ML2R Theory Nuggets
Computational Complexity of Max-Sum Diversification

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

Till Hendrik Schulz
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

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

ABSTRACT

We show how max-sum diversification can be used to solve the k-clique problem, a well-known NP-complete problem. This reduction proves that max-sum diversification is NP-hard and provides a simple and practical method to find cliques of a given size using Hopfield networks.

1 INTRODUCTION

In a recent coding nugget, Bauckhage et al. [2] have stated that the max-sum diversification problem is NP-hard. Rather than to solve the max-sum diversification problem exactly, they instead resorted to using Hopfield networks in order to find an approximate solution.

In this "theory nugget", we provide some background on this issue: Are Bauckhage et al. just too lazy to come up with an exact solution or do they simply apply Hopfield networks no matter the problem? To demonstrate that the answer is no tt both questions, we first elaborate on what it means for a problem to be NP-hard or NP-complete.

NP-hardness, for short means that some problem is at least as difficult to solve as any problem in the *complexity class* NP. This class contains all decision problems whose solutions can be verified in polynomial time but it is unknown if there exist polynomial time algorithms to solve all problems in NP (the so called P vs. NP problem). So how can we provide proof that a problem is as hard as any of the *infinite number of problems* in NP?

We employ an idea that long-time readers of this series should be well aware of: Showing how to solve a given problem A using tools developed for another problem B. For this, we require that the transformation from problem A into problem B can be done in polynomial time in the size of the input data. The added twist is that if we can solve an *NP-complete* problem with tools for other problems, then we can solve *all* problems in the class NP with these tools. Therefore, it suffices to find a polynomial time transformation for a single, selected NP-complete problem using tools for another problem, to show that the latter is NP-hard, i.e. as hard to solve as any problem in NP.

In particular, we will show a *polynomial time reduction* from the well-known NP-complete k-clique-problem [3] to max-sum diversification. Briefly speaking, we provide a way to solve the k-clique-problem by solving a max-sum diversification problem.

Our reduction is constructive and results in a simple way to find cliques of arbitrary size in a given graph using Hopfield networks, as discussed in Bauckhage et al. [2]. Readers who would like to experiment with our code should be passingly familiar with *NumPy* and only need to

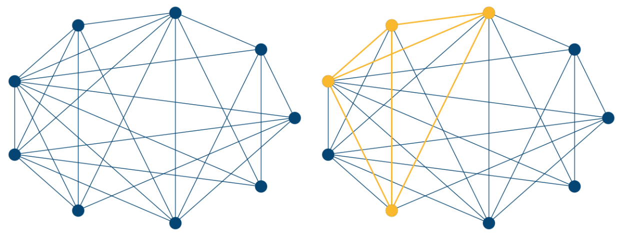

Figure 1: A random graph and a clique of size four in said graph.

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

1.1 Notions

Let us quickly recall the necessary notions: A graph G consists of a set of vertices $V(G)$ and a set of edges $E(G)$, connecting pairs of vertices. An example of a graph is given on the left of Figure 1, where vertices are drawn as points and edges are drawn as lines. A *clique* of size k (or a k-clique for short) is a subset C of the vertices of G with $|C| = k$ such that each pair of vertices in C is connected by an edge. Figure 1 shows a 4-clique in yellow on the right. The k-clique problem is then to decide, given a graph G and an integer k whether there exists a k-clique in G. This problem was among the first 21 problems to be identified as being NP-complete [5] and has since eluded an efficient (i.e. polynomial time) algorithm.

Also recall the max-sum diversification problem: Given some set $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and an appropriate distance measure $d(\cdot,\cdot)$, determine a subset $\mathcal{S} \subset \mathcal{X}$ of size $|\mathcal{S}| = k' < n$ of maximum dispersion. In other words, solve

$$\begin{aligned} \mathcal{S}^* = \underset{\mathcal{S}\subset\mathcal{X}}{\operatorname{argmax}} & \sum_{\boldsymbol{x}_i\in\mathcal{S}} \sum_{\boldsymbol{x}_j\in\mathcal{S}} d(\boldsymbol{x}_i,\boldsymbol{x}_j) \\ \text{s.t.} \quad & |\mathcal{S}| = k'. \end{aligned} \tag{1}$$

2 THEORY

Before showing how we can transform the k-clique problem to max-sum diversification, we will take a brief look at the complexity classes that are relevant for our endeavor. We will also define some requirements for our transformation .

2.1 NP-completeness and NP-hardness

These two *complexity classes* were introduced to formally be able to say "My problem A here is at least as hard as your problem B there" by showing that we can use an algorithm for A to solve

B with some bounded additional effort. We shall formalize "some additional effort" as a transformation of any instance of problem A to an instance of problem B, that can be done in time and space that is *polynomial* in the input size. In particular, we allow some preprocessing, followed by one or more applications of an algorithm for problem B, and some postprocessing to transform the output of the application(s) to an answer to our original problem A. We require the preprocessing and postprocessing steps to take polynomial time, and require that the transformation calls the algorithm for problem B only a polynomial number of times.

Looking at the example that is the topic of this brief article, we need to transform an instance of a k-clique problem, that consists of a graph G and some integer k to an instance of the max-sum diversification problem, i.e., some set $\mathcal{X}$, a distance measure d, and some integer k', and need to show how to answer the k-clique problem using a solution to our max-sum diversification problem.

Definition 2.1. A decision problem $\mathcal{P} \in$ NP is *NP-complete* if any problem in NP can be polynomially reduced[1] to $\mathcal{P}$.

More generally, a (not necessarily decision) problem $\mathcal{P}$ is *NP-hard* if any problem in NP can be polynomially transformed (as above) to $\mathcal{P}$. Note that we have dropped two restrictions here: First, we don't require $\mathcal{P}$ to belong to the class NP any more. Second, we don't even require $\mathcal{P}$ to be a decision problem.

Note that applying a polynomial time algorithm a constant or even polynomial number of times results in a polynomial time algorithm. Hence, if any problem in NP transforms in polynomial time to a NP-complete problem and we give a polynomial transformation from an NP-complete problem to our problem at hand, then *every problem in NP transforms in polynomial time to our problem.* NP-hardness is hence a useful notion to declare some computational problem intractable, or practically infeasible, as nobody is sure whether there exist polynomial time algorithms for all problems in NP.

2.2 A Reduction from the k-Clique Problem to Max-Sum Diversification

We will now show that the max-sum diversification problem (whose output is a set of size k' rather than a true/false decision) is NP-hard. As already mentioned, we will do this by providing a polynomial transformation from the k-clique problem, which is a well-known NP-complete problem and was among the first problems to be identified as NP-complete [3, 5].

This will be rather easy and we will need to define $\mathcal{X}$ and the corresponding metric and argue what the resulting set of maximum dispersion would tell us about the existence of a clique in a given graph.

Given a graph G and a target clique size k, we will look for a subset of the graph's vertices that has maximum dispersion for a hand-crafted distance function. That is, we set $\mathcal{X} = V(G)$ and define

$$\begin{aligned} d : V(G) \times V(G) &\rightarrow \mathbb{R}_{\geq 0} && (2)\\ d(v,w) &= \begin{cases} 0 & \text{if } v = w \\ 1 & \text{if } \{v,w\} \in E(G) \\ \frac{1}{2} & \text{if } \{v,w\} \notin E(G) \end{cases} && (3)\end{aligned}$$

Now, we note that the cliques of size k in G are exactly those subsets $\mathcal{S} \subset V(G)$ of size k that have dispersion of $k^2 - k$: A k-clique is a fully connected subgraph on k vertices. That is, this subgraph, which is induced by vertices in $\mathcal{S}$, contains exactly $\binom{k}{2} = \frac{k(k-1)}{2}$ edges. Hence, all pairs of vertices $v \neq w$ in the clique must fulfill $d(v,w) = 1$ and $d(v,v) = 0$. The resulting dispersion for this case is then

$$\sum_{\boldsymbol{x}_i \in \mathcal{S}} \sum_{\boldsymbol{x}_j \in \mathcal{S}} d(\boldsymbol{x}_i, \boldsymbol{x}_j) = 2\frac{k(k-1)}{2} = k^2 - k \ .$$

Note that each edge is counted twice in the summation. In contrast, non-clique subsets of size k have strictly smaller dispersion, as there exists, by definition, at least one pair of vertices $v \neq w$ that is not connected by an edge and hence $d(v,w) = \frac{1}{2}$.

As as result, it follows that

$$\max_{\mathcal{S} \subset \mathcal{X}, |\mathcal{S}| = k} \sum_{\boldsymbol{x}_i \in \mathcal{S}} \sum_{\boldsymbol{x}_j \in \mathcal{S}} d(\boldsymbol{x}_i, \boldsymbol{x}_j) = k^2 - k$$

if and only if there is a clique of size k in G. Hence, we can solve the k-clique problem by solving the max-sum dispersion problem for $|\mathcal{S}| = k$ and then check whether the maximizer $\mathcal{S}^*$ has dispersion $k^2 - k$.

This already concludes our little transformation and we are almost done. It remains to think about the time and space complexity of this approach. Graphs are typically given in a sparse format, that is, the size of the input is proportional to $|V(G)| + |E(G)|$. We note that the distance matrix

$$[\boldsymbol{D}]_{ij} = d(v_i, v_j)$$

is quadratic in $|V(G)|$, and can be created in time that is proportional to its size. Hence the preprocessing can be done in polynomial time. Checking whether the maximizer $\mathcal{S}^*$ has dispersion $k^2 - k$ can be done by summing over all pairs of elements in $\mathcal{S}^*$, which is quadratic in k (as $|\mathcal{S}^*| = k$).

It is interesting to note that d is a metric: it is symmetric, fulfills the triangle inequality $d(x,z) \leq d(x,y) + d(y,z)$ for all $x, y, z \in V(G)$, and the identity $d(x,y) = 0 \Leftrightarrow x = y$. Thus, we have actually shown that the max-sum diversification problem is NP-hard even if the distance function is restricted to be a metric.

2.3 Homework

We have now seen that max-sum diversification is NP-hard by giving a polynomial time reduction of the k-clique problem to max-sum diversification. We would like to encourage the interested reader to show that the complement variant

$$\begin{aligned} \mathcal{S}^* = \operatorname*{argmin}_{\mathcal{S} \subset \mathcal{X}} & \sum_{\boldsymbol{x}_i \in \mathcal{S}} \sum_{\boldsymbol{x}_j \in \mathcal{S}} d(\boldsymbol{x}_i, \boldsymbol{x}_j) \\ \text{s.t.} \quad & |\mathcal{S}| = k. \end{aligned} \quad (4)$$

[1]A polynomial reduction is a special form of polynomial transformation that we don't discuss in detail. The interested reader is referred to e.g. Korte and Vygen [6].

Listing 1: transformation of adjacency matrix to distance matrix

```python
def reductionDistance(matA):
    matD = matA
    matD[matD < 1] = 0.5
    np.fill_diagonal(matD, 0)
    return matD
```

of this problem, that we call *min-sum diversification*, for now, is also NP-hard. One way of achieving this would be to consider the k-Stable Set (also called k-Independet Set) problem, that asks, for a given graph G and integer k whether there exists a subset C of vertices of size k such that no two vertices in C are connected by an edge, which is another well-known NP-complete problem.

3 PRACTICE

We now turn to the practical implementation of our transformation of the k-clique problem to max-sum diversification in *NumPy* . The transformation itself is of little practical use if we do not use it to solve the k-clique problem practically. Luckily, we already know a way to come up with feasible but approximate solutions for a max-sum diversification in the form of Hopfield networks as shown in Bauckhage et al. [2]. We will use these previous results to find answers to the k-clique problem that have a one-sided error. In particular, if our algorithm returns "yes" to the question whether there is a clique of size k, this answer is always correct. If, however, the algorithm returns "no", there is still a chance that in fact there is a clique of size k.

We will now provide an implementation that neatly extends the existing code [2]. It is, however, not the most efficient: When working with real-world graphs, we are typically in the situation that these graphs are *sparse*, i.e., that they have much less edges than they could have. Hence, one would typically avoid working with an adjacency matrix that is stored as a *dense NumPy* array. While there are ways around this[2], we focus on ease of exposition and note that even this "inefficient" implementation fulfills our requirements for a polynomial time reduction, as mentioned in the previous section.

We will start with a graph provided in form of its adjacency matrix A, where A contains entry 1 at position ij if there is an edge between vertices v_i and v_j and a 0 otherwise. The adjacency matrix of the graph in Figure 1 can be defined as follows:

```
matA = [[0. 1. 0. 0. 1. 1. 1. 1. 0.]
        [1. 0. 1. 0. 1. 0. 0. 1. 1.]
        [0. 1. 0. 1. 1. 1. 1. 1. 1.]
        [0. 0. 1. 0. 1. 1. 1. 1. 0.]
        [1. 1. 1. 1. 0. 1. 1. 1. 1.]
        [1. 0. 1. 1. 1. 0. 1. 1. 1.]
        [1. 0. 1. 1. 1. 1. 0. 0. 0.]
        [1. 1. 1. 1. 1. 1. 0. 0. 0.]
        [0. 1. 1. 0. 1. 1. 0. 0. 0.]]
```

It can be read row-by-row and tells us, for example, that the first vertex is connected to the second, fifth, sixth, seventh, and eighth vertex.

A straightforward implementation of our reduction distance function d from the previous section is shown in Listing 1. It alters the adjacency matrix by setting each entry that was zero (i.e., where there is no edge) to $\frac{1}{2}$. As this also includes the diagonal entries, we have to reset them to zero afterwards, as $d(v_i, v_i)$ for all $v_i \in V(G)$ is required to be zero by our definition. Having this, we can define all input that we require for the max-sum diversification (e.g. for $k = 4$) by

```python
k = 4
matD = reductionDistance(matA)
```

We can now go on and use Hopfield networks to solve the k-clique problem (aproximately!) in our graph, as was shown in Bauckhage et al. [2]. To this end, we have to slightly alter their initialization method, as it did not only initialize the parameters of the Hopfield network, but also computed the distance matrix. Listing 2 shows the slightly altered initialization method, which now accepts a distance matrix as input. Hence

```python
matW, vecT = hnetInitParameters(matD, k)
vecS = -np.ones(n)
```

initializes a Hopfield network to solve the max-sum diversification problem corresponding to our k-clique problem and

```python
vecS = hnetRunRnd(vecS, matW, vecT)
```

finds a solution and produces an output.

We now tend to the postprocessing step. We need to transform the output of max-sum diversification which is a set that we assume to be encoded as a binary vector $s \in \{-1, 1\}^n$ to a yes/no answer for our question "Is there a clique of size $k = 4$ in our graph?" Listing 3 shows how this can be achieved in *NumPy* . Line 2 converts the $\{-1, 1\}$ vector vecS to a $\{0, 1\}$ vector that we can use to obtain the distance matrix that corresponds to the subgraph on the vertices selected by the Hopfield network. As this vector has a small chance to not exactly contain the target number k of vertices, we check this condition in Line 3. Lines 4 and 5 project the distance matrix

Listing 2: initializing parameters W and θ of a Hopfield net

```python
def hnetInitParameters(matD, k, l=None):
    _, n = matD.shape

    mat1 = np.ones((n,n))
    vec1 = np.ones(n)
    matI = np.eye(n)

    if l is None:
        l = 20*n

    matP =  l * mat1 - matD
    vecP = -l * 2*k * vec1

    matQ = 0.25 * (matP - l * matI)
    vecQ = 0.50 * (matP @ vec1 + vecP)

    matW = -2 * matQ
    vecT = vecQ

    return matW, vecT
```

Listing 3: transformation Hopfield net result to answer the k-clique-problem

```python
1 def reductionCheck(matD, vecS, k):
2     mask = np.where(vecS>0, True, False)
3     resultSize = print(np.sum(mask))
4     matS = matD[mask, :]
5     matS = matS[:, mask]
6     return (resultSize == k) and (np.sum(matS) > k * k - k - 0.25)
```

[2]Interested readers might want to check out `scipy.sparse` for useful data structures.

matD onto the distance matrix of the vertices that were selected by our solver to max-sum diversification. The final line sums up all elements in this projected matrix and checks if it is equal to $k^2 - k$. It returns true if the latter is the case and if the size of our result set is indeed k. To avoid numerical issues, we note that the largest value that can appear if $\boldsymbol{s}$ does not describe a clique is $k^2 - k - \frac{1}{2}$, and we avoid the rather brittle check for equality of floating point numbers. We can apply the postprocessing by

```
print(reductionCheck(matD, vecS, k))
```

If we are lucky, we obtain a feature vector, e.g.,

```
vecS = [-1. -1.  1. -1.  1.  1. -1. -1.  1.]
```

which corresponds to the clique that is drawn in yellow on the right hand side of Figure 1.

However, we should expect that this is not always the case and may revert to repeated iterations of our approach. If we repeat our randomized algorithm for Hopfield networks and one iteration finds a clique, we are sure that a clique exists, hence we can answer 'yes'. If our algorithm never finds a clique after a reasonable amount of iterations, we may conclude that there is no clique of size $k = 4$. In this case, there is the chance that we have made a mistake.

This strategy can be implemented by a simple loop

```
clique = None
for i in range(50):
    vecS = -np.ones(n)
    vecS = hnetRunRnd(vecS, matW, vecT)
    hasClique = reductionCheck(matD, vecS, k)
    if hasClique:
        clique = vecS
        break

print(clique)
```

4 CONCLUSION

In this short "theory nugget", we have shown that max-sum diversification is NP-hard, i.e., it is at least as hard to solve this as any problem in the class NP. The polynomial time reduction that we have used to do this is constructive and allows us to transform a k-clique problem into a max-sum diversification problem. Using code from a recent coding nugget [2] we can hence apply Hopfield networks to find cliques of a given size.

Many problems that appear as subtasks in machine learning and data mining are NP-hard, e.g. in the context of k-means clustering [1], decision tree learning [4], in the context of graph mining [7, 10, 11], or graph kernels [8]. Showing that some problem is NP-hard is an accepted reason to resort to efficient approximate solutions, as finding an efficient exact algorithm might not be possible. Inexact solutions with one sided error can often be used to great advantage, as they allow some reasoning about the overall results of combined algorithms in data mining scenarios, such as finding subsets [10] or supersets [7, 9] of some set that we are actually interested in.

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] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. 2009. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning* 75, 2 (2009), 245–248. https://doi.org/10.1007/s10994-009-5103-0

[2] Christian Bauckhage, Fabrice Beaumont, and Sebastian Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Max-Sum Diversification.* Technical Report. MLAI, University of Bonn.

[3] Michael. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[4] Laurent Hyafil and Ronald L. Rivest. 1976. Constructing Optimal Binary Decision Trees is NP-Complete. *Inform. Process. Lett.* 5, 1 (1976), 15–17. https://doi.org/10.1016/0020-0190(76)90095-8

[5] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations (The IBM Research Symposia Series).* Plenum Press, New York, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[6] Bernhard Korte and Jens Vygen. 2018. *Combinatorial Optimization* (sixth ed.). Springer.

[7] Till Hendrik Schulz, Tamás Horváth, Pascal Welke, and Stefan Wrobel. 2018. Mining Tree Patterns with Partially Injective Homomorphisms. In *European Conference on Machine Learning and Knowledge Discovery in Databases ECML PKDD Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 11052. Springer, 585–601. https://doi.org/10.1007/978-3-030-10928-8_35

[8] Till Hendrik Schulz, Tamás Horváth, Pascal Welke, and Stefan Wrobel. 2021. A Generalized Weisfeiler-Lehman Graph Kernel. (2021). arXiv:cs.LG/2101.08104v1

[9] Pascal Welke. 2017. Simple Necessary Conditions for the Existence of a Hamiltonian Path with Applications to Cactus Graphs. *CoRR* abs/1709.01367 (2017). http://arxiv.org/abs/1709.01367

[10] Pascal Welke, Tamás Horváth, and Stefan Wrobel. 2019. Probabilistic and Exact Frequent Subtree Mining in Graphs Beyond Forests. *Machine Learning* 108, 7 (2019), 1137–1164. https://doi.org/10.1007/s10994-019-05779-1

[11] Pascal Welke, Florian Seiffarth, Michael Kamp, and Stefan Wrobel. 2020. HOPS: Probabilistic Subtree Mining for Small and Large Graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD.* ACM, 1275–1284. https://dl.acm.org/doi/10.1145/3394486.3403180