QClassify Documentation

Yudong Cao, Sukin Sim

Feb 07, 2019

Contents

QClassify is a Python framework for implementing variational quantum classifiers. The goal is to provide a generally customizable way of performing classification tasks using gate-model quantum devices. The quantum devices can be either simulated by a quantum simulator or a cloud-based quantum processor accessible via Rigetti Computing's Quantum Cloud Services.

Variational quantum classification is a paradigm of supervised quantum machine learning that has been investigated actively in the quantum computing community (See for instance Farhi and Neven, Schuld et al., Mitarai et al. and Havlicek et al.). The general framework adopted in the design of QClassify follows from these contributions in the literature.

The workflow can be summarized in Figure 1:

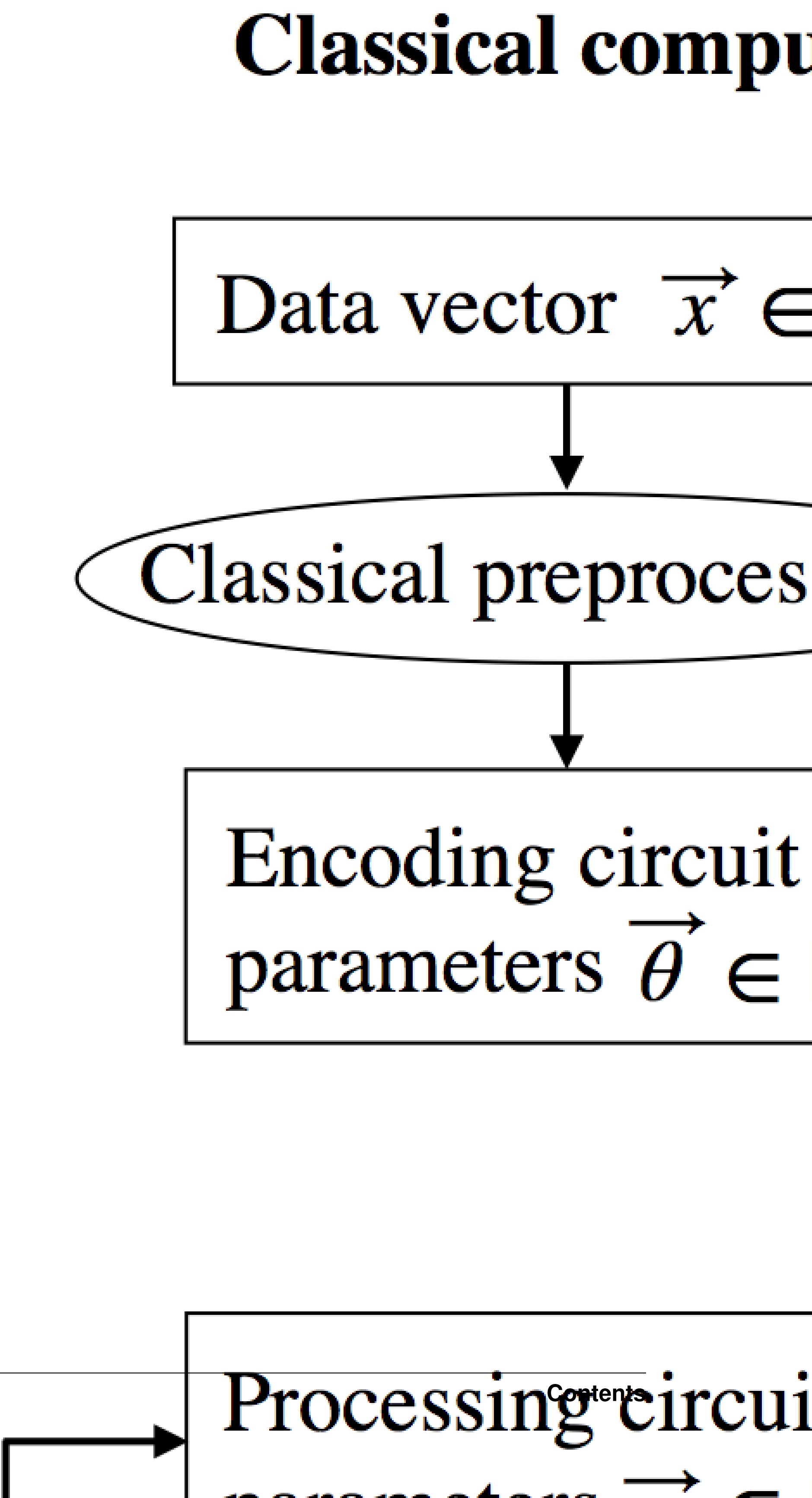

Classical compu
Data vector $\vec{x} \in$
Classical preproces
Encoding circuit
parameters $\vec{\theta} \in$
Processing circui

Figure 1: Diagram illustrating the workflow of QClassify. Each rectangle represents a data object and each oval represents a method.

CHAPTER 1

Contents

1.1 Installing QClassify

To install QClassify using `pip`:

```
pip install qclassify
```

To instead install QClassify from source, clone this repository, `cd` into it, and run:

```
git clone https://github.com/zapatacomputing/QClassify
cd QClassify
python -m pip install -e .
```

Try executing `import qclassify` to test the installation in your terminal.

Note that the pyQuil version used requires Python 3.6 or later. For installation on a user QMI, please click here.

1.2 Introduction to Quantum Classifiers

1.2.1 Main Components

The main data structure describing the quantum classifier setting is in *qclassifier.py*. The implementation allows for modular design of the following components of a quantum classifier (Figure 1):

1. **Encoder**: transforms a classical data vector into a quantum state. See *encoder.py*.
    - 1.1 **Classical preprocessor**: maps an input data vector to circuit parameters. See *preprocessing.py*.
    - 1.2 **Quantum state preparation**: applies the parametrized circuit to an all-zero input state to generate a quantum state encoding the input data. See *encoding_circ.py*.
2. **Processor**: extracts classical information from the encoded quantum state. See *processor.py*.

- 2.1 **Quantum state transformation**: applies a parametrized circuit to the encoded quantum state to transform it into a form more amenable for information extraction by measurement and classical postprocessing. See *proc_circ.py*.
- 2.2 **Information extraction**: extract classical information from the output quantum state. See *postprocessing.py*.
  - 2.2.1 **Measurement**: repeatedly run the quantum circuit, perform measurements and collect measurement statistics
  - 2.2.2 **Classical postprocessing**: Glean information from the measurement statistics and produce the output label of the quantum classifier.

1.2.2 Examples

We provide a Jupyter notebook to demonstrate the utility of QClassify.

Notebook	Feature(s)
qclassify_demo.ipynb	Uses a simple two-qubit circuit to learn the XOR dataset.

1.2.3 How to cite QClassify

When using QClassify for research projects, please cite:

Sukin Sim, Yudong Cao, Jonathan Romero, Peter D. Johnson and Alán Aspuru-Guzik. *A framework for algorithm deployment on cloud-based quantum computers*. arXiv:1810.10576. 2018.

1.3 Demo: Classification of XOR

Here we present a concrete example showing how to use the subroutines of QClassify. We consider binary classification of points on 2D plane using a 2-qubit quantum circuit. In particular the data points are distributed in an XOR-like fashion:

Here we encode a 2-dimensional vector into a 2-qubit state. For simplicity we set up the encoding such that the preprocessing step is trivial, namely each element of the data vector directly becomes a rotation angle on the circuit. A non-trivial classical preprocessing step may be found in for example this paper. The circuit encoding is then . We then use a parametrized circuit to perform further classification.

The complete variational circuit is the following:

The encoding circuit serves as a feature map which maps a data point into a 2-qubit state. The second part is a processing circuit, which is a variational circuit with two parameters and . The postprocessing steps consist of collecting the measurement statistics of the top qubit in the basis and gleaning the probability of measuring in the top qubit. The objective of training is to find a parameter setting which minimizes the cross entropy between the circuit outputs on the training data and the training labels:

where the sum is over the training set, is the training label of the -th data point and is the probability that the top qubit measures for the -th training point.

Now we describe step by step how to this up in QClassify. Before we start, note that the aim of this framework is to maximize customization. Therefore for a specific classification task the user can specify her own subroutines that suit the particular problem setting that is being considered.

First, we need to specify how classical data vectors are to be encoded in a quantum state. This is done in two parts:

1. Preprocess the classical data vector into circuit parameters. This is specified in `preprocessing.py`. For the purpose of this example we use the identity function `id_func` which simply returns the input.
2. For a given set of parameters , build an -qubit circuit which prepares the state . This is specified in `encoding_circ.py`. For the specific example considered here we have built the `x_product` function to realize the encoding circuit .

Having built these two functions, we specify the settings for encoding circuit in a dictionary as below.

```
[1]: import qclassify

from qclassify.preprocessing import *
from qclassify.encoding_circ import *

qencoder_options={
                'preprocessing':id_func,         # see preprocessing.py
                'encoding_circ':x_product,       # see encoding_circ.py
        }
```

We then move on to describe the processing of the encoded quantum state for classification. This is also a two-part effort:

1. Variational circuit for transforming the encoded state into a final state . This is specified in `proc_circ.py`. For the particular variational circuit used in this example we have built the function `layer_xz`. There is a specific template that a function should follow. The template is described in the doc string at the beginning of `proc_circ.py`.
2. Extracting information from the output state . This is specified in `postprocessing.py`. Here we have the opportunity to specify the measurements that are made to the output state, as well as any classical postprocessing functions that turn the measurement statistics into a class label. For this particular we use `measure_top` for specifying measurement on the top qubit and `prob_one` for computing the probability of measuring as the classifier output.

Having constructed these functions, we specify the settings for processing of the encoded state in a dictionary below. Here under `'postprocessing'` tab the `'quantum'` entry is for measurements and `'classical'` is for classical postprocessing of measurement statistics.

```
[2]: from qclassify.proc_circ import *
from qclassify.postprocessing import *

qproc_options={
                'proc_circ':layer_xz,    # see proc_circ.py
                'postprocessing':{
                        'quantum':measure_top, # see postprocessing.py
                        'classical':prob_one, # see postprocessing.py
                }
        }
```

With all the components of the classifier specified we can now create a `QClassify` object which encapsulates all the information in the components.

```
[3]: from qclassify.qclassifier import QClassifier

qubits_chosen = [0, 1] # indices of the qubits that the circuit will act on
qclassifier_options = {
                'encoder_options':qencoder_options,
                'proc_options':qproc_options,
        }
qc = QClassifier(qubits_chosen, qclassifier_options)
```

So far the `QClassifier` contains only abstract information about the classifier. In order to obtain a concrete circuit, let's plug in concrete assignments of the input vector and parameters .

[4]:
```
input_vec = [1, 1] # input vector
params = [3.0672044712460114,\
          3.3311348339721203] # parameters for the circuit

qc_prog = qc.circuit(input_vec, params) # assign the input vector and parameters into␣
↪the circuit
print(qc_prog)
```

```
RX(1) 0
RX(1) 1
CZ 0 1
RX(3.0672044712460114) 0
RX(3.3311348339721203) 1
DECLARE ro BIT[1]
MEASURE 0 ro[0]
```

The `circuit` method of the `QClassifier` object not only returns a concrete circuit with input vectors and parameters assignments, but also stores the circuit information internally. This allows one to execute the quantum classifier and return a particular classification output.

[5]:
```
prob = qc.execute()
print(prob)
```

```
0.7858
```

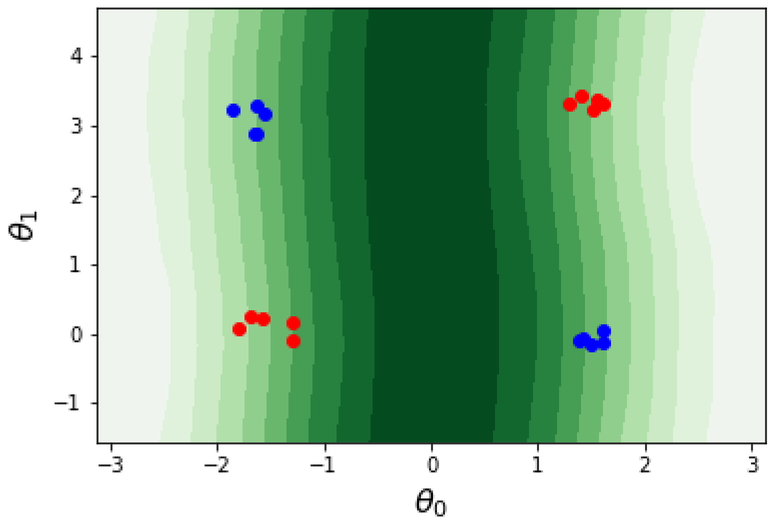
In addition if we would like to show the decision boundary of the classifier for a particular pair of features in the input vector we can use the `plot_decision_boundary` function.

[6]:
```
from math import pi

input_vec = [0,0]    # Input vector. Here [0,0] is used only as a placeholder since␣
↪both features are chosen to be
                     # swept over.
features_chosen = [0,1] # Indices of features that are chosen. Here there are only␣
↪two features so naturally we let it
                     # be [0,1].
filename = 'qclassifier_db.pdf' # name of the plot to be saved

plot_db_options = {
                'nmesh':10,       # number of grid points
                'xmin':-pi,      # min. limit on the x axis
                'xmax':pi,       # max. limit on the x axis
                'ymin':-pi/2,    # min. limit on the y axis
                'ymax':3*pi/2    # max. limit on the y axis
        }

qc.plot_decision_boundary(input_vec, features_chosen, filename, plot_db_options)
```



From the above decision boundary plot we can see that apparently the classifier is not doing a great job in separating the data points. Therefore we need to tune the parameters to improve it. The standard training objective function is the cross entropy (see `crossentropy` in `training.py`). The training data needs to be provided as a Python list of tuples `(features, label)` where `features` is a list of floating point numbers and `label` is either 0 or 1.

```
[7]: from qclassify.xor_example import *
     from qclassify.training import *

     train_options={
                     'training_data':XOR_TRAINING_DATA,       # Example. See xor_example.py
                     'objective_func':crossentropy,  # See training.py
                     'training_method':'nelder-mead', # Optimization algorithm
                     'init_params':[3.0672044712460114, 3.3311348339721203], # initial␣
     ↪guess for parameters
                     'maxiter':10, # maximum number of iterations
                     'xatol':1e-3, # tolerance in solution
                     'fatol':1e-3, # tolerance in function value
                     'verbose':True # Print intermediate values
             }

     qc.train()
```

```
Iter    Obj
   1    0.018
   2    0.018
   3    0.019
   4    0.014
   5    0.014
   6    0.013
   7    0.013
   8    0.013
   9    0.014
  10    0.013
  11    0.013
  12    0.014
  13    0.013
  14    0.014
  15    0.013
```

(continues on next page)

(continued from previous page)

```
16    0.013
17    0.014
18    0.013
19    0.014
```

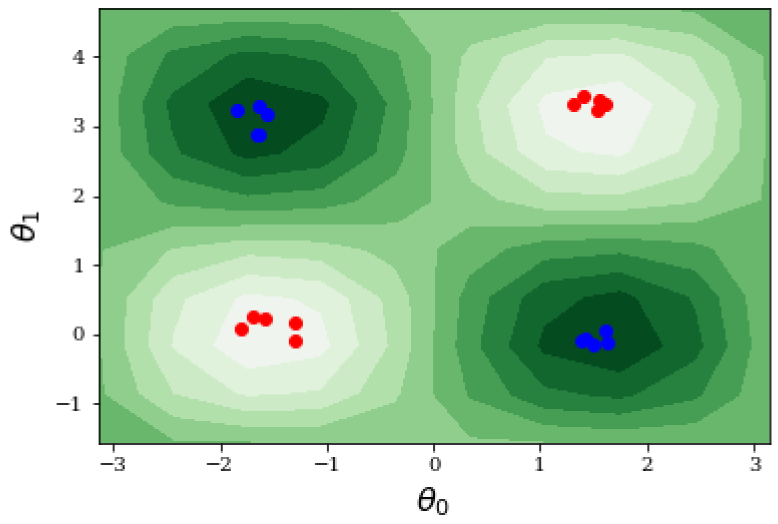
The `min_loss_history` attribute of `QClassifier` class stores the value of the objective function each time it is called during the optimization. This may be used for further processing (for example plotting the training curve).

[8]:
```
print(qc.min_loss_history)
```

```
[0.017852202344453517, 0.01848265965455959, 0.01860949110296039, 0.013617471648894883,␣
↪0.013810264525585902, 0.013235283510689566, 0.01337114296678497, 0.
↪01328582257290195, 0.01385394241480525, 0.012893824496953976, 0.01346527546182597,␣
↪0.013501980887877024, 0.012937323886782048, 0.013930548172307659, 0.
↪01339673874455748, 0.01258386751929006, 0.013696623567530642, 0.012857984171322261,␣
↪0.013550070439161298]
```

After training we again plot the decision boundary. It seems that the classifier is able to perform much better on the training data.

[9]:
```
qc.plot_decision_boundary([0,0], [0,1], 'qclassifier_db_train.pdf', plot_db_options)
```



We may also test the classifier on previously unseen data points to see how well it has performed. For this specific example we use `gen_xor` for generating random data points for testing. But in general this can come from a different source, as long as the format of the data set follows the same structure as described previously for training data: A list of tuples `(feature, label)` where `feature` is a list of floats and `label` is a discrete value in . The `test` method in the class `QClassifier` allows one to test the performance of the classifier on a given test data set.

[10]:
```
from qclassify.xor_example import *
from qclassify.training import *

test_data = gen_xor(100, pi/8.) # generate more XOR-like testing data. Here we use␣
↪100 data points. The second value
                                # specifies how spread out the testing data are. See␣
↪xor_example.py for details.
test_options = {
                'objective_func': crossentropy, # Objective function used for␣
↪evaluating the classifier performance
```

(continues on next page)

(continued from previous page)

```
        }
print(qc.test(test_data), test_options) # print the objective function value over the␣
↪testing data
```

```
0.01589438299249235 {'objective_func': <function crossentropy at 0x11df2f730>}
```

1.3.1 Thoughts and suggestions?

Of course this repo is by no means complete. There are many improvements and additions that can be considered. If you have more thoughts and suggestions please feel free to contact Yudong Cao at yudong@zapatacomputing.com.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search