Package 'pinnEV'

April 21, 2022

Type Package

Title Partially-Interpretable Neural Networks for Extreme Value modelling

Version 0.1.0

Author Jordan Richards

Maintainer The package maintainer <jordan.richards@kaust.edu.sa>

Description Methodology for fitting marginal extreme value (and associated) models using partially-interpretable neural networks. Networks are trained using the R interface to Keras with custom loss functions taken to be penalised versions of the negative log-likelihood for associated models.

License use_mit_license()

Encoding UTF-8

Depends R (>= 3.4)

Imports reticulate, keras,tfprobability,evd,ismev

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.2

R topics documented:

`train_Bern_NN` *Build and train a partially-interpretable neural network for a logistic regression model*

Description

Build and train a partially-interpretable neural network for a logistic regression model

Usage

```
train_Bern_NN(
  Y_train,
  Y_test = NULL,
  X_train,
  type = "MLP",
  n.ep = 100,
  batch.size = 100,
  init.p = 0.5,
  widths = c(6, 3),
  filter.dim = c(3, 3),
  seed = NULL
)

predict_Bern_nn(X_train, model)
```

Arguments

`Y_train, Y_test`

A 2 or 3 dimensional array of training or test response values, with entries of 0/1 for failure/success. Missing values can be handled by setting corresponding entries to `Y_train` or `Y_test` to `-1e5`. The first dimension should be the observation indices, e.g., time.

If `type=="CNN"`, then `Y_train` and `Y_test` must have three dimensions with the latter two corresponding to an M by N regular grid of spatial locations. If `Y_test==NULL`, no validation loss will be computed and the returned model will be that which minimises the training loss over `n.ep` epochs.

`X_train`

A list of arrays corresponding to complementary subsets of the $d \geq 1$ predictors which are used for modelling. Must contain at least one of the following three named entries:

`X_train_lin` A 3 or 4 dimensional array of "linear" predictor values. Same number of dimensions as `X_train_nn`. If `NULL`, a model without the linear component is built and trained. The first 2/3 dimensions should be equal to that of `Y_train`; the last dimension corresponds to the chosen $l \geq 0$ 'linear' predictor values.

`X_train_add_basis` A 4 or 5 dimensional array of basis function evaluations for the "additive" predictor values. The first 2/3 dimensions should be equal to that of `Y_train`; the penultimate dimensions corresponds to the chosen $a \geq 0$ 'linear' predictor values and the last dimension is equal to the number of knots used for estimating the splines. See example. If `NULL`, a model without the additive component is built and trained.

`X_train_nn` A 3 or 4 dimensional array of "non-additive" predictor values. If `NULL`, a model without the NN component is built and trained; if this is the case, then `type` has no efect. The first 2/3 dimensions should be equal to that of `Y_train`; the last dimension corresponds to the chosen $d-l-a \geq 0$ 'non-additive' predictor values.

Note that `X_train` is the predictors for both `Y_train` and `Y_test`.

type	A string defining the type of network to be built. If type=="MLP", the network will have all densely connected layers; if type=="CNN", the network will have all convolutional layers. Defaults to an MLP.
n.ep	Number of epochs used for training. Defaults to 1000.
batch.size	Batch size for stochastic gradient descent. If larger than dim(Y_train)[1], i.e., the number of observations, then regular gradient descent used.
init.p	Sets the initial probability estimate across all dimensions of Y_train.
widths	Vector of widths/filters for hidden dense/convolution layers. Number of layers is equal to length(widths). Defaults to 0.5.
filter.dim	If type=="CNN", this 2-vector gives the dimensions of the convolution filter kernel. The same filter is applied for each hidden layer.
seed	Seed for random initial weights and biases.
model	Fitted keras model. Output from train_Bern_NN.

Details

Consider a Bernoulli random variable, say $Z \sim \text{Bernoulli}(p)$, with probability mass function $\Pr(Z = 1) = p = 1 - \Pr(Z = 0) = 1 - p$. Let $Y \in \{0, 1\}$ be a univariate Boolean response and let $\mathbf{X}$ denote a d-dimensional predictor set with observations $\mathbf{x}$. For integers $l \geq 0, a \geq 0$ and $0 \leq l + a \leq d$, let $\mathbf{X}_L, \mathbf{X}_A$ and $\mathbf{X}_N$ be distinct sub-vectors of $\mathbf{X}$, with observations of each component denoted $\mathbf{x}_L, \mathbf{x}_A$ and $\mathbf{x}_N$, respectively; the lengths of the sub-vectors are l, a and $d - l - a$, respectively. We model $Y|\mathbf{X} = \mathbf{X} \sim \text{Bernoulli}(p(\mathbf{x}))$ with

$$p(\mathbf{x}) = h[\eta_0 + m_L\{\mathbf{x}_L\} + m_A\{x_A\} + m_N\{\mathbf{x}_N\}],$$

where h is the logistic link-function and η_0 is a constant intercept. The unknown functions m_L and m_A are estimated using a linear function and spline, respectively, and are both returned as outputs; m_N is estimated using a neural network.

The model is fitted by minimising the binary cross-entropy loss over n.ep training epochs. Although the model is trained by minimising the loss evaluated for Y_train, the final returned model may minimise some other loss. The current state of the model is saved after each epoch, using keras::callback_model_checkpoint, if the value of some criterion subcedes that of the model from the previous checkpoint; this criterion is the loss evaluated for validation/test set Y_test if !is.null(Y_test) and for Y_train, otherwise.

Value

train_Bern_NN returns the fitted model. predict_Bern_nn is a wrapper for keras::predict that returns the predicted probability estimates, and, if applicable, the linear regression coefficients and spline bases weights.

Examples

```
# Build and train a simple MLP for toy data
```

```
# Create 'nn', 'additive' and 'linear' predictors
X_train_nn<-rnorm(5000); X_train_add<-rnorm(2000); X_train_lin<-rnorm(3000)

#Re-shape to a 4d array. First dimension corresponds to observations,
#last to the different components of the predictor set
dim(X_train_nn)=c(10,10,10,5)
dim(X_train_lin)=c(10,10,10,3)
dim(X_train_add)=c(10,10,10,2)

# Create toy response data

#Linear contribution
m_L = 0.3*X_train_lin[,,,1]+0.6*X_train_lin[,,,2]-0.2*X_train_lin[,,,3]

# Additive contribution
m_A = 0.1*X_train_add[,,,1]^2+0.2*X_train_add[,,,1]-0.1*X_train_add[,,,2]^3+0.5*X_train_add[,,,2]^2

#Non-additive contribution - to be estimated by NN
m_N = exp(-3+X_train_nn[,,,2]+X_train_nn[,,,3])
+sin(X_train_nn[,,,1]-X_train_nn[,,,2])*(X_train_nn[,,,1]+X_train_nn[,,,2])

p=0.5+0.5*tanh((m_L+m_A+m_N)/2) #Logistic link
Y=apply(p,1:3,function(x) rbinom(1,1,x))

#Create training and test, respectively.
#We mask 20% of the Y values and use this for validation/testing.
#Masked values must be set to -1e5 and are treated as missing whilst training

mask_inds=sample(1:length(Y),size=length(Y)*0.8)

Y_train<-Y_test<-Y #Create training and test, respectively.
Y_train[-mask_inds]=-1e5
Y_test[mask_inds]=-1e5



#To build a model with an additive component, we require an array of evaluations of
#the basis functions for each pre-specified knot and entry to X_train_add

rad=function(x,c){ #Define a basis function. Here we use the radial bases
  out=abs(x-c)^2*log(abs(x-c))
  out[(x-c)==0]=0
  return(out)
}

n.knot = 5 # set number of knots. Must be the same for each additive predictor
knots=matrix(nrow=dim(X_train_add)[4],ncol=n.knot)

#We set knots to be equally-spaced marginal quantiles
for( i in 1:dim(X_train_add)[4]) knots[i,]=quantile(X_train_add[,,,i],probs=seq(0,1,length=n.knot))

X_train_add_basis<-array(dim=c(dim(X_train_add),n.knot))
```

```
for( i in 1:dim(X_train_add)[4]) {
for(k in 1:n.knot) {
X_train_add_basis[,,,i,k]= rad(x=X_train_add[,,,i],c=knots[i,k])
#Evaluate rad at all entries to X_train_add and for all knots
}}

X_train=list("X_train_nn"=X_train_nn, "X_train_lin"=X_train_lin,"X_train_add_basis"=NULL)

#Build and train a two-layered "lin+GAM+NN" MLP
model<-train_Bern_NN(Y_train, Y_test,X_train,  type="MLP",n.ep=2000,
                     batch.size=50,init.p=0.4, widths=c(6,3))

out<-predict_bernoulli_nn(X_train,model)
print(out$lin.coeff)
```

Index