Quantum-KNN

December 5, 2021

[1]:
```
from qiskit import *
import matplotlib.pyplot as plt
from qiskit import tools
from qiskit.tools.visualization import plot_histogram, plot_state_city
from qiskit.circuit.library import MCMT, MCXGate, Measure
from qiskit.extensions import UnitaryGate
import numpy as np
```

[2]:
```
import pprint
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
import math
from sklearn.model_selection import train_test_split
```

[3]:
```
def initializeKNN(Object, pattern, n, m, class_bit, k_neighbours, threshold,␣
↪shots):

        Object.pattern = pattern

        Object.m = m
        Object.n = n
        Object.class_n = class_bit
        Object.k_neighbours = k_neighbours
        Object.t = threshold

        Object.shots = shots

        Object.n_total = n+class_bit

        Object.main_pR = QuantumRegister(Object.n_total, "p")
        Object.main_uR = QuantumRegister(2,"u")
        Object.main_mR = QuantumRegister(Object.n_total, "m")

        Object.main_circuit = QuantumCircuit(Object.main_pR, Object.main_uR,␣
↪Object.main_mR)
```

```
        Object.one_state = [0,1]
        Object.zero_state = [1,0]
```

[4]:
```
def createAndTrainSuperPosition(Object):
        """
        This function creates a superposition of dataset as described in the␣
↪paper.
        """
        for i in range(Object.m):
            pR = QuantumRegister(Object.n_total, "p")
            uR = QuantumRegister(2,"u")
            mR = QuantumRegister(Object.n_total, "m")
            circuit = QuantumCircuit(pR,uR,mR, name="pattern"+str(i+1))
            circuit.draw()

            for j in range(Object.n_total):

                if Object.pattern[i][j] == 0:
                    circuit.initialize(Object.zero_state,pR[j])
                else:
                    circuit.initialize(Object.one_state,pR[j])

                circuit.ccx(pR[j],uR[1],mR[j])

            for j in range(Object.n_total):

                circuit.cx(pR[j],mR[j])
                circuit.x(mR[j])

            circuit.mcx(mR,uR[0])

            k = i+1
            data = np.array([[np.sqrt((k-1)/k),np.sqrt(1/k)],[-np.sqrt(1/k),np.
↪sqrt((k-1)/k)]])
            gate = UnitaryGate(data=data)
            gate = gate.control(1,ctrl_state="1")
            circuit.append(gate,[uR[0],uR[1]],[])

            circuit.mcx(mR,uR[0])

            for j in range(Object.n_total):

                circuit.x(mR[j])
                circuit.cx(pR[j],mR[j])

            for j in range(Object.n_total):
                circuit.ccx(pR[j],uR[1],mR[j])
```

```
            circuit.draw()

            Object.main_circuit.append(circuit.to_instruction(), Object.
↪main_pR[:Object.n_total]+ Object.main_uR[:2] + Object.main_mR[:Object.
↪n_total])

        return Object.main_circuit
```

[5]:
```
def fitVectorAndSuperPosition(Object, x):
        """
        A function to fit the test vector x with the superpositioned dataset.
        The circuit from previous set is appended to this circuit as there is␣
↪no concept of saving the data!
        """
        l = 2**(Object.k_neighbours)-Object.n
        a = Object.t+l
        a_binary = "{0:b}".format(a)
        a_len = Object.k_neighbours+1

        if len(a_binary) < a_len:
            a_binary = "0"*(a_len-len(a_binary))+a_binary

        xR = QuantumRegister(Object.n, "x")
        auR = QuantumRegister(1, "au")
        aR = QuantumRegister(a_len,"a")
        cR = ClassicalRegister(1, "c")
        oR = ClassicalRegister(Object.class_n, "o")

        predictCircuit = QuantumCircuit(xR, Object.main_mR, aR, auR, cR, oR)
        predictCircuit.draw()
        circuit = Object.main_circuit + predictCircuit

        circuit.barrier()

        for k in range(len(x)):

            circuit.cx(Object.main_mR[k],xR[k])
            circuit.x(xR[k])

        for i in range(a_len):

            if a_binary[::-1][i] == "0":
                circuit.initialize(Object.zero_state,aR[i])
            else:
                circuit.initialize(Object.one_state,aR[i])
```

```
        circuit.initialize(Object.one_state,auR)
        for k in range(len(x)):

            for i in range(a_len):

                circuit.ccx(xR[k],auR, aR[i])
                ctrlString = "1"+"0"*(i)+"1"
                tempmc = MCXGate(i+2,ctrl_state=ctrlString)
                circuit.append(tempmc,[xR[k]]+aR[:i+1]+[auR],[])

            circuit.x(auR)
            ctrlString ="0"*(a_len-1)+"1"
            tempmc = MCXGate(a_len,ctrl_state=ctrlString)
            circuit.append(tempmc,[xR[k]]+aR[0:a_len-1]+[auR],[])

        circuit.barrier()

        circuit.measure(auR, cR)
        for i in range(Object.class_n):
            circuit.measure(Object.main_mR[Object.n+i],oR[i])




        simulator = Aer.get_backend("qasm_simulator")
        results = execute(circuit,simulator, shots=Object.shots).result()

        result_dict = results.get_counts(circuit)

        return result_dict
```

[6]:
```
def setup():
    print("Setting Up Model Params")
    trainObj = Train
    testObj = Test
    data_size = 8 # higer number causes the creation of more Qubits.
    test_data_points = 2

    exponent = int(math.log(data_size, 2))

    data = np.array(np.arange(data_size), dtype= np.uint8)
    label = np.zeros(data_size)
    label[1::2] = 1
    data= np.flip((((data[:,None] & (1 << np.arange(exponent)))) > 0).
 ↪astype(int), axis=1)
```

```
    trainObj.Data,testObj.Data,trainObj.Label,testObj.Label =␣
↪train_test_split(data,label,test_size=test_data_points)

    print("training data points: {}".format(len(trainObj.Label)))
    print("testing data points: {}".format(len(testObj.Label)))

    return trainObj, testObj
```

[7]:
```
def ClassicKNNModel(k, trainObj, testObj):
    print("init : Classic KNN Model")

    model = KNeighborsClassifier(n_neighbors=k,algorithm="brute")
    model.fit(trainObj.Data,trainObj.Label)

    # evaluate the model and update the accuracies list
    CPredict = model.predict(testObj.Data)
    CScore = accuracy_score(testObj.Label,CPredict,normalize=True)

    return CPredict, CScore
```

[8]:
```
def QuantumKNNModel(k, trainObj, testObj):
    print("init : Quantum KNN Model")

    class_bit = 1
    t = 1 # random guess
    pattern_np = np.concatenate((trainObj.Data,trainObj.Label.reshape(trainObj.
↪Label.size,1)), axis=1)



    # Lesser shots often lead to class undetermined state.
    n = pattern_np.shape[1]-class_bit
    m=pattern_np.shape[0]

    QKNN_obj = QuantumKNN(pattern_np, n=pattern_np.
↪shape[1]-class_bit,m=pattern_np.shape[0],
                        class_bit=class_bit, k_neighbours=k, threshold =t,␣
↪shots=10000)
    QKNN_obj.createAndTrainSuperPosition()
    QPredict = []

    for x in testObj.Data:
        predict = QKNN_obj.fitVectorAndSuperPosition(x)

        ## Can be simplified Did it in a hasty way!!!
        key_List = np.array(list(predict.keys()))
        required_key = key_List[np.where(key_List.astype('<U1')=="1")[0]]
        print(required_key, required_key.size)
```

```python
        if not required_key.size > 0:
            assert False,"class not determined"
        else:
            val = []
            for key in required_key:
                val.append(predict[key])
            max_i = np.argmax(val)
            QPredict.append(int(required_key[max_i][2]))

    QScore = accuracy_score(testObj.Label,QPredict,normalize=True)

    return QPredict, QScore
```

[9]:
```python
class QuantumKNN:
    __init__                     = initializeKNN
    createAndTrainSuperPosition = createAndTrainSuperPosition
    fitVectorAndSuperPosition   = fitVectorAndSuperPosition

class Train:
    Data = 0
    Label = 0

class Test:
    Data = 0
    Label = 0
```

[10]:
```python
def main():
    k = 4
    trainObj, testObj = setup()
    CPredict, CScore = ClassicKNNModel(k, trainObj, testObj)
    QPredict, QScore = QuantumKNNModel(k, trainObj, testObj)

    print("for  KNN k=%d, accuracy=%.2f%%" % (k, CScore * 100))
    print("for QKNN k=%d, accuracy=%.2f%%" % (k, QScore * 100))
    print(testObj.Label)
    print(CPredict)
    print(QPredict)
```

[11]:
```python
main()
```

```
Setting Up Model Params
training data points: 6
testing data points: 2
init : Classic KNN Model
init : Quantum KNN Model
```

```
/tmp/ipykernel_1074/1633972580.py:22: DeprecationWarning: The
QuantumCircuit.__add__() method is being deprecated.Use the compose() method
```

```
which is more flexible w.r.t circuit register compatibility.
  circuit = Object.main_circuit + predictCircuit
/opt/conda/lib/python3.8/site-packages/qiskit/circuit/quantumcircuit.py:933:
DeprecationWarning: The QuantumCircuit.combine() method is being deprecated. Use
the compose() method which is more flexible w.r.t circuit register
compatibility.
  return self.combine(rhs)
```

```
['1 1' '1 0'] 2
['1 1' '1 0'] 2
for  KNN k=4, accuracy=0.00%
for QKNN k=4, accuracy=100.00%
[1. 1.]
[0. 0.]
[1, 1]
```