Tutorial

Troy P. Wixson

2025-05-06

nnadic

The goal of nnadic (**N**eural **N**etwork for **A**symptotic **D**ependence/**I**ndependence **C**lassification) is to classify bivariate data sets as either asymptotically dependent or independent using a trained convolutional neural network. The tool is set up to automatically:
- transform the marginal distribution
- take the top 5% of the data (using the l_∞-norm)
- resample or subsample as necessary to ensure the data is of the correct dimension
- output the predicted result from the neural network

Installation

You can install the development version of nnadic from GitHub. We suggest the following suggested install code which overcomes common installation issues:

```
install.packages("remotes", force = TRUE)
remotes::install_github("rstudio/tensorflow", force = TRUE)
remotes::install_github("rstudio/keras", force = TRUE)
reticulate::install_miniconda()
tensorflow::install_tensorflow()
reticulate::install_python(version = '3.9')
keras::install_keras()
options(timeout = 400)
install_github("twixson/nnadicTestData", force = TRUE)
install_github("twixson/nnadic")
```

Introduction

Classifier Structure

The `nnadic` classifier is a neural network that is composed of two component networks that are linked with a permutation invariant aggregation function. The first component network extracts features from each point. The aggregation function averages each feature across points. The second component network maps the averaged features to a value between zero (ADep) and one (AInd). This architecture fits in the DeepSets framework by of Zaheer et al. (2017).

Let $(\boldsymbol{W}_\psi^{(l)}, \boldsymbol{b}_\psi^{(l)})$ denote the parameters of the l^{th} layer of the feature network $\boldsymbol{\psi}$, $(\boldsymbol{W}_\phi^{(l)}, \boldsymbol{b}_\phi^{(l)})$ denote the parameters of the l^{th} layer of the inference network ϕ, $\boldsymbol{\gamma}_\psi = \left(\boldsymbol{W}_\psi^{(1)}, \boldsymbol{b}_\psi^{(1)}, \ldots, \boldsymbol{W}_\psi^{(L_\psi)}, \boldsymbol{b}_\psi^{(L_\psi)}\right)$, and $\boldsymbol{\gamma}_\phi = \left(\boldsymbol{W}_\phi^{(1)}, \boldsymbol{b}_\phi^{(1)}, \ldots, \boldsymbol{W}_\phi^{(L_\phi)}, \boldsymbol{b}_\phi^{(L_\phi)}\right)$. Let $\boldsymbol{\gamma} = (\boldsymbol{\gamma}_\psi, \boldsymbol{\gamma}_\phi)$ denote the collection of parameters in $\hat{g}$. Our network can

be represented as

$$\hat{g}(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}) = \phi\big[A(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\psi), \boldsymbol{\gamma}_\phi\big] \tag{1}$$

$$A(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\psi) = \frac{1}{m}\mathbf{1}^T \boldsymbol{\psi}(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\psi) \tag{2}$$

where $\mathbf{1}^T$ is a row-vector of ones and thus we are computing the column-wise (point-wise) averages. In the following representation of our network we adopt a notation that is standard to deep learning; for matrix $\boldsymbol{Z} = (\boldsymbol{z}_1 \ \ldots \ \boldsymbol{z}_q)$ (where the $\boldsymbol{z}_j$ are understood to be the column vectors of $\boldsymbol{Z}$) and column vector $\boldsymbol{v}$ define $\boldsymbol{Z} + \boldsymbol{v} := (\boldsymbol{z}_1 + \boldsymbol{v} \ \ldots \ \boldsymbol{z}_q + \boldsymbol{v})$. Our feature network can, due to the desired separability, be represented as

$$\boldsymbol{h}_\psi^{(1)} = \sigma\big(\boldsymbol{X}^{(m)}\boldsymbol{W}_\psi^{(1)} + \boldsymbol{b}_\psi^{(1)}\big), \tag{3}$$

$$\boldsymbol{h}_\psi^{(l)} = \sigma\big(\boldsymbol{h}_\psi^{(l-1)}\boldsymbol{W}_\psi^{(l)} + \boldsymbol{b}_\psi^{(l)}\big), \quad l = 2, \ldots, L_\psi - 1, \tag{4}$$

$$\boldsymbol{\psi}(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\psi) = \sigma\big(\boldsymbol{h}_\psi^{(L_\psi - 1)}\boldsymbol{W}_\psi^{(L_\psi)} + \boldsymbol{b}_\psi^{(L_\psi)}\big). \tag{5}$$

Here $\boldsymbol{h}_\psi^{(l)}$ is the output of the lth (hidden) layer of the $\boldsymbol{\psi}$-network, $\boldsymbol{\sigma} : \mathbb{R}^{q^{(l)}} \to \mathbb{R}^{q^{(l)}}$ is a componentwise activation function (defined below), and $\boldsymbol{W}_\psi^{(l)}$ contains column-wise feature weights.

Letting $q^{(l)}$ denote the feature dimension of the l^{th} layer, we note that each $\boldsymbol{W}^{(l)} \in \mathbb{R}^{q^{(l-1)} \times q^{(l)}}$ and thus the output from the l^{th} layer is $m \times q^{(l)}$. Aggregation across points results in a q-vector which is input into the inference network ϕ. The inference network can be represented as

$$\boldsymbol{h}_\phi^{(1)} = \sigma\big(\boldsymbol{W}_\phi^{(1)} A(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\psi) + \boldsymbol{b}_\phi^{(1)}\big), \tag{6}$$

$$\boldsymbol{h}_\phi^{(l)} = \sigma\big(\boldsymbol{W}_\phi^{(l)}\boldsymbol{h}_\phi^{(l-1)} + \boldsymbol{b}_\phi^{(l)}\big), \quad l = 2, \ldots, L_\phi - 1, \tag{7}$$

$$\boldsymbol{\phi}(\boldsymbol{X}^{(m)}, \boldsymbol{\gamma}_\phi) = \sigma^{(L_\phi)}\big(\boldsymbol{W}_\phi^{(L_\phi)}\boldsymbol{h}_\phi^{(L_\phi - 1)} + \boldsymbol{b}_\phi^{(L_\phi)}\big). \tag{8}$$

The activation function $\boldsymbol{\sigma}$ in equations (5)-(9) is defined componentwise so that for q-vector $\mathbf{y} = (y_1, \ldots, y_q)^{\mathrm{T}}$, $\boldsymbol{\sigma}(\mathbf{y}) = \{\sigma_1(y_1), \ldots, \sigma_q(y_q)\}^{\mathrm{T}}$. Each σ_j, $j = 1, \ldots, q$ is a the leaky ReLU function (Maas et al. 2013) which is a piecewise function: $\sigma_j(y_j) = y_j$ if $y_j \geq 0$ and, for some constant a close to zero (we use 0.01), $\sigma_j(y_j) = ay_j$ if $y_j < 0$. This function protects against so-called dead ReLU neurons due to large negative weights/biases which output zeros with gradients equal to zero. Zero gradients prevent gradient descent from updating the associated parameters and thus those neurons are never activated (Maas et al. 2013) which can cause issues when fitting NNs. The final layer (equation 10) is activated with a sigmoid (logistic) function $\sigma^{(L_\phi)}$ which ensures output values are between zero and one.

How to get inputs

Inputs to `nnadic` are the large observations from some dataset of interest.

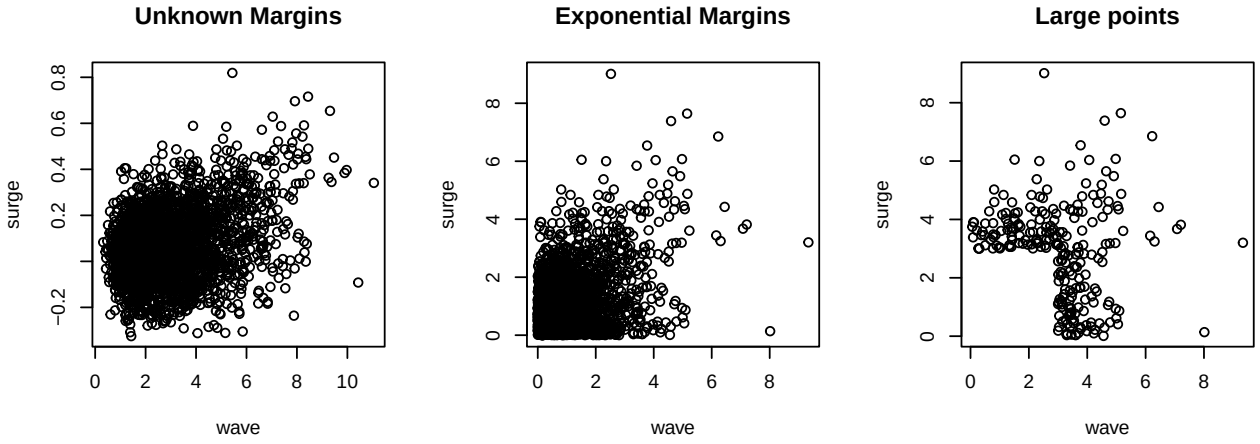
The data must first be transformed to exponential margins. This can be done with the `nnadic`'s function `transform_to_exponential()` which uses the empirical distribution function for points below the 0.95-quantile and a generalized Pareto distribution fit to points above the 0.95-quantile and to transform the data to be marginally uniformly distributed. The probability integral transformation is used to transform those data to be marginally exponential. This function works on one margin at a time and can be used in the following manner:

```
library(nnadic)
#> Welcome to nnadic!
#install.packages("ismev")
library(ismev)
#> Loading required package: mgcv
#> Loading required package: nlme
#> This is mgcv 1.9-3. For overview type 'help("mgcv-package")'.
data("wavesurge")
```

```
wavesurge_exp <- apply(wavesurge, 2, transform_to_exponential)
#> [1] "...  ...location: 6.08   scale: 1.325   shape: -0.183"
#> [1] "...  ...location: 0.322   scale: 0.093   shape: -0.04"
```

Once the data are on exponential margins we need to subset to include only points that have L_∞-norms greater than the 0.95-quantile (≈ 2.995). This is simple to accomplish with built-in `R` functions:

```
q95 <- qexp(p = 0.95)
large_points <-
  which(apply(wavesurge_exp, 1, q95 = q95,
              function(x, q95 = q95){ifelse(max(x) > q95, 1, 0)}) == 1)
wavesurge_subset <- wavesurge_exp[large_points, ]


par(mfrow = c(1, 3))
plot(wavesurge, main = "Unknown Margins")
plot(wavesurge_exp, asp = 1, main = "Exponential Margins")
plot(wavesurge_subset, asp = 1, main = "Large points")
```



Our classifier was trained on subsets of samples of size $n = 10000$. This means that we need $m = 500$ large points as input. Often real data do not have sample sizes of 10000 so we have to resample (when $n < 10000$) or subsample (when $n > 10000$) our large points to get 500. This is done with `nnadics` function `resample_to_500()`. This function sub/re-samples from the index vector that we created in the previous code chunk. This function allows the user to generate several different sets of sub/re-samples in case the user wants to assess the variability associated with the sub/re-sampling. If the dataset has fewer than 10000 points, then the user can choose whether to re-sample the large points (with replacement) 500 times or to ensure that all large points are included as many times as possible and randomly sample the remainder. This is done with the `include_all.` argument.

```
set.seed(1928234)
indices_500 <- resample_to_500(large_points,
                               num_datasets. = 25,
                               include_all. = TRUE)
#> [1] "...the first 486 points in each dataset"
#> [1] "...   are the top 5% of points (repeated when possible)."
#> [1] "...the rest of the points were subsampled so that all 25"
#> [1] "...   datasets have 500 points."
# note that the first points are the same in each dataset
print("First 5 points from first 10 resampled datasets:")
#> [1] "First 5 points from first 10 resampled datasets:"
indices_500[1:5, 1:10]
```

```
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]   37   37   37   37   37   37   37   37   37    37
#> [2,]   71   71   71   71   71   71   71   71   71    71
#> [3,]  176  176  176  176  176  176  176  176  176   176
#> [4,]  229  229  229  229  229  229  229  229  229   229
#> [5,]  230  230  230  230  230  230  230  230  230   230
# note that the last points in each dataset are randomly sampled
print("Last 5 points from first 10 resampled datasets:")
#> [1] "Last 5 points from first 10 resampled datasets:"
indices_500[496:500 ,1:10]
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,] 1517 2109 2213 2612  432 1736 1733 1472 2868  2607
#> [2,] 2709 1737 2607 2834  229 1466 2644  346 2710  2645
#> [3,] 2862 2637 2831 1479 2612 1660 2213 2645  527  1518
#> [4,]  901  310  827 2704 2787 2648 2706 2082  319   774
#> [5,] 1434 1464  772 2082 2831  797 1332 1329 2790   843

wavesurge_500 <- lapply(1:25, function(x){wavesurge_exp[indices_500[,x], ]})
```

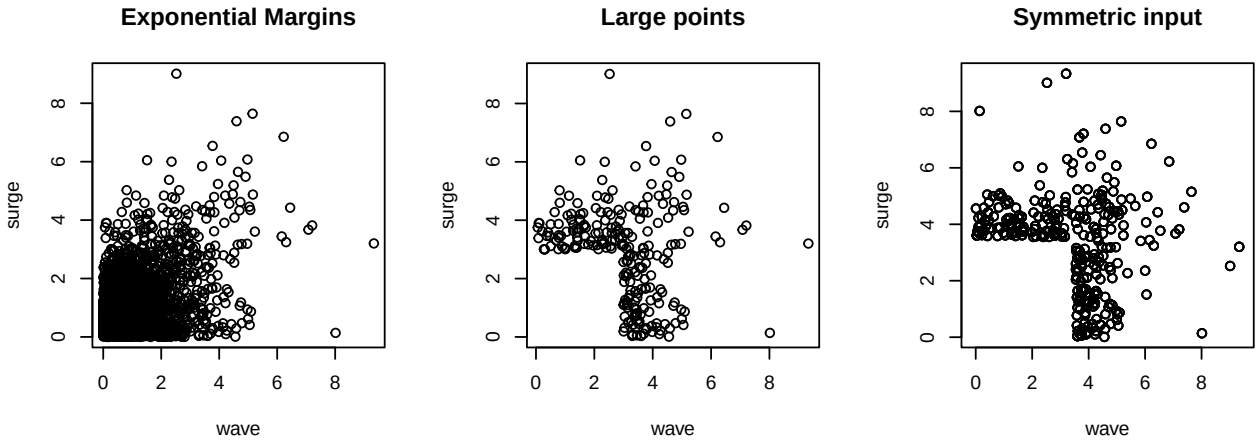
A copy of each of these points is added to the subsetted data after reflection across the identity line. This data augmentation ensures coordinate invariance and is performed with the `make_symmetric()` function. Finally, we need to transform the shape of our input object so that it is an `array` with dimensions `c(number_datasets, 1000, 2)`.

```
#install.packages("abind")
wavesurge_ready <- lapply(wavesurge_500, make_symmetric)
wavesurge_ready <- abind::abind(wavesurge_ready, along = 1)
dim(wavesurge_ready)
#> [1]   25 1000    2
```

We have combined all of the aforementioned simple steps into a single function which we call `get_nnadic_input()`. If the user believes the data are already Exponential or prefers to transform the data with a different method they can set `make_exponential = FALSE`. If the user wants to see how the largest 500 points from their dataset with $n > 10000$, then they can set `subsample = FALSE`. If the dataset is a time series (and thus the input is a `vector`) then the user can choose to assess the dependence at different lags with `comp_lag = 1` meaning the lag-1 dependence (`comp_lag` is ignored if the input is a matrix). If the user does not want to see the built-in comments they can set `verbose = FALSE`.

```
wavesurge_ready2 <- get_nnadic_input(wavesurge,
                                     make_exponential = TRUE,
                                     subsample = TRUE,
                                     num_datasets = 25,
                                     include_all = TRUE,
                                     comp_lag = 1,
                                     verbose = TRUE)
#> [1] "...transforming to exponential marginal distributions"
#> [1] "...   estimated gpd parameters in the marginal transformation were: "
#> [1] "...  ...location: 6.08   scale: 1.325   shape: -0.183"
#> [1] "...  ...location: 0.322   scale: 0.093   shape: -0.04"
#> [1] "...fewer than 10000 points detected, points above the 0.95 quantile"
#> [1] "...   will be resampled"
#> [1] "...   145 large points identified"
#> [1] "...the first 435 points in each dataset"
#> [1] "...   are the top 5% of points (repeated when possible)."
#> [1] "...the rest of the points were subsampled so that all 25"
```

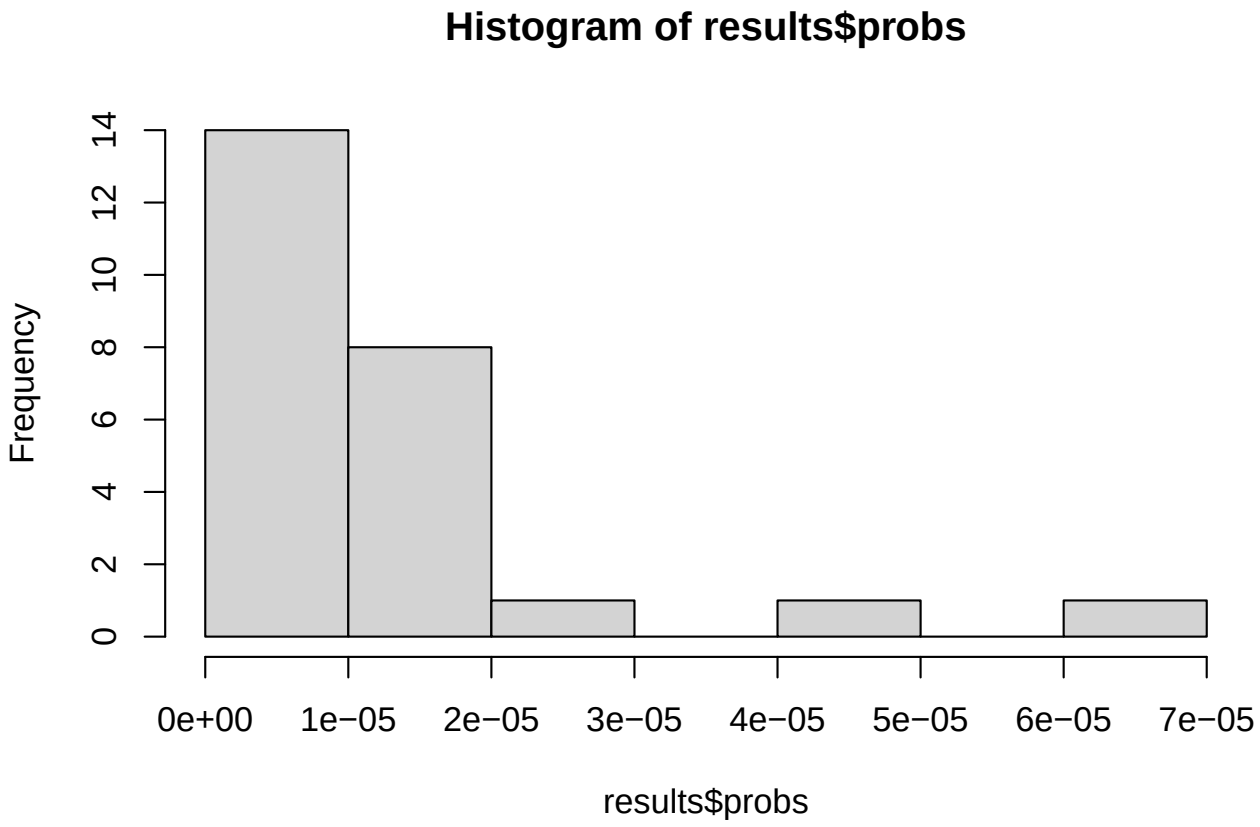
```
#> [1] "...   datasets have 500 points."
#> [1] "...each dataset was made symmetric and now has 1000 points."
par(mfrow = c(1, 3))
plot(wavesurge_exp, main = "Exponential Margins")
plot(wavesurge_subset, asp = 1, main = "Large points")
plot(wavesurge_ready2[1,,], asp = 1, main = "Symmetric input",
     xlab = "wave", ylab = "surge")
```



Classification

Classification is simple once the data are ready to be input into `nnadic`. One simply needs to call the `nnadic()` function and use the output from `get_nnadic_input()` as the first argument. In this example, where we have repetitions of a single dataset that we want classified, we set `one_test = TRUE`. This toggle ought to be switched to `FALSE` when classifying many datasets which occurs, for example, during simulation studies. Part of the default output is a histogram of the classification probabilities which can be toggled off by setting `make_hist = FALSE`. The user can toggle off the built-in by setting `verbose = FALSE`.

```
wavesurge_out <- nnadic(wavesurge_ready2, one_test = , make_hist = , verbose = )
#> 1/1 - 0s - 73ms/epoch - 73ms/step
```

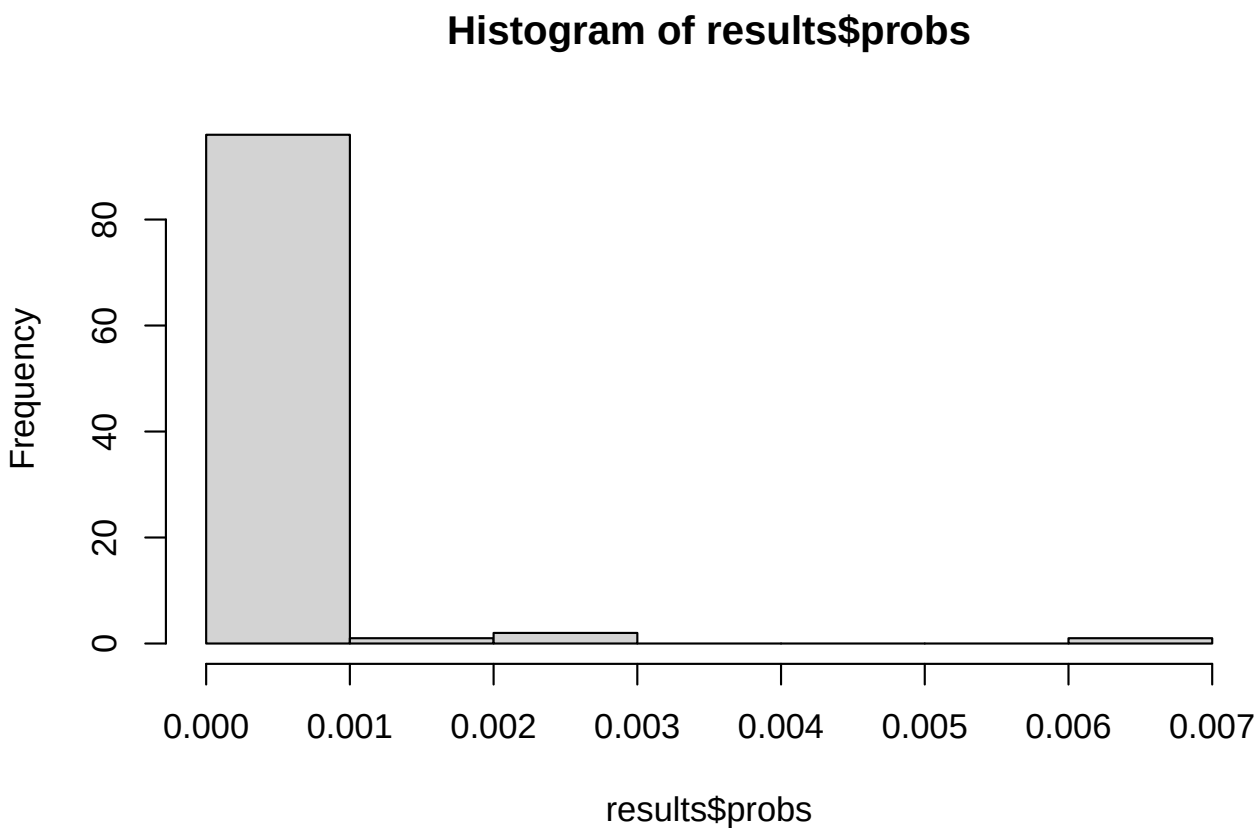



```
#> [1] "Probabilities and predictions for each dataset are being returned"
#> [1] "Each probability is the probability of AI which is coded as '1'"
#> [1] "###################"
#> [1] "The mean of the predictions is: 0"
#> [1] "This is `nnadic`'s probability that these data are AI"
```

Here we can see that all 25 of the resampled datasets are classified as asymptotically dependent with very little uncertainty (the largest classification probability is less than 0.0001).

It is straightforward to perform this entire classification task with only a few lines of code which we demonstrate by looking at the same `wavesurge` dataset but this time we randomly sample all 500 input points (`include_all = F`) and we generate 100 different resampled datasets (`num_datasets = 100`).

```
wavesurge_input_resample_all <- get_nnadic_input(wavesurge,
                                                 include_all = FALSE,
                                                 num_datasets = 100)
#> [1] "...transforming to exponential marginal distributions"
#> [1] "...   estimated gpd parameters in the marginal transformation were: "
#> [1] "...  ...location: 6.08   scale: 1.325   shape: -0.183"
#> [1] "...  ...location: 0.322   scale: 0.093   shape: -0.04"
#> [1] "...fewer than 10000 points detected, points above the 0.95 quantile"
#> [1] "...   will be resampled"
#> [1] "...   145 large points identified"
#> [1] "...the rest of the points were subsampled so that all 100"
#> [1] "...   datasets have 500 points."
#> [1] "...each dataset was made symmetric and now has 1000 points."
wavesurge_out_resample_all <- nnadic(wavesurge_input_resample_all)
#> 4/4 - 0s - 22ms/epoch - 6ms/step
```




```
#> [1] "Probabilities and predictions for each dataset are being returned"
#> [1] "Each probability is the probability of AI which is coded as '1'"
#> [1] "###################"
#> [1] "The mean of the predictions is: 0"
#> [1] "This is `nnadic`'s probability that these data are AI"
```

It is clear, once again, that our classifier considers these data to be asymptotically dependent.

Other Examples

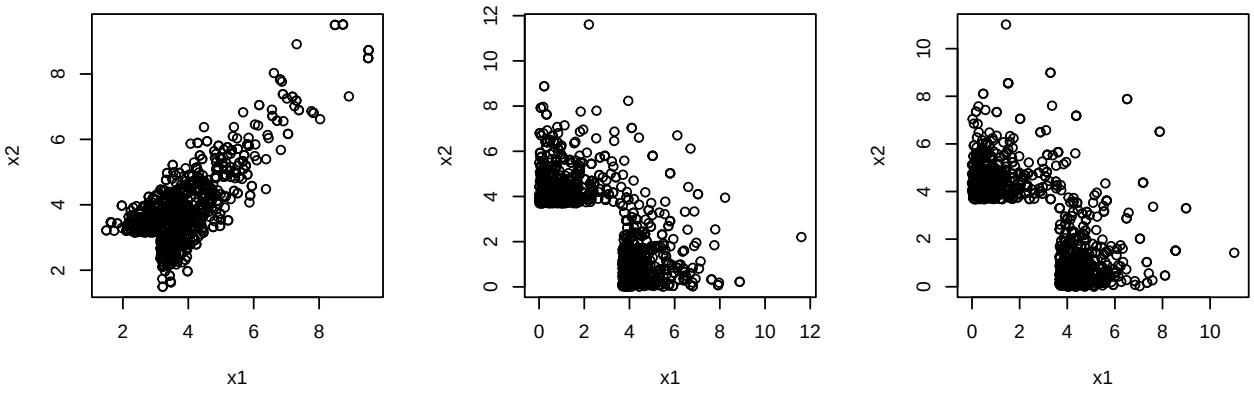
Testing `nnadic` on Gaussian and logistic data

Performing a simulation study is simple as long you can generate data from the models that you are interested in. Here we test one common models from each dependence regime. The first step is generating testing data. In this case we know the generating distribution so we use that information to transform to exponential margins.

```
library(mvtnorm) # for multivariate Gaussian
library(evd)     # for logistic
set.seed(72143) #92841
gaussian     <- logistic <- list()
num_tests    <- 10
sample_sizes <- sample(2000:18000, size = num_tests)
dependence   <- runif(num_tests)

for(i in 1:num_tests){
  temp_gaussian <- rmvnorm(sample_sizes[i],
                           sigma = matrix(c(1, dependence[i], dependence[i], 1),
                                          ncol = 2))
```

```
  temp_gaussian <- pnorm(temp_gaussian)
  gaussian[[i]] <- qexp(temp_gaussian)

  temp_logistic <- rbvevd(sample_sizes[i],
                          dep = 1 - dependence[i],
                          model = "log",
                          mar1 = c(1, 1, 1))
  temp_logistic <- pfrechet(temp_logistic)
  logistic[[i]] <- qexp(temp_logistic)
}

gaussian <- lapply(gaussian,
                   get_nnadic_input,
                   make_exponential = FALSE,
                   num_datasets = 1,
                   verbose = FALSE)
gaussian <- abind::abind(gaussian, along = 1)

logistic <- lapply(logistic,
                   get_nnadic_input,
                   make_exponential = FALSE,
                   num_datasets = 1,
                   verbose = FALSE)
logistic <- abind::abind(logistic, along = 1)

gaussian_out <- nnadic(gaussian, one_test = FALSE, make_hist = FALSE)
#> 1/1 - 0s - 9ms/epoch - 9ms/step
#> [1] "Probabilities and predictions for each dataset are being returned"
#> [1] "Each probability is the probability of AI which is coded as '1'"
logistic_out <- nnadic(logistic, one_test = FALSE, make_hist = FALSE)
#> 1/1 - 0s - 9ms/epoch - 9ms/step
#> [1] "Probabilities and predictions for each dataset are being returned"
#> [1] "Each probability is the probability of AI which is coded as '1'"
```

```
mean(gaussian_out$probs)
#> [1] 0.709944
mean(gaussian_out$preds)
#> [1] 0.7
(gaussian_wrong <- which(gaussian_out$preds != 1))
#> [1] 5 7 9
dependence[gaussian_wrong]
#> [1] 0.95550064 0.08915319 0.07037347
sample_sizes[gaussian_wrong]
#> [1] 13236 11780 15224
par(mfrow = c(1, 3))
plot(gaussian[gaussian_wrong[1],,], asp = 1, xlab = "x1", ylab = "x2")
plot(gaussian[gaussian_wrong[2],,], asp = 1, xlab = "x1", ylab = "x2")
plot(gaussian[gaussian_wrong[3],,], asp = 1, xlab = "x1", ylab = "x2")
```



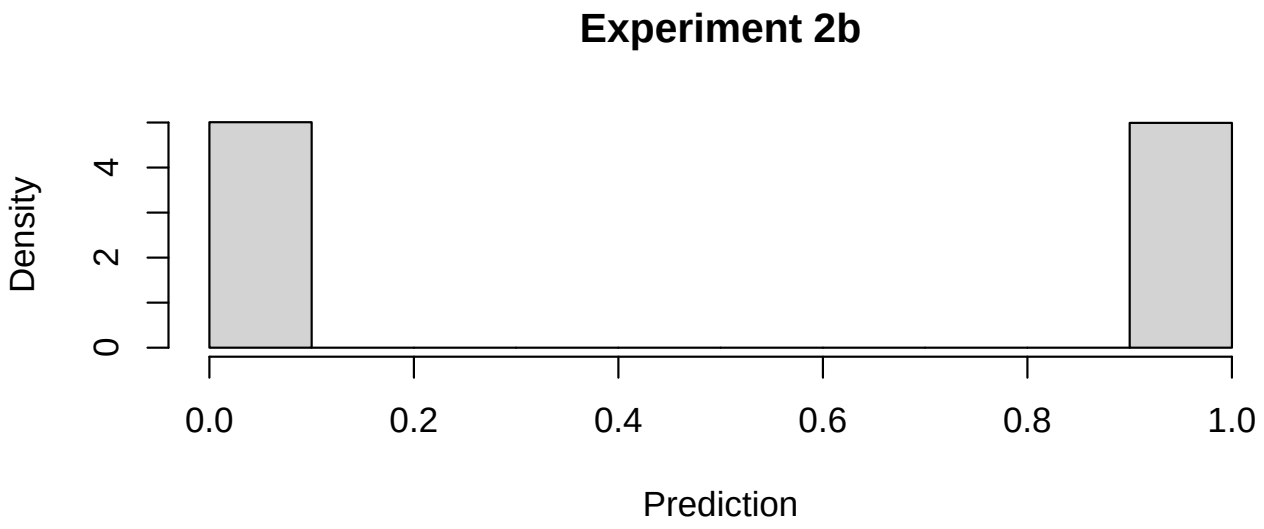

```
mean(logistic_out$probs)
#> [1] 0.00262937
mean(logistic_out$preds)
#> [1] 0
(logistic_wrong <- which(logistic_out$preds != 0))
#> integer(0)
```

When investigating the results we notice that some of the Gaussian datasets are incorrectly classified by `nnadic`. One of the incorrect datasets has dependence parameter close to 1 and the other two dependence parameters are close to zero. These datasets are nearly exactly dependent or perfectly independent.

Testing `nnadic` on the test data from the paper

Here we mimic one of the tests in Wixson and Cooley (2025) using pre-generated Gaussian, logistic, inverted logistic, and asymmetric logistic data. Here we show that one can make a histogram of output values from the `nnadic()` output.

```
library(nnadicTestData)

# the data were saved after resampling but before enforcing symmetry
test_data_four <- make_symmetric(test_data_four)

results <- nnadic(test_data_four, one_test = FALSE, make_hist = FALSE)
#> 125/125 - 0s - 462ms/epoch - 4ms/step
#> [1] "Probabilities and predictions for each dataset are being returned"
#> [1] "Each probability is the probability of AI which is coded as '1'"
hist(results$preds,
     freq = FALSE,
     xlab = "Prediction",
     main = "Experiment 2b")
```



Finally, we highlight one of the results in the paper; namely that `nnadic` correctly classifies nearly 97% of the test datasets!

```
mean(results$preds == test_response_four)
#> [1] 0.96925
```