ML2R Coding Nuggets
Hopfield Nets, Bipartition Clustering, and the Kernel Trick

Christian Bauckhage
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

Fabrice Beaumont
Computer Science
University of Bonn
Bonn, Germany

Sebastian Müller
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

ABSTRACT

We revisit Hopfield nets for bipartition clustering and show how to invoke the kernel trick to increase robustness and versatility. Our corresponding *NumPy* code is short and simple.

1 INTRODUCTION

Previously [3], we saw that Hopfield nets can, in principle, cluster the n elements $\boldsymbol{x}_i \in \mathbb{R}^m$ of a *centered* data set $\mathcal{X}$ into two disjoint salient clusters. In other words, we saw that Hopfield nets can solve bipartition clustering problems.

In order to obtain this result, we rewrote the objective of $k = 2$ means clustering in terms of an energy minimization problem

$$\boldsymbol{s}^* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} -\tfrac{1}{2}\, \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} \tag{1}$$

Here, the bipolar vectors $\boldsymbol{s}$ over which to minimize denote all possible states of a Hopfield network whose weight matrix is $\boldsymbol{W}$. The entries of this weight matrix are

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \boldsymbol{x}_i^\top \boldsymbol{x}_j & \text{otherwise} \end{cases} \tag{2}$$

so that $\boldsymbol{W}$ is symmetric and hollow ($\boldsymbol{W} = \boldsymbol{W}^\top$ and $\operatorname{diag}[\boldsymbol{W}] = \boldsymbol{0}$). This guarantees that, if the Hopfield net evolves in an asynchronous manner, i.e. if its neurons update one at a time, it will eventually settle in a stable state of (locally) minimum energy [13]. If we refer to this state as $\boldsymbol{s}^\infty$, we can consider its individual entries s_i^∞ as cluster membership indicators which partition the given data into two clusters, namely

$$\begin{aligned} \mathcal{X}_1 &= \left\{ x_i \in \mathcal{X} \;\middle|\; s_i^\infty = +1 \right\} \\ \mathcal{X}_2 &= \left\{ x_i \in \mathcal{X} \;\middle|\; s_i^\infty = -1 \right\} \end{aligned} \tag{3}$$

Why did we use the phrase "in principle" when we said that Hopfield nets can solve bipartition clustering problems? Because [3] also presented an example where our approach failed!

This shortcoming of our clustering Hopfield nets is due to our ansatz of deriving their energy functions from the $k = 2$ means clustering objective. Since k-means itself cannot always produce clusters which human observers would deem reasonable [1, 2], our corresponding Hopfield nets have this characteristic, too.

However, in [3] we promised that there is a "simple" remedy and we now get back to this promise. To make a long story short, section 2 will show how to kernelize the energy function in (1). This idea leads to more robust and more versatile Hopfield nets for clustering and section 3 will show that the underlying mathematical concepts are easily implemented in plain vanilla *NumPy*.

In what follows, we assume that readers know about Mercer kernels and the kernel trick and how these are used in machine learning. Those who would like to experiment with our code snippets should be familiar with *NumPy* and *SciPy* [16] and need to

```
import numpy as np
import numpy.random as rnd
import scipy.spatial as spt
```

2 THEORY

In the following, we suppose we were given an $m \times n$ data matrix

$$\boldsymbol{X} = \left[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \right] \tag{4}$$

whose columns $\boldsymbol{x}_i \in \mathbb{R}^m$ are to be clustered into two salient clusters.

Remeber that the corresponding Hopfield energy minimization problem we derevied in [3] crucially hinged on the assumption that the data in $\boldsymbol{X}$ are centered. We therefore briefly recall how to center a data matrix: Letting $\boldsymbol{1}$ denote the n-dimensional vector of all ones, we can compute the column mean of $\boldsymbol{X}$ as

$$\boldsymbol{\mu} = \tfrac{1}{n}\, \boldsymbol{X} \boldsymbol{1} \tag{5}$$

Next, we can use the vector of all ones in another capacity and compute the outer product matrix

$$\boldsymbol{\mu} \boldsymbol{1}^\top = \left[\boldsymbol{\mu}, \boldsymbol{\mu}, \ldots, \boldsymbol{\mu} \right] \tag{6}$$

whose n columns all are copies of $\boldsymbol{\mu}$. Given this matrix, we can now easily subtract $\boldsymbol{\mu}$ from *all* columns of $\boldsymbol{X}$ to obtain a *centered* data matrix

$$\boldsymbol{X}_c = \boldsymbol{X} - \boldsymbol{\mu}\, \boldsymbol{1}^\top \tag{7}$$

For this matrix, it is easy to see that its column mean is $\boldsymbol{0}$ or, in other words, that its columns are centered at $\boldsymbol{0}$. To be prudent, we also remark that, if we wanted to, we could reverse this centering (if the mean vector $\boldsymbol{\mu}$ has been stored) as follows

$$\boldsymbol{X} = \boldsymbol{X}_c + \boldsymbol{\mu}\, \boldsymbol{1}^\top \tag{8}$$

In this sense, our assumption of zero mean data does not lead to a loss of generality: To cluster the given data we center it, run a corresponding Hopfield network to determine appropriate cluster membership indications, and then de-center the data and cluster the resulting original data according to the respective indicators.

While we are at it, we point out another crucial fact about data centering. Plugging (5) into (7), we obtain

$$\boldsymbol{X}_c = \boldsymbol{X} - \tfrac{1}{n}\, \boldsymbol{X} \boldsymbol{1} \boldsymbol{1}^\top = \boldsymbol{X} \left[\boldsymbol{I} - \tfrac{1}{n}\, \boldsymbol{1} \boldsymbol{1}^\top \right] \equiv \boldsymbol{X}\, \boldsymbol{J} \tag{9}$$

where

$$\boldsymbol{J} = \boldsymbol{I} - \tfrac{1}{n}\, \boldsymbol{1} \boldsymbol{1}^\top \tag{10}$$

is called the **centering matrix** because it centers the columns of matrix $\boldsymbol{X}$ at $\boldsymbol{0}$.

Now recall that our considerations as to bipartition clustering of centered data led to the following **quadratic unconstrained binary optimization (QUBO) problem**

$$\boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\operatorname{argmin}} -\left\| \boldsymbol{X}_c \boldsymbol{s} \right\|^2 = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\operatorname{argmin}} -\boldsymbol{s}^\top \boldsymbol{X}_c^\top \boldsymbol{X}_c \, \boldsymbol{s} \tag{11}$$

This is the point, where we will begin our present discussion for real and show how to get a more robust QUBO formulation for the bipartition clustering problem.

To begin with, we use (9) and write our minimization objective in terms of $\boldsymbol{X}$ and $\boldsymbol{J}$, namely

$$-\boldsymbol{s}^\top \boldsymbol{X}_c^\top \boldsymbol{X}_c \, \boldsymbol{s} = -\boldsymbol{s}^\top \boldsymbol{J}^\top \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{J} \, \boldsymbol{s} = -\boldsymbol{s}^\top \boldsymbol{J} \, \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{J} \, \boldsymbol{s} \tag{12}$$

Note that the last step is possible because the centering matrix is symmetric so that $\boldsymbol{J}^\top = \boldsymbol{J}$.

Now, observe that matrix $\boldsymbol{X}^\top \boldsymbol{X}$ in (12) is a **Gram matrix** whose individual entries are given by

$$\left[\boldsymbol{X}^\top \boldsymbol{X}\right]_{ij} = \boldsymbol{x}_i^\top \boldsymbol{x}_j \tag{13}$$

With respect to the data, our minimization problem thus exclusively depends on inner product of pairs of data points so that we can invoke the **kernel trick**. That is, we can replace the Gramian by a kernel matrix $\boldsymbol{K}$ where

$$\left[\boldsymbol{K}\right]_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{14}$$

and $K : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ is a **Mercer kernel**. In our practical examples in section 3, we will consider the following Mercer kernels

- the linear kernel

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^\top \boldsymbol{x}_j \tag{15}$$

- the polynomial kernel

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\boldsymbol{x}_i^\top \boldsymbol{x}_j + c\right)^d \tag{16}$$

- the Gaussian Kernel

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\frac{1}{2\sigma^2} \| \boldsymbol{x}_i - \boldsymbol{x}_j \|^2} \tag{17}$$

Recall that, depending on the choice of kernel, kernels can introduce a lot of flexibility when solving optimization, learning, or analysis problems because what we implicitly do when working with Mercer kernels is to consider inner products

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\varphi}_i^\top \boldsymbol{\varphi}_j \tag{18}$$

Here, $\boldsymbol{\varphi}_i \equiv \boldsymbol{\varphi}(\boldsymbol{x}_i)$ where $\boldsymbol{\varphi} : \mathbb{R}^m \to \mathbb{H}$ is some generally unknown transformation from the data space into a potentially infinite dimensional feature space. Hence, if we introduce a whole kernel matrix, we may think of this as being given a feature matrix

$$\boldsymbol{\Phi} = \left[\boldsymbol{\varphi}_1, \boldsymbol{\varphi}_2, \ldots, \boldsymbol{\varphi}_n\right] \tag{19}$$

and computing the kernel matrix as

$$\boldsymbol{K} = \boldsymbol{\Phi}^\top \boldsymbol{\Phi} \tag{20}$$

With respect to our bipartitioning clustering problem in (11), there is therefore a catch! Since we required our original data points to be centered at $\boldsymbol{0} \in \mathbb{R}^m$ in order for our approach to work, we now have to make sure that the transformed data are are centered at $\boldsymbol{0} \in \mathbb{H}$ so that our kernelized approach works correctly.

Luckily, this is easily accomplished, because, if we invoke the kernel trick

$$-\boldsymbol{s}^\top \boldsymbol{J} \, \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{J} \, \boldsymbol{s} \quad \to \quad -\boldsymbol{s}^\top \boldsymbol{J} \, \boldsymbol{K} \, \boldsymbol{J} \, \boldsymbol{s} \tag{21}$$

we implicitly have

$$-\boldsymbol{s}^\top \boldsymbol{J} \, \boldsymbol{K} \, \boldsymbol{J} \, \boldsymbol{s} = -\boldsymbol{s}^\top \boldsymbol{J} \, \boldsymbol{\Phi}^\top \boldsymbol{\Phi} \, \boldsymbol{J} \, \boldsymbol{s} \tag{22}$$

$$= -\boldsymbol{s}^\top \boldsymbol{J}^\top \, \boldsymbol{\Phi}^\top \boldsymbol{\Phi} \, \boldsymbol{J} \, \boldsymbol{s} \tag{23}$$

$$= -\boldsymbol{s}^\top \left(\boldsymbol{\Phi} \, \boldsymbol{J}\right)^\top \left(\boldsymbol{\Phi} \, \boldsymbol{J}\right) \boldsymbol{s} = -\boldsymbol{s}^\top \boldsymbol{\Phi}_c^\top \boldsymbol{\Phi}_c \, \boldsymbol{s} \tag{24}$$

In other words, the presence of the centering matrix $\boldsymbol{J}$ in our objective function automatically guarantees that the feature space transformed data are centered, too.

With respect to our kernel matrix, which we practically compute using (14) rather than (18), this is to say that

$$\boldsymbol{K}_c = \boldsymbol{J} \, \boldsymbol{K} \, \boldsymbol{J} \tag{25}$$

is a *centered* kernel matrix. All in all, we can thus safely kernelize our original problem such that it becomes

$$\boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\operatorname{argmin}} -\boldsymbol{s}^\top \boldsymbol{K}_c \, \boldsymbol{s} \tag{26}$$

A final remark regarding centering in feature space pertains to a practical rather than a theoretical issue. Observe that

$$\boldsymbol{K}_c = \boldsymbol{J} \, \boldsymbol{K} \, \boldsymbol{J} \tag{27}$$

$$= \left[\boldsymbol{I} - \tfrac{1}{n} \boldsymbol{1}\boldsymbol{1}^\top\right] \boldsymbol{K} \left[\boldsymbol{I} - \tfrac{1}{n} \boldsymbol{1}\boldsymbol{1}^\top\right] \tag{28}$$

$$= \left[\boldsymbol{K} - \tfrac{1}{n} \boldsymbol{1}\boldsymbol{1}^\top \boldsymbol{K}\right] \left[\boldsymbol{I} - \tfrac{1}{n} \boldsymbol{1}\boldsymbol{1}^\top\right] \tag{29}$$

$$= \boldsymbol{K} - \tfrac{1}{n} \boldsymbol{1}\boldsymbol{1}^\top \boldsymbol{K} - \tfrac{1}{n} \boldsymbol{K}\boldsymbol{1}\boldsymbol{1}^\top + \tfrac{1}{n^2} \boldsymbol{1}\boldsymbol{1}^\top \boldsymbol{K} \, \boldsymbol{1}\boldsymbol{1}^\top \tag{30}$$

While this expanded expression for the centered kernel matrix may look daunting, we will see below that it allows for a very efficient way of implementing $\boldsymbol{K}_c$ in *NumPy*.

Finally, in order to turn the kernelized QUBO in (26) into a Hopfield energy minimization problem, we note that we can also write it as

$$\boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\operatorname{argmin}} -\boldsymbol{s}^\top \boldsymbol{H} \, \boldsymbol{s} - \boldsymbol{s}^\top \boldsymbol{D} \, \boldsymbol{s} \tag{31}$$

where the two matrices $\boldsymbol{H}$ and $\boldsymbol{D}$ are given by

$$[\boldsymbol{H}]_{ij} = \begin{cases} 0 & \text{if } i = j \\ \left[\boldsymbol{K}_c\right]_{ij} & \text{otherwise} \end{cases} \tag{32}$$

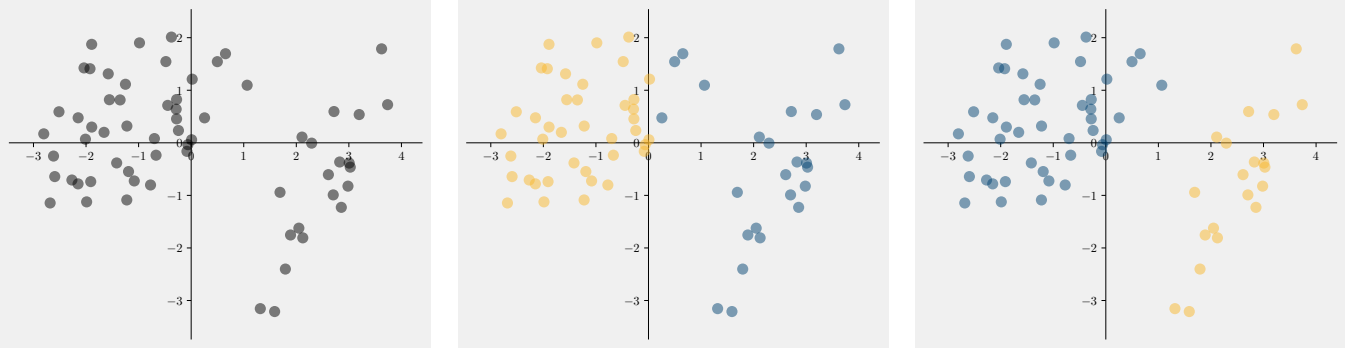
$$[\boldsymbol{D}]_{ij} = \begin{cases} \left[\boldsymbol{K}_c\right]_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{33}$$

Now, since $\boldsymbol{H}$ is a hollow matrix (i.e. has a diagonal of all zeros) and $\boldsymbol{D}$ is a diagonal matrix for which $\boldsymbol{s}^\top \boldsymbol{D} \, \boldsymbol{s} = const$, we may introduce a weight matrix

$$\boldsymbol{W} = 2 \, \boldsymbol{H} \tag{34}$$

to rewrite (26) as a Hopfield energy minimization problem of the form in (1), namely

$$\boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\operatorname{argmin}} -\tfrac{1}{2} \, \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{s} \tag{35}$$



(a) $n = 64$ data points $\boldsymbol{x}_i \in \mathbb{R}^2$ (b) clusters obtained with $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^\intercal \boldsymbol{x}_j$ (c) clusters obtained with $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^\intercal \boldsymbol{x}_j + 1)^3$

Figure 1: The data in this figure was previously discussed in [3]. There, we explained why our original Hopfield nets for bipartition clustering cannot uncover the two apparent clusters. Our original approach is equivalent to a linear kernel Hopfield net which also performs poorly on this data. However, when using non-linear kernels such as the polynomial kernel, a kernel Hopfield net can correctly identify the two salient clusters.

Listing 1: computing kernel matrices

```
def computeLinKernelMatrix(matX):
    return matX.T @ matX


def computePolyKernelMatrix(matX, d=3, c=1.):
    return (matX.T @ matX + c)**d


def computeGaussKernelMatrix(matX, sigma=1.):
    matD = spt.distance.pdist(matX.T, 'sqeuclidean')
    matD = spt.distance.squareform(matD)
    return np.exp(-0.5 / sigma**2 * matD)
```

Listing 2: centering kernel matrices

```
def centerKernelMatrixV1(matK):
    m, n = matK.shape
    matJ = np.eye(n) - 1/n * np.ones((n,n))
    return matJ @ matK @ matJ


def centerKernelMatrixV2(matK):
    m, n = matK.shape
    csum = np.sum(matK,axis=1).reshape(n,1)
    rsum = np.sum(matK,axis=0).reshape(1,n)
    tsum = np.sum(matK)
    return matK - 1/n * csum - 1/n * rsum + 1/n**2 * tsum


def centerKernelMatrixV3(matK):
    m, n = matK.shape
    rsum = np.sum(matK,axis=0).reshape(1,n)
    csum = rsum.reshape(n,1)
    tsum = np.sum(rsum)
    return matK - 1/n * csum - 1/n * rsum + 1/n**2 * tsum
```

3 PRACTICE

Given the above theory, we now discuss *NumPy* implementations of Hopfield nets for kernel bipartition clustering.

Throughout, we assume that we are given a data matrix $\boldsymbol{X}$ such as in (4) which we represent as a 2D *NumPy* array `matX` whose numbers of rows and columns can be determined using

```
m, n = matX.shape
```

To initialize the hollow weight matrix $\boldsymbol{W}$ of a kernelized Hopfield net for bipartition clustering, we first compute an appropriate kernel matrix $\boldsymbol{K}$. For the examples in Fig. 1(a), Fig. 1(b), Fig. 2(a), and Fig. 2(b), we used

```
matK = computeLinKernelMatrix(matX)
matK = computePolyKernelMatrix(matX, 3, 1)
matK = computeGaussKernelMatrix(matX, 0.5)
matK = computeGaussKernelMatrix(matX, 0.5)
```

to obtain linear, polynomial, and Gaussian kernel matrices. The respective *NumPy* methods in Listing 1 are direct implementations of the kernel functions in (15)–(17).

Next, we have to center the resulting kernel matrix. To this end, we may use either of the functions in Listing 2. Function `centerKernelMatrixV1` implements the expression on the right hand side of (27). The two functions `centerKernelMatrixV2` and `centerKernelMatrixV3` are numpythonic implementations of the right hand side of (30). The former is more readable but less efficient, the latter is more efficient but less readable. In either case, our implementations of (30) run much faster than the implementation of (27), especially if n is large. Hence, we use

```
matKc = centerKernelMatrixV3(matK)
```

to compute a centered kernel matrix $\boldsymbol{K}_c$. Given the centered kernel matrix, the weights of our Hopfield network can be computed as

```
matW = 2 * matKc
np.fill_diagonal(matW, 0)
```

Having initialized the network's weight matrix, we next set its initial state $\boldsymbol{s}$. For the examples in in Fig. 1(a), Fig. 1(b), and Fig. 2(a) we used the random initialization

```
vecS = 2 * rnd.binomial(n=1, p=0.5, size=n) - 1
```

which we already discussed in [3]. For the example in Fig. 2(b), we went with

```
vecS = np.ones(n)
```

Listing 3: greedily running a clustering Hopfield net

```
def signum(x):
    return np.where(x >= 0, +1, -1)


def hnetRunGreedy(vecS, matW, tmax=100):
    for t in range(tmax):
        dltE = vecS * (matW @ vecS)
        updt = np.argmin(dltE)

        vecS[updt] = vecS[updt] * signum(dltE[updt])

    return vecS
```

Given arrays `matW` and `vecS`, we can now run our Hopfield net for kernel bipartition clustering. Just as we did in [3], we again use `hnetRunGreedy` in Listing 3

```
vecS = hnetRunGreedy(vecS, matW, tmax=100)
```

and note that parameter `tmax` may have to be set to higher values, if the number n of given data points is large. Once this computation has terminated, we finally partition matrix $\boldsymbol{X}$ into two smaller matrices $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ which represent the two sought after clusters. Again, we proceed as in [3] and use

```
matX1 = matX[:,vecS>0]
matX2 = matX[:,vecS<0]
```

The results shown throughout this note exemplify that kernelized Hopfield nets for bipartition clustering are versatile and can indeed overcome the limitations of the more naïve networks in [3].

4 CONCLUSION

This note demonstrated that Hopfield networks can cluster data into two salient clusters even if the geometry of the data is challenging.

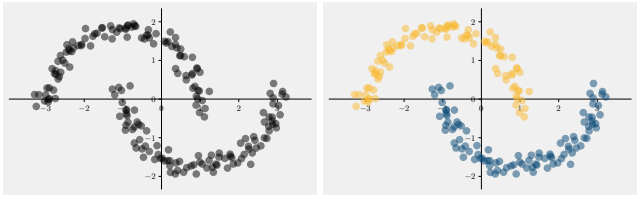
We again considered m-dimensional Euclidean data points and showed how to kernelize the QUBO formulation of bipartition clustering we derived in [3]. Using the kernel trick made our approach more robust and versatile However, as it is common with kernelized algorithms, kernel parameters, initial states, and runtimes of our extended Hopfield nets may need careful tuning to the problem at hand.

On the plus side, our approach in this note also applies to non-numeric data as long as we can define appropriate kernel functions. Interesting examples can be found in the area of text processing [9–12] and we will discuss further details soon.

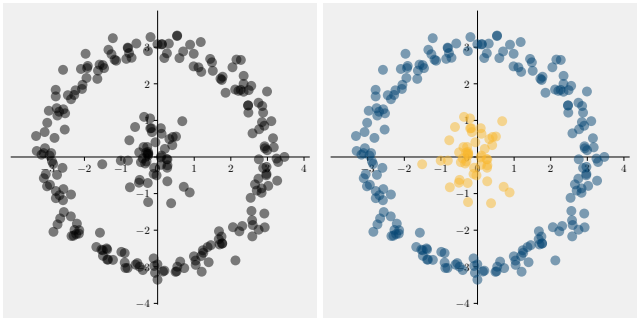
Because of the (conceptual) connection between Hopfield nets and adiabatic quantum computing [7], our ideas in this note also point to robust quantum algorithms for clustering [4, 5, 14, 15] or prototype learning [6, 8]. Details as to how to implement them will be discussed in upcoming coding nuggets.

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.



(a) two moons data and clustering result



(b) nested blob and circle data and clustering result

Figure 2: Hopfield networks for kernel bipartition clustering can identify clusters that are not linearly separable.

REFERENCES

[1] C. Bauckhage. 2015. Lecture Notes on Data Science: k-Means Clustering Is Gaussian Mixture Modeling. DOI: 10.13140/RG.2.1.3033.2646.
[2] C. Bauckhage. 2015. Lecture Notes on Data Science: k-Means Clustering Minimizes Within Cluster Variances. DOI: 10.13140/RG.2.1.1292.4649.
[3] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Bipartition Clustering.* Technical Report. MLAI, University of Bonn.
[4] 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.
[5] C. Bauckhage, C. Ojeda, R. Sifa, and S. Wrobel. 2018. Adiabatic Quantum Computing for Kernel k=2 Means Clustering. In *Proc. KDML-LWDA.*
[6] C. Bauckhage, N. Piatkowske, R. Sifa, D. Hecker, and S. Wrobel. 2019. A QUBO Formulation of the k-Medoids Problem. In *Proc. KDML-LWDA.*
[7] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN.* IEEE.
[8] C. Bauckhage, R. Sifa, and S. Wrobel. 2020. Adiabatic Quantum Computing for Max-Sum Diversication. In *Proc. SDM.* SIAM.
[9] M. Beeksma, M. van Gompel, F. Kunneman, L. Onrust, B. Regnerus, D. Vinke, E. Brito, C. Bauckhage, and R. Sifa. 2018. Detecting and Correcting Spelling Errors in High-quality Dutch Wikipedia Text. *Computational Linguistics in the Netherlands J.* 8 (2018), 122–137.
[10] E. Brito, B. Georgiev, D. Domingo-Fernandez, C.T. Hoyt, and C. Bauckhage. 2019. RatVec: A General Approach for Low-dimensional Distributed Vector Representations via Rational Kernels. In *Proc. KDML-LWDA.*
[11] E. Brito, R. Sifa, and C. Bauckhage. 2017. KPCA Embeddings: An Unsupervised Approach to Learn Vector Representations of Finite Domain Sequences. In *Proc. KDML-LWDA.*
[12] V. Gupta, S. Giesselbach, S. Rüping, and C. Bauckhage. 2019. Improving Word Embeddings Using Kernel PCA. In *Proc. Workshop on Representation Learning for NLP @ ACL.*
[13] J. Hopfield. 1982. Neural Networks and Physical Systems with Collective Computational Abilities. *PNAS* 79, 8 (1982).
[14] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD.*
[15] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. LWDA.*
[16] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).