Parameterised Quantum Circuits for High-dimensional Data

Qing Zhou, s2501597

16 June, 2021

1 Introduction

We are in an exciting era where quantum machine learning is on its advancement. The high dimensionality of Bloch sphere exibits quantum supremacy and might deliver solution to problems that are of quantum nature. The variational quantum classifier, takes advantages of both classical machine learning algorithms and quantum algorithms, is one of the hybrid quantum classical algorithms. It consists of quantum circuits which allow variations of parameters, which are otherwise fixed in traditional machine learning algorithms. Based on the quantum circuit, we can build a supervised learning model. Fig. 1 illustrates a supervised learning model using a parametrised quantum circuit (PQC).
PQCs often involve a series of gates, controlled gates such as controlled NOT gates and adjustable gates such as rotation gates. In HQC algorithms, the PQCs work as the interface between the quantum block and the classical block, where the former takes quantum data as inputs, and the latter tunes the parameters coming out of the quantum block. Based on a PQC we can build a supervised learning model.

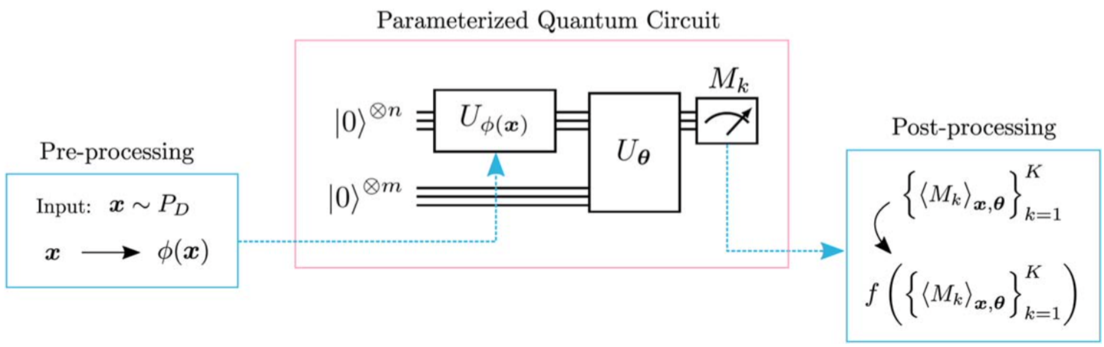

Figure 1: A supervised learning model implementing HQC algorithm. The model consists of three components: the classical pre/post processing blocks and the quantum block with PQC. Adapted from Fig. 2 by Benedetti et al. (2019).

In the pre-processing step, the classical data is embedded into quantum states. One way of doing this is by mapping the vector components of an input data into amplitudes of a quantum state. In addition, pre-processing methods such as standardising or Principal Component Analysis (PCA) also applies. The transformed data will subsequently be fed into the circuit with a sequence of gates, and the outcome will then be used to compute the loss.

2 Methods

2.1 PCA

Principal component analysis (PCA) is a statistical analysis algorithm which essentially transforms the features of observed data, which might be correlated, to a set of linearly uncorrelated variables,

these variables are known as "principal components". The rank of the principal components are determined according to their variances, i.e. the first principal component corresponds to that with the largest variance.
The original dataset consists of 569 entries, each with 30 features. There are two target groups: "0" for benign and "1" for malignant. Fig. 2 shows the distribution of the two target groups.

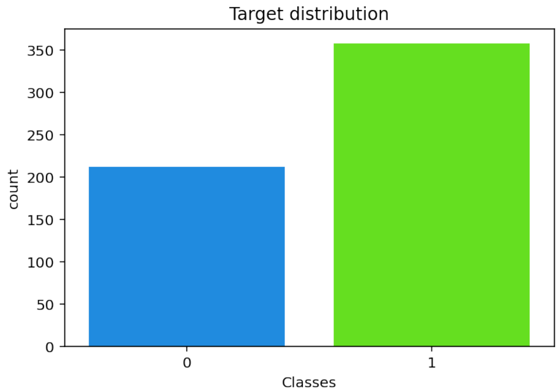

Figure 2: Distribution of the two target groups: "0" stands for benign and "1" for malignant.

There are 30 features in total, which is way beyond the scope of quantum circuits, thus we want to find a way to reduce the dimensionality of feature space. Since some of the features are correlated, and some appear to be self-independent, we want to quantitatively determine the degree of correlation between the features, for this we can take advantage of the correlation matrix, which calculates the correlation between any pair of features. Fig. 3 shows the correlation matrix of the 30 features (together with the target).

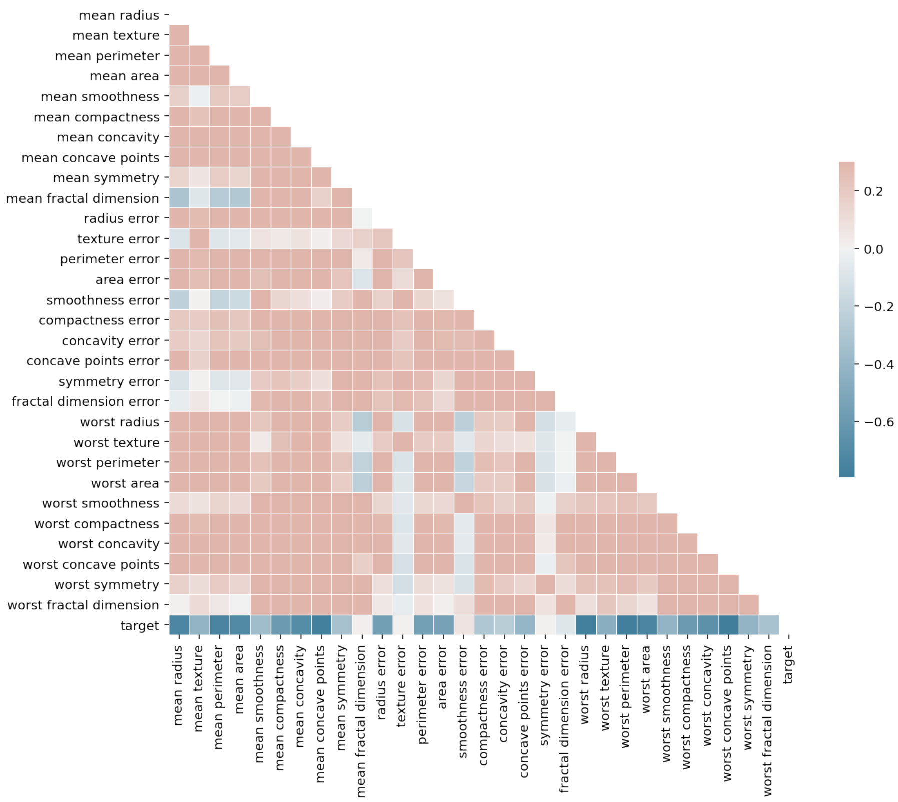

Figure 3: Correlation matrix showing the correlation between the features.

The colour(values) given by the correlation matrix measures the degree of correlation between two features, if it is close to 1, it means the two features are of positive correlation; if it is close to -1, then the two features are of negative correlation; if it is close to 0, then it means the two features are self-independent. The last row of the matrix gives the correlations between a feature and the target. To select features that are useful for successfully separating the two target groups, the following is done:

- In the last row of the correlation matrix, features that are of Pearson's correlation values greater than 0.75 are dropped.
- For the remaining rows, features with any value greater than 0.5 are dropped.

Fig. 4 shows the Pearson's correlation values between features and the target.

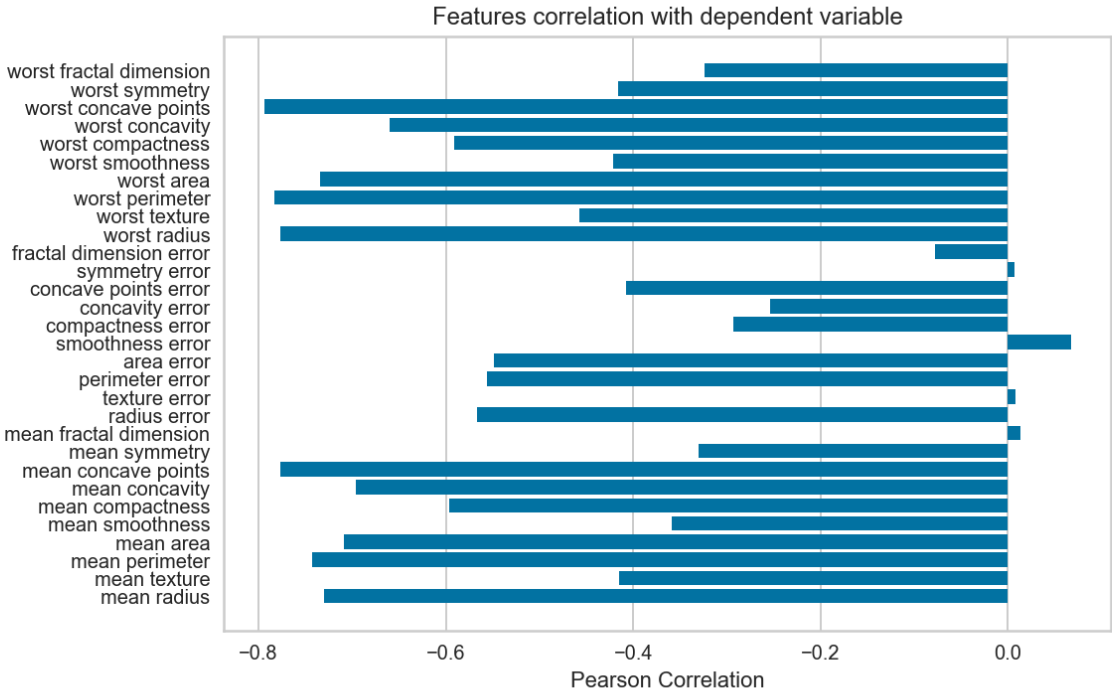

Figure 4: The Pearson correlation values between the features and the targets.

This leaves us 19 features in the end. Fig. 5 shows the distributions of the two target group over the selected feature space.

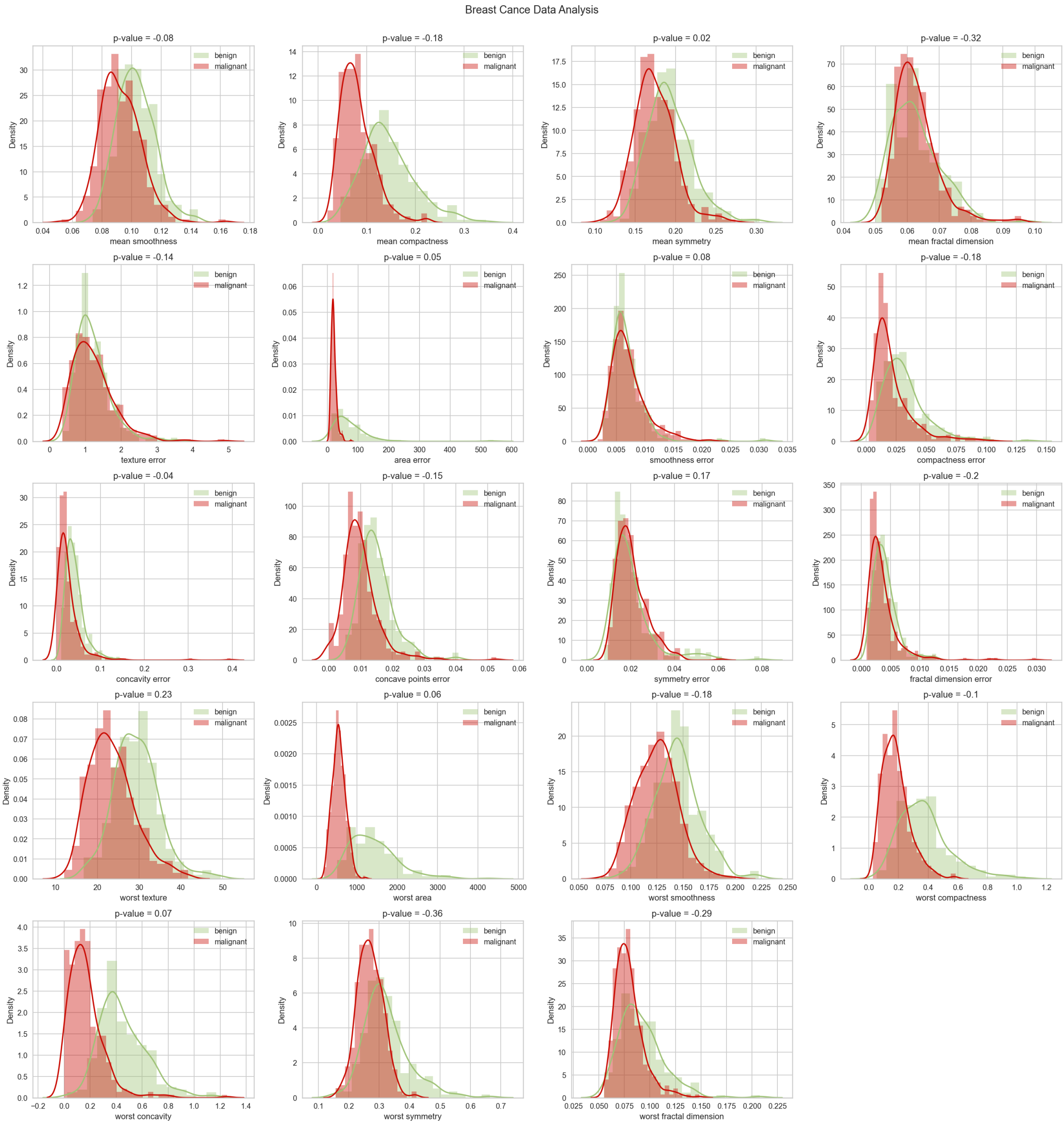

Figure 5: The distributions of the two target groups over different features.

With the selected 19 features, we can finally implement PCA algorithm. This is done by using `sklearn.decomposition.PCA`. With the parameter `n_components` specified as the number of qubits to use in the parametrised quantum circuit, we can reduce the dimensionality of the selected features to the number of qubits. Fig. 6 shows the scatter plot of the two target groups on principal component 1 and principal component 2, for the 2-qubit case.

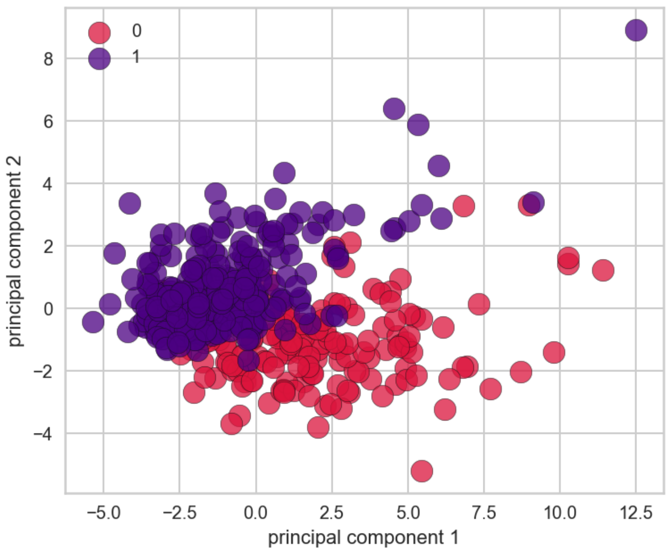

Figure 6: Scatter plot of the two target groups on the principal component space.

Now that the features have been prepared and ready for the quantum circuit, we need to build our quantum circuit to train the data. The quantum circuit consists of two part: a feature map, which maps classical data into quantum states. The other is the variational circuit, whose parameters are used to compute the cost function, and affects the results of training. From my experience of the previous Quantum Algorithms' project, the following feature map and variational circuit are chosen, which can be seen in Fig. 7 and Fig. 8, respectively.

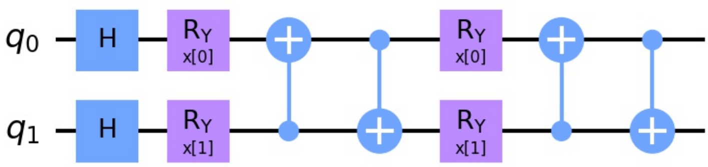

Figure 7: The feature map used in this project. The number of layers is set to be 2.

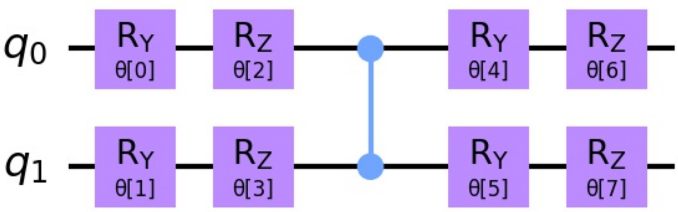

Figure 8: The variational circuit used in this project. The number of layers is set to be 3.

2.2 Cost function

Very much like a classical classifier, for the PQC we also need to specify a cost function. For this project, since the problem to solver is the classification between "0"s and "1"s, cross-entropy loss is chosen as the cost function. The cost function is defined as (Zhang & Sabuncu, 2018)

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{N} p_i \log q_i \tag{1}$$

where $\mathbf{p} \in [0,1]^N$ corresponds to the classification targets (labels), $\mathbf{q} \in [0,1]^N$ corresponds to the output predicted by the quantum classifier (predictions). One should notice that, in practice the cost function is normalised by a factor of N [1].

2.3 Cross validation

Lastly, cross validation is implemented, so that we can make sure the model does not overfit. It works as follows. Firstly, the data set with reduced features are split into k folds with equal size, and we loop over the divided folds: the iterated fold will be used for training the network, and the remaining two folds are used only when the training is completed and as test set. We do that for all folds and in the end we have k number of estimations of the accuracy on the test set. Then by taking the average value we have the final estimated accuracy on the test set. Additionally, another validation set can be created in the beginning, and can be used for final validation of the model in the end of k-fold training. For all experiments in this project, 3-fold cross validation is implemented.

3 Results

There are a number of parameters that may influence on the performance of the quantum circuit: number of shots, layers of the feature map, layers of the variational circuit, circuit architecture... In this project, I will focus on the impact of the number of qubits.

3.1 2 qubits

Fig. 9 shows the cost functions for the 3 folds (on the training set), for a 2-qubit parametrised quantum circuit with above described 2-layer feature map and 3-layer varational circuit.

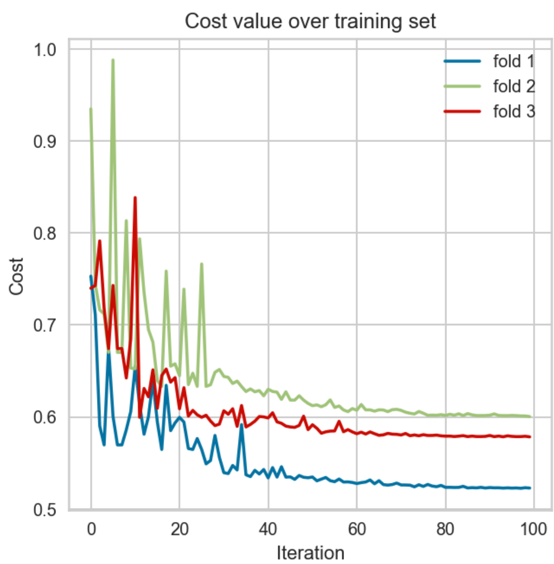

Figure 9: The cost functions for a 2-qubit parametrised quantum circuit, with 3-fold cross validation.

Fig. 9 shows the cost functions for the 3 folds, for a 3-qubit parametrised quantum circuit with above described 2-layer feature map and 3-layer varational circuit.

[1]see the qiskit documentation for details.

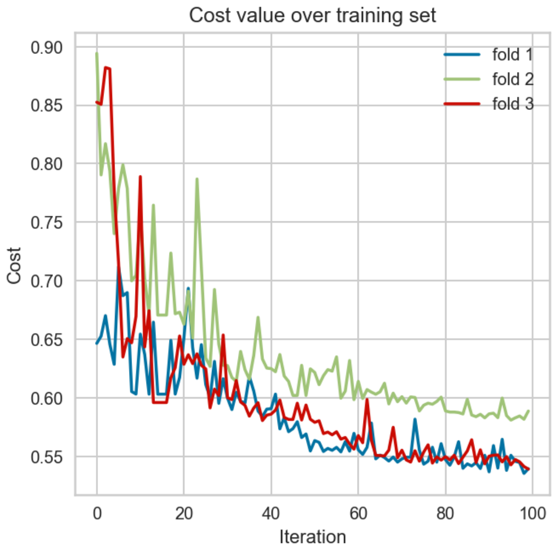

Figure 10: The cost functions for a 3-qubit parametrised quantum circuit, with 3-fold cross validation.

Fig. 9 shows the cost functions for the 3 folds, for a 4-qubit parametrised quantum circuit with above described 2-layer feature map and 3-layer varational circuit.

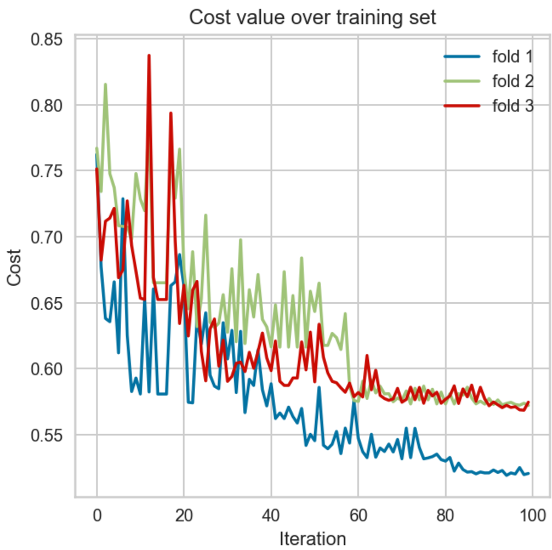

Figure 11: The cost functions for a 4-qubit parametrised quantum circuit, with 3-fold cross validation.

From Fig. 9- 11 it can be seen, for each case, the cost function reached convergence for all 3 folds, and did not escape from the local minimum. Now we can have a look at the accuracies on the test set.

Table 1: Test set accuracy for the 3 experiments.

Experiments	Fold 1	Fold 2	Fold 3	Avg.
2 qubit	0.60	0.73	0.72	0.68
3 qubit	0.68	0.65	0.69	0.67
4 qubit	0.61	0.63	0.79	0.68

From Table 1 it can be seen, with increasing number of qubits, the average accuracy on the test set over 3 folds does not improve significantly.

4 Conclusion

In this project, I implemented PQC to solve a classification problem with high-dimensional data. The BreastCancer data set initially consists of 569 entries, with 30 features. After feature selection based on their Pearson's correlation values, 19 features were left. These selected features were then standardised and reduced to the number of qubits for the following experiments, using PCA algorithm. It is found, with other parameters of the quantum circuits fixed: number of shots, circuit architecture, number of layers, etc., the accuracy on the test set stays almost invariant, with increasing number of qubits. From the cost functions we can learn that, the fluctuations grow with increasing number of qubits, making it harder and harder for the quantum circuit to be stablised. The fact that accuracy is almost invariant with increasing number of qubits, might be a result of increasing stochasticity. It can be learned that, if our goal is to increase the accuracy on the test set, a more promising solution might be to tune other parameters.

References

Benedetti, M., Lloyd, E., Sack, S., & Fiorentini, M. 2019, Quantum Science and Technology, 4, 043001

Zhang, Z., & Sabuncu, M. R. 2018, arXiv preprint arXiv:1805.07836

Appendix

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


get_ipython().run_line_magic('config', "InlineBackend.figure_format = 'retina'")

import time
import numpy as np

from sklearn.datasets import load_wine,load_breast_cancer


# In[2]:


data = load_breast_cancer()


# In[3]:


data.data.shape


# In[4]:


print(data['DESCR'])


# In[5]:


import matplotlib.pyplot as plt
```

```
import seaborn as sns
fig = plt.figure()
sns.countplot(x="target", data=data, palette="gist_rainbow_r")
plt.xlabel("Classes")
plt.title("Target distribution")
plt.show()


# In[9]:


import seaborn as sns
import pandas as pd

d = pd.DataFrame(data.data, columns=data.feature_names)
d['target'] = data['target']
d.head()
# Compute the correlation matrix
corr = d.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})


# In[10]:


from yellowbrick.target import FeatureCorrelation
from yellowbrick.target.feature_correlation import feature_correlation
X = data['data']
y = data['target']
features = np.array(data['feature_names'])
#visualizer = FeatureCorrelation(labels=features)
visualizer = feature_correlation(X, y,labels=features)

#visualizer.fit(X, y)        # Fit the data to the visualizer
visualizer.show()


# In[11]:



# absolute for positive values
abs_corr = abs(d.corr()["target"])
abs_corr = np.array(np.abs(abs_corr))
# random threshold for features to keep
relevant_features = abs_corr<0.75


# In[13]:


ind = np.arange(30)[relevant_features[:30]]


# In[45]:


def select_20(x,tol=.9):
    to_drop = []
    for i in range(x.shape[0]):
```

```
        for j in range(i):
            if np.abs(x[i][j])>tol:
                to_drop.append(j)
    return np.unique(to_drop)




# In[48]:


col_ = select_20(np.array(corr)[:30,:30])
col_rem = np.delete(np.arange(30),col_)
ind = np.intersect1d(ind,col_rem)
print(len(ind))


# In[49]:


from scipy.stats import pearsonr

fig = plt.figure(figsize = (20, 25))
j = 0
for num,i in enumerate(ind):
    ax = plt.subplot(6, 4, j+1)
    j += 1
    ax = sns.distplot(data.data[:,i][data.target==0], color='g', label = 'benign')
    ax = sns.distplot(data.data[:,i][data.target==1], color='r', label = 'malignant')
    p = pearsonr(data.data[:,i][data.target==0][:40],data.data[:,i][data.target==1][:40])[0]
    ax.set_title('p-value = '+str(round(p,2)))
    ax.set_xlabel(data.feature_names[i])
    ax.legend(loc='best')
fig.suptitle('Breast Cance Data Analysis')
fig.tight_layout()
fig.subplots_adjust(top=0.95)
plt.show()


# In[12]:


from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler



len_0 = len(data.target[data.target==0])
len_1 = len(data.target[data.target==1])
new_len = np.min([len_0,len_1])

newtarget = np.hstack((data.target[:new_len],data.target[len_1:len_1+new_len]))
pca = PCA(n_components=2)
scaler = StandardScaler()
features = scaler.fit_transform(data.data[:,ind])
features = pca.fit_transform(features)
newdata = np.vstack((features[:new_len],features[len_1:len_1+new_len]))
data0 = np.column_stack((newtarget.reshape(len(newtarget),1),newdata[:,:3]))


# In[51]:


plt.figure(figsize=(6,4))
lbs = data0[:,0]
i1 = np.where(lbs==0)[0]
i2 = np.where(lbs==1)[0]
plt.figure(figsize=(6,5))
plt.scatter(data0[:,1][i1],data0[:,2][i1],edgecolors='k',alpha=0.75,s=150,color='crimson',label='0')
plt.scatter(data0[:,1][i2],data0[:,2][i2],edgecolors='k',alpha=0.75,s=150,color='indigo',label='1')
```

```
plt.xlabel('principal component 1')
plt.ylabel('principal component 2')
plt.legend()
plt.show()


# In[6]:



from qiskit import Aer,BasicAer,QuantumCircuit
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.aqua.algorithms import VQC
from qiskit.aqua.components.optimizers import SPSA
from qiskit.circuit import ParameterVector
from qiskit.circuit.library import ZZFeatureMap, ZFeatureMap, PauliFeatureMap
from qiskit.circuit.library import  RealAmplitudes, EfficientSU2, TwoLocal,
    NLocal, PauliTwoDesign,RealAmplitudes
from qiskit.aqua.components.optimizers import ADAM, SPSA, COBYLA, AQGD, TNC,
    SLSQP, L_BFGS_B


class DataSet(object):

    """Docstring for CrossValidation. """

    def __init__(self, data, k=None, A=0, B=1):
        """TODO: to be defined. """
        self.data = data.copy()
        self.k = k
        self.A = A
        self.B = B
        #self.C = C
        self.dataA = self.data[self.data[:,0]==self.A]
        self.dataB = self.data[self.data[:,0]==self.B]
        #self.dataC = self.data[self.data[:,0]==self.C]

    def CV(self, data):
        k = self.k
        size = len(data)
        index = np.arange(size)
        np.random.shuffle(index)
        DATA_IND={}
        ind_split = np.array_split(index, k)
        for name in range(k):
            sub = {}
            train_split = []
            for i in range(k):
                if i==name:
                    sub['test_ind'] = ind_split[i]
                else:
                    train_split.append(ind_split[i])
            sub['train_ind'] = np.hstack(train_split)
            DATA_IND.update({name: sub})
        return DATA_IND, index

    def data_gen(self):
        ind, _ = self.CV(self.dataA)
        for i in range(self.k):
            tr_input = {
                'A': self.dataA[ind[i]['train_ind']][:,1:],
                'B': self.dataB[ind[i]['train_ind']][:,1:]}
            te_input = {
                'A': self.dataA[ind[i]['test_ind']][:,1:],
                'B': self.dataB[ind[i]['test_ind']][:,1:]}
            te_label = (self.dataA[ind[i]['test_ind']][:,:1],
                        self.dataB[ind[i]['test_ind']][:,:1])
            yield {
                'k': i,
                'tr_input': tr_input,
                'te_input': te_input,
                'testing_input': np.concatenate((te_input['A'], te_input['B'])),
```

```
                    'testing_label': np.concatenate(te_label)
                     }



class Main(object):

    """Docstring for CrossValidation. """

    def __init__(self, shots=2048, reps=2, num_qubits=3, fm_func=None, vc_func=None, seed=10598):
        """TODO: to be defined. """
        self.shots = shots
        self.reps = reps
        self.num_qubits = num_qubits
        self.fm_func = fm_func(reps=self.reps,num_qubits=self.num_qubits)
        self.vc_func = vc_func(reps=self.reps+1,num_qubits=self.num_qubits)
        self.seed = seed
        self.counts = []
        self.values = []
        self.backend = Aer.get_backend('qasm_simulator')
        self.backend_options = {"method": "statevector", "max_parallel_threads": 0, "max_parallel_experiments": 2,"max_parallel_shots": 1}

    def drawfmap(self, fname):
        return self.fm_func.draw(output='mpl', filename=fname+'_fmap.jpg')

    def drawvmap(self, fname):
        return self.vc_func.draw(output='mpl', filename=fname+'_var_circ.jpg')

    def call_back_vqc(self, eval_count, var_params, eval_val, index):
        text = "index({}): current cross entropy cost: {}".format(eval_count, eval_val)
        self.counts.append(eval_count)
        self.values.append(eval_val)
        #print(text)

    def optimization(self, training_input, test_input):
        self.quantum_instance = QuantumInstance(
            self.backend,
            shots=self.shots,
            seed_simulator=self.seed,
            seed_transpiler=self.seed,
            backend_options=self.backend_options)
        self.vqc = VQC(optimizer=self.opt,
          feature_map=self.fm_func,
          var_form=self.vc_func,
          callback=self.call_back_vqc,
          training_dataset=training_input,
          test_dataset=test_input)

    def train(self, training_input, test_input):
        self.optimization(training_input, test_input)
        start = time.process_time()

        result = self.vqc.run(self.quantum_instance)
        #val_prob, val_labels = self.vqc.predict(val_input)

        print("time taken: ")
        print(time.process_time() - start)
        print("testing success ratio: {}".format(result['testing_accuracy']))
        return result['testing_accuracy']






# In[7]:


np.__version__
```

```python


# In[8]:


import qiskit
qiskit.__version__


# In[9]:


def feature_map_expr1(reps=2, num_qubits=3):

    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        for i in range(num_qubits):
            feature_map.ry(x[i], i)
            feature_map.rz(x[i], i)
        for i in range(num_qubits - 1, 0, -1):
            feature_map.cx(i, i-1)

    return feature_map

def feature_map_expr2(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        for i in range(num_qubits):
            feature_map.rx(x[i], i)
            feature_map.rz(x[i], i)
        for i in range(num_qubits-1, 0, -1):
            feature_map.cx(i, i-1)
    return feature_map


def feature_map_expr3(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        for i in range(num_qubits):
            feature_map.rx(x[i], i)
            feature_map.rz(x[i], i)
        for i in range(num_qubits-1, 0, -1):
            feature_map.cz(i, i-1)
    return feature_map

def feature_map_expr4(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        for i in range(num_qubits):
            feature_map.rx(x[i], i)
            feature_map.rz(x[i], i)
        for control in range(num_qubits-1, -1, -1):
            for target in range(num_qubits-1, -1, -1):
                if control != target:
                    feature_map.rz(x[target], target)
                    feature_map.cx(control, target)
                    feature_map.rz(x[target], target)
    return feature_map

def feature_map_expr5(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
```

```
        for i in range(num_qubits):
            feature_map.rx(x[i], i)
            feature_map.rz(x[i], i)
        feature_map.barrier()
        for control in range(num_qubits-1, 0, -1):
            target = control - 1
            feature_map.rx(x[target], target)
            feature_map.cx(control, target)
            feature_map.rx(x[target], target)
            feature_map.barrier()
        for i in range(num_qubits):
            feature_map.rx(x[i], i)
            feature_map.rz(x[i], i)
        feature_map.barrier()
    return feature_map


def feature_map_expr6(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        for i in range(num_qubits):
            feature_map.ry(x[i], i)
            feature_map.rz(x[i], i)
        for i in range(num_qubits - 1, 0, -1):
            feature_map.cz(i, i-1)
        feature_map.ry(x[1], 1)
        feature_map.rz(x[1], 1)
    return feature_map


def feature_map_expr7(reps=2, num_qubits=3):
    feature_map = QuantumCircuit(num_qubits)
    x = ParameterVector('x', length=num_qubits)

    for _ in range(reps):
        #for i in range(num_qubits):
        #    feature_map.h(i)
        for i in range(num_qubits):
            feature_map.ry(x[i], i)
        feature_map.cx(num_qubits-1, 0)
        for i in range(num_qubits-1):
            feature_map.cx(i, i+1)
        for i in range(num_qubits):
            feature_map.ry(x[i], i)
        feature_map.cx(num_qubits - 1, num_qubits - 2)
        feature_map.cx(0, num_qubits - 1)
        for i in range(1, num_qubits - 1):
            feature_map.cx(i, i-1)
    return feature_map


def feature_map_expr8(reps=2, num_qubits=3):

    return ZZFeatureMap(feature_dimension=num_qubits, reps=reps)

def variational_circuit(reps=3, num_qubits=2):

    #var_circuit = EfficientSU2(num_qubits, entanglement='linear', reps=2,
    insert_barriers=True)
    var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], 'cz', reps=3)
    # var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], ['cx'],
    entanglement='linear', reps=4, insert_barriers=True)

    return var_circuit

def variational_circuit1(reps=3, num_qubits=2):

    var_circuit = EfficientSU2(num_qubits, entanglement='linear', reps=3)
    #var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], 'cz', reps=reps)
```

```
    # var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], ['cx'], entanglement='linear', reps=4, insert_barriers=True)

    return var_circuit
def variational_circuit2(reps=3, num_qubits=2):

    #var_circuit = EfficientSU2(num_qubits, entanglement='linear', reps=2, insert_barriers=True)
    var_circuit = PauliTwoDesign(num_qubits, reps=reps)
    # var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], ['cx'], entanglement='linear', reps=4, insert_barriers=True)

    return var_circuit

def variational_circuit3(reps=3, num_qubits=2):

    #var_circuit = EfficientSU2(num_qubits, entanglement='linear', reps=2, insert_barriers=True)
    var_circuit = RealAmplitudes(num_qubits, reps=reps)
    # var_circuit = TwoLocal(num_qubits, ['ry', 'rz'], ['cx'], entanglement='linear', reps=4, insert_barriers=True)

    return var_circuit


# In[10]:


def wrap(data, k=5, feature_map=None, opt_func=None, var_circuit=None, opt_params={}, maxiter=100, reps=2, num_qubits=3,shots=2048, callback=False):
    mds = DataSet(data, k=k, A=0, B=1)
    log = "nqubits({}), reps {}, feature_map {}, opt_func {}:".format(
        num_qubits,
        reps,
        feature_map.__name__,
        opt_func.__name__
    )
    print("="*80+"\n"+log)
    Accuracy = []
    plt.figure(figsize=(5,5))
    with open('result.log', 'a+') as fp:
        result_log = '#'+log+'\n'
        fp.writelines(result_log)
        for d in mds.data_gen():
            print(log+" fold {}...".format(d['k']))
            run = Main(shots=shots,
                       reps=reps, num_qubits=num_qubits, fm_func=feature_map, vc_func=var_circuit)
            run.opt = opt_func(maxiter=maxiter, **opt_params)
            acc = run.train(d['tr_input'], d['te_input'])
            fp.writelines("{}, {}\n".format(d['k'], acc))
            Accuracy.append(acc)
            count = run.counts
            vals = run.values
            plt.plot(count,vals,label='fold '+str(d['k']+1))
    plt.title('Cost value over training set')
    plt.xlabel('Iteration')
    plt.ylabel('Cost')
    plt.legend()
    plt.show()

    return np.asarray(Accuracy)
        #run.drawfmap(log.replace(' ',"").replace(",","").replace(":","").replace("(","").replace(")",""))
        #run.drawvmap(log.replace(' ',"").replace(",","").replace(":","").replace("(","").replace(")",""))



# ## Qubits

# In[13]:
```

```
len_0 = len(data.target[data.target==0])
len_1 = len(data.target[data.target==1])
new_len = np.min([len_0, len_1])

newtarget = np.hstack((data.target[:new_len],data.target[len_1:len_1+new_len]))
pca = PCA(n_components=3)
scaler = StandardScaler()
features = pca.fit_transform(data.data[:,ind])
features = scaler.fit_transform(features)
newdata = np.vstack((features[:new_len],features[len_1:len_1+new_len]))
data1 = np.column_stack((newtarget.reshape(len(newtarget),1),newdata[:,:3]))


# In[14]:


newtarget = np.hstack((data.target[:new_len],data.target[len_1:len_1+new_len]))
pca = PCA(n_components=4)
scaler = StandardScaler()
features = pca.fit_transform(data.data[:,ind])
features = scaler.fit_transform(features)
newdata = np.vstack((features[:new_len],features[len_1:len_1+new_len]))
data2 = np.column_stack((newtarget.reshape(len(newtarget),1),newdata[:,:4]))


# In[60]:


fig, ax = plt.subplots(1,2,figsize=(12,4))
lbs = data1[:,0]
i1 = np.where(lbs==0)[0]
i2 = np.where(lbs==1)[0]
ax[0].scatter(data1[:,1][i1],data1[:,2][i1],edgecolors='k',alpha=0.75,s=150,color='crimson',label='0')
ax[0].scatter(data1[:,1][i2],data1[:,2][i2],edgecolors='k',alpha=0.75,s=150,color='indigo',label='1')
ax[0].set_xlabel('feature 1')
ax[0].set_ylabel('feature 2')
ax[0].legend()

ax[1].scatter(data1[:,1][i1],data1[:,3][i1],edgecolors='k',alpha=0.75,s=150,color='crimson',label='0')
ax[1].scatter(data1[:,1][i2],data1[:,3][i2],edgecolors='k',alpha=0.75,s=150,color='indigo',label='1')
ax[1].set_xlabel('feature 1')
ax[1].set_ylabel('feature 3')
ax[1].legend()
plt.show()


# In[61]:


data0.shape


# In[ ]:


qubits = [data0,data1,data2]
acc_fmap = np.zeros((len(qubits),3))
for i in range(2):
    params = {"k":3, "maxiter":100, "num_qubits":qubits[i].shape[1]-1, "reps":2}
    params["var_circuit"] = variational_circuit
    params["feature_map"] = feature_map_expr7
    params["opt_func"] = COBYLA
    params["opt_params"] = {"disp":True, "tol":1e-6}
    acc_fmap[i] = wrap(qubits[i], **params)


# In[69]:


data2.shape
```

```python


# In[16]:


qubits = [data0,data1,data2]

params = {"k":3, "maxiter":200, "num_qubits":qubits[2].shape[1]-1, "reps":2}
params["var_circuit"] = variational_circuit
params["feature_map"] = feature_map_expr7
params["opt_func"] = COBYLA
params["opt_params"] = {"disp":True, "tol":1e-6}
acc_fmap[i] = wrap(qubits[2], **params)


# In[ ]:


acc_fmap = np.array([0.6018518518518519,0.7314814814814815,]).reshape(3,3)


# In[100]:


x = [2,3,4]
y = np.mean(acc_fmap,axis=1)
yerr = np.std(acc_fmap,axis=1)
plt.figure(figsize=(6,5))
plt.errorbar(x,y,yerr=yerr,ls='none',marker='o')
plt.title('Test set accuracy for different number of qubits')
plt.xlabel('Number of qubits')
plt.ylabel('Test set accuracy')
plt.show()
```

AQA_assignment1.py