test_LinRegJulia

January 16, 2024

1 Testing `LinReg.jl` functionality

This notebook contains a small showcase on how to use the functions inside the `LinReg.jl` file.

1.1 Testing

[ ]:
```julia
using CSV
using DataFrames
using Random
using StableRNGs

include("solution/LinReg.jl");
```

Read the data:

[ ]:
```julia
data = CSV.read("dataset.txt", DataFrame, header=0)
first(data, 5)
```

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	
1	8.0	1.0	0.19	0.33	0.02	0.9	0.12	0.17	0.34	...
2	53.0	1.0	0.0	0.16	0.12	0.74	0.45	0.07	0.26	...
3	24.0	1.0	0.0	0.42	0.49	0.56	0.17	0.04	0.39	...
4	34.0	1.0	0.04	0.77	1.0	0.08	0.12	0.1	0.51	...
5	42.0	1.0	0.01	0.55	0.02	0.95	0.09	0.05	0.38	...

Separate into a matrix of observations X and a target variable y:

[ ]:
```julia
y, X = unpack(data, ==(:Column102));
```

Implement the regressor. We recommend using the `LinearRegressor` from the *MLJLinearModels* package. Here is the documentation:

[ ]:
```julia
doc("LinearRegressor", pkg="MLJLinearModels")
```

`LinearRegressor`

A model type for constructing a linear regressor, based on MLJLinearModels.jl, and implementing the MLJ model interface.

From MLJ, the type can be imported using

```
LinearRegressor = @load LinearRegressor pkg=MLJLinearModels
```

Do `model = LinearRegressor()` to construct an instance with default hyper-parameters.

This model provides standard linear regression with objective function

$$|X\theta - y|_2^2/2$$

Different solver options exist, as indicated under "Hyperparameters" below.

2 Training data

In MLJ or MLJBase, bind an instance `model` to data with

```
mach = machine(model, X, y)
```

where:

- `X` is any table of input features (eg, a `DataFrame`) whose columns have `Continuous` scitype; check column scitypes with `schema(X)`
- `y` is the target, which can be any `AbstractVector` whose element scitype is `Continuous`; check the scitype with `scitype(y)`

Train the machine using `fit!(mach, rows=...)`.

3 Hyperparameters

- `fit_intercept::Bool`: whether to fit the intercept or not. Default: true
- `solver::Union{Nothing, MLJLinearModels.Solver}`: "any instance of `MLJLinearModels.Analytical`. Use `Analytical()` for Cholesky and `CG()=Analytical(iterative=true)` for conjugate-gradient.

  If `solver = nothing` (default) then `Analytical()` is used. Default: nothing

3.1 Example

```
using MLJ
X, y = make_regression()
mach = fit!(machine(LinearRegressor(), X, y))
predict(mach, X)
fitted_params(mach)
```

Load and instantiate the model (and suppress warnings and such):

[ ]:
```
LinearRegressor = @load LinearRegressor pkg=MLJLinearModels verbosity=0
model = LinearRegressor()
```

```
LinearRegressor(
  fit_intercept = true,
  solver = nothing)
```

Set a stable random number generator for reproducibility:

[ ]:
```
myRNG = StableRNG(123)
```

```
StableRNGs.LehmerRNG(state=0x000000000000000000000000000000f7)
```

Given a dummy binary array:

[ ]:
```
rand_ind = bitrand(101)
```

```
101-element BitVector:
 0
 1
 0
 0
 1
 0
 1
 1
 0
 0

 1
 0
 1
 0
 1
 0
 0
 1
 1
```

We can use the `get_columns` function provided in `LinReg.jl` to get the columns marked as 1 from the `data` and save it in a matrix X_{sub}

[ ]:
```
X_sub = get_columns(X, rand_ind)
```

	Column2	Column5	Column7	Column8	Column11	Column15	Column16	Column17	Column19
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.02	0.12	0.17	0.29	0.37	0.72	0.34	0.29
2	1.0	0.12	0.45	0.07	0.35	0.31	0.72	0.11	0.25
3	1.0	0.49	0.17	0.04	0.28	0.3	0.58	0.19	0.38
4	1.0	1.0	0.12	0.1	0.34	0.58	0.89	0.21	0.36
5	1.0	0.02	0.09	0.05	0.23	0.5	0.72	0.16	0.44
6	1.0	0.06	1.0	0.25	0.27	0.52	0.68	0.2	0.28
7	1.0	0.0	0.06	0.02	0.23	0.42	0.5	0.23	0.61
8	1.0	0.03	0.2	1.0	0.36	0.16	0.44	1.0	0.53
9	1.0	0.2	0.02	0.0	0.28	0.17	0.47	0.36	0.55
10	1.0	0.06	0.3	0.03	0.8	0.54	0.59	0.22	0.42
11	1.0	0.15	1.0	0.41	0.35	0.49	0.71	0.16	0.36
12	1.0	0.08	0.07	0.1	0.22	0.72	0.53	0.23	0.63
13	1.0	0.01	0.13	0.02	0.2	0.8	0.55	0.18	0.51
14	1.0	0.0	0.04	0.01	0.32	0.46	0.77	0.41	0.28
15	1.0	0.01	0.14	0.26	0.3	0.71	0.67	0.42	0.25
16	1.0	0.06	0.03	0.03	0.28	0.18	0.42	0.81	0.62
17	1.0	0.4	0.14	0.06	0.65	0.22	0.52	0.1	0.48
18	1.0	0.01	0.2	0.03	0.27	0.79	0.77	0.13	0.44
19	1.0	0.01	0.07	0.02	0.63	0.33	0.56	0.28	0.43
20	1.0	0.05	0.01	0.01	0.24	0.23	0.34	0.33	0.7
21	1.0	0.05	0.48	0.3	0.28	0.33	0.55	0.37	0.39
22	1.0	0.47	0.12	0.05	0.34	0.28	0.62	0.16	0.4
23	1.0	0.02	0.07	0.11	0.43	0.13	0.4	0.26	0.52
24	1.0	0.04	0.09	0.06	0.31	0.22	0.52	0.44	0.56
…	…	…	…	…	…	…	…	…	…

We finally use the `get_fitness` method to train, test and calculate the root mean square error of our prediction using our `LinearRegressor` model, X, and y (that we have separated earlier).

- The model we are using is our `LinearRegressor`
- Observations are taken from X: all rows, and all columns except the last one
- Targets are taken from the last column of `data`, which is y
- Optionally pass a random number generator to use as `rng`

[ ]:
```
get_fitness(model, X_sub, y; rng=myRNG)
```

```
0.1437366927254183
```

3.2 Documentation

All methods are well documented via docstrings, which can be understood both by humans and Julia. For example, we can use the `@doc` macro:

[ ]:
```
@doc get_fitness
```

`get_fitness(model, Xsub, y; rng=myRNG)`

Given a `model`, a subset of the data `Xsub` and a vector of targets `y` , return the square root of the

MSE of the model.

3.3 Parameters

- `model`: An *MLJ* model.
- `Xsub`: an nxm matrix of data that should be used for training the model.
- `y`: a vector of length n containing the regression (target) values of observations
- `rng`: a StableRNGs random number generator for reproducible results