PYTHON (QISKIT) CODES FOR QUANTUM MACHINE LEARNING

Kumar Ghosh

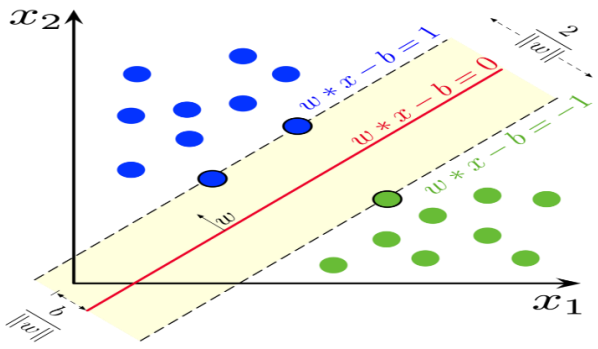
Quantum machine learning

Quantum computation is a branch of physics and computer science which solves different computational problems by using quantum-mechanical phenomena, for e.g. entanglement and superposition. Quantum computation and quantum information theory is a centre of attention in last few decades because it can outperform classical computation and information processing, because the quantum algorithms can give rise to exponential speedups over their classical counterparts. Many known quantum algorithms have a diverse application, such as: integer factorization, search algorithm solving constraint satisfaction problems, and quantum machine learning. Quantum machine learning is an emerging interdisciplinary research area at the intersection of quantum physics and machine learning, where machine learning algorithms for the analysis of classical data is executed on a quantum computer, for obtaining quantum generalizations of classical machine learning algorithms, which will provide possible speed-ups and other improvements over the existing classical learning models. In the following I am describing the quantum version of a partiular kind of mahine learning algorithm, namely: quanutm support vector machine.

Support vector machine (SVM):

Support vector machines (SVM) are a class of supervised machine learning algorithms for binary classifications. It can be used for both classification and regression challenges. However, it is mostly used in classification problems for e.g. image classification.

Consider a set of M data points $\{(\vec{x_j}, y_j)$: j=1, 2 … M$\}$, where each data point $\vec{x_j}$ is an N dimensional vector and y_j is the label of the data, which is $+1$ or -1. SVM finds the hyper plane $\vec{\omega} \cdot \vec{x} + b = 0$, which separates the whole data sets into two categories. In the following there is a schematic diagram of SVM where two kind of data are described by the blue and red dots respectively.



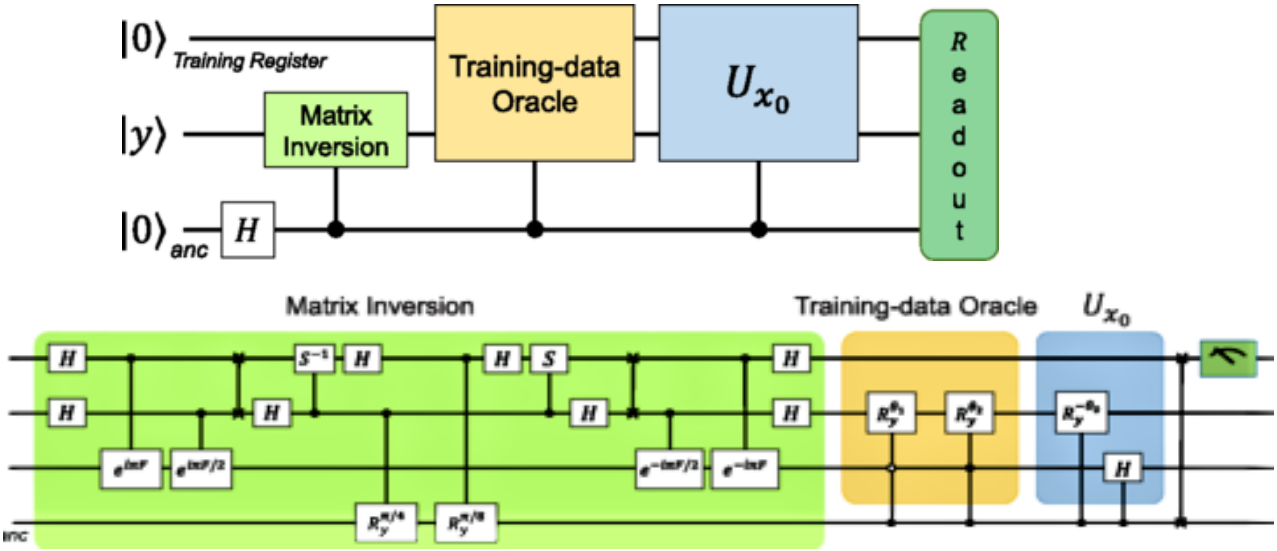
A version of SVM is called a Least squares SVM (LS-SVM) which approximates the hyper plane finding procedure of SVM by solving the following linear equation:

$$F\begin{bmatrix} b \\ \vec{\alpha} \end{bmatrix} \equiv \begin{bmatrix} 0 & \vec{1}^T \\ \vec{1} & K + \gamma.1 \end{bmatrix} \begin{bmatrix} b \\ \vec{\alpha} \end{bmatrix} = \begin{bmatrix} 0 \\ \vec{y} \end{bmatrix},$$

where $\boldsymbol{K_{ij}}$= $\overrightarrow{\mathrm{x_i}}^T \cdot \overrightarrow{x_j}$ is the symmetric kernel matrix, $\vec{\boldsymbol{y}} = (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{M})^T$, $\vec{\mathbf{1}} = (1, \ldots, 1)^T$, $\boldsymbol{\gamma}$ is the tuning parameter, and $(\boldsymbol{b}, \vec{\boldsymbol{\alpha}})$ are the parameters to determine the equation of the support vector plane.

Quantum Support vector machines:

Quantum version of SVM performs the LS-SVM algorithm using quantum computers. It calculates the linear kernel-matrix using the quantum algorithm for inner product on quantum random access memory (QRAM), solves the linear equation using a quantum algorithm for solving linear equations, and perform the classification of a query data using the trained qubits with a quantum algorithm. Below we give a schematic diagram to implement QSVM and a matrix inversion circuit respectively.



In the above diagram the matrix inversion is employed to obtain the hyperplane parameters $(\boldsymbol{b}, \vec{\boldsymbol{\alpha}})$. The training data oracle can be implemented by preparing a desired quantum state through controlled rotation described before.

The overall complexity of the quantum SVM is $\mathbf{O}\ (\log\ (\mathbf{NM}))$, whereas classical complexity of the LS-SVM is $\mathbf{O}\ (\boldsymbol{M}^2\ (\mathbf{M+N}))$. So we get exponential speed up in this procedure.

Although theoretically very efficient, the downside of the quantum SVM algoritm is involving Hamiltonian simulation, which itself is very complex. We can see the above quantum circuit involves lots of quantum gates just to calculate inverse of a simple $(2\mathrm{x}2)$ matrix. One of the open problem is to successfully construct a quanum Support Vector Machine for non-linear kernel to clasify more complex data.

The main computational part of this algorithm involves the matrix inversion. In the following we present the quantum circuit for computing $|X\rangle = A^{-1}\,|B\rangle$, with A= $\begin{bmatrix} \frac{5}{4} & -\frac{\sqrt{3}}{4} \\ -\frac{\sqrt{3}}{4} & \frac{7}{4} \end{bmatrix}$ and $|B\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. In the end of calculation, we obtain the Solution for the normalized vector $|x\rangle = \frac{|X\rangle}{||X\rangle|}$.

After measurement the theoretical result for obatining the states $|0\rangle$ and $|1\rangle$ are 0.62 and 0.38. respectively. Whereas, we simulate the quantum circuit through Qiskit from IBM-Q (qasm simulator) and obtained the results 0.58 and 0.42 respectively.

Qiskit python code and quantum circuit:

In the following I am attaching the Qiskit python code and quantum circuit

```
# Useful additional packages
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from math import pi
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, exe
cute
from qiskit.tools.visualization import circuit_drawer
from qiskit.quantum_info import state_fidelity
from qiskit import BasicAer

backend = BasicAer.get_backend('unitary_simulator')

q= QuantumRegister(3, 'q')
q_a = QuantumRegister(1, name='qa')
c= ClassicalRegister(1, 'c')
circ= QuantumCircuit(q, q_a, c)
#circt= QuantumCircuit(q, c)
circ.h(q[0])
circ.h(q[1])
circ.h(q[2])
circ.cu3(-2*(pi/3), 0, 0, q[1], q[2])
circ.cu3(2*(pi), -(pi/2), 0, q[1], q[2])
circ.cu3(2*(pi/3), 0, 0, q[1], q[2])
circ.cu3(4*(pi/3), pi, 0, q[0], q[2])

circ.swap(q[0], q[1])
circ.h(q[1])
circ.cu1(-(pi/2), q[0], q[1])
circ.h(q[0])

circ.swap(q[0], q[1])
circ.cu3(pi/2,0,0,q[1],q_a[0])
circ.cu3(pi/4,0,0,q[0],q_a[0])
circ.swap(q[0], q[1])

circ.h(q[0])
circ.cu1(pi/2, q[1], q[0])
circ.h(q[1])
circ.swap(q[0], q[1])
```

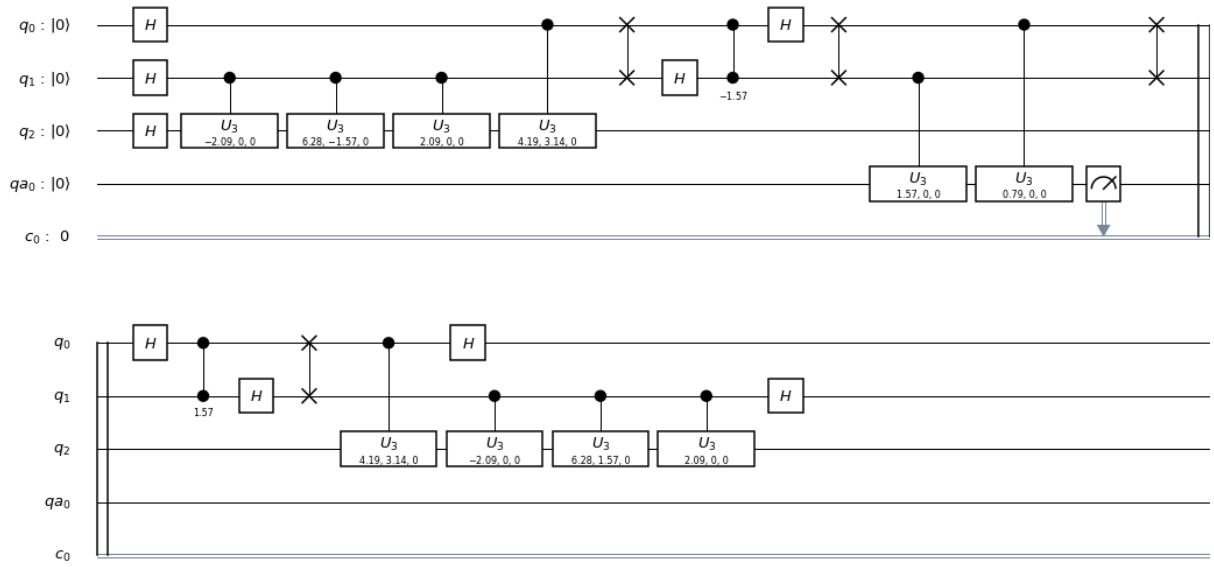
```
circ.cu3(4*(pi/3), pi, 0, q[0], q[2])
circ.cu3(-2*(pi/3), 0, 0, q[1], q[2])
circ.cu3(2*(pi), (pi/2), 0, q[1], q[2])
circ.cu3(2*(pi/3), 0, 0, q[1], q[2])

circ.h(q[0])
circ.h(q[1])

circuit_measure = circ.measure(q_a[0], c[0])

circ.draw(output='mpl')

############################################################
```




```
#circuit_measure1 = circt.measure(q[0],  c[1])
#circuit_measure2 = circt.measure(q[1],  c[2])

backend_sim = BasicAer.get_backend('qasm_simulator')

# Execute the circuit on the qasm simulator.
# We've set the number of repeats of the circuit
# to be 1024, which is the default.
job_sim = execute(circ, backend_sim, shots=10000)

# Grab the results from the job.
result_sim = job_sim.result()

counts = result_sim.get_counts(circ)
print(counts)
#circ.draw(output='mpl')
```

```
from qiskit.tools.visualization import plot_histogram
plot_histogram(counts)
```

```
{'0': 4228, '1': 5772}
```