Package 'RFAE'

May 7, 2026

Title Autoencoding Random Forests

Version 0.1.0

Maintainer Binh Duc Vu <vuducbinh2210@gmail.com>

Description Autoencoding Random Forests ('RFAE') provide a method to autoencode mixed-type tabular data using Random Forests ('RF'), which involves projecting the data to a latent feature space of user-chosen dimensionality (usually a lower dimension), and then decoding the latent representations back into the input space. The encoding stage is useful for feature engineering and data visualisation tasks, akin to how principal component analysis ('PCA') is used, and the decoding stage is useful for compression and denoising tasks. At its core, 'RFAE' is a post-processing pipeline on a trained random forest model. This means that it can accept any trained RF of 'ranger' object type: 'RF', 'URF' or 'ARF'. Because of this, it inherits Random Forests' robust performance and capacity to seamlessly handle mixed-type tabular data. For more details, see Vu et al. (2025) <doi:10.48550/arXiv.2505.21441>.

License GPL (>= 3)

URL https://github.com/bips-hb/RFAE

BugReports https://github.com/bips-hb/RFAE/issues

Depends R (>= 4.4.0)

Imports caret, data.table, foreach, Matrix, methods, mgcv, ranger, RANN, RSpectra, stats, tibble

Suggests arf, ggplot2, knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

NeedsCompilation no

Author Binh Duc Vu [aut, cre] (ORCID: <https://orcid.org/0009-0001-4552-5367>), Jan Kapar [aut] (ORCID: <https://orcid.org/0009-0000-6408-2840>), Marvin N. Wright [aut] (ORCID: <https://orcid.org/0000-0002-8542-6291>), David S. Watson [aut] (ORCID: <https://orcid.org/0000-0001-9632-2159>)

Repository CRAN

Date/Publication 2026-01-17 11:20:07 UTC

Contents

`decode_knn` *Decode RF Embeddings*

Description

Maps the low-dimensional KPCA embedding of a random forest back to the input space via iterative k-nearest neighbors.

Usage

```
decode_knn(rf, emap, z, x_tilde = NULL, k = 5, parallel = TRUE)
```

Arguments

`rf`	Pre-trained random forest object of class `ranger`.
`emap`	Spectral embedding learned via `eigenmap`.
`z`	Matrix of embedded data to map back to the input space.
`x_tilde`	Supplied training data, if none supplied then the RF is used to generate synthetic training data according to the eForest scheme. Default is NULL.
`k`	Number of nearest neighbors to evaluate.
`parallel`	Compute in parallel? Must register backend beforehand, e.g. via `doParallel`.

Details

`decode_knn` decodes the embedded data back to the original input space using a k-nearest neighbors (kNN) (Cover & Hart, 1967) approach. For a given embedding vector, decoding works by first finding the k nearest embeddings within the training set. Then, `x_tilde` is either supplied or generated from the RF (if generated, using the 'eForest' scheme (Feng & Zhou, 2018)), which provides a proxy for the training samples associated with these embeddings, to avoid needing to retain training data. Finally, data is reconstructed by weighted averaging for numerical features, and the most likely value for categorical features.

Value

Decoded dataset.

References

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory, 13*(1), 21–27.

Feng, J., & Zhou, Z. H. (2018, April). Autoencoder by forest. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 32 , No. 1).

Examples

```
# Set seed
set.seed(1)

# Split training and test
trn <- sample(1:nrow(iris), 100)
tst <- setdiff(1:nrow(iris), trn)

# Train RF, learn the encodings and project test points.
rf <- ranger::ranger(Species ~ ., data = iris[trn, ], num.trees=50)
emap <- encode(rf, iris[trn, ], k=2)
emb <- predict(emap, rf, iris[tst, ])

# Decode test samples back to the input space
out <- decode_knn(rf, emap, emb, k=5)$x_hat
```

---

encode *Encoding with Diffusion Maps*

Description

Computes the diffusion map of a random forest kernel, including a spectral decomposition and associated weights.

Usage

```
encode(rf, x, k = 5L, stepsize = 1L, parallel = TRUE)
```

Arguments

`rf`	Pre-trained random forest object of class `ranger`.
`x`	Training data for estimating embedding weights.
`k`	Dimensionality of the spectral embedding.
`stepsize`	Number of steps of a random walk for the diffusion process. See Details.
`parallel`	Compute in parallel? Must register backend beforehand, e.g. via `doParallel`.

Details

encode learns a low-dimensional embedding of the data implied by the adjacency matrix of the rf. Random forests can be understood as an adaptive nearest neighbors algorithm, where proximity between samples is determined by how often they are routed to the same leaves. We compute the spectral decomposition of the model adjacencies over the training data X, and take the leading k eigenvectors and eigenvalues. The function returns the resulting diffusion map, eigenvectors, eigenvalues, and leaf sizes.

Let K be the weighted adjacency matrix of code x implied by rf. This defines a weighted, undirected graph over the training data, which we can also interpret as the transitions of a Markov process 'between' data points. Spectral analysis produces the decomposition $K = V\lambda V^{-1}$, where we can take leading nonconstant eigenvectors. The diffusion map $Z = \sqrt{n}V\lambda^t$ (Coifman & Lafon, 2006) represents the long-run connectivity structure of the graph after t time steps of a Markov process, with some nice optimization properties (von Luxburg, 2007). We can embed new data into this space using the Nyström formula (Bengio et al., 2004).

Value

A list with eight elements: (1) Z: a k-dimensional nonlinear embedding of x implied by rf. (2) A: the normalized adjacency matrix (3) v: the leading k eigenvectors; (4) lambda: the leading k eigenvalues; (5) stepsize: the number of steps in the random walk. (6) leafIDs: a matrix with nrow(x) rows and rf$num.trees columns, representing the terminal nodes of each training sample in each tree; (7) the number of samples in each leaf; (8) metadata about the rf.

References

Bengio, Y., Delalleau, O., Le Roux, N., Paiement, J., Vincent, P., & Ouimet, M. (2004). Learning eigenfunctions links spectral embedding and kernel PCA. *Neural Computation, 16*(10): 2197-2219.

Coifman, R. R., & Lafon, S. (2006). Diffusion maps. *Applied and Computational Harmonic Analysis, 21*(1), 5–30.

von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and Computing, 17*(4), 395–416.

See Also

adversarial_rf

Examples

```
# Train ARF
arf <- arf::adversarial_rf(iris)

# Embed the data
emap <- encode(arf, iris)
```

post_x *Post-process data*

Description

This function prepares output data.

Usage

```
post_x(x, meta, round = TRUE)
```

Arguments

x	Input data.frame.
meta	Metadata.
round	Round continuous variables to their respective maximum precision in the real data set?

Value

A data.frame which follows the structure and ordering of the input dataset.

predict.encode *Predict Spectral Embeddings*

Description

Projects test data into the forest embedding space using a pre-trained encoding map.

Usage

```
## S3 method for class 'encode'
predict(object, rf, x, parallel = TRUE, ...)
```

Arguments

object	Spectral embedding for the rf learned via eigenmap.
rf	Pre-trained random forest object of class ranger.
x	Data to be embedded.
parallel	Compute in parallel? Must register backend beforehand, e.g. via doParallel.
...	Additional arguments passed to methods.

Details

This function uses the weights learned via `eigenmap` to project new data into the low-dimensional embedding space using the Nyström formula. For details, see Bengio et al. (2004).

Value

A matrix of embeddings, with `nrow(x)` rows and `k` columns, the latter argument used to learn the `eigenmap`.

References

Bengio, Y., Delalleau, O., Le Roux, N., Paiement, J., Vincent, P., & Ouimet, M. (2004). Learning eigenfunctions links spectral embedding and kernel PCA. *Neural Computation, 16*(10): 2197-2219.

See Also

`adversarial_rf`

Examples

```
# Set seed
set.seed(1)

# Split training and test
trn <- sample(1:nrow(iris), 100)
tst <- setdiff(1:nrow(iris), trn)

# Train RF. You can also use RF variants, such as the Adversarial Random
# Forests (ARF).
rf <- ranger::ranger(Species ~ ., data = iris[trn, ], num.trees=50)

# Learn the encodings, which are found using diffusion maps.
emap <- encode(rf, iris[trn, ], k=2)

# Embed test points
emb <- predict(emap, rf, iris[tst, ])
```

---

prep_x *Preprocess input data*

Description

This function prepares input data.

Usage

```
prep_x(x, to_numeric = NULL, to_factor = NULL, default = 5)
```

Arguments

x	Input data.frame.
to_numeric	List of variables to force as numeric.
to_factor	List of variables to force as factor.
default	Threshold to classify a variable as numeric (more than default unique values) or factor (less or equal to unique values).

Value

Preprocessed data.frame.

---

reconstruction_error *Mixed-type Reconstruction Error*

Description

Computes the reconstruction error of a decoded dataset compared to the original.

Usage

```
reconstruction_error(Xhat, X)
```

Arguments

Xhat	Reconstructed dataset
X	Ground truth dataset

Details

In standard AEs, reconstruction error is generally estimated via L_2 loss. This is not sensible with a mix of continuous and categorical data, so we devise a measure that evaluates distortion on continuous variables as $1 - R^2$, and categorical variables as prediction error.

Value

A list containing column-wise reconstruction error, and the average reconstruction error for categorical and numeric variables. Values lie between 0-1, where 0 represents perfect reconstruction, and 1 represents no reconstruction.

Examples

```
# Set seed
set.seed(1)

# Split training and test
trn <- sample(1:nrow(iris), 100)
tst <- setdiff(1:nrow(iris), trn)

# Train RF, learn the encodings and project test points.
rf <- ranger::ranger(Species ~ ., data = iris[trn, ], num.trees=50)
emap <- encode(rf, iris[trn, ], k=2)
emb <- predict(emap, rf, iris[tst, ])

# Decode test samples back to the input space
out <- decode_knn(rf, emap, emb, k=5)$x_hat

# Compute the reconstruction error
error <- reconstruction_error(out, iris[tst, ])
```

Index