ML2R Coding Nuggets
Greedy Set Cover with Native *Python* Data Types

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

ABSTRACT

In preparation for things to come, we discuss a plain vanilla *Python* implementation of "the" greedy approximation algorithm for the set cover problem.

1 INTRODUCTION

While most of our **ML2R** coding nuggets discuss (implementations of) machine learning algorithms, this note deals with a notorious combinatorial optimization problem and a popular approximation algorithm for its solution. That is, we look at the **set cover problem** and discuss *Python* implementations of *"the" greedy algorithm for polynomial time approximations of set covering*.

Why would this be of interest to machine learners? Remember that we previously studied neural network training without error backpropagation [1, 8]? Soon, we will learn about another derivative-free training method for neural networks and this method will require us to solve set cover problems.

We therefore recall that set covering is one of Karp's original NP-complete problems [5] and, in its simplest unweighted form, is specified as follows:

> Given a set or *universe* $\mathcal{U}$ of n elements and a set or *collection* $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$ of $m \leq 2^n$ subsets $\mathcal{S}_i \subseteq \mathcal{U}$ whose union equals or *covers* $\mathcal{U}$, find the smallest *sub-collection* $C \subseteq \mathcal{S}$ whose union covers $\mathcal{U}$.

Here is a simple example which illustrates what this specification is all about: Consider a universe of $n = 10$ elements, say, the integers from zero to nine

$$\mathcal{U} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{1}$$

Further consider a collection of $m = 6$ subsets of this unviverse

$$\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\} \tag{2}$$

where

$$\mathcal{S}_1 = \{1\} \tag{3}$$
$$\mathcal{S}_2 = \{0, 1, 2\} \tag{4}$$
$$\mathcal{S}_3 = \{3, 4, 9\} \tag{5}$$
$$\mathcal{S}_4 = \{7, 8, 9\} \tag{6}$$
$$\mathcal{S}_5 = \{3, 4, 5, 6\} \tag{7}$$
$$\mathcal{S}_6 = \{0, 2, 4, 6, 8\} \tag{8}$$

Given these ingredients, we recognize a valid instance $(\mathcal{U}, \mathcal{S})$ of the set cover problem because the subsets $\mathcal{S}_i$ of $\mathcal{U}$ in $\mathcal{S}$ cover the universe. More formally, we are dealing with a setting where

$$\bigcup_{\mathcal{S}_i \in \mathcal{S}} \mathcal{S}_i = \mathcal{U}$$

Then what about sub-collections $C \subseteq \mathcal{S}$? Are there any with

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

i.e. are there any sub-collections which cover the universe? Yes, there are even 10 of them, namely

$$C_1 = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$$
$$C_2 = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$$
$$C_3 = \{\mathcal{S}_1, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$
$$C_4 = \{\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5\}$$
$$C_5 = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$
$$C_6 = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5\}$$
$$C_7 = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$
$$C_8 = \{\mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$
$$C_9 = \{\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$
$$C_{10} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$$

Looking at this list, we immediately realize that sub-collection C_1 solves our problem as its size $|C_1| = 3$ is smaller than those of its peers.

Well that was easy, wasn't it? All we had to do was, first, to determine all sub-collections C of $\mathcal{S}$ which cover $\mathcal{U}$ and, second, to search for the smallest one among them.

Alas, while this strategy works for small problems as in our example, it becomes infeasible once we face practically relevant settings. Why? Well, consider this: To truly rest assured that we will find a minimum set cover for $(\mathcal{U}, \mathcal{S})$ with $|\mathcal{S}| = m$, we would have to check all 2^m possible combinations of the subsets in $\mathcal{S}$.

Exhaustive or brute force searches for solutions to a set cover problem are therefore of exponential complexity. For $m = 6$ as in our example, this is still manageable because $2^6 = 64$ are really not that many combinations to check. However, in practice, we typically have to deal with problems where m is much larger. This is problematic because, say, for a still moderate choice of $m = 1000$, we would already have to check $2^{1000} \approx 1.07 \times 10^{301}$ sub-collections.[1] There simply is no classical (super-)computer that could do this in reasonable time.

There are, however, more efficient strategies for approximately solving set cover problems. Some involve QUBOs which could be solved using Hopfield nets or quantum computing [2] and will be studied later. In this present note, we simply consider the greedy algorithm discussed in the next section.

[1] Compare this to the number of atoms in the real physical universe which is reasonably estimated to be about 10^{82} which is nothing compared to 10^{301}.

2 THE GREEDY SET COVER ALGORITHM

Without much further ado, assuming a valid problem instance $(\mathcal{U}, \mathcal{S})$ with $|\mathcal{U}| = n$ and $|\mathcal{S}| = m$, here is "the" greedy algorithm for approximative set covering

---

$C \leftarrow \emptyset$ // 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 $\mathcal{U}$

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

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

This simple algorithm has a runtime polynomial in n and m (details depend on the sophistication of its practical implementation which may involve efficient data structures such as bucket queues [3]). It is also clear that the algorithm will yield a set cover which, alas, may not be optimal. This is indeed typical for any greedy algorithm which, at each stage of its operation, makes the best current choice. The crux is that a best local choice is not necessarily a best global choice.

On the plus side, one can show that results produced by the greedy set cover algorithm are guaranteed to be close to optimal. That is, one can show that, if the optimal cover consists of k sets, then the greedy algorithm will always find a cover consistsing of at most $k \cdot \ln n$ sets [3, 4]. In other words, the greedy set cover algorithm is guaranteed to give an $O(\ln n)$ approximation to the optimal solution and one can even further show that no polynomial time algorithm can do better unless $P = NP$ [4].

Now, polynomial runtime sounds good and logarithmic approximation does not sound too bad. However, the next section will show that the latter can still be disappointing. In addition to a straightforward *Python* implementation of the above pseudo-code, we will therefore also present slightly more elaborate code that can avoid some of the obvious blunders in local decision making.

3 NATIVE *PYTHON* IMPLEMENTATIONS

This section presents *Python* implementations of the greedy set cover algorithm which only require the *Python* standard library. To get an impression for how they perform in practice, we will apply our code to the exemplary problem specified in (1)–(8).

The first design decision we have to make is how to implement a universe of integers. Well, *Python* has an inbuilt data type `set` which provides methods such as `union()`, `intersection()`, or `difference()`. These compute exactly what their names suggest and also come with convenient infix operators, for instance `|`, `&`, and `-` for union, intersection, and difference, respectively. All of this simply suggests to implement $\mathcal{U}$ in (1) as

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

Alas, when it comes to sets of sets such as $\mathcal{S}$ in (2), things are not as straightforward. Since *Python* considers a set to be an unordered collection of *hashable* objects, the inner sets of a set of sets would have to be of type `frozenset`. While this is a minor inconvenience, a bigger drawback is that *Python* sets do not support indexing. We therefore opt to implement sets of sets as *Python* dictionaries whose keys are integers and whose values are sets. This way, $\mathcal{S}$ in (2) can be represented 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 which we point out that our enumeration counter starts at 1.

Given these preparations, an almost verbatim implementation of the above pseudo code is as easy as shown in Listing 1. To pretty print results produced this code snippet, we may use, say

```
C = greedySetCoverV1(U, S)
for j in sorted(C):
    print ('S{0} = {1}'.format(j, C[j]))
```

This results in

```
>>> S1 = {1}
>>> S3 = {9, 3, 4}
>>> S4 = {8, 9, 7}
>>> S5 = {3, 4, 5, 6}
>>> S6 = {0, 2, 4, 6, 8}
```

which corresponds to the sub-collection $C_8 = \{\mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$ we saw in the introduction.

Thus, our result is a covering sub-collection but a bit disappointing nevertheless. This is because we already know that the optimal set cover for our exemplary problem is $C_1 = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$ which is "considerably" smaller than C_8.

So, is there anything we could do to get a better performance out of the greedy algorithm? Yes, there is! We could, for example, use a simple heuristic for a better initialization of the sub-collection C which, so far, is simply initialized to the empty set $\emptyset$.

Let's have another look at our example in (1)–(8) to see what we may mean by a "better" initialization of C.

Note that elements $5, 7 \in \mathcal{U}$ only occur in subsets $\mathcal{S}_5$ and $\mathcal{S}_4$, respectively. Any smallest cover of our universe $\mathcal{U}$ must therefore necessarily contain sets $\mathcal{S}_4$ and $\mathcal{S}_5$. But especially $\mathcal{S}_4$ is not large enough to be picked in the initial stages of greedy selection and, due to possibly unfortunate tie-breaks, may not even be selected quickly during the latter stages. However, this issue could be circumvented if we made use of our knowledge that the rather small set $\mathcal{S}_4$ has

Listing 1: simple implementation of greedy set cover

```
def greedySetCoverV1(U, S):
    # initialize C
    C = {}

    while U:
        # select Sj in S covering most of U
        j, Sj = max(S.items(), key = lambda itm : len(itm[1] & U))

        # add Sj to C
        C[j] = Sj

        # remove element of Sj from U
        U = U - Sj

    return C
```

Listing 2: improved implementation of greedy set cover

```
1  def greedySetCoverV2(U, S):
2      # build inverted index
3      I = {x : [i for i in S if x in S[i]] for x in U}
4
5      # pre-select necessary sets Sj into C
6      C = {j : S[j] for j in [I[x][0] for x in I if len(I[x]) == 1]}
7
8      # remove elements of union of sets Sj in C from U
9      U = U - set().union(*[C[j] for j in C])
10
11
12     while U:
13         j = max(S.keys(), key = lambda k : len(S[k] & U))
14
15         C[j] = S[j]
16
17         U = U - S[j]
18
19     return C
```

to be contained in C. Indeed, we could (and should) initialize C to contain all those subsets $\mathcal{S}_j \in \mathcal{S}$ which must be part of the solution simply because they contain elements of $\mathcal{U}$ that do not occur in any other subset.

How do we determine these $\mathcal{S}_j$? By building an inverted index known from information retrieval [6]! That is, we create a data structure which, for every $x \in \mathcal{U}$, registers in which of the $\mathcal{S}_i \in \mathcal{S}$ it occurs in. Next, we may iterate over this inverted index and determine which of the x point to only one $\mathcal{S}_j$ and add those $\mathcal{S}_j$ to the initial sub-collection C.

In *Python*, the "natural" data type for realizing these ideas is again the dictionary. It is therefore no surprise that it features prominently in our implementation of function `greedySetCoverV2` in Listing 2.

In line 3, we use a dictionary to implement an inverted index I and, in line 6, we use I to populate the initial sub-collection C.

Now that our initial C is not empty anymore, we must not forget to remove all elements in the union of the $\mathcal{S}_j \in C$ from $\mathcal{U}$, that is, we have to compute

$$\mathcal{U} \leftarrow \mathcal{U} \setminus \bigcup_{\mathcal{S}_j \in C} \mathcal{S}_j$$

This happens in very pythonic manner in line 9. Once pre-selection and cleanup are done, we can proceed with the greedy set cover algorithm. The remaining lines of Listing 2 are therefore almost identical to those in Listing 1. However, for the fun of it, we tweaked them a bit. Can you spot what we did and can you explain why our tweaked code still works? And, while we are at it, can you guess which code is slightly more efficient, the one in Listing 1 or the one in Listing 2?

To pretty print outcomes obtained form our more sophisticated greedy set cover implementation, we may again use

```
C = greedySetCoverV2(U, S)
for j in sorted(C):
    print ('S{0} = {1}'.format(j, C[j]))
```

which now results in

```
>>> S2 = {0, 1, 2}
>>> S4 = {8, 9, 7}
>>> S5 = {3, 4, 5, 6}
```

We recognize this as sub-collection $C_1 = \{\mathcal{S}_2, \mathcal{S}_4, \mathcal{S}_5\}$ from the introduction and thus as the true optimal solution to our problem.

All in all, this goes to show that it is a good idea to incorporate domain specific prior knowledge into models or methods [7]. However, we must emphasize the following: While the greedy set cover algorithm comes with provable performance guarantees and is therefore an *approximation algorithm*, our modified, intelligent initialization does not come with such guarantees and is therefore a *heuristic*. Indeed, there may be situations where our intelligent initialization does not improve results. In other words, there may be situations where the additional efforts for building and traversing an inverted index do not amortize.

4 SUMMARY AND OUTLOOK

In this note, we had our first look at the combinatorial set cover problem and discussed straightforward *Python* implementations of "the" greedy algorithm for polynomial time approximations. Working with *Python* sets and dictionaries, coding was a joy! Even our suggestions for a better initialization for greedy set cover did not cause much overhead w.r.t. implementations efforts.

In upcoming notes, we will revisit set covering and discuss alternative implementations of the greedy algorithm. These will make use of indicator vectors and thus involve the use of *NumPy* / *SciPy*.

Once we have familiarized ourselves with these ideas, we will switch gears and see that set cover can be understood as an integer linear programming problem as well as a quadratic unconstrained binary optimization problem (QUBO). We encountered these before when we studied Hopfield nets for problem solving. As we are already familiar with Hopfield nets, it will be easy to apply them for solving set cover problems.

Finally, we will of course get back to our promise in the introduction and discuss where and how set cover problems arise in the context of neural network training.

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. 2018. NumPy / SciPy Recipes for Data Science: Training Neural Networks Without Backpropagation. researchgate.net.
[2] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN*. IEEE.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction to Algorithms* (2nd ed.). MIT Press.
[4] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. 2006. *Algorithms*. McGraw-Hill.
[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] C.D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.
[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] B. Wulff, J. Schuecker, and C. Bauckhage. 2018. SPSA for Layer-Wise Training of Deep Networks. In *Proc. ICANN*.