EmpiricalRisks Documentation

Release 0.2.3

Dahua Lin and contributors

September 09, 2015

Contents

This package provides the basic components for *(regularized) empirical risk minization*, which is generally formulated as follows

$$\sum_{i=1}^{n} loss(f(x_i; \theta), y_i) + r(\theta)$$

As we can see, this formulation involves several components:

- **Prediction model:** $f(x; \theta)$, which takes an input x and a parameter θ and produces an output (say u).
- **Loss function:** $loss(u, y)$, which compares the predicted output u and a desired response y, and produces a real value that measuring the *loss*. Generally, better prediction yields smaller loss.
- **Risk model:** $loss(f(x; \theta), y)$, the *prediction model* and the *loss* together are referred to as the *risk model*. When the data *x* and *y* are given, the risk model can be considered as a function of *theta*.
- **Regularizer:** $r(\theta)$ is often introduced to regularize the parameter, which, when used properly, can improve the numerical stability of the problem and the generalization performance of the estimated model.

This package provides these components, as well as the gradient computation routines and proximal operators, to support the implementation of various empirical risk minimization algorithms.

All functions in this packages are well optimized and systematically tested.

CHAPTER 1

Contents:

1.1 Prediction Models

A *prediction model* $f(x; \theta)$ is a function with two arguments: the input feature x and the predictor parameter θ. All prediction models are instances of an the abstract type `PredictionModel`, defined as follows:

```
abstract PredictionModel{NDIn, NDOut}

# NDIn:  The number of dimensions of each input (0: scalar, 1: vector, 2: matrix, ...)
# NDOut: The number of dimensions of each output (0: scalar, 1: vector, 2: matrix, ...)
```

1.1.1 Common Methods

Each prediction model implements the following methods:

inputlen (*pm*)
: Return the length of each input.

inputsize (*pm*)
: Return the size of each input.

outputlen (*pm*)
: Return the length of each output.

outputsize (*pm*)
: Return the size of each output.

paramlen (*pm*)
: Return the length of the parameter.

paramsize (*pm*)
: Return the size of the parameter.

ninputs (*pm*, *x*)
: Verify the validity of `x` as a single input or as a batch of inputs. If `x` is valid, it returns the number of inputs in array `x`, otherwise, it raises an error.

predict (*pm*, *theta*, *x*)
: Predict the output given the parameter `theta` and the input `x`.

  Here, `x` can be either a sample or an array comprised of multiple samples.

1.1.2 Predefined Models

The package provides the following prediction models:

(Univariate) Linear Prediction

$$f(x;\theta) = \theta^T x$$

- **parameter:** θ, a vector of length `d`.
- **input:**: x, a vector of length `d`.
- **output:**: a scalar.

```
immutable LinearPred <: PredictionModel{1,0}
    dim::Int

    LinearPred(d::Int) = new(d)
end
```

(Univariate) Affine Prediction

$$f(x;\theta) = w^T x + a \cdot b$$

Here, `b` is a model constant to serve as the base of the bias term.

- **parameter:** θ, a vector of length `d + 1`, in the form `[w; a]`.
- **input:** x, a vector of length `d`.
- **output:**: a scalar.

```
immutable AffinePred <: PredictionModel{1,0}
    dim::Int
    bias::Float64

    AffinePred(d::Int) = new(d, 1.0)
    AffinePred(d::Int, b::Real) = new(d, convert(Float64, b))
end
```

(Multivariate) Linear Prediction

$$f(x;\theta) = W \cdot x$$

- **parameter:** $\theta = W$, a matrix of size `(k, d)`.
- **input:** x, a vector of length `d`.
- **output:** a vector of length `k`.

```
immutable MvLinearPred <: PredictionModel{1,1}
    dim::Int
    k::Int

    MvLinearPred(d::Int, k::Int) = new(d, k)
end
```

(Multivariate) Affine Prediction

$$f(x;\theta) = W \cdot x + a \cdot b$$

Here, `b` is a model constant to serve as the base of the bias term.

- **parameter:** θ, a matrix of size `(k, d+1)`, in the form `[W a]`, where `W` is a coefficient matrix of size `(k, d)` and `a` is a bias-coefficient vector of size `(k,)`.
- **input:** x, a vector of length `d`.
- **output:** a vector of length `k`.

```
immutable MvAffinePred <: PredictionModel{1,1}
    dim::Int
    k::Int
    bias::Float64

    MvAffinePred(d::Int, k::Int) = new(d, k, 1.0)
    MvAffinePred(d::Int, k::Int, b::Real) = new(d, k, convert(Float64, b))
end
```

1.1.3 Examples

Here is an example that illustrates a prediction model.

```
pm = MvLinearPred(5, 3)    # construct a prediction model
                           # with input dimension 5
                           #      output dimension 3

inputlen(pm)      # --> 5
inputsize(pm)     # --> (5,)
outputlen(pm)     # --> 3
outputsize(pm)    # --> (3,)
paramlen(pm)      # --> 15
paramsize(pm)     # --> (3, 5)

W = randn(3, 5)      # W is a parameter matrix
x = randn(3)         # x is a single input
ninputs(pm, x)       # --> 1
predict(pm, W, x)    # make prediction: --> W * x

X = randn(3, 10)     # X is a matrix with 10 samples
ninputs(pm, X)       # --> 10
predict(pm, W, X)    # make predictions: --> W * X
```

1.2 Loss Functions

Generally, a *loss function* $loss(u, y)$ is to measure the *loss* between the predicted output `u` and the desired response `y`. In this package, all loss functions are instances of the abstract type `Loss`, defined as below:

```
# N is the number of dimensions of each predicted output
# 0 - scalar
# 1 - vector
# 2 - matrix, ...
```

```
#
abstract Loss{N}

typealias UnivariateLoss Loss{0}
typealias MultivariateLoss Loss{1}
```

1.2.1 Common Methods

Methods for Univariate Loss

Each **univariate loss** function implements the following methods:

value(*loss*, *u*, *y*)
Compute the loss value, given the predicted output `u` and the desired response `y`.

deriv(*loss*, *u*, *y*)
Compute the derivative *w.r.t.* `u`.

value_and_deriv(*loss*, *u*, *y*)
Compute both the loss value and derivative (*w.r.t.* `u`) at the same time.

Note: This can be more efficient than calling `value` and `deriv` respectively, when you need both the value and derivative.

Methods for Multivariate Loss

Each **multivariate loss** function implements the following methods:

value(*loss*, *u*, *y*)
Compute the loss value, given the predicted output `u` and the desired response `y`.

grad!(loss, g, u, y)
Compute the gradient *w.r.t.* `u`, and write the results to `g`. This function returns `g`.

Note: `g` is allowed to be the same as `u`, in which case, the content of `u` will be overrided by the derivative values.

---

value_and_grad!(loss, g, u, y)
Compute both the loss value and the derivative *w.r.t.* `u` at the same time. This function returns `(v, g)`, where `v` is the loss value.

Note: `g` is allowed to be the same as `u`, in which case, the content of `u` will be overrided by the derivative values.

For multivariate loss functions, the package also provides the following two generic functions for convenience.

grad(*loss*, *u*, *y*)
Compute and return the gradient *w.r.t.* `u`.

value_and_grad(*loss*, *u*, *y*)
Compute and return both the loss value and the gradient *w.r.t.* `u`, and return them as a 2-tuple.

Both `grad` and `value_and_grad` are thin wrappers of the type-specific methods `grad!` and `value_and_grad!`.

1.2.2 Predefined Loss Functions

This package provides a collection of loss functions that are commonly used in machine learning practice.

Absolute Loss

The *absolute loss*, defined below, is often used for real-valued robust regression:

$$loss(u, y) = |u - y|$$

```
immutable AbsLoss <: UnivariateLoss end
```

Squared Loss

The *squared loss*, defined below, is widely used in real-valued regression:

$$loss(u, y) = \frac{1}{2}(u - y)^2$$

```
immutable SqrLoss <: UnivariateLoss end
```

Quantile Loss

The *quantile loss*, defined below, is used in models for predicting typical values. It can be considered as a skewed version of the *absolute loss*.

$$loss(u, y) = \begin{cases} t \cdot (u - y) & (u \geq y) \\ (1 - t) \cdot (y - u) & (u < y) \end{cases}$$

```
immutable QuantileLoss <: UnivariateLoss
    t::Float64

    function QuantileLoss(t::Real)
        ...
    end
end
```

Huber Loss

The *Huber loss*, defined below, is used mostly in real-valued regression, which is a smoothed version of the *absolute loss*.

$$loss(u, y) = \begin{cases} \frac{1}{2}(u - y)^2 & (|u - y| \leq h) \\ h \cdot |u - y| - \frac{h^2}{2} & (|u - y| > h) \end{cases}$$

```
immutable HuberLoss <: UnivariateLoss
    h::Float64

    function HuberLoss(h::Real)
        ...
    end
end
```

Hinge Loss

The *hinge loss*, defined below, is mainly used for large-margin classification (*e.g.* SVM).

$$loss(u, y) = \max(1 - y \cdot u, 0)$$

```
immutable HingeLoss <: UnivariateLoss end
```

Smoothed Hinge Loss

The *smoothed hinge loss*, defined below, is a smoothed version of the *hinge loss*, which is differentiable everywhere.

$$loss(u, y) = \begin{cases} 0 & (y \cdot u > 1 + h) \\ 1 - y \cdot u & (y \cdot u < 1 - h) \\ \frac{1}{4h}(1 + h - y \cdot u)^2 & (\text{otherwise}) \end{cases}$$

```
immutable SmoothedHingeLoss <: UnivariateLoss
    h::Float64

    function SmoothedHingeLoss(h::Real)
        ...
    end
end
```

Logistic Loss

The *logistic loss*, defined below, is the loss used in the logistic regression.

$$loss(u, y) = log(1 + exp(-y \cdot u))$$

```
immutable LogisticLoss <: UnivariateLoss end
```

Sum Loss

The package provides the *SumLoss* type that turns a univariate loss into a multivariate loss. The definition is given below:

$$loss(u, y) = \sum_{i=1}^{k} intern(u_i, y_i)$$

Here, `intern` is the *internal univariate loss*.

```
immutable SumLoss{L<:UnivariateLoss} <: MultivariateLoss
    intern::L
end

SumLoss{L<:UnivariateLoss}(loss::L) = SumLoss{L}(loss)
```

Moreover, recognizing that sum of squared difference is very widely used. We provide a `SumSqrLoss` as a typealias as follows:

```
typealias SumSqrLoss SumLoss{SqrLoss}
SumSqrLoss() = SumLoss{SqrLoss}(SqrLoss())
```

Multinomial Logistic Loss

The *multinomial logistic loss*, defined below, is the loss used in multinomial logistic regression (for multi-way classification).

$$loss(u, y) = \log\left(\sum_{i=1}^{k} \exp(u_i)\right) - u[y]$$

Here, `y` is the index of the correct class.

```
immutable MultiLogisticLoss <: MultivariateLoss
```

1.3 Risk Models

The prediction model together with a (compatible) loss function constitutes a *risk model*, which can be expressed as $loss(f(x; \theta), y)$.

In this package, we use a type `SupervisedRiskModel` to capture this:

```
abstract RiskModel

immutable SupervisedRiskModel{PM<:PredictionModel,L<:Loss} <: RiskModel
    predmodel::PM
    loss::L
end
```

We also provide a function to construct a risk model:

`riskmodel` (*pm*, *loss*)

Construct a risk model, given the predictio model `pm` and a loss function `loss`.

Here, `pm` and `loss` need to be *compatible*, which means that the output of the prediction and the first argument of the loss function should have the same number of dimensions.

Actually, the definition of `riskmodel` explicitly enforces this consistency:

```
    riskmodel{N,M}(pm::PredictionModel{N,M}, loss::Loss{M}) =
        SupervisedRiskModel{typeof(pm), typeof(loss)}(pm,loss)
```

Note: We may provide other risk model in addition to supervised risk model in future. Currently, the *supervised risk models*, which evaluate the risk by comparing the predictions and the desired responses, are what we focus on.

1.3.1 Common Methods

When a set of inputs and the corresponding outputs are given, the *risk model* can be considered as a function of the parameter θ.

The package provides methods for computing the *total risk* and the derivative of the total risk *w.r.t.* the parameter.

value(*rmodel*, *theta*, *x*, *y*)
Compute the total risk *w.r.t.* the risk model *rmodel*, given

- the prediction parameter `theta`;
- the inputs `x`; and
- the desired responses `y`.

Here, `x` and `y` can be a single sample or matrices comprised of a set of samples.

Example:

```
# constructs a risk model, with a linear prediction
# and a squared loss.
#
#   risk := (theta'x - y)^2 / 2
#
rmodel = risk_model(LinearPred(5), SqrLoss())

theta = randn(5)  # parameter
x = randn(5)      # a single input
y = randn()       # a single output

risk(rmodel, theta, x, y)  # evaluate risk on a single sample (x, y)

X = randn(5, 8)   # a matrix of 8 inputs
Y = randn(8)      # corresponding outputs

risk(rmodel, theta, X, Y)  # evaluate the total risk on (X, Y)
```

value_and_addgrad!(rmodel, beta, g, alpha, theta, x, y)
Compute the total risk on `x` and `y`, and its gradient *w.r.t.* the parameter `theta`, and add it to `g` in the following manner:

$$g \leftarrow \beta g + \alpha \nabla_\theta \mathrm{Risk}(x, y; \theta)$$

Here, `x` and `y` can be a single sample or a set of multiple samples. The function returns both the evaluated value and `g` as a 2-tuple.

Note: When `beta` is zero, the computed gradient (or its scaled version) will be written to `g` without using the original data in `g` (in this case, `g` need not be initialized).

---

value_and_grad(*rmodel*, *theta*, *x*, *y*)
Compute and return the gradient of the total risk on `x` and `y`, *w.r.t.* the parameter `g`.

This is just a thin wrapper of `value_and_addgrad!`.

Note that the `addgrad!` method is provided for risk model with certain combinations of prediction models and loss functions. Below is a list of combinations that we currently support:

- `LinearPred` + `UnivariateLoss`
- `AffinePred` + `UnivariateLoss`

- `MvLinearPred` + `MultivariateLoss`
- `MvAffinePred` + `MultivariateLoss`

If you have a new prediction model that is not defined by the package, you can write your own `addgrad!` method, based on the description above.

1.4 Regularizers

Regularization is important, especially when we don't have a huge amount of training data. Effective regularization can often substantially improve the generalization performance of the estimated model.

In this package, all *regularizers* are instances of the abstract type `Regularizer`.

1.4.1 Common Methods

Each regularizer type implements the following methods:

`value`(*reg*, *theta*)
Evaluate the regularization value at `theta` and return the value.

`value_and_addgrad!(reg, beta, g, alpha, theta)`
Compute the regularization value, and its gradient *w.r.t.* `theta` and add it to `g` in the following way:

$$g \leftarrow \beta g + \alpha \nabla_\theta Reg(\theta)$$

Note: When `beta` is zero, the computed gradient (or its scaled version) will be written to `g` without using the original data in `g` (in this case, `g` need not be initialized).

`prox!(reg, r, theta, lambda)`
Evaluate the proximal operator, as follows:

$$r \leftarrow \operatorname*{argmin}_x \frac{1}{2}\|x - \theta\|^2 + \lambda \cdot \mathrm{Reg}(x)$$

This method is needed when proximal methods are used to solve the problem.

In addition, the package also provides a set of generic wrappers to simplify some use cases.

`value_and_grad`(*reg*, *theta*)
Compute and return the regularization value and its gradient *w.r.t.* `theta`.

This is a wrapper of `value_and_addgrad!`.

`prox`(*reg*, *theta*[, *lambda*])
Evaluate the proximal operator at `theta`. When `lambda` is omitted, it is set to `1` by default.

This is a wrapper of `prox!`.

1.4.2 Predefined Regularizers

The package provides several commonly used regularizers:

Zero Regularizer

The *zero regularizer* always yields the zero value, which is mainly used to supply an regularizer argument to functions to request it (but you do not intend to impose any regularization).

$$reg(\theta) = 0$$

```
immutable ZeroReg <: Regularizer end
```

Squared L2 Regularizer

This is one of the most widely used regularizer in practice.

$$reg(\theta) = \frac{c}{2} \cdot \|\theta\|_2^2.$$

```
immutable SqrL2Reg{T<:FloatingPoint} <: Regularizer
    c::T
end

SqrL2Reg{T<:FloatingPoint}(c::T) = SqrL2Reg{T}(c)
```

L1 Regularizer

This is often used for sparse learning.

$$reg(\theta) = c \cdot \|\theta\|_1$$

```
immutable L1Reg{T<:FloatingPoint} <: Regularizer
    c::T
end

L1Reg{T<:FloatingPoint}(c::T) = L1Reg{T}(c)
```

Elastic Regularizer

This is also known as *L1/L2 regularizer*, which is used in the Elastic Net formulation.

$$reg(\theta) = c_1 \cdot \|\theta\|_1 + \frac{c_2}{2}\|\theta\|_2^2$$

```
immutable ElasticReg{T<:FloatingPoint} <: Regularizer
    c1::T
    c2::T
end

ElasticReg{T<:FloatingPoint}(c1::T, c2::T) = ElasticReg{T}(c1, c2)
```

1.5 Auxiliary Utilities

The package also provides other convenience tools.

no_op(...)
: This function accepts arbitrary arguments and returns nothing.

  It is mainly used as a callback function where you don't really need to callback to do anything.

shrink(x, t)
: Compute the following function:

$$(x, t) \mapsto \begin{cases} x - t & (x > t) \\ 0 & (|x| \leq t) \\ x + t & (x < -t) \end{cases}$$

  Here, `x` can be either a scalar or an array (for vectorized computation).

D

G

I

N

O

P

R

S

V