Supporting information S1 for Pichler & Hartig – Machine Learning and Deep Learning – A review for Ecologists

Pichler & Hartig, 2022

Table of contents

**Summary:** This document provides supporting information on Pichler & Hartig – Machine Learning and Deep Learning – A review for ecologists. The first section describes the methods for the trend analysis and the word clouds, the second section demnstrates the application of common supervised Machine learning and Deep learning algorithms in the R programming languages based on code examples.

## 0.1 Trend analysis

For the global trend analysis in Figure 1, we used the R package ‘europepmc’ (v0.4.1, Jahn (2021)) to search from 1920 to 2021 the PubMed and Medline NLM databases. We used the following queries ‘deep learning’, (‘machine learning’ OR ‘machine-learning’), and (‘p value’ OR ‘p-value’ OR ‘statistically significant) as representatives for Deep Learning, Machine Learning, and classical statistical approaches. The number of hits were normalized by total hits in each year. For the stream charts in Figure 2, we used the search queries Table S1 and added them to the query’ AND (“ecology” OR “ecolog\*” OR “evolution”) to restrict the queries to hits from the ecology and evolution field.

| Queries | ML and DL algorithm |
| --- | --- |
| (“artificial neural network” OR “deep neural network” OR |  |
| “multi-layer perceptron” OR “fully connected neural network”) | Deep neural network (ANN) |
| (“convolutional neural network” OR “object detection”) | Convolutional neural network (CNN) |
| (“recurrent neural network”) | Recurrent neural network (RNN) |
| (“graph neural network” OR “graph convolutional”) | Graph neural network (GNN) |
| (“random forest”) | Random Forest (RF) |
| (“boosted regression tree” OR “boosted reg” OR |  |
| “gradient boosting” OR “adaboost”) | Boosted Regression Trees (BRT) |
| (“k-nearest-neighbor”) | k-nearest neighbor (kNN) |
| (“ridge regression” OR “lasso regression” OR |  |
| “elastic-net” OR “elastic net”) | Ridge, lasso, or elastic-net regression |
| (“support vector machine” OR “support vector”) | Support vector machine (SVM) |

Search queries and their corresponding ML and DL algorithm.

For the word clouds in Table 1, we used again the ‘europepmc’ R package to search abstracts and titles within the ML and DL algorithm specific queries (Table 1) for the following ecological keywords: species distribution, species interaction, mortality, remote sensing, invasive, decision making, ecosystem, species identification, species detection, extinction, functional trait, ecological network, biodiversity, and camera trap.

We used the R packages ‘tm’ (Feinerer, Hornik, and Meyer (2008)), ‘wordcloud’ (Fellows (2018)) ,and ‘wordcloud2’ (Lang and Chien (2018)) to analyze and create the final word cloud plots.

## 0.2 Algorithms

The goal of this chapter is to provide short code examples for all common supervised ML algorithms. All examples are shown in the R programming language and are demonstrated at the example of the iris dataset (4 continuous and 1 nominal variables). We demonstrate a) how to apply the algorithms on classification tasks (response = nominal, three species) and how to generate class specific (quasi) probability predictions and b) how to apply the algorithms to regression tasks (response = continuous) and make continuous predictions.

# 1. Ridge, LASSO, and elastic-net regression

We can use the ‘glmnet’ R package (Simon et al. (2011)) for Ridge, LASSO, or elastic-net regularization. The ‘glmnet’ package supports different response families including ‘gaussian’, ‘binomial’ and ‘Poisson’. The strength of the regularization is set by the ‘lambda’ argument () and the weighting between Ridge and LASSO regularization by the ‘alpha’ parameter ():

Setting alpha = 0 turns off the LASSO and alpha = 1 the Ridge. Alphas between (0,1) will use both regularization types, turning the model into an elastic-net regularization.

When using regularization, it is important to scale all features otherwise effects for features that are on a larger scale are stronger regularized.

In python, the ‘scikit-learn’ package provides an interface for many different ML algorithms, including elastic-net regression models (Pedregosa et al. (2011)).

The ‘MLJ’ package provides a generic interface for different ML algorithms (Blaom et al. (2019)). Elastic-net regression models can be accessed via the ‘MLJLinearModels’ package within MLJ.

## 1.1 Classification

Build models (for regularization it is important to scale the features):

#### 1.1.0.1 R

library(glmnet)  
X = scale(iris[,1:4])  
Y = iris$Species  
  
# Ridge:  
ridge = glmnet(X, Y, family = "multinomial", alpha = 0, lambda = 0.01)  
  
# LASSO:  
lasso = glmnet(X, Y, family = "multinomial", alpha = 1, lambda = 0.01)  
  
# Elastic-net:  
elastic = glmnet(X, Y, family = "multinomial", alpha = 0.5, lambda = 0.01)

Make predictions (class probabilities):

head(predict(lasso, newx = X, type = "response")[,,1], n = 3)

setosa versicolor virginica  
[1,] 0.9858987 0.01410131 3.438452e-09  
[2,] 0.9668897 0.03311031 1.397684e-08  
[3,] 0.9815369 0.01846312 5.279315e-09

#### 1.1.0.2 Python

In the sklearn implementation the regularization strength parameter ‘C’ corresponds to the lambda parameter from glmnet:

from sklearn.linear\_model import LogisticRegression  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target

Models:

# Ridge:  
ridge = LogisticRegression(multi\_class='multinomial',   
 penalty = "l2",   
 C = 0.01,   
 solver="saga")  
ridge.fit(X, Y)  
  
# LASSO:

LogisticRegression(C=0.01, multi\_class='multinomial', solver='saga')

lasso = LogisticRegression(multi\_class='multinomial',   
 penalty = "l1",   
 C = 0.01,   
 solver="saga")  
lasso.fit(X, Y)  
  
# Elastic-net:

LogisticRegression(C=0.01, multi\_class='multinomial', penalty='l1',  
 solver='saga')

elastic = LogisticRegression(multi\_class='multinomial',   
 penalty = "elasticnet",   
 C = 0.01,   
 l1\_ratio=0.5,   
 solver="saga")  
elastic.fit(X, Y)

LogisticRegression(C=0.01, l1\_ratio=0.5, multi\_class='multinomial',  
 penalty='elasticnet', solver='saga')

Predictions (probabilities):

lasso.predict\_proba(X)[0:5,:]

array([[0.33523502, 0.30150185, 0.36326313],  
 [0.33523502, 0.30150185, 0.36326313],  
 [0.33523502, 0.30150185, 0.36326313],  
 [0.33523502, 0.30150185, 0.36326313],  
 [0.33523502, 0.30150185, 0.36326313]])

#### 1.1.0.3 Julia

import StatsBase;  
using MLJ;  
using MLJLinearModels;  
@load MultinomialClassifier pkg=MLJLinearModels;  
using RDatasets;  
using StatsBase;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 1:4]);  
Y = iris[:, 5];

Models:

# Ridge  
ridge = fit!(machine(MultinomialClassifier(lambda = 0.01, penalty = "l2"), X, Y));  
  
# Lasso  
lasso = fit!(machine(MultinomialClassifier(lambda = 0.01, penalty = "l1"), X, Y));  
  
  
# Elastic-net  
elastic = fit!(machine(MultinomialClassifier(lambda = 0.01, gamma = 0.01, penalty = "en"), X, Y));

Predictions:

MLJ.predict(lasso, X)[1:5]

5-element CategoricalDistributions.UnivariateFiniteVector{Multiclass{3}, String, UInt8, Float64}:  
 UnivariateFinite{Multiclass{3}}(setosa=>0.992, versicolor=>0.00829, virginica=>2.2e-9)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.969, versicolor=>0.031, virginica=>1.4e-8)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.985, versicolor=>0.0149, virginica=>4.6e-9)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.971, versicolor=>0.0293, virginica=>1.41e-8)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.994, versicolor=>0.00635, virginica=>1.51e-9)

## 1.2 Regression

#### 1.2.0.1 R

X = scale(iris[,2:4])  
Y = iris[,1]  
  
# Ridge:  
ridge = glmnet(X, Y, family = gaussian(), alpha = 0, lambda = 0.01)  
  
# LASSO:  
lasso = glmnet(X, Y, family = gaussian(), alpha = 1, lambda = 0.01)  
  
# Elastic-net:  
elastic = glmnet(X, Y, family = gaussian(), alpha = 0.5, lambda = 0.01)

Make predictions (class probabilities):

head(predict(lasso, newx = X), n = 3)

s0  
[1,] 5.006484  
[2,] 4.720600  
[3,] 4.781548

#### 1.2.0.2 Python

For regressions we can use the ElasticNet model class, here, however, lambda corresponds to alpha and l1\_ratio to the alpha parameter.

from sklearn.linear\_model import ElasticNet  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
  
# Ridge:  
ridge = ElasticNet(alpha = 0.01,  
 l1\_ratio = 0.011)  
ridge.fit(X, Y)  
  
# LASSO:

ElasticNet(alpha=0.01, l1\_ratio=0.011)

lasso = ElasticNet(alpha = 0.01,  
 l1\_ratio = 1.0)  
lasso.fit(X, Y)  
  
# Elastic-net:

ElasticNet(alpha=0.01, l1\_ratio=1.0)

elastic = ElasticNet(alpha = 0.01,  
 l1\_ratio = 0.5)  
elastic.fit(X, Y)  
  
# Make predictions:

ElasticNet(alpha=0.01)

lasso.predict(X)[0:10]

array([5.0064384 , 4.72032938, 4.78125162, 4.83107256, 5.06366021,  
 5.36149937, 4.93202142, 5.00273797, 4.66310758, 4.84826774])

#### 1.2.0.3 Julia

import StatsBase;  
using MLJ;  
using MLJLinearModels;  
@load LassoRegressor pkg=MLJLinearModels;  
@load RidgeRegressor pkg=MLJLinearModels;  
@load ElasticNetRegressor pkg=MLJLinearModels;  
using RDatasets;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 2:4]);  
Y = iris[:, 1];

Models:

# Ridge  
ridge = fit!(machine(RidgeRegressor(lambda = 0.01), X, Y));  
  
# Lasso  
lasso = fit!(machine(LassoRegressor(lambda = 0.01), X, Y));  
  
  
# Elastic-net  
elastic = fit!(machine(ElasticNetRegressor(lambda = 0.01, gamma = 0.01), X, Y));

Predictions (probabilities):

MLJ.predict(lasso, X)[1:5]

5-element Vector{Float64}:  
 5.007709152258313  
 4.711530001523257  
 4.770100849643125  
 4.830666643844422  
 5.0669449824053245

# 2. Support Vector Machines

The support vector machine (SVM) algorithm estimates hyper-planes to separate our response species. In the following we use the ‘e1071’ package which supports a variety of different SVM algorithms (Meyer et al. (2022)) (Python: ‘scikit-learn’ (Pedregosa et al. (2011)), Julia: ‘MLJ’ (Blaom et al. (2019))).

## 2.1 Classification

#### 2.1.0.1 R

library(e1071)  
X = scale(iris[,1:4])  
Y = iris$Species  
  
sv = svm(X, Y, probability = TRUE)   
summary(sv)

Call:  
svm.default(x = X, y = Y, probability = TRUE)  
  
  
Parameters:  
 SVM-Type: C-classification   
 SVM-Kernel: radial   
 cost: 1   
  
Number of Support Vectors: 51  
  
 ( 8 22 21 )  
  
  
Number of Classes: 3   
  
Levels:   
 setosa versicolor virginica

Make predictions (class probabilities):

head(attr(predict(sv, newdata = X, probability = TRUE), "probabilities"), n = 3)

setosa versicolor virginica  
1 0.9804478 0.01112896 0.008423274  
2 0.9730746 0.01787024 0.009055150  
3 0.9791601 0.01175270 0.009087235

#### 2.1.0.2 Python

from sklearn import svm  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target  
  
model = svm.SVC(probability=True).fit(X, Y)  
  
# Make predictions (class probabilities):  
  
model.predict\_proba(X)[0:10,:]

array([[0.98064837, 0.01083985, 0.00851178],  
 [0.97368258, 0.01717545, 0.00914197],  
 [0.97944037, 0.01137462, 0.00918501],  
 [0.97556873, 0.01456613, 0.00986514],  
 [0.97981441, 0.01117669, 0.0090089 ],  
 [0.97458032, 0.01605555, 0.00936413],  
 [0.97612616, 0.01293974, 0.0109341 ],  
 [0.9805555 , 0.01094575, 0.00849875],  
 [0.96713676, 0.02134948, 0.01151375],  
 [0.97731269, 0.01341187, 0.00927545]])

#### 2.1.0.3 Julia

import StatsBase;  
using MLJ;  
SVM\_classifier = @load NuSVC pkg=LIBSVM;  
using RDatasets;  
using StatsBase;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 1:4]);  
Y = iris[:, 5];

Models:

model = fit!(machine(SVM\_classifier(), X, Y))

trained Machine; caches model-specific representations of data  
 model: NuSVC(kernel = RadialBasis, …)  
 args:   
 1: Source @926 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @536 ⏎ AbstractVector{Multiclass{3}}

Predictions:

MLJ.predict(model, X)[1:5]

5-element CategoricalArrays.CategoricalArray{String,1,UInt8}:  
 "setosa"  
 "setosa"  
 "setosa"  
 "setosa"  
 "setosa"

## 2.2 Regression

#### 2.2.0.1 R

library(e1071)  
X = scale(iris[,2:4])  
Y = iris[,1]  
  
sv = svm(X, Y)   
summary(sv)

Call:  
svm.default(x = X, y = Y)  
  
  
Parameters:  
 SVM-Type: eps-regression   
 SVM-Kernel: radial   
 cost: 1   
 gamma: 0.3333333   
 epsilon: 0.1   
  
  
Number of Support Vectors: 124

Make predictions (class probabilities):

head(predict(sv, newdata = X), n = 3)

1 2 3   
5.042085 4.711768 4.836291

#### 2.2.0.2 Python

from sklearn import svm  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
model = svm.SVR().fit(X, Y)  
  
# Make predictions:  
  
model.predict(X)[0:10]

array([5.03583855, 4.69496586, 4.81438855, 4.77951854, 5.10018373,  
 5.29981857, 4.97308737, 4.98199033, 4.63701656, 4.78431078])

#### 2.2.0.3 Julia

import StatsBase;  
using MLJ;  
SVM\_regressor = @load NuSVR pkg=LIBSVM;  
using RDatasets;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 2:4]);  
Y = iris[:, 1];

Model:

model = fit!(machine(SVM\_regressor(), X, Y))

trained Machine; caches model-specific representations of data  
 model: NuSVR(kernel = RadialBasis, …)  
 args:   
 1: Source @935 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @838 ⏎ AbstractVector{Continuous}

Predictions:

MLJ.predict(model, X)[1:5]

5-element Vector{Float64}:  
 5.058471741834634  
 4.6717512552719604  
 4.799641470830148  
 4.75734816087994  
 5.133728219775252

# 3. k-nearest-neighbor

The k-nearest-neighbor algorithm doesn’t really learn from the data, predictions for new observations are made based on the class affiliation (or response value) of the nearest neighbors, e.g. by majority voting or averaging. The nearest neighbors are found by calculating the distance of the new observation to all observations in the train dataset.

In the following we use the ‘kknn’ package (Schliep and Hechenbichler (2016)) (Python: ‘scikit-learn’ (Pedregosa et al. (2011)), Julia: ‘MLJ’ (Blaom et al. (2019))). Different to other ML packages we can provide here already the test dataset in the fit function.

## 3.1 Classification

#### 3.1.0.1 R

library(kknn)  
X = scale(iris[,1:4])  
Y = iris[,5,drop=FALSE]  
data = cbind(Y, X)  
  
knn = kknn(Species~., train = data, test = data)

Make predictions (class probabilities):

head(knn$prob, n = 3)

setosa versicolor virginica  
[1,] 1 0 0  
[2,] 1 0 0  
[3,] 1 0 0

#### 3.1.0.2 Python

from sklearn.neighbors import KNeighborsClassifier  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target  
  
model = KNeighborsClassifier().fit(X, Y)  
  
# Make predictions:  
  
model.predict\_proba(X)[0:10,:]

array([[1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.]])

#### 3.1.0.3 Julia

import StatsBase;  
using MLJ;  
kNN\_classifier = @load KNNClassifier pkg=NearestNeighborModels;  
using RDatasets;  
using StatsBase;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 1:4]);  
Y = iris[:, 5];

Models:

model = fit!(machine(kNN\_classifier(), X, Y))

trained Machine; caches model-specific representations of data  
 model: KNNClassifier(K = 5, …)  
 args:   
 1: Source @488 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @201 ⏎ AbstractVector{Multiclass{3}}

Predictions:

MLJ.predict(model, X)[1:5]

5-element CategoricalDistributions.UnivariateFiniteVector{Multiclass{3}, String, UInt8, Float64}:  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)

## 3.2 Regression

#### 3.2.0.1 R

library(e1071)  
X = scale(iris[,2:4])  
data = cbind(iris[,1,drop=FALSE], X)  
  
knn = kknn(Sepal.Length~., train = data, test = data)

Make predictions (class probabilities):

head(predict(knn), n = 3)

[1] 5.188492 4.739986 4.685332

#### 3.2.0.2 Python

from sklearn.neighbors import KNeighborsRegressor  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
model = KNeighborsRegressor().fit(X, Y)  
  
# Make predictions:  
  
model.predict(X)[0:10]

array([5.18, 4.78, 4.68, 4.76, 4.98, 5.34, 5.06, 5.1 , 4.7 , 4.8 ])

#### 3.2.0.3 Julia

import StatsBase;  
using MLJ;  
kNN\_regressor = @load KNNRegressor pkg=NearestNeighborModels;  
using RDatasets;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 2:4]);  
Y = iris[:, 1];

Model:

model = fit!(machine(kNN\_regressor(), X, Y))

trained Machine; caches model-specific representations of data  
 model: KNNRegressor(K = 5, …)  
 args:   
 1: Source @677 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @876 ⏎ AbstractVector{Continuous}

Predictions:

MLJ.predict(model, X)[1:5]

5-element Vector{Float64}:  
 5.18  
 4.779999999999999  
 4.68  
 4.82  
 5.0200000000000005

# 4. Random forest

The random forest (RF) algorithm is probably one of the most famous ML algorithms, and not without reason. Compared to other well performing algorithms, the RF algorithm has only a few hyper-parameters and because of the bagging and the random sampling of available variables in for the node splits, it has a well working internal complexity adaption.

In the following, we use the ‘ranger’ package (Wright and Ziegler (2017)) (Python: ‘scikit-learn’ (Pedregosa et al. (2011)), Julia: ‘MLJ’ (Blaom et al. (2019))).

## 4.1 Classification

#### 4.1.0.1 R

library(ranger)  
X = iris[,1:4]  
Y = iris[,5,drop=FALSE]  
data = cbind(Y, X)  
  
rf = ranger(Species~., data = data, probability = TRUE, importance = "impurity")

Show feature importances:

importance(rf)

Sepal.Length Sepal.Width Petal.Length Petal.Width   
 8.185696 1.419584 43.166251 43.083224

Make predictions (class probabilities):

head(predict(rf, data = data)$predictions, n = 3)

setosa versicolor virginica  
[1,] 1.0000000 0.00000 0.0000000000  
[2,] 0.9982778 0.00125 0.0004722222  
[3,] 1.0000000 0.00000 0.0000000000

#### 4.1.0.2 Python

from sklearn.ensemble import RandomForestClassifier  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target  
  
model = RandomForestClassifier().fit(X, Y)

Feature importance

print(model.feature\_importances\_)

[0.09564264 0.02873799 0.45808523 0.41753414]

Make predictions:

model.predict\_proba(X)[0:10,:]

array([[1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.],  
 [1., 0., 0.]])

#### 4.1.0.3 Julia

import StatsBase;  
using MLJ;  
RF\_classifier = @load RandomForestClassifier pkg=DecisionTree;  
using RDatasets;  
using StatsBase;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 1:4]);  
Y = iris[:, 5];

Models:

model = fit!(machine(RF\_classifier(), X, Y))

trained Machine; caches model-specific representations of data  
 model: RandomForestClassifier(max\_depth = -1, …)  
 args:   
 1: Source @857 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @360 ⏎ AbstractVector{Multiclass{3}}

Feature importance:

feature\_importances(model)

4-element Vector{Pair{Symbol, Float64}}:  
 :PetalWidth => 0.7027784291025588  
 :PetalLength => 0.19029600810272967  
 :SepalLength => 0.089917282879327  
 :SepalWidth => 0.017008279915384585

Predictions:

MLJ.predict(model, X)[1:5]

5-element CategoricalDistributions.UnivariateFiniteVector{Multiclass{3}, String, UInt8, Float64}:  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)  
 UnivariateFinite{Multiclass{3}}(setosa=>1.0, versicolor=>0.0, virginica=>0.0)

## 4.2 Regression

#### 4.2.0.1 R

library(ranger)  
X = iris[,2:4]  
data = cbind(iris[,1,drop=FALSE], X)  
  
rf = ranger(Sepal.Length~., data = data, importance = "impurity")

Show feature importances:

importance(rf)

Sepal.Width Petal.Length Petal.Width   
 12.94158 45.22093 37.71692

Make predictions (class probabilities):

head(predict(rf, data = data)$predictions, n = 3)

[1] 5.122784 4.756676 4.658971

#### 4.2.0.2 Python

from sklearn.ensemble import RandomForestRegressor  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
model = RandomForestRegressor().fit(X, Y)

Feature importance:

print(model.feature\_importances\_)

[0.08018641 0.85115054 0.06866305]

Make predictions:

model.predict(X)[0:10]

array([5.11 , 4.8015 , 4.55519286, 4.76646429, 5.037 ,  
 5.407 , 4.808 , 5.05981667, 4.5615 , 4.86225714])

#### 4.2.0.3 Julia

import StatsBase;  
using MLJ;  
RF\_regressor = @load RandomForestRegressor pkg=DecisionTree;  
using RDatasets;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 2:4]);  
Y = iris[:, 1];

Model:

model = fit!(machine(RF\_regressor(), X, Y))

trained Machine; caches model-specific representations of data  
 model: RandomForestRegressor(max\_depth = -1, …)  
 args:   
 1: Source @676 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @981 ⏎ AbstractVector{Continuous}

Feature importance:

feature\_importances(model)

3-element Vector{Pair{Symbol, Float64}}:  
 :PetalLength => 0.6126470157766716  
 :PetalWidth => 0.3154759300487962  
 :SepalWidth => 0.07187705417453213

Predictions:

MLJ.predict(model, X)[1:5]

5-element Vector{Float64}:  
 4.99  
 4.8999999999999995  
 4.3999999999999995  
 4.8999999999999995  
 4.99

# 5. Boosted gradient trees

Boosted gradient machines achieve currently state-of-the-art performance for structured (tabular) data which makes them probably one of the most important algorithms for E&E where structured data dominates the field.

In the following, we use the ‘xgboost’ package (Chen et al. (2022)) (Python: ‘xgboost’ (Chen et al. (2022)), Julia: ‘MLJ’ (Blaom et al. (2019)))

## 5.1 Classification

#### 5.1.0.1 R

library(xgboost)  
X = as.matrix(iris[,1:4])  
Y = as.integer(iris[,5]) - 1 # classes must be integers starting from 0  
  
xgdata = xgb.DMatrix(X, label = Y)  
  
# nrounds = number of trees in the ensemble  
brt = xgboost(data = xgdata,   
 objective="multi:softprob",   
 nrounds = 50,   
 num\_class = 3,  
 verbose = 0)

Show feature importances:

xgb.importance(model = brt)

Feature Gain Cover Frequency  
1: Petal.Length 0.671879438 0.57441039 0.3792049  
2: Petal.Width 0.311535837 0.29261084 0.3088685  
3: Sepal.Width 0.010177107 0.04910115 0.1162080  
4: Sepal.Length 0.006407618 0.08387763 0.1957187

Make predictions (class probabilities):

head(matrix(predict(brt, newdata = xgb.DMatrix(X)), ncol =3), n = 3)

[,1] [,2] [,3]  
[1,] 0.995287061 0.002195822 0.001027058  
[2,] 0.003323558 0.995396435 0.001592265  
[3,] 0.001389398 0.002407764 0.997380674

#### 5.1.0.2 Python

import xgboost as xgb  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target  
  
# Parameters:  
param = {  
 'max\_depth':2,   
 'eta':1,   
 'objective':'multi:softmax' }  
num\_round = 50  
  
model = xgb.XGBClassifier(param, num\_round, verbosity = 0).fit(X, Y)

Feature importance

model.feature\_importances\_

array([0.00959796, 0.01645038, 0.6765859 , 0.29736578], dtype=float32)

Make predictions:

model.predict\_proba(X)[0:10,:]

array([[9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9636227e-01, 2.3820533e-03, 1.2557388e-03],  
 [9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9679452e-01, 2.3830866e-03, 8.2237815e-04],  
 [9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9680281e-01, 2.3831066e-03, 8.1413286e-04],  
 [9.9636227e-01, 2.3820533e-03, 1.2557388e-03],  
 [9.9679452e-01, 2.3830866e-03, 8.2237815e-04]], dtype=float32)

#### 5.1.0.3 Julia

import StatsBase;  
using MLJ;  
BRT\_classifier = @load XGBoostClassifier pkg=XGBoost;  
using RDatasets;  
using StatsBase;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 1:4]);  
Y = iris[:, 5];

Models:

model = fit!(machine(BRT\_classifier(), X, Y))

trained Machine; caches model-specific representations of data  
 model: XGBoostClassifier(num\_round = 100, …)  
 args:   
 1: Source @488 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @574 ⏎ AbstractVector{Multiclass{3}}

Predictions:

MLJ.predict(model, X)[1:5]

5-element CategoricalDistributions.UnivariateFiniteVector{Multiclass{3}, String, UInt8, Float32}:  
 UnivariateFinite{Multiclass{3}}(setosa=>0.997, versicolor=>0.00238, virginica=>0.000814)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.996, versicolor=>0.00238, virginica=>0.00126)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.997, versicolor=>0.00238, virginica=>0.000814)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.997, versicolor=>0.00238, virginica=>0.000822)  
 UnivariateFinite{Multiclass{3}}(setosa=>0.997, versicolor=>0.00238, virginica=>0.000814)

## 5.2 Regression

#### 5.2.0.1 R

library(xgboost)  
X = as.matrix(iris[,2:4])  
Y = iris[,1]  
  
xgdata = xgb.DMatrix(X, label = Y)  
  
# nrounds = number of trees in the ensemble  
brt = xgboost(data = xgdata,   
 objective="reg:squarederror",   
 nrounds = 50,   
 verbose = 0)

Show feature importances:

xgb.importance(model = brt)

Feature Gain Cover Frequency  
1: Petal.Length 0.86781219 0.4789538 0.3789062  
2: Petal.Width 0.06987880 0.2128402 0.2626953  
3: Sepal.Width 0.06230901 0.3082060 0.3583984

Make predictions:

head(predict(brt, newdata = xgb.DMatrix(X), n = 3))

[16:00:18] WARNING: amalgamation/../src/c\_api/c\_api.cc:785: `ntree\_limit` is deprecated, use `iteration\_range` instead.

[1] 3.506606 3.506606 3.506606 3.506606 3.506606 3.506606

#### 5.2.0.2 Python

import xgboost as xgb  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
# Parameters:  
model = xgb.XGBRegressor(  
 objective = 'reg:squarederror',  
 max\_depth = 2,   
 n\_estimators = 50,   
 verbosity = 0).fit(X, Y)

Feature importance:

print(model.feature\_importances\_)

[0.08471056 0.835755 0.07953447]

Make predictions:

model.predict(X)[0:10]

array([5.0407157, 4.6844926, 4.711238 , 4.917956 , 5.0407157, 5.450946 ,  
 4.928966 , 4.986462 , 4.6750975, 4.917956 ], dtype=float32)

#### 5.2.0.3 Julia

import StatsBase;  
using MLJ;  
BRT\_regressor = @load XGBoostRegressor pkg=XGBoost;  
using RDatasets;  
using DataFrames;

iris = dataset("datasets", "iris");  
X = mapcols(StatsBase.zscore, iris[:, 2:4]);  
Y = iris[:, 1];

Model:

model = fit!(machine(BRT\_regressor(), X, Y))

trained Machine; caches model-specific representations of data  
 model: XGBoostRegressor(num\_round = 100, …)  
 args:   
 1: Source @598 ⏎ Table{AbstractVector{Continuous}}  
 2: Source @306 ⏎ AbstractVector{Continuous}

Predictions:

MLJ.predict(model, X)[1:5]

5-element Vector{Float32}:  
 5.1509466  
 4.8569074  
 4.551141  
 4.7587333  
 4.999504

# 6. Deep neural networks

Deep neural networks, or more precisely here fully connected neural networks, can be flexibly built which makes their application more challenging than other ML algorithms.

In the following, we use the ‘keras’ (Allaire and Chollet (2022); Chollet et al. (2015)) (Python: ‘keras’ (Chollet et al. (2015)); Julia: ‘Flux’ (Innes et al. (2018))) package which is a higher level API on the python ‘tensorflow’ framework (Abadi et al. (2016)).

## 6.1 Classification

#### 6.1.0.1 R

library(keras)  
X = scale(as.matrix(iris[,1:4]))  
Y = as.integer(iris$Species)  
# We need to one hot encode our response classes  
YT = k\_one\_hot(Y-1L, num\_classes = 3)  
  
DNN = keras\_model\_sequential() %>%   
 # first hidden layer  
 layer\_dense(input\_shape = ncol(X),   
 units = 10,   
 activation = "relu") %>%   
 # second hidden layer with regularization  
 layer\_dense(units = 20,   
 activation = "relu",  
 kernel\_regularizer = regularizer\_l1()) %>%   
 # output layer, 3 output neurons for our three classes  
 # and softmax activation to get quasi probabilities   
 # that sum up to 1 for each observation  
 layer\_dense(units = 3,   
 activation = "softmax")  
  
# print architecture  
summary(DNN)

Model: "sequential"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
================================================================================  
 dense\_2 (Dense) (None, 10) 50   
 dense\_1 (Dense) (None, 20) 220   
 dense (Dense) (None, 3) 63   
================================================================================  
Total params: 333  
Trainable params: 333  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# add loss function and optimizer  
DNN %>%   
 compile(loss = loss\_categorical\_crossentropy,  
 optimizer = optimizer\_adamax(0.01))  
  
# train model  
DNN %>%   
 fit(X, YT, epochs = 50, verbose = 0)

Make predictions (class probabilities):

head(predict(DNN, X), n = 3)

[,1] [,2] [,3]  
[1,] 0.9925383 0.007370423 9.125301e-05  
[2,] 0.9797051 0.020253975 4.086332e-05  
[3,] 0.9931964 0.006752769 5.078966e-05

#### 6.1.0.2 Python

from tensorflow import keras  
from tensorflow.keras.layers import \*  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
X = scale(iris.data)  
Y = iris.target  
  
# We need to one hot encode our response classes  
YT = keras.utils.to\_categorical(Y, num\_classes = 3)  
  
DNN = keras.Sequential()  
 # first hidden layer  
DNN.add(Dense(  
 input\_shape=[X.shape[1]],   
 units = 10,   
 activation = "relu"))   
 # second hidden layer with regularization  
DNN.add(Dense(  
 units = 20,   
 activation = "relu",  
 kernel\_regularizer = keras.regularizers.l1()))  
 # output layer, 3 output neurons for our three classes  
 # and softmax activation to get quasi probabilities   
 # that sum up to 1 for each observation  
DNN.add(Dense(  
 units = 3,   
 activation = "softmax"))  
  
# print architecture  
DNN.summary()  
  
# add loss function and optimizer

Model: "sequential\_1"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 dense\_3 (Dense) (None, 10) 50   
   
 dense\_4 (Dense) (None, 20) 220   
   
 dense\_5 (Dense) (None, 3) 63   
   
=================================================================  
Total params: 333  
Trainable params: 333  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DNN.compile(loss = keras.losses.categorical\_crossentropy,  
 optimizer = keras.optimizers.Adamax(0.01))  
  
# train model  
DNN.fit(X, YT, epochs = 50, verbose = 0)

<keras.callbacks.History object at 0x7fb9c77cfdc0>

Make predictions:

DNN.predict(X)[0:10,:]

1/5 [=====>........................] - ETA: 0s  
5/5 [==============================] - 0s 698us/step  
array([[9.9766332e-01, 2.1197675e-03, 2.1679739e-04],  
 [9.8416871e-01, 1.5420077e-02, 4.1104638e-04],  
 [9.9686289e-01, 2.9126892e-03, 2.2447560e-04],  
 [9.9516070e-01, 4.5454120e-03, 2.9384211e-04],  
 [9.9886596e-01, 9.6661545e-04, 1.6737207e-04],  
 [9.9814212e-01, 1.5376722e-03, 3.2015322e-04],  
 [9.9869108e-01, 1.0995086e-03, 2.0941059e-04],  
 [9.9676615e-01, 2.9762946e-03, 2.5754355e-04],  
 [9.9318153e-01, 6.5093539e-03, 3.0912014e-04],  
 [9.9132729e-01, 8.3744926e-03, 2.9821257e-04]], dtype=float32)

#### 6.1.0.3 Julia

import StatsBase  
using RDatasets  
using StatsBase  
using DataFrames  
import MLJBase.int  
using Flux, Statistics  
using Flux.Data: DataLoader  
using Flux: onehotbatch, onecold, @epochs  
using Flux.Losses: logitcrossentropy

Data preparation:

iris = dataset("datasets", "iris");  
X = transpose(Matrix(mapcols(StatsBase.zscore, iris[:, 1:4])));  
Y = int(iris[:, 5], type = Int);  
classes = sort(unique(Y));  
YT = onehotbatch(Y, classes);  
data\_loader = DataLoader((X, YT), batchsize=10, shuffle=true);

Create model (similar to Keras):

model = Chain(  
 Dense(4, 20, relu),  
 Dense(20, 20, relu),  
 Dense(20, 3)  
)

Chain(  
 Dense(4 => 20, relu), # 100 parameters  
 Dense(20 => 20, relu), # 420 parameters  
 Dense(20 => 3), # 63 parameters  
) # Total: 6 arrays, 583 parameters, 2.652 KiB.

Train/optimize Model:

parameters = Flux.params(model);  
optimizer = ADAM(0.01);  
  
# Help functions  
loss(x, y) = logitcrossentropy(model(x), y);  
  
get\_loss() = @show sum(logitcrossentropy(model(X), YT));  
  
## Training  
for epoch in 1:20  
 Flux.train!(loss, parameters, data\_loader, optimizer, cb = Flux.throttle(get\_loss, 5))  
end

sum(logitcrossentropy(model(X), YT)) = 1.0953894766955314  
sum(logitcrossentropy(model(X), YT)) = 0.4263538406265198  
sum(logitcrossentropy(model(X), YT)) = 0.25887544110545124  
sum(logitcrossentropy(model(X), YT)) = 0.16457717750624204  
sum(logitcrossentropy(model(X), YT)) = 0.1195979307823707  
sum(logitcrossentropy(model(X), YT)) = 0.0827330103796263  
sum(logitcrossentropy(model(X), YT)) = 0.0694383287246746  
sum(logitcrossentropy(model(X), YT)) = 0.05777151167121077  
sum(logitcrossentropy(model(X), YT)) = 0.051968173257980456  
sum(logitcrossentropy(model(X), YT)) = 0.04482071628752984  
sum(logitcrossentropy(model(X), YT)) = 0.05517545921764145  
sum(logitcrossentropy(model(X), YT)) = 0.04026439174969125  
sum(logitcrossentropy(model(X), YT)) = 0.038823221265110584  
sum(logitcrossentropy(model(X), YT)) = 0.0371395810324732  
sum(logitcrossentropy(model(X), YT)) = 0.040059074804427504  
sum(logitcrossentropy(model(X), YT)) = 0.035628827115435024  
sum(logitcrossentropy(model(X), YT)) = 0.03545859831885094  
sum(logitcrossentropy(model(X), YT)) = 0.1066915874671683  
sum(logitcrossentropy(model(X), YT)) = 0.04959715920131334  
sum(logitcrossentropy(model(X), YT)) = 0.049808973487405386

Predictions:

transpose(softmax(model(X)))[1:5,:]

5×3 Matrix{Float64}:  
 0.999943 5.68508e-5 7.98199e-11  
 0.999671 0.000328977 2.35956e-9  
 0.999971 2.89877e-5 1.20405e-10  
 0.999948 5.18364e-5 3.66296e-10  
 0.999979 2.14537e-5 2.27473e-11

## 6.2 Regression

#### 6.2.0.1 R

library(keras)  
X = scale(as.matrix(iris[,2:4]))  
Y = as.matrix(iris[,1,drop=FALSE])  
  
DNN = keras\_model\_sequential() %>%   
 # first hidden layer  
 layer\_dense(input\_shape = ncol(X),   
 units = 10,   
 activation = "relu") %>%   
 # second hidden layer with regularization  
 layer\_dense(units = 20,   
 activation = "relu",  
 kernel\_regularizer = regularizer\_l1()) %>%   
 # output layer, one output neuron for one response  
 # and no activation function  
 layer\_dense(units = 1)  
  
# print architecture  
summary(DNN)

Model: "sequential\_2"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
================================================================================  
 dense\_8 (Dense) (None, 10) 40   
 dense\_7 (Dense) (None, 20) 220   
 dense\_6 (Dense) (None, 1) 21   
================================================================================  
Total params: 281  
Trainable params: 281  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# add loss function and optimizer  
DNN %>%   
 compile(loss = loss\_mean\_squared\_error,  
 optimizer = optimizer\_adamax(0.01))  
  
# train model  
DNN %>%   
 fit(X, YT, epochs = 50, verbose = 0)

Make predictions:

head(predict(DNN, X), n = 3)

[,1]  
[1,] 0.3291783  
[2,] 0.3246770  
[3,] 0.3321629

#### 6.2.0.2 Python

from tensorflow import keras  
from tensorflow.keras.layers import \*  
from sklearn import datasets  
from sklearn.preprocessing import scale  
iris = datasets.load\_iris()  
data = iris.data  
X = scale(data[:,1:4])  
Y = data[:,0]  
  
DNN = keras.Sequential()  
 # first hidden layer  
DNN.add(Dense(  
 input\_shape=[X.shape[1]],   
 units = 10,   
 activation = "relu"))   
 # second hidden layer with regularization  
DNN.add(Dense(  
 units = 20,   
 activation = "relu",  
 kernel\_regularizer = keras.regularizers.l1()))  
 # output layer, 3 output neurons for our three classes  
 # and softmax activation to get quasi probabilities   
 # that sum up to 1 for each observation  
DNN.add(Dense(  
 units = 1,   
 activation = None))  
  
# print architecture  
DNN.summary()  
  
# add loss function and optimizer

Model: "sequential\_3"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 dense\_9 (Dense) (None, 10) 40   
   
 dense\_10 (Dense) (None, 20) 220   
   
 dense\_11 (Dense) (None, 1) 21   
   
=================================================================  
Total params: 281  
Trainable params: 281  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DNN.compile(loss = keras.losses.mean\_squared\_error,  
 optimizer = keras.optimizers.Adamax(0.01))  
  
# train model  
DNN.fit(X, Y, epochs = 50, verbose = 0)

<keras.callbacks.History object at 0x7fb9ab7aaa90>

Make predictions:

DNN.predict(X)[0:10]

1/5 [=====>........................] - ETA: 0s  
5/5 [==============================] - 0s 742us/step  
array([[5.145269 ],  
 [4.755928 ],  
 [4.90053 ],  
 [4.8877606],  
 [5.2147093],  
 [5.314453 ],  
 [4.939465 ],  
 [5.108178 ],  
 [4.649039 ],  
 [5.0300593]], dtype=float32)

#### 6.2.0.3 Julia

import StatsBase  
using RDatasets  
using StatsBase  
using DataFrames  
import MLJBase.int  
using Flux, Statistics  
using Flux.Data: DataLoader  
using Flux: onehotbatch, onecold, @epochs  
using Flux.Losses: mse

Data preparation:

iris = dataset("datasets", "iris");  
X = transpose(Matrix(mapcols(StatsBase.zscore, iris[:, 2:4])));  
YT = iris[:, 1];  
YT = reshape(YT, 1, length(YT));  
  
data\_loader = DataLoader((X, YT), batchsize=10, shuffle=true);

Create model (similar to Keras):

model = Chain(  
 Dense(3, 20, relu),  
 Dense(20, 20, relu),  
 Dense(20, 1)  
)

Chain(  
 Dense(3 => 20, relu), # 80 parameters  
 Dense(20 => 20, relu), # 420 parameters  
 Dense(20 => 1), # 21 parameters  
) # Total: 6 arrays, 521 parameters, 2.410 KiB.

Train/optimize Model:

parameters = Flux.params(model);  
optimizer = ADAM(0.01);  
  
# Help functions  
loss(x, y) = mse(model(x), y);  
  
get\_loss() = @show sum(mse(model(X), YT));  
  
## Training  
for epoch in 1:20  
 Flux.train!(loss, parameters, data\_loader, optimizer, cb = Flux.throttle(get\_loss, 5))  
end

sum(mse(model(X), YT)) = 32.700644663996684  
sum(mse(model(X), YT)) = 6.238014457453128  
sum(mse(model(X), YT)) = 2.4814857188835284  
sum(mse(model(X), YT)) = 1.4879798222750726  
sum(mse(model(X), YT)) = 0.9339094797260101  
sum(mse(model(X), YT)) = 0.6294413628947929  
sum(mse(model(X), YT)) = 0.4001049104306762  
sum(mse(model(X), YT)) = 0.26098187775701015  
sum(mse(model(X), YT)) = 0.17825249282750188  
sum(mse(model(X), YT)) = 0.15775642389143868  
sum(mse(model(X), YT)) = 0.13210250468190513  
sum(mse(model(X), YT)) = 0.1066759334140655  
sum(mse(model(X), YT)) = 0.11969561605129814  
sum(mse(model(X), YT)) = 0.10018220269838507  
sum(mse(model(X), YT)) = 0.11994605863801423  
sum(mse(model(X), YT)) = 0.09796934727353351  
sum(mse(model(X), YT)) = 0.09360532331274031  
sum(mse(model(X), YT)) = 0.0939330481001742  
sum(mse(model(X), YT)) = 0.09852683445736722  
sum(mse(model(X), YT)) = 0.09423482923177973

Predictions:

transpose(model(X))[1:5]

5-element Vector{Float64}:  
 5.1439179712375935  
 4.648019985570975  
 4.760993338285283  
 4.723636501700876  
 5.243341550073481

# 7. Convolutional neural networks

Convolutional neural networks (CNN) are also deep neural networks but they are based on convolutional layers, which is a biologically inspired variation optimized to process image-based data (LeCun, Bengio, and Hinton (2015)). CNNs consist of two stages, in the first, the images are passed through convolutional layers and the models learns to detect edges and shapes in the images. In the second stage, the dimensions are dropped and fully-connected layers are used to classify the previously identified shapes.

In the following, we will use again the ‘keras’ package (Python: ‘keras’ (Chollet et al. (2015)); Julia: ‘Flux’ (Innes et al. (2018))) but we will not differentiate between classification and regression because the only difference would be to change the last layer and the loss function (see section ‘Deep neural networks’).

We will demonstrate the application of CNNs with the MNIST dataset which consists of handwritten digits. The objective of the CNNs is to classify the images. The MNIST dataset is one of the most famous benchmark dataset for image-based tasks (LeCun, Cortes, and Burges (2010)).

#### 7.0.0.1 R

library(keras)  
data = keras::dataset\_mnist()  
train = data$train  
X = train$x/255  
# we have to add a dimension that   
# informs the network about the channels  
# of the images  
X = array(X, dim = c(dim(X), 1))  
YT = k\_one\_hot(train$y, num\_classes = 10)  
  
  
  
CNN =   
 keras\_model\_sequential() %>%   
 # first hidden layer  
 layer\_conv\_2d(input\_shape = list(28, 28, 1),   
 filters = 16,  
 kernel\_size = c(2, 2),  
 activation = "relu") %>%  
 layer\_average\_pooling\_2d() %>%   
 layer\_conv\_2d(filters = 8,  
 kernel\_size = c(2, 2),  
 activation = "relu") %>%  
 # we use a normal DNN on top of the CNN:  
 # the layer flatten will remove the additional   
 # dimensions  
 layer\_flatten() %>%   
 layer\_dense(units = 20,   
 activation = "relu") %>%  
 # 10 output neurons for 10 classes  
 layer\_dense(units = 10,   
 activation = "softmax")  
  
  
# print architecture  
summary(CNN)

Model: "sequential\_4"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
================================================================================  
 conv2d\_1 (Conv2D) (None, 27, 27, 16) 80   
 average\_pooling2d (AveragePooling2 (None, 13, 13, 16) 0   
 D)   
 conv2d (Conv2D) (None, 12, 12, 8) 520   
 flatten (Flatten) (None, 1152) 0   
 dense\_13 (Dense) (None, 20) 23060   
 dense\_12 (Dense) (None, 10) 210   
================================================================================  
Total params: 23,870  
Trainable params: 23,870  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# add loss function and optimizer  
CNN %>%   
 compile(loss = loss\_categorical\_crossentropy,  
 optimizer = optimizer\_adamax(0.01))  
  
CNN %>%   
 fit(X, YT, epochs = 3, batch\_size = 125, verbose = 0)

Make predictions (class probabilites):

head(predict(CNN, X[1:100,,,,drop=FALSE]), n = 3)

[,1] [,2] [,3] [,4] [,5]  
[1,] 4.412973e-09 6.342341e-09 2.908698e-08 5.843263e-01 1.287379e-11  
[2,] 9.999059e-01 6.565648e-10 6.279070e-05 9.517643e-06 2.711661e-08  
[3,] 1.278754e-08 9.702469e-06 9.062645e-06 2.158921e-05 9.991369e-01  
 [,6] [,7] [,8] [,9] [,10]  
[1,] 4.156368e-01 6.798913e-10 8.393456e-08 7.530062e-06 2.922853e-05  
[2,] 2.418813e-08 9.055913e-06 4.663352e-07 7.174623e-07 1.144793e-05  
[3,] 1.294366e-06 1.019209e-07 1.004604e-04 2.668805e-04 4.539639e-04

#### 7.0.0.2 Python

from tensorflow import keras  
from tensorflow.keras.layers import \*  
data = keras.datasets.mnist.load\_data()  
train = data[0][0]  
labels = data[0][1]  
  
# We need to one hot encode our response classes  
YT = keras.utils.to\_categorical(labels, num\_classes = 10)  
  
CNN = keras.Sequential()  
 # first hidden layer  
CNN.add(Conv2D(input\_shape = [28, 28, 1],   
 filters = 16,  
 kernel\_size = (2, 2),  
 activation = "relu"))  
CNN.add(AveragePooling2D())  
CNN.add(Conv2D(filters = 8,  
 kernel\_size = (2, 2),  
 activation = "relu"))  
 # we use a normal DNN on top of the CNN:  
 # the layer flatten will remove the additional   
 # dimensions  
CNN.add(Flatten())  
 # output layer, 3 output neurons for our three classes  
 # and softmax activation to get quasi probabilities   
 # that sum up to 1 for each observation  
CNN.add(Dense(  
 units = 10,   
 activation = "softmax"))  
  
# print architecture  
CNN.summary()  
  
# add loss function and optimizer

Model: "sequential\_5"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv2d\_2 (Conv2D) (None, 27, 27, 16) 80   
   
 average\_pooling2d\_1 (Averag (None, 13, 13, 16) 0   
 ePooling2D)   
   
 conv2d\_3 (Conv2D) (None, 12, 12, 8) 520   
   
 flatten\_1 (Flatten) (None, 1152) 0   
   
 dense\_14 (Dense) (None, 10) 11530   
   
=================================================================  
Total params: 12,130  
Trainable params: 12,130  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CNN.compile(loss = keras.losses.categorical\_crossentropy,  
 optimizer = keras.optimizers.Adamax(0.01))  
  
# train model  
CNN.fit(train, YT, epochs = 5, verbose = 0)

<keras.callbacks.History object at 0x7fb9a9ff7c10>

Make predictions:

CNN.predict(train[0:10,:,:])

1/1 [==============================] - ETA: 0s  
1/1 [==============================] - 0s 48ms/step  
array([[1.2222529e-11, 2.8201530e-12, 9.0259692e-12, 1.2420694e-03,  
 4.2651195e-12, 9.9875784e-01, 1.0388055e-11, 4.6128470e-08,  
 1.9979696e-08, 1.6245238e-08],  
 [9.9999899e-01, 1.3500277e-13, 1.3882715e-07, 8.1885637e-11,  
 2.2356149e-12, 1.1065329e-11, 8.0222378e-07, 3.0945517e-12,  
 1.2624608e-09, 1.1810444e-09],  
 [1.8054818e-11, 2.6328000e-05, 5.0233207e-06, 3.5783730e-06,  
 9.9963522e-01, 1.1635276e-11, 4.2748433e-10, 2.6884692e-04,  
 1.3575789e-07, 6.0916496e-05],  
 [1.5490822e-11, 9.9999660e-01, 4.9433731e-07, 2.3444646e-11,  
 1.5400302e-06, 9.6778051e-11, 4.5313344e-09, 7.1953203e-08,  
 1.2214517e-06, 4.0205303e-10],  
 [1.2333402e-10, 1.0526487e-06, 2.7007390e-08, 2.0779573e-04,  
 2.1000829e-04, 3.7476293e-09, 2.6458549e-09, 4.3013491e-04,  
 5.7277004e-05, 9.9909365e-01],  
 [8.2029947e-12, 1.7567077e-09, 9.9999994e-01, 1.2566320e-08,  
 1.4631742e-10, 1.1474030e-14, 5.3585206e-15, 7.7715157e-10,  
 5.2869265e-08, 1.9127966e-09],  
 [8.7086727e-10, 9.9978518e-01, 3.3956206e-07, 1.4938166e-06,  
 1.1573491e-06, 4.3300457e-09, 1.2763194e-07, 1.9874746e-09,  
 2.1161574e-04, 1.9069188e-09],  
 [1.1698318e-14, 3.0522359e-11, 7.4313363e-08, 9.9999911e-01,  
 1.7408040e-15, 8.3791102e-10, 9.7806729e-18, 1.5416136e-08,  
 6.2611048e-07, 4.6363290e-08],  
 [1.8712758e-09, 9.9815720e-01, 2.8449236e-07, 1.0312405e-06,  
 1.7815427e-03, 4.8631119e-07, 7.7936047e-06, 5.4332600e-07,  
 5.0812545e-05, 3.7370430e-07],  
 [2.3495934e-07, 1.6328103e-08, 4.1669747e-08, 1.8563534e-11,  
 9.9999976e-01, 7.4493844e-10, 1.0436355e-09, 3.5820992e-08,  
 5.7783129e-08, 9.9025302e-13]], dtype=float32)

#### 7.0.0.3 Julia

import StatsBase  
using RDatasets  
using StatsBase  
using DataFrames  
import MLJBase.int  
using MLDatasets: MNIST  
using Flux, Statistics  
using Flux.Data: DataLoader  
using Flux: onehotbatch, onecold, @epochs  
using Flux.Losses: logitcrossentropy

Data preparation:

ENV["DATADEPS\_ALWAYS\_ACCEPT"] = "true"

"true"

xtrain, ytrain = MNIST(:train)[:];  
xtrain = reshape(xtrain/255., 28, 28, 1, 60000);  
ytrain = onehotbatch(ytrain, 0:9);  
  
data\_loader = DataLoader((xtrain, ytrain), batchsize=100, shuffle=true);

Create model (similar to Keras):

model = Chain(  
 Conv((2, 2), 1=>16, pad = (1, 1), relu),  
 MeanPool((2, 2)),  
 Conv((2, 2), 16=>8, pad = (1, 1), relu),  
 MeanPool((2, 2)),  
 Flux.flatten,  
 Dense(392, 20, relu),  
 Dense(20, 10)  
)

Chain(  
 Conv((2, 2), 1 => 16, relu, pad=1), # 80 parameters  
 MeanPool((2, 2)),  
 Conv((2, 2), 16 => 8, relu, pad=1), # 520 parameters  
 MeanPool((2, 2)),  
 Flux.flatten,  
 Dense(392 => 20, relu), # 7\_860 parameters  
 Dense(20 => 10), # 210 parameters  
) # Total: 8 arrays, 8\_670 parameters, 34.977 KiB.

Train/optimize Model:

parameters = Flux.params(model);  
optimizer = ADAM(0.01);  
  
# Help functions  
loss(x, y) = logitcrossentropy(model(x), y);  
  
get\_loss() = @show sum(logitcrossentropy(model(xtrain[:,:,:,1:100]), ytrain[:,1:100]));  
  
## Training  
for epoch in 1:1  
 Flux.train!(loss, parameters, data\_loader, optimizer, cb = Flux.throttle(get\_loss, 6000))  
end

sum(logitcrossentropy(model(xtrain[:, :, :, 1:100]), ytrain[:, 1:100])) = 2.3040986983709155

Predictions:

softmax(model(xtrain[:,:,:,1:5]))[:,1]

10-element Vector{Float64}:  
 0.10005078638292585  
 0.11245028122802601  
 0.10638743840147145  
 0.1026495535917804  
 0.09442762091573628  
 0.08654225254258163  
 0.10082212623172848  
 0.10158492903532627  
 0.09910200186575867  
 0.09598300980466473

# 8. Recurrent neural networks

Recurrent neural networks are also deep neural networks but use layers specialized to handle time-series. In the following, we will use again the ‘keras’ package (Python: ‘keras’ (Chollet et al. (2015)); Julia: ‘Flux’ (Innes et al. (2018))) but we will not differentiate between classification and regression because the only difference would be to change the last layer and the loss function (see section ‘Deep neural networks’).

About the data, we simulated in the following one time series from a simple ARIMA process, using the ‘arima.sim’ function. Our goal is to train a net which is able to predict the next 10 time points based on the previous 10 time points.

#### 8.0.0.1 R

## RNNs  
library(keras)  
data = as.matrix(arima.sim(n = 1000, list(ar = c(0.3, -0.7)) ))  
# We use here a simplified way to create X and Y   
# since the focus is on creating the RNNs  
data = matrix(data, ncol = 10L, byrow = TRUE)  
X = array(data[seq(1, 100, by = 2), ], dim = c(50, 10, 1))  
Y = data[seq(2, 100, by = 2), ]  
  
RNN =   
 keras\_model\_sequential() %>%   
 # first hidden layer  
 layer\_gru(input\_shape = list(10L, 1L),  
 units = 50,   
 activation = "relu") %>%  
 # we want to predict the next 10 time steps  
 layer\_dense(units = 10)  
  
  
# add loss function and optimizer  
RNN %>%   
 compile(loss = loss\_mean\_squared\_error,  
 optimizer = optimizer\_adamax(0.01))  
  
RNN %>%   
 fit(X, Y, epochs = 5, verbose = 0)

Make predictions:

head(predict(RNN, X), n = 3)

[,1] [,2] [,3] [,4] [,5] [,6]  
[1,] -0.33588454 -0.20743205 -0.3007165 0.16366243 -0.08658758 -0.093666822  
[2,] -0.09402724 0.03769331 -0.1232934 -0.09395418 -0.05223484 0.103397779  
[3,] -0.18888274 -0.10091367 -0.2105696 0.06386845 -0.04689572 -0.009097762  
 [,7] [,8] [,9] [,10]  
[1,] 0.005928583 -0.01642237 0.25050706 0.16062395  
[2,] 0.142853200 -0.07787192 0.03283456 0.08624709  
[3,] 0.056737989 -0.07029341 0.14080283 0.12900156

#### 8.0.0.2 Python

from tensorflow import keras  
from tensorflow.keras.layers import \*  
X = r.X # get data from R  
Y = r.Y   
  
RNN = keras.Sequential()  
 # first hidden layer  
RNN.add(GRU(input\_shape = [10, 1],units = 50, activation = "relu"))  
RNN.add(Dense(units = 10))  
  
RNN.summary()  
  
# add loss function and optimizer

Model: "sequential\_7"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 gru\_1 (GRU) (None, 50) 7950   
   
 dense\_16 (Dense) (None, 10) 510   
   
=================================================================  
Total params: 8,460  
Trainable params: 8,460  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RNN.compile(loss = keras.losses.mean\_squared\_error,  
 optimizer = keras.optimizers.Adamax(0.01))  
  
# train model  
RNN.fit(X, Y, epochs = 5, verbose = 0)

<keras.callbacks.History object at 0x7fb9ab7aa460>

Make predictions:

RNN.predict(X)[0:10,:]

1/2 [==============>...............] - ETA: 0s  
2/2 [==============================] - 0s 3ms/step  
array([[-0.38465482, -0.41859597, -0.02743661, 0.13201939, 0.24694398,  
 -0.01636903, 0.03915604, -0.21624549, 0.15798992, 0.09606393],  
 [-0.09663856, 0.01404856, -0.07643997, -0.05320521, -0.02295429,  
 0.07126729, 0.148587 , -0.0139749 , -0.01798547, 0.02168824],  
 [-0.23548815, -0.2272258 , -0.05757517, 0.057076 , 0.1334207 ,  
 0.02728583, 0.06991484, -0.11131115, 0.04773262, 0.07174627],  
 [-0.25417262, -0.2220317 , -0.04801505, 0.06234731, 0.11366995,  
 0.03485882, 0.11755048, -0.09529305, 0.08806616, 0.06258951],  
 [-0.19432755, -0.06652889, -0.05130089, 0.02758526, 0.04181171,  
 0.04648231, 0.17930824, -0.04062552, 0.06287453, 0.00769253],  
 [-0.05717446, 0.597986 , -0.05397581, -0.6034371 , -0.76668596,  
 0.29614076, 0.5022045 , -0.04225525, 0.02560524, 0.02416247],  
 [-0.10118802, 0.31108534, -0.02332222, -0.25043517, -0.32576844,  
 0.12247837, 0.40015176, -0.02832408, 0.05361507, -0.03354261],  
 [-0.23485383, -0.15943694, -0.00620798, 0.08316425, 0.09176438,  
 0.00700938, 0.13452147, -0.06171318, 0.09433301, 0.02001512],  
 [-0.07895298, 0.15879202, -0.0934405 , -0.1683502 , -0.17538187,  
 0.13678485, 0.24379435, -0.00400311, -0.02202763, 0.01408674],  
 [-0.08533065, 0.3923275 , -0.03386774, -0.40053195, -0.49471754,  
 0.1850509 , 0.38264304, -0.05235155, 0.03393514, 0.00791531]],  
 dtype=float32)  
  
WARNING:tensorflow:5 out of the last 13 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fba102fd310> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce\_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling\_retracing and https://www.tensorflow.org/api\_docs/python/tf/function for more details.

#### 8.0.0.3 Julia

import StatsBase;  
using RDatasets;  
using StatsBase;  
using DataFrames;  
import MLJBase.int;  
using Flux, Statistics;  
using Flux.Losses: mse;  
using ARFIMA;

Data preparation:

X = transpose(reshape(convert(Vector{Float32}, arfima(1000,0.5, 0.3, SVector(-0.7))), 100, 10));  
xtrain = X[:, collect(1:2:100)];  
ytrain = X[:, collect(2:2:100)];

Create model (similar to Keras):

model = Chain(  
 GRU(10=>50),  
 Dense(50, 10)  
)

Chain(  
 Recur(  
 GRUCell(10 => 50), # 9\_200 parameters  
 ),  
 Dense(50 => 10), # 510 parameters  
) # Total: 6 trainable arrays, 9\_710 parameters,  
 # plus 1 non-trainable, 50 parameters, summarysize 38.297 KiB.

Train/optimize Model:

parameters = Flux.params(model);  
optimizer = ADAM(0.01);  
  
for epoch in 1:10  
 Flux.reset!(model);   
 grads = gradient(parameters) do   
 Pred = [model( xtrain[:,i]) for i in 1:50];  
 loss = mean([mse(Pred[i], ytrain[:,i]) for i in 1:50]);  
 println(loss);  
 loss  
 end  
 Flux.update!(optimizer, parameters, grads);  
end

0.41154057  
0.33427083  
0.29357198  
0.2782965  
0.27118874  
0.26072773  
0.24815777  
0.23578274  
0.2244305  
0.21443754

Predictions:

Pred = [model( xtrain[:,i]) for i in 1:50];  
Pred[1]

10-element Vector{Float32}:  
 0.010387432  
 -0.05032792  
 -0.020344056  
 -0.056029417  
 -0.37598398  
 -0.14112368  
 0.07555613  
 0.10049449  
 0.05639108  
 0.050382867

# 9. Graph (convolutional) neural networks

Graph neural networks (GNN) is a young representative of the deep neural network family but is receiving more and more attention in the last years because of their ability to process non-Euclidean data such as graphs.

Currently there is no R package for GNNs available. However, we can use the ‘reticulate’ package (Ushey, Allaire, and Tang (2022)) to use the python packages ‘torch’ and ‘torch\_geometric’ (Paszke et al. (2019), 2019; Fey and Lenssen (2019)).

The following example was mostly adapted from the ‘Node Classification with Graph Neural Networks’ example from the torch\_geometric documentation (<https://pytorch-geometric.readthedocs.io/en/latest/notes/colabs.html>).

The dataset is also provided by the ‘torch\_geometric’ package and consists of molecules presented as graphs and the task is to predict whether HIV virus replication is inhibited by the molecule or not (classification, binary classification).

We have not implemented this example in Julia because there is not yet a well-established library for GNNs.

#### 9.0.0.1 R

library(reticulate)  
# Load python packages torch and torch\_geometric via the reticulate R package  
torch = import("torch")   
torch\_geometric = import("torch\_geometric")  
  
# helper functions from the torch\_geometric modules  
GCNConv = torch\_geometric$nn$GCNConv  
global\_mean\_pool = torch\_geometric$nn$global\_mean\_pool  
  
  
# Download the MUTAG TUDataset  
dataset = torch\_geometric$datasets$TUDataset(root='data/TUDataset',   
 name='MUTAG')  
dataloader = torch\_geometric$loader$DataLoader(dataset,   
 batch\_size=64L,  
 shuffle=TRUE)  
  
# Create the model with a python class  
# There are two classes in the response variable  
GCN = PyClass(  
 "GCN",   
 inherit = torch$nn$Module,   
 defs = list(  
 `\_\_init\_\_` = function(self, hidden\_channels) {  
 super()$`\_\_init\_\_`()  
 torch$manual\_seed(42L)  
 self$conv = GCNConv(dataset$num\_node\_features, hidden\_channels)  
 self$linear = torch$nn$Linear(hidden\_channels, dataset$num\_classes)  
 NULL  
 },  
 forward = function(self, x, edge\_index, batch) {  
 x = self$conv(x, edge\_index)  
 x = x$relu()  
 x = global\_mean\_pool(x, batch)  
   
 x = torch$nn$functional$dropout(x, p = 0.5, training=self$training)  
 x = self$linear(x)  
 return(x)  
 }  
 ))

Training loop:

# create model object  
model = GCN(hidden\_channels = 64L)  
  
# get optimizer and loss function  
optimizer = torch$optim$Adamax(model$parameters(), lr = 0.01)  
loss\_func = torch$nn$CrossEntropyLoss()  
  
# set model into training mode (because of the dropout layer)  
model$train()

GCN(  
 (conv): GCNConv(7, 64)  
 (linear): Linear(in\_features=64, out\_features=2, bias=True)  
)

# train model  
for(e in 1:50) {  
 iterator = reticulate::as\_iterator(dataloader)  
 coro::loop(for (b in iterator) {   
 pred = model(b$x, b$edge\_index, b$batch)  
 loss = loss\_func(pred, b$y)  
 loss$backward()  
 optimizer$step()  
 optimizer$zero\_grad()  
 })  
 if(e %% 10 ==0) cat(paste0("Epoch: ",e," Loss: ", round(loss$item()[1], 4), "\n"))  
}

Epoch: 10 Loss: 0.6151  
Epoch: 20 Loss: 0.6163  
Epoch: 30 Loss: 0.5745  
Epoch: 40 Loss: 0.5362  
Epoch: 50 Loss: 0.5829

Make predictions:

preds = list()  
test = torch\_geometric$loader$DataLoader(dataset, batch\_size=64L,shuffle=FALSE)  
iterator = reticulate::as\_iterator(test)  
model$eval()

GCN(  
 (conv): GCNConv(7, 64)  
 (linear): Linear(in\_features=64, out\_features=2, bias=True)  
)

counter = 1  
coro::loop(for (b in iterator) {  
 preds[[counter]] = model(b$x, b$edge\_index, b$batch)  
 counter <<- counter + 1  
 })  
head(torch$concat(preds)$sigmoid()$data$cpu()$numpy(), n = 3)

[,1] [,2]  
[1,] 0.3076028 0.6427078  
[2,] 0.4121239 0.5515330  
[3,] 0.4119514 0.5516798

#### 9.0.0.2 Python

# Load python packages torch and torch\_geometric via the reticulate R package  
import torch  
import torch\_geometric  
  
# helper functions from the torch\_geometric modules  
GCNConv = torch\_geometric.nn.GCNConv  
global\_mean\_pool = torch\_geometric.nn.global\_mean\_pool  
  
  
# Download the MUTAG TUDataset  
dataset = torch\_geometric.datasets.TUDataset(root='data/TUDataset',   
 name='MUTAG')  
dataloader = torch\_geometric.loader.DataLoader(dataset,   
 batch\_size=64,  
 shuffle=True)  
  
# Create the model with a python class  
# There are two classes in the response variable  
class GCN(torch.nn.Module):  
 def \_\_init\_\_(self, hidden\_channels):  
 super().\_\_init\_\_()  
 torch.manual\_seed(42)  
 self.conv = GCNConv(dataset.num\_node\_features, hidden\_channels)  
 self.linear = torch.nn.Linear(hidden\_channels, dataset.num\_classes)  
   
 def forward(self, x, edge\_index, batch):  
 x = self.conv(x, edge\_index)  
 x = x.relu()  
 x = global\_mean\_pool(x, batch)  
 x = torch.nn.functional.dropout(x, p = 0.5, training=self.training)  
 x = self.linear(x)  
 return x

Training loop:

# create model object  
model = GCN(hidden\_channels = 64)  
  
# get optimizer and loss function  
optimizer = torch.optim.Adamax(model.parameters(), lr = 0.01)  
loss\_func = torch.nn.CrossEntropyLoss()  
  
# set model into training mode (because of the dropout layer)  
model.train()  
  
# train model

GCN(  
 (conv): GCNConv(7, 64)  
 (linear): Linear(in\_features=64, out\_features=2, bias=True)  
)

for e in range(50):  
 for b in dataloader:  
   
 pred = model(b.x, b.edge\_index, b.batch)  
 loss = loss\_func(pred, b.y)  
 loss.backward()  
 optimizer.step()  
 optimizer.zero\_grad()  
   
 if e % 10 ==0:  
 print("Epoch: ", e ," Loss: ", loss.item(), "\n")

Epoch: 0 Loss: 0.6617004871368408   
  
Epoch: 10 Loss: 0.614981472492218   
  
Epoch: 20 Loss: 0.6161867380142212   
  
Epoch: 30 Loss: 0.5802667737007141   
  
Epoch: 40 Loss: 0.5124867558479309

Make predictions:

preds = []  
test = torch\_geometric.loader.DataLoader(dataset, batch\_size=64,shuffle=False)  
model.eval()

GCN(  
 (conv): GCNConv(7, 64)  
 (linear): Linear(in\_features=64, out\_features=2, bias=True)  
)

counter = 1  
for b in test:  
 preds.append( model(b.x, b.edge\_index, b.batch) )  
   
   
torch.concat(preds).sigmoid().data.cpu().numpy()[0:10]

array([[0.30760282, 0.64270777],  
 [0.41212386, 0.551533 ],  
 [0.4119514 , 0.5516798 ],  
 [0.29887193, 0.650517 ],  
 [0.48894534, 0.48584774],  
 [0.4310807 , 0.5360305 ],  
 [0.31375578, 0.63721913],  
 [0.34597102, 0.6093393 ],  
 [0.50279325, 0.4740774 ],  
 [0.30924183, 0.6412629 ]], dtype=float32)

## References

Abadi, Martı́n, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. “Tensorflow: A System for Large-Scale Machine Learning.” In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–83.

Allaire, JJ, and François Chollet. 2022. *Keras: R Interface to ’Keras’*. <https://CRAN.R-project.org/package=keras>.

Blaom, Anthony, Franz Kiraly, Thibaut Lienart, and Sebastian Vollmer. 2019. *Alan-Turing-Institute/MLJ.jl: V0.5.3* (version v0.5.3). Zenodo. <https://doi.org/10.5281/zenodo.3541506>.

Chen, Tianqi, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. 2022. *Xgboost: Extreme Gradient Boosting*. <https://CRAN.R-project.org/package=xgboost>.

Chollet, Francois et al. 2015. “Keras.” GitHub. 2015. <https://github.com/fchollet/keras>.

Feinerer, Ingo, Kurt Hornik, and David Meyer. 2008. “Text Mining Infrastructure in r.” *Journal of Statistical Software* 25 (5): 1–54. <https://www.jstatsoft.org/v25/i05/>.

Fellows, Ian. 2018. *Wordcloud: Word Clouds*. <https://CRAN.R-project.org/package=wordcloud>.

Fey, Matthias, and Jan E. Lenssen. 2019. “Fast Graph Representation Learning with PyTorch Geometric.” In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Innes, Michael, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. 2018. “Fashionable Modelling with Flux.” *CoRR* abs/1811.01457. <https://arxiv.org/abs/1811.01457>.

Jahn, Najko. 2021. *Europepmc: R Interface to the Europe PubMed Central RESTful Web Service*. <https://CRAN.R-project.org/package=europepmc>.

Lang, Dawei, and Guan-tin Chien. 2018. *Wordcloud2: Create Word Cloud by ’Htmlwidget’*. <https://CRAN.R-project.org/package=wordcloud2>.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. “Deep Learning.” *Nature* 521 (7553): 436–44.

LeCun, Yann, Corinna Cortes, and CJ Burges. 2010. “MNIST Handwritten Digit Database.” *ATT Labs [Online]. Available: Http://Yann.lecun.com/Exdb/Mnist* 2.

Meyer, David, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch. 2022. *E1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. <https://CRAN.R-project.org/package=e1071>.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” In *Advances in Neural Information Processing Systems 32*, 8024–35. Curran Associates, Inc. <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. “Scikit-Learn: Machine Learning in Python.” *Journal of Machine Learning Research* 12: 2825–30.

Schliep, Klaus, and Klaus Hechenbichler. 2016. *Kknn: Weighted k-Nearest Neighbors*. <https://CRAN.R-project.org/package=kknn>.

Simon, Noah, Jerome Friedman, Trevor Hastie, and Rob Tibshirani. 2011. “Regularization Paths for Cox’s Proportional Hazards Model via Coordinate Descent.” *Journal of Statistical Software* 39 (5): 1–13. <https://doi.org/10.18637/jss.v039.i05>.

Ushey, Kevin, JJ Allaire, and Yuan Tang. 2022. *Reticulate: Interface to ’Python’*. <https://CRAN.R-project.org/package=reticulate>.

Wright, Marvin N., and Andreas Ziegler. 2017. “ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R.” *Journal of Statistical Software* 77 (1): 1–17. <https://doi.org/10.18637/jss.v077.i01>.