Computational Statistics
https://doi.org/10.1007/s00180-024-01535-9

SOFTWARE ARTICLE

BARMPy: Bayesian additive regression models Python package

Danielle Van Boxel[1,2]

Received: 6 April 2024 / Accepted: 23 July 2024

Abstract
We make Bayesian additive regression networks (BARN) available as a Python package, `barmpy`, with documentation at https://dvbuntu.github.io/barmpy/ for general machine learning practitioners. Our object-oriented design is compatible with SciKit-Learn, allowing usage of their tools like cross-validation. To ease learning to use `barmpy`, we produce a companion tutorial that expands on reference information in the documentation. Any interested user can `pip install barmpy` from the official PyPi repository. `barmpy` also serves as a baseline Python library for generic Bayesian additive regression models.

Keywords Machine learning · Python · MCMC · Software

1 Introduction

We implement Bayesian additive regression networks (BARN) as a software package, `barmpy` (for Bayesian Additive Regression Models in Python). This algorithm is another approach to the general regression problem of finding some function, $f(x_i)$, to approximate a noisy relationship, $y_i = u(x_i) + \epsilon_i$, where $\epsilon_i \sim N(0, \sigma)$ is a noise term and $u(x_i)$ represents some true underlying function. BARN works by sampling from a posterior distribution on ensembles of neural networks (NNs), similar to Bayesian Additive Regression Trees (BART) (Chipman et al. 2010), but with a different backbone. We cover some of the necessary practical mathematical background in Sect. 2, but here we focus more on implementation designs for the library.

New methods in machine learning and broader mathematics arise all the time, but they are not always readily accessible to data science practitioners (Patel et al. 2008). Part of the explosion of machine learning was the development of libraries

✉ Danielle Van Boxel
vanboxel@arizona.edu

[1] Applied Math GIDP, University of Arizona, Tucson, AZ, USA

[2] Data Diversity Lab, University of Arizona, Tucson, AZ, USA

Published online: 04 August 2024

like Scikit-Learn (Pedregosa et al. 2011), TensorFlow (Abadi et al. 2015), and Keras (Chollet et al. 2015). Such tools not only freed data scientists from having to implement machine learning algorithms manually, they also provide detailed documentation with examples. This kind of broad support is the difference between a research algorithm and an accessible library.

Making `barmpy` accessible means more than publishing code. We integrate tightly with existing popular Python machine learning libraries like Scikit-Learn (and TensorFlow as an alternative), described in more detail in Sect. 3. This includes following those libraries' best practices regarding complete documentation and example tutorials. We even take the algorithmic improvement beyond Scikit-Learn by implementing custom model callbacks, such as for early stopping, detailed in Sect. 3.5. To show `barmpy`'s utility and limitations, we conduct benchmark testing and a small case study. Part of this is computation time information, as such metrics are often a concern for machine learning practitioners. We note that computation time is itself an accessibility issue; large language models like ChatGPT are not generally trainable to users with typical hardware resources (Ouyang et al. 2022). Making a new method like BARN usable in terms of speed, capability, and understandability makes `barmpy` more than an algorithm.

2 Mathematical background

Whereas we describe the methodology of BARN in detail in Van Boxel (2024), here we review key points, as relevant to potential users, of BARN as implemented in `barmpy`. Recall that the related method, BART, is made of an ensemble of decision trees which sum to the prediction (Chipman et al. 2010). While structurally similar to a random forest (Breiman 2001), BART is distinct in that we train it by sampling from the posterior distribution of trees using Markov Chain Monte Carlo (MCMC) (Chipman et al. 1998). By carefully setting transition, prior, and evidence probability functions, BART can calculate an MCMC acceptance ratio for accepting a changed tree. After many iterations over all the trees, we realize convergence to the desired posterior. BARN works similarly to this but uses neural networks in the ensemble rather than decision trees. From an algorithmic (and software design) perspective, however, we need to define the MCMC steps of model proposal, transition, and posterior.

BARN, like BART, is an ensemble of smaller models. BARN, however, uses an ensemble of neural networks instead of decision trees. To train these neural networks, we sample from a posterior distribution *on the space of neural networks*. For simplicity, consider a single network, M. The posterior distribution on M involves a Bayesian prior, $P(M)$, which incorporates our prior beliefs about the network (e.g. how many neurons should it have?). The posterior also includes the evidence, $P(Y|M, X)$, which is the likelihood of seeing the target data, Y, given the model, M, and input data, X. Intuitively, this measures how well M fits the given data. To sample from this posterior, we create a Markov Chain that has this posterior as the stationary distribution. As in Chipman et al. (1998), creating such an MCMC involves proposing a new model, M', from the old one using some transition probability,

$q(M, M')$. Then we compute the posterior probability of both old and new models to find the acceptance ratio, $A(M, M')$ in Eq. (1). This ratio is the probability that we accept M' as the replacement for M. After many iterations, this process converges to the desired posterior stationary distribution.

$$A(M, M') = min\left(1, \frac{q(M|M')P(Y|M', X)P(M')}{q(M', M)P(Y|M, X)P(M)}\right) \tag{1}$$

Consider the transition probabilities, which in a way, encapsulate the model proposal. In BARN, as in BART, we apply Gibbs sampling to ensembling, proposing, training, and potentially accepting a single neural network at a time. BARN allows only 2 transitions: adding one neuron or subtracting one neuron. Therefore, we capture this in a single parameter, p, the probability of adding a neuron to the existing network. This proposes the new network size, but to fully specify that network, we also need model weights. That involves training the network with standard optimization techniques, again as described in Van Boxel (2024). The final proposed new network in a step is then the result of this procedure.

As noted, to compute the MCMC acceptance ratio, we also need the posterior probability of the old and new networks. Note that in BART, this calculation is the closed form of an integral over the weights (Chipman et al. 2010). In BARN, this is an approximation, so we only need a closed form for the prior and evidence. Our default prior depends only on the number of neurons and uses a discrete Poisson distribution. And finally, the evidence component of the BARN posterior for model k is the likelihood of the target residual value, $P(R_k|M_k, X)$, where $R_k = Y - \sum_{j \neq k} M_j(X)$ and $M_j(X)$ is the jth neural network in the ensemble applied to input X. This likelihood assumes a normal distribution of errors (and sampled σ value for each MCMC step). The prior times this evidence gives us the posterior, which then multiplied by the transition proposal probability, contributes to the acceptance ratio. This provides all the key aspects of a BARN model.

Consider a small example with three networks in the ensemble: M_1, M_2, and M_3. We wish to propose a possible replacement for M_3, so first we fix M_1 and M_2. Then we subtract their contribution to the model prediction to obtain the residual, $R_3 = Y - M_1(X) - M_2(X)$. Now we transition M_3 to potential new model, M'_3, by using the transition probability, $q(M_3, M'_3)$. In the example Fig. 1, M'_3 has one less neuron than M_3. With the architecture of M'_3 specified, we find the approximate maximum likelihood estimate for the weights, $w_3 \approx \arg\max_{w_3} P(R_3|M_3, X)$ by using gradient descent or other suitable optimization routine (Kingma and Ba 2014). After training M'_3, we are ready to compute the posterior of both M_3 and M'_3 on the latest residual and use those posteriors to obtain the acceptance ratio, $A(M_3, M'_3)$. In the example in Fig. 1, $A(M_3, M'_3) = 0.42$, and the random number generated is less than this, so we accept the new assignment $M_3 \leftarrow M'_3$. Fixing M_3, we can then turn to M_1 and repeat this process. We keep cycling through each network in the ensemble, accepting or rejecting transitions, until we converge to the stationary posterior distribution.

As briefly mentioned, `barmpy` users can supply their own parameters or methods for inputs like the prior distribution. Section 3.4 describes in detail how to

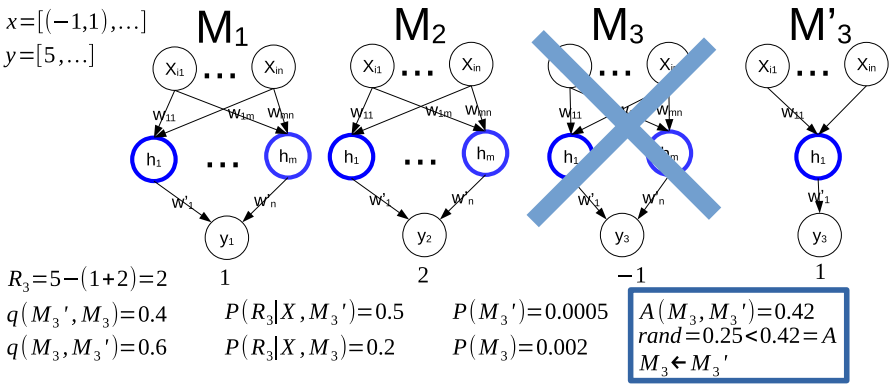

Fig. 1 Toy example showing acceptance of new M_3' proposed transition as an MCMC step toward convergence to the posterior distribution in BARN

use Scikit-Learn's own cross-validation with `barmpy`. And because the library is object-oriented, data scientists familiar with Python and Scikit-Learn can quickly subclass our `BARN` or `NN` classes to their own specifications. A good use case for this would adapt BARN for binary classification by changing a few methods that control the MCMC process. So even if `barmpy` basic methods and defaults are not directly applicable, they can serve as a starting point for rapid prototyping.

3 Library features

In developing `barmpy`, we seek not only to implement BARN for regression and classification, but also to create an accessible library for generic BART or BARN-like algorithms. Part of that means weighing different programming language options, not only for ease of our own coding, but for future open-source development as well. Additionally, we explore some of the choices behind the overall object-oriented design. This design includes tight integration with Scikit-Learn (Pedregosa et al. 2011), which multiplies `barmpy`'s capabilities. Next, we note the importance of not only documentation, but fully worked tutorials for practitioners. And finally, we discuss some practical concerns like distribution on PyPi (PyPi Maintainers 2023) and GitHub (Escamilla et al. 2022). Our goals in developing `barmpy` go beyond merely implementing accurate statistics.

3.1 Related software

We choose to develop `barmpy` in Python, even though most BART packages are written in R. The original researchers into BART are responsible for multiple packages in R, including variants (Chipman et al. 2022; McCulloch et al. 2024), generally available on the Comprehensive R Archive Network (CRAN) (Hornik 2012). Other BART-derived methods like MOTR-BART also develop in R (Prado et al. 2021). While BARN is related to BART, we intend `barmpy` to be of general use not

only to statisticians, but to professional data scientists as well. And these data scientists almost overwhelmingly choose Python, partly due to its broader ecosystem of libraries, documentation, and developer tools (Srinath 2017). Therefore, we develop `barmpy` in Python to reach the data scientists where they are.

There are a few BART implementations in Python, which we briefly review here. `BARTPy` uses a high-level SciKit-Learn style (Pedregosa et al. 2011) model interface, though it hasn't seen activity in several years (Coltman 2022). A more up-to-date BART library in Python is PyMC-BART (Quiroga et al. 2022). PyMC-BART, however, focuses on probabilistic programming within the PyMC framework Abril-Pla et al. (2023), which lacks tight SciKit-Learn integration. A library which implements BART under active maintenance and with SciKit-Learn principles is `ISLP` (James et al. 2023). This is actually the code companion to a textbook, but it uses SciKit-Learn classes like `BaseEstimator` in a direct imperative style of programming familiar to those with SciKit-Learn and NumPy (Harris et al. 2020) experience. We keep these Python BART libraries' strengths and limitations in mind when developing `barmpy`.

SciKit learn is particularly relevant because it implements a huge variety of machine learning algorithms with a standardized object-oriented application program interface (API) (Pedregosa et al. 2011). It has modules for many popular machine learning approaches such as linear regression, random forests, neural networks, and more. Further, SciKit-Learn provides variable transformation routines which are especially helpful when including a model within a SciKit-Learn pipeline with pre- and post-processing. While other Python libraries like `statsmodels` provide detailed statistics in a manner similar to R packages (Seabold and Perktold 2010), Scikit-Learn focuses on practical usage and extensibility. For example, `statsmodels` requires some workarounds to enable using models for prediction on new data, but every Scikit-Learn `Estimator` has a built-in `predict` method for exactly this. Further, Scikit-Learn implements a method for cross-validated hyperparameter tuning; anything that subclasses a Scikit-Learn `Estimator` may tune this way. Much like `ISLP`, `barmpy` inherits from these SciKit-Learn classes. And therefore, `barmpy` automatically benefits from the entire Scikit-Learn ecosystem.

3.2 Regression and binary classification

As our BARN ensemble is made of many small neural networks, our fundamental class is `NN`, which uses the Scikit-Learn primitive, `MLPRegressor` (for "Multi-Layer Perceptron", i.e. a fully connected neural network). A short example of training BARN is in Snippet 1. Our class includes helper methods to compute the various MCMC log-likelihood and prior probabilities given a network, as this is done on a per-network level under Gibbs sampling. There are also routines to quickly save or load results, and handle the weight donation to other neural networks when transitioning. Building an ensemble of these `NN` objects, we have the general `BARN` class. This is more than a list of `NN` objects; it includes parameters to customize the algorithm priors. Further, it has a critical method, `BARN.fit`,

which implements the full BARN procedure with Gibbs sampling. While not parallelized (as models must be fit sequentially), it does avoid duplication of computation by caching residual values and only updating them with the networks that have changed. This turns an $O(N)$ operation, where N is the number of networks in the ensemble, into an $O(1)$ (i.e. constant time) operation. And like `NN`, this class has some helper methods for features like Monte Carlo batch means analysis and built-in visualization of key results using `matplotlib` (Hunter 2007). From a user perspective, they need only instantiate a `BARN` object, setup the Bayesian parameters, and supply data for training.

BARN for classification works similar to regression. Currently, the library supports binary classification with targets encoded as $y_i \in \{0, 1\}$. In code, one swaps `BARN_bin` for `BARN`. After training, `BARN_bin`'s predictions lie in (0, 1), and represent the model's predicted probability of the true class being 1, as in a probit model. If desired, one can directly produce the model z-scores which equate to these probabilities. For usage, we again offer a small example in Snippet 1. Internally, `BARN_bin` inherits from the same base class as `BARN`, `BARN_base`, which implements most of the sampling logic. Because of this, `BARN_bin` uses the same `MLPRegressor` (not `MLPClassifier`) for each component of the ensemble. Options like prior distribution mean value are identical, making it easy to switch between these BARN modes for different problems as needed.

Snippet 1: BARN in regression and classification using `sklearn`

```
from sklearn import datasets
import sklearn.metrics
from barmpy.barn import BARN, BARN_bin
import numpy as np

# Regression problem
db = datasets.load_diabetes()
model = BARN(num_nets=10,
             random_state=0,
             warm_start=True,
             solver='lbfgs',
             l=1)
model.fit(db.data, db.target)
pred = model.predict(db.data)
print(sklearn.metrics.r2_score(db.target, pred))

# Classification problem
bc = datasets.load_breast_cancer()
bmodel = BARN_bin(num_nets=10,
             random_state=0,
             warm_start=True,
             solver='lbfgs',
             l=1)
bmodel.fit(bc.data, bc.target)
pred = bmodel.predict(bc.data)
print(sklearn.metrics.classification_report(bc.target, np.round(pred)))
```

We note that BARN does not yet support training on data like counts nor multiclass classification. There exist extensions to BART which cover these cases, but they require some additional care when computing the MCMC acceptance ratio

(Linero 2022). Adapting those types of data to BARN will require additional analysis and implementation.

Our BARN implementation comes with reasonable defaults for ease of use by scientific practitioners. We recommend the NN growth transition probability be set to $p = 0.4$. This mildly encourages the algorithm to test relatively small networks, mitigating the chance of a single network dominating the ensemble. Similarly, we advise setting the network size prior distribution mean to $\lambda = 1$ or another small value to again encourage networks to be individually weak learners. If one is using BARN for pure architecture search (i.e. only a single network in the ensemble), however, then λ should be larger to accommodate more complexity. Additionally, users can supply their own generic probability mass function if they wish to control the prior more carefully. The number of networks in the ensemble, as mentioned, is itself a settable parameter. We default to 10 as this balances ensembling to improve generalization with increased computation time from additional network training. For the neural network training itself, parameters like learning rate and weight regularization penalties are more problem-dependent. We suggest learning rate $lr = 0.01$ and L1L2 regularization $r = 0.01$, but note that users should experiment with these particular settings. Additional details on tweakable parameters are available in the BARN documentation (Van Boxel 2023).

3.3 Improving software accessibility

To further ease usage and additional development, we have adopted several more general software engineering principles. First, being a Python library, we naturally distribute `barmpy` as a package on PyPi.org (PyPi Maintainers 2023). This enables hassle-free installation for new users. Next, as noted earlier, we maintain all development history in a Git repository on Github. This repository also logs issues, which can be reported bugs or plans for future features that users can explore. To ensure functional correctness even in the face of seemingly unrelated changes, we run a suite a unit tests with every commit to the `main` branch. Each test runs a small chunk of code using `barmpy` as a library and compares the output to a known good result. When a test fails, we can see exactly where and if this needs attention. In addition to assisting with development, unit tests also act as examples for new users. Beyond such rigorous tests, we also wrote and deployed a complete walkthrough via an R Markdown (Baumer and Udwin 2015) script. This walkthrough describes a problem end-to-end, from generating data to running BARN and interpreting the results. Further, because this is in R Markdown, users can run the code chunks themselves (by `knit`ting the script or copying it into a Python terminal). Example output is provided, as in Fig. 2, for users to verify their results, thereby ensuring they can learn how to apply `barmpy` on their own. Finally, when users or developers need more details, they can review the low-level documentation we developed using Python's Sphinx library (Brandl

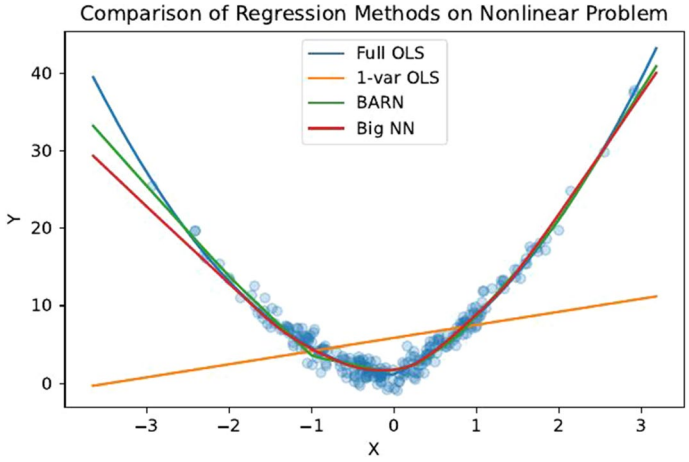

Fig. 2 Example tutorial output showing BARN may outperform OLS and even a much larger neural network. Note this example uses no cross-validated tuning

2021). This documentation is in part automatically generated from the BARN Python docstrings themselves, though we include additional mathematical information at an appropriate level, such as for doing cross-validation on BARN with Scikit-Learn. The source documents are part of the repository itself, but they are also online as a Github Pages website (Van Boxel 2023). These tools enable new users and developers to quickly understand, use, and improve on the BARN algorithm for data science projects.

3.4 Cross-validated tuning via Scikit-Learn

BARN is implemented as a `sklearn` class, meaning we can use standard `sklearn` methods like `GridSearchCV` to tune the hyperparameters for the best possible result. Note that each additional parameter choice increases the computation time multiplicatively, so one should be mindful when considering the number of possible hyperparameter values.

All arguments to `BARN` which accept different values can be tuned this way. In Snippet 2, we show an example that tunes the prior distribution mean value parameter, λ. Also, when using a method like `RandomizedSearchCV`, one should be careful to supply appropriate distributions. Here, l takes discrete values, so we specify a discrete Poisson probability distribution to sample from. Note, however, that this distribution is only for cross-validation sampling of the prior parameters, not for BARN to use in internal MCMC transitions.

Snippet 2: BARN CV Tuning Example using `sklearn`

```
from sklearn import datasets
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from barmpy.barn import BARN
db = datasets.load_diabetes()
scoring = 'neg_root_mean_squared_error'

# exhaustive grid search
## first make prototype with fixed parameters
bmodel = BARN(num_nets=10,
          random_state=0,
          warm_start=True,
          solver='lbfgs')
## declare parameters to exhaust over
parameters = {'l': (1,2,3)}
barncv = GridSearchCV(bmodel, parameters,
                refit=True, verbose=4,
                scoring=scoring)
barncv.fit(db.data, db.target)
print(barncv.best_params_)

# randomized search with distributions
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import poisson
## first make prototype with fixed parameters
bmodel = BARN(num_nets=10,
          random_state=0,
          warm_start=True,
          solver='lbfgs')
## declare parameters and distributions
parameters = {'l': poisson(mu=2)}
barncv = RandomizedSearchCV(bmodel, parameters,
                refit=True, verbose=4,
                scoring=scoring, n_iter=3)
barncv.fit(db.data, db.target)
print(barncv.best_params_)
```

3.5 Early stopping approaches

In machine learning, even when training some model over many iterations, it is common to stop the process early under some conditions. Typically, these involve checking some error metric against held-out validation data (Gençay and Qi 2001). If the metric fails to improve, then one stops training in order to avoid overfitting to training data. Given the MCMC-based training process of BARN, however, there are several possibilities for such metrics.

In addition to the standard approach of checking validation error, we explore alternatives measuring stability in the posterior. As the MCMC posterior is some probability distribution, we can estimate it from our samples once we reach convergence. If this estimate is stable, then we infer that convergence has been reached and we can stop. One reasonable metric is the earth-mover distance (also known as the one-Wasserstein metric (Solomon et al. 2014)) from one estimate of the distribution to the next. In our case, this means evaluating the distribution of the number of neurons in each network of the ensemble and setting a change threshold. Though some researchers explored similar ideas (Durmus and Moulines 2015), they focused more on mixing rate. A similar though simpler heuristic is to simply check how many

proposed model transitions BARN accepted in the previous iteration. If the model has converged, then the error rates are already low and it will be relatively difficult to dislodge an existing model. Therefore, so long as a model continues to accept transitions, it advances to the next MCMC iteration, as Snippet 3 details in Python. By default, this method stops if less than 20% of the networks in the ensemble transitioned. A final, more rigorous alternative is to check not just the stationarity of the MCMC calculation, but the complete convergence of batch means as well. The relative fixed-width stopping rule constructs a t-stat to check recent convergence of relative batch means, implying stationarity (Flegal and Gong 2015). These are all relatively quick to implement, so we make them available to users as a model callback.

Snippet 3: "Not Trans Enough" Early Stopping Callback

```
@staticmethod
def trans_enough(self, check_every=None, skip_first=0, ntrans=None):
    '''
    Stop early if fewer than `ntrans` transitions

    Skip the first `skip_first` iters without checking
    '''
    i = self.i
    # only check every so many, default every 10%
    if check_every is None:
        check_every = max(self.n_iter//10, 1)
    # not an iteration to stop on
    if i == 0 or i % check_every != 0:
        return None
    if i < skip_first:
        return None
    # default minimum transitions to continue is 20%
    if ntrans is None:
        ntrans = max(self.num_nets//5,1)
    # compare most recent count of accepted transitions
    if self.ntrans_iter[i-1] < ntrans:
        raise JackPot # early stopping flag
```

To assist users developing their own callback, or for general model evaluation, `barmpy` internally tracks a few variables at each MCMC iteration. Snippet 3 uses `self.ntrans_iter`, which counts how many proposed transitions were accepted for each iteration. Additionally, let $\phi_i = \sqrt{\sum_i \frac{\left(Y_i - \sum_k M_k(X_i)\right)^2}{N}}$ be the root mean squared error (RMSE) over the N validation data points. Then `self.phi` contains this validation RMSE for each iteration. Similarly, `self.actual_num_neurons` tracks the number of neurons in each network in the ensemble, and `self.sigma` maintains a list of the current σ estimate of the noise. These arrays of variables provide information about a given BARN run's training process and serve as inputs to custom callbacks.

In practice, however, we expect most data scientists to use the more common check on the current model validation error than these other methods like Snippet 3. In various evaluations, we found most methods provide similar results (about a 20% reduction in computation time), with validation error anecdotally being the most stable. We still expose them, not only for their nominal purpose, but also as examples of generic custom model callbacks that can affect the training procedure.

4 Evaluation

To see in what contexts `barmpy` is most useful, we analyze its error and timing metrics in different situations. We focus on both a small case study with data from an active problem in biology as well as a review of computation time on different synthetic data sets.

Before discussing these results, we quickly note how Van Boxel (2024) covers a broad range of real and synthetic data sets to show where BARN is most effective. In particular, their analysis of specific synthetic data sets provides some of the most insight. Without repeating the analysis there, we note that BARN does better than other methods on problems where there is a strong functional nonlinear relationship like Friedman $F2$ or $F3$. So BARN may be practically appropriate as an approximation to a complex system that cannot be easily directly modeled.

4.1 Case study: isotope modeling

While Van Boxel (2024) runs BARN on a wide variety data sets, we focus here on one case study on clumped isotope paleothermometry. The modeling problem itself is to predict carbonate clumped isotope thermometry, Δ_{47}, as a function of temperature (Eiler and Schauble 2004). This is a calibration process; in practice one uses Δ_{47} as a surrogate for historical temperature that was not measured (and therefore something invertible like OLS is typically preferable). The ecological details are beyond the scope of this paper, but there are various studies on this topic (Eiler and Schauble 2004; Román Palacios et al. 2022; Petersen et al. 2019). A recent study (Román Palacios et al. 2022) demonstrated the effectiveness of a Bayesian least squares approach to this data. Such a method uses a linear model as in OLS, but employs priors on the estimated parameter values informed by earlier studies. BARN also uses priors but on the model structure (by affecting the size of learned networks) rather than the parameters directly.

As this data set is in a single variable, we can visually inspect the relationship between temperature and Δ_{47}. Figure 3 plots Δ_{47} against the inverse of squared temperature, showing a strong linear relationship, though with some spread. Scientists training models on this data need to be able to invert the model (i.e. change $\Delta_{47} = f(T)$ into $T = f^{-1}(\Delta_{47})$) to predict historical temperatures. So even if BARN outperforms other approaches, it will likely not replace linear methods on this particular problem. We focus on BARN's performance on the data as an area of active research.

In Fig. 4 we inspect results on this "isotope" data set, and we see that BARN performs well relative to the other methods. Note that the output has been rescaled from the original for this calculation. BARN produces very similar results to OLS (test RMSE about 0.298, 4% less than the next-best method's error). And, BARN works nearly as well without cross-validated tuning as with (less than 0.1% test RMSE difference). This performance, even without tuning, does require an increased computational cost, as we shall see in Sect. 4.2. We caution, again,

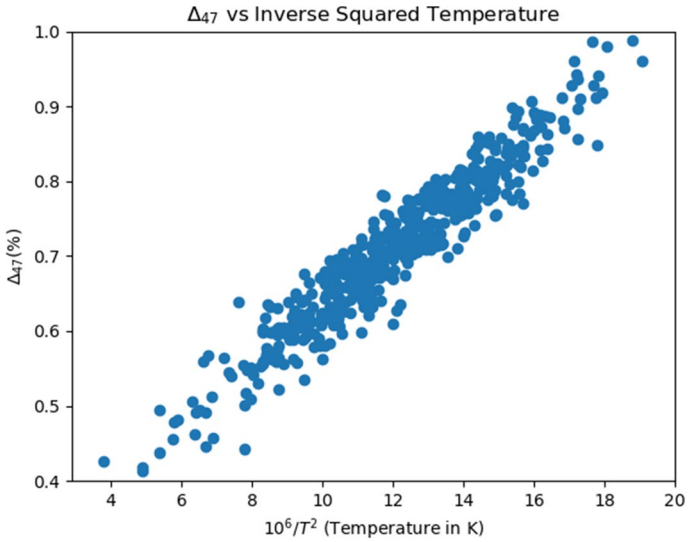

Fig. 3 Apparent linear relationship between temperature and Δ_{47}

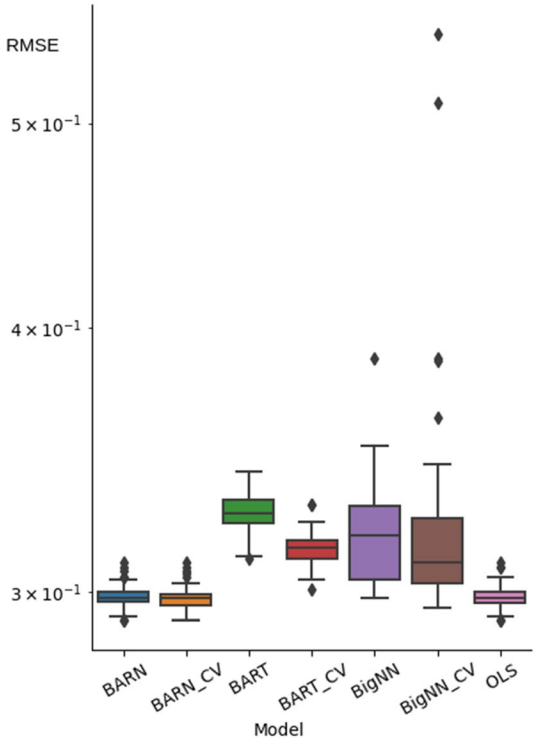

Fig. 4 Absolute RMSE boxplot of various methods on the isotope data set. BARN (with or without tuning) and OLS have similar profiles, while other methods are significantly worse (but still rather accurate; note log scale)

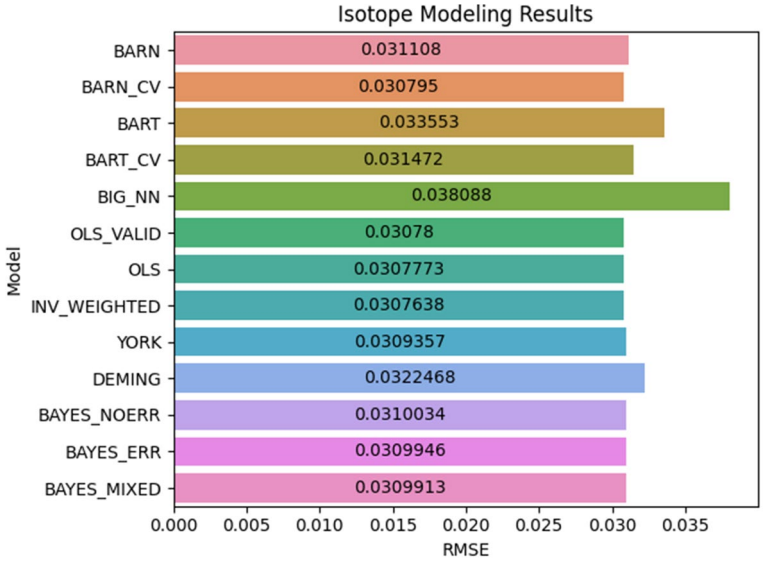

Fig. 5 Testing RMSE point estimates (only a single split performed) on Δ_{47} testing data for various methods. BARN performs similarly to linear methods (Román Palacios et al. 2022) even when a big NN and BART perform significantly worse. Note that methods used in this study (top six models) reserve 25% of the training data for validation (hence why "OLS_VALID" is separate from "OLS")

that our BARN analysis here is for demonstration only, as this particular problem requires invertibility.

The state-of-the art in this area uses Bayesian linear models. We show Fig. 5 to quickly compare existing methods with our approaches on a specific data split of interest (hence why there are only point estimates of the error). Further, we note that these are on the *original data scales*, hence why all the errors appear so much smaller than in Fig. 4. BARN appears to be in the same class of error levels as the best linear approaches. BARN's error is only about 1% higher than the best method (and even closer for tuned BARN). This is especially interesting because the other nonlinear methods we tested (the big NN and BART) actually perform significantly *worse* than OLS and BARN when not using cross-validated tuning. It is possible that BARN is able to simplify to an OLS-like model that is appropriate for this problem (which has a single explanatory variable) where other nonlinear methods would require additional training data for such a reduction. This demonstrates some of the adaptability and broad applicability of BARN.

4.2 Computational costs of BARN

While Van Boxel (2024) described BARN's performance in terms of error, here we consider the computational cost of running it. There can be a trade-off here. Some problems, like targeted display advertising (Shah et al. 2020), benefit from speed of computation (at the expense of accuracy); others, like medical imaging (Aggarwal

Table 1 Mean training time in seconds over 40 trials of various methods on different data sets

Dataset	BARN	BARN CV	BART	BART CV	Big NN	Big NN CV	OLS
cali small	70.315	3051.940	0.821	240.575	0.149	29.332	0.002
concrete	76.898	3418.810	0.494	148.01	0.066	20.202	0.002
crimes	270.220	14175.800	0.905	258.248	0.183	30.963	0.011
diabetes	16.364	2231.910	0.177	73.3536	0.035	9.663	0.002
fires	33.866	3302.060	0.168	78.0964	0.036	10.802	0.002
isotope	7.521	526.695	0.350	136.935	0.032	6.532	0.001
mpg	30.834	1923.640	0.170	66.4122	0.031	8.533	0.002
random	21.660	1117.350	0.574	146.616	0.039	15.839	0.002
wisconsin	58.132	3734.430	0.120	48.0919	0.043	7.964	0.002

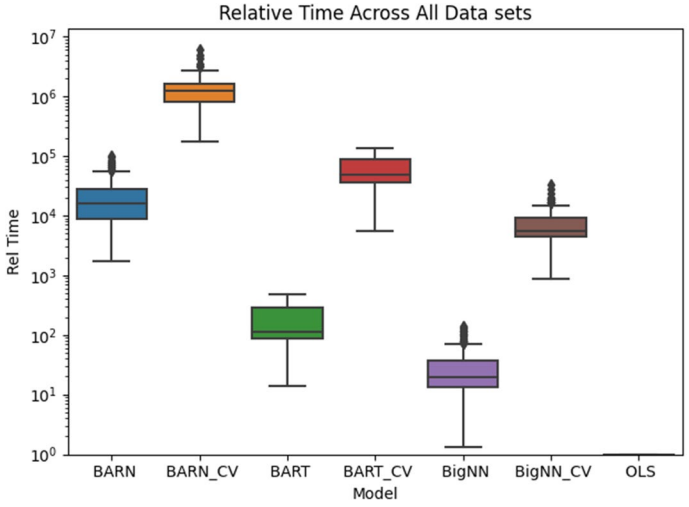

Fig. 6 BARN is comparable in time to other methods with cross-validation

et al. 2021), require very low prediction error and are willing to invest computational resources to achieve it. To assess BARN on this axis, we consider the data sets described in Van Boxel (2024). While these data sets are modest in size (about 1000 data points and 10 features), they are sufficient to realize the differences in computation times, in seconds, shown in Table 1. These times do vary across runs, but not to the extent of the order of magnitude differences in times across methods.

In Fig. 6, we see the relative computation times for our various methods on all data sets. OLS, being only linear algebra, is always the fastest (hence the relative time of 1). Next, training a single neural network with gradient descent takes 10–100 times as long (still less than a second on any given problem). BART is solidly 100 times slower than OLS, about 1 s of real time. Plain BARN is about 10,000 times slower than OLS, taking on the order of 10 s to a 1 min on a given problem. This is primarily due to the necessity of training new small neural networks for all the

MCMC iterations. While each one is very fast (close to 10 times faster than the single big network), doing this for 200 MCMC iterations is a significant cost. BART avoids this cost because it does not "train" in a traditional sense (i.e. it does not set weights with the standard CART procedure), so the MCMC iterations are not as computationally intense. On the face of it, BARN seems like it is very slow.

When we consider the methods with cross-validated hyperparameter tuning, however, we see that BARN is actually time-competitive with the other nonlinear methods. Those methods are on the order of tens of seconds for this data. Now, from earlier studies (Van Boxel 2024), we know that plain BARN is nearly as accurate as BARN with tuning. Across real data benchmarks in that study, BARN without tuning had only 0.2% higher median relative test RMSE than BARN with tuning. In situations where extensive computational resources are available or minimizing error as much as possible is critical, tuning BARN with cross-validation can provide a modest improvement over untuned BARN.

Yet even untuned BARN still produces lower error than BART or the big neural network even when those methods are tuned. Looking at Fig. 6 again, we see that plain BARN takes about the same amount of relative time as other methods when those are tuned. Those methods benefit significantly from such tuning, whereas BARN may be adaptable even without it. For situations requiring low testing error in regression, BARN is time-competitive with other nonlinear algorithms.

We did explore various speedups to BARN. Recall that we chose to implement BARN in a Scikit-Learn compatible way, including using their `MLPRegressor` class. While this is convenient, it may not be the fastest NN implementation. So we also implemented BARN using TensorFlow-based neural networks, including training these NNs on GPUs (Abadi et al. 2015). For large neural networks, GPUs often provide a 5 times or better speedup from parallelism (Lind and Pantigoso Velasquez 2019), but with BARN, we found just the opposite. Because BARN typically uses tiny neural networks, these map poorly to GPUs. Significant time, relative to the computation cost, is lost simply moving data between CPU and GPU. Another improvement we tried, however, was more helpful. Initially, we trained BARN networks using typical gradient-descent style solvers common in machine learning and TensorFlow. But since we were using Scikit-Learn, we also had easily available quasi-Newton methods like BFGS (Nocedal and Wright 1999). At the scale of networks and data sets considered, we found switching away from gradient descent provided a 2–4 times speedup. Understanding that this is problem dependent, however, we enable setting this parameter in the function and provide a sane default that selects based on network size. Such techniques provide BARN with some speed enhancements, though more research in this area is needed.

5 Conclusion

We reviewed the design and capabilities of the new Python package, Bayesian Additive Regression Models in Python (`barmpy`). In addition to the favorable results of lower error rates on benchmark data seen in Van Boxel (2024), we found `barmpy` to be fast enough on relevant problems. While it is an order of magnitude slower

than BART, BARN does not need hyperparameter tuning to do well, making it generally time-competitive. Still, additional research into faster implementations of BARN would be beneficial. TensorFlow wasn't able to improve speeds, but another linear algebra library, one tuned for many small matrices, might be appropriate. Or, BARN may benefit from an algorithmic change. For example, rather than learning weights via the neural network training procedure, we could sample them directly as part of the MCMC process. This has the downside of ignoring existing optimization approaches, but something similar works for BART, so it may work here. Beyond direct metrics, we also emphasized the importance of accessibility for `barmpy`. This is why we chose to develop it in Python with tight integration with Scikit-Learn. We meet the practitioners where they are. Likewise, we recognize the importance of self-teaching in learning new software. So we provide not only the library itself, but supporting documentation and tutorials. Finally, we consider some future additions to the package. As the name, `barmpy`, suggests, we seek to support a generic model backbone, not just neural networks. Provided one can (hopefully rigorously) supply transition and posterior probability methods, this ought to be broadly applicable. For example, support vector machines may be a straightforward next backbone option to implement. Additionally, we note that because BARN ensembles are structurally equivalent to neural networks, techniques specific to NNs are applicable to BARN. For example, sensitivity analysis attempts to find relevant features by finding those which are sensitive relative to changes in the target (Pizarroso et al. 2020). All these implementation details and other "extras" are necessary for enabling users to learn `barmpy` and employ it effectively.

Acknowledgements I must thank both of my PhD co-advisors, Xueying Tang and Cristian Román-Palacios, for their constant guidance and support. Prof. Tang provided key mathematical insight and ensured ongoing statistical rigor. Prof. Román-Palacios balanced this with practical machine learning advice as well as the perspective of a research scientist.

Code availability The `barmpy` library is available in full on GitHub at https://github.com/dvbuntu/barmpy.

Declarations

Conflict of interest This research was performed in part while employed by the Data Diversity Lab within the School of Information at The University of Arizona.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado G. S, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/

Abril-Pla O, Andreani V, Carroll C, Dong L, Fonnesbeck CJ, Kochurov M, Kumar R, Lao J, Luhmann CC, Martin OA et al (2023) PyMC: a modern, and comprehensive probabilistic programming framework in Python. PeerJ Comput Sci 9:e1516. https://doi.org/10.7717/peerj-cs.1516

Aggarwal R, Sounderajah V, Martin G, Ting DS, Karthikesalingam A, King D, Ashrafian H, Darzi A (2021) Diagnostic accuracy of deep learning in medical imaging: a systematic review and meta-analysis. NPJ Digit Med 4(1):65

Baumer B, Udwin D (2015) R markdown. Wiley Interdiscip Rev Comput Stat 7(3):167–177

Brandl G (2021) Sphinx documentation. http://sphinx-doc.org/sphinx.pdf

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Chipman HA, George EI, McCulloch RE (1998) Bayesian CART model search. J Am Stat Assoc 93(443):935–948

Chipman HA, George EI, McCulloch RE et al (2010) BART: Bayesian additive regression trees. Ann Appl Stat 4(1):266–298

Chipman H, McCulloch R, Chipman G (2022) Package "bayestree". https://CRAN.R-project.org/package=BayesTree. R package version 1.4

Chollet F et al (2015) Keras. https://keras.io

Coltman J |(2022) BARTPy. https://github.com/JakeColtman/bartpy

Durmus A, Moulines É (2015) Quantitative bounds of convergence for geometrically ergodic Markov chain in the Wasserstein distance with application to the Metropolis adjusted Langevin algorithm. Stat Comput 25:5–19

Eiler JM, Schauble E (2004) 18O13C16O in earth's atmosphere. Geochim Cosmochim Acta 68(23):4767–4777

Escamilla E, Klein M, Cooper T, Rampin V, Weigle MC, Nelson ML (2022) The rise of GitHub in scholarly publications. In: International conference on theory and practice of digital libraries. Springer, p 187–200

Flegal JM, Gong L (2015) Relative fixed-width stopping rules for Markov chain monte Carlo simulations. Stat Sin 25:655–675

Gençay R, Qi M (2001) Pricing and hedging derivative securities with neural networks: Bayesian regularization, early stopping, and bagging. IEEE Trans Neural Netw 12(4):726–734

Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020) Array programming with NumPy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2

Hornik K (2012) The comprehensive R archive network. Wiley Interdiscip Rev Comput Stat 4(4):394–398

Hunter JD (2007) Matplotlib: a 2D graphics environment. Comput Sci Eng 9(3):90–95. https://doi.org/10.1109/MCSE.2007.55

James G, Witten D, Hastie T, Tibshirani R, Taylor J (2023) An introduction to statistical learning: with applications in Python. Springer Nature, New York City

Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. Preprint at arXiv:1412.6980

Lind E, Pantigoso Velasquez Ä (2019) A performance comparison between CPU and GPU in tensorflow. Examensarbete inom teknik, KTH Royal Institute of Technology

Linero AR (2022) Generalized Bayesian additive regression trees models: beyond conditional conjugacy. Preprint at arXiv:2202.09924

McCulloch R, Sparapani R, Gramacy R, Pratola M, Spanbauer C, Plummer M, Best N, Cowles K Kate and Vines Package "bart" (2024) https://CRAN.R-project.org/package=BART. R package version 2.9.6

Nocedal J, Wright SJ (1999) Numerical optimization. Springer, New York City

Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A et al (2022) Training language models to follow instructions with human feedback. Adv Neural Inf Process Syst 35:27730–27744

Patel K, Fogarty J, Landay JA, Harrison BL (2008) Examining difficulties software developers encounter in the adoption of statistical machine learning. In: AAAI, pp 1563–1566

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830

Petersen SV, Defliese WF, Saenger C, Daëron M, Huntington KW, John CM, Kelson JR, Bernasconi SM, Colman AS, Kluge T et al (2019) Effects of improved 17o correction on interlaboratory agreement in clumped isotope calibrations, estimates of mineral-specific offsets, and temperature dependence of acid digestion fractionation. Geochem Geophys Geosyst 20(7):3495–3519

Pizarroso J, Portela J, Muñoz A (2020) Neuralsens: sensitivity analysis of neural networks. Preprint at arXiv:2002.11423

Prado EB, Moral RA, Parnell AC (2021) Bayesian additive regression trees with model trees. Stat Comput 31:1–13

PyPi Maintainers (2023) Python package index - PyPi. https://pypi.org/

Quiroga M, Garay PG, Alonso JM, Loyola JM, Martin OA (2022) Bayesian additive regression trees for probabilistic programming. https://arxiv.org/abs/2206.03619

Román Palacios C, Carroll H, Arnold A, Flores R, Petersen S ,McKinnon K, Tripati A, Gan Q (2022) Bayclump: Bayesian calibration and temperature reconstructions for clumped isotope thermometry. Authorea Preprints

Seabold S, Perktold J (2010) Statsmodels: econometric and statistical modeling with python. In: Proceedings of the 9th Python in science conference, vol 57. Austin, TX, pp 10–25080

Shah N, Engineer S, Bhagat N, Chauhan H, Shah M (2020) Research trends on the usage of machine learning and artificial intelligence in advertising. Augment Hum Res 5:1–15

Solomon J, Rustamov R, Guibas L, Butscher A (2014) Earth mover's distances on discrete surfaces. ACM Trans Graph (ToG) 33(4):1–12

Srinath K (2017) Python-the fastest growing programming language. Int Res J Eng Technol 4(12):354–357

Van Boxel D (2023) Barmpy documentation. https://dvbuntu.github.io/dvbuntu/barmpy

Van Boxel D (2024) Bayesian additive regression networks. Preprint at arXiv:2404.04425