# ML2R Coding Nuggets
# Greedy Set Cover with Binary *NumPy* Arrays

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

## ABSTRACT

We revisit the minimum set cover problem and formulate it as an integer linear program over binary indicator vectors. Next, we simply adapt our earlier code for greedy set covering to indicator vector representations.

## 1 INTRODUCTION

When we first looked at the **set cover problem** [1], we specified it as follows: Given a set $\mathcal{U}$ of $n$ elements and a set $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ of $m$ subsets $\mathcal{S}_i \subseteq \mathcal{U}$, find a minimum size subset $C \subseteq \mathcal{S}$ such that the union of the $\mathcal{S}_j \in C$ covers $\mathcal{U}$, i.e. such that

$$\bigcup_{\mathcal{S}_j \in C} \mathcal{S}_j = \mathcal{U}$$

We then saw that this seemingly innocent problem is actually very difficult. Indeed, its decision version ("is there a cover $C$ of $\mathcal{U}$ of size $|C| = k < m$ ?") is NP-complete [5]. Hence, as far as we know today, any algorithm guaranteed to find a minimum cover has a runtime exponential in $m$. Yet, we also discussed "the" greedy algorithm for polynomial time approximations of set covering and implemented *Python* code using the data types `set` and `dict`.

In this note, we consider a more abstract way of thinking about set covering. In particular, we explain that subsets can be represented in terms of indicator vectors and that such representations allow us to cast set covering as an **integer linear program (ILP)** over binary variables.

While the ILP formulation of set cover may be less intuitive than the specification stated above, it can lead to more efficient implementations. Having said this, we must make sure not to be misunderstood: We do not claim that indicator vector representations of set cover problems lead to polynomial rather than exponential algorithms. In fact, binary ILPs are generally NP-complete [5].

What we claim is that the indicator vector point of view on set covering allows for implementations with low memory footprints irrespective of the nature of the sets at hand. (They could contain integers just as in our example in [1] but they could also contain, say, memory intensive high resolution photographs. For the former, our *Python* code in [1] may be acceptable; for the latter it is not.) Moreover, the ILP we derive below can be turned into a QUBO and thus be solved using Hopfield nets or (adiabatic) quantum computers [2].

However, realizations of this last idea will be left to later. Here, we simply adapt the greedy algorithm for polynomial time approximations of set covering to indicator vector representations. Our respective code involves *NumPy* `array` objects [6] and requires to

```
import numpy as np
```

## 2 FORMULATING SET COVER AS AN ILP

To segue into the general topic of this section, we will base our discussion on a specific example. Let us therefore replicate the simple set cover problem in [1], albeit in a more general manner.

Recall that our example in [1] involved a set $\mathcal{U}$ of $n = 10$ integers and that $\mathcal{S}$ contained $m = 6$ subsets of these integers. Now, instead of forcing the elements of our set and subsets to be integers, we allow them to be any kind of objects and therefore use indexed variables $x_1, x_2, \ldots, x_{10}$ to refer to them.

Assuming this more general perspective, our exemplary set $\mathcal{U}$ and the respective set $\mathcal{S}$ of subsets $\mathcal{S}_1, \ldots, \mathcal{S}_6$ can be written down in the following rather suggestive tabular manner

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

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

Why is this a "suggestive" manner of expressing our problem? Well, first of all, this tabular representation makes it easy to *see* that, say, the union of $\mathcal{S}_2$, $\mathcal{S}_4$, and $\mathcal{S}_5$ covers $\mathcal{U}$. It further makes it fairly easy to *see* that $C_* = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$ constitutes the optimal cover.

Alas, these advantages w.r.t. visual problem solving only manifest for rather small problems. Imagine we were dealing with a problem where $n = |\mathcal{U}|$ and $m = |\mathcal{S}|$ were very large. Who would, let alone could, visually inspect tables of millions of rows and columns?

Second of all, we therefore observe that this representation in which subsets form rows of a table and subset elements are placed in certain columns hints at another way of formalizing set cover problems.

**Consider this**: Whenever we are given a set $\mathcal{U} = \{x_1, \ldots, x_n\}$ of an arbitrary number $n$ of arbitrary yet indexed objects $x_j$, we can *represent* any subset $\mathcal{S}_i \subseteq \mathcal{U}$ in terms of a binary **indicator vector**

$$\boldsymbol{s}_i \in \{0,1\}^n$$

whose entries are given by

$$\big[\boldsymbol{s}_i\big]_j = \begin{cases} 1 & \text{if } x_j \in \mathcal{S}_i \\ 0 & \text{otherwise} \end{cases}$$

**Note:** Throughout, we write $[\boldsymbol{v}]_k$ to denote the $k$-th entry of a finite dimensional vector $\boldsymbol{v}$.

For instance, in our running example, subsets $\mathcal{S}_1$ and $\mathcal{S}_6$ of $\mathcal{U}$ and $\mathcal{U}$ itself can be encoded by the following 10-dimensional binary vectors

$$\begin{aligned} \boldsymbol{s}_1 &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^\top \\ \boldsymbol{s}_6 &= \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}^\top \\ \boldsymbol{u} &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^\top \end{aligned}$$

Here is another example: Subset $C_*$ of $\mathcal{S}$ can be encoded in terms of this 6-dimensional binary vector

$$\boldsymbol{c}_* = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}^\top$$

Next, we observe that the idea of representing subsets in terms of indicator vectors easily extends towards sets of subsets.

**Consider this:** If we are given a whole set $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ of subsets $\mathcal{S}_i \subseteq \mathcal{U}$, we can represent this collections of $m$ subsets of an $n$ element set as a binary indicator matrix $S \in \{0,1\}^{n \times m}$ where

$$S = \begin{bmatrix} \boldsymbol{s}_1 & \boldsymbol{s}_1 & \cdots & \boldsymbol{s}_m \end{bmatrix}$$

For instance, if we represent subsets $\mathcal{S}_1, \ldots, \mathcal{S}_6$ of our running example as binary indicator vectors $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_6$, collect these in a matrix $S$, and look at the transpose of this matrix, we find

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

Now, the obvious question is what this way of representing the ingredients of a set cover problem buys us?

**Consider this:** Working with a binary matrix $S$ whose $m$ column vectors represent subsets of a set of $n$ elements, the problem of finding a cover of said set consists in finding a collection of column vectors which, when added, produce a vector of non-zero entries.

How do we add a collection of columns of an $n \times m$ matrix $S$? By multiplying it with an $m$-dimensional indicator vector $\boldsymbol{c}$ whose entries indicate which columns to add!

But this means that we can think of set covering as a linear algebraic search problem: If we could find a vector $\boldsymbol{c}$ such that

$$\hat{\boldsymbol{u}} = S\,\boldsymbol{c}$$

obeys

$$\hat{\boldsymbol{u}} \geq \mathbf{1}_n$$

where $\mathbf{1}_n$ denotes the $n$-dimensional vector of all ones, then we would have found a cover of the set whose subsets are represented by the columns of matrix $S$.

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.

How do we determine the size of a cover $C$ represented in terms of a binary indicator vector $\boldsymbol{c}$? By adding the entries of this binary vector! That is, by computing

$$|C| = \sum_{j=1}^{m} c_j = \sum_{j=1}^{m} \big(1 \cdot c_j\big) = \mathbf{1}_m^\top \boldsymbol{c}$$

where $\mathbf{1}_m$ denotes the $m$-dimensional vector of all ones.

All of this is then to say that we can formulate the minimum set cover problem in terms of the following **integer linear program**

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

Since we studied linear programming problems before [8, 9], we immediately recognize (1) as an LP. What turns it into an ILP is this crucial restriction: The vectors $\boldsymbol{c}$ over which to optimize cannot vary continuously in $\mathbb{R}^m$ but must be vectors in $\{0,1\}^m \subset \mathbb{Z}^m$.

This restriction makes ILPs difficult to solve; indeed, they are known to be NP-complete in general [5]. Practitioners therefore typically work with approximation algorithms or heuristics. This is rather easy (at least to those who can afford it) because, since large scale ILPs arise in many industrial applications, there exist various commercial solvers (see, for instance, [3] or [4]).

Our strategies for solving (1) will be different though. Below, we adapt the greedy set cover algorithm to indicator vector / matrix representations of the problem. In upcoming notes, we will then show how to rewrite the above ILP in terms of a QUBO which can be solved using Hopfield nets or (adiabatic) quantum computing.

## 3 *NUMPY* IMPLEMENTATIONS

Before we discuss ideas for *NumPy* implementations of indicator vector versions of the greedy set cover algorithm, we will first compare pseudo code of its set-based variant (Alg. 1) to the pseudo code for an indicator vector-based variant (Alg. 2).

As we can see, the logic for both variants is the same but we swapped operations on sets in Alg. 1 for operations on indicator vectors in Alg. 2. These operations crucially exploit the fact that we are dealing with binary vectors. For instance, we swapped the operation $\mathcal{S}_i \cap \mathcal{U}$ for $\boldsymbol{s}_i \odot \boldsymbol{u}$ where $\odot$ is the **Hadamard product**, i.e. the element-wise product of two vectors. This works, because, for $\boldsymbol{s}_i, \boldsymbol{u} \in \{0,1\}^n$, we have

$$\big[\boldsymbol{s}_i \odot \boldsymbol{u}\big]_k = \begin{cases} 1 & \text{if } [\boldsymbol{s}_i]_k = 1 \wedge [\boldsymbol{u}]_k = 1 \\ 0 & \text{otherwise} \end{cases}$$

By the same token, we swapped the operation $\mathcal{U} \backslash \mathcal{S}_j$ for $\boldsymbol{u} \odot \big[\mathbf{1}_n - \boldsymbol{s}_j\big]$ and encourage our readers to verify that this is reasonable. Finally, when working with binary indicator vectors, the set operation $C \cup \{S_j\}$ simply becomes to set $[\boldsymbol{c}]_j$ to 1.

**Note:** If we wanted to, we could also understand our binary vectors as bit-strings. In this case, we would compute $\boldsymbol{s}_i \odot \boldsymbol{u}$ as $\boldsymbol{s}_i \wedge \boldsymbol{u}$ and $\boldsymbol{u} \odot \big[\mathbf{1}_n - \boldsymbol{s}_j\big]$ as $\boldsymbol{u} \wedge \neg \boldsymbol{s}_j$ where $\wedge$ denotes the element-wise conjunction of two bit strings and $\neg$ is the element-wise negation of a bit string. In practice, this point of view may lead to particularly efficient code. However, we deliberately decided to express our pseudo code in terms of linear algebraic operations. Even though *NumPy* ships with methods for computing logical operations, we still would have to mix them with non-logical operation for counting the number of 1s in a bit string.

Speaking of *NumPy*, let's now practically implement the pseudo code in Alg. 2. To assess the practical performance of our implementations, we will again apply our code to the exemplary problem in [1]. There we implemented the set $\mathcal{U}$ as

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

**Algorithm 1: greedy set cover with sets**

$C \leftarrow \emptyset$ // initialize $C$

**while** $\mathcal{U} \neq \emptyset$

$\mathcal{S}_j \leftarrow \underset{\mathcal{S}_i \in \mathcal{S}}{\operatorname{argmax}} |\mathcal{S}_i \cap \mathcal{U}|$ // select $\mathcal{S}_j \in \mathcal{S}$ covering most of $\mathcal{U}$

$\mathcal{U} \leftarrow \mathcal{U} \setminus \mathcal{S}_j$ // remove elements of $\mathcal{S}_j$ from $\mathcal{U}$

$C \leftarrow C \cup \{\mathcal{S}_j\}$ // add $\mathcal{S}_j$ to $C$

**Algorithm 2: greedy set cover with binary indicator vectors**

$\boldsymbol{c} \leftarrow \mathbf{0}_m$ // initialize $\boldsymbol{c}$

**while** $\boldsymbol{u} \neq \mathbf{0}_n$

$j \leftarrow \underset{\boldsymbol{s}_i \in \boldsymbol{S}}{\operatorname{argmax}} \mathbf{1}_n^\intercal [\boldsymbol{s}_i \odot \boldsymbol{u}]$ // select $\boldsymbol{s}_j$ in $\boldsymbol{S}$ covering most of $\boldsymbol{u}$

$\boldsymbol{u} \leftarrow \boldsymbol{u} \odot [\mathbf{1}_n - \boldsymbol{s}_j]$ // set $[\boldsymbol{u}]_k$ to 0, if $[\boldsymbol{s}_j]_k = 1$

$[\boldsymbol{c}]_j \leftarrow 1$ // set entry $j$ of $\boldsymbol{c}$ to 1

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

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

For our purposes in this note, we need to translate this set and dictionary of sets into a binary vector and a binary matrix, respectively. To this end, we may use something like function `sets2vecs` in Listing 1. Indeed, if we call this function using

```
vecU, matS = sets2vecs(U, S)
```

and subsequently print `matS`, we obtain

```
[[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]]
```

and recognize this matrix $\boldsymbol{S}$ as an indicator matrix representing our set of sets $\mathcal{S}$.

Now that we have *NumPy* array representations of vector $\boldsymbol{u}$ and matrix $\boldsymbol{S}$ which specify our set cover instance, we next implement the pseudo code in Alg. 2. This is as easy as shown in Listing 2 where the body of function `greedySetCoverV1` is an almost verbatim *NumPy* implementation of the pseudo code. However, we should probably emphasize three points:

First, the second parameter of `greedySetCoverV1` is called `matSt` and represents the transpose $\boldsymbol{S}^\intercal$ of our indicator matrix $\boldsymbol{S}$; we opt to work with the transpose $\boldsymbol{S}^\intercal$ of $\boldsymbol{S}$ because operations on array rows are slightly more efficient than operations on array columns. Second, in our `while` loop, we use the *NumPy* function `any` to test the condition $\boldsymbol{u} \neq \mathbf{0}_n$; `np.any(vecU)` evaluates to `False` unless all elements of array `vecU` are `0`. Third, applied to *NumPy* arrays, the operator `*` behaves like the Hadamard product $\odot$.

Given this simple piece of code, we now can call

```
vecC = greedySetCoverV1(vecU, matS.T)
```

**Listing 1: converting sets to indicator vectors / matrices**

```
def sets2vecs(setU, dctS):
    m, n = len(dctS), len(setU)
    vecU = np.ones(n).astype(np.uint8)
    matS = np.zeros((n,m)).astype(np.uint8)

    for i in dctS:
        matS[list(dctS[i]), i-1] = 1

    return vecU, matS
```

**Listing 2: simple implementation of greedy set cover**

```
def greedySetCoverV1(vecU, matSt):
    m, n = matSt.shape
    vecC = np.zeros(m).astype(np.uint8)

    while np.any(vecU):
        j = np.argmax(np.sum(vecU * matSt, axis=1))
        vecU = vecU * (1 - matSt[j])
        vecC[j] = 1

    return vecC
```

This will produce an array `vecC` which represents an indicator vector $\boldsymbol{c}$ which, in turn, represents a cover $C$ of $\mathcal{U}$. To see that this works, we may

```
print (matS[:,vecC.astype(bool)])
```

and obtain

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

Upon inspection, this result turns out to be an indicator matrix representation of the collection $C = \{\mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$ which covers $\mathcal{U}$. Indeed, this is the exact same result we obtained from our "naïve" implementation of set-based greedy set covering in [1].

There, we also discussed that this result is a bit disappointing because, for our simple example, we actually know that the optimal, i.e. minimum, set cover is given by $C_* = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$.

**Listing 3: improved implementation of greedy set cover**

```
1  def greedySetCoverV2(vecU, matSt):
2      m, n = matSt.shape
3      vecC = np.zeros(m).astype(np.uint8)
4
5      cols = np.where(np.sum(matSt, axis=0) == 1)[0]
6      rows = np.where(matSt[:,cols] == 1)[0]
7      vecC[rows] = 1
8
9      vecU = vecU * (1 - np.sum(matSt[rows], axis=0) > 0)
10
11     while np.any(vecU):
12         j = np.argmax(np.sum(vecU * matSt, axis=1))
13         vecU = vecU * (1 - matSt[j])
14         vecC[j] = 1
15
16     return vecC
```

In order to improve the behavior of our previous "naïve" code, we therefore considered a heuristic which initialized $C$ to contain all those subsets $\mathcal{S}_j \in \mathcal{S}$ which must be part of an optimal solution $C_*$ simply because they contain elements of $\mathcal{U}$ that do not occur in any other subset.

How could we implement that heuristic for our current indicator vector based approach? As easy as in Listing 3.

Remember that, in [1], we had to compute an inverted index to determine the corresponding $\mathcal{S}_j$. However, when working with indicator vectors and matrices, such shenanigans are superfluous. We can simply use

```
cols = np.where(np.sum(matSt, axis=0) == 1)[0]
```

to determine which columns of array `matSt` contain a single 1. Given the indices of these columns, we can then use

```
rows = np.where(matSt[:,cols] == 1)[0]
```

to determine in which rows of array `matSt` these single 1s occur. These rows of `matSt` represent the subsets $\mathcal{S}_j$ which have to be part of an optimal set cover. Hence, if array `vecC` has been initialized to all `0`s, we may now use

```
vecC[rows] = 1
```

to pre-select these necessary components of the solution.

In Listing 3, all of this happens in lines 5–7 and these lines cause no harm if there aren't any subsets with elements not occurring in other subsets. This is because the *NumPy* function `where` in line 5 will return an empty array `cols` if its condition is not met. Should this be the case, then `rows` in line 6 will be an empty array, too, so that the statement in line 7 will have no effect.

Having pre-selected necessary components of an optimal set cover solution, we next need to remove their elements from $\mathcal{U}$. Working with indicator vector / matrix representations, this is again easy. In Listing 3, it happens in line 9 and we note that this piece of code is very *numpythonic* way of realizing the mathematical expression

$$\boldsymbol{u} \leftarrow \boldsymbol{u} \odot \left[\mathbf{1}_n - \min\{1, S\,\boldsymbol{c}\}\right]$$

where min is understood as an element-wise operation. Readers as very much encouraged to verify that this expression really represents the removal of the elements contained in the $\mathcal{S}_j$ from $\mathcal{U}$.

From here on out, the code in Listing 3 is the same as in Listing 2 and thus again an almost verbatim implementation of the pseudo code in Alg, 2.

To examine the behavior of our code enriched by the initialization heuristic, we may call

```
vecC = greedySetCoverV2(vecU, matS.T)
```

which will once again will produce a *NumPy* array `vecC` which represents an indicator vector $\boldsymbol{c}$ which represents a cover $C$ of $\mathcal{U}$. Once again using

```
print (matS[:,vecC.astype(bool)])
```

we now obtain

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

This we recognize as the collection $C_* = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$ which is indeed the optimal solution to our simple, exemplary problem.

Again, we therefore find that it may be beneficial to build domain specific prior knowledge into models or methods [7]. However, just as we did in [1], we must emphasize that the greedy set cover algorithm comes with provable performance guarantees and is therefore an *approximation algorithm*. Our informed initialization of this algorithm does not come with such guarantees and is therefore but a *heuristic*. In other words, there may be situations where our initialization does not improve results.

## 4 SUMMARY AND OUTLOOK

Revisiting the combinatorial set cover problem, we saw that we may think about it in terms of indicator vectors and matrices and that corresponding *NumPy* implementations of "the" greedy algorithm for polynomial time approximations are straightforward.

In upcoming notes, we will revisit set covering once more and show how to rewrite its ILP formulation in (1) as a quadratic unconstrained binary optimization problem (QUBO). We already learned about QUBOs when we worked with Hopfield nets for problem solving. It will therefore be straightforward to apply Hopfield nets to set cover problems.

## 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 Native Python Data Types.* Technical Report. MLAI, University of Bonn.
[2] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN.* IEEE.
[3] IBM ILOG Cplex. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009).
[4] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual.

[5] R.M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computation*, R.E. Miller, J.W. Thatcher, and J.D. Bohlinger (Eds.). Springer.
[6] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[7] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).
[8] P. Welke and C. Bauckhage. 2020. *ML2R Coding Nuggets: Linear Programming for Robust Regression.* Technical Report. MLAI, University of Bonn.
[9] P. Welke and C. Bauckhage. 2020. *ML2R Coding Nuggets: Solving Linear Programming Problems.* Technical Report. MLAI, University of Bonn.