ML2R Coding Nuggets
Hopfield Nets for Set Cover

Christian Bauckhage
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

ABSTRACT

We once again revisit the minimum set cover problem and show that the underlying integer linear program can be rewritten as a quadratic unconstrained binary optimization problem. This can then be cast as an energy minimization problem which we solve by running Hopfield nets. Using multiple restarts, our simple *NumPy* implementation consistently produces good results.

1 INTRODUCTION

Previously [1, 2], we looked at this basic[1] variant of the **set cover problem**: Given a set $\mathcal{X}$ of m elements x_i and a set $\mathcal{Y}$ of n subsets $\mathcal{Y}_j$ of $\mathcal{X}$ which are guaranteed to cover $\mathcal{X}$, determine a smallest subset $\mathcal{Z}$ of $\mathcal{Y}$ that covers $\mathcal{X}$.

To better grasp the nature of this problem, we considered a simple example where

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$$

$$\mathcal{Y} = \begin{Bmatrix} \mathcal{Y}_1 = \{ & x_2, & & & & & & & & \} \\ \mathcal{Y}_2 = \{ & x_1, & x_2, & x_3, & & & & & & \} \\ \mathcal{Y}_3 = \{ & & & & x_4, & x_5, & & & & x_{10} \} \\ \mathcal{Y}_4 = \{ & & & & & & & x_8, & x_9, & x_{10} \} \\ \mathcal{Y}_5 = \{ & & & & x_4, & x_5, & x_6, & x_7, & & \} \\ \mathcal{Y}_6 = \{ & x_1, & & x_3, & & x_5, & & x_7, & & x_9, & \} \end{Bmatrix}$$

Looking at this example, it is clear that the $\mathcal{Y}_j \in \mathcal{Y}$ cover $\mathcal{X}$, because

$$\bigcup_{\mathcal{Y}_j \in \mathcal{Y}} \mathcal{Y}_j = \mathcal{X}$$

We further realize that the following smaller subset $\mathcal{Z} = \{\mathcal{Y}_2, \mathcal{Y}_4, \mathcal{Y}_5\}$ of $\mathcal{Y}$ also covers $\mathcal{X}$, because

$$\bigcup_{\mathcal{Y}_j \in \mathcal{Z}} \mathcal{Y}_j = \mathcal{X}$$

Moreover, it is easy to verify that $\mathcal{Z}$ is indeed the single smallest subset of $\mathcal{Y}$ that covers $\mathcal{X}$ and therefore solves our problem.

While this example makes set covering look easy, we discussed that it is NP-hard in general [2]. This was further confirmed by our observation that minimum set cover is actually an integer linear programming problem in disguise [1].

To see this once again, recall that, if we are given a certain set $\mathcal{X} = \{x_1, \ldots, x_m\}$, we can represent any subset $\mathcal{Y}_j \subseteq \mathcal{X}$ in terms of a binary indicator vector $\boldsymbol{y}_j \in \{0,1\}^m$ where

$$\big[\boldsymbol{y}_j\big]_i = \begin{cases} 1 & \text{if } x_i \in \mathcal{Y}_j \\ 0 & \text{otherwise} \end{cases}$$

[1]There also exists the more general notion of *weighted* set cover problems.

We also recall that indicator vector representations extend to sets of subsets: If we are given a whole set $\mathcal{Y} = \{\mathcal{Y}_1, \ldots, \mathcal{Y}_n\}$ of subsets $\mathcal{Y}_j \subseteq \mathcal{X}$, we can represent this collection as a matrix $Y \in \{0,1\}^{m \times n}$ where $Y = \begin{bmatrix} \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_n \end{bmatrix}$.

For instance, if we represent the subsets $\mathcal{Y}_1, \ldots, \mathcal{Y}_6$ of our simple example using indicator vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_6$, gather them as the columns of a matrix Y, and look at the transpose of this matrix, we find

$$Y^\top = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

which is easily recognized as an abstract representation of set $\mathcal{Y}$ as presented above.

Finally, we recall that we may represent the given set $\mathcal{X}$ in terms of an indicator vector $\boldsymbol{x} = \mathbf{1}_m$ of all ones and any subset $\mathcal{Z} \subseteq \mathcal{Y}$ as as an indicator vector $\boldsymbol{z} \in \{0,1\}^n$. But this means that we may think of set covering as a linear algebra problem: If we could find a vector $\boldsymbol{z}$ such that the linear combination $\boldsymbol{x}(\boldsymbol{z}) = \boldsymbol{y}_1 \cdot z_1 + \ldots + \boldsymbol{y}_n \cdot z_n = Y\boldsymbol{z}$ obeys $\boldsymbol{x}(\boldsymbol{z}) \geq \mathbf{1}_m$, we would have found an indicator vector which identifies a collection $\mathcal{Z}$ of subsets $\mathcal{Y}_j$ of $\mathcal{Y}$ which cover $\mathcal{X}$.

Note, however, that the set cover problem does not simply ask for any cover of a given set but for a minimum cover of said set. Since we can express this objective by demanding that the sum of the entries of $\boldsymbol{z}$, i.e. the inner product $\mathbf{1}_n^\top \boldsymbol{z}$, is to be minimized, we arrive at the following formulation of the minimum set cover problem in terms of an **integer linear program (ILP)**

$$\begin{aligned} \boldsymbol{z}^* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\text{argmin}} \;\; & \mathbf{1}_n^\top \boldsymbol{z} \\ \text{s.t.} \;\; & Y\boldsymbol{z} \geq \mathbf{1}_m \end{aligned} \tag{1}$$

Previously, we solved this ILP using correspondingly adapted earlier code for approximative greedy set covering. Our strategy in this note will be fundamentally different.

Next, in section 2, we will rewrite the ILP in (1) as a **quadratic unconstrained binary optimization problem (QUBO)**. This will require some work but also allows us to treat minimum set cover as a Hopfield energy minimization problem. We therefore also review Hopfield nets for problem solving. In section 3, we then put them to work and present corresponding *NumPy* implementations. Readers who would like to experiment with our code snippets should be familiar with *NumPy* / *SciPy* [16] and need to

```
import numpy as np
import numpy.random as rnd
```

2 THEORY

Our primary goal in this section is to establish that the set cover problem can be cast as a QUBO of the following form

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^N}{\operatorname{argmin}} \; \boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} \tag{2}$$

where the matrix $\boldsymbol{Q} \in \mathbb{R}^{N \times N}$ and vector $\boldsymbol{q} \in \mathbb{R}^N$ contain problem specific parameters.

The fact that this can be done is not new. However, the frequently cited derivation of this result in [12] lacks details. We therefore get inspiration from the general recipes in [9], adapt them to the set cover problem, and present a step-by-step derivation starting from the ILP in (1).

Our secondary goal is to tie in the QUBO formulation of set cover with Hopfield nets for combinatorial problem solving [10, 11]. Here, our discussion will largely recap points we made in earlier notes (see, for instance, [3, 4, 6]).

2.1 Set Cover as a QUBO

Our first step in turning the ILP in (1) into a QUBO as in (2) will be to rewrite the inequality constraint

$$\boldsymbol{Y}\boldsymbol{z} \geq \boldsymbol{1}_m \tag{3}$$

as an equality constraint.

To this end, we introduce slack variables $o_1, \ldots o_m$, gather them in a vector $\boldsymbol{o} \in \mathbb{R}^m$, and consider

$$\boldsymbol{Y}\boldsymbol{z} - \boldsymbol{o} = \boldsymbol{1}_n \tag{4}$$

If we rearrange this equation, we obtain the following expression for the slack variables

$$\boldsymbol{o} = \boldsymbol{Y}\boldsymbol{z} - \boldsymbol{1}_m \tag{5}$$

Why do we call our slack variables o_i? Because they they allow us to cope with "overshooting". To see what we mean by this, we need to take a step back …

From the point of view of linear algebra with indicator vectors, finding a set cover is tantamount to finding linear coefficients $z_j \in \{0, 1\}$ such that the linear combination $\boldsymbol{x}(\boldsymbol{z}) = \boldsymbol{y}_1 \cdot z_1 + \ldots + \boldsymbol{y}_n \cdot z_n$ obeys $\boldsymbol{x}(\boldsymbol{z}) \geq \boldsymbol{1}_m$.

Since the $\boldsymbol{y}_j$ form the columns of matrix $\boldsymbol{Y}$, we usually express this goal as having to find a coefficient vector $\boldsymbol{z}$ such that $\boldsymbol{x}(\boldsymbol{z}) = \boldsymbol{Y}\boldsymbol{z} \geq \boldsymbol{1}_m$.

To better understand the rational behind this idea, we shall briefly motivate it w.r.t. the exemplary set cover problem from the introduction.

There, we are given a set of subsets $\mathcal{Y} = \{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_6\}$ which we represent in terms of an indicator matrix $\boldsymbol{Y} = [\boldsymbol{y}_1\, \boldsymbol{y}_2\, \cdots\, \boldsymbol{y}_6]$ which reads

$$\boldsymbol{Y} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

We already know that the collection $\mathcal{Z}_{245} = \{\mathcal{Y}_2, \mathcal{Y}_4, \mathcal{Y}_5\}$ optimally solves our problem; it can be represented in terms of the indicator vector $\boldsymbol{z}_{245} = [0\,1\,0\,1\,1\,0]^\top$.

Another, sub-optimal cover is given by $\mathcal{Z}_{1456} = \{\mathcal{Y}_1, \mathcal{Y}_4, \mathcal{Y}_5, \mathcal{Y}_6\}$ which can be represented by the indicator vector $\boldsymbol{z}_{1456} = [1\,0\,0\,1\,1\,1]^\top$

If we compute the expression $\boldsymbol{x}(\boldsymbol{z}_{245}) = \boldsymbol{Y}\boldsymbol{z}_{245}$, we find

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Since all the $m = 10$ entries of the vector $\boldsymbol{x}(\boldsymbol{z}_{245})$ are covered, i.e. are greater then 0, we have found a vector $\boldsymbol{z} = \boldsymbol{z}_{245}$ such that $\boldsymbol{Y}\boldsymbol{z} \geq \boldsymbol{1}_{10}$. In fact, in this particular case, we actually have equality $\boldsymbol{Y}\boldsymbol{z} = \boldsymbol{1}_{10}$.

If we compute the expression $\boldsymbol{x}(\boldsymbol{z}_{1456}) = \boldsymbol{Y}\boldsymbol{z}_{1456}$, we find

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 1 \\ 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

Again, all entries of $\boldsymbol{x}(\boldsymbol{z}_{1456})$ are covered so that have found another vector $\boldsymbol{z} = \boldsymbol{z}_{1456}$ such that $\boldsymbol{Y}\boldsymbol{z} \geq \boldsymbol{1}_{10}$. However, the 5th and 9th entry of $\boldsymbol{x}(\boldsymbol{z}_{1456})$ are covered twice. Speaking in terms of the sets $\mathcal{Y}_1$, $\mathcal{Y}_4$, $\mathcal{Y}_5$, and $\mathcal{Y}_6$, this is to say that they cover the given universe $\mathcal{X}$ but are not disjoint.

In our silly example this means that the cover $\mathcal{Z}_{1456} = \{\mathcal{Y}_1, \mathcal{Y}_4, \mathcal{Y}_5, \mathcal{Y}_6\}$ is sub-optimal. However, in general, it may well happen that a minimum set cover involves overlapping subsets.

From the point of view of integer linear programming for set cover, this means that we have to incorporate inequalities into our considerations. Fundamentally, this is because the union ($\cup$) of sets behaves differently than the addition (+) of indicator vectors. For instance, the union of $\mathcal{Y}_2 = \{x_1, x_2, x_3\}$ and $\mathcal{Y}_6 = \{x_1, x_3, x_5, x_7, x_9\}$ is given by

$$\mathcal{Y}_2 \cup \mathcal{Y}_6 = \{x_1, x_2, x_3, x_5, x_7, x_9\}$$

whereas the sum of the corresponding indicator vectors $\boldsymbol{y}_2 = [1\,1\,1\,0\,0\,0\,0\,0\,0\,0]^\top$ and $\boldsymbol{y}_6 = [1\,0\,1\,0\,1\,0\,1\,0\,1\,0]^\top$ amounts to

$$\boldsymbol{y}_2 + \boldsymbol{y}_6 = [1\,1\,2\,0\,1\,0\,1\,0\,1\,0]^\top$$

Given these observations it is clear why we say that an indicator vector $\boldsymbol{z}$ that represents a subset $\mathcal{Z} \subseteq \mathcal{Y}$ also represents a cover of $\mathcal{X}$ exactly if $\boldsymbol{x}(\boldsymbol{z}) = \boldsymbol{Y}\boldsymbol{z} \geq \boldsymbol{1}_m$. For those entries $[\boldsymbol{x}(\boldsymbol{z})]_i$ of $\boldsymbol{x}(\boldsymbol{z})$ where $[\boldsymbol{x}(\boldsymbol{z})]_i > 1$, we colloquially say that the linear algebraic solution "overshot" the target.

Our next step in turning the ILP in (1) into a QUBO as in (2) will be to treat the slack variables in $\boldsymbol{o}$ as additional latent or hidden variables. In fact, equation (5) suggests that $\boldsymbol{o}$ is not fixed but depends on the sought after indicator vector $\boldsymbol{z}$.

How small or large can the entries o_i of $\boldsymbol{o}$ possibly be? Well, if $\boldsymbol{z}$ solves the ILP in (1), then we have the following lower bound

$$o_{i,\min} = 0 \leq o_i = \sum_{j=1}^{n} Y_{ij}\, z_j - 1 \tag{6}$$

because at least one Y_{ij} must have a value of 1 and must have been selected into the set cover ($\exists\, j : z_j = 1$). By the same token, we have the following upper bound

$$o_i = \sum_{j=1}^{n} Y_{ij}\, z_j - 1 \leq \sum_{j=1}^{n} Y_{ij} - 1 = o_{i,\max} \tag{7}$$

because all Y_{ij} may have a value of 1 and may have been selected into the set cover ($\forall\, j : z_j = 1$)

Our next crucial observation is that the o_i must necessarily be integers. This is because $Y_{ij} \in \{0, 1\}$ and $z_j \in \{0, 1\}$ such that any sum of products of these binaries will be an elements of $\mathbb{Z}_+$. But

this is to say that each o_i can be written as a binary number or, more precisely, as a binary expansion. That is, letting

$$P_i = \max\Big\{0, \big\lfloor \log_2(o_{i,\max}) \big\rfloor\Big\} \tag{8}$$

we may write

$$o_i = \sum_{p=0}^{P_i} 2^p \, c_p \tag{9}$$

where the $c_p \in \{0,1\}$ are binary expansion coefficients. Even more, if we introduce binary coefficient vectors $\boldsymbol{c}_i \in \{0,1\}^{P_i+1}$ and "power of two" vectors

$$\boldsymbol{p}_i = \begin{bmatrix} 2^0 & 2^1 & \cdots & 2^{P_i} \end{bmatrix}^\top \tag{10}$$

we may write

$$o_i = \boldsymbol{p}_i^\top \boldsymbol{c}_i \tag{11}$$

Even better, if we define an overall "power of two" matrix and coefficient vector

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p}_1^\top & \boldsymbol{0}^\top & \cdots & \boldsymbol{0}^\top \\ \boldsymbol{0}^\top & \boldsymbol{p}_2^\top & \cdots & \boldsymbol{0}^\top \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{0}^\top & \boldsymbol{0}^\top & \cdots & \boldsymbol{p}_n^\top \end{bmatrix} \tag{12}$$

$$\boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \\ \vdots \\ \boldsymbol{c}_n \end{bmatrix} \tag{13}$$

we can express the whole vector $\boldsymbol{o}$ of slack variables o_i as

$$\boldsymbol{o} = \boldsymbol{P}\boldsymbol{c} \tag{14}$$

Note: the number of columns of $\boldsymbol{P}$ and the corresponding number of rows of $\boldsymbol{c}$ is a variable that depends on the specific set cover problem we are dealing with. It is given by

$$n' = \sum_{i=1}^{m} (P_i + 1) \tag{15}$$

Plugging our new expression for $\boldsymbol{o}$ into equation (4), we obtain

$$\boldsymbol{Y}\boldsymbol{z} - \boldsymbol{P}\boldsymbol{c} = \boldsymbol{1}_m \tag{16}$$

Next, we will further rewrite this equation to make it more compact. To this end, we horizontally stack matrices $\boldsymbol{Y}$ and $-\boldsymbol{P}$ and vertically stack vectors $\boldsymbol{z}$ and $\boldsymbol{c}$, i.e. we introduce

$$\hat{\boldsymbol{Y}} = \begin{bmatrix} \boldsymbol{Y} & -\boldsymbol{P} \end{bmatrix} \tag{17}$$

$$\hat{\boldsymbol{z}} = \begin{bmatrix} \boldsymbol{z} \\ \boldsymbol{c} \end{bmatrix} \tag{18}$$

because this allows us to write (16) as

$$\hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} = \boldsymbol{1}_m \tag{19}$$

As an important intermediate result our our efforts so far, we therefore have that inequality constrained ILP in (1) can also be written as an equality constrained ILP, namely

$$\begin{aligned} \hat{\boldsymbol{z}}^* = \underset{\hat{\boldsymbol{z}} \in \{0,1\}^N}{\operatorname{argmin}} \;\; & \boldsymbol{1}_N^\top \hat{\boldsymbol{z}} \\ \text{s.t.} \;\; & \hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} = \boldsymbol{1}_m \end{aligned} \tag{20}$$

where $N = n + n'$. Since $N \geq n$, the price we have to pay for this "simplification" is that we now must solve for N binary variables $\hat{z}_j^*$ rather than for n binary variables z_j^* as before. However, this will pay off because it is now easy to turn the ILP in (20) into a QUBO as in (2)

Continuing with our derivation, we next note the equivalence

$$\hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} = \boldsymbol{1}_m \quad \Leftrightarrow \quad \big\|\hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} - \boldsymbol{1}_m\big\|^2 = 0 \tag{21}$$

and the fact that we may write this squared Euclidean distance as

$$\big\|\hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} - \boldsymbol{1}_m\big\|^2 = \hat{\boldsymbol{z}}^\top \hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} - 2\,\boldsymbol{1}_m^\top \hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} + \boldsymbol{1}_m^\top \boldsymbol{1}_m \tag{22}$$

Since the inner product $\boldsymbol{1}_m^\top \boldsymbol{1}_m$ is a constant independent of $\hat{\boldsymbol{z}}$, we therefore have the following Lagrangian for the problem in (20)

$$\begin{aligned} L(\hat{\boldsymbol{z}}, \lambda) &= \boldsymbol{1}_N^\top \hat{\boldsymbol{z}} + \lambda\big[\hat{\boldsymbol{z}}^\top \hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}}\hat{\boldsymbol{z}} - 2\,\boldsymbol{1}_m^\top \hat{\boldsymbol{Y}}\hat{\boldsymbol{z}}\big] \\ &= \hat{\boldsymbol{z}}^\top \big[\lambda \hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}}\big]\hat{\boldsymbol{z}} + \big[\boldsymbol{1}_N^\top - 2\,\lambda\,\boldsymbol{1}_m^\top \hat{\boldsymbol{Y}}\big]\hat{\boldsymbol{z}} \\ &\equiv \hat{\boldsymbol{z}}^\top \boldsymbol{R}\,\hat{\boldsymbol{z}} + \boldsymbol{r}^\top \hat{\boldsymbol{z}} \end{aligned} \tag{23}$$

Here, λ is a Lagrange multiplier which we henceforth treat as a parameter that needs to be set manually.

Now, all of this is to say that the original ILP in (1) or its modified version in (20) can just as well be cast a QUBO over binary decision variables $\hat{\boldsymbol{z}} \in \{0,1\}^N$, namely

$$\hat{\boldsymbol{z}}^* = \underset{\hat{\boldsymbol{z}} \in \{0,1\}^N}{\operatorname{argmin}} \; \hat{\boldsymbol{z}}^\top \boldsymbol{R}\,\hat{\boldsymbol{z}} + \boldsymbol{r}^\top \hat{\boldsymbol{z}} \tag{24}$$

Finally, in order to turn this into a QUBO over bipolar decision variables $\boldsymbol{s} \in \{\pm 1\}^N$ such as in (2), we recall that binary and bipolar vectors are isomorphic in the sense that

$$\boldsymbol{s} = 2\,\hat{\boldsymbol{z}} - \boldsymbol{1}_N \;\; \Leftrightarrow \;\; \hat{\boldsymbol{z}} = \tfrac{1}{2}\big[\boldsymbol{s} + \boldsymbol{1}_N\big] \tag{25}$$

Hence, if we plug $\hat{\boldsymbol{z}} = \frac{1}{2}[\boldsymbol{s} + \boldsymbol{1}_N]$ into the objective function of our QUBO in (24), we obtain

$$\begin{aligned} \hat{\boldsymbol{z}}^\top \boldsymbol{R}\,\hat{\boldsymbol{z}} + \boldsymbol{r}^\top \hat{\boldsymbol{z}} &= \tfrac{1}{4}\big[\boldsymbol{s} + \boldsymbol{1}_N\big]^\top \boldsymbol{R}\big[\boldsymbol{s} + \boldsymbol{1}_N\big] + \tfrac{1}{2}\boldsymbol{r}^\top\big[\boldsymbol{s} + \boldsymbol{1}_N\big] \\ &= \tfrac{1}{4}\boldsymbol{s}^\top \boldsymbol{R}\,\boldsymbol{s} + \tfrac{1}{2}\big[\boldsymbol{R}\,\boldsymbol{1}_N + \boldsymbol{r}\big]^\top \boldsymbol{s} + const \\ &\equiv \boldsymbol{s}^\top \boldsymbol{Q}\,\boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} + const \end{aligned} \tag{26}$$

In other words, we find that set covering can also be understood as the problem of solving

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^N}{\operatorname{argmin}} \; \boldsymbol{s}^\top \boldsymbol{Q}\,\boldsymbol{s} + \boldsymbol{q}^\top \boldsymbol{s} \tag{27}$$

where

$$\boldsymbol{Q} = \tfrac{1}{4}\boldsymbol{R} \tag{28}$$

$$\boldsymbol{q} = \tfrac{1}{2}\big[\boldsymbol{R}\,\boldsymbol{1}_N + \boldsymbol{r}\big] \tag{29}$$

and

$$\boldsymbol{R} = \lambda\,\hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}} \tag{30}$$

$$\boldsymbol{r} = \boldsymbol{1}_N - 2\,\lambda\,\hat{\boldsymbol{Y}}^\top \boldsymbol{1}_m \tag{31}$$

That is it! We now have a QUBO formulation of the set cover problem and may run Hopfield nets to try to solve it. If we do this, we will obtain a bipolar vector $\boldsymbol{s}^* \in \{-1,+1\}^N$ which we turn into a binary vector $\hat{\boldsymbol{z}}^* = \frac{1}{2}\big[\boldsymbol{s} + \boldsymbol{1}_N\big]$. The actually sought after indicator vector $\boldsymbol{z}^*$ is then contained in the first n components of $\hat{\boldsymbol{z}}^*$.

2.2 Hopfield Nets for Set Cover

Recall that a Hopfield net is a recurrent neural network of N neurons $s_1, \ldots, s_N$ which are bipolar threshold units. Hence, if $\boldsymbol{w}_j \in \mathbb{R}^N$ and $\theta_j \in \mathbb{R}$ denote the input weights and threshold value of neuron s_j and if $\boldsymbol{s} = [s_1, \ldots, s_N]^\top$ denotes the current global state of a Hopfield net, then neuron s_j computes its activation or local state as

$$s_j = \begin{cases} +1 & \text{if } \boldsymbol{w}_j^\top \boldsymbol{s} - \theta_j \geq 0 \\ -1 & \text{otherwise} \end{cases} \\ = \text{sign}\big(\boldsymbol{w}_j^\top \boldsymbol{s} - \theta_j\big) \tag{32}$$

Consequently, the global state of a Hopfield net always coincides with a bipolar vector $\boldsymbol{s} \in \{-1, +1\}^N$.

Also recall that we may gather all the weight- and threshold parameters of a Hopfield net in a matrix and a vector

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_N^\top \end{bmatrix} \quad \text{and} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_N \end{bmatrix} \tag{33}$$

Given this weight matrix and threshold vector, the energy of a Hopfield net in state $\boldsymbol{s}$ can be written as

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\, \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} + \boldsymbol{\theta}^\top \boldsymbol{s} \tag{34}$$

Finally, recall the following crucial fact: *If* the weight matrix of a Hopfield net is symmetric ($\boldsymbol{W} = \boldsymbol{W}^\top$) and hollow ($\text{diag}[\boldsymbol{W}] = \boldsymbol{0}$) and *if* its neurons update asynchronously (compute their activation one at a time), *then* the energy of a Hopfield net can never increase. As there are "only" 2^N distinct global states a Hopfield net can be in, it will converge to a state of (locally) minimum energy after finitely many updates [10].

All of this suggests that a Hopfield net can (approximately) solve a QUBO. Apparently, we only need to let $\boldsymbol{W} = -2\,\boldsymbol{Q}$ and $\boldsymbol{\theta} = \boldsymbol{q}$ to obtain a Hopfield net whose energy function corresponds to the QUBO's minimization objective.

However, there is a crucial subtlety we must pay attention to: Matrix $\boldsymbol{Q}$ in (27) may not be hollow, i.e. may have non-zero diagonal elements. Yet, hollowness of the weight matrix $\boldsymbol{W}$ of a Hopfield net is required to be able to guarantee that its global state eventually converges to a (local) minimum of its energy function.

We therefore note that we can write matrix $\boldsymbol{Q}$ in (27) as a sum of a diagonal matrix $\boldsymbol{D}$ and an off-diagonal matrix $\boldsymbol{O}$. The former represents the diagonal entries of $\boldsymbol{Q}$ and the latter represents the off-diagonal entries of $\boldsymbol{Q}$. This way, the quadratic term in (27) becomes

$$\boldsymbol{s}^\top \boldsymbol{Q}\, \boldsymbol{s} = \boldsymbol{s}^\top \big[\boldsymbol{D} + \boldsymbol{O}\big]\, \boldsymbol{s} = \boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} + \boldsymbol{s}^\top \boldsymbol{O}\, \boldsymbol{s} \tag{35}$$

Next, we point out that the term $\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s}$ necessarily amounts to

$$\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} = \sum_j \sum_i s_j\, D_{ij}\, s_i = \sum_j D_{jj}\, s_j^2 \tag{36}$$

because $\boldsymbol{D}$ is a diagonal matrix. Moreover, we will always have $s_j^2 = 1$, because $s_j \in \{-1, +1\}$. But this is to say that the term

$$\boldsymbol{s}^\top \boldsymbol{D}\, \boldsymbol{s} = \sum_j D_{jj} = const \tag{37}$$

has no impact on the solution of (27). Consequently, we may let

$$\boldsymbol{W} = -2\,\big[\boldsymbol{Q} - \boldsymbol{D}\big] \tag{38}$$

$$\boldsymbol{\theta} = \boldsymbol{q} \tag{39}$$

to obtain a Hopfield net with a hollow weight matrix that can (approximately) solve the problem in (27).

With respect to the update mechanism that governs the temporal evolution of a Hopfield net, we note that neuron s_u which updates its state in round t is typically chosen randomly. While this random selection mechanism is guaranteed to converge to a global state of (locally) minimum energy, it typically entails very slow convergence. We can do better than this.

Let $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t+1}$ be two consecutive states during the temporal evolution of a Hopfield net and assume that neuron s_u updated at time t. It is well known that the energy difference in this situation amounts to $\Delta E = E(\boldsymbol{s}_{t+1}) - E(\boldsymbol{s}_t) = 2 \cdot \big[\boldsymbol{s}_t\big]_u \cdot \big(\boldsymbol{w}_u^\top \boldsymbol{s}_t - \theta_u\big) \leq 0$.

This can be used to determine which neuron should update its activation in iteration t so as to maximally decrease the network's current energy $E(\boldsymbol{s}_t)$. Letting $\odot$ denote the Hadamard product, we may compute a vector

$$\Delta \boldsymbol{e} = 2 \cdot \boldsymbol{s}_t \odot \big[\boldsymbol{W}\, \boldsymbol{s}_t - \boldsymbol{\theta}\big] \tag{40}$$

of expected energy decrements, determine the index u of its most negative entry

$$u = \underset{j}{\text{argmin}}\, \big[\Delta \boldsymbol{e}\big]_j \tag{41}$$

and then update neuron s_u to obtain a new global state where

$$\big[\boldsymbol{s}_{t+1}\big]_j = \begin{cases} \big[\boldsymbol{s}_t\big]_u \cdot \text{sign}\big[\Delta \boldsymbol{e}\big]_u & \text{if } j = u \\ \big[\boldsymbol{s}_t\big]_j & \text{otherwise} \end{cases} \tag{42}$$

This greedy selection of the updating neurons will obviously minimize the network's energy but does so much faster than random selections. Alas, this mechanism may get trapped more easily in local minima of the energy landscape. However, set cover problems as considered in this note come with very rugged energy landscapes anyway so that random updates, too, are very likely to miss global minima. A valid strategy is thus to restart a Hopfield net many times from random global states and to keep track of the best solution found so far. Within this meta-algorithm, greedy updates of the states of the network will definitely be superior as they considerably reduce the duration of each round.

3 *NUMPY* IMPLEMENTATIONS

In this section, we will implement the above theory in *NumPy*. As a simple practical example, we once again consider the exemplary set cover problem in [2]. There we implemented the set $\mathcal{X}$ as

```
X = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

and realized the set of subsets $\mathcal{Y}$ among which to find a cover $\mathcal{Z}$ as

```
Y = dict(enumerate([{1},
                    {0, 1, 2},
                    {3, 4, 9},
                    {7, 8, 9},
                    {3, 4, 5, 6},
                    {0, 2, 4, 6, 8}]))
```

Listing 1: converting sets to indicator vectors matrices

```
def sets2mat(setX, dctY):
    m, n = len(setX), len(dctY),
    matY = np.zeros((n,m))
    for key, vals in dctY.items():
        matY[key, list(vals)] = 1

    return matY.T
```

Listing 2: setting the parameters of a set cover Hopfield net

```
1  def hnet_init_params(matY, l=1.):
2      m, n = matY.shape
3
4      omax = np.sum(matY, axis=1) - np.ones(m)
5      pwrs = [np.floor(np.log2(o)) if o>0 else 0 for o in omax]
6
7      nprm = np.sum(pwrs).astype(int) + m
8      matP = np.zeros((m, nprm))
9
10     j = 0
11     for i, p in enumerate(pwrs):
12         vecP = 2**np.arange(p+1)
13         dimP = len(vecP)
14         matP[i,j:j+dimP] = vecP
15         j += dimP
16
17     matY = np.hstack((matY, -matP))
18     m, N = matY.shape
19
20     vec1m = np.ones(m)
21     vec1N = np.ones(N)
22
23     matR = l * matY.T @ matY
24     vecR = vec1N - 2 * l * matY.T @ vec1m
25
26     matQ = 0.25 *  matR
27     vecQ = 0.50 * (matR @ vec1N + vecR)
28
29     matW = -2 * matQ; np.fill_diagonal(matW, 0)
30     vecT =      vecQ
31
32     return matW, vecT
```

For our purposes in this note, we need to translate the dictionary representing $\mathcal{Y}$ into a binary matrix Y. To this end, we may use function `sets2mat` in Listing 1 which resembles a procedure we discussed in [1]. Indeed,

```
matY = sets2mat(X, Y)
```

will produce a corresponding *NumPy* array `matY` of shape

```
m, n = matY.shape
```

Given this array, we can compute arrays `matW` and `vecT` which represent the weight matrix $\boldsymbol{W}$ and the threshold vector $\boldsymbol{\theta}$ of a set cover Hopfield net. To this end we call

```
matW, vecT = hnet_init_params(matY)
```

where function `hnet_init_params` is defined in Listing 2. Its first parameter `matY` is self explanatory; its second parameter `l` represent the Lagrange multiplier λ which, when in doubt, defaults to 1.

Line 4 computes the vector $\boldsymbol{o}_{\max}$ according to what has been worked out in (7). Line 5 computes the P_i according to (8) and line 7 computes n' according to (15).

Lines 8–15 compute matrix $\boldsymbol{P}$ defied in (12). First, we initialize an $m \times n'$ array `matP` of all zeros. We then iterate over the elements of list `pwrs` to compute arrays `vecP` representing the vectors $\boldsymbol{p}_i$ in (10). These are inserted at appropriate locations into `matP`.

Listing 3: greedy state updates for Hopfield nets

```
def signum(x):
    return np.where(x >= 0, +1, -1)


def hnet_run_greedy(vecS, matW, vecT, tmax=100):
    for t in range(tmax):
        grad = matW @ vecS - vecT
        updt = np.argmin(vecS * grad)
        vecS[updt] = signum(grad[updt])

    return vecS
```

Listing 4: (re-)running Hopfield nets for set cover

```
1  ebst = np.inf
2  for trial in range(100):
3      vecS = rnd.binomial(n=1, p=0.5, size=len(vecT)) * 2 - 1
4      vecS = hnet_run_greedy(vecS, matW, vecT, tmax=2*len(vecT))
5      eact = -0.5 * vecS @ matW @ vecS + vecT @ vecS
6      if eact < ebst:
7          sbst = vecS
8          ebst = eact
```

Line 17 creates an array representing matrix $\hat{\boldsymbol{Y}}$ in (17) and lines 20–21 implement the vectors $\mathbf{1}_m$ and $\mathbf{1}_N$. Lines 23–24 realize $\boldsymbol{R}$ and $\boldsymbol{r}$ in (30) and (31). These are then used to implement $\boldsymbol{Q}$ and $\boldsymbol{q}$ in (28) and (29), respectively. Finally, lines 29–30 implement *NumPy* arrays `matW` and `vecT` which represent our Hopfield net parameters as defied in (38) and (39). Regarding the former, we point out our use of `np.fill_diagonal` which is simple (and correct!) way of making sure that network's weight matrix is hollow.

Code for greedy activation updates for evolving a Hopfield net is provided in function `hnet_run_greedy` in Listing 3. This is an almost verbatim *NumPy* implementation of the above mathematics; however, we note our use of the custom made sign function `signum` which we have applied several times before [3, 4, 6].

Having all required ingredients and mechanisms in place, we can now run the meta-algorithm for solving set cover problems which we sketched above. Corresponding code is shown in Listing 4.

In line 1, we initialize a variable `ebst` to keep track of energy values at the end of each run of our Hopfield net. In this example, we perform 100 runs. In each run, we first randomly initialize an array `vecS` representing the initial state $\boldsymbol{s}_0$ of the Hopfield net. We then run `hnet_run_greedy` to determine a state $\boldsymbol{s}_t$ (where we here choose $t = 100$). Variable `eact` gets assigned the energy value $E(\boldsymbol{s}_t)$; if it is less than the current best energy value in `ebst`, we assign $\boldsymbol{s}_t$ to `sbst` and update our bookkeeping variable `ebst`.

For our running example, we may inspect the outcome of the whole procedure using

```
vecZ = (sbst+1) / 2
vecZ = vecZ[:n]
print (vecZ.astype(int))
print ((matY @ vecZ).astype(int))
```

This produces

```
>>> [0 1 0 1 1 0]
>>> [1 1 1 1 1 1 1 1 1 1]
```

and therefore indicates that our approach was successfully able to solve an (admittedly very simple) set cover problem.

4 CONCLUSION

In this note, we made good on an earlier promise and showed that the combinatorial problem of finding minimum set covers can be tackled using Hopfield nets.

The main work we had to do was to transform the set cover ILP we derived in [1] into a QUBO. For those who have never seen the use of slack variables in QUBOs before, this may have been a rough ride. We therefore note that a more gentle introduction to this truly important technique can be found in an excellent tutorial paper by Glover and colleagues [9].

Once we derived a QUBO formulation of the set cover problem, it was straightforward to turn it into an energy minimization problem that can be solved by Hopfield nets. This is of practical interest, because set covering problems frequently arise in industry (examples include job scheduling, shift planning, location planning, route selection, and other logistics tasks).

The crux is that the NP-hardness of set cover requires (very) special purpose software to be able to solve such problems at scale. Hopfield nets, on the other hand, are simple omnipurpose tools that can be implemented on modern high performance architectures and therefore allow for highly efficient number crunching [7, 8, 14, 15].

Indeed, as we never fail to mention when discussing Hopfield nets, if a problem can be solved by running a Hopfield nets, it can also be solved on (adiabatic) quantum computers [5]. While these have not yet reached the level of technological maturity required for large scale industrial applications, there is considerable progress with respect to practical applications of quantum computing [13, 17, 18]. Should these developments continue at their current pace, it seems likely that quantum set cover solvers will be deployed in practice soon.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] C. Bauckhage. 2022. *ML2R Coding Nuggets: Greedy Set Cover with Binary Numpy Arrays.* Technical Report. MLAI, University of Bonn.
[2] C. Bauckhage. 2022. *ML2R Coding Nuggets: Greedy Set Cover with Native Python Data Types.* Technical Report. MLAI, University of Bonn.
[3] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Bipartition Clustering.* Technical Report. MLAI, University of Bonn.
[4] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Max-Sum Diversification.* Technical Report. MLAI, University of Bonn.
[5] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN.* IEEE.
[6] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Hopfield Nets for Sorting.* Technical Report. MLAI, University of Bonn.
[7] D. Biesner, R. Sifa, and C. Bauckhage. 2022. Solving Subset Sum Problems using Binary Optimization with Applications in Auditing and Financial Data Analysis. *TechRxiv* (2022).
[8] S. Buschjäger, L. Pfahler, J. Buss, K. Morik, and W. Rhode. 2021. On-Site Gamma-Hadron Separation with Deep Learning on FPGAs. In *Proc. ECML PKDD.*
[9] F. Glover, G. Kochenberger, , R. Hennig, and Y. Du. 2022. Quantum Bridge Analytics I: A Tutorial on Formulating and Using QUBO Models. *Annals of Operations Research* 314 (2022).
[10] J.J. Hopfield. 1982. Neural Networks and Physical Systems with Collective Computational Abilities. *PNAS* 79, 8 (1982).
[11] J.J. Hopfield and D.W. Tank. 1985. "Neural" Computation of Decisions in Optimization Problems. *Biological Cybernetics* 52 (1985), 141–152.
[12] A. Lucas. 2014. Ising Formulations of Many NP Problems. *Frontiers in Physics* 2, 5 (2014).
[13] S. Mücke, R. Heese, S. Müller, M. Wolter, and N. Piatkowski. 2022. Quantum feature Selection. *arXiv:2203.13261 [quant-ph]* (2022).
[14] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD.*
[15] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. KDML-LWDA.*
[16] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[17] N. Piatkowski, T. Gerlach, R. Hugues, R. Sifa, C. Bauckhage, and F. Barbaresco. 2022. Towards Bundle Adjustment for Satellite Imaging via Quantum Machine Learning. In *Proc. FUSION.*
[18] N. Piatkowski and C. Zoufal. 2022. On Quantum Circuits for Discrete Graphical Models. *arXiv:2206.00398 [quant-ph]* (2022).