May 2, 20242024

ITEC 4305 M Final Project Assignment

217882986 – ALI RIZA NAZLI

Part I: How QUBO can be used to accomplish feature selection

In recent times, the Quadratic Unconstrained Binary Optimization (QUBO) model has risen to prominence due to its versatility in solving a wide range of optimization problems. Originating from the Ising problem in physics, the QUBO model has found applications in quantum annealing within the realm of quantum computing and has also captured interest in neuromorphic computing.

This model lies at the core of experimentation conducted with advanced computing technologies, such as quantum computers developed by D-Wave Systems and neuromorphic computers pioneered by IBM. These connections have sparked exploration and initiatives by major players like Google, Amazon, and Lockheed Martin in the business arena, while research institutions like Los Alamos National Laboratory, Oak Ridge National Laboratory, Lawrence Livermore National Laboratory, and NASA's Ames Research Center have delved into its potential in the public sector.

Both classical and quantum computing communities are actively accumulating computational expertise, showcasing not only the promise of the QUBO model but also its efficacy as an alternative to conventional modeling and problem-solving approaches.

Feature selection using Quadratic Unconstrained Binary Optimization (QUBO) involves translating the problem into a binary optimization task. Here's a general outline of how QUBO can be employed for feature selection:

Data Representation: Organize the input data into a matrix where each row represents a data point, and each column corresponds to a feature.

Response Data: Represent the response data as a vector, where each entry corresponds to the response for a specific data point.

QUBO Formulation: Design a quadratic objective function that captures the relevance of each feature to the response variable. This often includes pairwise feature interactions.

Optimization: Utilize an optimization algorithm or solver to find the optimal combination of binary variables that maximizes the objective function. Binary variables represent the presence or absence of features in the selected subset.

Feature Selection: Based on the optimized QUBO model, select features that have a binary value of 1, indicating their importance for predicting the response variable.

QUBO provides a versatile framework for feature selection by enabling the exploration of various objective functions that balance relevance, redundancy, and computational complexity. This flexibility allows researchers to tailor the feature selection process to their specific needs and data characteristics.

Part II: Identify classical feature selection strategies that can be translated into a QUBO format.

Translating classical feature selection methods into the Quadratic Unconstrained Binary Optimization (QUBO) format involves leveraging quantum computing techniques to efficiently identify the most effective subset of features for models. This process begins by defining binary variables representing feature selection and constructing an objective function that aligns with the goals of the classical methods. Multiple traditional feature selection techniques can be reformulated as Quadratic Unconstrained Binary Optimization (QUBO) problems. Some examples include correlation-based feature selection, forward selection, backward elimination, L1-regularized regression (Lasso), and Minimum Redundancy Maximum Relevance (mRMR).

To convert these strategies into QUBO problems, we need to design an appropriate objective function. For instance, in correlation-based feature selection, the function should reward features highly correlated with the target variable while penalizing those with high mutual correlations. Similarly, in forward selection, the function should penalize error increase as we add more features, and binary variables can indicate whether a feature is included or not.

Lasso regression can also be expressed as a QUBO problem by using binary variables to denote non-zero coefficients, effectively selecting the corresponding features. Lastly, in mRMR, the objective function should encourage high relevance to the target variable while discouraging redundancy among features, with binary variables representing feature selection.

For instance, in the Mutual Information-Based Feature Selection method, which focuses on selecting features with strong associations to the target variable, QUBO assigns binary values to features, indicating their selection status, and sets up the problem to maximize the mutual information between the selected features and the target variable.

Similarly, in Correlation-Based Feature Selection, where features with high correlation to the target variable and low correlation among themselves are chosen, QUBO represents features as binary variables and aims to maximize the correlation between the selected features and the target while minimizing the correlation between selected features.

In R-squared-Based Feature Selection, the method aims to maximize the R-squared value to measure how well-selected features explain variance in the target variable. In the QUBO format, an objective function is constructed to maximize the R-squared value, indicating how much variance in the target variable is explained by the selected features.

Lastly, P-value-Based Feature Selection considers features with low p-values as significant predictors of the target variable. In QUBO, binary variables represent feature selection, and the problem is set up to minimize p-values of selected features, ensuring their statistical significance.

By reformulating these classical feature selection methods in the QUBO format, we can effectively adapt and solve feature selection problems using quantum computing techniques.

Part III Provide a basic overview of the quantum aspect.

Quantum computing harnesses principles from quantum mechanics, a branch of physics that describes how particles behave at the smallest scales. At the heart of quantum computing are units of information called qubits, which are analogous to classical bits but with some fascinating differences.

Qubits and Superposition: Classical bits are like switches that can be either on (1) or off (0). In contrast, qubits can exist in multiple states simultaneously due to a phenomenon called superposition. This means a qubit can be both 0 and 1 at the same time, allowing quantum computers to perform many calculations in parallel.

Entanglement: Another crucial concept is entanglement, where qubits become correlated with each other in such a way that the state of one qubit instantly influences the state of another, even if they are far apart. This property enables quantum computers to process information more efficiently by exploiting these correlations.

Quantum Gates and Operations: Quantum gates are operations that manipulate qubits to perform computations. These gates are represented by matrices and can perform complex transformations on qubits. For example, the Hadamard gate puts qubits into a superposition of states, while the CNOT gate creates entanglement between qubits.

Measurement and Probabilities: When a quantum system is measured, its "collapses" into one of its possible states, with the probability of each outcome determined by the quantum state's amplitudes. This introduces randomness into quantum computations, and repeated measurements can yield different results.

Quantum Algorithms: Quantum algorithms, such as Shor's algorithm and Grover's algorithm, leverage these quantum phenomena to solve certain types of problems more efficiently than classical algorithms. For example, Shor's algorithm can factor large numbers exponentially faster than classical algorithms, posing a potential threat to modern cryptographic systems.

Overall, quantum computing offers the potential for exponential speedups in solving certain types of problems by leveraging the principles of quantum mechanics. While still in its early stages, quantum computing holds promise for revolutionizing fields such as cryptography, optimization, and material science.

To incorporate QUBO (Quadratic Unconstrained Binary Optimization) into feature selection methodologies like mutual information and others, we can follow these general steps:

Encoding Features: Represent features as binary variables, where a value of 1 indicates that the feature is selected, and 0 means it is not.

Formulating Objective Function: Construct a QUBO objective function that captures the desired criteria. For mutual information, this function should reward high mutual information between features and the target variable, while penalizing redundancy among features.

Penalty Terms: Incorporate penalty terms to enforce constraints on the number of features to select or other problem-specific requirements.

Optimization: Utilize a QUBO solver, such as quantum annealing or classical optimization techniques, to find the optimal set of binary variables that maximizes the objective function.

Selecting Features: Based on the optimized QUBO model, select features corresponding to binary variables with a value of 1.

For other feature selection techniques, the QUBO objective function can be tailored to capture their specific criteria, such as correlation-based feature selection or R-squared-based feature selection. The flexibility of QUBO allows researchers to experiment with various problem formulations and objective functions to best suit their feature selection needs and data characteristics.

Part IV Reproduction of related works and studies

In this part, I read all the papers and another documents I can find on the internet environment. I tried to reproduce the results of dataset which are used in the provided articles however, it was challenging to handle to create algorithms on them. Then, I found another basic and simple problem to use QUBO format. I used QUBO format on combinatorial optimization problems. Combinatorial optimization problems involve finding the best way to arrange a set of objects or values to achieve a specific goal. I used the Knapsack Problem for using QUBO. I used opanQUAO example for reproduce result.

Let's assume we have ball, computer, camera, books, guitar. We would like to bring all of them with us. However, we do not have enough size on our backpack. We can create two different list. First one for the importance value of the items and the second one for the weight of the items. Also, we have a capacity at our backpack.

```
import numpy as np


def sum_weight(bitstring, items_weight):
  weight = 0
  for n, i in enumerate(items_weight):
    if bitstring[n] == "1":
      weight += i
```

```
    return weight



def sum_values(bitstring, items_value):
    value = 0
    for n, i in enumerate(items_value):
        if bitstring[n] == "1":
            value += i
    return value


items = list(items_values.keys())
n_items = len(items)
combinations = {}
max_value = 0
for case_i in range(2**n_items):  # all possible options
    combinations[case_i] = {}
    bitstring = np.binary_repr(
        case_i, n_items
    )  # bitstring representation of a possible combination, e.g, "01100" in our problem means bringing (-💻📸--)
    combinations[case_i]["items"] = [items[n] for n, i in enumerate(bitstring) if i == "1"]
    combinations[case_i]["value"] = sum_values(bitstring, values_list)
    combinations[case_i]["weight"] = sum_values(bitstring, weights_list)
    # save the information of the optimal solution (the one that maximizes the value while respecting the maximum weight)
    if (
        combinations[case_i]["value"] > max_value
        and combinations[case_i]["weight"] <= maximum_weight
```

```
    ):
        max_value = combinations[case_i]["value"]
        optimal_solution = {
            "items": combinations[case_i]["items"],
            "value": combinations[case_i]["value"],
            "weight": combinations[case_i]["weight"],
        }


print(
    f"The best combination is {optimal_solution['items']} with a total value:
{optimal_solution['value']} and total weight {optimal_solution['weight']} "
)
```

The best combination is ['⚽', '💻', '🎸'] with a total value: 71 and total weight 19

If we have 50 or 100 items, the results are going to be taken more time. We need to create mathematical representation of our problem. Then we can convert our problem to QUBO representation using a upper triangular matrix.

```
Q = -np.diag(list(items_values.values()))  # Matrix Q for the problem.
x_opt = np.array(
    [[1 if i in optimal_solution["items"] else 0] for i in items_values.keys()]
)  # Optimal solution.
opt_str = "".join(str(i[0]) for i in x_opt)
min_cost = (x_opt.T @ Q @ x_opt)[0, 0]  # using Equation created for our problem
print(f"Q={Q}")
print(f"The minimum cost is  {min_cost}")
```

However, solely relying on this function isn't sufficient to resolve the problem. We also have to consider the weight constraint. This is where a crucial step comes into play: integrating our objective function with this inequality constraint. One common approach is to incorporate the constraint as a penalty term within the objective function. Ideally, this penalty term should remain zero when the total weight of the items is less than or equal to 26 but increase significantly otherwise. To ensure the penalty term remains zero within the constraint's validity range, it's common practice to introduce slack variables. There's an alternative method known as unbalanced penalization, which has shown superior performance.

```
N = round(np.ceil(np.log2(maximum_weight)))  # number of slack variables

weights = list(items_weight.values()) + [2**k for k in range(N)]


QT = np.pad(Q, ((0, N), (0, N)))  # adding the extra slack variables at the end of the Q matrix

n_qubits = len(QT)

lambd = 2  # We choose a lambda parameter enough large for the constraint to always be fulfilled

# Adding the terms for the penalty term

for i in range(len(QT)):
   QT[i, i] += lambd * weights[i] * (weights[i] - 2 * maximum_weight)  # Eq. 10
   for j in range(i + 1, len(QT)):
      QT[i, j] += 2 * lambd * weights[i] * weights[j]  # Eq. 9

offset = lambd * maximum_weight**2

print(f"Q={QT}")

# optimal string slack string

slack_string = np.binary_repr(maximum_weight - optimal_solution["weight"], N)[::-1]

x_opt_slack = np.concatenate(
   (x_opt, np.array([[int(i)] for i in slack_string]))
)  # combining the optimal string and slack string

opt_str_slack = "".join(str(i[0]) for i in x_opt_slack)
```

```
cost = (x_opt_slack.T @ QT @ x_opt_slack)[0, 0] + offset  # Optimal cost using equation 3

print(f"Cost:{cost}")


# At this point, we have encoded the problem in a format that we can use to solve it on quantum

# computers. Now it only remains to solve it using quantum algorithms!
```

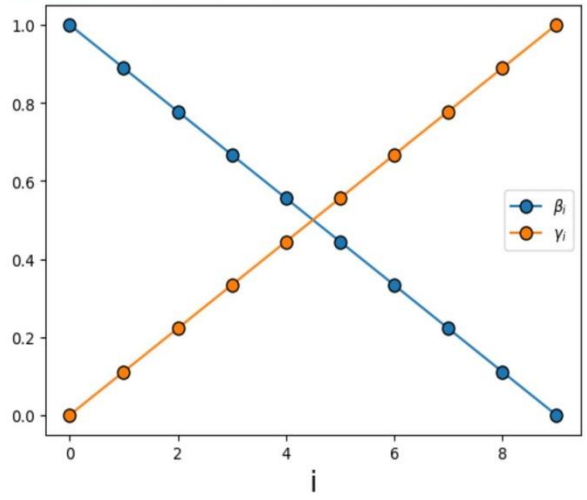
```
Q=[[ -302   132   168   228    60    12    24    48    96   192]
 [    0  -949   616   836   220    44    88   176   352   704]
 [    0     0 -1074  1064   280    56   112   224   448   896]
 [    0     0     0 -1259   380    76   152   304   608  1216]
 [    0     0     0     0  -486    20    40    80   160   320]
 [    0     0     0     0     0  -102     8    16    32    64]
 [    0     0     0     0     0     0  -200    32    64   128]
 [    0     0     0     0     0     0     0  -384   128   256]
 [    0     0     0     0     0     0     0     0  -704   512]
 [    0     0     0     0     0     0     0     0     0 -1152]]
Cost:-71
```

```
# ----------------------------   QAOA circuit ----------------------------------

from collections import defaultdict

import pennylane as qml


shots = 5000  # Number of samples used

dev = qml.device("default.qubit", shots=shots)




@qml.qnode(dev)

def qaoa_circuit(gammas, betas, h, J, num_qubits):

   wmax = max(

      np.max(np.abs(list(h.values()))), np.max(np.abs(list(h.values())))

   )  # Normalizing the Hamiltonian is a good idea

   p = len(gammas)
```

```
    # Apply the initial layer of Hadamard gates to all qubits
    for i in range(num_qubits):
        qml.Hadamard(wires=i)
    # repeat p layers the circuit shown in Fig. 1
    for layer in range(p):
        # ---------- COST HAMILTONIAN ----------
        for ki, v in h.items():  # single-qubit terms
            qml.RZ(2 * gammas[layer] * v / wmax, wires=ki[0])
        for kij, vij in J.items():  # two-qubit terms
            qml.CNOT(wires=[kij[0], kij[1]])
            qml.RZ(2 * gammas[layer] * vij / wmax, wires=kij[1])
            qml.CNOT(wires=[kij[0], kij[1]])
        # ---------- MIXER HAMILTONIAN ----------
        for i in range(num_qubits):
            qml.RX(-2 * betas[layer], wires=i)
    return qml.sample()



def samples_dict(samples, n_items):
    """Just sorting the outputs in a dictionary"""
    results = defaultdict(int)
    for sample in samples:
        results["".join(str(i) for i in sample)[:n_items]] += 1
    return results
import matplotlib.pyplot as plt
# Annealing schedule for QAOA
betas = np.linspace(0, 1, 10)[::-1]  # Parameters for the mixer Hamiltonian
```

```
gammas = np.linspace(0, 1, 10)  # Parameters for the cost Hamiltonian (Our Knapsack problem)


fig, ax = plt.subplots()

ax.plot(betas, label=r"$\beta_i$", marker="o", markersize=8, markeredgecolor="black")

ax.plot(gammas, label=r"$\gamma_i$", marker="o", markersize=8, markeredgecolor="black")

ax.set_xlabel("i", fontsize=18)

ax.legend()

fig.show()
```

```
C:\Users\Butterfly\AppData\Local\Temp\ipykernel_10984\3297791593.py:11: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```




```
def from_Q_to_Ising(Q, offset):
    """Convert the matrix Q of Eq.3 into Eq.13 elements J and h"""
    n_qubits = len(Q)  # Get the number of qubits (variables) in the QUBO matrix
    # Create default dictionaries to store h and pairwise interactions J
    h = defaultdict(int)
    J = defaultdict(int)
```

```
    # Loop over each qubit (variable) in the QUBO matrix
    for i in range(n_qubits):
        # Update the magnetic field for qubit i based on its diagonal element in Q
        h[(i,)] -= Q[i, i] / 2
        # Update the offset based on the diagonal element in Q
        offset += Q[i, i] / 2
        # Loop over other qubits (variables) to calculate pairwise interactions
        for j in range(i + 1, n_qubits):
            # Update the pairwise interaction strength (J) between qubits i and j
            J[(i, j)] += Q[i, j] / 4
            # Update the magnetic fields for qubits i and j based on their interactions in Q
            h[(i,)] -= Q[i, j] / 4
            h[(j,)] -= Q[i, j] / 4
            # Update the offset based on the interaction strength between qubits i and j
            offset += Q[i, j] / 4
    # Return the magnetic fields, pairwise interactions, and the updated offset
    return h, J, offset



def energy_Ising(z, h, J, offset):
    """
    Calculate the energy of an Ising model given spin configurations.

    Parameters:
    - z: A dictionary representing the spin configurations for each qubit.
    - h: A dictionary representing the magnetic fields for each qubit.
    - J: A dictionary representing the pairwise interactions between qubits.
```

```
    - offset: An offset value.

    Returns:
    - energy: The total energy of the Ising model.
    """
    if isinstance(z, str):
        z = [(1 if int(i) == 0 else -1) for i in z]
    # Initialize the energy with the offset term
    energy = offset
    # Loop over the magnetic fields (h) for each qubit and update the energy
    for k, v in h.items():
        energy += v * z[k[0]]
    # Loop over the pairwise interactions (J) between qubits and update the energy
    for k, v in J.items():
        energy += v * z[k[0]] * z[k[1]]
    # Return the total energy of the Ising model
    return energy


# Our previous example should give us the same result
z_exp = [
    (1 if i == 0 else -1) for i in x_opt_slack
]  # Converting the optimal solution from (0,1) to (1, -1)
h, J, zoffset = from_Q_to_Ising(QT, offset)  # Eq.13 for our problem
energy = energy_Ising(
    z_exp, h, J, zoffset
)  # Caluclating the energy (Should be the same that for the QUBO)
```

```
print(f"Minimum energy:{energy}")


samples_slack = samples_dict(qaoa_circuit(gammas, betas, h, J, num_qubits=len(QT)),
n_qubits)
values_slack = {
  sum_values(sample_i, values_list): count
  for sample_i, count in samples_slack.items()
  if sum_weight(sample_i, weights_list) <= maximum_weight
}  # saving only the solutions that fulfill the constraint
print(
  f"The number of optimal solutions using slack variables is
{samples_slack[opt_str_slack]} out of {shots}"
)
```

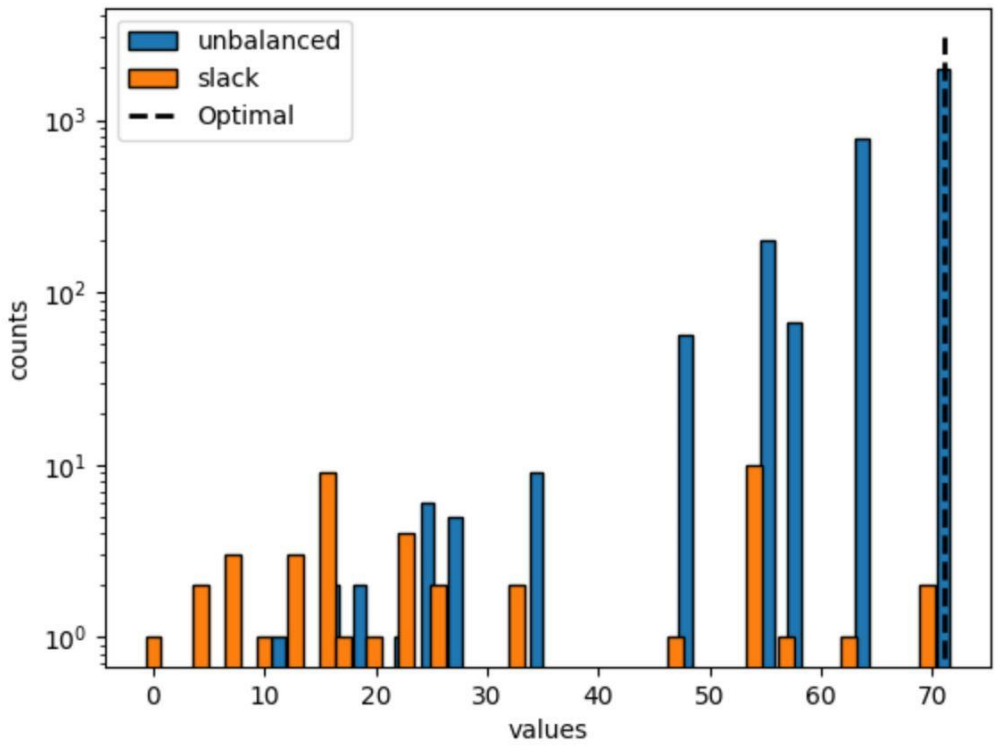
Minimum energy:-71.0

```
Unbalanced penalization (An alternative to slack variables)
from openqaoa.problems import FromDocplex2IsingModel
from docplex.mp.model import Model



def Knapsack(values, weights, maximum_weight):
  """Create a docplex model of the problem. (Docplex is a classical solver from IBM)"""
  n_items = len(values)
  mdl = Model()
  x = mdl.binary_var_list(range(n_items), name="x")
  cost = -mdl.sum(x[i] * values[i] for i in range(n_items))
  mdl.minimize(cost)
```

```
    mdl.add_constraint(mdl.sum(x[i] * weights[i] for i in range(n_items)) <=
maximum_weight)
    return mdl



# Docplex model, we need to convert our problem in this format to use the unbalanced
penalization approach
mdl = Knapsack(values_list, weights_list, maximum_weight)
lambda_1, lambda_2 = (
    0.96,
    0.0371,
)  # Parameters of the unbalanced penalization function (They are in the main paper)
ising_hamiltonian = FromDocplex2IsingModel(
    mdl,
    unbalanced_const=True,
    strength_ineq=[lambda_1, lambda_2],  # https://arxiv.org/abs/2211.13914
).ising_model


h_new = {
    tuple(i): w for i, w in zip(ising_hamiltonian.terms, ising_hamiltonian.weights) if len(i) ==
1
}
J_new = {
    tuple(i): w for i, w in zip(ising_hamiltonian.terms, ising_hamiltonian.weights) if len(i) ==
2
}


samples_unbalanced = samples_dict(
```

```
    qaoa_circuit(gammas, betas, h_new, J_new, num_qubits=n_items), n_items
)
values_unbalanced = {
    sum_values(sample_i, values_list): count
    for sample_i, count in samples_unbalanced.items()
    if sum_weight(sample_i, weights_list) <= maximum_weight
}  # saving only the solutions that fulfill the constraint


print(
    f"The number of solutions using unbalanced penalization is
{samples_unbalanced[opt_str]} out of {shots}"
)
fig, ax = plt.subplots()
ax.hist(
    values_unbalanced.keys(),
    weights=values_unbalanced.values(),
    bins=50,
    edgecolor="black",
    label="unbalanced",
    align="right",
)
ax.hist(
    values_slack.keys(),
    weights=values_slack.values(),
    bins=50,
    edgecolor="black",
    label="slack",
```

```
    align="left",
)
ax.vlines(-min_cost, 0, 3000, linestyle="--", color="black", label="Optimal", linewidth=2)
ax.set_yscale("log")
ax.legend()
ax.set_ylabel("counts")
ax.set_xlabel("values")
fig.show()
```

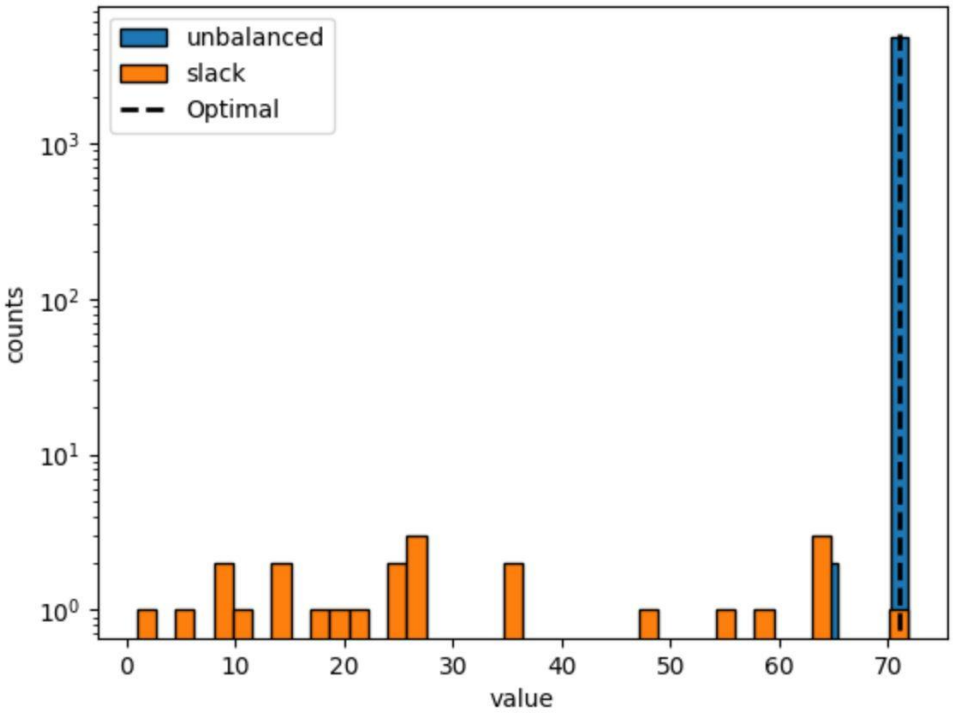


Quantum Annealing Solution

```
from dwave.system import DWaveSampler, EmbeddingComposite
from dwave.cloud import Client
import dimod
import pandas as pd


bqm = {}
# BQM - Binary Quadratic Model
# This creates the QUBO model of our Knapsack problem using the slack variables approach
# offset is the constant term in our QUBO formulation
# ----------- SLACK METHOD -----------
bqm["slack"] = dimod.BQM.from_qubo(QT, offset=lambd * maximum_weight**2)
bqm["slack"].relabel_variables({i: f"x_{i}" for i in range(bqm["slack"].num_variables)})
# -----------  UNBALANCED METHOD -----------
lagrange_multiplier = [0.96, 0.0371]  # Again values from the paper
bqm["unbalanced"] = dimod.BQM.from_qubo(Q)  # This adds the objective function to the model
bqm["unbalanced"].add_linear_inequality_constraint(
   [(n, i) for n, i in enumerate(weights_list)],  # This adds the constraint
   lagrange_multiplier,
   "unbalanced",
   ub=maximum_weight,
   penalization_method="unbalanced",
)
bqm["unbalanced"].relabel_variables({i: f"x_{i}" for i in range(bqm["unbalanced"].num_variables)})
```

```
# If you have an account you can execute the following code, otherwise read the file.
account = False
df = {}
if account:
  # Replace with your client information
  sampler = DWaveSampler(region="eu-central-1")
  sampler_qpu = EmbeddingComposite(sampler)
  for method in ["slack", "unbalanced"]:
    samples = sampler_qpu.sample(bqm[method], num_reads=5000)  # Executing on real hardware
    df[method] = (
      samples.to_pandas_dataframe().sort_values("energy").reset_index(drop=True)
    )  # Converting the sampling information and sort it by cost
    df[method].to_json(f"QUBO/dwave_results_{method}.json")  # save the results
else:
  df = {}
  for method in ["slack", "unbalanced"]:
    df[method] = pd.read_json(f"QUBO/dwave_results_{method}.json")
    # Loading the data from an execution on D-Wave Advantage



samples_dwave = {}
values = {}
for method in ["slack", "unbalanced"]:
  samples_dwave[method] = defaultdict(int)
  for i, row in df[method].iterrows():
    # Postprocessing the information
```

```
        sample_i = "".join(str(round(row[q])) for q in bqm[method].variables)
        samples_dwave[method][sample_i] += row["num_occurrences"]
    values[method] = {
        sum_values(sample_i, values_list): count
        for sample_i, count in samples_dwave[method].items()
        if sum_weight(sample_i, weights_list) <= maximum_weight
    }
fig, ax = plt.subplots()
bins = {"unbalanced": 5, "slack": 40}
for method in ["unbalanced", "slack"]:
    ax.hist(
        values[method].keys(),
        weights=values[method].values(),
        bins=bins[method],
        edgecolor="black",
        label=method,
        align="right",
    )
ax.vlines(-min_cost, 0, 5000, linestyle="--", color="black", label="Optimal", linewidth=2)
ax.set_yscale("log")
ax.legend()
ax.set_ylabel("counts")
ax.set_xlabel("value")
fig.show()
```


unbalanced
slack
Optimal
counts
10^3
10^2
10^1
10^0
0
10
20
30
40
50
60
70
value

References

https://openqaoa.entropicalabs.com/.

https://kth.diva-portal.org/smash/get/diva2:1799823/FULLTEXT01.pdf

https://arxiv.org/pdf/2104.04049

https://arxiv.org/pdf/2203.13261.pdf

https://arxiv.org/pdf/2211.02861.pdf

https://arxiv.org/pdf/2306.10591

https://arxiv.org/abs/2211.13914

https://github.com/qcpolimi/SIGIR22_QuantumFeatureSelection/tree/main/Letor

https://www.mdpi.com/1099-4300/23/8/970

https://www.analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/

https://www.mathworks.com/help/matlab/math/feature-selection-quantum-annealing.html

https://link.springer.com/article/10.1007/s42484-023-00099-z