructs quantum kernel circuits for each data point given a Feature Map

```python
def __compute_kernel_val(self, idx, job, measurement_basis):
    """
    Computes the kernel values form the results of the inner products.
    """
    if self._statevector_sim:
        raise BackendError
    else:
        result = job.result().get_counts(idx)

        kernel_value = result.get(measurement_basis, 0) / sum(result.values())
    return kernel_value
```

(b) Function that computes the quantum kernel values from the results of the inner products

Figure 19 Code Snipits of helper functions used to compute the Quantum Kernel

github-quantum-kernel-estimation

Implementation of Quantum Support Vector Machines

Quantum Kernel Estimation - Code Snipit

```
N, D = x_vec.shape
circuits = []
for i in range(N):
    for j in range(N):
        circuits.append(self.construct_circuit(x_vec[i], x_vec[j]))

k_values = []
# calculate the inner products via the unintary operator
job = execute(circuits, self._quantum_backend, shots=self.sim_params['shots'],
                seed_simulator=self.sim_params['seed'], see_transpiler=self.sim_params['seed'])
# get the results
for j in range(len(circuits)):
    # calculate the kernel values
    k_values.append(self.__compute_kernel_val(j, job, measurement_basis))

kernel = np.array(k_values).reshape(x_vec.shape[0], x_vec.shape[0])
```

Figure 20 Quantum Kernel Function implemented in Python using QISKIT quantum circuits

Implementation of Quantum Support Vector Machines

Quantum SVC

How is the Quantum Kernel embedded into the SVM protocol?

Implementation of Quantum Support Vector Machines

Quantum SVC

How is the Quantum Kernel embedded into the SVM protocol?

- instead of using a classical kernel in the constraint optimization problem, eq. (25), just insert the quantum kernel estimate

$$K_{ij} = |\langle 0|^{\otimes n} \mathcal{U}^{T}_{\Phi(x)} \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n}|^2$$

Implementation of Quantum Support Vector Machines
Quantum SVC

How is the Quantum Kernel embedded into the SVM protocol?

- instead of using a classical kernel in the constraint optimization problem, eq. (25), just insert the quantum kernel estimate

$$K_{ij} = |\langle 0|^{\otimes n} \mathcal{U}^T_{\Phi(x)} \mathcal{U}_{\Phi(x)} |0\rangle^{\otimes n}|^2$$

- now we have a **Quantum Support Vector Machine**

Outline

Results

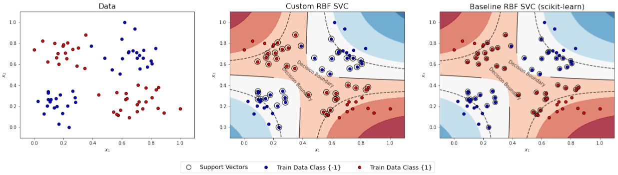
Benchmark SVC

	Linear	Polynomoinal	Sigmoid	RBF C=10	RBF C=100
Linearly Separately Data	0.975	0.975	0.975	0.975	0.975
XOR Data	0.350	0.950	0.400	1.000	1.000
Circles Data	0.400	1.000	0.275	1.000	1.000
Moons Data	0.750	0.750	0.750	0.750	0.850
Adhoc Data	0.300	0.250	0.300	0.500	0.550

(a) Test accuracy Custom SVC

	Linear	Polynomoinal	Sigmoid	RBF C=10	RBF C=100
Linearly Separately Data	0.975	0.975	0.95	0.975	0.975
XOR Data	0.500	0.950	0.35	1.000	1.000
Circles Data	0.400	0.800	0.30	1.000	1.000
Moons Data	0.750	0.725	0.75	0.725	0.875
Adhoc Data	0.350	0.300	0.40	0.500	0.550

(b) Test accuracy benchmark SVC



(c) Example plot on XOR data

Figure 21 Benchmark SVC

Results

Benchmark Quantum Kernel

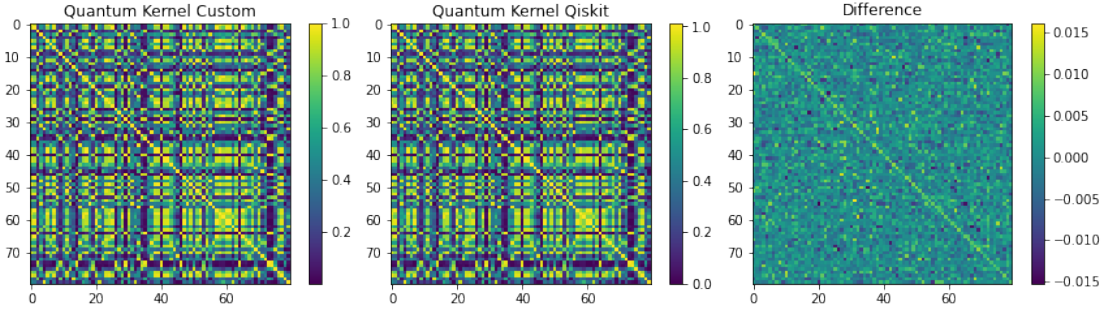


Figure 22 Comparison of Custom Quantum Kernel and QISKIT Quantum Kernel

Results

Benchmark Quantum SVC

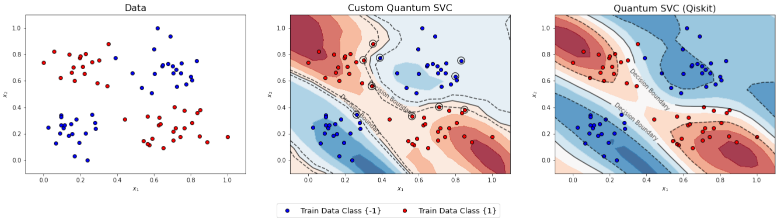


Figure 23 Comparison of Custom Quantum SVC and QISKIT baseline Quantum SVC

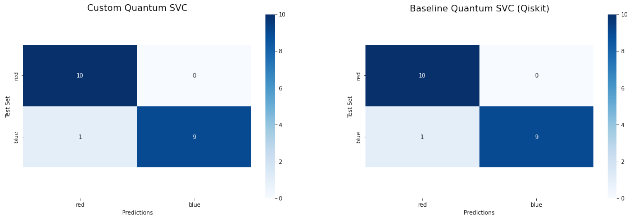


Figure 24 Confusion Matrices for custom Quantum SVC and SCIKIT baseline Quantum SVC

Results

Comparison of Different Data Maps

	Default Data Map	Exp Data Map	Sin Data Map	Cos Data Map
XOR Data	0.9375	1.000	1.0000	1.0000
Cirles Data	1.0000	1.000	1.0000	1.0000
Moons Data	0.6875	0.625	0.8125	0.8125
Adhoc Data	1.0000	0.500	0.7500	0.7500

Figure 25 Test accuracy of different data maps on different data-sets

Results
Comparison of Feature Maps

Still need to compute

Results

Comparison of a quantum kernel and a rbf kernel on adhoc Data

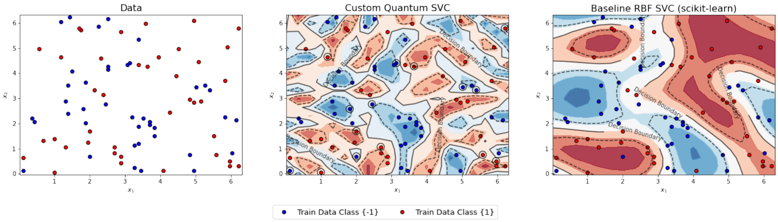


Figure 26 Benchmark on adhoc data-set

	Default Data Map	Exp Data Map	Sin Data Map	Cos Data Map	RBF SVC
XOR Data	0.9375	1.000	1.0000	1.0000	1.00
Cirles Data	1.0000	1.000	1.0000	1.0000	1.00
Moons Data	0.6875	0.625	0.8125	0.8125	0.85
Adhoc Data	1.0000	0.500	0.7500	0.7500	0.55

Figure 27 Test accuracy of different quantum kernels in comparison with an rbf kernel

Results

Comparison of different kernels on different data

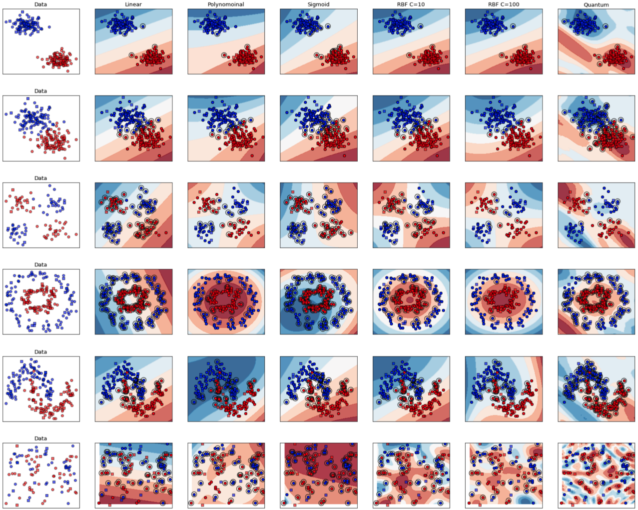


Figure 28 Comparison of different Classification Models

Results
Conclusion

- Implemented a classical Support Vector Machine
- Implemented a Quantum Kernel
- Implemented a Quantum Support Vector Classifier
- Showed that quantum kernels can perform better on high dimensional data

- Potential Improvements:
  - improve efficiency of code
    - remove unnecessary for-loops
    - mirror quantum kernel in estimation as $K_{ij} = K_{ji}$
    - initialize the diagonal elements as 1
  - implement a general Pauli-Feature Map
  - implement a state vector calculation scheme

Outline

References I

N. Mishra, M. Kapil, H. Rakesh, A. Anand, N. Mishra, A. Warke, S. Sarkar, S. Dutta, S. Gupta, A. P. Dash, *et al.*, "Quantum machine learning: A review and current status", *Data Management, Analytics and Innovation*, pp. 101–145, 2021.

Y. Liu, S. Arunachalam, and K. Temme, "A rigorous and robust quantum speed-up in supervised machine learning", *Nature Physics*, vol. 17, no. 9, pp. 1013–1017, 2021.

J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning", *Nature*, vol. 549, no. 7671, pp. 195–202, 2017.

S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

Y. Suzuki, H. Yano, Q. Gao, S. Uno, T. Tanaka, M. Akiyama, and N. Yamamoto, "Analysis and synthesis of feature map for kernel-based quantum classifier", *Quantum Machine Intelligence*, vol. 2, no. 1, pp. 1–9, 2020.

References II

V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta, "Supervised learning with quantum-enhanced feature spaces", *Nature*, vol. 567, no. 7747, pp. 209–212, 2019.

A. Phan. (2021), 2021 qiskit global summer school on quantum machine learning - lab 3: Introduction to quantum kernels and svms, [Online]. Available: `https://learn.qiskit.org/summer-school/2021/lab3-introduction-quantum-kernels-support-vector-machines` (visited on 11/29/2021).