SGDOptim Documentation

Release 0.1.0

Dahua Lin and contributors

April 11, 2015

Contents

SGDOptim is a Julia package for Stochastic Gradient Descent (SGD), which has become increasingly popular in solving machine learning problems, especially in the context where large-scale datasets are involved.

This package provides types and functions for users to construct *(regularized) empirical risk minization problems*, and use SGD or its variants to solve the problem. Specifically, the package is comprised of the following modules:

Contents:

CHAPTER 1

Construction of Optimization Problems

This package targets an important family of problems in machine learning and data analytics, namely, *(regularized) empirical risk minimization*. Such problems can generally be expressed as:

$$\text{minimize} \ \frac{1}{n}\sum_{i=1}^{n} \text{loss}(f(\theta, x_i), y_i) + \text{reg}(\theta)$$

This objective is comprised of two parts: the *data terms*, which is the average loss evaluated at all samples, and a *regularization term* that encourages lower model complexity. In particular, each *loss term* compares the output of the predictor `f` and a desired output. To sum up, three components are involved: *predictor*, *loss function*, and *regularizer*.

This package provides facilities to construct these components respectively, and the SGD solver will take these components as inputs.

1.1 Predictors

All predictors in this package are organized with the following type hierarchy:

```
abstract Predictor

abstract UnivariatePredictor <: Predictor       # produces scalar output
abstract MultivariatePredictor <: Predictor     # produces vector output
```

1.1.1 Methods

The following methods are provided for each predictor type. Let `pred` be a predictor.

`predict` (*pred*, *theta*, *x*)

Evaluate and return the predicted output, given the parameter `theta` and the input `x`.

The form of the output depends on both the predictor type and the input.

predictor type	input `x`	output
`UnivariatePredictor`	a vector of length `d`	a scalar
`UnivariatePredictor`	a matrix of size `(d, n)`	a vector of length `n`
`MultivariatePredictor`	a vector of length `d`	a vector of length `q`
`MultivariatePredictor`	a matrix of size `(d, n)`	a matrix of size `(q, n)`

Note: for multivariate predictors, the output dimension `q` need not be equal to the input dimension `d`.

scaled_grad!(pred, g, c, theta, x)

Evaluate scaled gradient(s) at given sample(s), writing the resultant gradient(s) to `g`.

When `x` is a vector that represents a single sample, it computes `c` times the gradient and writes the resultant gradient to a pre-allocated vector `g`.

When `x` is a matrix that comprises `n` samples (each being a column), then `c` must be a vector of length `n`, it computes the linear combination of the gradients evaluated at the given samples, using the values in `c` as coefficients. Likewise, the resultant accumulated gradient is written to `g`.

`g` should be of the same size as `theta`.

Note: this function is mainly used by the internal of the optimization algorithms.

The package already provides several commonly used predictors as follows. Users can also implement customized predictors by creating subtypes of `Predictor` and implementing the methods above.

1.1.2 Linear predictor

A *linear predictor* is a real-valued linear functional $f : R^d \to R$, given by

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

In the package, a linear predictor is represented by the type `LinearPredictor`:

```
type LinearPredictor <: UnivariatePredictor
end
```

1.1.3 Affine predictor

An *affine predictor* is a real-valued linear functional $f : R^d \to R$, given by

$$f(x; \theta) := \theta_{1:d}^T x + \theta_{d+1} \cdot \text{bias}$$

Note that the parameter θ is an `d+1`-dimensional vector, which stacks the coefficients for features and a coefficient for the bias.

In the package, an affine predictor is represented by the type `AffinePredictor`:

```
type AffinePredictor{T<:FloatingPoint} <: UnivariatePredictor
    bias::T
end

AffinePredictor{T<:FloatingPoint}(bias::T) = AffinePredictor{T}(bias)
AffinePredictor() = AffinePredictor(1.0)
```

1.1.4 Multivariate linear predictor

A *multivariate linear predictor* is a vector-valued linear functional $f : R^d \to R^q$, given by

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

The parameter θ is a matrix of size `(d, q)`.

In the package, a multivariate linear predictor is represented by the type `MvLinearPredictor`:

```
type MvLinearPredictor <: MultivariatePredictor
end
```

1.1.5 Multivariate affine predictor

A *multivariate affine predictor* is a vector-valued linear functional $f : R^d \to R^q$, given by

$$f_i(x;\theta) := \theta_{1:d,i}^T x + \theta_{d+1,i} \cdot \text{bias},\ \forall i = 1, \ldots, q$$

The parameter $theta$ is a matrix of size `(d+1, q)`.

In the package, a multivariate affine predictor is represented by the type `MvAffinePredictor`:

```
type MvAffinePredictor{T<:FloatingPoint} <: MultivariatePredictor
    bias::T
end

MvAffinePredictor{T<:FloatingPoint}(bias::T) = MvAffinePredictor{T}(bias)
MvAffinePredictor() = MvAffinePredictor(1.0)
```

Note: In the context of classification, one should *directly* use the value(s) yielded by the linear or affine predictors as arguments to the loss function (*e.g. logistic loss* or *multinomial logistic loss*), without converting them into class labels.

1.2 Loss Functions

All loss functions in the package are organized with the following type hierarchy:

```
abstract Loss

abstract UnivariateLoss <: Loss       # for univariate predictions
abstract MultivariateLoss <: Loss     # for multivariate predictions
```

1.2.1 Methods

All *univariate* loss functions should implement the following methods:

`value_and_deriv`(*loss*, *u*, *y*)
: Compute both the loss value and the derivative *w.r.t.* the prediction and return them as a pair, given both the prediction `u` and expected output `y`.

All *multivariate* loss functions should implement the following methods:

`value_and_deriv!(loss, u, y)`
: Compute both the loss value and the derivatives *w.r.t.* the vector-valued predictions, given both the predicted vector `u` and the expected output `y`. It returns the loss value, and overrides `u` with the partial derivatives.

This package already provides a few commonly used loss functions. One can implement customized loss functions by creating subtypes of `Loss` and providing the required methods as above.

1.2.2 Squared loss

The *squared loss*, as defined below, is usually used in linear regression or curve fitting problems:

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

It is represented by the type `SqrLoss`, as:

```
type SqrLoss <: UnivariateLoss
end
```

1.2.3 Hinge loss

The *hinge loss*, as defined below, is usually used for large-margin classification, *e.g.* SVM:

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

It is represented by the type `HingeLoss`, as:

```
type HingeLoss <: UnivariateLoss
end
```

1.2.4 Logisitc loss

The *logistic loss*, as defined below, is usually used for logistic regression:

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

It is represented by the type `LogisticLoss`, as:

```
type LogisticLoss <: UnivariateLoss
end
```

1.2.5 Multinomial Logistic loss

The *multinomial logistic loss*, as defined below, is usually used for multinomial logistic regression (this is often used in the context of multi-way classification):

$$loss(u, y) = \log\left(\sum_{i=1}^{k} e^{u_i}\right) - u_y, \quad u \in R^k, \ y \in 1, \ldots, k$$

Here, `k` is the number of classes. This loss function should be used with a `k`-dimensional multivariate predictor.

It is represented by the type `MultiLogisticLoss`, as:

```
type MultiLogisticLoss <: MultivariateLoss
end
```

1.3 Regularizers

Regularization is important. Using *regularization* can ensures numerical stability and often improves the generalization performance of a model. In this package, regularization is done through *regularizers*, which can be understood as functionals that yield a cost value given a parameter.

1.3.1 Methods

All *regularizers* are subtypes of an abstract type `Regularizer`, and should implement the following methods:

`value_and_addgrad!(reg, g, theta)`
Compute the regularization value and the gradient at the parameter `theta`. It returns the regularization value and writes the gradient to a pre-allocated array `g`.

The size of `g` should be equal to that of `theta`.

The package provides some commonly used regularizers.

1.3.2 No regularization

In certain cases, *e.g.* with a large sample set, people may choose to *not* use regularization. We provide a type `NoReg`, defined below, to indicate no regularization.

type NoReg <: Regularizer end

In SGD algorithms, if no regularizer is explicitly specified, `NoReg()` will be used by default.

1.3.3 Squared L2 norm

The *squared L2 norm regularizer* is defined as

$$r(\theta) = \frac{c}{2}\|\theta\|_2^2$$

It is represented by the type `SqrL2Reg`, as:

```
type SqrL2Reg <: Regularizer
    coef::Float64
end
```

1.3.4 L1 norm

The *L1 norm regularizer* is defined as

$$r(\theta) = c|\theta|_1$$

It is represented by the type `L1Reg`, as:

```
type L1Reg <: Regularizer
    coef::Float64
end
```

This regularizer is often used for sparse learning, *e.g. LASSO*.

1.3.5 Elastic regularizer

The *elastic regularizer* is defined as a combination of L1 norm and squared L2 norm, as:

$$r(\theta) = c_1\|\theta_1\|_1 + c_2\|\theta_2\|_2^2$$

It is represented by the type `ElasticReg`, as:

```
type ElasticReg <: Regularizer
    coef1::Float64
    coef2::Float64
end
```

This is the regularizer used in the well-known algorithm *Elastic Net*.

CHAPTER 2

Sample Streams

Unlike conventional methods, SGD and its variants look at a single sample or a small batch of samples at each iteration. In other words, data are viewed as a stream of samples or minibatches.

This package provides a variety of ways to construct data streams. Each data stream is essentially an iterator that implements the `start`, `done`, and `next` methods (see here for details of Julia's iteration patterns). Each item from a data stream can be either a sample (as a pair of input and output) or a mini-batch (as a pair of multi-input array and multi-output array).

Note: All SGD algorithms in this package support both sample streams and mini-batch streams. At each iteration, the algorithm works on a single item from the stream, which can be either a sample or a mini-batch.

2.1 Wrap Arrays into Data Streams

The package provides several methods to construct streams of samples or minibatches.

`sample_seq`(*X*, *Y*[, *ord*])

Wrap an input array `X` and an output array `Y` into a stream of individual samples.

Each item of the stream is a pair, comprised of an item from `X` and a corresponding item from `Y`. If `X` is a vector, then each item of `X` is a scalar, if `X` is a matrix, then each item of `X` is a column vector. The same applies to `Y`.

The `ord` argument is an instance of `AbstractVector` that specifies the order in which the samples are scanned. If `ord` is omitted, it is, by default, set to the natural order, namely, `1:n`, where `n` is the number of samples in the data set.

`minibatch_seq`(*X*, *Y*, *bsize*[, *ord*])

Wrap an input array `X` and an output array `Y` into a stream of mini-batches of size `bsize` or smaller.

For example, if `X` and `Y` have `28` samples, by setting `bsize` to `10`, we partition the data set into three mini-batches, respectively corresponding to the indices `1:10`, `11:20`, and `21:28`.

The `ord` argument specifies the order in which the mini-batches are used. For example, if `ord` is set to `[3, 2, 1]`, it first takes the 3rd batch, then 2nd, and finally 1st. If `ord` is omitted, it is, by default, set to the natural order, namely, `1:m`, where `m` is the number of mini-batches.

CHAPTER 3

Optimization Algorithms

The package provides functions that implement SGD and its variants.

sgd (*pred*, *loss*, *theta*, *stream*[, ...])
Standard Stochastic Gradient Descent.

Parameters

- **pred** – The predictor.
- **loss** – The loss function.
- **theta** – The initial guess of the solution.
- **stream** – The data stream.

Returns The resultant solution.

This function also supports the following keyword arguments.

name	default	description
`reg`	`NoReg()`	The regularizer.
`lrate`	`t->1.0/(1.0 + t)`	The rule of learning rate, which should be a function of the iteration number `t`.
`cbinterval`	`0`	The interval of invoking callback. •`0`: never invoke callback. •`1`: invoke callback at each iteration. •`k`: invoke callback every `k` iterations.
`callback`	`simple_trace`	The callback function. (See *Callbacks* for details).

Note: The number of iterations is determined by the number of items in the data stream. One can change the behavior of the algorithm by constructing the data stream in different ways. (See *Sample Streams* for details)

More algorithms are being implemented. We will document these algorithms as we proceed.

CHAPTER 4

Callbacks

The algorithms provided in this package interoperate with the rest of the world through *callbacks*. In particular, it allows a third party (*e.g.* a higher-level script, a user, a GUI, etc) to monitor the progress of the optimization and take proper actions.

Generally, a *callback* is an arbitrary function (or closure) that can be called in the following way:

callback(*theta*, *t*, *n*, *v*)

Parameters

- **theta** – The current solution.
- **t** – The number of elapsed iterations.
- **n** – The number of samples that have been used.
- **v** – The objective value of the last item, which can be an objective evaluated on a single sample or the total objective value evaluated on the last batch of samples.

The package already provides some callbacks for simple use:

simple_trace()

Simply print the optimization trace, including the number of iterations, and the average loss of the last iteration.

This is the default choice for most algorithms.

gtcompare_trace(*theta_g*)

In addition to printing the optimization trace, it also computes and shows the deviation from a given oracle `theta_g`.

Note: `gtcompare_trace` is a high-level function, and `gtcompare_trace(theta_g)` produces a callback function.

C

G

M

P

S

V