Empirical Risk Minimization in Julia (`EmpiricalRiskMinimization.jl`)

Reese Pathak, Sanjay Lall

EE104
Stanford University

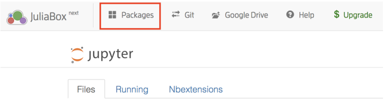
Basics

`EmpiricalRiskMinimization.jl`

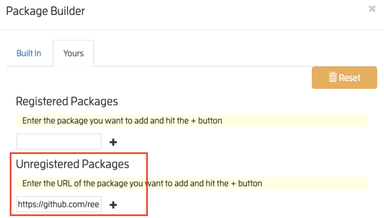
- abbreviated `ERM.jl` (or just `ERM`)
- allows you to quickly develop standard machine learning models in Julia
- your job: specify a model via a supported loss and regularizer
- `ERM`'s job: use a solver to determine optimal model parameters
- replaces explicit calls to specialized codes (*e.g.*, gradient descent, prox-gradient, *etc.*)
    - but, is often slower than specialized code

Loading `ERM.jl` in JuliaBox

navigate to JuliaBox, then click on the *Packages* button



click the *Yours* button under *Package Builder*



under *Unregistered Packages*, type in the link below and click $+$

`https://github.com/reesepathak/EmpiricalRiskMinimization.jl.git`.

Usage notes

users with a local Julia installation can still access `ERM.jl`

```
Pkg.add("https://github.com/reesepathak/EmpiricalRiskMinimization.jl.git")
using EmpiricalRiskMinimization
```

all users should *frequently* update the package

```
Pkg.update()
using EmpiricalRiskMinimization
```

(`ERM` is currently in version 0.0.1, so fixes to bugs are constantly being pushed)

Ridge regression

Ridge regression in `ERM.jl`

code below generates a random data set

```
srand(100)
n, k = 5000, 10; d = k + 1;
U = randn(n, k); theta_true = randn(d);
v = [ones(n) U] * theta_true + randn(n)
```

the following Julia code uses `ERM` to train a ridge regression model

```
using EmpiricalRiskMinimization
include("regression_data.jl") # loads data

M = Model(U, v, embedall=true, verbose=false)
train(M, lambda=1e-3)   # automatically splits data
status(M)  # will display a summary of training step
```

Ridge regression in `ERM.jl`

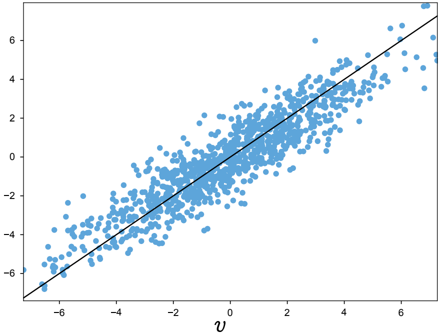
a single run of the previous code

```
----------------------------------------
results for single train/test
  training  loss: 0.149055202518492
  test loss: 0.16689449857352331
  training samples: 4000
  test samples: 1000
  columns in X: 11
----------------------------------------
```

notice that `ERM.jl` defaulted to a 80-20 train-test split, automatically appended the constant feature, and standardized the data.

Ridge regression in `ERM.jl`

v versus $\hat{v}$ (RMS error ≈ 1.051)



- retrieve train, test losses with `trainloss(M)`, `testloss(M)`
- get optimal theta with `thetaopt(M)`
- predictions on test set with `predict_v_from_test(M)`

Structure of `EmpiricalRiskMinimization.jl`

The `Model` type

computations with `ERM.jl` require a `Model`

```
M = Model(U, V, loss=SquareLoss(), reg=L1Reg(),
                embedall=true, verbose=false)
```

specifying a `Model`

- pass in raw numerical data
- pick a loss function with the `loss` keyword (default: MSE)
- pick a regularizer with the `reg` keyword (default: Tykhonov)
- turn our usual embedding on/off with the `embedall` keyword (default: off)
  - when `embedall=true`, $X = [\mathbf{1}\ \tilde{U}]$, where $\tilde{U}$ has standardized columns.

(more on all of this shortly)

Supported losses and regularizers

the current list of supported losses and regularizers is available at the docs page

`https://reesepathak.github.io/EmpiricalRiskMinimization.jl/latest/`

the *Usage* page has a section on losses and a section on regularizers

Training, validating, and predicting

training a model is achieved with

```
train(M, lambda=1e-6, trainfrac=0.6)
```

- keyword arguments `lambda`, `trainfrac` set the regularization weights and fraction of data used for training
- by default, `lambda=1e-10` and `trainfrac=0.8`

prediction and validation

- to get train and test losses, run `trainloss(M)` and `testloss(M)`
- `predict_v_from_test(M)` outputs predictions on the test set
- `predict_v_from_u(M, U)` outputs predictions on `U`

(more functions are available, see docs)

Cross validation and regularization paths

sweeping over regularization weights is automatic in `ERM`

```
trainpath(M, lambda=logspace(-3, 3, 50), trainfrac=0.75)
```

- `lambda` is the list of weights, defaults to `logspace(-5, 5, 100)`
- `trainfrac` is the fraction of data to use for training, defaults to `0.8`
- call `trainlosspath` or `testlosspath` to get the train or test losses
- call `thetaopt` or `lambdaopt` to get the optimal model parameters

cross validation is also possible

```
trainfolds(M, lambda=1e-6, nfolds=15)
```

(defaults are `1e-10` and `5`)

Robust regression

Data

code below generates some random data

```
srand(50)
n, k = 1000, 10; d = k + 1;
U = randn(n, k); theta_true = randn(d);
v = [ones(n) U] * theta_true + randn(n)
v += 15 * (1. * (rand(n) .> 0.85)) .* randn(n) # outliers
```

(we've artificially added some outliers to keep things interesting)

Huber regression, with regularization path

the following Julia code uses ERM to train a Huber regression model

```
using EmpiricalRiskMinimization
include("regpath_data.jl") # data (from previous slide)

# compile model
M = Model(U, v, loss=HuberLoss(), embedall=true, verbose=false)

# train over multiple regularization weights
trainpath(M, lambdas=logspace(-3, 3, 100), trainfrac=0.8)

# capture the list of optimal thetas for each lambda
lambdas, thetas = lambdapath(M), thetapath(M)

# capture the list of train/test losses for each lambda
train_losses, test_losses = trainlosspath(M), testlosspath(M)

# store lambda (and corresponding theta) giving lowest test error
theta_opt, lambda_opt = thetaopt(M), lambdaopt(M)
```

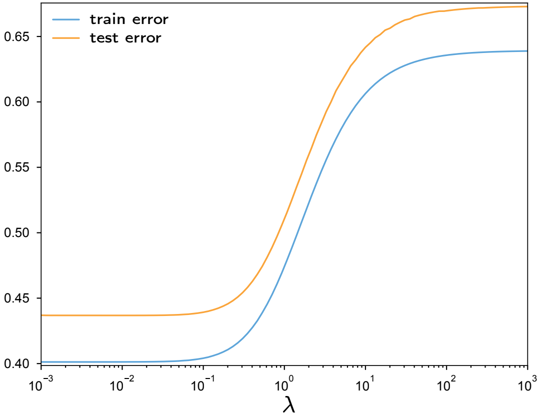
Results

running `status(M)` gives us the following information

```
----------------------------------------
Optimal results along regpath
  optimal lambda: 0.03274549162877728
  optimal test loss: 0.13207774630515404
  training samples: 800
  test samples: 200
  columns in X: 401
----------------------------------------
```

`ERM` took care of the usual embedding, splitting train and test sets, and sweeping over the regularization weights we specified

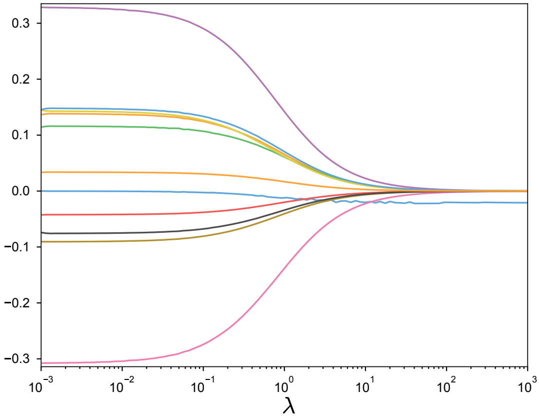
Results



- we retrieved the train and test losses with `trainlosspath(M)` and `testlosspath(M)`
  - this was (much) easier than doing it by hand...

Regularization path

we can plot the components θ_i against λ



- to create the figure above, all we had to do was call `plot` on `thetapath(M)'`.

Summary

Summary

- you'll use `ERM` on future homework assignments
- it's easy to get started; docs are available on the course webpage
- you've now seen a couple of examples covering (some) of the course material
- this package should make it easier to train, test, and validate models
- `ERM` also supports regularization paths, cross validation
- let us know if you enounter bugs! we hope you won't find any...