ML2R Coding Nuggets
Sorting as Linear Programming

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

Pascal Welke†
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

ABSTRACT

Linear programming is a surprisingly versatile tool. That is, many problems we would not usually think of in terms of a linear programming problem can actually be expressed as such. In this note, we show that sorting is such a problem and discuss how to solve linear programs for sorting using *SciPy*.

1 INTRODUCTION

We once again revisit the idea of **linear programming** [9, 10] and recall that it is all about solving optimization problems of the following form

$$\begin{aligned} \boldsymbol{z}^* = \underset{\boldsymbol{z}}{\text{argmin}} \quad & \boldsymbol{c}^\mathsf{T}\boldsymbol{z} \\ \text{s.t.} \quad & \boldsymbol{A}\,\boldsymbol{z} \preceq \boldsymbol{b} \\ & \boldsymbol{C}\,\boldsymbol{z} = \boldsymbol{d} \end{aligned} \tag{1}$$

The practical application we consider in this note is the fundamental problem of sorting an unordered collection of real numbers in ascending order. Every first year computer science student knows that this is not difficult and that there are numerous algorithms (such as *quicksort*) which sort quickly and reliably.

A much lesser known fact is that the sorting problem can be expressed as a linear programming problem of the form in (1). Does this have practical advantages? No, at least not on commodity hardware! Good conventional sorting algorithms are fast enough ($O(n \log n)$) to deal with millions of numbers. Linear programs, on the other hand, are more demanding in that their solution requires polynomial effort ($O(n^d)$ where $d > 2$). In this sense, the method we discuss in this note is of theoretical rather than practical interest. Our goals are, first of all, to demonstrate that linear programming is very widely applicable and, second of all, to show that even well known problems often allow for different points of view. The resulting computational solutions may come in handy once next generation devices (such as quantum computers or neuromorphic computers) become more readily available.

Next, we show how to derive a linear program for the sorting problem (section 2). This then allows us to apply function `linprog` in *SciPy*'s `optimize` package to sort lists of numbers (section 3). Readers who would like to experiment with our exemplary code should be familiar with *NumPy* and *SciPy* [8] and only need to

```
import numpy as np
import numpy.random as rnd
import scipy.optimize as opt
```

* 0000-0001-6615-2128
† 0000-0002-2123-3781

2 THEORY

In what follows, we consider an arbitrary n-dimensional vector of real numbers

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\mathsf{T} \tag{2}$$

whose entries x_i are supposed to be sorted in ascending order.

Working with vectors allows us to unleash linear algebraic tools on the sorting problem. Indeed, sorting the entries of $\boldsymbol{x}$ can also be understood as applying an $n \times n$ **permutation matrix** $\boldsymbol{P}$ such that the entries of the permuted vector

$$\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x} \tag{3}$$

obey $y_1 \leq y_2 \leq \ldots \leq y_n$. This way, the sorting problem becomes the problem of finding an appropriate permutation matrix.

In order to devise an objective function whose minimization would yield the sought after permutation matrix, we next introduce a very specific, auxiliary n-dimensional vector, namely

$$\boldsymbol{n} = [1, 2, \ldots, n]^\mathsf{T} \tag{4}$$

Given this vector, the **rearrangement inequality** due to Hardy, Littlewood, and Polya [5] implies that $-\boldsymbol{n}^\mathsf{T}\boldsymbol{P}\,\boldsymbol{x} = -\boldsymbol{n}^\mathsf{T}\boldsymbol{y}$ is minimal, if $y_1 \leq y_2 \leq \ldots \leq y_n$. In other words, the expression $-\boldsymbol{n}^\mathsf{T}\boldsymbol{P}\,\boldsymbol{x}$ is minimal if the permutation matrix $\boldsymbol{P}$ causes the entries of $\boldsymbol{x}$ to be sorted in the same order as the entries of $\boldsymbol{n}$.

Next, we recall that a **doubly stochastic matrix** is a square, non-negative matrix whose rows and columns all sum to one. That is, $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ is doubly stochastic, if $\boldsymbol{M} \geq \boldsymbol{0}_{n \times n}$, $\boldsymbol{M}\boldsymbol{1}_n = \boldsymbol{1}_n$, and $\boldsymbol{M}^\mathsf{T}\boldsymbol{1}_n = \boldsymbol{1}_n$ where $\boldsymbol{0}_{n \times n}$ denotes the $n \times n$ matrix of all zeros and $\boldsymbol{1}_n$ is the n vector of all ones. We also recall that the **Birkhoff polytope** $\mathcal{B}_n$ is the convex hull of all doubly stochastic matrices of size $n \times n$ and that the **Birkhoff-von Neumann theorem** establishes that the $n!$ vertices of $\mathcal{B}_n$ are the $n \times n$ permutation matrices [12].

All of this is to say that the problem of sorting the elements of $\boldsymbol{x}$, i.e. the problem of finding an appropriate permutation matrix $\boldsymbol{P}$, can be cast as a linear programming problem, namely

$$\begin{aligned} \boldsymbol{P} = \underset{\boldsymbol{M} \in \mathbb{R}^{n \times n}}{\text{argmin}} \quad & -\boldsymbol{n}^\mathsf{T}\boldsymbol{M}\,\boldsymbol{x} \\ & \boldsymbol{M} \geq \boldsymbol{0}_{n \times n} \\ \text{s.t.} \quad & \boldsymbol{M}\boldsymbol{1}_n = \boldsymbol{1}_n \\ & \boldsymbol{M}^\mathsf{T}\boldsymbol{1}_n = \boldsymbol{1}_n \end{aligned} \tag{5}$$

Indeed, due to its constraints, the feasible set of this minimization problem is the Birkhoff polytope $\mathcal{B}_n$ which is a convex set. Also, as shown in the appendix, the objective $-\boldsymbol{n}^\mathsf{T}\boldsymbol{M}\,\boldsymbol{x}$ is linear in $\boldsymbol{M}$. Since the minimum of a linear function over a convex set coincides with a vertex of said set and since the vertices of $\mathcal{B}_n$ are permutation matrices, the solution to (5) will be a permutation matrix.

At this point, we have already reached our first major goal. That is, we have established that sorting can be accomplished via linear programming.

However, our linear program in (5) is a linear program over matrices and thus not of the commonly considered form in (1). Since most software tools for linear programming assume the problem to be formulated in terms of vectors (*SciPy*'s `linprog` is no exception), we still have work to do before we can turn our new insights into practical code. In other words, we have to rewrite the problem in (5) as a problem of the form in (1).

For this purpose, it will be more convenient to work with the transpose $-\boldsymbol{x}^\intercal \boldsymbol{M}^\intercal \boldsymbol{n}$ of the objective function in (5). In the appendix, we prove the following identity

$$\boldsymbol{M}^\intercal \boldsymbol{n} = \boldsymbol{N}\,\boldsymbol{m} \tag{6}$$

where the vector $\boldsymbol{m} \in \mathbb{R}^{n^2}$ and matrix $\boldsymbol{N} \in \mathbb{R}^{n \times n^2}$ are defined as

$$\boldsymbol{m} = \text{vec}\,(\boldsymbol{M}) \tag{7}$$

$$\boldsymbol{N} = \boldsymbol{I}_{n\times n} \otimes \boldsymbol{n}^\intercal \tag{8}$$

where $\boldsymbol{I}_{n\times n}$ is the $n \times n$ identity matrix and $\otimes$ denotes the Kronecker product. In other words, we can express the product of the $n \times n$ matrix $\boldsymbol{M}^\intercal$ and the n vector $\boldsymbol{n}$ as the product of an $n \times n^2$ matrix $\boldsymbol{N}$ and an n^2 vector $\boldsymbol{m}$. The vector $\boldsymbol{m} = \text{vec}\,(\boldsymbol{M})$ contains the stacked columns of $\boldsymbol{M}$ and the matrix $\boldsymbol{N} = \boldsymbol{I}_{n\times n} \otimes \boldsymbol{n}^\intercal$ amounts to

$$\boldsymbol{N} = \begin{bmatrix} \boldsymbol{n}^\intercal & \boldsymbol{0}_n^\intercal & \boldsymbol{0}_n^\intercal & \cdots & \boldsymbol{0}_n^\intercal & \boldsymbol{0}_n^\intercal \\ \boldsymbol{0}_n^\intercal & \boldsymbol{n}^\intercal & \boldsymbol{0}_n^\intercal & \cdots & \boldsymbol{0}_n^\intercal & \boldsymbol{0}_n^\intercal \\ & & \cdots & \cdots & & \\ \boldsymbol{0}_n^\intercal & \boldsymbol{0}_n^\intercal & \boldsymbol{0}_n^\intercal & \cdots & \boldsymbol{0}_n^\intercal & \boldsymbol{n}^\intercal \end{bmatrix} \tag{9}$$

where $\boldsymbol{0}_n$ denotes the n vector of all zeros.

Similar arguments apply to the expressions in the two equality constraints of problem (5). First of all, we have

$$\boldsymbol{M}^\intercal \boldsymbol{1}_n = \boldsymbol{C}_c\,\boldsymbol{m} \tag{10}$$

where the vector $\boldsymbol{m}$ is given as above and matrix $\boldsymbol{C}_c$ is defined as

$$\boldsymbol{C}_c = \boldsymbol{I}_{n\times n} \otimes \boldsymbol{1}_n^\intercal \tag{11}$$

Second of all, we observe that

$$\boldsymbol{M}\,\boldsymbol{1}_n = \boldsymbol{C}_r\,\boldsymbol{m} \tag{12}$$

where matrix $\boldsymbol{C}_r$ is defined as

$$\boldsymbol{C}_r = \boldsymbol{1}_n^\intercal \otimes \boldsymbol{I}_{n\times n} \tag{13}$$

With respect to the inequality constraint of problem (5), we note that $\boldsymbol{M} \geq \boldsymbol{0}_{n\times n} \Leftrightarrow -\boldsymbol{M} \leq \boldsymbol{0}_{n\times n}$ as well as

$$-\boldsymbol{M} \leq \boldsymbol{0}_{n\times n} \Leftrightarrow -\boldsymbol{m} \leq \boldsymbol{0}_{n^2} \Leftrightarrow -\boldsymbol{I}_{n^2\times n^2}\boldsymbol{m} \leq \boldsymbol{0}_{n^2} \tag{14}$$

Using all of this, we can therefore rewrite our matrix linear programming problem in terms of a vector linear programming problem, namely

$$\begin{aligned} \boldsymbol{p} = \underset{\boldsymbol{m}\in\mathbb{R}^{n^2}}{\text{argmin}} \;& -\boldsymbol{x}^\intercal \boldsymbol{N}\,\boldsymbol{m} \\ & -\boldsymbol{I}_{n^2\times n^2}\boldsymbol{m} \leq \boldsymbol{0}_{n^2} \\ \text{s.t.} \quad & \boldsymbol{C}_r\boldsymbol{m} = \boldsymbol{1}_n \\ & \boldsymbol{C}_c\boldsymbol{m} = \boldsymbol{1}_n \end{aligned} \tag{15}$$

Assuming this problem has been solved, we can recover the originally sought after permutation matrix as $\boldsymbol{P} = \text{mat}\,(\boldsymbol{p})$ so as to then obtain the sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of $\boldsymbol{x}$.

However, while (15) is beginning to resemble the common form of a linear programming problem in (1), we are not quite there yet. The major difference between (1) and (15) is that the former involves only one equality constraint whereas the latter involves two.

But we can easily iron out these (mainly notational) differences. Indeed, the substitutions

$$\boldsymbol{c}^\intercal = -\boldsymbol{x}^\intercal \boldsymbol{N} \tag{16}$$

$$\boldsymbol{A} = -\boldsymbol{I}_{n^2\times n^2} \tag{17}$$

$$\boldsymbol{b} = \boldsymbol{0}_{n^2} \tag{18}$$

and, crucially

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_r \\ \boldsymbol{C}_c \end{bmatrix} \tag{19}$$

$$\boldsymbol{d} = \begin{bmatrix} \boldsymbol{1}_n \\ \boldsymbol{1}_n \end{bmatrix} \tag{20}$$

allow us to also write the problem in (15) as follows

$$\begin{aligned} \boldsymbol{p} = \underset{\boldsymbol{m}\in\mathbb{R}^{n^2}}{\text{argmin}} \;& \boldsymbol{c}^\intercal \boldsymbol{m} \\ \text{s.t.} \quad & \boldsymbol{A}\,\boldsymbol{m} \leq \boldsymbol{b} \\ & \boldsymbol{C}\,\boldsymbol{m} = \boldsymbol{d} \end{aligned} \tag{21}$$

This is now a problem of the exact same form as in (1). In practice, we can therefore apply *SciPy*'s `linprog` function to solve it. In other words, we can practically use linear programming to sort a collection of numbers.

3 PRACTICAL COMPUTATION

Next, we look at how to use *SciPy* to solve the linear program in (21) and thus to sort the entries of a vector $\boldsymbol{x} \in \mathbb{R}^n$.

To work with a practical example, we first create a random vector $\boldsymbol{x}$ of, say, $n = 5$ entries $0 \leq x_i \leq 100$ which we represent as a one-dimensional *NumPy* array. To keep things legible, we will force the x_i to be integers and proceed as follows

```
n    = 5
vecX = rnd.randint(100, size=n)
```

In order to inspect the entries of this random vector, we simply use

```
print (vecX)
```

and may obtain something like this

```
[46 52 12 10 51]
```

Given $\boldsymbol{x}$, we next initialize a *NumPy* array which represent the auxiliary vector $\boldsymbol{n}$. To this end, we may use

```
vecN = np.arange(n) + 1
```

to obtain

```
[1 2 3 4 5]
```

Having completed all these preparatory steps, we next set up the ingredients of the linear program in (21). To this end, we may proceed as shown in Listing 1.

Listing 1: setting up the linear programming problem in (21)

```
1  matI = np.eye(n)
2  vec1 = np.ones(n)
3
4  matN  = np.kron(matI, vecN)
5  matCc = np.kron(matI, vec1)
6  matCr = np.kron(vec1, matI)
7
8  matC = np.vstack((matCr, matCc))
9  vecD = np.hstack((vec1, vec1))
10
11 vecC = -vecX @ matN
```

Arrays `matI` and `vec1` represent the $n \times n$ identity matrix and the n-dimensional vector of all ones, respectively.

Arrays `matN`, `matCc`, and `matCr` represent the matrices $\boldsymbol{N}$, $\boldsymbol{C}_c$, and $\boldsymbol{C}_r$ which we defined in equations (8), (11), and (13). In order to compute these arrays, we apply the *NumPy* function `kron` which implements the Kronecker product. In other words, lines 4–6 in Listing 1 directly correspond to the mathematical expressions in (8), (11), and (13).

In line 8, we initialize an array `matC` which represents the $2\,n \times n^2$ matrix $\boldsymbol{C}$ defined in (19) and array `vecD` in line 9 represents vector $\boldsymbol{d}$ defined in (20).

Finally, in line 11, we compute an array `vecC` which represents vector $\boldsymbol{c}$ in (16). Long-time readers of the series will have noticed our use of

```
-vecX @ matN
```

instead of a more cumbersome expression such as

```
-np.dot(vecX, matN)
```

Both expressions are indeed equivalent since the infix operator `@` realizes matrix-vector- or matrix-matrix multiplication. While we rarely used `@` in previous notes, PEP 465 recommends its use and we have lately come to appreciate the convenience it provides.

Attentive readers will have also noticed that we did not implement matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ in (17) and (18), respectively. In our current setting, this is indeed unnecessary. To understand why, we recall our discussion in [9] where we pointed out that the questionable default behavior of *SciPy*'s `linprog` function is to assume that the solution of a linear program is non-negative. In our current setting, this is actually convenient as we do not have to care about the non-negativity enforcing inequality constraint in (21).

Hence, we are good to go and can finally invoke function `linprog` in *SciPy*'s `optimize` module. To solve our problem, we simply call

```
result = opt.linprog(vecC, A_eq=matC, b_eq=vecD)
```

Note that, in contrast to our examples in [9, 10], we now set the parameters `A_eq` and `b_eq` which define equality constraints. Another difference to our earlier examples is that `linprog` now issues the following warning

> OptimizeWarning: `A_eq` does not appear to be of full row rank. To improve performance, check the problem formulation for redundant equality constraints.

This is certainly correct; matrix $\boldsymbol{C}$ passed in the argument `A_eq` is rank deficient but there is nothing we can do about it. Indeed, the warning appears to be inconsequential, because printing `result` for inspection yields

```
     con: array([-3.96327415e-12, ...])
     fun: -636.0000000020979
 message: 'Optimization terminated successfully.'
     nit: 7
   slack: array([], dtype=float64)
  status: 0
 success: True
       x: array([4.88698700e-13, ...])
```

which indicates that `linprog` succeeded in finding a solution.

The following statements therefore determine the solution $\boldsymbol{p}$ of our linear program and the corresponding permutation matrix $\boldsymbol{P}$

```
vecP = result.x
matP = vecP.reshape(n,n).T
```

and subsequent (rounded) printing of the array `matP` results in

```
[[0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0.]]
```

Finally, in order to verify that the above permutation matrix solves our problem, we use

```
print ('vecX = ', vecX.astype(float))
print ('vecY = ', matP @ vecX)
```

and obtain

```
vecX =  [46. 52. 12. 10. 51.]
vecY =  [10. 12. 46. 51. 52.]
```

Success! We have solved a linear programming problem to get a sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of an unordered vector $\boldsymbol{x}$.

NOTE: While our example corroborates that sorting can be done via linear programming, we must point out that this approach is rather slow and typically suffers from numerical instabilities. Above, we see (from looking at `result.con`) that `linprog` found a solution with non-zero residuals for the equality constraint of our problem. As a consequence, the entries of `result.x` are very close to 0 or 1 but not exactly 0 or 1. When working with a small ($n = 5$) input vector $\boldsymbol{x}$, this (floating point) imprecision does not have any noticeable impact. However, tiny errors may add up. Readers are encouraged to verify this for themselves and see what happens when considering a vector $\boldsymbol{x}$ of, say, $n = 100$ elements.

4 SUMMARY AND OUTLOOK

This note showed that linear programming is a surprisingly versatile tool. That is, many familiar problems we would not usually consider as a linear programming problem can actually be expressed as such and therefore be solved using linear programming solvers.

The particular problem we explored was the problem of sorting a lists of real numbers. We derived a corresponding linear program and solved it using function `linprog` in *SciPy*'s `optimize` module. In contrast to the practical applications in our previous notes [9, 10], the sorting problem required us to work with equality constraints which was easily done. Yet, for the particular equality constraint we had to consider, `linprog` issued a rank deficiency warning. While this was of little consequence for our toy example of sorting $n = 5$ numbers, such warnings are there for a reason.

Indeed, we cannot recommend the above approach as a practical solution. For larger n, it is much slower than conventional sorting algorithms and will suffer from floating point imprecision.

On the other hand, the fact that we may conceptualize sorting as a linear algebraic optimization problem is of interest to those working on next generation computing paradigms. Indeed, (21) can be further rewritten and cast as a binary unconstrained quadratic optimization problem for quantum computing [1, 2, 4]. This, too, is not really remarkable because sorting is not the kind of problem that calls for heavy machinery. However, the way of thinking we sketched in this note promises new solutions for much, much more demanding permutation problems [3, 6, 7, 11] and we will get back to it later.

A APPENDIX

In this appendix, we prove the not so obvious claims we made in section 2.

Lemma A.1. *If $\boldsymbol{n}, \boldsymbol{x} \in \mathbb{R}^n$ are two given vectors and $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ denotes a variable matrix, then the function $f : \mathbb{R}^{n \times n} \longrightarrow \mathbb{R}$ with*

$$f(\boldsymbol{M}) = \boldsymbol{n}^\mathsf{T} \boldsymbol{M} \boldsymbol{x}$$

is linear in $\boldsymbol{M}$.

Proof. Consider $a, b \in \mathbb{R}$ and $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$, then

$$\begin{aligned} f(a\boldsymbol{A} + b\boldsymbol{B}) &= \boldsymbol{n}^\mathsf{T}(a\boldsymbol{A} + b\boldsymbol{B})\boldsymbol{x} \\ &= a\,\boldsymbol{n}^\mathsf{T}\boldsymbol{A}\boldsymbol{x} + b\,\boldsymbol{n}^\mathsf{T}\boldsymbol{B}\boldsymbol{x} = a\,f(\boldsymbol{A}) + b\,f(\boldsymbol{B}) \end{aligned}$$

□

Lemma A.2. *Let $\boldsymbol{n} \in \mathbb{R}^n$ and $\boldsymbol{M} \in \mathbb{R}^{n \times n}$. Then there exist a vector $\boldsymbol{m} \in \mathbb{R}^{n^2}$ and a matrix $\boldsymbol{N} \in \mathbb{R}^{n \times n^2}$ such that*

$$\boldsymbol{M}^\mathsf{T}\boldsymbol{n} = \boldsymbol{N}\boldsymbol{m}$$

In particular, the claim hods true for

$$\begin{aligned} \boldsymbol{m} &= \mathrm{vec}(\boldsymbol{M}) \\ \boldsymbol{N} &= \boldsymbol{I} \otimes \boldsymbol{n}^\mathsf{T} \end{aligned}$$

where $\boldsymbol{I}$ is the $n \times n$ identity matrix and $\otimes$ denotes the Kronecker product.

Proof. Consider the vector $\boldsymbol{v} = \boldsymbol{M}^\mathsf{T}\boldsymbol{n}$. In terms of column $\boldsymbol{m}_i$ of matrix $\boldsymbol{M} = [\boldsymbol{m}_1, \ldots, \boldsymbol{m}_n]$, entry v_i of $\boldsymbol{v}$ is given by $v_i = \boldsymbol{m}_i^\mathsf{T}\boldsymbol{n}$ or, equivalently, $v_i = \boldsymbol{n}^\mathsf{T}\boldsymbol{m}_i$.

If we introduce two (much) larger vectors $\boldsymbol{m} \in \mathbb{R}^{n^2}$ and $\boldsymbol{n}_i \in \mathbb{R}^{n^2}$ where $\boldsymbol{m} = \mathrm{vec}(\boldsymbol{M})$ contains the stacked columns of $\boldsymbol{M}$ and

$$\boldsymbol{n}_i^\mathsf{T} = \big[\underbrace{\boldsymbol{0}^\mathsf{T} \cdots \boldsymbol{0}^\mathsf{T}}_{i-1 \text{ times}}\ \boldsymbol{n}^\mathsf{T}\ \underbrace{\boldsymbol{0}^\mathsf{T} \cdots \boldsymbol{0}^\mathsf{T}}_{n-i \text{ times}}\big]$$

with $\boldsymbol{0} \in \mathbb{R}^n$, we can also write $v_i = \boldsymbol{n}_i^\mathsf{T}\boldsymbol{m}$.

Hence, if we gather the n different vectors $\boldsymbol{n}_i^\mathsf{T}$ as the rows of a matrix $\boldsymbol{N} \in \mathbb{R}^{n \times n^2}$, i.e. if we consider

$$\boldsymbol{N} = \begin{bmatrix} \boldsymbol{n}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} & \cdots & \boldsymbol{0}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} \\ \boldsymbol{0}^\mathsf{T} & \boldsymbol{n}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} & \cdots & \boldsymbol{0}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{0}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} & \boldsymbol{0}^\mathsf{T} & \cdots & \boldsymbol{0}^\mathsf{T} & \boldsymbol{n}^\mathsf{T} \end{bmatrix} = \boldsymbol{I} \otimes \boldsymbol{n}^\mathsf{T}$$

we find that $\boldsymbol{v}$ can also be written as $\boldsymbol{v} = \boldsymbol{N}\boldsymbol{m}$. □

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. 01|S18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] C. Bauckhage, E. Brito, K. Cvejoski, C. Ojeda, R. Sifa, and S. Wrobel. 2017. Ising Models for Binary Clustering via Adiabatic Quantum Computing. In *Proc. EMMCVPR*. Springer.
[2] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN*. IEEE.
[3] C. Bauckhage, R. Sifa, and S. Wrobel. 2020. Adiabatic Quantum Computing for Max-Sum Diversication. In *Proc. SDM*. SIAM.
[4] F. Glover, G. Kochenberger, and Y. Du. 2018. A Tutorial on Formulating and Using QUBO Models. *arXiv:1811.11538 [cs.DS]* (2018).
[5] G.H. Hardy, J.E. Littlewood, and G. Polya. 1952. *Inequalities*. Cambridge University Press.
[6] J. Kunegis, D. Fay, and C. Bauckhage. 2010. Network Growth and the Spectral Evolution Model. In *Proc. CIKM*. ACM.
[7] A. Nowak, S. Villar, A.S. Bandeira, and J. Bruna. 2017. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450 [stat.ML]* (2017).
[8] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[9] P. Welke and C. Bauckhage. 2020. *ML2R Coding Nuggets: Linear Programming for Robust Regression*. Technical Report. MLAI, University of Bonn.
[10] P. Welke and C. Bauckhage. 2020. *ML2R Coding Nuggets: Solving Linear Programming Problems*. Technical Report. MLAI, University of Bonn.
[11] M.M. Zavlanos and G. J. Pappas. 2008. A Dynamical Systems Approach to Weighted Graph Matching. *Automatica* 44, 11 (2008).
[12] G.M. Ziegler. 1995. *Lectures on Polytopes*. Springer.