Quantum Machine Learning in Practice

Solving an Optimization Problem with the Variational Quantum Eigensolver

Kauê Miziara

2026-01-08

0 - Problem Definition

Consider a group of 6 people we want to divide into two teams, such that the skill level of each one is similar.

- Each person has a skill level
- Some people are goalkeepers

`Objective`: separate the players into two teams

- Teams must have equal (or similar) skills
- Not everyone needs to be chosen
- Teams do not necessarily need to have the same number of members

`Constraints`:

- Each player cannot be on more than one team
- Each team must have **exactly** one goalkeeper

Player Data

Player	Skill	Goalkeeper?
Alice	5	No
Bob	6	Yes
Claire	8	Yes
Danny	4	No
Eve	6	No

Player	Skill	Goalkeeper?
Frank	9	No

1 - Mapping

We'll use binary variables to write the problem in the QUBO model.

The variables $(x_0, \dots, x_5)$ represent the players in **Team A**
The variables $(x_6, \dots, x_{11})$ represent the players in **Team B**

If $x_i = 1$, the corresponding player is in the team associated with i; otherwise, $x_i = 0$.

> e.i., $x_0 = 1$ and $x_6 = 0$ means that Alice *is* in Team A and *is not* in Team B.

Objective Function

If we are going to create two teams, we can define the function as:

$$Obj = \left(\sum s_A - \sum s_B\right)^2$$

Where $\sum s_i$ represents the total skill of the players in Team i.

We square it so that the value is always positive, preventing the algorithm from assigning all players to the same team.

- This error could happen if Team B's skill is much higher than Team A's, generating a very negative result.
- By squaring, the lowest possible value is zero (if the teams have equal skill).

To make the result generic for N teams (besides enabling us to make adjustments to the model of specific teams), we can generalize the objective function by calculating specific objectives for each team.

To do this, we can calculate the skill difference between each team i and the "fair" distribution of skills, ED:

$$ED = \frac{\sum s_i}{N}$$

Where s_i is each person's skill and N is the number of teams.

In this case, the objective function will be:

$$Obj = \sum_i O_i$$

Where:

$$O_i = \left(\sum (s_i x_i) - ED\right)$$

With x_i being the variable indicating whether i should be on the team or not.

- We are referring to Alice, for instance, when either $i = 0$ or $i = 6$; therefore, $s_0 = s_6 = 5$.

Expanding the function in our case, we have:

$$ED = \frac{\sum s_i}{2} = 19$$

$$O_0 = \left(\sum_{i \in [0,5]} s_i x_i - ED\right)^2 = [(5x_0 + 6x_1 + 8x_2 + 4x_3 + 6x_4 + 9x_5) - ED]^2$$

$$O_1 = \left(\sum_{i \in [6,11]} s_i x_i - ED\right)^2 = [(5x_6 + 6x_7 + 8x_8 + 4x_9 + 6x_{10} + 9x_{11}) - ED]^2$$

$$Obj = [(5x_0 + 6x_1 + 8x_2 + 4x_3 + 6x_4 + 9x_5) - 19]^2 + [(5x_6 + 6x_7 + 8x_8 + 4x_9 + 6x_{10} + 9x_{11}) - 19]^2$$

Although this is a valid QUBO model problem, we can still encounter issues, such as:

- Allocating all players to a single team
- Allocating one player to all teams
- Not allocating players, keeping N teams with 0 members (consequently, $Obj = 0$, which is what we wanted, but not the intended way)

To get around possible problems, we may add constraints to the model.

Penalty Functions

For each constraint in the problem, we add a term to our model containing a function representing it.

The first constraint (a player cannot be on more than one team) can be written as:

$$x + y \leq 1$$

Since our variables can only take values 0 or 1, the inequality forces *at least* one of the variables to be equal to 0.

This type of constraint is so common that we have a ready-made penalty function for it:

$$\lambda(xy)$$

Where λ is a penalty constant, which will be decided later.

In this case, for each player (and their associated pair of variables), we will have a penalty term. For Alice:

$$x_0 + x_6 \leq 1 \rightarrow \lambda_1(x_0 x_6) = c_1$$

Our second type of constraint concerns the goalkeepers; for them, we do something similar.

To ensure each goalkeeper is in at most one team, we create a constraint stating that we cannot have more than one goalkeeper per team.

Since our example has two goalkeepers:

$$x_1 + x_2 \leq 1 \rightarrow \lambda_2(x_1 x_2) = c_7$$
$$x_7 + x_8 \leq 1 \rightarrow \lambda_2(x_7 x_8) = c_8$$

We will add the penalty functions to the objective function terms, ensuring that the constraints apply.

To choose the λ values, keep in mind that:

- We often use integer values
- They must be large enough to increase the objective function value if the constraint is violated
- They cannot be too high, so as not to bias the objective function
    - The model would prioritize satisfying all constraints over minimizing the function
- They cannot be too low, otherwise the model would allow too many violations

---

Mapping Code

Thus, our final function will be the cost function C such that:

$$C = Obj + \sum c_i$$

Where c_i are the penalty functions.

Therefore:

$$C = \sum O_i + \sum c_i$$

$$C = \left(\sum_{i \in [0,5]} s_i x_i - ED \right)^2 + \left(\sum_{i \in [6,11]} s_i x_i - ED \right)^2 + \lambda_1 (x_0 x_6 + \cdots + x_5 x_{11}) + \lambda_2 (x_1 x_2 + x_7 x_8)$$

We can finally map our problem to a quantum computer using Qiskit.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from typing import Sequence
```

```
from qiskit import QuantumCircuit
from qiskit.circuit.library import efficient_su2
```

```
from qiskit_aer.primitives import Estimator as AerEstimator
from qiskit_aer.primitives import Sampler as AerSampler
```

```python
from qiskit_optimization import QuadraticProgram
from qiskit_optimization.converters import QuadraticProgramToQubo
```

```python
# Problem data
players_data = [
    {"name": "Alice",  "strength": 5, "isGoalkeeper": False},
    {"name": "Bob",    "strength": 6, "isGoalkeeper": True },
    {"name": "Claire", "strength": 8, "isGoalkeeper": True },
    {"name": "Danny",  "strength": 4, "isGoalkeeper": False},
    {"name": "Eve",    "strength": 6, "isGoalkeeper": False},
    {"name": "Frank",  "strength": 9, "isGoalkeeper": False},
]

teams = ["Alpha", "Beta"]
num_players = len(players_data)
num_teams = len(teams)
```

```python
strengths = [p['strength'] for p in players_data]
goalkeeper_indices = [i for i, p in enumerate(players_data) if p['isGoalkeeper']]
```

```python
total_strength = sum(strengths)
equal_dist = total_strength / num_teams
```

```python
# Create the optimization model
qp = QuadraticProgram("FootballTeamSelection")
```

```python
# 1. Define binary variables
for i in range(num_players):
    for t in range(num_teams):
        qp.binary_var(name=f'x_{i}_{t}')
```

The objective function in the code comes from the expansion of our O_i. Let X_i be the total strength of team i:

$$O_i = (X_i - ED)^2 = X_i^2 - 2 \cdot X_i \cdot ED + ED^2$$

- X_i^2 comes from the total team strength; in the expansion, each player i is multiplied by every player j on the same team (since it is the multiplication of a sum).
  - That is why we say this is the *quadratic* term, as it depends on two players at a time.
- $(-2) \cdot X_i \cdot ED$ is a coefficient that depends only on each player i once.
  - That is why we say this is the *linear* term, as it only varies with one player.
- ED^2, in turn, does not depend on any player.
  - Since its value is always the same, we call it the *constant* term.

```python
# 2. Define the objective function

linear_coeffs = {}
quadratic_coeffs = {}
constant = num_teams * (equal_dist ** 2)

for t in range(num_teams):
    for i in range(num_players):
        # Linear term:
        var_name = f'x_{i}_{t}'
        linear_coeffs[var_name] = -2 * strengths[i] * equal_dist

        for j in range(num_players):
            # Quadratic term:
            var1_name = f'x_{i}_{t}'
            var2_name = f'x_{j}_{t}'
            quadratic_coeffs[(var1_name, var2_name)] = strengths[i] * strengths[j]

qp.minimize(constant=constant, linear=linear_coeffs, quadratic=quadratic_coeffs)
```

```python
# 3. Add constraints

# Each player in exactly one team
for i in range(num_players):
    player_assignment_vars = {f'x_{i}_{t}': 1 for t in range(num_teams)}
    qp.linear_constraint(linear=player_assignment_vars,
        sense='==', rhs=1, name=f'player_{i}_assignment')

# At most one goalkeeper per team
for t in range(num_teams):
    goalkeeper_assignment_vars = {f'x_{i}_{t}': 1 for i in goalkeeper_indices}
    qp.linear_constraint(linear=goalkeeper_assignment_vars,
        sense='<=', rhs=1, name=f'team_{t}_goalkeeper')
```

```
print("Optimization Model (QuadraticProgram) defined.")
print(qp.prettyprint())
```

```
Optimization Model (QuadraticProgram) defined.
Problem name: FootballTeamSelection

Minimize
  25*x_0_0^2 + 60*x_0_0*x_1_0 + 80*x_0_0*x_2_0 + 40*x_0_0*x_3_0 + 60*x_0_0*x_4_0
  + 90*x_0_0*x_5_0 + 25*x_0_1^2 + 60*x_0_1*x_1_1 + 80*x_0_1*x_2_1
  + 40*x_0_1*x_3_1 + 60*x_0_1*x_4_1 + 90*x_0_1*x_5_1 + 36*x_1_0^2
  + 96*x_1_0*x_2_0 + 48*x_1_0*x_3_0 + 72*x_1_0*x_4_0 + 108*x_1_0*x_5_0
  + 36*x_1_1^2 + 96*x_1_1*x_2_1 + 48*x_1_1*x_3_1 + 72*x_1_1*x_4_1
  + 108*x_1_1*x_5_1 + 64*x_2_0^2 + 64*x_2_0*x_3_0 + 96*x_2_0*x_4_0
  + 144*x_2_0*x_5_0 + 64*x_2_1^2 + 64*x_2_1*x_3_1 + 96*x_2_1*x_4_1
  + 144*x_2_1*x_5_1 + 16*x_3_0^2 + 48*x_3_0*x_4_0 + 72*x_3_0*x_5_0 + 16*x_3_1^2
  + 48*x_3_1*x_4_1 + 72*x_3_1*x_5_1 + 36*x_4_0^2 + 108*x_4_0*x_5_0 + 36*x_4_1^2
  + 108*x_4_1*x_5_1 + 81*x_5_0^2 + 81*x_5_1^2 - 190*x_0_0 - 190*x_0_1
  - 228*x_1_0 - 228*x_1_1 - 304*x_2_0 - 304*x_2_1 - 152*x_3_0 - 152*x_3_1
  - 228*x_4_0 - 228*x_4_1 - 342*x_5_0 - 342*x_5_1 + 722

Subject to
  Linear constraints (8)
    x_0_0 + x_0_1 == 1  'player_0_assignment'
    x_1_0 + x_1_1 == 1  'player_1_assignment'
    x_2_0 + x_2_1 == 1  'player_2_assignment'
    x_3_0 + x_3_1 == 1  'player_3_assignment'
    x_4_0 + x_4_1 == 1  'player_4_assignment'
    x_5_0 + x_5_1 == 1  'player_5_assignment'
    x_1_0 + x_2_0 <= 1  'team_0_goalkeeper'
    x_1_1 + x_2_1 <= 1  'team_1_goalkeeper'
  Binary variables (12)
    x_0_0 x_0_1 x_1_0 x_1_1 x_2_0 x_2_1 x_3_0 x_3_1 x_4_0 x_4_1 x_5_0 x_5_1
```

```python
converter = QuadraticProgramToQubo()
qubo = converter.convert(qp)
hamiltonian, offset = qubo.to_ising()
num_qubits = hamiltonian.num_qubits
```

```python
print(f"The problem was mapped to a Hamiltonian of {num_qubits} qubits.")
```

```
The problem was mapped to a Hamiltonian of 12 qubits.
```

```python
print(hamiltonian)
```

```
SparsePauliOp(['IIIIIIIIIZII', 'IIIIIIIZIIII', 'IIIIIIIIZIII', 'IIIIIZIIIIII', 'IIIIIIIIIIZZ
              coeffs=[-1444.25+0.j, -1444.25+0.j, -1444.25+0.j, -1444.25+0.j,  8665.5 +0.j,
    15.  +0.j,    20.  +0.j,    10.  +0.j,    15.  +0.j,    22.5 +0.j,
    15.  +0.j,    20.  +0.j,    10.  +0.j,    15.  +0.j,    22.5 +0.j,
  8665.5 +0.j,  1468.25+0.j,    12.  +0.j,    18.  +0.j,    27.  +0.j,
  1468.25+0.j,    12.  +0.j,    18.  +0.j,    27.  +0.j,  8665.5 +0.j,
    16.  +0.j,    24.  +0.j,    36.  +0.j,    16.  +0.j,    24.  +0.j,
    36.  +0.j,  8665.5 +0.j,    12.  +0.j,    18.  +0.j,    12.  +0.j,
    18.  +0.j,  8665.5 +0.j,    27.  +0.j,    27.  +0.j,  8665.5 +0.j])
```

2 - VQE Preparation

Now that we have our Hamiltonian operator, we can start defining the VQE itself.

Since we'll use basic simulators, there's no need to worry about the transpiler or error mitigation. Therefore, let's start with the Ansatz.

Ansatz

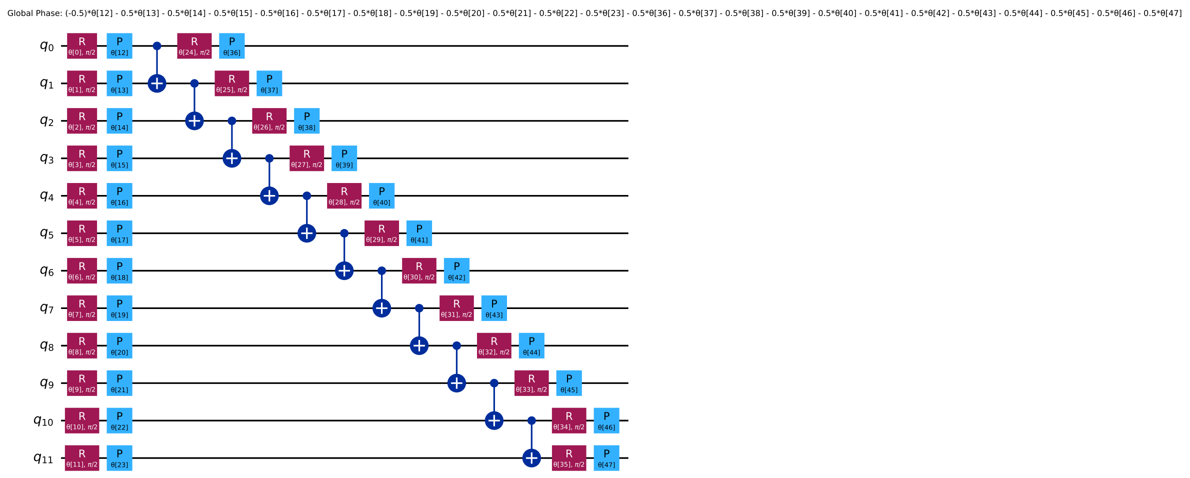
It is a parameterized quantum circuit. The VQE will try to find the parameters that prepare the state of lowest energy.

- `EfficientSU2` is a common choice, which tends to be good in many simulated cases.

Being a circuit, under the hood it implements logic gates and entanglement modes.

For educational purposes, we can check the internal circuit, but it is not strictly necessary.

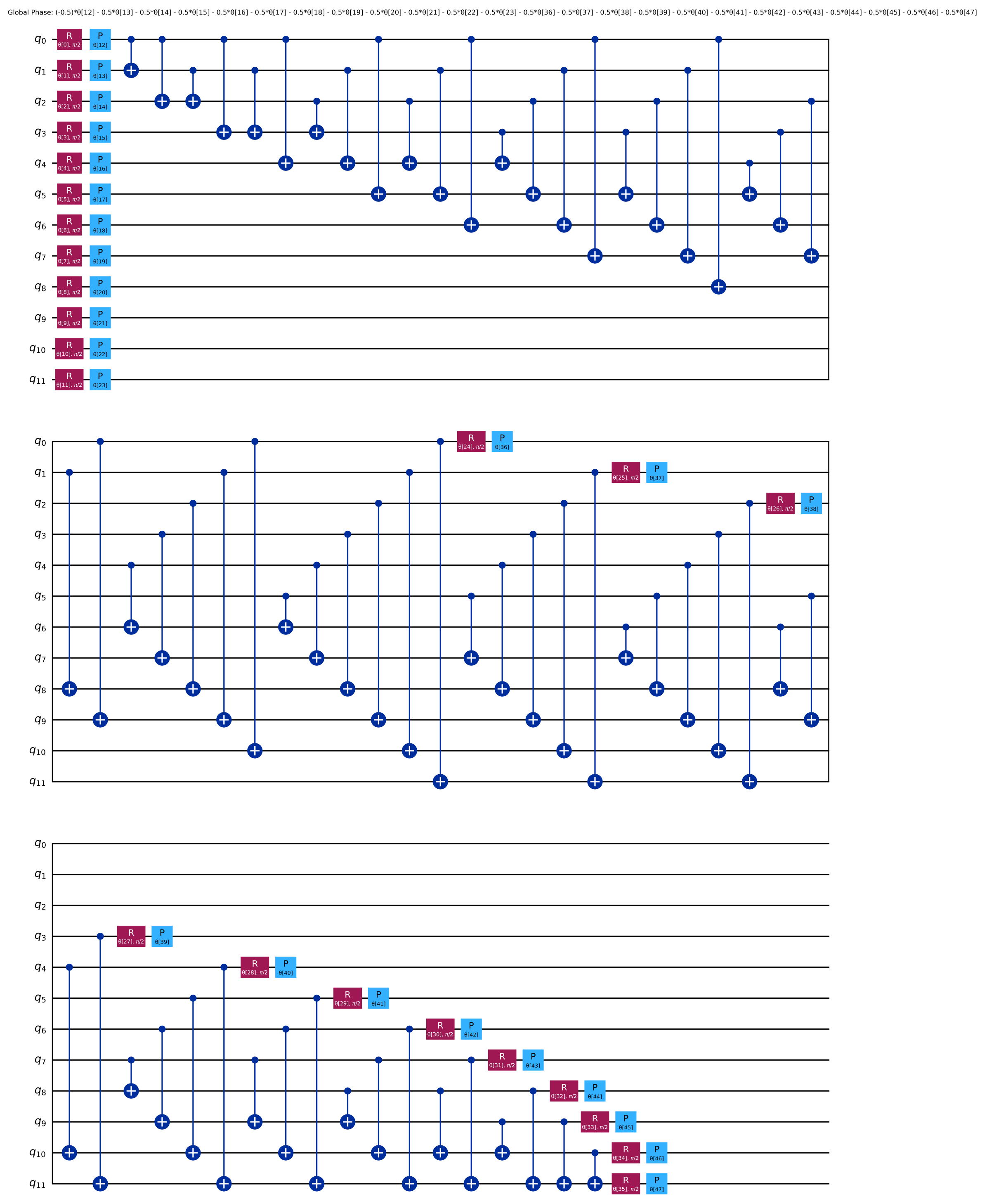
```
ansatz = efficient_su2(num_qubits, reps=1, entanglement='linear')

ansatz.decompose().draw("mpl")
```


Global Phase: (-0.5)*θ[12] - 0.5*θ[13] - 0.5*θ[14] - 0.5*θ[15] - 0.5*θ[16] - 0.5*θ[17] - 0.5*θ[18] - 0.5*θ[19] - 0.5*θ[20] - 0.5*θ[21] - 0.5*θ[22] - 0.5*θ[23] - 0.5*θ[36] - 0.5*θ[37] - 0.5*θ[38] - 0.5*θ[39] - 0.5*θ[40] - 0.5*θ[41] - 0.5*θ[42] - 0.5*θ[43] - 0.5*θ[44] - 0.5*θ[45] - 0.5*θ[46] - 0.5*θ[47]
q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11

We can alter parameters, such as repetition (creates more layers of rotations and CNOT gates) and entanglement type.

For example, check the circuit with full entanglement:

```
efficient_su2(num_qubits, reps=1, entanglement='full').decompose().draw("mpl")
```


Global Phase: (-0.5)*θ[12] - 0.5*θ[13] - 0.5*θ[14] - 0.5*θ[15] - 0.5*θ[16] - 0.5*θ[17] - 0.5*θ[18] - 0.5*θ[19] - 0.5*θ[20] - 0.5*θ[21] - 0.5*θ[22] - 0.5*θ[23] - 0.5*θ[36] - 0.5*θ[37] - 0.5*θ[38] - 0.5*θ[39] - 0.5*θ[40] - 0.5*θ[41] - 0.5*θ[42] - 0.5*θ[43] - 0.5*θ[44] - 0.5*θ[45] - 0.5*θ[46] - 0.5*θ[47]
q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
R
P

Although it provides stronger entanglement between qubits, the number of applied logic gates is much higher.

This can cause slowness in the algorithm execution, since the circuit is more complex and dense Even worse, it exponentially increases the probability of errors if executing the circuit on real hardware.

For our example, let’s stick with linear entanglement, but using two repetitions.

```
ansatz = efficient_su2(num_qubits, reps=2, entanglement='linear')
```

Classical Optimizer

It is the classical algorithm that will adjust the Ansatz parameters.

- `COBYLA` is a good option for our case, since it does not require gradients, making it suitable for simulators.

Furthermore, we need to define an estimator, which will execute the circuit and measure the results.

- For our case in the simulator, we will use `AerEstimator`.

```
estimator = AerEstimator(
    # An ideal simulator without shot noise
    run_options={"shots": None, "seed": 42},
    # Allows exact calculation of expectation value, without sampling
    approximation=True
)
```

Cost Function

Here, we will define the cost function in code that will be minimized by the algorithm.

We'll add some logs showing how the energy changes at each iteration.

```
# List to store cost history for later plotting
cost_history = []
iteration_count = 0
```

```
# Cost function to be minimized (calculates energy)
def cost_func(params: Sequence) -> float:
    """Calculates the expectation value (energy) and logs progress."""
    global iteration_count

    # Each pub is a tuple of (circuit, observables, parameter_values)
    result = estimator.run(
        circuits=[ansatz],
        observables=[hamiltonian],
        parameter_values=[params]
    ).result()

    energy = result.values[0]

    cost_history.append(energy)
    iteration_count += 1
    # print(f"Iteration: {iteration_count} | Cost (Energy): {energy:.5f}", end="\r", flush=T

    return energy
```

Algorithm Execution

Here, we define the initial parameters for the algorithm.

Then, we construct the minimization function using `scipy`, passing the optimizer we chose earlier, and run the algorithm.

```python
# Random initial parameters
num_params = ansatz.num_parameters
np.random.seed(42) # Setting seed for reproducibility
initial_params = 2 * np.pi * np.random.random(num_params)
```

```python
print("Starting optimization with scipy.minimize...")

# Optimization using COBYLA method via SciPy
res = minimize(
    fun=cost_func,
    x0=initial_params,
    method="COBYLA",
    options={"maxiter": 300},
)
print("\nOptimization completed.")
```

```
Starting optimization with scipy.minimize...

Optimization completed.
```

```python
optimal_params = res.x

# The function value returned by `minimize` is the Hamiltonian value
min_hamiltonian_value = res.fun
# The actual energy of the original problem includes the conversion offset to Ising
min_energy = min_hamiltonian_value + offset
```

```python
print(f"Minimum Hamiltonian value (eigenvalue): {min_hamiltonian_value:.5f}")
print(f"Conversion offset: {offset:.5f}")
print(f"Final minimum solution energy (objective value): {min_energy:.5f}")
```

```
Minimum Hamiltonian value (eigenvalue): -43096.23013
Conversion offset: 55010.50000
Final minimum solution energy (objective value): 11914.26987
```

3 - Post-Processing and Analysis

Finally, we need to post-process the VQE results to make them interpretable.

First, let's run the circuit one last time, using the parameters optimized by the VQE:

```
# 1. Build the optimal circuit with the found parameters
optimal_circuit = ansatz.assign_parameters(optimal_params)
optimal_circuit.measure_all()
```

```
# 2. Sample the final state to find the most probable bitstring
sampler = AerSampler()
job = sampler.run([optimal_circuit], shots=1024)
result_sampler = job.result()
counts = result_sampler.quasi_dists[0].binary_probabilities()
```

```
solution_bitstring = max(counts, key=counts.get)
```

Now that we have our final result, let's interpret it as the result of our problem.

The first step is to translate the obtained bitstring, considering the differences between Qiskit's convention and the usual one.

```
# 3. Translate the bitstring back to the problem solution

# The bitstring is converted to an array of integers and reversed (little-endian).
qubo_solution_vector = np.array([int(s) for s in solution_bitstring[::-1]])
solution_vars_list = converter.interpret(qubo_solution_vector)
```

```
# Create a dictionary to facilitate variable lookup by name
solution_vars_dict = {var.name: val for var, val in zip(qp.variables, solution_vars_list)}
```

Then, we can create a helper function to show the results:

```
# 4. Display the solution in a readable way
def display_solution(solution_dict, players_data, teams):
    team_A = {"name": teams[0], "players": [], "strength": 0}
    team_B = {"name": teams[1], "players": [], "strength": 0}

    for i, player in enumerate(players_data):
        if solution_dict.get(f'x_{i}_0', 0) > 0.5:
            team_A["players"].append(player['name'])
            team_A["strength"] += player['strength']
        elif solution_dict.get(f'x_{i}_1', 0) > 0.5:
            team_B["players"].append(player['name'])
            team_B["strength"] += player['strength']

    return team_A, team_B
```

With this, we may print our results:

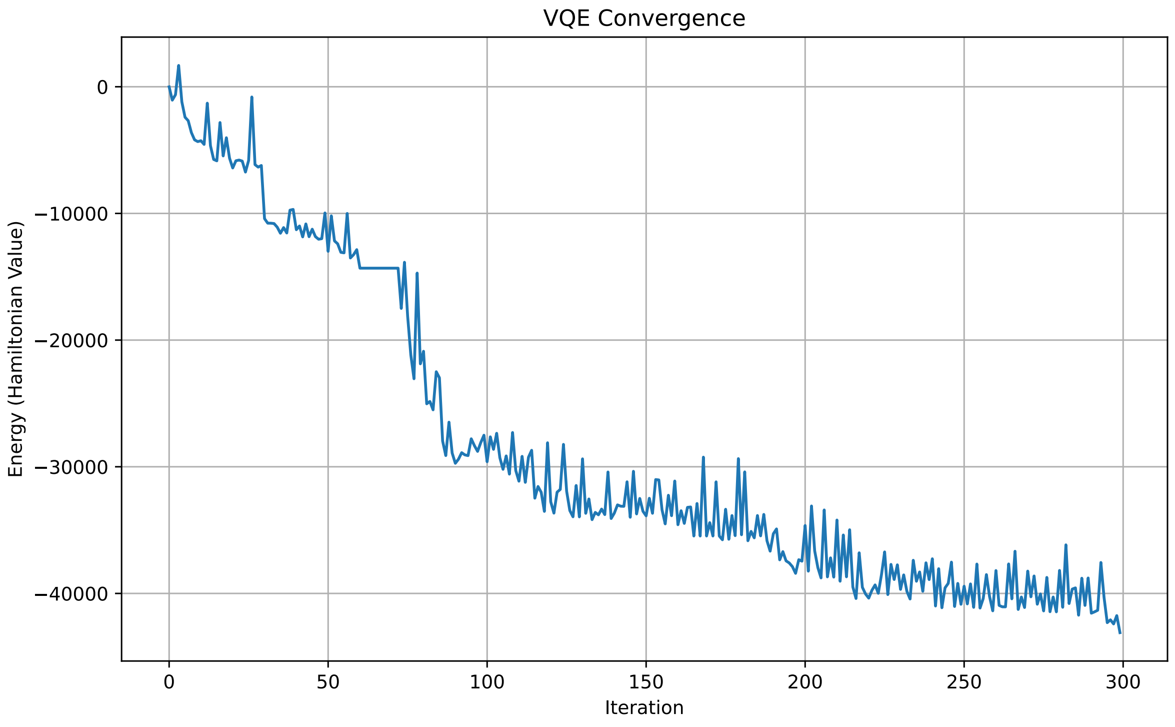
```
team_A, team_B = display_solution(solution_vars_dict, players_data, teams)
```

```
print("\n--- Final Team Composition ---")
print(f"Solution bitstring (little-endian): {solution_bitstring}")
print(f"Team {team_A['name']}:")
print(f"  Players: {', '.join(team_A['players'])}")
print(f"  Total Strength: {team_A['strength']}")
print("-" * 20)
print(f"Team {team_B['name']}:")
print(f"  Players: {', '.join(team_B['players'])}")
print(f"  Total Strength: {team_B['strength']}")
```

```
--- Final Team Composition ---
Solution bitstring (little-endian): 101001100101
Team Alpha:
  Players: Alice, Bob, Danny
  Total Strength: 15
--------------------
Team Beta:
  Players: Claire, Eve, Frank
  Total Strength: 23
```

Furthermore, since we saved the results of the intermediate iterations, we can examine the algorithm's behavior over time:

```
# Visualize energy convergence
plt.figure(figsize=(10, 6))
plt.plot(cost_history)
plt.xlabel("Iteration")
plt.ylabel("Energy (Hamiltonian Value)")
plt.title("VQE Convergence")
plt.grid(True)
plt.show()
```



Resources

- Example adapted from *The QuBlog*: https://www.thequblog.com/posts/how-to-solve-optimization-problems-with-aqua-qio-3/

- VQE Course (*IBM Quantum Learning*): https://quantum.cloud.ibm.com/learning/en/courses/variational-algorithm-design
- Qiskit Documentation: https://quantum.cloud.ibm.com/docs/en
- Qiskit Aer Documentation: https://qiskit.github.io/qiskit-aer/apidocs/aer.html
- Qiskit Optimization Documentation: https://qiskit-community.github.io/qiskit-optimization/apidocs/qiskit_optimization.html
- D-Wave Documentation (Quadratic Model Construction): https://docs.dwavequantum.com/en/latest/industrial_optimization/model_construction_qm.html#opt-model-construction-qm

Copyright and Licensing Note