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 \( S = \{ S_1, S_2, \ldots, S_m \} \) of \( m \leq 2^n \) subsets \( S_i \subseteq \mathcal{U} \) whose union equals or covers \( \mathcal{U} \), find the smallest sub-collection \( C \subseteq S \) 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

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

Further consider a collection of \( m = 6 \) subsets of this universe

\[
S = \{ S_1, S_2, S_3, S_4, S_5, S_6 \}
\]

where

\[
S_1 = \{ 1 \}
\]

\[
S_2 = \{ 0, 1, 2 \}
\]

\[
S_3 = \{ 3, 4, 9 \}
\]

\[
S_4 = \{ 7, 8, 9 \}
\]

\[
S_5 = \{ 3, 4, 5, 6 \}
\]

\[
S_6 = \{ 0, 2, 4, 6, 8 \}
\]

Then what about sub-collections \( C \subseteq S \)? Are there any with

\[
\bigcup_{S_j \in C} S_j = \mathcal{U}
\]

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

\[
C_1 = \{ S_2, S_3, S_5 \}
\]

\[
C_2 = \{ S_1, S_2, S_4, S_3 \}
\]

\[
C_3 = \{ S_1, S_4, S_5, S_6 \}
\]

\[
C_4 = \{ S_2, S_3, S_4, S_5 \}
\]

\[
C_5 = \{ S_2, S_5, S_6 \}
\]

\[
C_6 = \{ S_1, S_2, S_3, S_4, S_5, S_6 \}
\]

\[
C_7 = \{ S_1, S_2, S_4, S_5, S_6 \}
\]

\[
C_8 = \{ S_1, S_3, S_5, S_6 \}
\]

\[
C_9 = \{ S_2, S_3, S_4, S_5, S_6 \}
\]

\[
C_{10} = \{ S_1, S_2, S_3, S_4, S_5, S_6 \}
\]

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 \( S \) 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 have found the smallest one among them, we would have to check all \( 2^m \) possible combinations of the subsets in \( S \).

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

¹Compare this to the number of atoms in the real physical universe which is reasonably estimated to be about \( 10^{85} \) which is nothing compared to \( 10^{301} \).

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.
2 THE GREEDY SET COVER ALGORITHM

Without much further ado, assuming a valid problem instance $(U, S)$ with $|U| = n$ and $|S| = m$, here is “the” greedy algorithm for approximative set covering.

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 consisting of at most $k \cdot \log n$ sets [3, 4]. In other words, the greedy set cover algorithm is guaranteed to give an $O(\log n)$ 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 $U$ in (1) as

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

Alas, when it comes to sets of sets such as $S$ 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, $S$ in (2) can be represented as

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

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

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

which corresponds to the sub-collection $C_8 = \{S_1, S_3, S_4, S_5, S_6\}$ 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 $C_1 = \{S_2, S_3, S_5\}$ 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 $5, 7 \in U$ only occur in subsets $S_3$ and $S_4$, respectively. Any smallest cover of our universe $U$ must therefore necessarily contain sets $S_3$ and $S_4$. But especially $S_4$ 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 $S_4$ has the 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 consisting of at most $k \cdot \log n$ sets [3, 4]. In other words, the greedy set cover algorithm is guaranteed to give an $O(\log n)$ 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 $U$ in (1) as

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

Alas, when it comes to sets of sets such as $S$ 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, $S$ in (2) can be represented as

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

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

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

This results in

```python
>>> 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 $C_8 = \{S_1, S_3, S_4, S_5, S_6\}$ 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 $C_1 = \{S_2, S_3, S_5\}$ 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 $5, 7 \in U$ only occur in subsets $S_3$ and $S_4$, respectively. Any smallest cover of our universe $U$ must therefore necessarily contain sets $S_3$ and $S_4$. But especially $S_4$ 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 $S_4$ has...
to be contained in C. Indeed, we could (and should) initialize C to contain all those subsets S_j ∈ S which must be part of the solution simply because they contain elements of U that do not occur in any other subset.

How do we determine these S_j? By building an inverted index known from information retrieval [6]: That is, we create a data structure which, for every x ∈ U, registers in which of the S_j ∈ S it occurs in. Next, we may iterate over this inverted index and determine which of the x point to only one S_j and add those 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 S_j ∈ C from U, that is, we have to compute

\[ U \leftarrow U \setminus \bigcup_{S_j \in C} S_j \]

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?

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. 01IS1803C). The authors gratefully acknowledge this support.

REFERENCES