paper1

March 8, 2021

[1]:
```
%matplotlib inline
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, execute, Aer, IBMQ, QuantumRegister,␣
 ↪ClassicalRegister
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *

# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

[2]:
```
import qiskit.extensions.quantum_initializer as qi
from qiskit.circuit.controlledgate import ControlledGate
import qiskit.circuit as qcirc
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
import qiskit.aqua.utils.subsystem as ss
import scipy.stats as stats
import itertools
import random
```

[3]:
```
def encode(classical_X):
    alpha = np.sqrt(np.einsum('ij,ij->i', classical_X, classical_X))
    alpha[alpha==0] = 1
    norm_alpha = classical_X / alpha[:, np.newaxis]
    return norm_alpha
```

[4]:
```
def initializeState(reg_to_init, init_state, name):
    initCirc = QuantumCircuit(reg_to_init, name=name)
    init = qi.Isometry(init_state, 0,0)
    initCirc.append(init, reg_to_init)

    basisGates = ['u1','u2','u3','cx']
    transpiled = transpile(initCirc, basis_gates=basisGates)
    initGate = transpiled.to_gate()
```

```
    return initGate
```

[5]:
```
def controlled_initialize(reg_to_init, init_state, ncb, name):
    init_gate = initializeState(reg_to_init, init_state, name)
    controlled_init = init_gate.control(num_ctrl_qubits=ncb)
    return controlled_init
```

[6]:
```
def whereToApply(bin_length):
    powerSeries = 2 ** np.arange(bin_length)
    indices = [
                [ind for ind, val in enumerate(powerSeries)
                     if (val & (pos ^ (pos-1))) == val
                ] for pos in range(2**bin_length)
              ]
    return indices
```

[7]:
```
def createOracle(train_data):
    train_shape = np.shape(train_data)
    m,n = np.log2(train_shape)
    if not m.is_integer():
        m = np.ceil(m)

    r_train = QuantumRegister(n, name='train_state')
    r_comp_basis = QuantumRegister(m, name='comp_basis')

    controlled_inits = [ControlledGate] * train_shape[0]
    oracleCirc = QuantumCircuit(r_train, r_comp_basis, name='oracle')

    for i, train_state in enumerate(train_data):
        controlled_inits[i] = controlled_initialize(r_train,␣
↪train_state,ncb=r_comp_basis.size, name="phi_{}".format(i))

    where_x = whereToApply(r_comp_basis.size)

    for i, (c_init, x_idx) in enumerate(zip(controlled_inits, where_x)):
        oracleCirc.x(r_comp_basis[x_idx])
        oracleCirc.append(c_init, r_comp_basis[:] + r_train[:])

    return oracleCirc
```

[8]:
```
def initialize_qknn(log2dim, log2NSamps, testState):

    r_0 = QuantumRegister(1, name="control")
    r_1 = QuantumRegister(log2dim, name="state_to_classify")
    r_2 = QuantumRegister(log2dim, name="train_states")
    r_3 = QuantumRegister(log2NSamps, name="comp_basis")
```

```
    c_0 = ClassicalRegister(r_0.size, name="control_measure")
    c_1 = ClassicalRegister(r_3.size, name="comp_basis_measure")

    initCirc = QuantumCircuit(r_0,r_1,r_2,r_3,c_0,c_1)
    #init = qi.Isometry(testState, 0, 0)
    #init.name = "init test state"
    #initCirc.append(init, r_1)
    initCirc.barrier()
    return initCirc
```

[9]:
```
def stateTransformation(qknnCirc, oracle):
    [control, test, train, comp_basis] = qknnCirc.qregs
    qknnCirc.h(control)
    qknnCirc.h(comp_basis)
    qknnCirc.append(oracle, train[:]+comp_basis[:])

    for psi, phi in zip(test, train):
        qknnCirc.cswap(control, psi, phi)

    qknnCirc.h(control)
    qknnCirc.barrier()

    return qknnCirc
```

[10]:
```
def addMeasurement(qknnCirc):
    comp_basis_c = qknnCirc.cregs[-1]
    comp_basis_q = qknnCirc.qregs[-1]
    qknnCirc.measure(qknnCirc.qregs[0], qknnCirc.cregs[0])

    for qbit,cbit in zip(comp_basis_q, reversed(comp_basis_c)):
        qknnCirc.measure(qbit,cbit)

    return qknnCirc
```

[11]:
```
def construct_circuit(testState, trainState):
    state_dim = len(testState)
    oracle = createOracle(trainState)
    n = np.log2(state_dim)
    m = oracle.num_qubits - n

    qknnCirc = initialize_qknn(n,m,testState)
    qknnCirc = stateTransformation(qknnCirc, oracle)
    qknnCirc = addMeasurement(qknnCirc)

    qknnCirc.draw(output='mpl')
    return qknnCirc
```

[12]:
```
def setupControlCounts(controlCounts):
    controlStates = np.array(['0','1'])
    if controlStates[0] not in controlCounts:
        toSub = int(controlStates[0])
    elif controlStates[1] not in controlCounts:
        toSub = int(controlStates[1])
    else:
        toSub = None

    if toSub is not None:
        sole = -1 * (toSub - 1)
        controlCounts = {str(toSub): 0, str(sole): controlCounts[str(sole)]}

    return controlCounts
```

[13]:
```
def calculate_fidelity(count):
    subsystemCounts = ss.get_subsystems_counts(counts)
    controlCounts = setupControlCounts(subsystemCounts[1])
    totalCounts = controlCounts['0'] + controlCounts['1']
    exp_fidelity = np.abs(controlCounts['0'] - controlCounts['1']) / totalCounts
    nQ = len(list(subsystemCounts))
    compBasis = list(itertools.product(['0','1'], repeat=nQ))
    fidelities = np.zeros(2**nQ, dtype=float)
    for compState in compBasis:
        compState = ''.join(compState)
        fidelity = 0
        for controlState in controlCounts.keys():
            stateStr = compState + ' ' + controlState
            fidelity += (-1) ** int(controlState) * int(counts[stateStr]) /␣
 ↪(controlCounts[controlState]) * (1-exp_fidelity ** 2)
        indexState = int(compState,2)
        fidelity *= 2 ** nQ/2
        fidelity += exp_fidelity
    return fidelity
```

[14]:
```
iris = datasets.load_iris()

X = iris.data
Y = iris.target
k = 3

data = list(zip(X,Y))
random.shuffle(data)
X, Y = zip(*data)
X = np.array(X)

n_variables = 2
```

```
n_train = 4
n_test = 2

encode_data = encode(X[:,:n_variables])
train_data = encode_data[:n_train]
train_labels = Y[:n_train]

test_data = encode_data[n_train:(n_train+n_test),:n_variables]
test_labels = Y[n_train:(n_train+n_test)]


circ = construct_circuit(test_data, train_data)
print(circ)
circ.draw()
backend = Aer.get_backend('qasm_simulator')

result = execute(circ, backend).result()
```

```
          control_0:    H           H    M

 state_to_classify_0:           X

      train_states_0:     0         X

        comp_basis_0:    H  1 oracle          M

        comp_basis_1:    H  2                   M

    control_measure: 1/
                                                         0

comp_basis_measure: 2/
                                                            1  0
```

[15]:
```
counts = result.get_counts()
```

[16]:
```
nQ = len(list(counts)) - 2

nOcc = len(counts)
nData = 2 ** nQ

allFid = np.empty(shape = (nOcc, nData))

for i, count in enumerate(counts):
    allFid[i] = calculate_fidelity(count)
```

[17]:
```
sorted_neighbors = np.argpartition(1-allFid, k)
n_queries = len(train_labels)
sorted_neighbors = sorted_neighbors[sorted_neighbors < n_queries].
 ↪reshape(sorted_neighbors.shape[0], n_queries)

if n_queries == 1:
    kNN = sorted_neighbors[:k]
else:
    kNN = sorted_neighbors[:,:k]

voter_labels = np.take(train_labels, kNN)
if n_queries == 1:
    votes, c = stats.mode(voter_labels)
else:
    votes, c = stats.mode(voter_labels, axis=0)
```

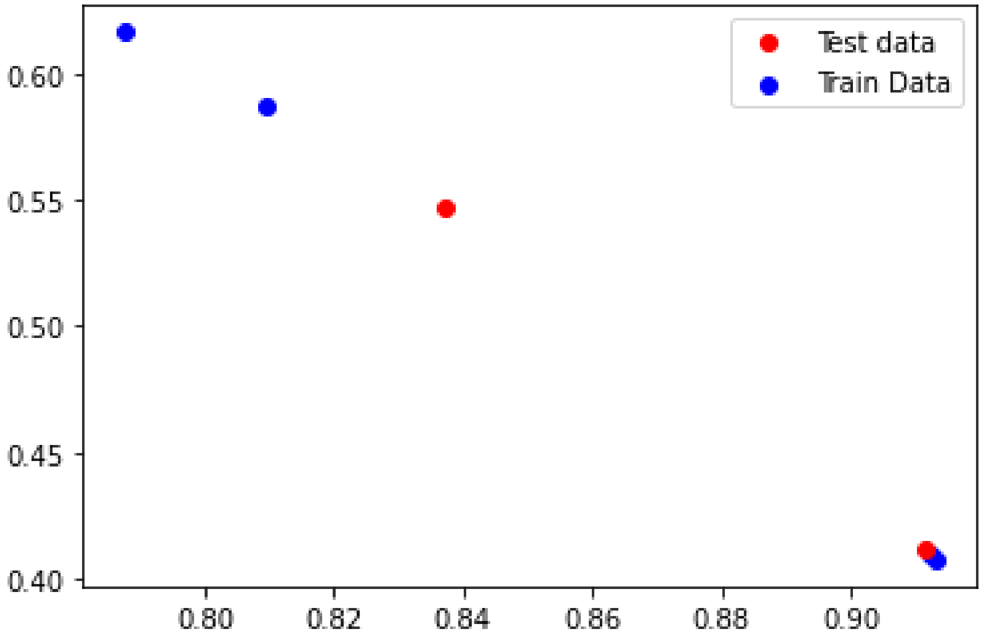
[18]:
```
# Analysis
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(train_data)

plt.scatter(train_data[:, 0], train_data[:, 1],  color='blue')

pcaTest = PCA(n_components=2)
pcaTest.fit(test_data)
plt.plot(test_data[:,0], test_data[:,1], 'ro')
legend=['Test data', 'Train Data']
plt.legend(legend)

print("Training data:")
for i in range(n_train):
    print(train_data[i], " -> ", train_labels[i])
print("Test data: \t\t Actual Label")
print(test_data[0], " -> ", test_labels[0])
print(test_data[1], " -> ", test_labels[1])
```

```
Training data:
[0.787505   0.61630826]  ->  0
[0.9121687  0.40981492]  ->  1
[0.91313788 0.40765084]  ->  1
[0.80942185 0.58722762]  ->  0
Test data:               Actual Label
[0.83696961 0.54724936]  ->  0
[0.9113706  0.41158672]  ->  2
```



[19]:
```
y_pred = votes.real.flatten()
print(f"{k} nearest label(s) is/are: {y_pred}")
```

```
3 nearest label(s) is/are: [0 1 0]
```