

Qiskit Machine Learning Tutorial

Yuma Nakamura

Qiskit Advocate



Agenda

この2つのチュートリアルを解説していきます。

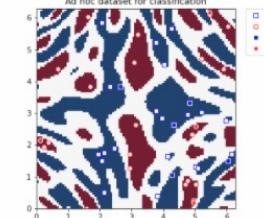
Machine Learning Tutorials



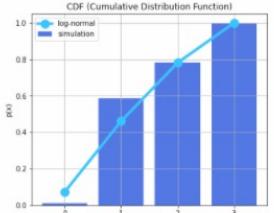
Quantum Neural Networks



*Neural Network Classifier &
Regressor*



Quantum Kernel Machine Learning



*qGANs for Loading Random
Distributions*



Torch Connector

前置き

以前にQSVM, QGANの解説を行ったことがあるので、今回はTutorialの流れに忠実に解説していきます。
Tutorialの行間、コード部分の大変な箇所を理解してもらうことが目的です。

以前の解説(YouTube):

QSVM (量子サポートベクトルマシン)



<https://www.youtube.com/watch?v=tuCaRRHoSdI>

QGAN(量子敵対的生成ネットワーク)



<https://www.youtube.com/watch?v=hcq5Bf6M-yk>

こんな感じでTutorialのスクリプトを貼りながら適宜補足
大事なパラメータなどはハイライト

カーネルとは

カーネルとは2つのデータの「類似度」を定義する関数です(「類似度」の定義は様々)。

学習データ内の全ての組み合わせで「類似度」を計算した行列をカーネル行列と呼び、SVMで使用されています。

量子状態の内積 $|\langle \phi^\dagger(x_j)|\phi(x_i)\rangle|$ でカーネル行列を定義したSVMを、量子カーネル機械学習と呼びます。

Quantum Kernel Machine Learning

The general task of machine learning is to find and study patterns in data. For many datasets, the datapoints are better understood in a higher dimensional feature space, through the use of a kernel function: $k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle$ where k is the kernel function, \vec{x}_i, \vec{x}_j are n dimensional inputs, f is a map from n -dimension to m -dimension space and $\langle a, b \rangle$ denotes the dot product. When considering finite data, a kernel function can be represented as a matrix: $K_{ij} = k(\vec{x}_i, \vec{x}_j)$.

In quantum kernel machine learning, a quantum feature map $\phi(\vec{x})$ is used to map a classical feature vector \vec{x} to a quantum Hilbert space, $|\phi(\vec{x})\rangle\langle\phi(\vec{x})|$, such that $K_{ij} = |\langle\phi(\vec{x}_j)|\phi(\vec{x}_i)\rangle|^2$. See [Supervised learning with quantum enhanced feature spaces](#) for more details.

In this notebook, we use `qiskit` to calculate a kernel matrix using a quantum feature map, then use this kernel matrix in `scikit-learn` classification and clustering algorithms.

```
[1]: import matplotlib.pyplot as plt
import numpy as np

from sklearn.svm import SVC
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import QSVC
from qiskit_machine_learning.kernels import QuantumKernel
from qiskit_machine_learning.datasets import ad_hoc_data

seed = 12345
algorithm_globals.random_seed = seed
```

一般的機械学習ライブラリ

Qiskitの量子機械学習ライブラリ

古典カーネルとしてよく使われるのがガウスカーネル

$$K(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle = C \exp\left(-\frac{|\vec{x}_i - \vec{x}_j|^2}{2\sigma^2}\right)$$

$$* f(\vec{x}_i) = \exp\left(-\frac{|\vec{x}_i - \vec{r}|^2}{2\sigma^2}\right)$$

入力となる特徴量 \vec{x}_i の例:

`sklearn`のデータセットIris(アヤメの品種)の場合

データ	品種 (予測対象)	ガク の長さ	ガク の幅	花弁 の長さ	花弁 の幅
\vec{x}_0	setosa	5.1	3.5	1.4	0.2
\vec{x}_1	setosa	4.9	3.0	1.4	0.2
\vec{x}_2	versicolor	7.0	3.2	4.7	1.4
\vec{x}_3	versicolor	6.4	3.2	4.5	1.5
:					

ではSVMとカーネルの関係とは?

Agenda

- 量子カーネル機械学習
 - 1. カーネルとは
 - 2. scikit-learn + 量子カーネルでの分類(SVM)
 - 量子カーネルを計算する関数を使用
 - 量子カーネル行列そのものを使用
 - 3. scikit-learn + 量子カーネルでのクラスタリング
 - 量子カーネル行列そのものを使用
 - 4. そのほかの量子カーネルの活用候補
- qGAN(量子敵対的生成ネットワーク)
 - 1. 量子状態と確率分布
 - 2. qGANのアーキテクチャ
 - 1. Discriminator
 - 2. Generator
 - 3. 量子回路
 - 3. 学習方法
 - 1. 損失関数
 - 2. 相対エントロピー
 - 3. 結果

カーネルとは

カーネルとは2つのデータの「類似度」を定義する関数です(「類似度」の定義は様々)。

学習データ内の全ての組み合わせで「類似度」を計算した行列をカーネル行列と呼び、SVMで使用されています。

量子状態の内積 $|\langle\phi(x_j)|\phi(x_i)\rangle|^2$ でカーネル行列を定義したSVMを、量子カーネル機械学習と呼びます。

Quantum Kernel Machine Learning

The general task of machine learning is to find and study patterns in data. For many datasets, the datapoints are better understood in a higher dimensional feature space, through the use of a kernel function: $k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle$ where k is the kernel function, \vec{x}_i, \vec{x}_j are n dimensional inputs, f is a map from n -dimension to m -dimension space and $\langle a, b \rangle$ denotes the dot product. When considering finite data, a kernel function can be represented as a matrix: $K_{ij} = k(\vec{x}_i, \vec{x}_j)$.

In quantum kernel machine learning, a quantum feature map $\phi(\vec{x})$ is used to map a classical feature vector \vec{x} to a quantum Hilbert space, $|\phi(\vec{x})\rangle\langle\phi(\vec{x})|$, such that $K_{ij} = |\langle\phi^\dagger(\vec{x}_j)|\phi(\vec{x}_i)\rangle|^2$. See [Supervised learning with quantum enhanced feature spaces](#) for more details.

In this notebook, we use `qiskit` to calculate a kernel matrix using a quantum feature map, then use this kernel matrix in `scikit-learn` classification and clustering algorithms.

```
[1]: import matplotlib.pyplot as plt
import numpy as np

from sklearn.svm import SVC
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import QSVC
from qiskit_machine_learning.kernels import QuantumKernel
from qiskit_machine_learning.datasets import ad_hoc_data

seed = 12345
algorithm_globals.random_seed = seed
```

一般の機械学習ライブラリ

Qiskitの量子機械学習ライブラリ

古典カーネルとしてよく使われるのがガウスカーネル

$$K(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle = C \exp\left(-\frac{|\vec{x}_i - \vec{x}_j|^2}{2\sigma^2}\right)$$

$$* f(\vec{x}_i) = \exp\left(-\frac{|\vec{x}_i - \vec{r}|^2}{2\sigma^2}\right)$$

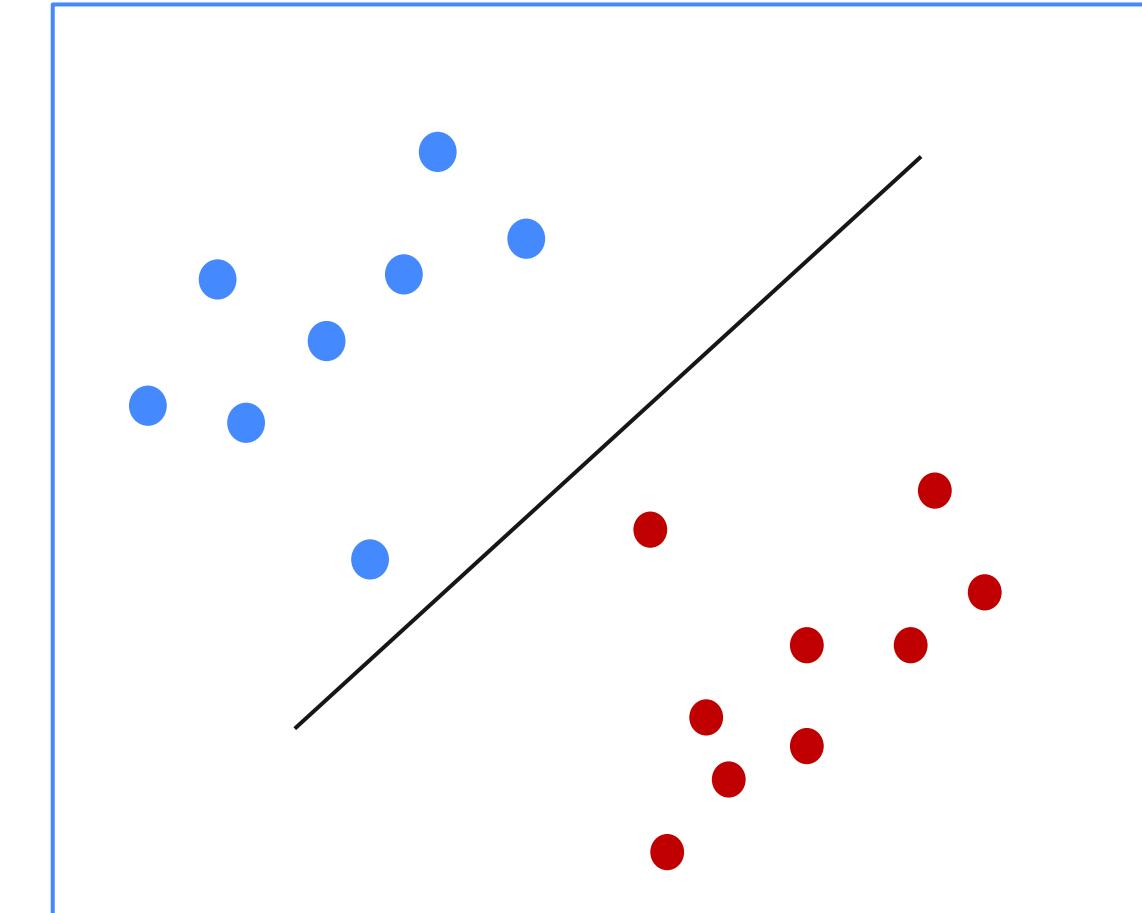
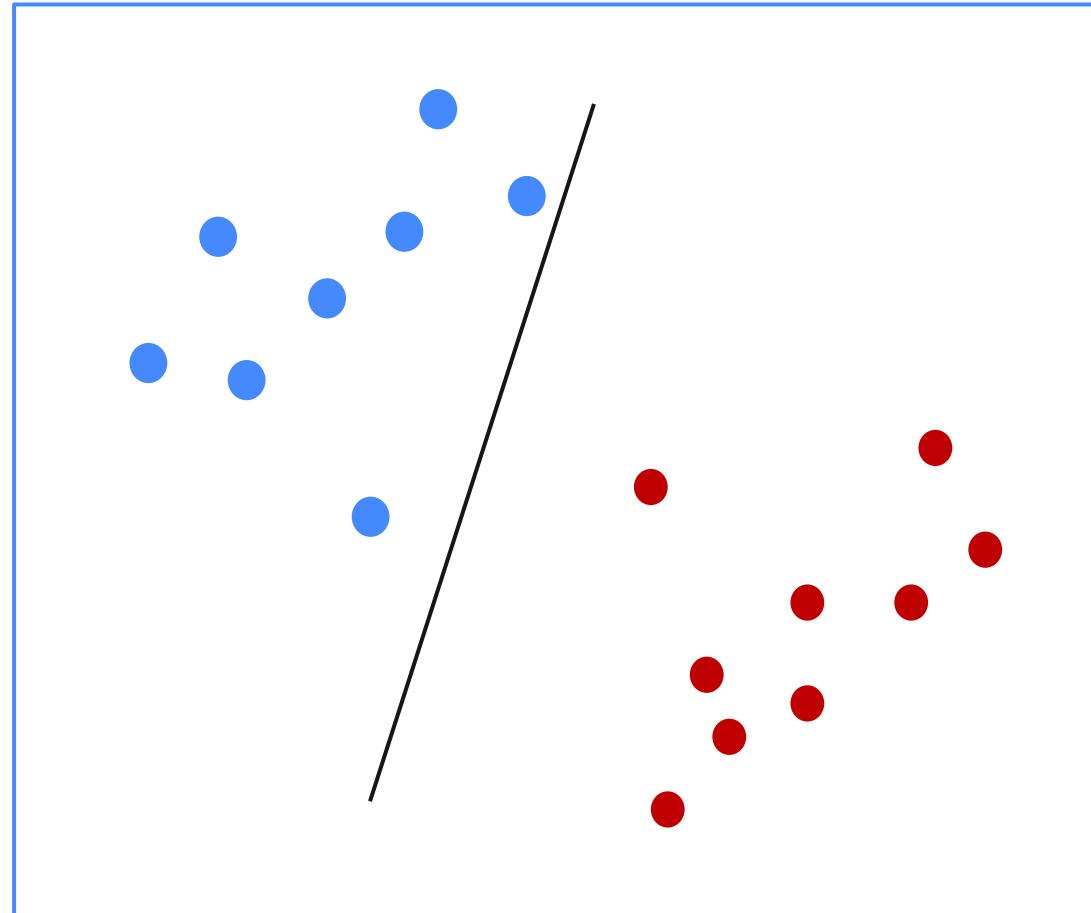
入力となる特徴量 \vec{x}_i の例:
sklearnのデータセットIris(アヤメの品種)の場合

データ	品種 (予測対象)	ガク の長さ	ガク の幅	花弁 の長さ	花弁 の幅
\vec{x}_0	setosa	5.1	3.5	1.4	0.2
\vec{x}_1	setosa	4.9	3.0	1.4	0.2
\vec{x}_2	versicolor	7.0	3.2	4.7	1.4
\vec{x}_3	versicolor	6.4	3.2	4.5	1.5
:					

ではSVMとカーネルの関係とは?

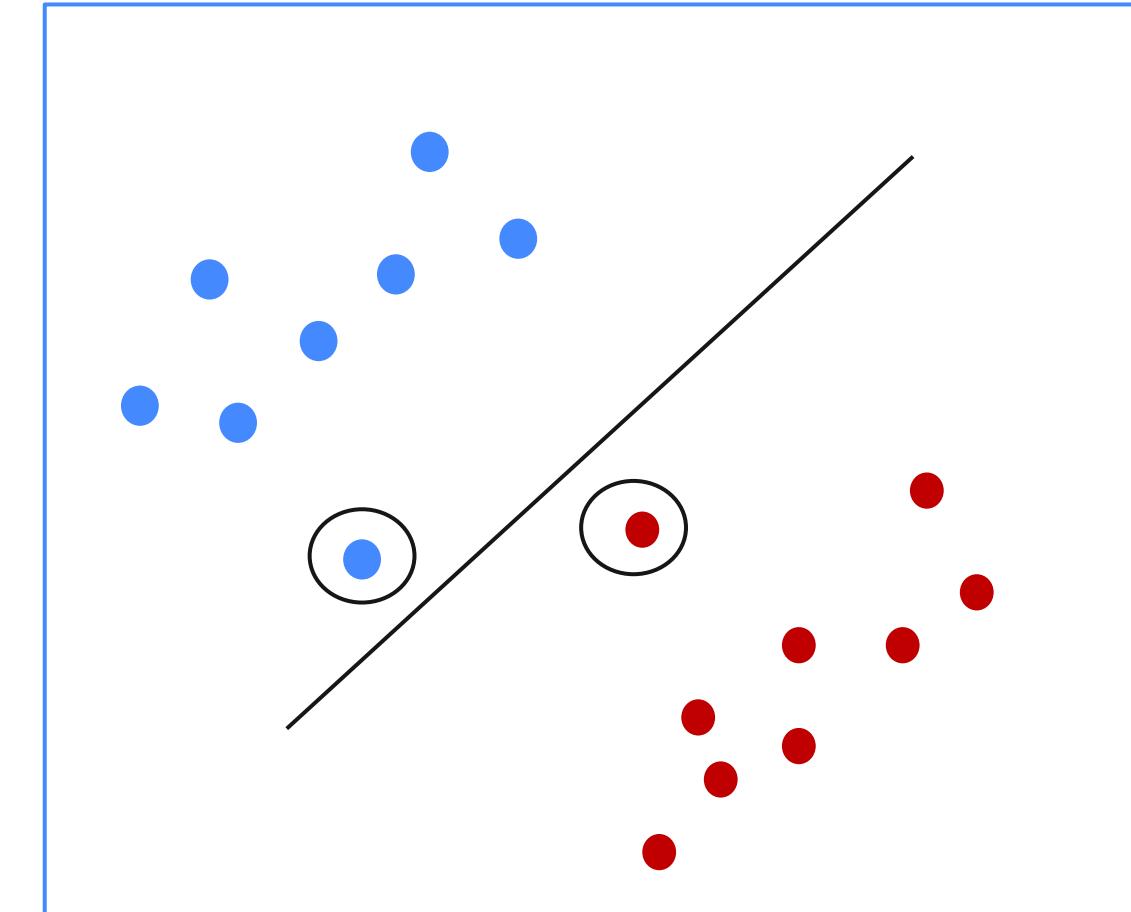
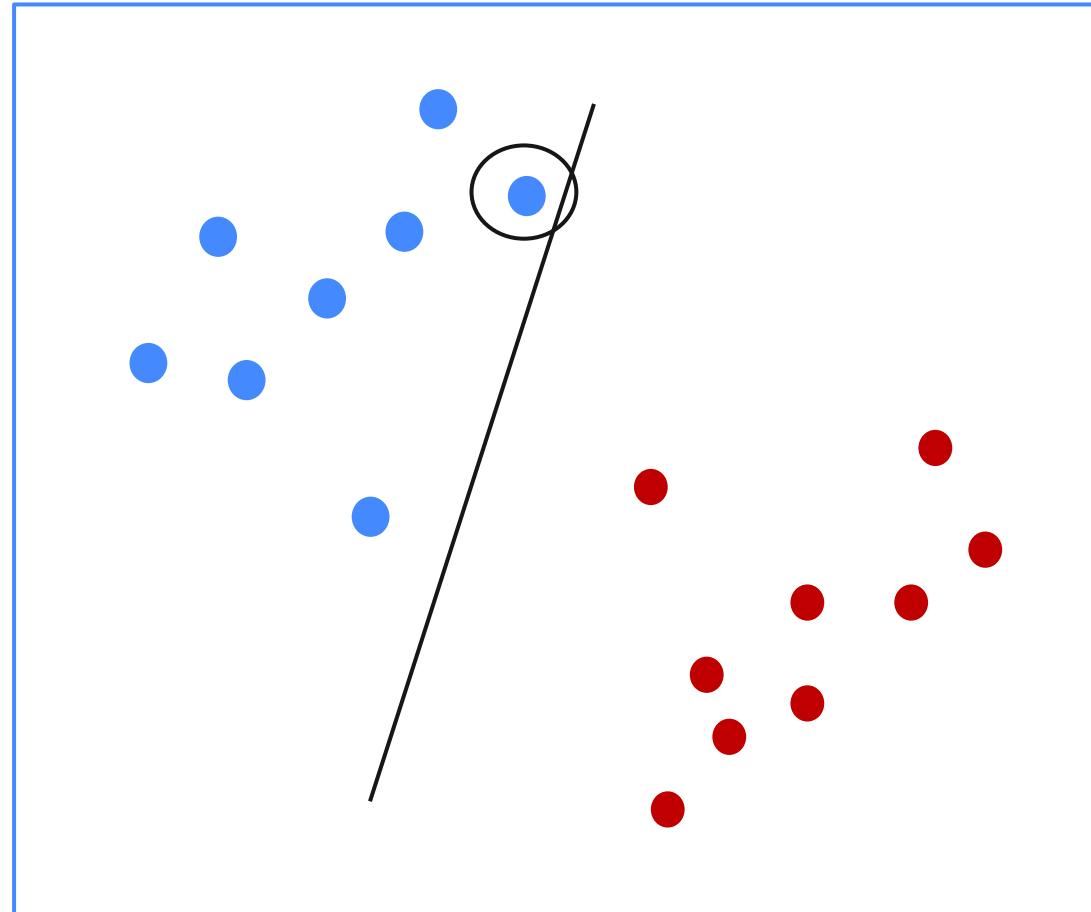
SVMとは

Q : A,Bグループのデータセットがあります。これを下図のように分けるとき、どちらの方が良いでしょうか？



SVMとは

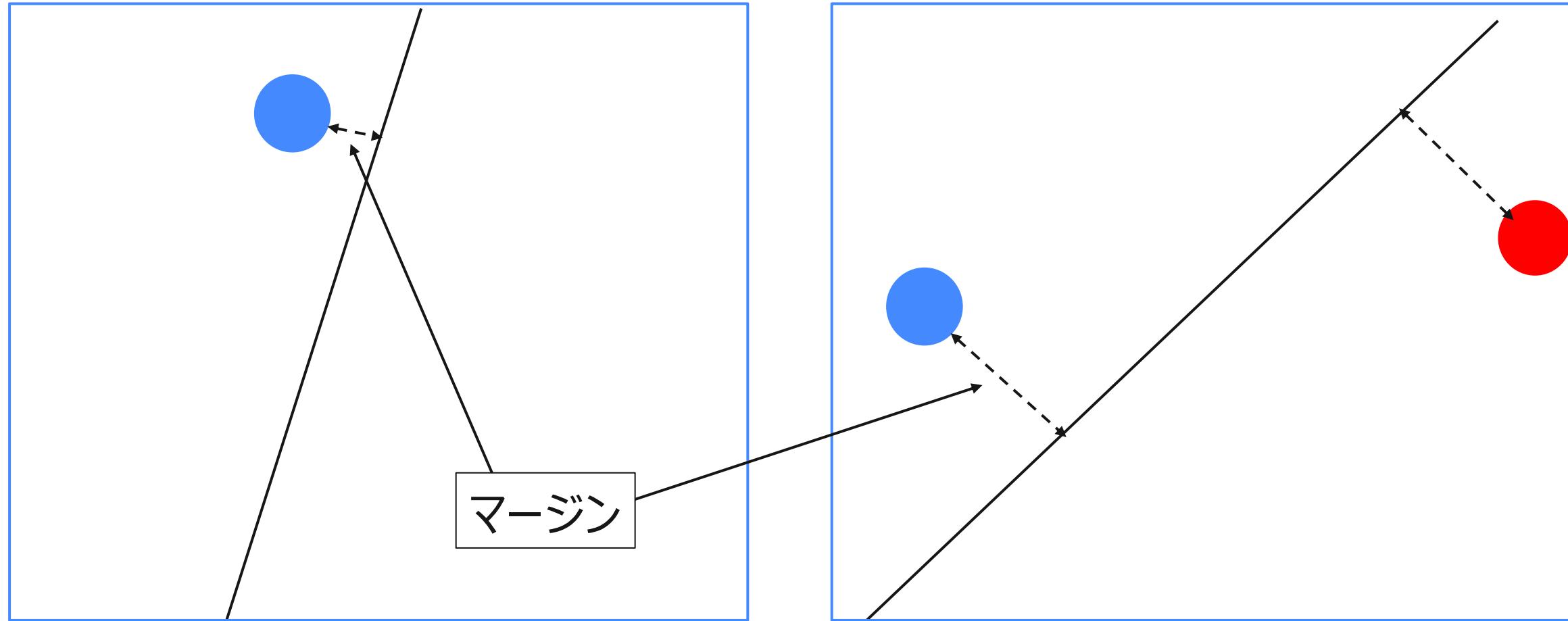
Q : A,Bグループのデータセットがあります。これを下図のように分けるとき、どちらの方が良いでしょうか？



SVMとは

境界線付近の拡大図

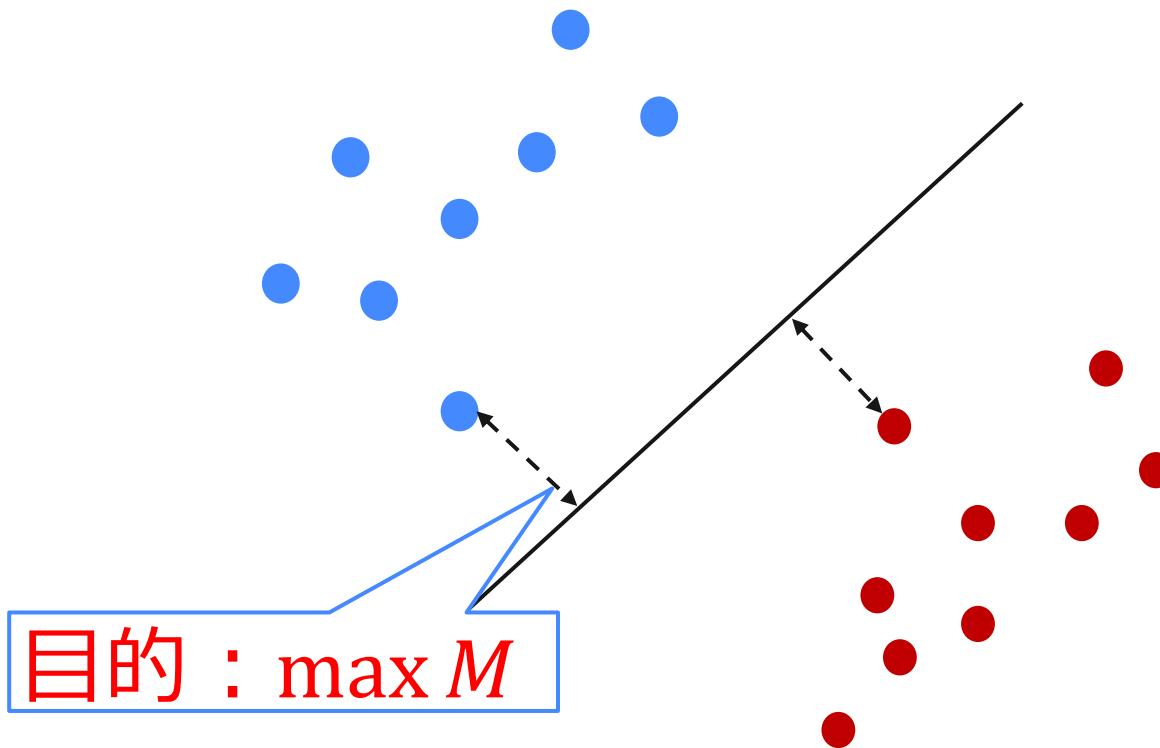
- 右図の方が境界線と最近接データ点との距離(マージン)が長い→左図より安定した分け方



SVMとは

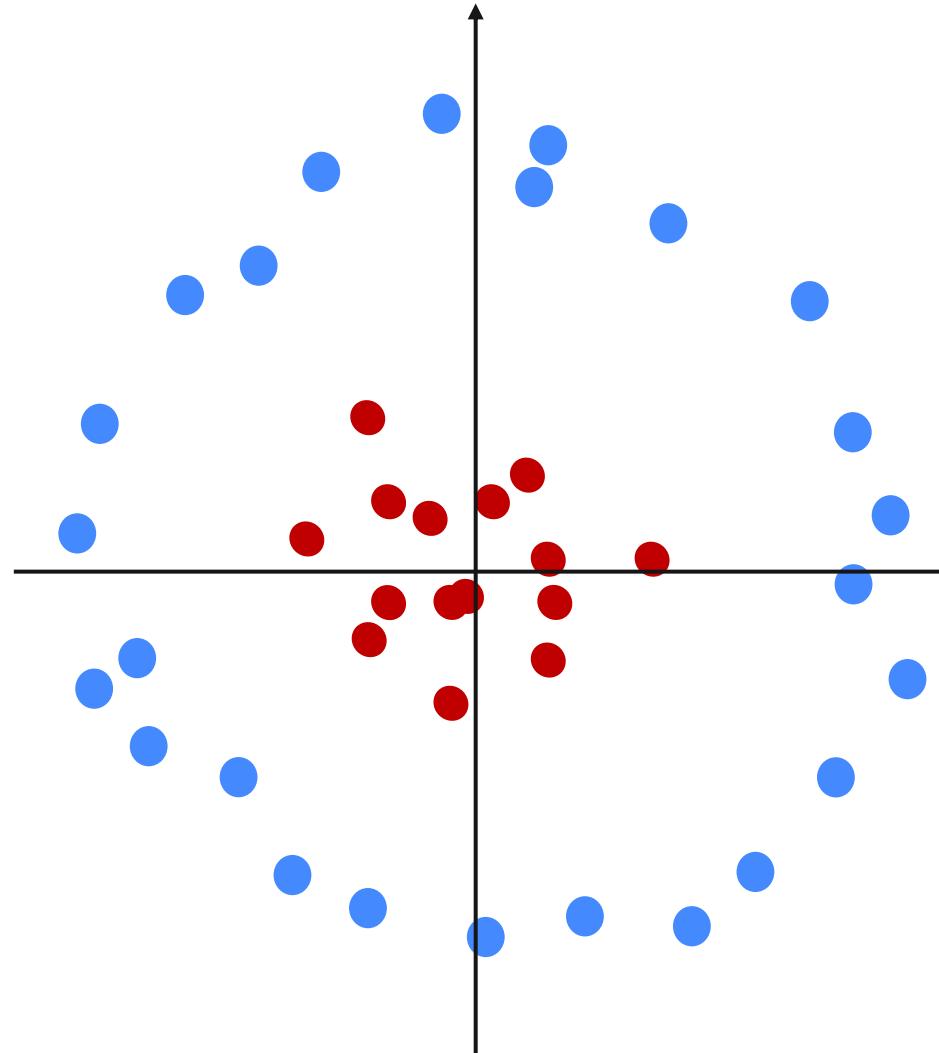
SVM

- ・ グループ間の境界面を定める分析手法
- ・ マージンMをできるだけ大きく取るように最適化→汎化性能をできるだけ高くする



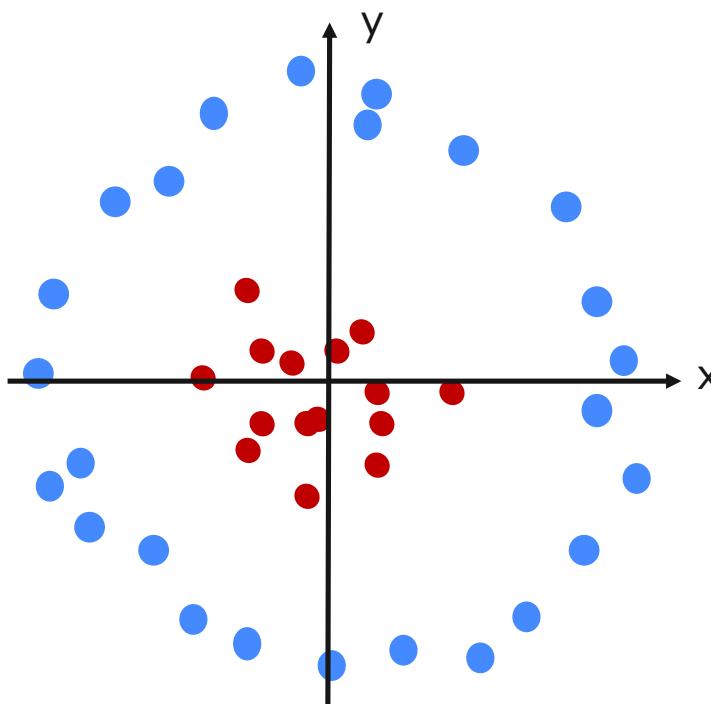
SVMの問題点

境界面は線形である必要があり、下記のようなグループを分けることができない

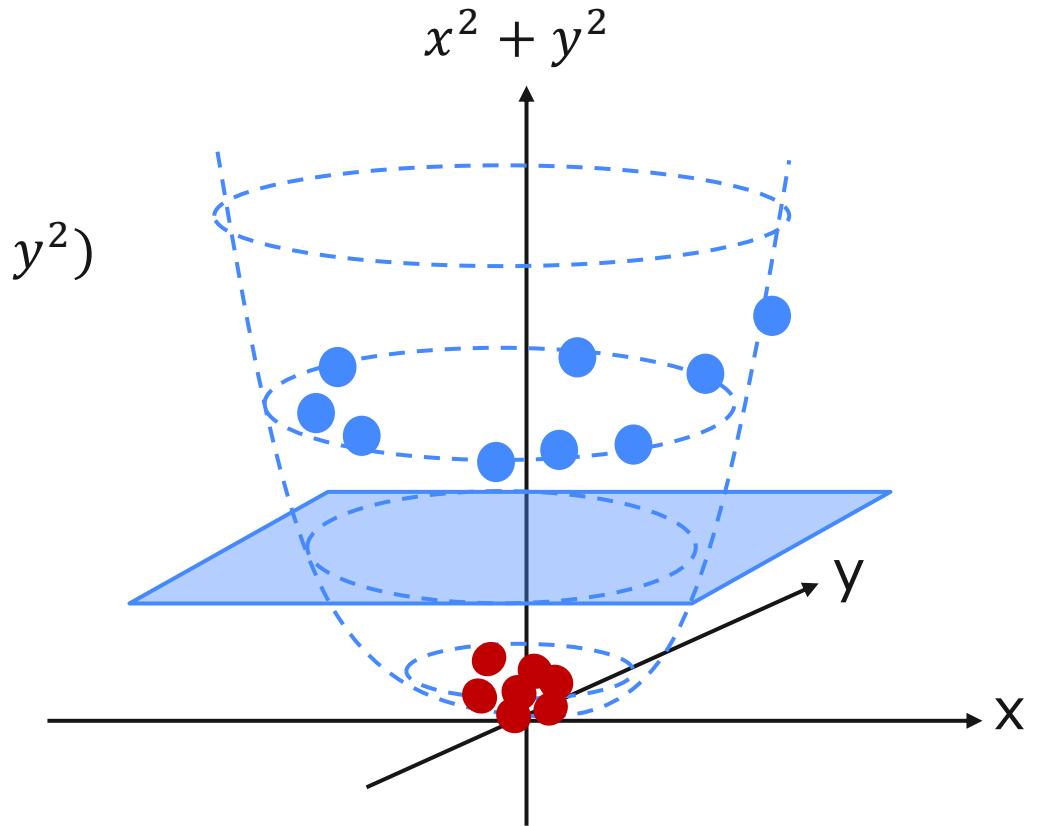


非線形SVM

特徴量を高次元化(特徴量マッピング)することで、線形な境界面で切り分ける



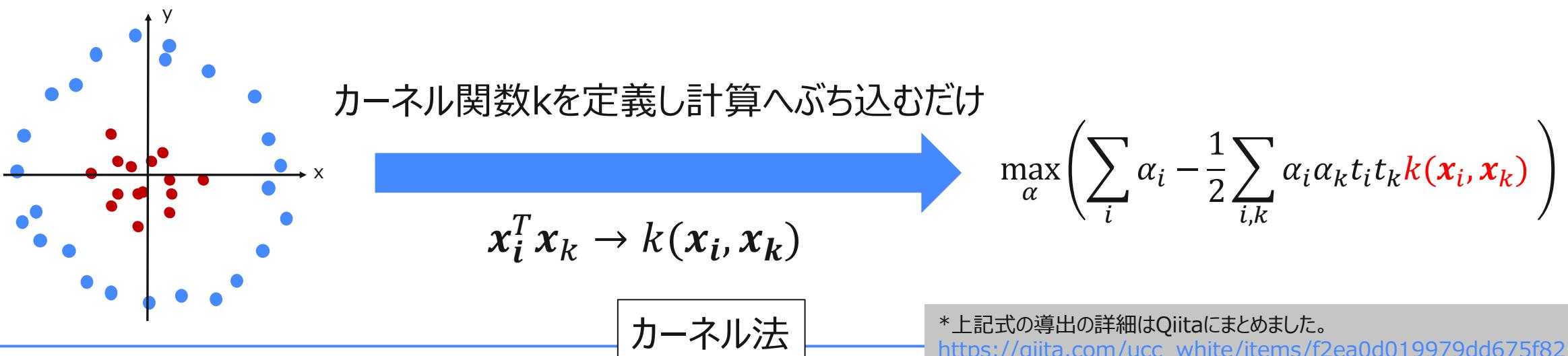
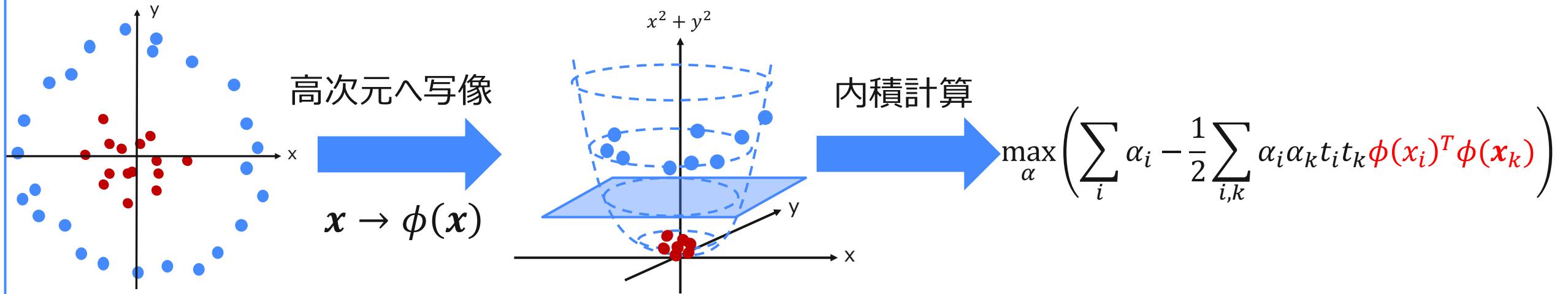
$z = x^2 + y^2$ を追加
 $x \rightarrow \phi(x) = (x, y, x^2 + y^2)$



カーネルトリック

カーネル法…特徴量を高次元化することなく、その計算結果だけを使う手法

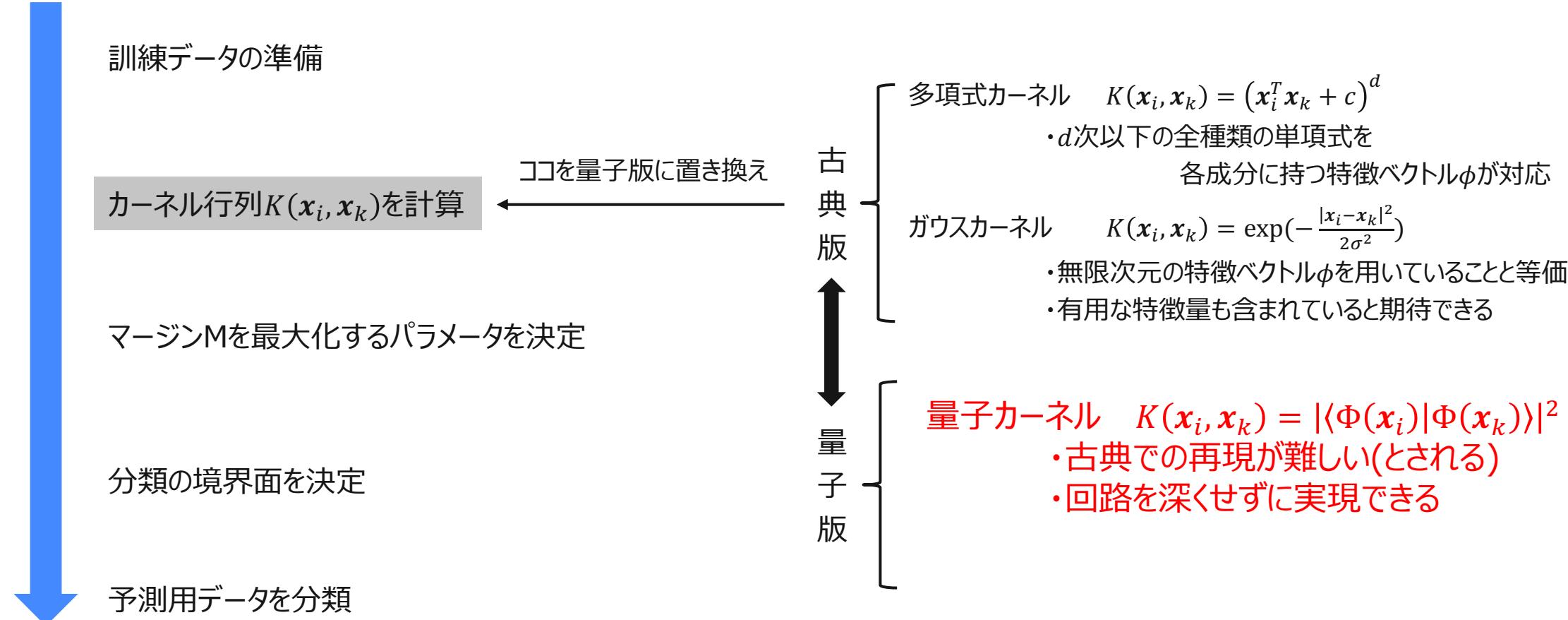
従来の手法



量子カーネル機械学習(QSVM)

古典SVMのカーネル計算を、量子カーネル(量子状態の内積)で置き換えたものを量子カーネル機械学習と呼びます。

SVMの実行フロー



カーネルとは(再掲)

カーネルとは2つのデータの「類似度」を定義する関数です(「類似度」の定義は様々)。

学習データ内の全ての組み合わせで「類似度」を計算した行列をカーネル行列と呼び、SVMで使用されています。

量子状態の内積 $|\langle\phi^\dagger(x_j)|\phi(x_i)\rangle|^2$ でカーネル行列を定義したSVMを、量子カーネル機械学習と呼びます。

Quantum Kernel Machine Learning

The general task of machine learning is to find and study patterns in data. For many datasets, the datapoints are better understood in a higher dimensional feature space, through the use of a kernel function: $k(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle$ where k is the kernel function, \vec{x}_i, \vec{x}_j are n dimensional inputs, f is a map from n -dimension to m -dimension space and $\langle a, b \rangle$ denotes the dot product. When considering finite data, a kernel function can be represented as a matrix: $K_{ij} = k(\vec{x}_i, \vec{x}_j)$.

In quantum kernel machine learning, a quantum feature map $\phi(\vec{x})$ is used to map a classical feature vector \vec{x} to a quantum Hilbert space, $|\phi(\vec{x})\rangle\langle\phi(\vec{x})|$, such that $K_{ij} = |\langle\phi^\dagger(\vec{x}_j)|\phi(\vec{x}_i)\rangle|^2$. See [Supervised learning with quantum enhanced feature spaces](#) for more details.

In this notebook, we use `qiskit` to calculate a kernel matrix using a quantum feature map, then use this kernel matrix in `scikit-learn` classification and clustering algorithms.

```
[1]: import matplotlib.pyplot as plt
import numpy as np

from sklearn.svm import SVC
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import QSVC
from qiskit_machine_learning.kernels import QuantumKernel
from qiskit_machine_learning.datasets import ad_hoc_data

seed = 12345
algorithm_globals.random_seed = seed
```

一般の機械学習ライブラリ

Qiskitの量子機械学習ライブラリ

古典カーネルとしてよく使われるのがガウスカーネル

$$K(\vec{x}_i, \vec{x}_j) = \langle f(\vec{x}_i), f(\vec{x}_j) \rangle = C \exp\left(-\frac{|\vec{x}_i - \vec{x}_j|^2}{2\sigma^2}\right)$$

$$* f(\vec{x}_i) = \exp\left(-\frac{|\vec{x}_i - \vec{r}|^2}{2\sigma^2}\right)$$

入力となる特徴量 \vec{x}_i の例:
sklearnのデータセットIris(アヤメの品種)の場合

データ	品種 (予測対象)	ガク の長さ	ガク の幅	花弁 の長さ	花弁 の幅
\vec{x}_0	setosa	5.1	3.5	1.4	0.2
\vec{x}_1	setosa	4.9	3.0	1.4	0.2
\vec{x}_2	versicolor	7.0	3.2	4.7	1.4
\vec{x}_3	versicolor	6.4	3.2	4.5	1.5
:					

ではSVMとカーネルの関係とは?

scikit-learn + 量子カーネルでの分類(SVM)

量子カーネル学習用のデータセットad hoc datasetを使ってSVMによる分類を行います。

ad hoc datasetとは、量子カーネル学習が解きやすい問題となるように設定した特殊なデータセットです。

Classification

For our classification example, we will use the `ad hoc dataset` as described in [Supervised learning with quantum enhanced feature spaces](#), and the `scikit-learn` support vector machine classification (`svc`) algorithm.

```
[2]: adhoc_dimension = 2
train_features, train_labels, test_features, test_labels, adhoc_total = ad_hoc_data(
    training_size=20,
    test_size=5,
    n=adhoc_dimension,
    gap=0.3,
    plot_data=False, one_hot=False, include_sample_total=True
)

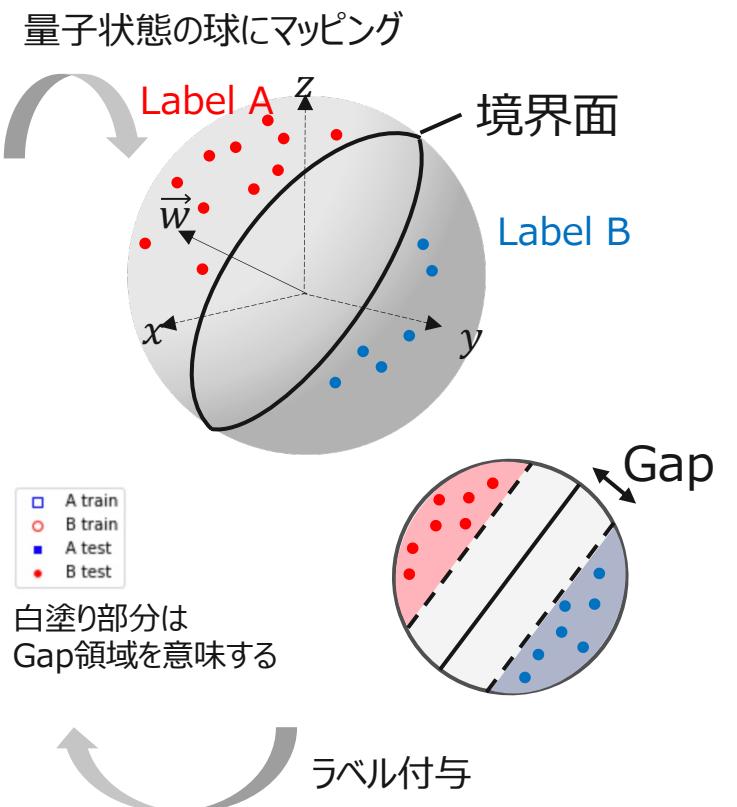
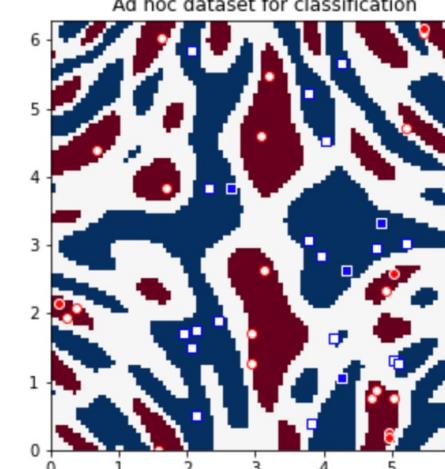
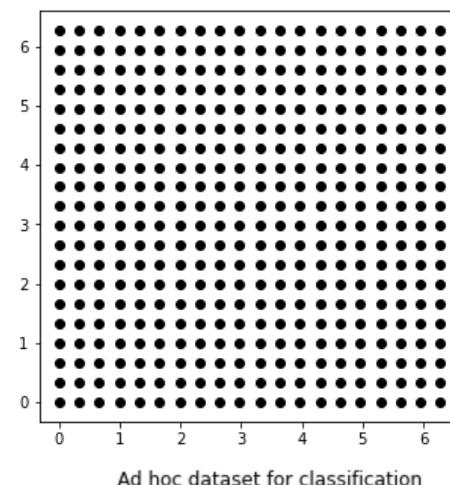
plt.figure(figsize=(5, 5))
plt.ylim(0, 2 * np.pi)
plt.xlim(0, 2 * np.pi)
plt.imshow(np.asmatrix(adhoc_total).T, interpolation='nearest',
           origin='lower', cmap='RdBu', extent=[0, 2 * np.pi, 0, 2 * np.pi])

plt.scatter(train_features[np.where(train_labels[:] == 0), 0], train_features[np.where(train_labels[:] == 0), 1],
            marker='s', facecolors='w', edgecolors='b', label="A train")
plt.scatter(train_features[np.where(train_labels[:] == 1), 0], train_features[np.where(train_labels[:] == 1), 1],
            marker='o', facecolors='w', edgecolors='r', label="B train")
plt.scatter(test_features[np.where(test_labels[:] == 0), 0], test_features[np.where(test_labels[:] == 0), 1],
            marker='s', facecolors='b', edgecolors='w', label="A test")
plt.scatter(test_features[np.where(test_labels[:] == 1), 0], test_features[np.where(test_labels[:] == 1), 1],
            marker='o', facecolors='r', edgecolors='w', label="B test")

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.title("Ad hoc dataset for classification")

plt.show()
```

Ad hocデータは2次元上の座標に応じて量子状態の球にマッピングし、北極側か南極側かに応じてラベルをつけている



では具体的にどのような
マッピングをしているか？

scikit-learn + 量子カーネルでの分類(SVM)

量子カーネル行列とad hoc datasetはZZFeatureMapに基づいて生成されます。
繰り返し数(reps)が2のZZFeatureMapでは古典での再現が難しくなるとされています。

With our training and testing datasets ready, we set up the `QuantumKernel` class to calculate a kernel matrix using the `ZZFeatureMap`, and the `BasicAer qasm_simulator` using 1024 shots.

```
[3]: adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension,
                                         reps=2, entanglement='linear')

adhoc_backend = QuantumInstance(BasicAer.get_backend('qasm_simulator'), shots=1024,
                                 seed_simulator=seed, seed_transpiler=seed)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map, quantum_instance=adhoc_backend)
```

$x \rightarrow |\Phi_{ZZ}(x)\rangle$ のマッピング

特徴量が2次元の場合

$$x_i = (\alpha_i, \beta_i)$$

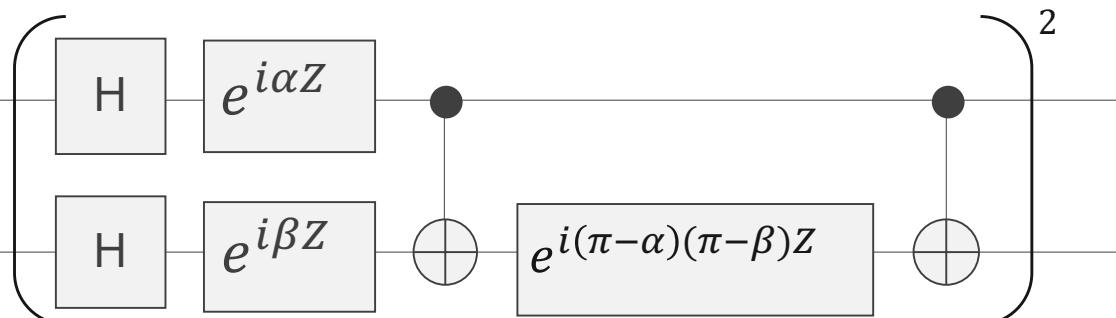
$$\begin{aligned} \rightarrow |\Phi_{ZZ}(x)\rangle &= A(\alpha_i, \beta_i) |00\rangle + B(\alpha_i, \beta_i) |01\rangle \\ &\quad + C(\alpha_i, \beta_i) |10\rangle + D(\alpha_i, \beta_i) |11\rangle \end{aligned}$$

連続な値を量子状態の位相にエンコードする

ZZFeatureMap

入力データ $x = (\alpha, \beta)$ に対応する量子状態

$$|\Phi_{ZZ}(x)\rangle = (e^{i(\alpha Z_1 + \beta Z_2 + (\pi - \alpha)(\pi - \beta)Z_1 Z_2)} H^2)^2 |0\rangle$$

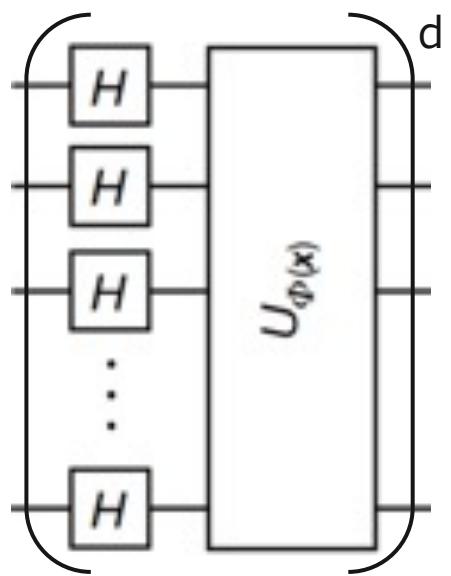


scikit-learn + 量子カーネルでの分類(SVM)

FeatureMapは一般化すると次のように表されます。

ZZFeatureMapはこの式の $\phi_S(x), P_K$ の選び方の一例です。

$$x \rightarrow |\Phi(x)\rangle = \Pi_d U_{\Phi(x)} H^n |0\rangle$$



$$\text{where } U_{\Phi(x)} = \exp \left(i \sum_S \phi_S(x) \Pi_{K \in S} P_K \right)$$

$$= \exp \left(i \sum_i \phi_i(x) P_i + i \sum_{i,j} \phi_{ij}(x) P_{ij} + \sum_{i,j,k} \phi_{ijk}(x) P_{ijk} + \dots \right)$$

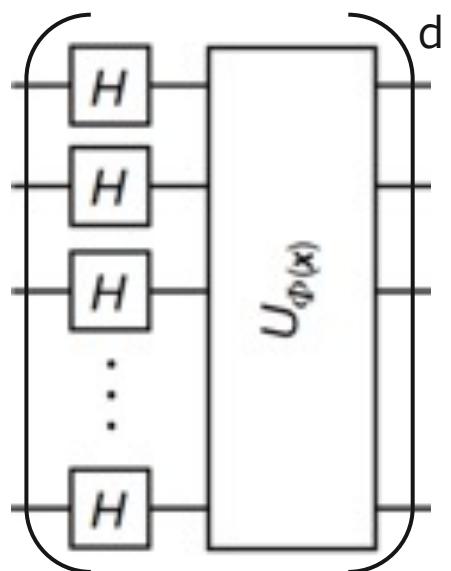
- S はエンタングルさせるビットの組み合わせ
- $\phi_i(x)$ はパワリ行列 P_i の係数
- d は繰り返し数 (reps)

} マッピングの自由度

scikit-learn + 量子カーネルでの分類(SVM)

一番簡単な例としてZ-FeatureMapがありますが、これでは量子の優位性はありません。

$$x \rightarrow |\Phi(x)\rangle = \Pi_d U_{\Phi(x)} H^n |0\rangle$$



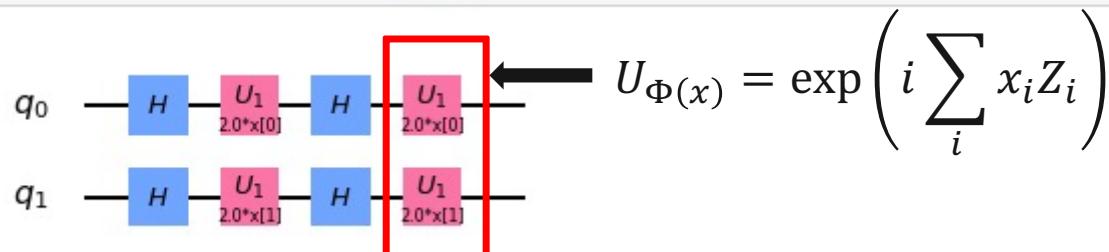
$$\text{where } U_{\Phi(x)} = \exp \left(i \sum_S \phi_S(x) \Pi_{K \in S} P_K \right)$$

$$= \exp \left(i \sum_i \phi_i(x) P_i + i \sum_{i,j} \phi_{ij}(x) P_{ij} + \sum_{i,j,k} \phi_{ijk}(x) P_{ijk} + \dots \right)$$

- S はエンタングルさせるビットの組み合わせ
- $\phi_i(x)$ はパウリ行列 P_i の係数

$$\begin{cases} \phi_S(x) = \phi_i(x) = x_i \\ P_K = Z_i \end{cases}$$

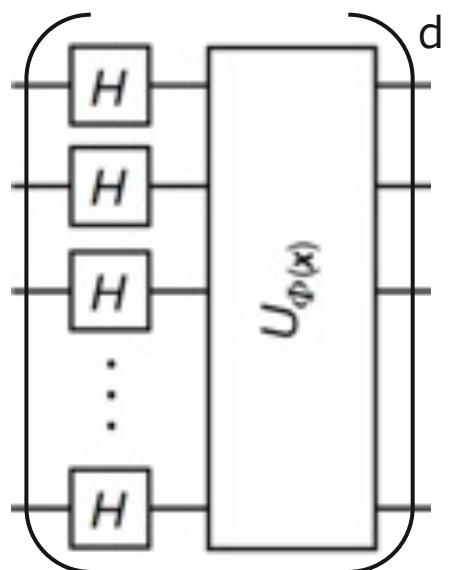
```
from qiskit import *
from qiskit.circuit.library import ZZFeatureMap, ZFeatureMap, PauliFeatureMap
Z_feature_map = ZFeatureMap(feature_dimension=2, reps=2) # d=2
Z_feature_map.draw(output="mpl")
```



scikit-learn + 量子カーネルでの分類(SVM)

量子カーネル機械学習の実装論文・チュートリアルではZZ-FeatureMapが採用されており、
 $d=2$ (2回の繰り返し)以上で古典での再現が難しくなるとされています。

$$x \rightarrow |\Phi(x)\rangle = \Pi_d U_{\Phi(x)} H^n |0\rangle$$



$$\text{where } U_{\Phi(x)} = \exp \left(i \sum_S \phi_S(x) \Pi_{K \in S} P_K \right)$$

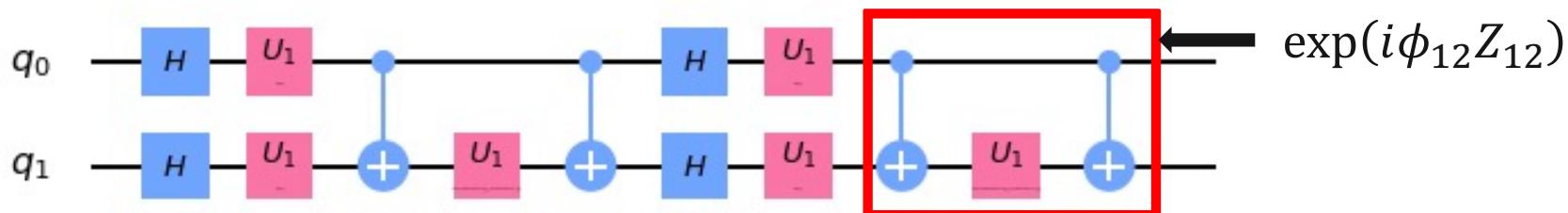
$$= \exp \left(i \sum_i \phi_i(x) P_i + i \sum_{i,j} \phi_{ij}(x) P_{ij} + \sum_{i,j,k} \phi_{ijk}(x) P_{ijk} + \dots \right)$$

- S はエンタングルさせるビットの組み合わせ
- $\phi_i(x)$ はパワリ行列 P_i の係数

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

$$P_K = \begin{cases} Z_i & K = i \\ Z_{ij} & K = ij \end{cases}$$

```
ZZ_feature_map = ZZFeatureMap(feature_dimension=2, reps=2) # d=2
ZZ_feature_map.draw(output="mpl", style={"subfontsize":0})
```



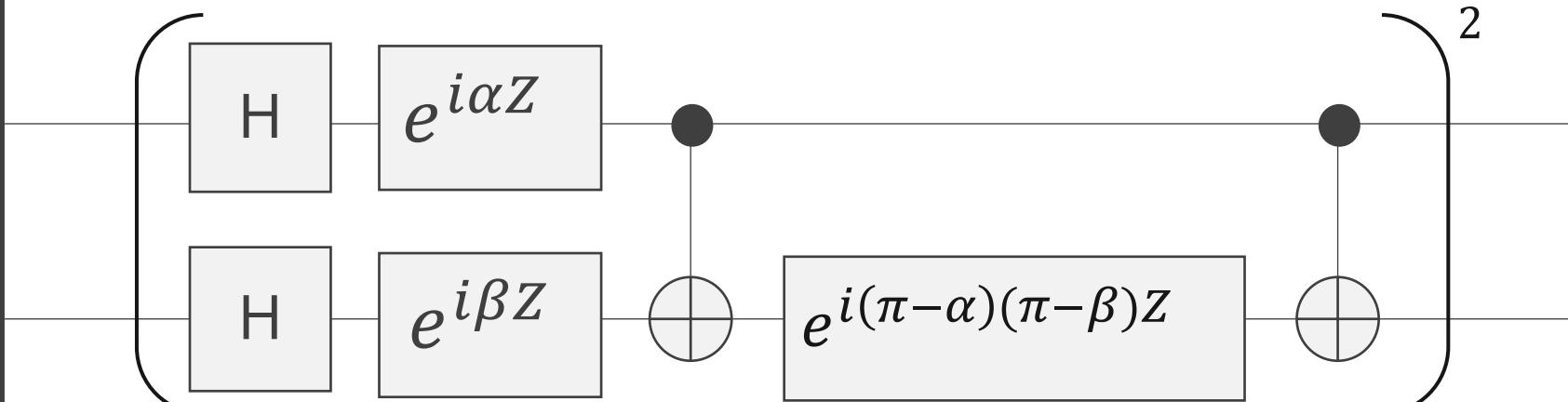
scikit-learn + 量子カーネルでの分類(SVM)

ZZ-FeatureMapは $H, e^{i\theta Z}, CNOT$ ゲートのみで構成ができます。

$$x \rightarrow |\Phi_{ZZ}(x)\rangle = U_{\Phi(x)} H^n U_{\Phi(x)} H^n |0\rangle$$

入力データ $x = (\alpha, \beta)$ に対応する量子状態

$$|\Phi_{ZZ}(x)\rangle = (e^{i(\alpha Z_1 + \beta Z_2 + (\pi - \alpha)(\pi - \beta)Z_1 Z_2)} H^2)^2 |0\rangle$$



scikit-learn + 量子カーネルでの分類(SVM)

量子カーネルと古典機械学習(sklearn SVM)をHybridするには、2種類の方法があります。

- (1)量子カーネルを計算する関数を古典SVMで指定
- (2)量子カーネル行列を古典SVMで指定 (すでにカーネル行列を計算していた場合に使用)

The scikit-learn `SVC` algorithm allows us to define a `custom kernel` in two ways: by providing the kernel as a callable function or by precomputing the kernel matrix. We can do either of these using the `QuantumKernel` class in qiskit.

The following code gives the kernel as a callable function:

```
[4]: adhoc_svc = SVC(kernel=adhoc_kernel.evaluate)
adhoc_svc.fit(train_features, train_labels)
adhoc_score = adhoc_svc.score(test_features, test_labels)

print(f'Callable kernel classification test score: {adhoc_score}')
```

Callable kernel classification test score: 1.0

訓練データの準備

カーネル行列 $K(x_i, x_k)$ を計算

カーネル行列 $K(x_i, x_k)$ を利用して
分類の境界面を決定

予測用データを分類

(1)でのsklearnタスク

(2)でのsklearnタスク

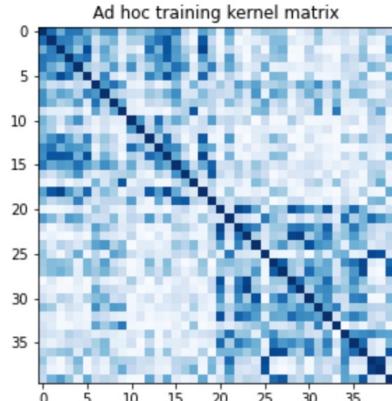
The following code precomputes and plots the training and testing kernel matrices before providing them to the scikit-learn `SVC` algorithm:

```
[5]: adhoc_matrix_train = adhoc_kernel.evaluate(x_vec=train_features,
                                              y_vec=train_features)
adhoc_matrix_test = adhoc_kernel.evaluate(x_vec=test_features,
                                         y_vec=train_features)

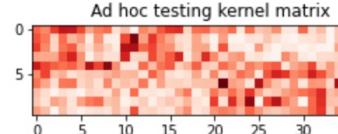
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(np.asmatrix(adhoc_matrix_train),
             interpolation='nearest', origin='upper', cmap='Blues')
axs[0].set_title("Ad hoc training kernel matrix")
axs[1].imshow(np.asmatrix(adhoc_matrix_test),
             interpolation='nearest', origin='upper', cmap='Reds')
axs[1].set_title("Ad hoc testing kernel matrix")
plt.show()

adhoc_svc = SVC(kernel='precomputed')
adhoc_svc.fit(adhoc_matrix_train, train_labels)
adhoc_score = adhoc_svc.score(adhoc_matrix_test, test_labels)

print(f'Precomputed kernel classification test score: {adhoc_score}')
```



学習データ同士の組み合わせで
カーネル行列を構築



学習データとテストデータの
組み合わせでカーネル行列を構築

Precomputed kernel classification test score: 1.0

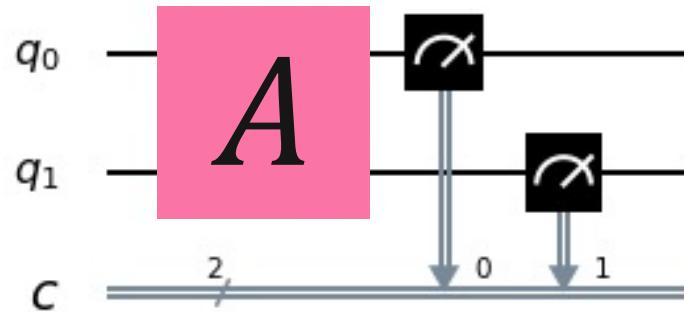
scikit-learn + 量子カーネルでの分類(SVM)

ここまで $|\Phi(x)\rangle$ の実装方法を学びました。

次は量子カーネル $K(x_i, x_k) = |\langle\Phi(x_i)|\Phi(x_k)\rangle|^2$ の計算方法です。

初期状態 $|00\rangle$ にユニタリ行列 A をかけた状態 $|\psi\rangle$ の観測考えます

$$|\psi\rangle = A|00\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$$

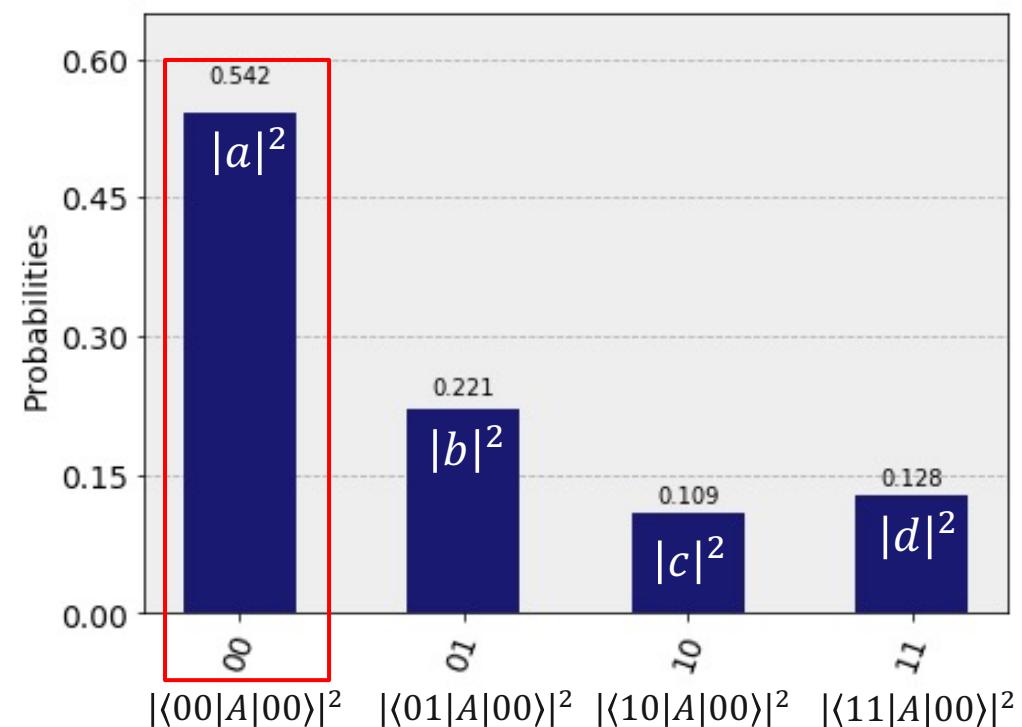


このとき観測される状態が $|00\rangle, \dots, |11\rangle$ となる確率はそれぞれ $|a|^2, \dots, |d|^2$ となります(量子力学のルール)。

$\langle i|j\rangle = \delta_{ij}$ なので、 $|\langle 00|A|00\rangle|^2 = |a|^2$ です。

Shot=1000 (千回観測)したときのヒストグラム

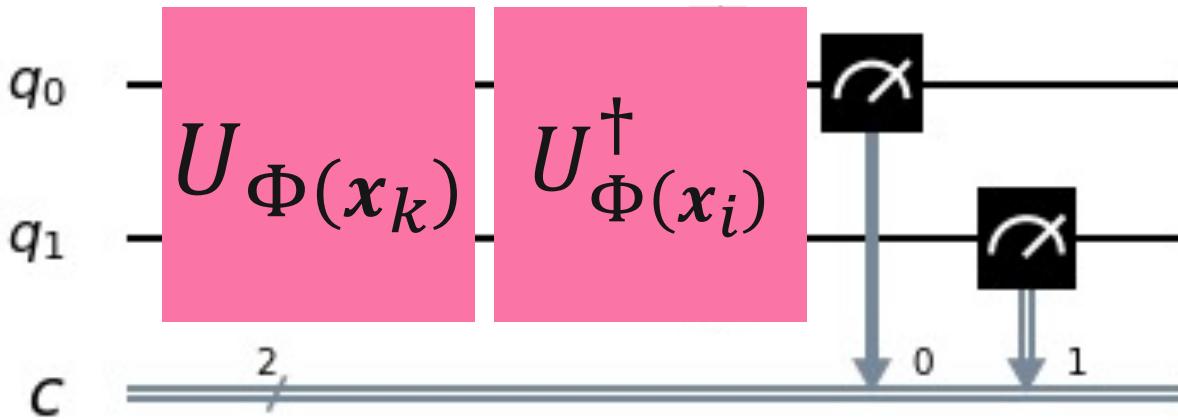
{'11': 128, '01': 221, '10': 109, '00': 542}



scikit-learn + 量子カーネルでの分類(SVM)

量子カーネル $K(x_i, x_k) = |\langle \Phi(x_i) | \Phi(x_k) \rangle|^2 = \left| \langle 0 | U_{\Phi(x_i)}^\dagger U_{\Phi(x_k)} | 0 \rangle \right|^2$ より同様に計算できます。

特徴量が2次元の場合($|0\rangle := |00\rangle$)、次のように計算できます



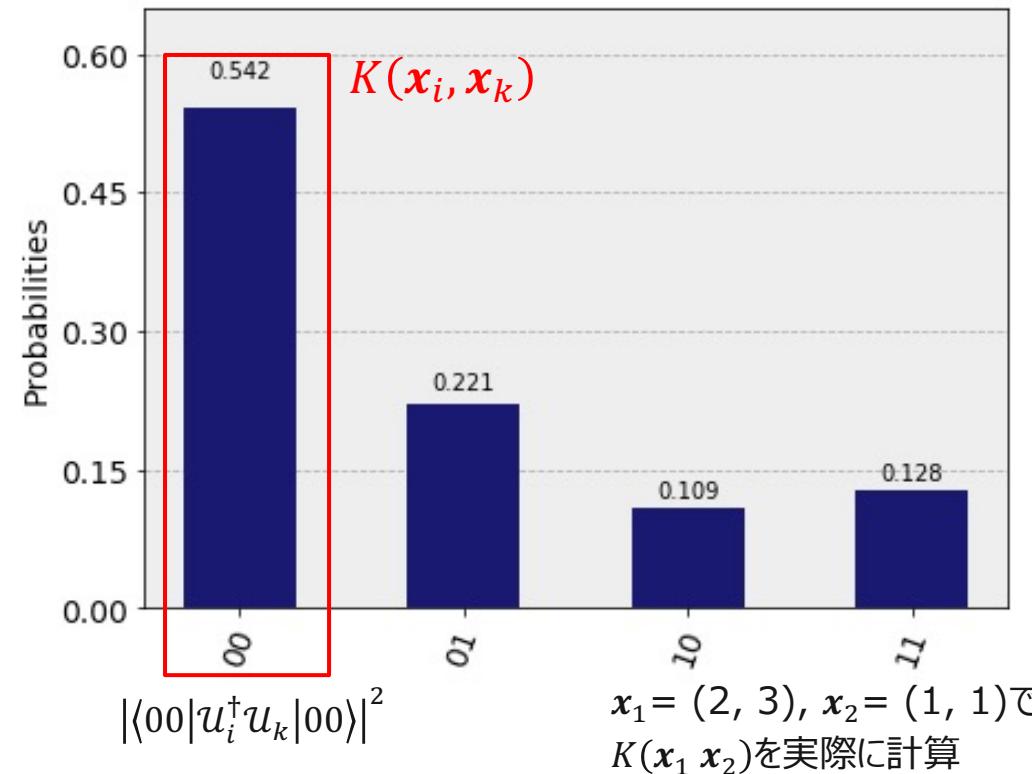
をR回測定して $|00\rangle$ がr回出現したとき、

$$K(x_i, x_k) = \frac{r}{R}$$

となります。

* $U_{\Phi(x_k)} = U_{\Phi(x)} H^n U_{\Phi(x)} H^n$

Shot=1000 (千回観測)したときのヒストグラム
`{'11': 128, '01': 221, '10': 109, '00': 542}`



scikit-learn + 量子カーネルでの分類(SVM)

sklearnモジュールを使わなくても、QiskitのQSVMクラスを使って分類ができます。

qiskit 也 contains the `qsvc` class that extends the `sklearn svc` class, that can be used as follows:

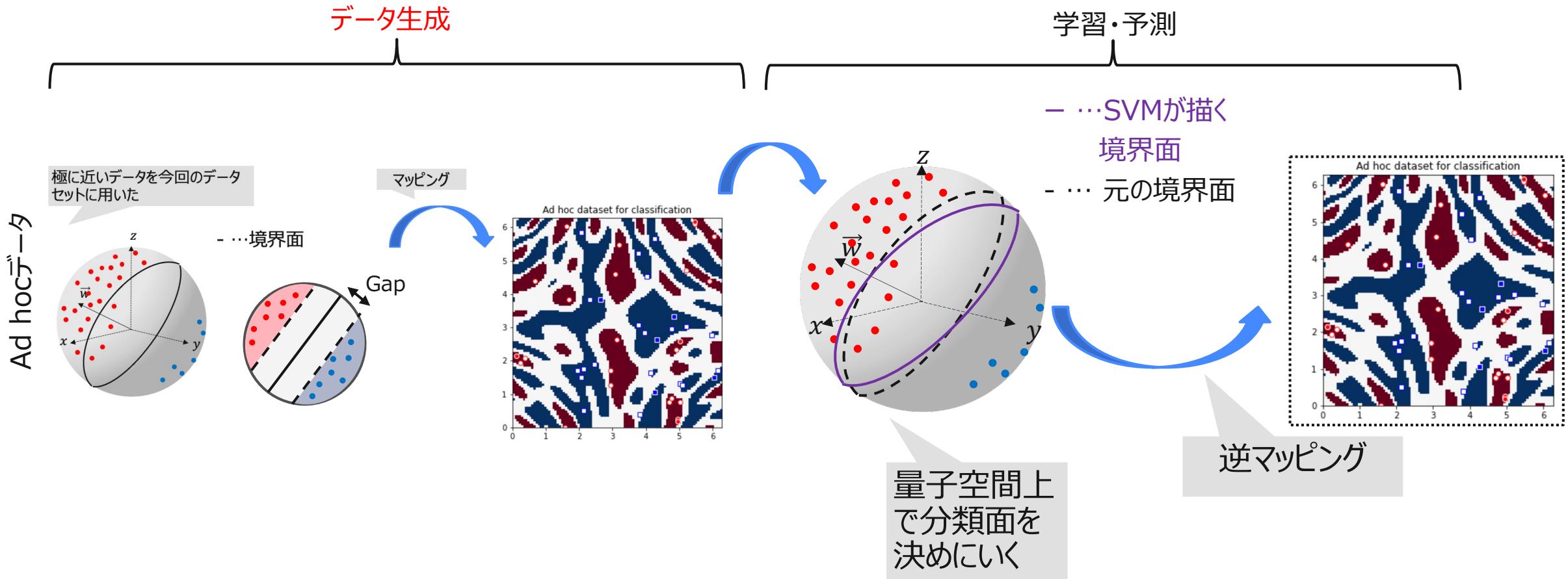
```
[6]: qsvc = QSVC(quantum_kernel=adhoc_kernel)
qsvc.fit(train_features, train_labels)
qsvc_score = qsvc.score(test_features, test_labels)

print(f'QSVC classification test score: {adhoc_score}')
```

QSVC classification test score: 1.0

scikit-learn + 量子カーネルでの分類(SVM)

「データ生成」と「学習・予測」の両方でプロット球へのマッピングを通して行なっているので、SVMの分類面がデータ生成時の境界面を再現できることが考えられます。



scikit-learn + 量子カーネルでのクラスタリング

ここまで話は分類タスクという教師あり学習でした。事前に教師データのラベルAとラベルBの分類が既知でした。Clusteringではこのラベル情報がない教師なし学習です。この場合はテストデータを使いません。

Clustering

For our clustering example, we will again use the *ad hoc dataset* as described in [Supervised learning with quantum enhanced feature spaces](#), and the `scikit-learn spectral` clustering algorithm.

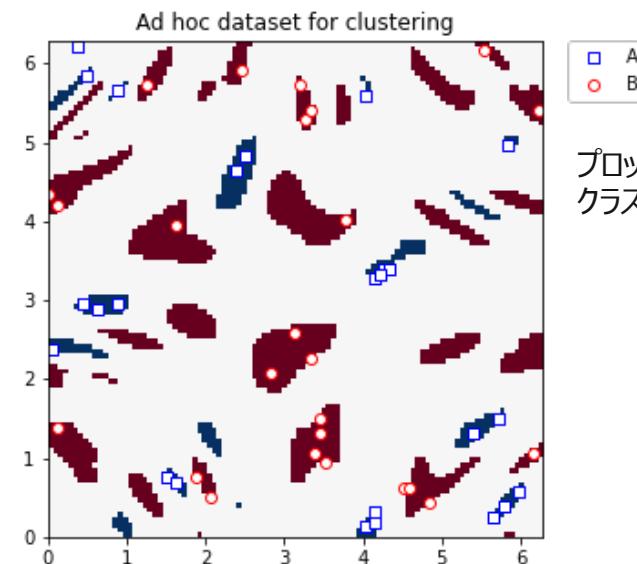
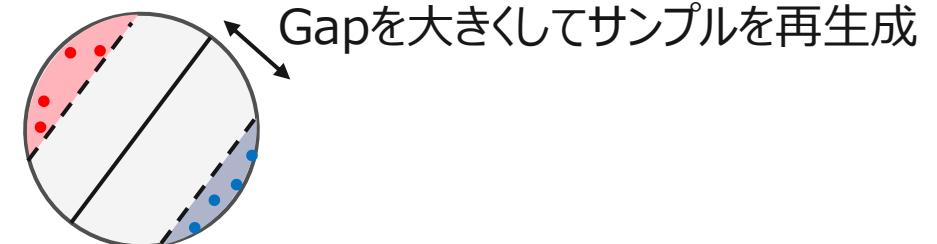
We will regenerate the dataset with a larger gap between the two classes, and as clustering is an unsupervised machine learning task, we don't need a test sample.

```
[7]: adhoc_dimension = 2
train_features, train_labels, test_features, test_labels, adhoc_total = ad_hoc_data(
    training_size=25,
    test_size=0,
    n=adhoc_dimension,
    gap=0.6,
    plot_data=False, one_hot=False, include_sample_total=True
)

plt.figure(figsize=(5, 5))
plt.ylim(0, 2 * np.pi)
plt.xlim(0, 2 * np.pi)
plt.imshow(np.asmatrix(adhoc_total).T, interpolation='nearest',
           origin='lower', cmap='RdBu', extent=[0, 2 * np.pi, 0, 2 * np.pi])
plt.scatter(train_features[np.where(train_labels[:] == 0), 0], train_features[np.where(train_labels[:] == 0), 1],
            marker='s', facecolors='w', edgecolors='b', label="A")
plt.scatter(train_features[np.where(train_labels[:] == 1), 0], train_features[np.where(train_labels[:] == 1), 1],
            marker='o', facecolors='w', edgecolors='r', label="B")

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.title("Ad hoc dataset for clustering")

plt.show()
```



プロット上はA, Bと表示しているが
クラスタリング時はA, Bを未知として実行

scikit-learn + 量子カーネルでのクラスタリング

クラスタリングのタスクではsklearnのSpectralClusteringクラスを使います。
この場合は計算済みのカーネル行列をsklearnに指定しなければいけません。

We again set up the `QuantumKernel` class to calculate a kernel matrix using the `ZZFeatureMap`, and the BasicAer `qasm_simulator` using 1024 shots.

```
[8]: adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension,
                                         reps=2, entanglement='linear')

adhoc_backend = QuantumInstance(BasicAer.get_backend('qasm_simulator'), shots=1024,
                                 seed_simulator=seed, seed_transpiler=seed)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map, quantum_instance=adhoc_backend)
```

The scikit-learn spectral clustering algorithm allows us to define a [custom kernel] in two ways: by providing the kernel as a callable function or by precomputing the kernel matrix. Using the `QuantumKernel` class in qiskit, we can only use the latter.

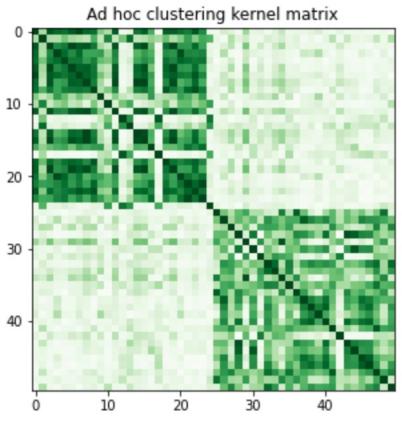
The following code precomputes and plots the kernel matrices before providing it to the scikit-learn spectral clustering algorithm, and scoring the labels using normalized mutual information, since we apriori know the class labels.

```
[9]: adhoc_matrix = adhoc_kernel.evaluate(x_vec=train_features)

plt.figure(figsize=(5, 5))
plt.imshow(np.asmatrix(adhoc_matrix), interpolation='nearest', origin='upper', cmap='Greens')
plt.title("Ad hoc clustering kernel matrix")
plt.show()

adhoc_spectral = SpectralClustering(2, affinity="precomputed")
cluster_labels = adhoc_spectral.fit_predict(adhoc_matrix)
cluster_score = normalized_mutual_info_score(cluster_labels, train_labels)

print(f'Clustering score: {cluster_score!r}')
```

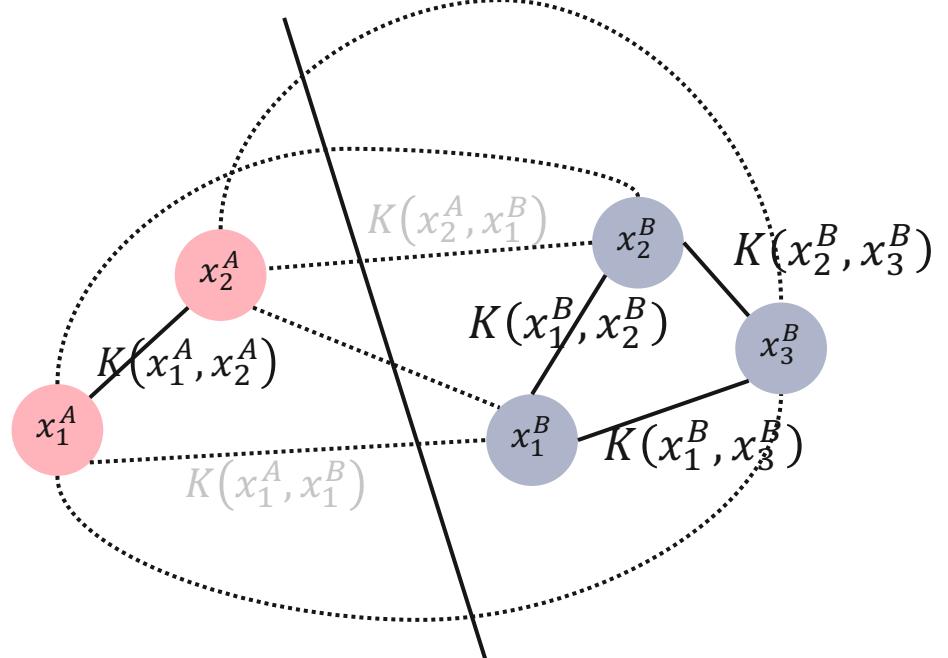


ラベルA同士のデータの内積	ラベルAとラベルBのデータの内積
ラベルBとラベルAのデータの内積	ラベルB同士のデータの内積

類似のデータとの内積が大きくなるため
左図のパターンの行列になる

Spectral Clustering:

カーネル行列をノード間の結びつきとして解釈し、
結びつきの弱い部分でグラフを切断する問題へ帰着



参考:

<https://www.slideshare.net/pecorarista/ss-51761860>

scikit-learn + 量子カーネルでのクラスタリング

クラスタリングのタスクではsklearnのSpectralClusteringクラスを使います。
この場合は計算済みのカーネル行列をsklearnに指定しなければいけません。

We again set up the `QuantumKernel` class to calculate a kernel matrix using the `ZZFeatureMap`, and the BasicAer `qasm_simulator` using 1024 shots.

```
[8]: adhoc_feature_map = ZZFeatureMap(feature_dimension=adhoc_dimension,
                                       reps=2, entanglement='linear')

adhoc_backend = QuantumInstance(BasicAer.get_backend('qasm_simulator'), shots=1024,
                                 seed_simulator=seed, seed_transpiler=seed)

adhoc_kernel = QuantumKernel(feature_map=adhoc_feature_map, quantum_instance=adhoc_backend)
```

The scikit-learn spectral clustering algorithm allows us to define a [custom kernel] in two ways: by providing the kernel as a callable function or by precomputing the kernel matrix. Using the `QuantumKernel` class in qiskit, we can only use the latter.

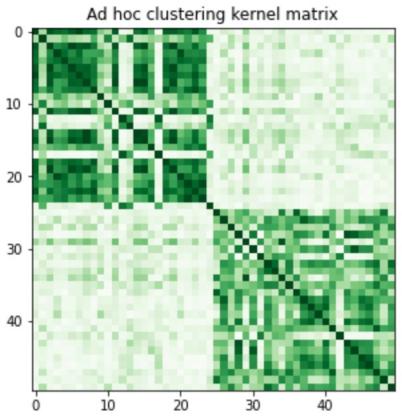
The following code precomputes and plots the kernel matrices before providing it to the scikit-learn spectral clustering algorithm, and scoring the labels using normalized mutual information, since we apriori know the class labels.

```
[9]: adhoc_matrix = adhoc_kernel.evaluate(x_vec=train_features)

plt.figure(figsize=(5, 5))
plt.imshow(np.asmatrix(adhoc_matrix), interpolation='nearest', origin='upper', cmap='Greens')
plt.title("Ad hoc clustering kernel matrix")
plt.show()

adhoc_spectral = SpectralClustering(2, affinity="precomputed")
cluster_labels = adhoc_spectral.fit_predict(adhoc_matrix)
cluster_score = normalized_mutual_info_score(cluster_labels, train_labels)

print(f'Clustering score: {cluster_score!r}')
```



類似のデータとの内積が大きくなるため
左図のパターンの行列になる

normalized_mutual_info_score:

クラスタリング結果(0と1が反転していてもOK)

[1, 1, 0, 0, 0, 1]

本来のラベル

[0, 0, 0, 1, 1, 1]

一致度の高くなるラベルの対応を見つけ、
ラベルの正解率を計算

クラスタリングラベル

[±, ±, 0, θ, θ, 1] ←→ [0, 0, 0, 1, 1, 1]

本来のラベル

[0, 0, 0, 1, 1, 1]

[0, 0, ±, 1, 1, θ] ←→ [0, 0, 0, 1, 1, 1] こちらの方が
ラベル対応が良い

そのほかの量子カーネルの活用候補

分類やクラスタリングの他にもカーネル行列を使った機械学習手法があるので、量子カーネルの活用が期待されます。

- Agglomerative clustering (凝縮型クラスタリング)

全てのデータ対の距離(カーネル)を計算し、連結距離を最小にするクラスターのペアを再帰的に結合する。

参考: <https://www.kamishima.net/archive/clustering.pdf>

- Support vector regression (サポートベクター回帰)

SVM分類と同様に境界線を定め、回帰曲線とする。

参考: <https://yuyumoyuyu.com/2021/01/10/supportvectorregression/>

- Ridge regression (リッジ回帰)

サポートベクター回帰と同様だが、誤差のペナルティ項が異なる。

参考: <https://datachemeng.com/rlasso/>

- Gaussian process regression (ガウス過程)

ガウスカーネルを用いる、ベイズ(確率過程)に基づく回帰手法。

参考: <https://www.slideshare.net/simizu706/stan-136716178>

- Principal component analysis (主成分分析)

多次元のデータを次元削減する手法。

カーネルを用いる場合はカーネルに対応するマッピングをしてから次元削減を行う。

参考: https://statistics.co.jp/reference/software_R/statR_9_principal.pdf

まとめ

- ・量子カーネル機械学習とはカーネル行列に量子状態の内積 $|\langle\phi(x_j)|\phi(x_i)\rangle|^2$ を使った機械学習である。
- ・SVMではカーネルを使っており、量子カーネルに置き換えたものは量子SVMと呼ばれる。
- ・カーネルは特徴量マッピングした空間での内積であり、
本チュートリアルはZZFeatureMapでマッピングしている。
- ・ad hoc datasetは量子SVMが分類しやすく、古典SVMが分類しにくいように
設計された特殊なデータセットである。
- ・カーネルはSVM分類だけでなくクラスタリングや回帰でも使われるため、
量子カーネルを応用が見込まれる。

より詳しい説明はQiitaに書きました:

- ・Qiskitで量子SVMを実装して性能評価してみた

https://qiita.com/ucc_white/items/f2ea0d019979dd675f82

QGAN

Agenda

- 量子カーネル機械学習
 - 1. カーネルとは
 - 2. scikit-learn + 量子カーネルでの分類(SVM)
 - 量子カーネルを計算する関数を使用
 - 量子カーネル行列そのものを使用
 - 3. scikit-learn + 量子カーネルでのクラスタリング
 - 量子カーネル行列そのものを使用
 - 4. そのほかの量子カーネルの活用候補
- qGAN(量子敵対的生成ネットワーク)
 - 1. 量子状態と確率分布
 - 2. qGANのアーキテクチャ
 - 1. Discriminator
 - 2. Generator
 - 3. 量子回路
 - 3. 学習方法
 - 1. 損失関数
 - 2. 相対エントロピー
 - 3. 結果

量子状態と確率分布

qGAN(量子敵対的生成ネットワーク)なるものを使って確率分布を表現する量子状態を作ることを目指します。

学習データ $X = \{x^0, \dots, x^{k-1}\}$ を使って表現したい確率分布を作る量子回路をqGANが学習していきます。

qGANs for Loading Random Distributions

Given k -dimensional data samples, we employ a quantum Generative Adversarial Network (qGAN) to learn the data's underlying random distribution and to load it directly into a quantum state:

$$|g_\theta\rangle = \sum_{j=0}^{2^n-1} \sqrt{p_\theta^j} |j\rangle$$

where p_θ^j describe the occurrence probabilities of the basis states $|j\rangle$.

The aim of the qGAN training is to generate a state $|g_\theta\rangle$ where p_θ^j , for $j \in \{0, \dots, 2^n - 1\}$, describe a probability distribution that is close to the distribution underlying the training data $X = \{x^0, \dots, x^{k-1}\}$.

For further details please refer to [Quantum Generative Adversarial Networks for Learning and Loading Random Distributions Zoufal, Lucchi, Woerner \[2019\]](#).

For an example of how to use a trained qGAN in an application, the pricing of financial derivatives, please see the [Option Pricing with qGANs](#) tutorial.

```
[1]: import numpy as np
seed = 71
np.random.seed = seed

import matplotlib.pyplot as plt
%matplotlib inline

from qiskit import QuantumRegister, QuantumCircuit, BasicAer
from qiskit.circuit.library import TwoLocal, UniformDistribution

from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_machine_learning.algorithms import NumPyDiscriminator, QGAN

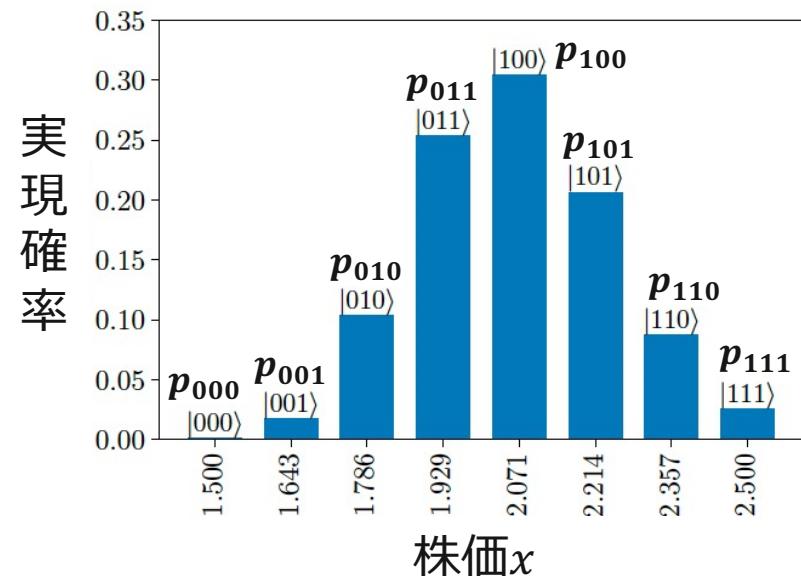
algorithm_globals.random_seed = seed
```

金融での応用事例

将来の株価 x の分布は対数正規分布で近似できる
(Black-Scholes model)

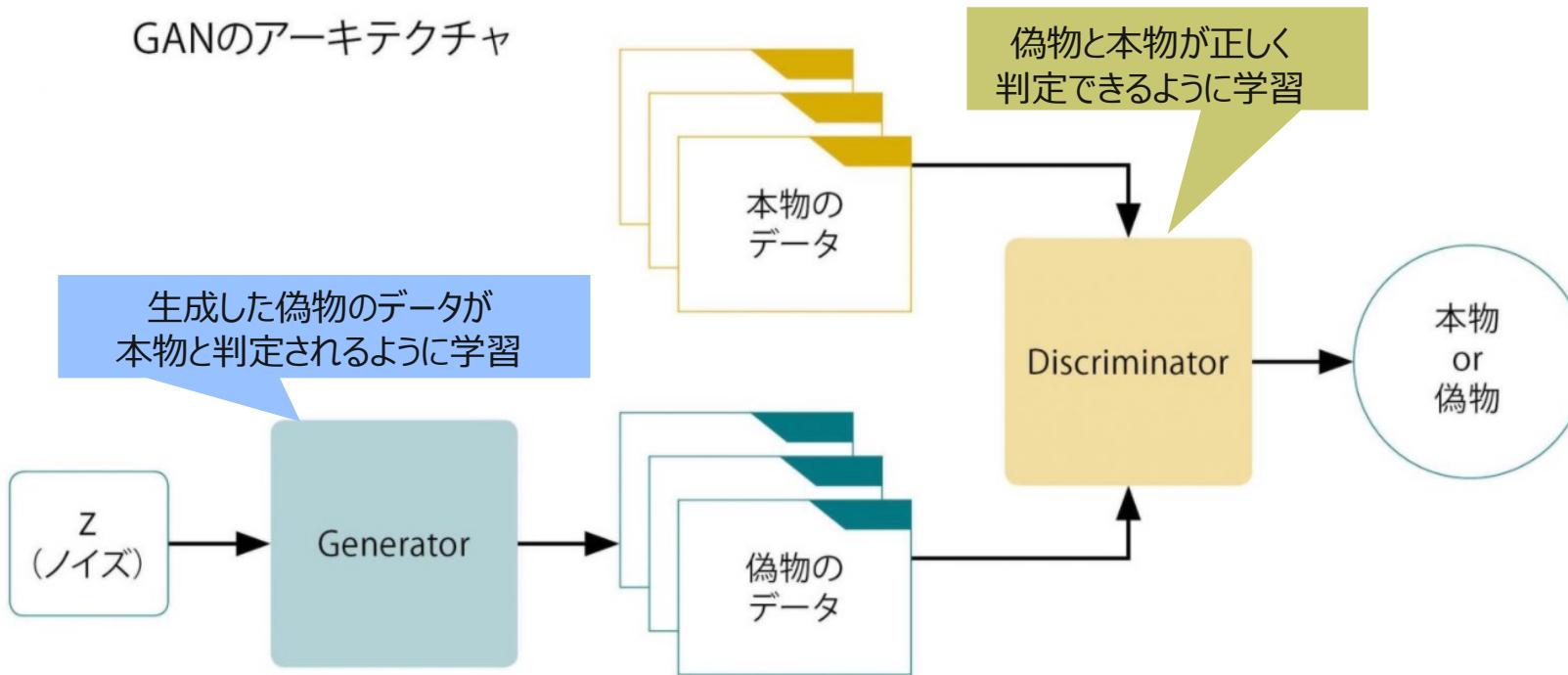
$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{(\log x - \mu)^2}{2\sigma^2}\right)$$

この分布を離散化して量子状態で表してみる



GANとは

GAN(Generative Adversarial Network)は2つのニューラルネットを競わせながら学習させる教師なし学習です。古典GANでは画像の擬似生成に使われることが多いです。



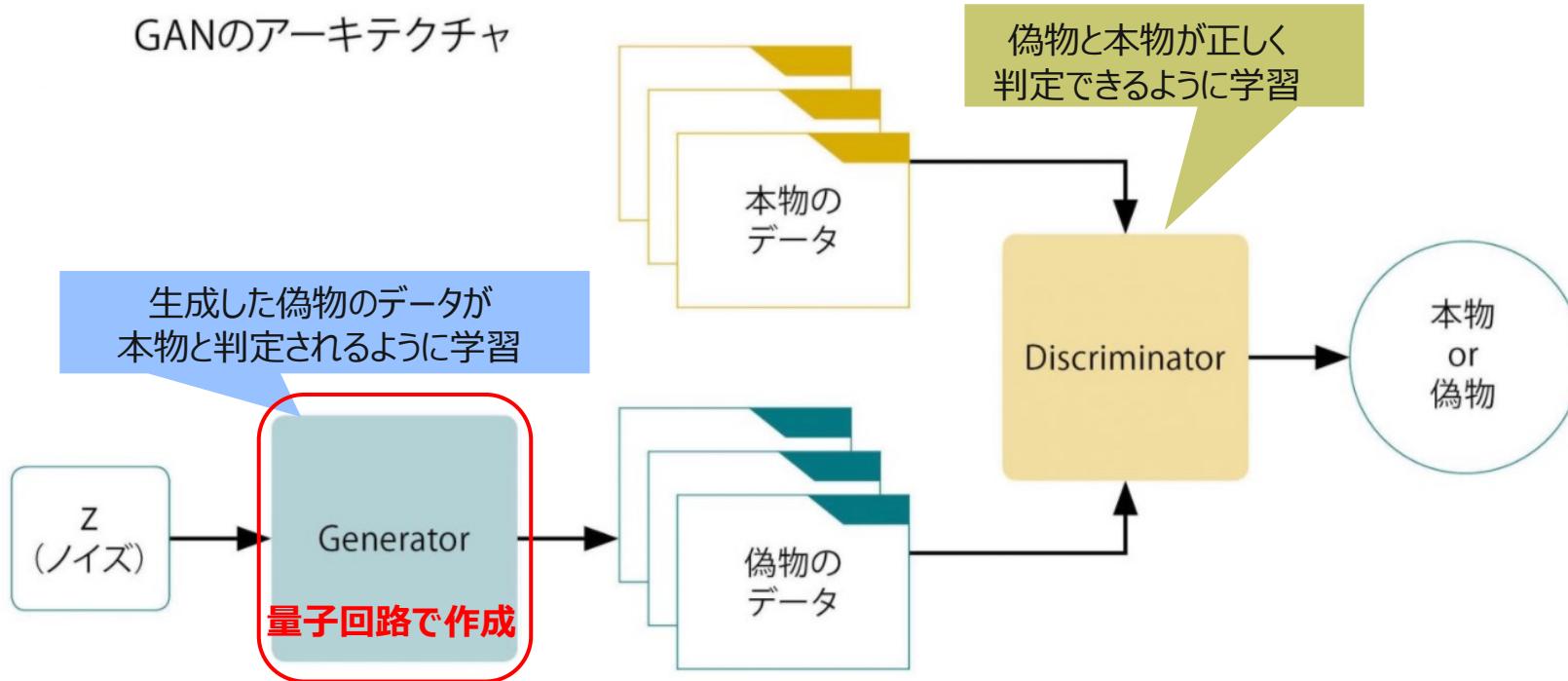
GANを利用することで実在しない画像を生成することができる
笑顔を徐々に変化させていく



図の引用: <https://bit.ly/2UqbtPa>

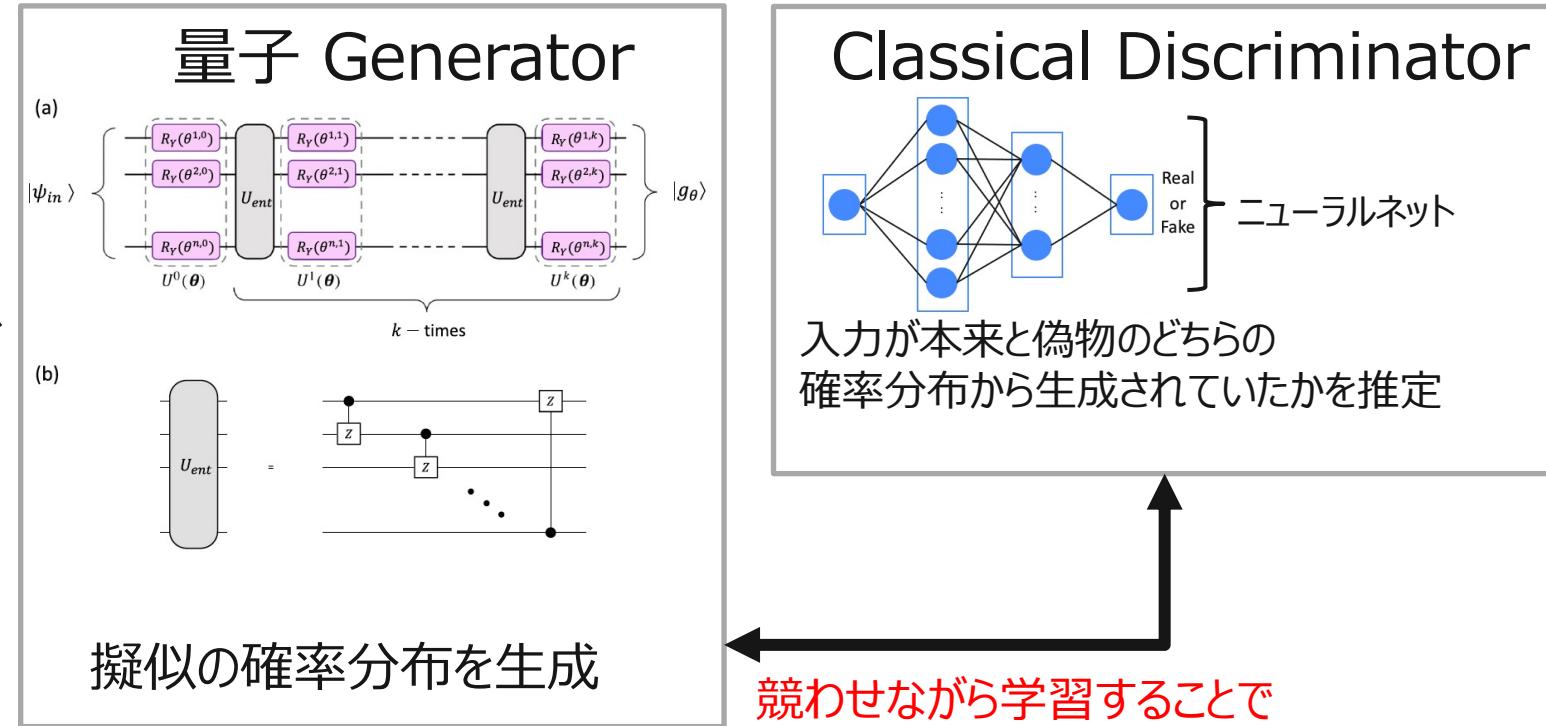
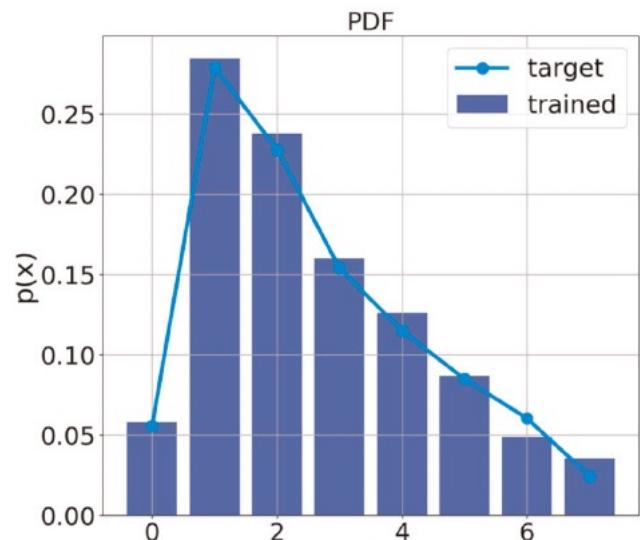
qGANとは

qGANではGenerator部分を量子回路で置き換えます。



qGANのアーキテクチャ

qGANでは量子部分のGeneratorと古典部分のDiscriminatorを競わせながら学習を行います。
そして今回擬似生成するのは画像ではなく、確率分布です。



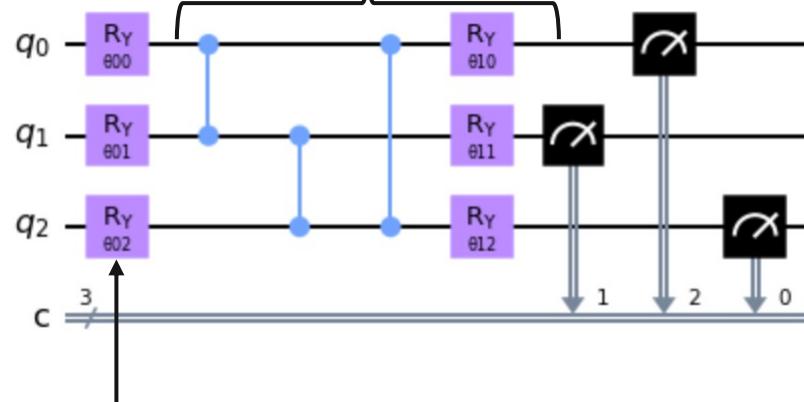
*Generator部分だけ量子回路で置き換えたのは、訓練完了後の量子状態生成に使うのはGenerator部分のみだからです。一般にDiscriminatorはモデルの訓練にのみ使用します。

qGANのアーキテクチャー

量子Generatorは変分量子回路であり、回転ゲートの角度 θ_{ij} を適切に設定することで本物に近い確率分布を生成。
古典Discriminatorは古典ニューラルネットであり、エッジの重み ϕ_{mn} を適切に設定することで入力を識別。

量子Generator (G_θ)

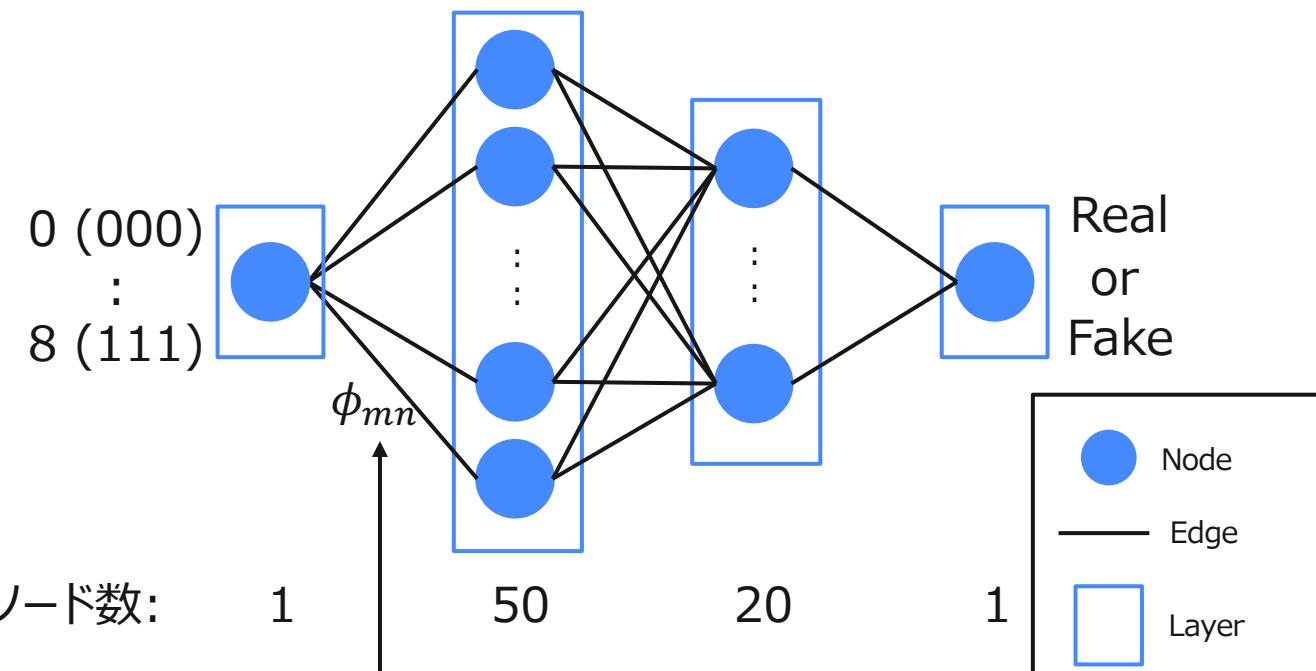
繰り返し数 $k=1$
 k はハイパーパラメータとして調整



所望の確率分布を再現するように
パラメータ θ_{ij} を学習

古典Discriminator (D_ϕ)

入力層 隠れ層1 隠れ層2 出力層



ノード数:

1

50

20

1

本物と偽物を区別できるようにエッジの重み ϕ_{mn} を学習

学習データの準備

Tutorialでは対数正規分布を量子状態として表現していくことを目指します。
対数正規分布からサンプリングしたもの学習データとします。

Load the Training Data

First, we need to load the k -dimensional training data samples (here $k=1$).

Next, the data resolution is set, i.e. the min/max data values and the number of qubits used to represent each data dimension.

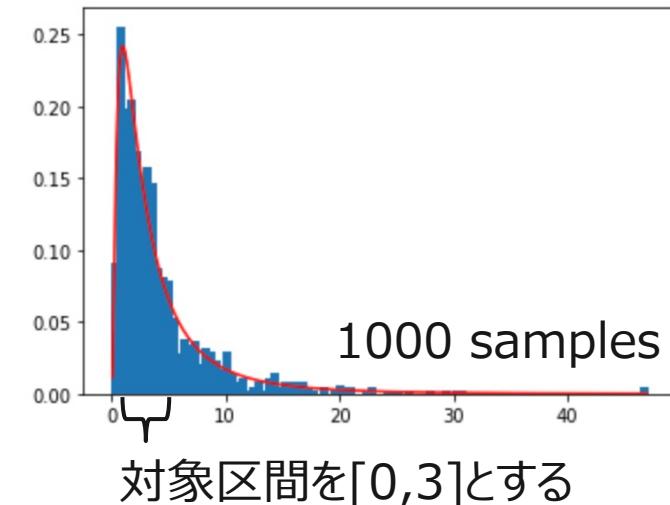
```
[2]: # Number training data samples
N = 1000

# Load data samples from log-normal distribution with mean=1 and standard deviation=1
mu = 1
sigma = 1
real_data = np.random.lognormal(mean=mu, sigma=sigma, size=N)

# Set the data resolution
# Set upper and lower data values as list of k min/max data values [[min_0,max_0],..., [min_k-1,max_k-1]]
bounds = np.array([0., 3.]) 分布の対象区間
# Set number of qubits per data dimension as list of k qubit values [#q_0,...,#q_k-1]
num_qubits = [2] 2量子ビットを使うので4つのBINで離散化
k = len(num_qubits)
```

平均1、標準偏差1の
対数正規分布からサンプリング

4



対象区間を[0,3]とする

学習データの準備

Tutorialでは明示的に行われていませんが、QGAN内部でサンプリングデータを加工して離散化処理をしています。

Initialize the qGAN

The qGAN consists of a quantum generator G_θ , i.e., an ansatz, and a classical discriminator D_ϕ , a neural network.

To implement the quantum generator, we choose a depth-1 ansatz that implements R_Y rotations and CZ gates which takes a uniform distribution as an input state. Notably, for $k > 1$ the generator's parameters must be chosen carefully. For example, the circuit depth should be > 1 because higher circuit depths enable the representation of more complex structures.

The classical discriminator used here is based on a neural network implementation using NumPy. There is also a discriminator based on PyTorch which is not installed by default when installing Qiskit - see [Optional Install](#) for more information.

Here, both networks are updated with the ADAM optimization algorithm (ADAM is qGAN optimizer default).

```
[3]: # Set number of training epochs
# Note: The algorithm's runtime can be shortened by reducing the number of training epochs.
num_epochs = 10
# Batch size
batch_size = 100

# Initialize qGAN
qgan = QGAN(real_data, bounds, num_qubits, batch_size, num_epochs, snapshot_dir=None)
qgan.seed = 1

# Set quantum instance to run the quantum generator
quantum_instance = QuantumInstance(backend=BasicAer.get_backend('statevector_simulator'),
                                    seed_transpiler=seed, seed_simulator=seed)

# Set entangler map
entangler_map = [[0, 1]]

# Set an initial state for the generator circuit
init_dist = UniformDistribution(sum(num_qubits))

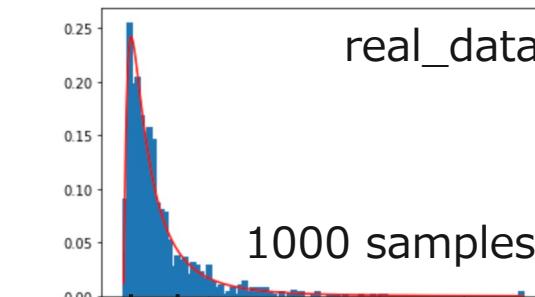
# Set the ansatz circuit
ansatz = TwoLocal(int(np.sum(num_qubits)), 'ry', 'cz', entanglement=entangler_map, reps=1)

# Set generator's initial parameters - in order to reduce the training time and hence the
# total running time for this notebook
init_params = [3., 1., 0.6, 1.6]

# You can increase the number of training epochs and use random initial parameters.
# init_params = np.random.rand(ansatz.num_parameters_settable) * 2 * np.pi

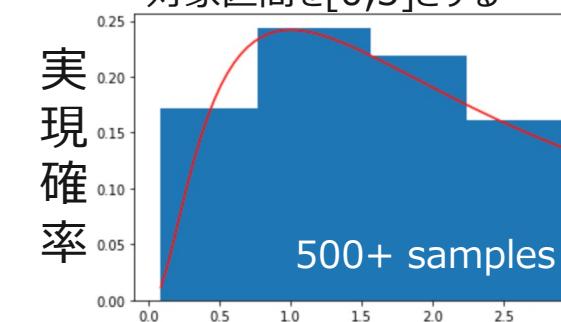
# Set generator circuit by adding the initial distribution in front of the ansatz
g_circuit = ansatz.compose(init_dist, front=True)
```

目的の確率分布からサンプリング



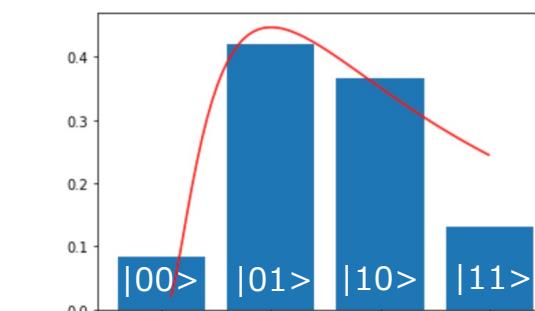
real_data

1000 samples



実現確率

500+ samples



*確率分布曲線は区間内で再度規格化

対象区間(bounds)の抽出
(truncate)

[00, 01, 10, 11]
で離散化

サンプリング例:
[1, 2, 1, 0, 0, 1, 2, 3, ...]
500+ samples

学習データの準備

次に加工したデータセットをバッチに分割します。qGANのパラメータの更新回数は、エポック数×バッチ数 です

Initialize the qGAN

The qGAN consists of a quantum generator G_θ , i.e., an ansatz, and a classical discriminator D_ϕ , a neural network.

To implement the quantum generator, we choose a depth-1 ansatz that implements R_Y rotations and CZ gates which takes a uniform distribution as an input state. Notably, for $k > 1$ the generator's parameters must be chosen carefully. For example, the circuit depth should be > 1 because higher circuit depths enable the representation of more complex structures.

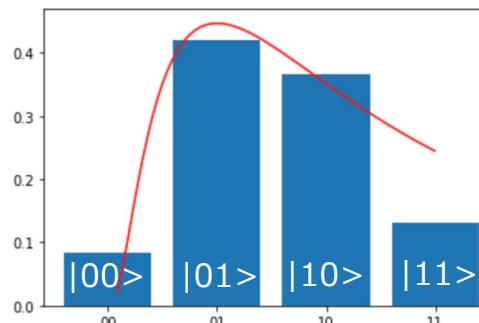
The classical discriminator used here is based on a neural network implementation using NumPy. There is also a discriminator based on PyTorch which is not installed by default when installing Qiskit - see [Optional Install](#) for more information.

Here, both networks are updated with the ADAM optimization algorithm (ADAM is qGAN optimizer default).

```
[3]: # Set number of training epochs  
# Note: The algorithm's runtime can be shortened by reducing the number of training epochs.  
num_epochs = 10  
# Batch size  
batch_size = 100
```

```
# Initialize qGAN  
qgan = QGAN(real_data, bounds, num_qubits, batch_size, num_epochs, snapshot_dir=None)  
qgan.seed = 1  
# Set quantum instance to run the quantum generator  
quantum_instance = QuantumInstance(backend=BasicAer.get_backend('statevector_simulator'),  
                                     seed_transpiler=seed, seed_simulator=seed)  
  
# Set entangler map  
entangler_map = [[0, 1]]  
  
# Set an initial state for the generator circuit  
init_dist = UniformDistribution(sum(num_qubits))  
  
# Set the ansatz circuit  
ansatz = TwoLocal(int(np.sum(num_qubits)), 'ry', 'cz', entanglement=entangler_map, reps=1)  
  
# Set generator's initial parameters - in order to reduce the training time and hence the  
# total running time for this notebook  
init_params = [3., 1., 0.6, 1.6]  
  
# You can increase the number of training epochs and use random initial parameters.  
# init_params = np.random.rand(ansatz.num_parameters_settable) * 2 * np.pi  
  
# Set generator circuit by adding the initial distribution in front of the ansatz  
g_circuit = ansatz.compose(init_dist, front=True)
```

加工済みサンプル



*確率分布曲線は区間内で再度規格化

サンプリング例:
[1, 2, 1, 0, 0, 1, 2, 3, ...]
500+ samples

サンプルをバッチサイズ(=100)毎に分割して
Discriminatorの入力群とする

頻度	00	01	10	11
batch_1	13	45	31	11
batch_2	15	45	32	8
batch_3	0	43	42	15
batch_4	12	40	35	13
batch_5	5	37	42	15

*100に満たない6番目のバッチは除外

量子Generator回路

次に量子Generator回路の初期状態を設定します。初期状態は一様分布か初期分布が設定可能です。

Initialize the qGAN

The qGAN consists of a quantum generator G_θ , i.e., an ansatz, and a classical discriminator D_ϕ , a neural network.

To implement the quantum generator, we choose a depth-1 ansatz that implements R_Y rotations and CZ gates which takes a uniform distribution as an input state. Notably, for $k > 1$ the generator's parameters must be chosen carefully. For example, the circuit depth should be > 1 because higher circuit depths enable the representation of more complex structures.

The classical discriminator used here is based on a neural network implementation using NumPy. There is also a discriminator based on PyTorch which is not installed by default when installing Qiskit - see [Optional Install](#) for more information.

Here, both networks are updated with the ADAM optimization algorithm (ADAM is qGAN optimizer default).

```
[3]: # Set number of training epochs
# Note: The algorithm's runtime can be shortened by reducing the number of training epochs.
num_epochs = 10
# Batch size
batch_size = 100

# Initialize qGAN
qgan = QGAN(real_data, bounds, num_qubits, batch_size, num_epochs, snapshot_dir=None)
qgan.seed = 1

# Set quantum instance to run the quantum generator
quantum_instance = QuantumInstance(backend=BasicAer.get_backend('statevector_simulator'),
                                    seed_transpiler=seed, seed_simulator=seed)

# Set entangler map
entangler_map = [[0, 1]]

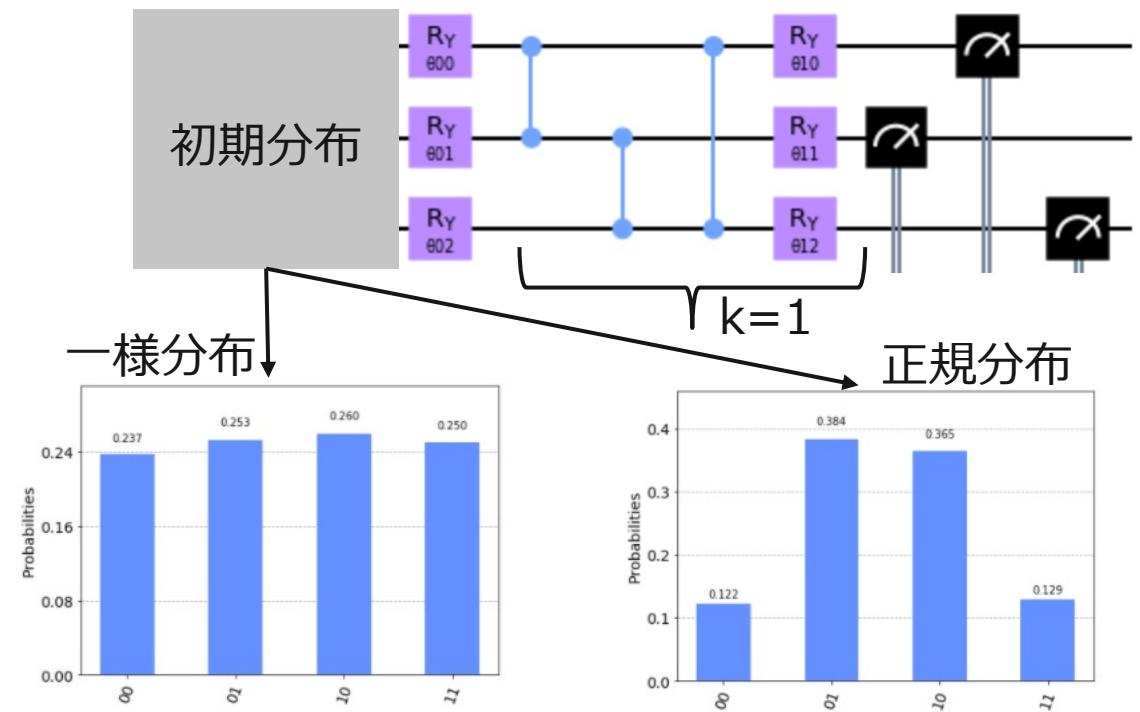
# Set an initial state for the generator circuit
init_dist = UniformDistribution(sum(num_qubits))

# Set the ansatz circuit
ansatz = TwoLocal(int(np.sum(num_qubits)), 'ry', 'cz', entanglement=entangler_map, reps=1)

# Set generator's initial parameters - in order to reduce the training time and hence the
# total running time for this notebook
init_params = [3., 1., 0.6, 1.6]

# You can increase the number of training epochs and use random initial parameters.
# init_params = np.random.rand(ansatz.num_parameters_settable) * 2 * np.pi

# Set generator circuit by adding the initial distribution in front of the ansatz
g_circuit = ansatz.compose(init_dist, front=True)
```



今回の回路は2量子ビット回路だがエンタングル方法を示すため3量子ビット表示

量子Generator回路

変分量子回路の構成(ansatz)はRyゲートとCZゲートで作成します。こうすると $\theta_{ij} = 0$ で振幅が不変になります。

Initialize the qGAN

The qGAN consists of a quantum generator G_θ , i.e., an ansatz, and a classical discriminator D_ϕ , a neural network.

To implement the quantum generator, we choose a depth-1 ansatz that implements R_Y rotations and CZ gates which takes a uniform distribution as an input state. Notably, for $k > 1$ the generator's parameters must be chosen carefully. For example, the circuit depth should be > 1 because higher circuit depths enable the representation of more complex structures.

The classical discriminator used here is based on a neural network implementation using NumPy. There is also a discriminator based on PyTorch which is not installed by default when installing Qiskit - see [Optional Install](#) for more information.

Here, both networks are updated with the ADAM optimization algorithm (ADAM is qGAN optimizer default).

```
[3]: # Set number of training epochs
# Note: The algorithm's runtime can be shortened by reducing the number of training epochs.
num_epochs = 10
# Batch size
batch_size = 100

# Initialize qGAN
qgan = QGAN(real_data, bounds, num_qubits, batch_size, num_epochs, snapshot_dir=None)
qgan.seed = 1

# Set quantum instance to run the quantum generator
quantum_instance = QuantumInstance(backend=BasicAer.get_backend('statevector_simulator'),
                                    seed_transpiler=seed, seed_simulator=seed)

# Set entangler map
entangler_map = [[0, 1]] ]量子ビット0と1をエンタングルさせるように指定

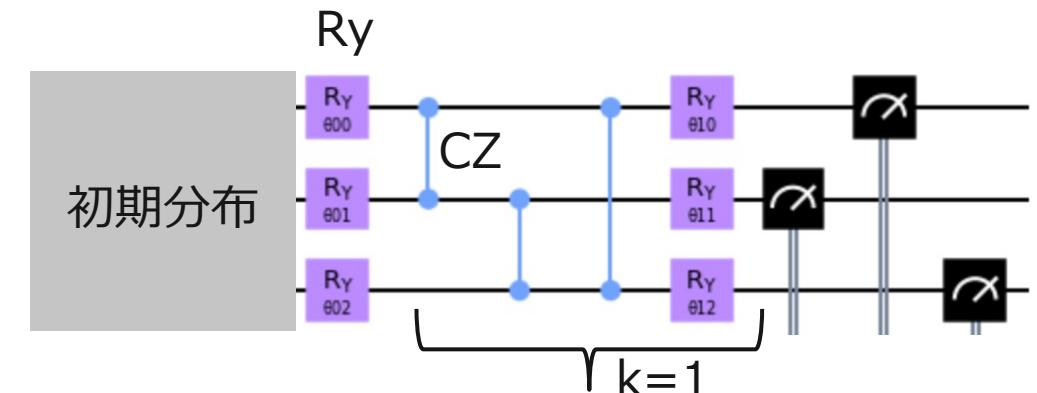
# Set an initial state for the generator circuit
init_dist = UniformDistribution(sum(num_qubits))

# Set the ansatz circuit
ansatz = TwoLocal(int(np.sum(num_qubits)), 'ry', 'cz', entanglement=entangler_map, reps=1) ]繰り返し数は1回を指定

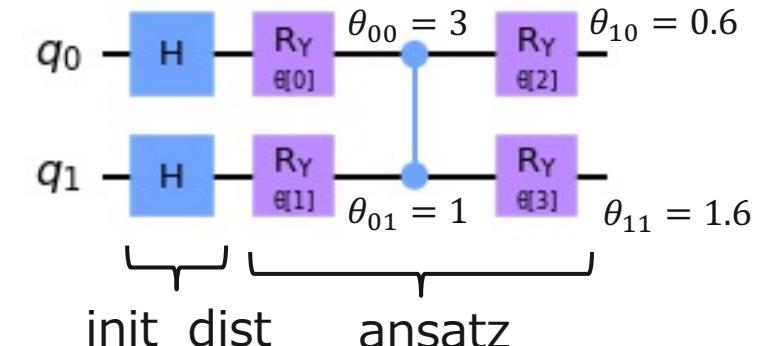
# Set generator's initial parameters - in order to reduce the training time and hence the
# total running time for this notebook
init_params = [3., 1., 0.6, 1.6] ]初期パラメータ $\theta_{ij}$ を設定(この例では学習済みパラメータを使っている…)
# You can increase the number of training epochs and use random initial parameters.
# init_params = np.random.rand(ansatz.num_parameters_settable) * 2 * np.pi

# Set generator circuit by adding the initial distribution in front of the ansatz
g_circuit = ansatz.compose(init_dist, front=True)]初期分布回路とansatzを結合
```

量子Generator回路(3量子ビットの場合)



今回の回路



学習方法

Generator G_θ のパラメータ θ_{ij} (Ryの回転角)とDiscriminator D_ϕ のパラメータ ϕ_{mn} (ニューラルネットの重み)の更新方法を考えます。
qGANではnon-saturating lossという損失関数の最小化/最大化でパラメータ更新をします。

Run the qGAN Training

During the training the discriminator's and the generator's parameters are updated alternately w.r.t the following loss functions:

$$L_G(\phi, \theta) = -\frac{1}{m} \sum_{l=1}^m [\log(D_\phi(g^l))]$$

and

$$L_D(\phi, \theta) = \frac{1}{m} \sum_{l=1}^m [\log D_\phi(x^l) + \log(1 - D_\phi(g^l))],$$

with m denoting the batch size and g^l describing the data samples generated by the quantum generator.

Please note that the training, for the purpose of this notebook, has been kept briefer by the selection of a known initial point (`init_params`). Without such prior knowledge be aware training may take some while.

```
[4]: # Run qGAN
result = qgan.run(quantum_instance)
```

```
[5]: print('Training results:')
for key, value in result.items():
    print(f' {key} : {value}')
```

```
Training results:
params_d : [ 0.0364864  0.61061501 -0.48109538 ... -0.16695524 -0.20145609
-0.08609052]
params_g : [2.95010894 0.95007515 0.55006681 1.64997574]
loss_d : 0.692
loss_g : [0.7367]
rel_entr : 0.1534
```

m : バッチサイズ

x^l : l 番目のオリジナルデータ

g^l : l 番目の擬似生成データ

$D_\phi(x)$ オリジナルデータがオリジナルデータ
と判定される確率 (確信度)

$D_\phi(g)$: 擬似生成データがオリジナルデータ
と判定される確率 (確信度)

Generatorの損失関数:

$$L_G(\phi, \theta) = -\frac{1}{m} \underbrace{\sum_{l=1}^m [\log(D_\phi(g^l))]}_{D_\phi(g) \text{が増加すると } L_G(\phi, \theta) \text{が減少}}$$

→ $L_G(\phi, \theta)$ を最小化していけばよい

学習方法

Generator G_θ のパラメータ θ_{ij} (Ryの回転角)とDiscriminator D_ϕ のパラメータ ϕ_{mn} (ニューラルネットの重み)の更新方法を考えます。
qGANではnon-saturating lossという損失関数の最小化/最大化でパラメータ更新をします。

Run the qGAN Training

During the training the discriminator's and the generator's parameters are updated alternately w.r.t the following loss functions:

$$L_G(\phi, \theta) = -\frac{1}{m} \sum_{l=1}^m [\log(D_\phi(g^l))]$$

and

$$L_D(\phi, \theta) = \frac{1}{m} \sum_{l=1}^m [\log D_\phi(x^l) + \log(1 - D_\phi(g^l))],$$

with m denoting the batch size and g^l describing the data samples generated by the quantum generator.

Please note that the training, for the purpose of this notebook, has been kept briefer by the selection of a known initial point (`init_params`). Without such prior knowledge be aware training may take some while.

```
[4]: # Run qGAN
result = qgan.run(quantum_instance)
```

```
[5]: print('Training results:')
for key, value in result.items():
    print(f' {key} : {value}')
```

```
Training results:
params_d : [ 0.0364864  0.61061501 -0.48109538 ... -0.16695524 -0.20145609
-0.08609052]
params_g : [ 2.95010894  0.95007515  0.55006681  1.64997574]
loss_d : 0.692
loss_g : [0.7367]
rel_entr : 0.1534
```

m : バッチサイズ

x^l : l 番目のオリジナルデータ

g^l : l 番目の擬似生成データ

$D_\phi(x)$: オリジナルデータがオリジナルデータ
と判定される確率 (確信度)

$D_\phi(g)$: 擬似生成データがオリジナルデータ
と判定される確率 (確信度)

Discriminatorの損失関数:

$$L_D(\phi, \theta) = \frac{1}{m} \sum_{l=1}^m [\underbrace{\log D_\phi(x^l)}_{D_\phi(x) \text{が増加すると}} + \underbrace{\log(1 - D_\phi(g^l))}_{D_\phi(g) \text{が減少すると}}]$$

$D_\phi(x)$ が増加すると
 $L_D(\phi, \theta)$ が増加

$D_\phi(g)$ が減少すると
 $L_D(\phi, \theta)$ が増加

→ $L_D(\phi, \theta)$ を最大化していけばよい

学習方法

GANが提案された際*はmin-max lossという損失関数が使われていました。

しかしGeneratorの学習が停止(saturate)しやすい欠点があり、qGANではnon-saturating lossが使われています。

min-max loss

$$\min_G \max_D \frac{1}{m} \sum_{l=1}^m [\log D_\phi(x^l) + \log(1 - D_\phi(g^l))]$$

Generator

$$\begin{aligned} \min_G L_G(\phi, \theta) &= \min_G \frac{1}{m} \sum_{l=1}^m [\log D_\phi(x^l) + \log(1 - D_\phi(g^l))] \\ &= \min_G \frac{1}{m} \sum_{l=1}^m [\log(1 - D_\phi(g^l))] \end{aligned}$$

Discriminator

$$\max_D L_D(\phi, \theta) = \max_D \frac{1}{m} \sum_{l=1}^m [\log D_\phi(x^l) + \log(1 - D_\phi(g^l))]$$

m : バッチサイズ

x^l : l 番目のオリジナルデータ

g^l : l 番目の擬似生成データ

$D_\phi(x)$: オリジナルデータがオリジナルデータと判定される確率 (確信度)

$D_\phi(g)$: 擬似生成データがオリジナルデータと判定される確率 (確信度)



学習が止まりやすい(Discriminatorの学習が早すぎる場合など)

non-saturating lossが提案される (学習が安定する)

$$L_G(\phi, \theta) = -\frac{1}{m} \sum_{l=1}^m [\log(D_\phi(g^l))]$$

GANの提案論文: [Ian J. Goodfellow, et. al. 2014](#)

min-max lossとnon-saturating lossの詳細: <https://neptune.ai/blog/gan-loss-functions>

学習方法

画像生成のGANではDiscriminatorの入力が画像なので、本物か偽物かの識別をしていることがイメージしやすいです。今回の確率分布生成のGANでは入力が{0, 1, 2, 3}の数字であり、条件付き分布を計算していることに対応します。

画像生成GANの場合(例: 手書き数字4の生成)

オリジナルデータ

$$D_{\phi}(\text{4}) = 0.85$$

擬似データ

$$D_{\phi}(\text{4}) = 0.32$$

擬似データ画像:

<https://link.springer.com/article/10.1007/s11042-019-08600-2/figures/6>

確率分布生成GANの場合(Generatorが作った分布が一様分布とする)

Discriminatorの入力で「0」が来た場合

$$D_{\phi}(0) = P(\text{input} \in X | \text{input} = 0) = 0.21$$

X はオリジナルデータの集合

*Discriminatorは入力の分布を直接知っているわけではないが、
学習の結果知っているかのように振る舞う

m : バッチサイズ

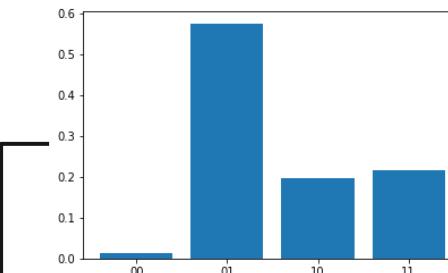
x^l : l 番目のオリジナルデータ

g^l : l 番目の擬似生成データ

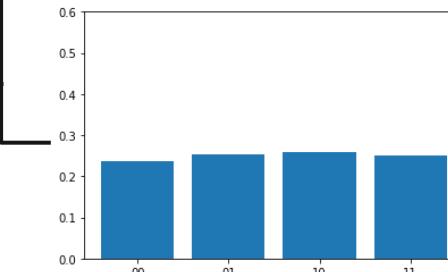
$D_{\phi}(x)$: オリジナルデータがオリジナルデータ
と判定される確率 (確信度)

$D_{\phi}(g)$: 擬似生成データがオリジナルデータ
と判定される確率 (確信度)

サンプルデータの分布



擬似生成された分布



学習方法

qGANでは、次のようなアルゴリズムで学習(パラメータ更新)が進んでいます。

```
[4]: # Run qGAN  
result = qgan.run(quantum_instance)
```

mはバッチサイズ

```
for i in range(エポック数):  
    for j in range(バッチ数):  
        (1) j番目のバッチのオリジナルデータからm個のデータ $\{x_j^0, \dots, x_j^{m-1}\}$ を取得  
        (2) Generatorからm個の擬似データ $\{g_j^0, \dots, g_j^{m-1}\}$ を生成  
        (3) オリジナルデータと擬似データを正しく区別できるように、  
            Discriminatorの損失関数 $L_D(\phi, \theta)$ を最大化する方向に $\phi$ を更新
```

$$L_D(\phi, \theta) = \frac{1}{m} \sum_{l=1}^m \left[\log D_\phi(x^l) + \log (1 - D_\phi(g^l)) \right]$$

```
(4) 擬似データをオリジナルデータと誤分類させるように、  
    Generatorの損失関数 $L_G(\phi, \theta)$ を最小化する方向に $\theta$ を更新
```

$$L_G(\phi, \theta) = -\frac{1}{m} \sum_{l=1}^m \left[\log (D_\phi(g^l)) \right]$$

学習方法

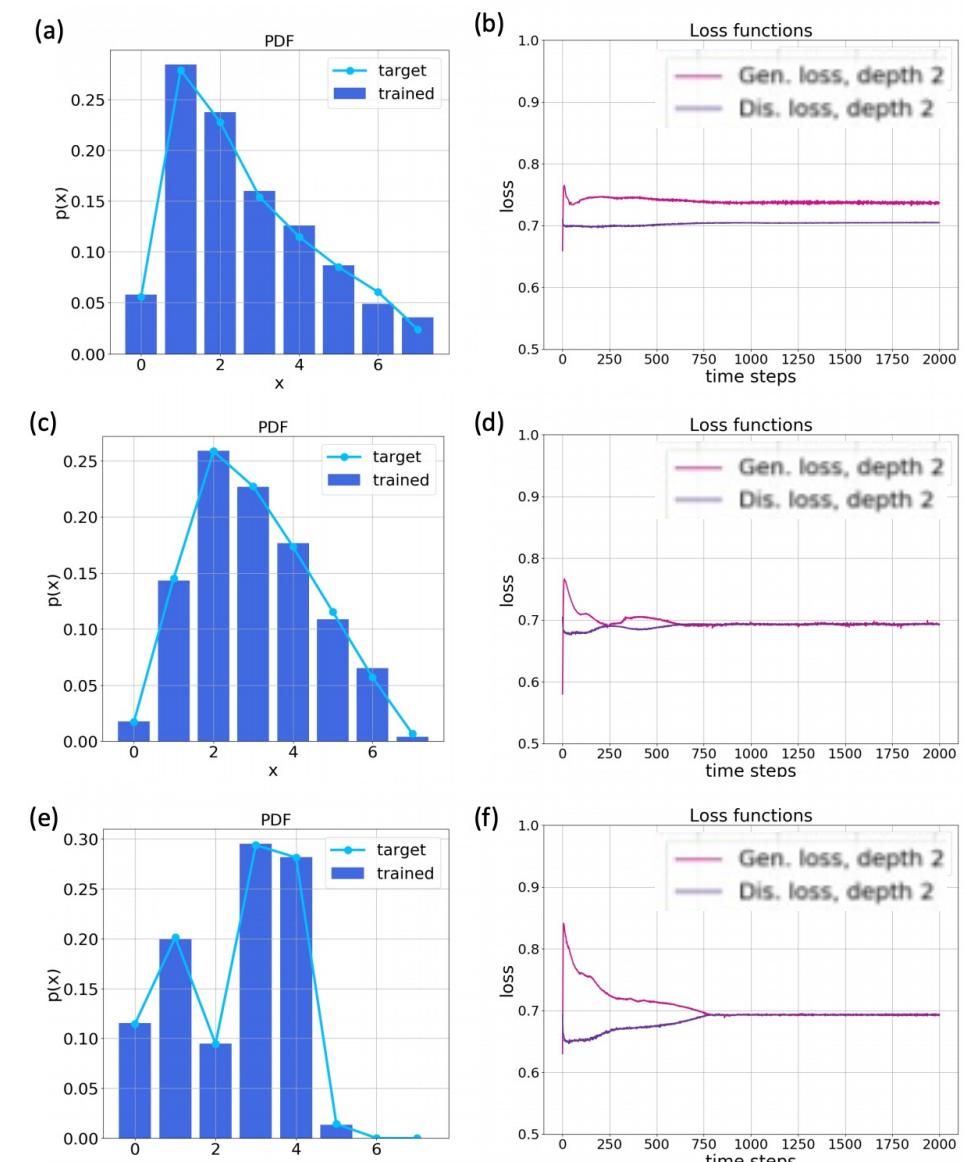
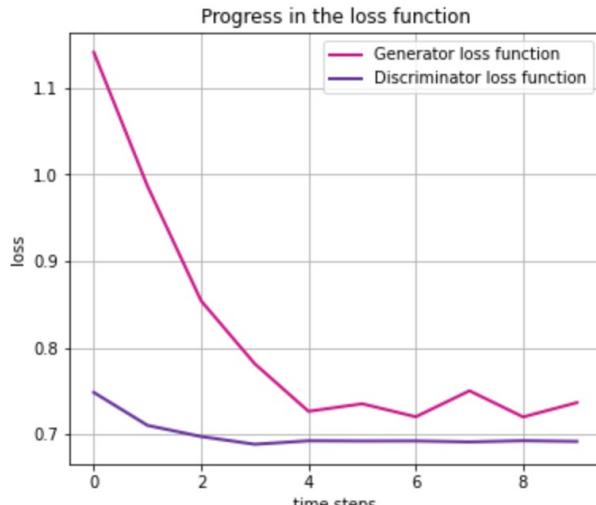
学習中の損失関数の推移を見るとGeneratorの損失関数 $L_G(\phi, \theta)$ が減少しているのが分かります。
(Discriminatorの損失関数 $L_D(\phi, \theta)$ は本来増加していくはず。。)

Training Progress & Outcome 🎉

Now, we plot the evolution of the generator's and the discriminator's loss functions during the training, as well as the progress in the relative entropy between the trained and the target distribution.

Finally, we also compare the cumulative distribution function (CDF) of the trained distribution to the CDF of the target distribution.

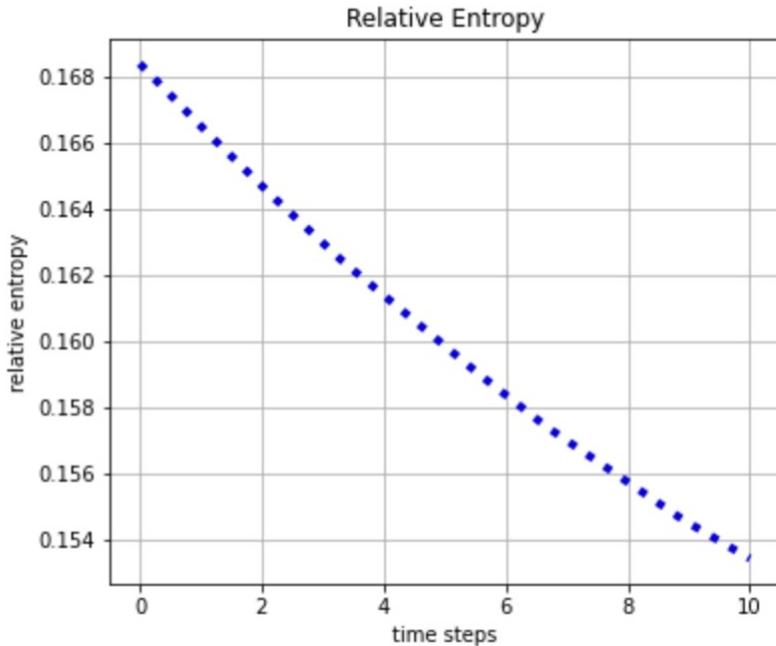
```
[6]: # Plot progress w.r.t the generator's and the discriminator's loss function
t_steps = np.arange(num_epochs)
plt.figure(figsize=(6,5))
plt.title("Progress in the loss function")
plt.plot(t_steps, qgan.g_loss, label='Generator loss function', color='mediumvioletred', linewidth=2)
plt.plot(t_steps, qgan.d_loss, label='Discriminator loss function', color='rebeccapurple', linewidth=2)
plt.grid()
plt.legend(loc='best')
plt.xlabel('time steps')
plt.ylabel('loss')
plt.show()
```



学習方法

相対エントロピーによって本来の確率分布と生成された確率分布の差異を定量化します。

```
[7]: # Plot progress w.r.t relative entropy
plt.figure(figsize=(6,5))
plt.title('Relative Entropy')
plt.plot(np.linspace(0, num_epochs, len(qgan.rel_entr)), qgan.rel_entr, color='mediumblue', lw=4,
ls=':')
plt.grid()
plt.xlabel('time steps')
plt.ylabel('relative entropy')
plt.show()
```



$$\text{相対エントロピー} \\ D_{\text{KL}}(P\|Q) = \sum_j p_j \log \frac{p_j}{q_j}.$$

p_j : 生成された分布Pから変数jが出現する確率
 q_j : 本来の分布Qから変数jが出現する確率

次のように擬似データ/オリジナルの分布が得られたとする

Probability	00	01	10	11
Generated	0.04	0.43	0.42	0.10
Original	0.08	0.42	0.37	0.13

この時

$$D_{\text{KL}} = 0.04 \log \frac{0.04}{0.08} + 0.43 \log \frac{0.43}{0.42} + 0.42 \log \frac{0.42}{0.37} + 0.10 \log \frac{0.10}{0.13} = 0.02$$

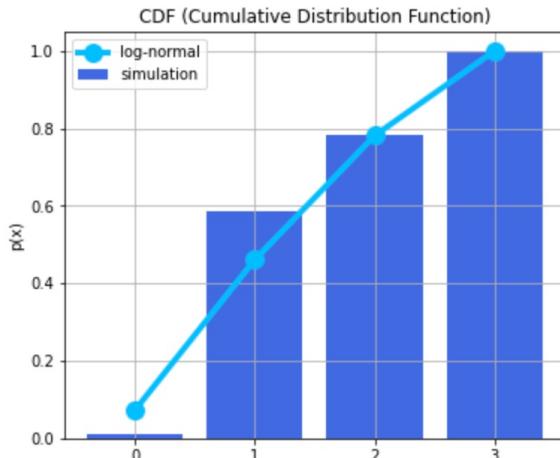
小さい値になるほど類似

学習方法

最後に得られた分布をプロットします。Tutorialでは累積分布を表示しています。

```
[8]: #Plot the CDF of the resulting distribution against the target distribution, i.e. log-normal
log_normal = np.random.lognormal(mean=1, sigma=1, size=100000)
log_normal = np.round(log_normal)
log_normal = log_normal[log_normal <= bounds[1]]
temp = []
for i in range(int(bounds[1] + 1)):
    temp += [np.sum(log_normal==i)]
log_normal = np.array(temp / sum(temp))

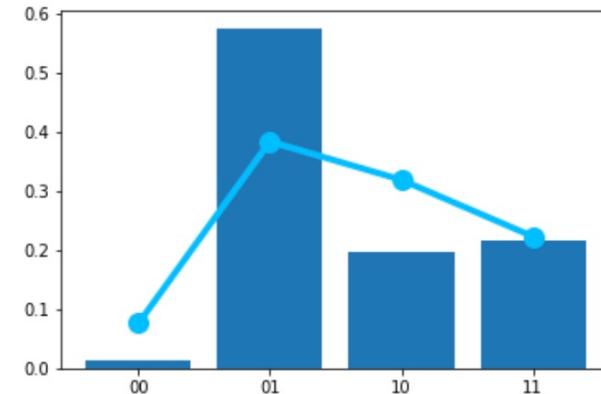
plt.figure(figsize=(6,5))
plt.title('CDF (Cumulative Distribution Function)')
samples_g, prob_g = qgan.generator.get_output(qgan.quantum_instance, shots=10000)
samples_g = np.array(samples_g)
samples_g = samples_g.flatten()
num_bins = len(prob_g)
plt.bar(samples_g, np.cumsum(prob_g), color='royalblue', width= 0.8, label='simulation')
plt.plot( np.cumsum(log_normal),'-o', label='log-normal', color='deepskyblue', linewidth=4,
markersize=12)
plt.xticks(np.arange(min(samples_g), max(samples_g)+1, 1.0))
plt.grid()
plt.xlabel('x')
plt.ylabel('p(x)')
plt.legend(loc='best')
plt.show()
```



直接確率分布を表示してみると再現度合いがイマイチ。。

```
: plt.plot( log_normal,'-o', label='log-normal', color='deepskyblue', linewidth=4, markersize=12)
plt.bar(["00", "01", "10", "11"], prob_g)
```

: <BarContainer object of 4 artists>



考えられる原因:

- エポック数が足りない(100くらい必要)
- NumpyDiscriminatorクラスの挙動が不安定
 - 現在github issueに挙げられている
 - PyTorchDiscriminatorで代用可能

まとめ

- ・GANはGenerator部分とDiscriminator部分のニューラルネットで構成され、両者を競わせながら学習している。
- ・量子GANはGenerator部分を量子ニューラルネットに置き換えたものである。
- ・量子Generatorのパラメータ θ を最適化することで所望の確率分布を再現する測定結果が得られる。
- ・パラメータ θ を最適化する過程で古典Discriminatorの学習(パラメータ ϕ の最適化)が必要

より詳しい説明はQiitaに書きました:

- ・量子GANを使って任意の確率分布を少数ゲートで再現する[理論編]
https://qiita.com/ucc_white/items/dca1248067fe0804be36
- ・量子GANを使って任意の確率分布を少数ゲートで再現する[実装編]
https://qiita.com/ucc_white/private/1fccf6bff9f1f3dc16fa