PYTHON CODES FOR CLASSICAL AND QUANTUM MACHINE LEARNING

I am attaching the codes from two selective projects in the following: one from classical machine learning and one from quantum machine learning

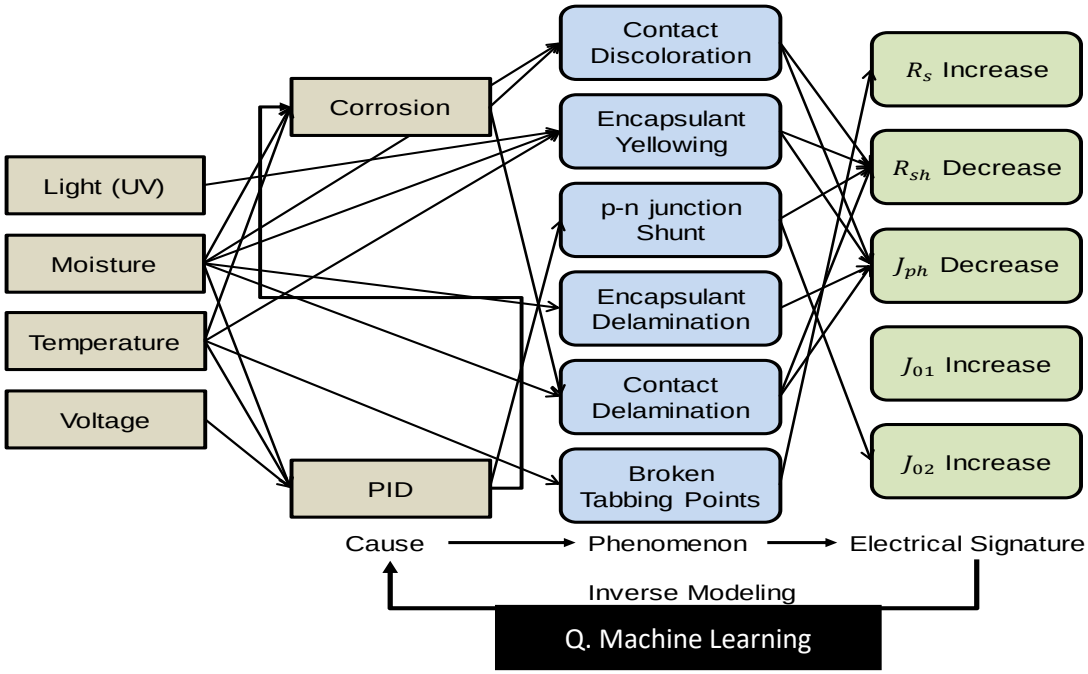
Kumar Ghosh

Classical machine learning for Solar farm project:

In a rapidly-growing solar energy industry, there has been a constant effort to reduce the levelized cost of energy (LCOE) for which the reliability of the PV system has attracted major attentions. The lifetimes and performances of various systems have been measured through field, qualification and accelerated stress tests, yet there is no clear proof of identification and deconvolution of variegated degradation or fault mechanisms until recently. The degradation mechanisms involved are of numerous varieties and high complexity. Moreover, the mechanisms are intertwined, meaning that each mechanism affects more than one feature or parameter of the field data. This poses a major issue in analyzing the faults/degradation mechanisms, deriving analytical forms and physical models and further deconvolve the mechanisms. With the advent of machine learning techniques, we realize that our problem is a classic case of pattern recognition that can be solved using these techniques. Hence, we solve a simplified version of the complex problem of identification and deconvolution of degradation mechanisms in PV systems using machine learning to gauge the viability of our approach. Figure below describes the workflow of our methodology. A real-world problem involves field data and weather data from a wide variety of farms across the globe. This will help make important decisions regarding deployment of solar farms at a particular geographical location based on economic and environmental viability. However, this will require faster machine learning algorithms, since the data is large and there are several physical and environmental parameters affecting various degradation mechanisms simultaneously.

We are collaboating with a solar farm project, where we predict the efficiency degradation of a solar cell depending on few parameters for eg. contact corrosion, potential induced degradation (PID) etc. If the efficiency of a solar cell is less than a threshold (75%), then it is treated as a bad PV cell. This problem can be interprited as a classifier problem where the whole data (fianl output) can be divided into two classes namely: good and bad solar cell.

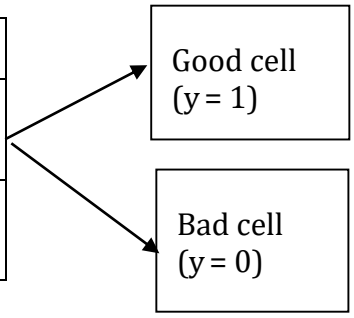
A rough outline of this project is desribed by the following diagram:



In the above diagram we show different causes' path to impact on performance and electrical signature, for e.g. corrosion, Potential Induced Degradation (PID) etc and how it effects the output variables for e.g. current and resistance.

A schematic table of the data structure (from the above diagram) is described in the following, where we have three main independent parameters (Contact Corrosion (x_1), Potential Induced Degradation (PID) (x_2), Encapsulant Yellowing (x_3)) and output dependent variable (y) is efficiency of the solar cell. Although efficiency is a cntinuous variable but we can classify these variables into two classes, namely good class, y= 1 (efficiency is above the threshold), and bad class, y= 0 (efficiency is below the threshold)

R_S Series resistance (x_1)	Contact Corrosion (x_1)
R_{sh} Shunt Resistance (x_2)	Potential Induced Degradation (PID) (x_2)
J_{ph} Photo-current (x_3)	Encapsulant Yellowing (x_3)



We first split our data into training (60%) and test (40%) sets. We use different supervised machine learning algorithm for e.g. Logistic regression, k-nearest neighbours, Random Forest, Support Vector Machine algorithms to classify the data. In the below we summarize the best results got from different machine learning algorithms.

ML Algorithm	Highest testing accuracy achieved
Random Forest	99 %
Support Vector Machine	95 %
k-nearest neighbors	98%
Logistic regression	95 %

Although the overall testing accuracy is comparable for all the algorithms above, but Random forest gives the best result for individual class accuracy.

In the next page I am attaching python codes for data analysis and machine learning for the solar farm data:

Data analysis part

Importing the packages

```
import numpy as np
from sklearn import datasets, svm
from __future__ import division, print_function
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import pandas as pd
import seaborn as sns
from time import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
```

```
df = pd.read_excel('test2.xlsx')
df. columns
```

```
Index(['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)', 'Eff_loss (25%)',
       'Class', 'Column number'],
      dtype='object')
```

```
df['Class'].value_counts()
```

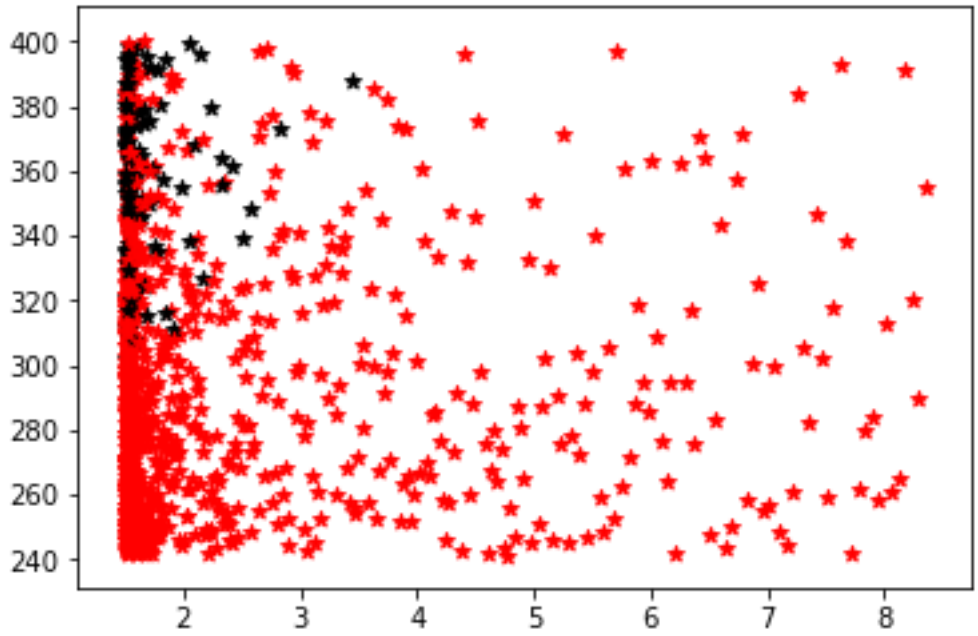
```
0    902
1     98
Name: Class, dtype: int64
```

```
x_data=df[['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)']]
y_data=df['Class']
```

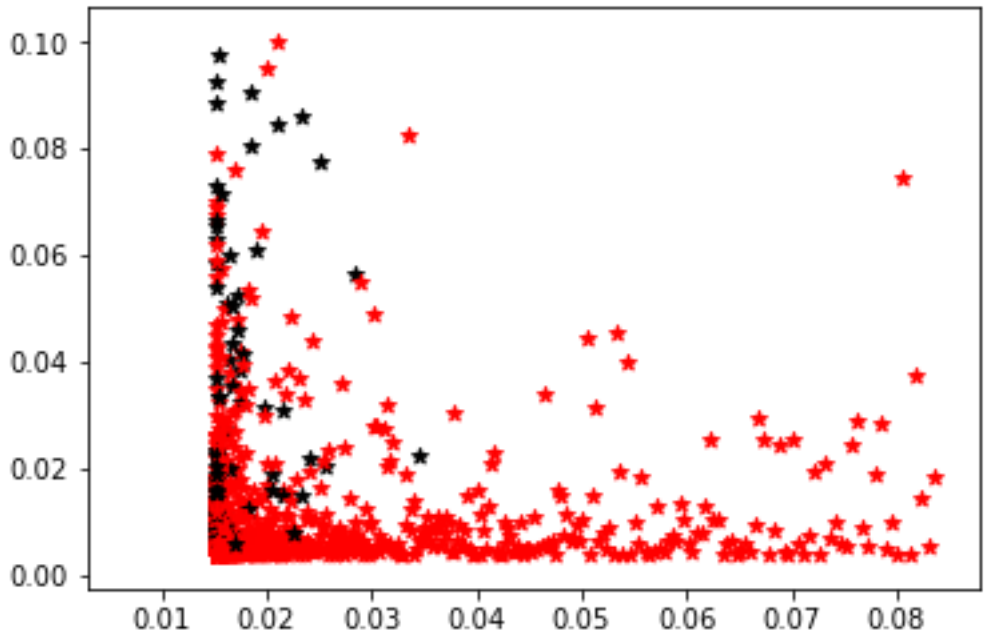
```
yx= x_data.iloc[0,0] #iloc[row,column]# or loc[r,c] also works
```

Plotting the different features of the data

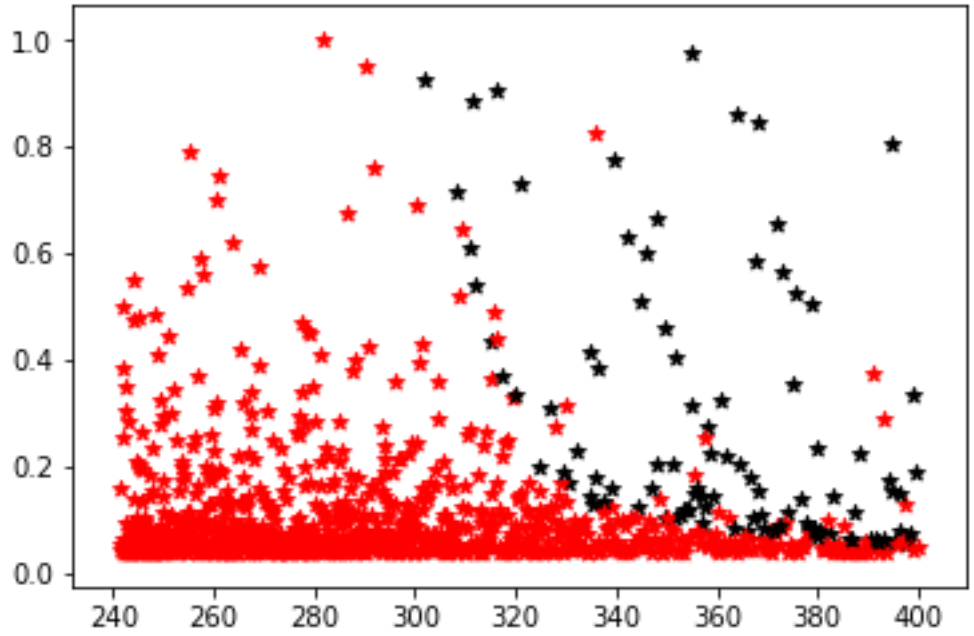
```
for i in range(0, len(y_data)):
    if y[i] ==1:
        #print(x_data.iloc[i,1])
        #print(x[i])
        plt.scatter(x_data.iloc[i,0]*10000, x_data.iloc[i,1], color='k', m
arker='*')
    else:
        plt.scatter(x_data.iloc[i,0]*10000, x_data.iloc[i,1], color='r', m
arker='*')
```




```
for i in range(0, len(y_data)):
    if y[i] ==1:
        #print(x_data.iloc[i,1])
        #print(x[i])
        plt.scatter(x_data.iloc[i,0]*100, x_data.iloc[i,2], color='k', mar
ker='*')
    else:
        plt.scatter(x_data.iloc[i,0]*100, x_data.iloc[i,2], color='r', mar
ker='*')
```



```
for i in range(0, len(y_data)):
    if y[i] ==1:
        #print(x_data.iloc[i,1])
        #print(x[i])
        plt.scatter(x_data.iloc[i,1], x_data.iloc[i,2]*10, color='k', mark
er='*')
    else:
        plt.scatter(x_data.iloc[i,1], x_data.iloc[i,2]*10, color='r', mark
er='*')
```



Machine learnin part

K-Nearest-Neighbors Classifier (with varying parameter k=1 to 20)

Importing the packages

```
import numpy as np
from sklearn import datasets, svm
from __future__ import division, print_function
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import pandas as pd
import seaborn as sns
```

```
from time import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report

df = pd.read_excel('test2.xlsx')
df. columns
```

```
Index(['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)', 'Eff_loss (20%)',
       'Class', 'Column number'],
      dtype='object')
```

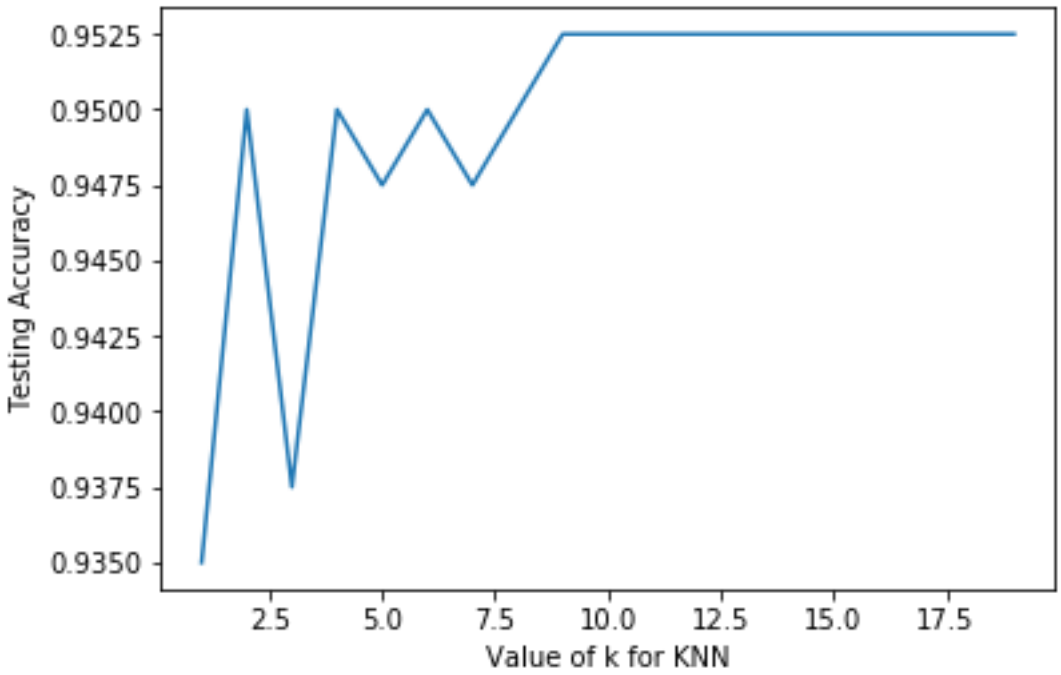
```
x_data=df[['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)']]
y_data=df['Class']


# split  x and y into training and testing sets


from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x_data,y_data,test_siz
e=0.4, random_state=4)

#KNN with N=3
from  sklearn.neighbors import KNeighborsClassifier
knn= KNeighborsClassifier(n_neighbors=3)
knn.fit(x_train,y_train)

y_pred = knn.predict(x_test)

print(metrics.accuracy_score(y_test,y_pred))
```

```
0.9375
```

```
# KNN with N= 1 to 20

k_range = list(range(1,20))
scores=[]
for i in k_range:
    knn= KNeighborsClassifier(n_neighbors=i)
    knn.fit(x_train,y_train)
    y_pred = knn.predict(x_test)
    print(metrics.accuracy_score(y_test,y_pred))
    scores.append(metrics.accuracy_score(y_test, y_pred))
    print(classification_report(y_test, y_pred,  digits=4))

print(scores)
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(k_range, scores)
plt.xlabel('Value of k for KNN')
plt.ylabel('Testing Accuracy')
```

```
[0.935, 0.95, 0.9375, 0.95, 0.9475, 0.95, 0.9475, 0.95, 0.9525, 0.9525,
```

0.9525, 0.9525, 0.9525, 0.9525, 0.9525, 0.9525, 0.9525, 0.9525, 0.9525]



Support Vector Machine Classifier (with linear and rbf kernel)

```
# support vector classifier linear kernel


from sklearn.svm import SVC # "Support Vector Classifier"
svmclf = SVC(kernel='linear')

# fitting x samples and y classes
svmclf.fit(x_train,y_train)
#svmclf.predict([[387.09, 0.011]])
#x3_new = [[0.00015, 387.09, 0.011], [0.00015, 312.95, 0.0117]]
svmclf.predict(x_test)
print(metrics.accuracy_score(y_test,y_pred))
```

```
0.9525
```

```
# support vector classifier rbf kernel


from sklearn.svm import SVC # "Support Vector Classifier"
svmclf = SVC(kernel='rbf', gamma=.7)

# fitting x samples and y classes
svmclf.fit(x_train,y_train)
#svmclf.predict([[387.09, 0.011]])
#x3_new = [[0.00015, 387.09, 0.011], [0.00015, 312.95, 0.0117]]
```

```
svmclf.predict(x_test)
print(metrics.accuracy_score(y_test,y_pred))
```

```
0.9525
```

Random Forest Classifier (varying different parameters):

```
df = pd.read_excel('test2.xlsx')
df. columns
```

```
Index(['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)', 'Eff_loss (20%)',
       'Class', 'Column number'],
      dtype='object')
```

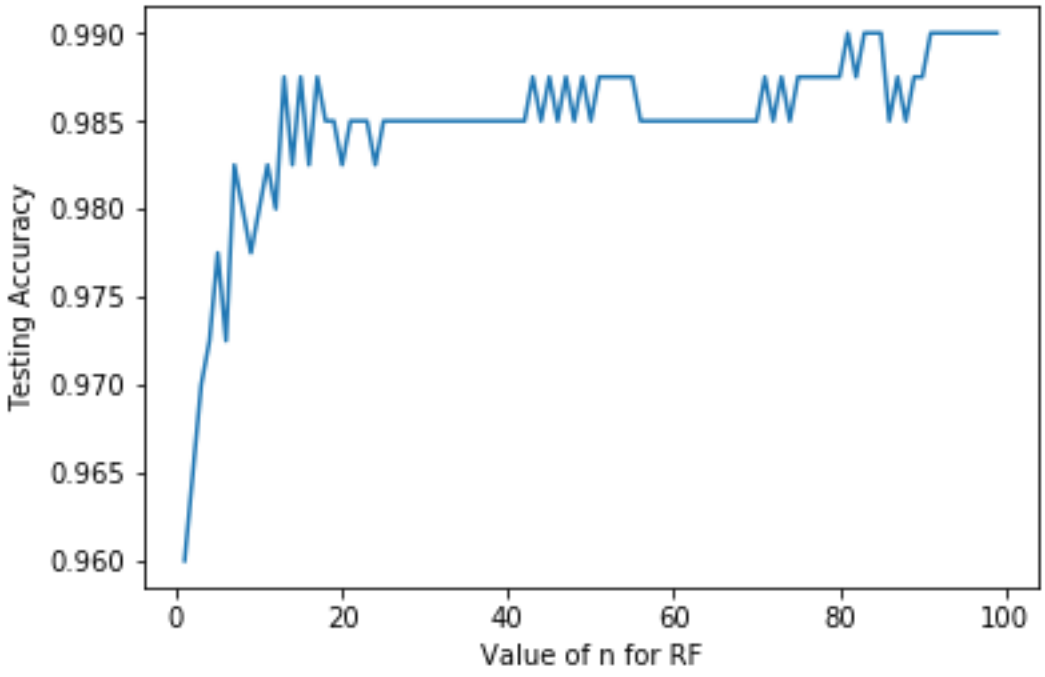
```
#df

x_data=df[['Rs (Corrosion)', 'Jph (Yellowing)', 'Rsh (PID)']]
y_data=df['Class']

from sklearn.ensemble import RandomForestClassifier
import pandas as pd

#creating a random forest classifier
clf = RandomForestClassifier(n_jobs=2, random_state=0)
#training classifier
clf.fit(x_data,y_data)


#step 1: split x and y into training and testing sets


from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x_data,y_data,test_siz
e=0.4, random_state=4)

#creating a random forest classifier
clf = RandomForestClassifier(n_jobs=2, random_state=0, n_estimators=10)


#training and testing the classifier


clf.fit(x_train,y_train)

y_pred = clf.predict(x_test)
print(metrics.accuracy_score(y_test,y_pred))
print(classification_report(y_test, y_pred,  digits=4))
```

```
0.98
              precision    recall f1-score   support

           0     0.9895    0.9895    0.9895       381
           1     0.7895    0.7895    0.7895        19

   micro avg     0.9800    0.9800    0.9800       400
```

```
   macro avg     0.8895    0.8895    0.8895       400
weighted avg     0.9800    0.9800    0.9800       400
```

```
n_range = list(range(1,100))
scores=[]
for i in n_range:
    clf = RandomForestClassifier(n_jobs=2, random_state=0, n_estimators= i
)
    clf.fit(x_train,y_train)
    y_pred = clf.predict(x_test)
    #print(metrics.accuracy_score(y_test,y_pred))
    scores.append(metrics.accuracy_score(y_test, y_pred))
    print(classification_report(y_test, y_pred,  digits=4))

print(scores)
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(n_range, scores)
plt.xlabel('Value of n for RF')
plt.ylabel('Testing Accuracy')
```

```
[0.96, 0.965, 0.97, 0.9725, 0.9775, 0.9725, 0.9825, 0.98, 0.9775, 0.98, 0.
9825, 0.98, 0.9875, 0.9825, 0.9875, 0.9825, 0.9875, 0.985, 0.985, 0.9825,
0.985, 0.985, 0.985, 0.9825, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.9
85, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985,
0.985, 0.9875, 0.985, 0.9875, 0.985, 0.9875, 0.985, 0.9875, 0.985, 0.9875,
0.9875, 0.9875, 0.9875, 0.9875, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985,
0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.985, 0.9875, 0.9
85, 0.9875, 0.985, 0.9875, 0.9875, 0.9875, 0.9875, 0.9875, 0.9875, 0.99, 0
.9875, 0.99, 0.99, 0.99, 0.985, 0.9875, 0.985, 0.9875, 0.9875, 0.99, 0.99,
0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99]
```



Logistic Regression Classifier

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression() #using default paameters
logreg.fit(x_train,y_train)
```

```
y_pred = logreg.predict(x_test)
print(metrics.accuracy_score(y_test,y_pred))
```

```
0.955
```

multilayer perceptron (MLP) Classifier

```
from sklearn.linear_model import MLPClassifier
```

```
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(5, 2), random_state=1)
```

```
clf.fit(x_train,y_train)
```

```
y_pred = clf.predict(x_test)
```

```
print(metrics.accuracy_score(y_test,y_pred))
```

```
0.9525
```

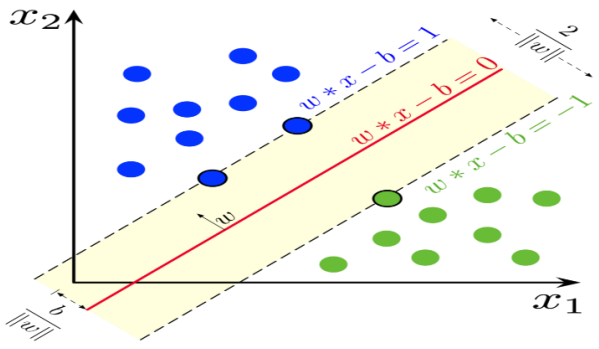
Quantum machine learning

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.



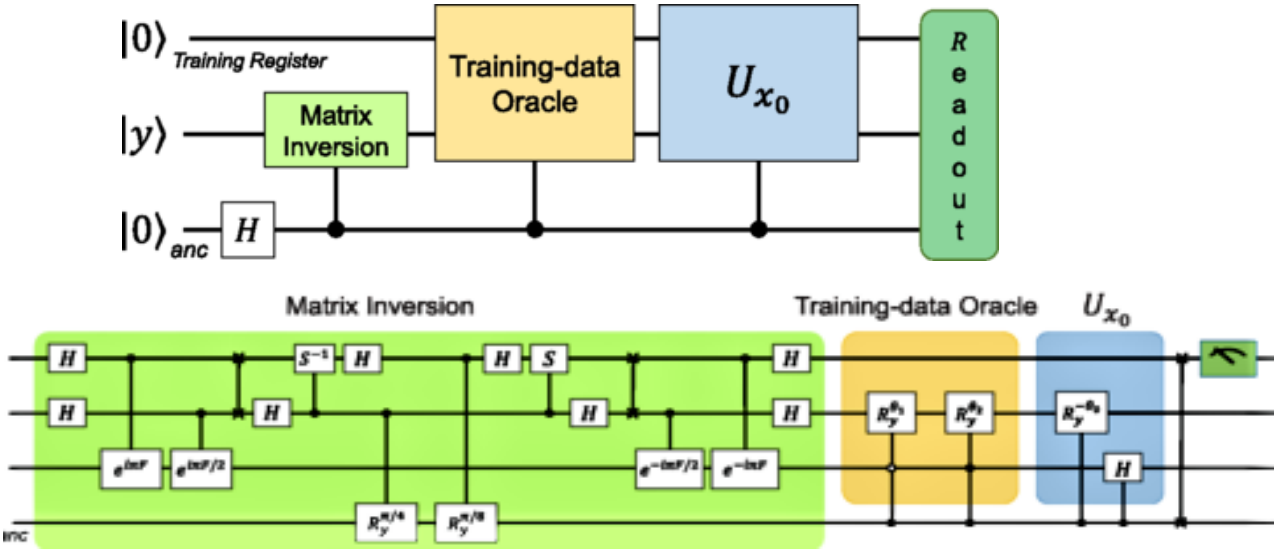
A version of SVM is called a Least squares SVM (LS-SVM) which approximates the hyper plane finding procedure of SVM by solving the following linear equation:

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

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 $|x\rangle = \frac{|X\rangle}{||X\rangle|}$.

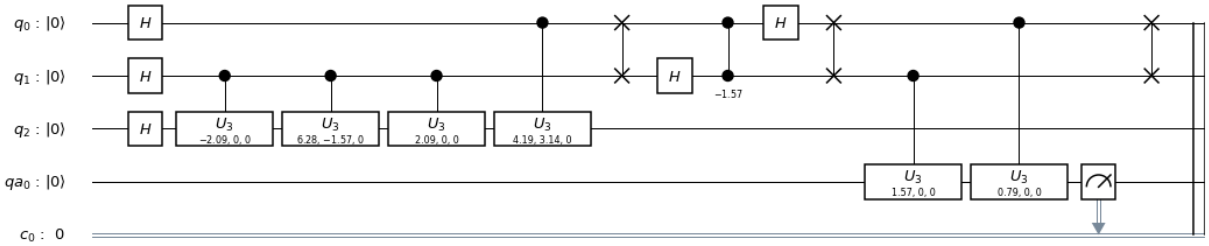
After measurement the theoretical result for obatining the states $|0\rangle$ and $|1\rangle$ are 0.62 and 0.38. respectively. Whereas, we simulate the quantum circuit through Qiskit from IBM-Q (qasm simulator) and obtained the results 0.58 and 0.42 respectively.

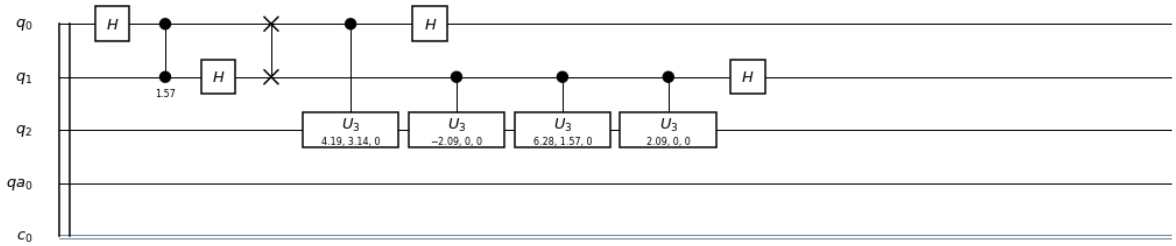
Qiskit python code and quantum circuit:

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}
```