Nuclear ADMM API

Hristo Paskov

September 29, 2019

1 Introduction

The NuclearADMM Matlab package is a fast, ADMM-based algorithm for solving Nuclear Norm regularized regression problems. Specifically, we wish to solve T regression problems jointly so that information is "shared" between tasks. What enables this information sharing is that each of the regression tasks uses the same set of d predictors, i.e. each regression problem solves for a d-dimensional coefficient vector. The t^{th} task has n_t observations represented via a target vector $\mathbf{y}_t \in \mathbb{R}^{n_t}$ and a feature matrix $\mathbf{X}_t \in \mathbb{R}^{n_i \times d}$. The $\mathbf{X}_t$ are (row-wise) submatrices of a joint feature matrix $X \in \mathbb{R}^{n \times d}$ that stores *all* training examples aggregated across each of the tasks. We wish to learn a matrix $W \in \mathbb{R}^{d \times T}$ of regression coefficients whose t^{th} column, W_t, stores the regression coefficients for task t. Our learning problem can be expressed as a minimization problem over W and $b \in \mathbb{R}^T$ (a vector of offsets) with objective

$$\underset{\substack{W \in \mathbb{R}^{d \times T} \\ b \in \mathbb{R}^T}}{\text{minimize}} \; \frac{1}{2} \sum_{t=1}^{T} \|\mathbf{y}_t - \mathbf{X}_t W_t - b_t \mathbf{1}\|_2^2 + \lambda \|W\|_* \,.$$

The key to this package's speed is that computations can be reused for tasks that share a large number of training examples in common. While a detailed discussion of how this structure is leveraged is outside the scope of this document, efficiencies come from observing that the difference between covariance matrices $\mathbf{X}_t^\top \mathbf{X}_t - \mathbf{X}_s^\top \mathbf{X}_s$ is low rank whenever tasks s, t share many points in common. To this end, it will be useful to partition the tasks into G non-overlapping sets of tasks $J_i \subset \{1, \ldots, T\}$ for $i = 1, \ldots, G$. This organization is up to end user. Tasks that have similar training sets should be placed into the same group to save on computation time and memory. For example, Figure 1 plots 20 tasks that naturally cluster into 3 distinct task sets that result in optimal performance for NuclearADMM. Please note that dissimilar tasks that are incorrectly grouped together may increase solving times substantially.

Figure 1: A 15×20 grid in which each row corresponds to a distinct training example and each column to a task. A red square at position row i, column t indicates that point i is used as training data for task t.

2 API Reference

With these concepts in mind, we now describe the API for training, cross-validation, and prediction. The functions this API describes were written in a modular way to allow for a variety of losses and regularization penalties. We focus on parameters that specifically implement the aforementioned cache-optimized regression loss and a nuclear-norm regularizer.

2.1 Training

`[W_path,b_path] = ADMMPath(X, Y, task_indices, lambdas, init_loss, init_reg, prox_loss, prox_reg, init_loss_args, init_reg_args, finalize, update_rho)`

Description: Trains a nuclear-norm regularized multi-task regression problem over a sequence of regularization values.

Input:

- `X` : the $n \times d$ joint feature matrix containing all training examples that can be encountered any of the tasks.
- `Y` : a T-dimensional cell array in which `Y{t}`$= \mathbf{y}_t$.
- task_indices : a T-dimensional cell array in which `task_indices{t}` is a n_t-dimensional array of 1-based indices in the set $\{1, \ldots, n\}$. `task_indices{t}(j)` specifies that target value `Y{t}(j)` matches up to row `task_indices{t}(j)` in `X`.
- `lambdas` : an L-dimensional array of decreasing regularization values, i.e. of λ-values for the objective. Please see documentation below for how to select the regularization grid.
- `init_loss` : set to `@init_regression` (Matlab function handle) for regression problems. This argument will later be extended to allow for classification, etc.

- `init_reg` : set to `@init_nuclear` (Matlab function handle) for nuclear norm regularization. This argument will later be extended to allow for other forms of regularization.

- `prox_loss` : set to `@prox_regression` (Matlab function handle) for regression problems. This argument will later be extended to allow for classification, etc.

- `prox_reg` : set to `@prox_nuclear` (Matlab function handle) for nuclear norm regularization. This argument will later be extended to allow for other forms of regularization.

- `init_loss_args` : cell array of arguments for the regression. This should either be empty or of even length and it follows the format, for i odd, `init_loss_args{i}="option"` and `init_loss_args{i+1}=parameter` . Default values are used for any options that are not specified. The following table lists the available options:

  - `groups` : A cell array of length G that splits the T tasks into G task sets. The i^{th} cell is an array of task indices in the range $\{1, \ldots, T\}$ that specifies which tasks belong to this task set (i.e. the J_i). Note that each task must appear one and only one time in a cell (i.e. each task must belong to exactly one task set). Default value is `{1:T}`.
  - `xscale` : How to scale the columns in each *task set* after mean centering (note that all tasks in the same task set use the same scaling):
    - `"Std"` : divide each column by its standard deviation
    - `"L1"` : divide each column by its ℓ_1 norm
    - `"None"` : perform no normalization
    - Default : "Std"
  - `yscale` : How to scale each `Y{t}` (after mean centering):
    - `"L2"` : divide by the ℓ_2 norm
    - `"L1"` : divide by the ℓ_2 norm
    - `"Linf"` : divide by the maximum absolute value
    - `"None"` : perform no normalization
    - Default : 'None'

- `init_reg_args` : set to `[]` (i.e. empty); the nuclear norm does not require arguments

- `finalize` : set to `@finalize_regression` (Matlab function handle) for regression problems. This argument enables later extensions and is useful for teardown.

- `update_rho` : boolean for how to update ADMM's step size, should be set to true, but can be set to false if ADMM behaves erratically (but may lead to very slow convergence).

Output:

- `W_path` : a T-dimensional cell array in which cell `W_path{t}` is a $d \times L$ (L is number of regularization parameters to try) matrix of regression coefficients for task t. `W_path{t}(:,i)` is the regularization coefficients for the i^{th} regularization parameter.

- `b_path` : a T-dimensional cell array in which `b_path{t}` is a $1 \times L$ vector of offsets corresponding to each regularization parameter.

2.2 Cross-Validation

`[predicted, foldID ] = ADMMCV(foldID, method, nfolds, X, Y, task_indices, lambdas, init_loss, init_reg, prox_loss, prox_reg, init_loss_args, init_reg_args, finalize, update_rho)`

Description: Performs nfold-cross-validation on a nuclear norm regularized multitask regression problem using either prespecified splits or automatically generated ones.

Input:

- `foldID` : either `[]` (empty) for automatic splitting of points into cross-validation folds or a T-dimensional cell array specifying how to split points. If specified, `foldID{t}` is a n_t-dimensional array with entries in $1, \ldots,$ nfolds and `foldID{t}[j]` specifies that the example with target value `Y{t}[j]` will be held out at fold number `foldID{t}[j]`.

- `method` : only checked if `foldID` is empty. `"full"` specifies that the training example in the j^{th} row of X will be held out across all tasks (in which it appears) when it is held out, whereas `"random"` allows the point to be held out at different folds for different tasks.

- `nfolds` : the number of folds of cross-validation. If `foldID` is specified this must equal the largest value in `foldID`.

- All other inputs are the same as in `ADMMPath`.

Output:

- `predicted` : a cell array of size T in which `predicted{t}` is a $n_t \times L$ (L is the number of regularization parameters to try) matrix that gives the predicted value for each point when it was held out for each of the regularization parameters in `lambdas`.

- `foldID` : if `foldID` was not specified as input, this is filled in and specifies which fold each point was held out in. Otherwise it returns the argument.

2.3 Regularization Grid Estimation

`[ lambdas ] = nuclear_lambda( X, Y, task_indices, init_loss_args )`

Description: Returns a grid of 100 decreasing regularization parameters to try for the nuclear norm regularized multitask regression problem. This function is still being improved, so if you see that testing/cross-validation error as a function of regularization parameter is flat or still decreasing at the end of the grid, it will be beneficial to modify the suggested grid.

Input: All arguments are as in `ADMMPath`.

Output:

- `lambdas` : a grid of 100 regularization parameters to try.

2.4 Prediction

`[ predicted ] = Predict( X, task_indices, W, b )`

Description: For each task t, predicts the target values for the subset of points in X specified by `task_indices{t}` using the regression coefficients in `W{t}` and offset `b{t}`.

Input:

- `X` : a $m \times d$ matrix specifying all the points to make predictions on.
- `task_indices` : a T-dimensional cell array in which `task_indices{t}` is a p_t-dimensional array with values in $\{1, \ldots, m\}$ that specifies which subset of points in X to make predictions for, for task t.
- `W` : a T-dimensional cell array in which cell `W{t}` is a $d \times L$ (L is the number of regularization parameters to try) matrix of regression coefficients for task t to make predictions with.
- `b` : a T-dimensional cell array in which `b{t}` is a 1 $timesL$ vector of offsets corresponding to each regularization parameter used to make predictions for task t.

Output:

- `predicted` : a T-dimensional cell array in which `predicted{t}` is a $p_t \times L$ matrix containing the predictions for task t. The order of points matches the order specified in `task_indices`.