SALSA Documentation

Release 0.0.5

Vilen Jumutc

August 10, 2016

Contents

SALSA: `Software` Lab for `Advanced` Machine `Learning` with `Stochastic` `Algorithms` is a native Julia implementation of stochastic algorithms for:

- linear and non-linear **Support Vector Machines**
- **sparse linear modelling**

SALSA is an open source project available at Github under the GPLv3 license.

CHAPTER 1

Installation

The **SALSA** package can be installed from the `Julia` command line with `Pkg.add("SALSA")` or by running the same command directly with `Julia` executable by `julia -e 'Pkg.add("SALSA")'`.

CHAPTER 2

Mathematical background

The **SALSA** package aims at stochastically learning a classifier or regressor via the Regularized Empirical Risk Minimization *[Vapnik1992]* framework. We approach a family of the well-known Machine Learning problems of the type:

$$\min_{\mathbf{w}} \sum_{i=1}^{n} \ell(\mathbf{w}, \xi_i) + \Omega(\mathbf{w}),$$

where $\xi_i = (\mathbf{x}_i, y_i)$ is given as a pair of input-output variables and belongs to a set $\mathcal{S} = \{\xi_t\}_{1 \leq t \leq n}$ of independent observations, the loss functions $\ell(\mathbf{w}, \xi_i)$ measures the disagreement between the true target y and the model prediction $\hat{y}$ while the regularization term $\Omega(\mathbf{w})$ penalizes the complexity of the model $\mathbf{w}$. We draw uniformly ξ_i from $\mathcal{S}$ at most T times due of the i.i.d. assumption and a fixed computational budget. Online passes and optimization with the full dataset are available too. The package includes stochastic algorithms for linear and non-linear Support Vector Machines *[Boser1992]* and sparse linear modelling *[Hastie2015]*.

Particular choices of loss functions are (but are not restricted to the selection below):

- hinge loss
- logistic loss
- least squares loss
- etc.

Particular choices of the regularization term are:

- l_2-regularization, *i.e.* $\|w\|_2^2$
- l_1-regularization, *i.e.* $\|w\|_1$
- reweighted l_2-regularization
- reweighted l_1-regularization

CHAPTER 3

References

SALSA is stemmed from the following algorithmic approaches:

- Pegasos: S. Shalev-Shwartz, Y. Singer, N. Srebro, Pegasos: Primal Estimated sub-GrAdient SOlver for SVM, in: Proceedings of the 24th international conference on Machine learning, ICML '07, New York, NY, USA, 2007, pp. 807–814.
- RDA: L. Xiao, Dual averaging methods for regularized stochastic learning and online optimization, J. Mach. Learn. Res. 11 (2010), pp. 2543–2596.
- Adaptive RDA: J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. 12 (2011), pp. 2121–2159.
- Reweighted RDA: V. Jumutc, J.A.K. Suykens, Reweighted stochastic learning, Neurocomputing Special Issue - ISNN2014, 2015. (In Press)

CHAPTER 4

Dependencies

- MLBase: to support generic Machine Learning routines
- StatsBase: to support generic routines from Statistics
- Distances: to support distance metrics between vectors
- Distributions: to support sampling from various distributions
- DataFrames: to support and process files instead of in-memory matrices
- Clustering: to support Stochastic K-means Clustering (experimental feature)
- ProgressMeter: to support progress bars and ETA of different routines

CHAPTER 5

Indices and tables

- genindex
- search

5.1 Data Preprocessing

This part of the package provides a simple set of preprocessing utilities.

5.1.1 Data Normalization

mapstd(*X*)
Normalizes each column of `X` to zero mean and one standard deviation. Output normalized matrix `X` with extracted column-wise means and standard deviations.

```
using SALSA

mapstd([0 1; -1 2]) # --> ([0.707107  -0.707107; -0.707107   0.707107], [-0.5  1.5], [0.707107 0
```

mapstd(*X*, *mean*, *std*)
Normalizes each column of `A` to the specified column-wise `mean` and `std`. Output normalized matrix `X`.

```
using SALSA

mapstd([0 1; -1 2], [-0.5  1.5], [0.707107 0.707107]) # --> [0.707107  -0.707107; -0.707107   0.
```

5.1.2 Sparse Data Preparation

make_sparse(*tuples*[, *sizes*, *delim*])
Creates `SparseMatrixCSC` object from matrix of tuples `Matrix{ASCIIString}` containing `index:value` pairs. The index and value pair can be separated by `delim` character, e.g. `:`. The user can optionally specify final dimensions of the `SparseMatrixCSC` object as `sizes` tuple.

Parameters

- **tuples** – matrix of tuples `Matrix{ASCIIString}` containing `index:value` pairs
- **sizes** – optional tuple of final dimensions, e.g. `(100000,10)` (empty by default)

- **delim** – optional character separating index and value pair in each cell of `tuples`, default is ":"

Returns `SparseMatrixCSC` object.

5.1.3 Data Management

DelimitedFile (*name*, *header*, *delim*)

Creates a wrapper around any delimited file which can be passed to low-level *routines*, for instance *pegasos_alg()*. `DelimitedFile` will be processed in the online mode regardless of the `online_pass==0` flag passed to low-level *routines*.

Parameters

- **name** – file name
- **header** – flag indicating if a header is present
- **delim** – delimiting character

5.2 Classification

A classification example explained by the usage of SALSA package on the Ripley data set. The SALSA package provides many different options for stochastically learning a classification model.

This package provides a function `salsa` and explanation on `SALSAModel` which accompanies and complements it. The package provides full-stack functionality including cross-validation of all model- and algorithm-related hyperparameters.

5.2.1 Knowledge agnostic usage

salsa (*X*, *Y*[, *Xtest*])

Create a linear classification model with the predicted output $\hat{y}$:

$$\hat{y} = \text{sign}(\langle x, w \rangle + b)$$

based on data given in `X` and labeling specified in `Y`. Optionally evaluate it on `Xtest`. Data should be given in row-wise format (one sample per row). The classification model is embedded into the returned `model` as `model.output`. The choice of different algorithms, loss functions and modes will be explained further on this page.

```
using SALSA, MAT, Base.Test

srand(1234)
ripley = matread(joinpath(Pkg.dir("SALSA"), "data", "ripley.mat"))

model = salsa(ripley["X"], ripley["Y"], ripley["Xt"]) # --> SALSAModel(...)
@test_approx_eq_eps mean(ripley["Yt"] .== model.output.Ytest) 0.89 0.01
```

salsa (*mode*, *algorithm*, *loss*, *X*, *Y*, *Xtest*)

Create a classification model with the specified choice of algorithm, mode and loss function.

Parameters

- **mode** – LINEAR vs. NONLINEAR mode specifies whether to use a simple linear classification model or to apply the Nyström method for approximating the feature map before proceeding with the learning scheme
- **algorithm** – stochastic algorithm to learn a classification model, e.g. PEGASOS, L1RDA etc.
- **loss** – loss function to use when learning a classification model, e.g. HINGE, LOGISTIC etc.
- **X** – training data (samples) represented by Matrix or SparseMatrixCSC
- **Y** – training labels
- **Xtest** – test data for out-of-sample evaluation

Returns SALSAModel object.

```
using SALSA, MAT, Base.Test

srand(1234)
ripley = matread(joinpath(Pkg.dir("SALSA"), "data", "ripley.mat"))

model = salsa(LINEAR, PEGASOS, HINGE, ripley["X"], ripley["Y"], ripley["Xt"])
@test_approx_eq_eps mean(ripley["Yt"] .== model.output.Ytest) 0.89 0.01
```

5.2.2 Model-based usage

salsa(*X*, *Y*, *model*, *Xtest*)

Create a classification model based on the provided model and input data

Parameters

- **X** – training data (samples) represented by Matrix or SparseMatrixCSC
- **Y** – training labels
- **Xtest** – test data for out-of-sample evaluation
- **model** – model is of type SALSAModel{L <: Loss, A <: Algorithm, M <: Mode, K <: Kernel} and can be summarized as follows (with default values for named parameters):

•mode::Type{M}: mode used to learn the model: LINEAR vs. NONLINEAR (mandatory parameter)

•algorithm::A: algorithm used to learn the model, e.g. PEGASOS (mandatory parameter)

•loss_function::Type{L}: type of a loss function used to learn the model, e.g. HINGE (mandatory parameter)

•kernel::Type{K} = RBFKernel: kernel used in NONLINEAR mode to compute Nyström approximation

•global_opt::GlobalOpt = CSA(): global optimization techniques for tuning hyperparameters

•subset_size::Float64 = 5e-1: subset size used in NONLINEAR mode to compute Nyström approximation

•max_cv_iter::Int = 1000: maximal number of iterations (budget) for any algorithm in training CV

•max_iter::Int = 1000: maximal number of iterations (budget) for any algorithm for final training

- `max_cv_k::Int = 1`: maximal number of data points used to compute loss derivative in training CV
- `max_k::Int = 1`: maximal number of data points used to compute loss derivative for final training
- `online_pass::Int = 0`: if > 0 we are in the online learning setting going through the entire dataset `online_pass` times
- `normalized::Bool = true`: normalize data (extracting mean and std) before passing it to CV and final learning
- `process_labels::Bool = true`: process labels to comply with binary (-1 vs. 1) or multi-class classification encoding
- `tolerance::Float64 = 1e-5`: the criterion is evaluated for early stopping (`online_pass==0`) $||w_{t+1} - w_t|| <= tolerance$
- `sparsity_cv::Float64 = 2e-1`: sparsity weight in the combined cross-validation/sparsity criterion used for the RDA type of algorithms
- `validation_criterion = MISCLASS()`: validation criterion used to verify the generalization capabilities of the model in cross-validation

Returns `SALSAModel` object with `model.output` of type `OutputModel` structured as follows:

- `dfunc::Function`: loss function derived from the type specified in `loss_function::Type{L}` (above)
- `alg_params::Vector`: vector of model- and algorithm-specific hyperparameters obtained via cross-validation
- `X_mean::Matrix`: row (vector) of extracted column-wise means of input `X` if `normalized::Bool = true`
- `X_std::Matrix`: row (vector) of extracted column-wise standard deviations of input `X` if `normalized::Bool = true`
- `mode::M`: mode used to learn the model: LINEAR vs. NONLINEAR
- `w`: found solution vector (matrix)
- `b`: found solution offset (bias)

```
using SALSA, MAT, Base.Test

srand(1234)
ripley = matread(joinpath(Pkg.dir("SALSA"), "data", "ripley.mat"))

model = SALSAModel(NONLINEAR, R_L1RDA(), HINGE, global_opt=CSA())
model = salsa(ripley["X"], ripley["Y"], model, ripley["Xt"])
@test_approx_eq_eps mean(ripley["Yt"] .== model.output.Ytest) 0.895 0.01
```

5.3 Regression

A regression example is explained for the SALSA package by the `sinc(x) = sin(x)./x` function.

This package provides a function `salsa` and explanation on `SALSAModel` for the regression case. This use case is supported by the Fixed-Size approach *[FS2010]* and Nyström approximation with the specific *`LEAST_SQUARES()`* loss function and cross-validation criterion *`mse()`* (mean-squared error).

```
using SALSA, Base.Test

srand(1234)
sinc(x) = sin(x)./x
X = linspace(0.1,20,100)''
Xtest = linspace(0.11,19.9,100)''
y = sinc(X)

model = SALSAModel(NONLINEAR, SIMPLE_SGD(), LEAST_SQUARES,
                           validation_criterion=MSE(), process_labels=false)
model = salsa(X, y, model, Xtest)

@test_approx_eq_eps mse(sinc(Xtest), model.output.Ytest) 0.05 0.01
```

By taking a look at the code snippet above we can notice a major difference with the Classification example. The model is equipped with the `NONLINEAR` mode, `LEAST_SQUARES` loss function while the cross-validation criterion is given by `MSE`. Another important model-related parameter is `process_labels` which should be set to `false` in order to switch into regression mode. These four essential components unambiguously define a regression problem solved stochastically by the `SALSA` package.

5.4 Clustering

A clustering example is explained for the SALSA package on the Iris dataset *[UCI2010]*.

This package provides a function `salsa` and explanation on `SALSAModel` for the clustering case. This use case is supported by the particular choices of loss functions and distance metrics applied within the Regularized K-Means approach *[JS2015]* and cross-validation criterion `SILHOUETTE` (Silhouette index).

```
using SALSA, Clustering, Distances, MLBase, Base.Test

Xf = readcsv(joinpath(Pkg.dir("SALSA"), "data", "iris.data.csv"))
Y = convert(Array{Int}, Xf[:,end])
k_clusters = length(unique(Y))
dY = Array{Int}(length(Y))
X = Xf[:,1:end-1]
srand(1234)

algorithm = RK_MEANS(k_clusters)
model = SALSAModel(LINEAR, algorithm, LEAST_SQUARES,
         validation_criterion=SILHOUETTE(),
         global_opt=DS([-1]), process_labels=false,
         cv_gen = Nullable{CrossValGenerator}(Kfold(length(Y),3)))
model = salsa(X, dY, model, X)
mappings = model.output.Ytest
```

By taking a close look at the code snippet above we can notice that we use a special type of an algorithm *RK_MEANS()* which implements approach in *[JS2015]*. By instantiating `RK_MEANS(k_clusters)` we provide a maximum number of clusters to be extracted. Learning of individual prototype vectors will be repeated `algorithm.max_iter` times after re-partitioning of the dataset `X` (by default `algorithm.max_iter==20`). The default choice of the loss function is `LEAST_SQUARES` and the distance metric is `Euclidean()` [1]. This corresponds to the original setting of the unregularized K-Means approach. Please refer to Algorithms section and *RK_MEANS()* function for more details regarding which combinations of loss functions and metrics are supported.

[1] metric types are defined in Distances.jl package

5.5 Loss Functions

This part of the package provides a description and mathematical background of the implemented loss functions. Every loss function can be supplied to `salsa` subroutines either directly (see *salsa()*) or passed within `SALSAModel`. In the definitions below $l(y, p)$ stands for the loss loss function evaluated at the true label y and a prediction p.

HINGE()
Defines an implementation of the Hinge Loss function, *i.e.* $l(y, p) = \max(0, 1 - yp)$.

LOGISTIC()
Defines an implementation of the Logistic Loss function, *i.e.* $l(y, p) = \log(1 + \exp(-yp))$.

LEAST_SQUARES()
Defines an implementation of the Least Squares Loss function, *i.e.* $l(y, p) = \frac{1}{2}(p - y)^2$.

SQUARED_HINGE()
Defines an implementation of the Squared Hinge Loss function, *i.e.* $l(y, p) = \max(0, 1 - yp)^2$.

PINBALL()
Defines an implementation of the Pinball (Quantile) Loss function, *i.e.*

$$l(y, p) = \begin{cases} 1 - yp, & \text{if yp} \leq 1, \\ \tau(yp - 1), & \text{otherwise} \end{cases}$$

If `PINBALL` loss is selected τ parameter will be tuned by the build-in cross-validation routines.

MODIFIED_HUBER()
Defines an implementation of the Modified Huber Loss function, *i.e.*

$$l(y, p) = \begin{cases} -4yp, & \text{if yp} < -1 \\ \max(0, 1 - yp)^2, & \text{otherwise} \end{cases}$$

loss_derivative(*type*)
Defines a derivative of the loss function. One can pass any type of the loss function, *e.g.* `HINGE` or an entire algorithm, for instance *RK_MEANS()*.

Parameters **type** – type of the loss function, *e.g.* `HINGE` or an entire algorithm

Returns `Function` which calculates a derivative at the current iterate w_t, subsample $\mathcal{A}_t$ and label y_t

5.6 Algorithms

This part of the package provides a description, API and references to the implemented core algorithmic schemes (solvers) available in the SALSA package. Every algorithm can be supplied as a type to `salsa` subroutines either directly (see *salsa()*) or passed within `SALSAModel`. Please refer to Classification section for examples. Another available API is shipped with direct calls to algorithmic schemes. The latter is the most primitive and basic way of using SALSA package.

5.6.1 Available high-level API

PEGASOS()
Defines an implementation (see *pegasos_alg()*) of the Pegasos: Primal Estimated sub-GrAdient SOlver for SVM which solves l_2-regularized problem defined *here*.

L1RDA()
Defines an implementation (see *l1rda_alg()*) of the l1-Regularized Dual Averaging solver which solves l_1-regularized problem defined *here*.

ADA_L1RDA()
Defines an implementation (see *adaptive_l1rda_alg()*) of the Adaptive l1-Regularized Dual Averaging solver which solves l_1-regularized problem defined *here* in an adaptive way [1].

R_L1RDA()
Defines an implementation (see *reweighted_l1rda_alg()*) of the Reweighted l1-Regularized Dual Averaging solver which approximates l_0-regularized problem in a limit.

R_L2RDA()
Defines an implementation (see *reweighted_l2rda_alg()*) of the Reweighted l2-Regularized Dual Averaging solver which approximates l_0-regularized problem in a limit.

SIMPLE_SGD()
Defines an implementation (see *sgd_alg()*) of the unconstrained Stochastic Gradient Descent scheme which solves l_2-regularized problem defined *here*.

RK_MEANS(*support_alg*, *k_clusters*, *max_iter*, *metric*)
Defines an implementation (see *stochastic_rk_means()*) of the Regularized Stochastic K-Means approach *[JS2015]*. Please refer to Clustering section for examples.

Parameters

- **support_alg** – underlying support algorithm, *e.g.* `PEGASOS`
- **k_clusters** – number of clusters to be extracted
- **max_iter** – maximum number of outer iterations
- **metric** – metric to evaluate distances to centroids [2]

Selected `metric` unambiguously define a loss function used to learn centroids. Currently supported metrics are:

•`Euclidean()` which is complemented by *LEAST_SQUARES()* loss function

•`CosineDist()` which is complemented by *HINGE()* loss function

5.6.2 Available low-level API

pegasos_alg(*dfunc*, *X*, *Y*, λ, *k*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)
- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$

[1] adaptation is taken with respect to observed (sub)gradients of the loss function
[2] metric types are defined in Distances.jl package

- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

sgd_alg (*dfunc*, *X*, *Y*, λ, *k*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)
- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

l1rda_alg (*dfunc*, *X*, *Y*, λ, γ, ρ, *k*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)
- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- γ – hyperparameter involved in elastic-net regularization
- ρ – hyperparameter involved in elastic-net regularization
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

adaptive_l1rda_alg (*dfunc*, *X*, *Y*, λ, γ, ρ, *k*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)

- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- γ – hyperparameter involved in elastic-net regularization
- ρ – hyperparameter involved in elastic-net regularization
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

reweighted_l1rda_alg (*dfunc*, *X*, *Y*, λ, γ, ρ, , *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)
- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- γ – hyperparameter involved in reweighted formulation of a regularization term
- ρ – hyperparameter involved in reweighted formulation of a regularization term
- – reweighting hyperparameter
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

reweighted_l2rda_alg (*dfunc*, *X*, *Y*, λ, , *var*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **dfunc** – supplied loss function derivative (see *loss_derivative()*)
- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **Y** – training labels corresponding to `X`
- λ – trade-off hyperparameter
- – reweighting hyperparameter

- **var** – sparsification hyperparameter
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

stochastic_rk_means (*X*, *rk_means*, *alg_params*, *max_iter*, *tolerance*[, *online_pass=0*, *train_idx=[]*])

Parameters

- **X** – training data (samples are stacked row-wise) represented by `Matrix`, `SparseMatrixCSC` or *DelimitedFile()*
- **rk_means** – algorithm defined by *RK_MEANS()*
- **alg_params** – hyperparameter of the supporting algorithm in `rk_means.support_alg`
- **k** – sampling size at each iteration t
- **max_iter** – maximum number of iterations (budget)
- **tolerance** – early stopping threshold, *i.e.* $||w_{t+1} - w_t|| <= tolerance$
- **online_pass** – number of online passes through data, `online_pass=0` indicates a default stochastic mode instead of an online mode
- **train_idx** – subset of indices from `X` used to learn a model (w, b)

Returns w, b

5.7 Model Tuning

This part of the package provides a simple API for model-tuning routines.

gen_cross_validate (*evalfun*, *n*, *model*)

Perform in parallel a generic cross-validation (CV) routine defined in `evalfun` by the splitting specified in `model.cv_gen`.

Parameters

- **evalfun** – function to evaluate
- **n** – total number of data points (instances) to create `Kfold` CV generator if `model.cv_gen` is undefined (null)
- **model** – `SALSAModel` which contains the `cv_gen` field of type `Nullable{CrossValGenerator}` [1] or `model.output.cv_folds` field containing predefined indices for each fold

Returns an average of `evalfun` evaluations.

misclass (*y*, *yhat*)

Calculate misclassification rate as $\frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$.

[1] wrapper around the type defined in MLBase.jl package

mse (*y*, *yhat*)
Calculate mean squared error as $\frac{1}{n}\|y - \hat{y}\|^2$

auc (*y*, *yhat*[, *n=100*])
Calculate Area Under ROC Curve. Default number of thresholds is 100.

5.8 Nyström Approximation

While linear techniques operating in the primal (input) space are able to achieve good generalization capabilities in some specific application areas, one cannot in general approximate with the linear model more complex or highly nonlinear functions. We apply a Fixed-Size approach *[FS2010]* and Nyström approximation *[WS2001]* to approximate a kernel-induced feature map with some higher dimensional explicit and approximate feature vector.

We select prototype vectors (a small working sample of size $m \ll n$) and construct, for instance an RBF kernel matrix K with

$$K_{ij} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}.$$

By following the approach in *[WS2001]* an expression for the entries of the approximated feature map $\hat{\Phi}(x) : \mathbb{R}^d \to \mathbb{R}^m$, with $\hat{\Phi}(x) = (\hat{\Phi}_1(x), \ldots, \hat{\Phi}_m(x))^T$ is given by

$$\hat{\Phi}_i(x) = \frac{1}{\sqrt{\lambda_{i,m}}} \sum_{t=1}^{m} u_{ti,m} k(x_t, x),$$

where $\lambda_{i,m}$ and $u_{i,m}$ denote the *i*-th eigenvalue and the *i*-th eigenvector of K.

5.8.1 Available API

AFEm (*Xs*, *kernel*, *X*)
Performs Automatic Feature Extraction (AFE) by Nyström method *[WS2001]* using a subsample $X_s \in X$. We restrict `kernel <: Kernel` to be a subclass of `Kernel`, for instance `RBFKernel`.

Parameters

- **Xs** – subset which is used to construct kernel matrix K
- **kernel** – kernel function, *e.g.* `RBFKernel()`, used to construct kernel matrix K
- **X** – full dataset

Returns new dataset X_f derived from stacking together feature maps for every $x_i \in X$

entropy_subset (*X*, *kernel*, *subset_size*)
Performs maximization of the quadratic Rényi Entropy by the representative points selection from `X` which can be supplied to `AFEm` as `Xs` subset.

Parameters

- **X** – full dataset
- **kernel** – kernel function, *e.g.* `RBFKernel()`, used to construct kernel matrix K over which we compute Rényi Entropy
- **subset_size** – number of representative data points

5.8.2 Available Kernel Functions

LinearKernel()
Defines an implementation of the Linear Kernel, *i.e.* $k(x, y) = \langle x, y \rangle$.

PolynomialKernel()
Defines an implementation of the Polynomial Kernel, *i.e.* $k(x, y) = (\langle x, y \rangle + \tau)^d$.

RBFKernel()
Defines an implementation of the Radial Basis Function (RBF) Kernel, *i.e.* $k(x, y) = \exp(-\frac{\|x-y\|^2}{2\sigma^2})$.

5.9 Examples & notebooks

5.9.1 Prerequisites

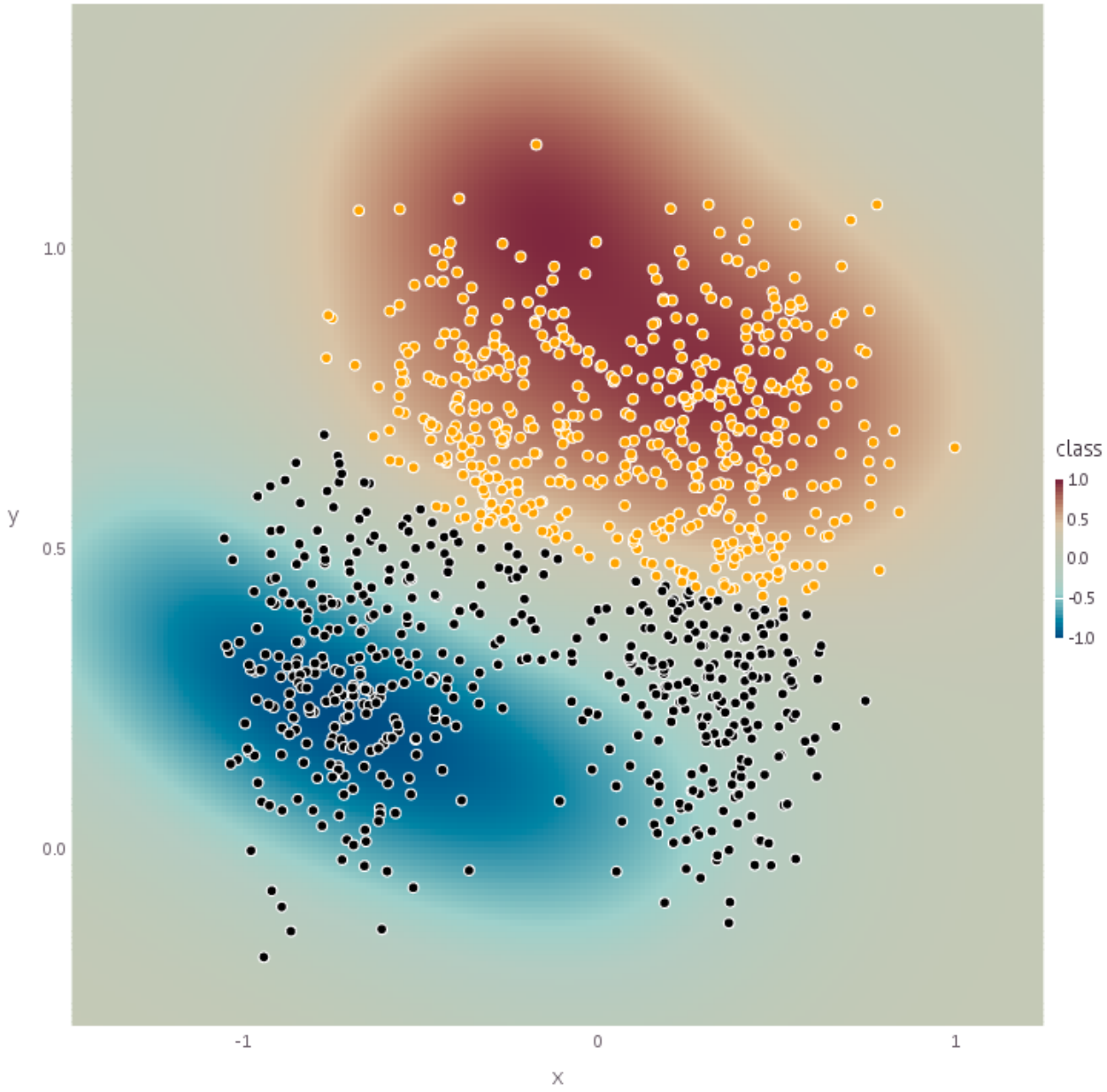
Please refer to Julia downloads page for installing **Julia** language and all dependencies. The instructions for installing the **SALSA** package can be found *here*. Some additional plotting and data management packages might be required to run examples below (like `Gadfly`, `MAT` or `DataFrames`). If you prefer Python-style notebooks please refer to the Project Jupyter and IJulia package for instructions. In this section we provide code snippets which can be easily copied into the **Julia** console or **Jupyter** notebook. Please find an explanation on examples and functional `IJulia` notebooks online.

5.9.2 Advanced Classification

This example provides a use-case for nonlinear classification using Nyström approximation and Area Under ROC Curve (with 100 thresholds) as a cross-validation criterion.

```
using SALSA, MAT

ripley = matread(joinpath(Pkg.dir("SALSA"), "data", "ripley.mat")); srand(123);
model = SALSAModel(NONLINEAR, PEGASOS(), LOGISTIC, validation_criterion=AUC(100));
model = salsa(ripley["X"], ripley["Y"], model, ripley["Xt"]);

range1 = linspace(-1.5,1.5,200);
range2 = linspace(-0.5,1.5,200);
grid = [[i j] for i in range1, j in range2];

Xgrid = foldl(vcat, grid);
Xtest = ripley["Xt"];

yhat = model.output.Ytest;
yplot = map_predict_latent(model,Xgrid);
yplot = yplot - minimum(yplot);
yplot = 2*(yplot ./ maximum(yplot)) - 1;

using DataFrames
df = DataFrame();
df[:X] = Xgrid[:,1][:];
df[:Y] = Xgrid[:,2][:];
df[:class] = yplot[:];

using Gadfly
set_default_plot_size(20cm, 20cm);
plot(layer(x=Xtest[yhat.>0,1], y=Xtest[yhat.>0,2], Geom.point, Theme(default_color=colorant"orange"))
```

```
    layer(x=Xtest[yhat.<0,1], y=Xtest[yhat.<0,2], Geom.point, Theme(default_color=colorant"black")),
    layer(df, x="X", y="Y", color="class", Geom.rectbin))
```

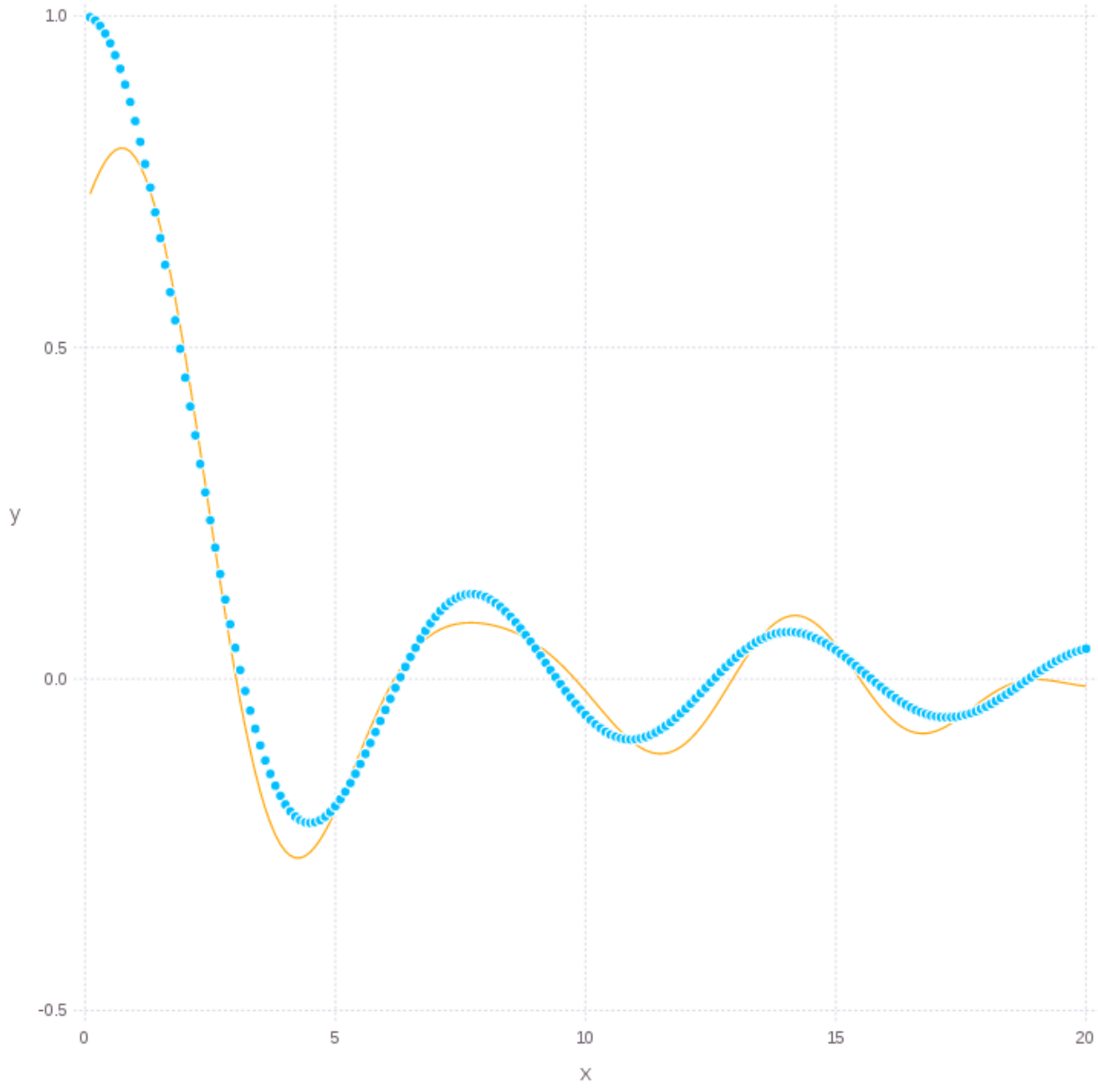


5.9.3 Advanced Regression

This example provides a use-case for regression using Nyström approximation and *mse()* (Mean Squared Error) as a criterion in the Leave-One-Out cross-validation defined in MLBase.jl package.

```
using SALSA, MLBase

sinc(x) = sin(x)./x;
X = linspace(0.1,20,100)'';
Xtest = linspace(0.1,20,200)'';
Y = sinc(X);
```

```
srand(1234);

model = SALSAModel(NONLINEAR, PEGASOS(), LEAST_SQUARES,
     cv_gen=Nullable{CrossValGenerator}(LOOCV(100)),
     validation_criterion=MSE(), process_labels=false, subset_size=5.0);
model = salsa(X, Y, model, Xtest);

using Gadfly
set_default_plot_size(20cm, 20cm);
plot(layer(x=Xtest[:], y=sinc(Xtest), Geom.point),
     layer(x=Xtest[:], y=model.output.Ytest, Geom.line, Theme(default_color=colorant"orange")))
```



Bibliography

[Vapnik1992] Vapnik, Vladimir. “Principles of risk minimization for learning theory”, In Advances in neural information processing systems (NIPS), pp. 831-838. 1992.

[Boser1992] Boser, B., Guyon, I., Vapnik, V. “A training algorithm for optimal margin classifiers”, In Proceedings of the fifth annual workshop on Computational learning theory - COLT‘92., pp. 144-152, 1992.

[Hastie2015] Hastie T., Tibshirani R., Wainwright M. Statistical Learning with Sparsity: The Lasso and Generalizations, Chapman & Hall/CRC Monographs on Statistics & Applied Probability, 2015.

[FS2010] De Brabanter K., De Brabanter J., Suykens J.A.K., De Moor B., “Optimized Fixed-Size Kernel Models for Large Data Sets”, Computational Statistics & Data Analysis, vol. 54, no. 6, Jun. 2010, pp. 1484-1504.

[UCI2010] Lichman, M. (2013). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

[JS2015] Jumutc V., Suykens J.A.K., “Regularized and Sparse Stochastic K-Means for Distributed Large-Scale Clustering”, Internal Report 15-126, ESAT-SISTA, KU Leuven (Leuven, Belgium), 2015.

[WS2001] Williams C. and Seeger M., “Using the Nyström method to speed up kernel machines”, in Proceedings of the 14th Annual Conference on Neural Information Processing (NIPS), pp. 682-688, 2001.