ADMML: Alternating Direction Method based scalable Machine Learning on Spark

Sauptik Dhar SAUPTIK.DHAR@US.BOSCH.COM
Jeff Irion JEFF.IRION@US.BOSCH.COM
Jiayi Liu JIAYI.LIU2@US.BOSCH.COM
Unmesh Kurup UNMESH.KURUP@US.BOSCH.COM
Mohak Shah [*] MOHAK.SHAH@US.BOSCH.COM
Bosch Center for Artificial Intelligence, 4009 Miranda Ave, Palo Alto, CA 94304

Contents

[*]. also at University of Illinois at Chicago, IL

1. Introduction

The advent of big-data has seen an emergence of research on scalable machine learning (ML) algorithms and big-data platforms. Several storage and analytics frameworks have been introduced to handle this data deluge such as, (Hadoop), Haloop (Bu et al., 2010) and (Spark). Among them, Spark has been widely used by the ML community. Spark supports distributed in-memory computations and provides practitioners with a powerful, fast, scalable, and easy way to build ML algorithms. Although there have been several Spark-based ML libraries, only a few of them cover a wide range of problems with fast and accurate results. Further, the stochastic gradient descent (SGD) and limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) based approaches adopted in these libraries present with the following limitations,

- SGD: convergence dependent on step-size, conditionality of the data (Bertsekas, 1999),
- L-BFGS: adapting to non-differentiable functions is non-trivial (Henao, 2014).

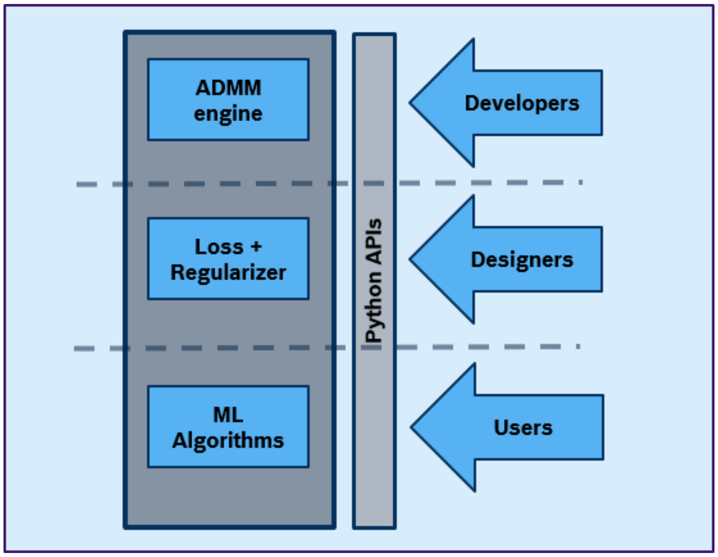

Figure 1: ADMML tool accesibility for the Python ML community.

In this report we introduce the **A**lternating **D**irection **M**ethod based scalable **M**achine **L**earning (ADMML) toolkit on Spark. ADMML solves a generic machine learning formulation that covers a wide range of algorithms (including non-differentiable loss functions) with robust guarantees on convergence and accuracy (Boyd et al., 2011). Finally, the entire tool is implemented in Python. This provides big-data analytics capabilities for the Python community (see Fig. 1), including:

- *Users* interested in using ML algorithms directly for several real-world applications.
- *Designers* interested in designing newer algorithms by combining different loss/regularization functions.

– *Developers* interested in modifying the ADMM (optimization) engine.

Currently, none of the existing open-source big-data machine learning tools on Spark, like MLLib (Meng et al., 2016), PhotonML (LinkedIn, 2016), etc. provide access to this entire spectrum of development for Python users.

2. ADMML Machine Learning algorithms

ADMM was first proposed in the mid-70s by Glowinski and Marroco (1975) and Gabay and Mercier (1976) as a general convex optimization algorithm and has recently been popularized by (Boyd et al., 2011). Lately there has been a tremendous amount of research on ADMM due to its applicability to the distributed data setting. As an outcome of this research, ADMM presents itself as a competitive technique for distributed optimization. A critical feature of the ADMM formulation is that it divides an optimization problem into smaller sub-problems and enables solutions to them in a distributed setting. Next we present a brief description of the ADMM methodology. A more detailed description can be found in (Boyd et al., 2011).

2.1 The ADMM algorithm (Basic Form)

Let's consider optimization problems of the following form[1]:

$$\begin{aligned} \min_{\mathbf{w},\mathbf{z}} \quad & f(\mathbf{w}) + g(\mathbf{z}) \\ \text{s.t.} \quad & A\mathbf{w} + B\mathbf{z} = \mathbf{c} \end{aligned} \tag{1}$$

where $\mathbf{w}, \mathbf{z} \in \Re^D$. We form the *augmented Lagrangian* given below,

$$L_\rho(\mathbf{w}, \mathbf{z}, \mathbf{y}) = f(\mathbf{w}) + g(\mathbf{z}) + \mathbf{y}^\top (A\mathbf{w} + B\mathbf{z} - \mathbf{c}) + \frac{\rho}{2}\|A\mathbf{w} + B\mathbf{z} - \mathbf{c}\|_2^2, \tag{2}$$

where $\mathbf{y}$ is the Lagrange multiplier. Note that the augmented Lagrangian contains a quadratic penalty term in addition to the usual Lagrangian which is controlled by the penalization factor ρ (see (Boyd et al., 2011) for details). Then the ADMM iterations to solve eq. 1 are:

$$\begin{aligned} \mathbf{w}^{k+1} &= \arg\min_{\mathbf{w}} \quad L_\rho(\mathbf{w}, \mathbf{z}^k, \mathbf{y}^k) \\ \mathbf{z}^{k+1} &= \arg\min_{\mathbf{z}} \quad L_\rho(\mathbf{w}^{k+1}, \mathbf{z}, \mathbf{y}^k) \\ \mathbf{y}^{k+1} &= \mathbf{y}^k + \rho(A\mathbf{w}^{k+1} + B\mathbf{z}^{k+1} - \mathbf{c}). \end{aligned} \tag{3}$$

For practical purposes, a more widely used version is the scaled ADMM. Typically in that case, the linear and the quadratic terms of the primal residual $r = A\mathbf{w} + B\mathbf{z} - \mathbf{c}$ in eq. 3

[1] Note that we use lowercase bold alphabets for representing vectors throughout the paper.

are combined and the resulting ADMM updates become

$$\begin{aligned}
\mathbf{w}^{k+1} &= \arg\min_{\mathbf{w}} \; f(\mathbf{w}) + \frac{\rho}{2}\|A\mathbf{w} + B\mathbf{z}^k - \mathbf{c} + \mathbf{u}^k\|_2^2 \\
\mathbf{z}^{k+1} &= \arg\min_{\mathbf{z}} \; g(\mathbf{z}) + \frac{\rho}{2}\|A\mathbf{w}^{k+1} + B\mathbf{z} - \mathbf{c} + \mathbf{u}^k\|_2^2 \\
\mathbf{u}^{k+1} &= \mathbf{u}^k + (A\mathbf{w}^{k+1} + B\mathbf{z}^{k+1} - \mathbf{c}), \qquad (4)
\end{aligned}$$

where $\mathbf{u} = (\frac{1}{\rho})\mathbf{y}$. For the rest of the paper we shall use this scaled version of ADMM.

2.2 Machine Learning Algorithms

The generic optimization problem that is solved for most supervised learning settings is:
Given, training samples $(\mathbf{x}_i, y_i)_{i=1}^N$, where, $\mathbf{x}_i \in \Re^D$, $y_i \in \left\{ \begin{array}{ll} \{+1,-1\} & ;(\text{Classification}) \\ \Re & ;(\text{Regression}) \end{array} \right.$,
Solve,

$$\min_{\mathbf{w}} \;\; \frac{1}{N}\sum_{i=1}^{N} L(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i) + \lambda R(\mathbf{w}) \quad \equiv \underset{\text{ADMM Form}}{\boxed{\min_{\mathbf{w}} \;\; \frac{1}{N}\sum_{i=1}^{N} L(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i) + \lambda R(\mathbf{z}) \quad s.t. \quad \mathbf{w} = \mathbf{z}}}. \qquad (5)$$

Here, N is the number of training samples, and D is the dimension of each sample. Further, we limit the setting to linear parameterizations, where $\hat{f}_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$. Moreover, $L(\cdot)$ is a convex loss function which measures the discrepancy between the model estimates and their true values/labels, and $R(\cdot)$ is a convex regularizer that penalizes the model complexity for better generalization on unseen future test samples. This formulation in eq. (5) covers a wide-spectrum of machine learning algorithms, as seen in Table 1.

In this toolkit we adopt the ADMM framework to solve this generic ML problem. The main idea for ADMM is to decouple a large problem into smaller sub-problems in $\mathbf{w}$ and $\mathbf{z}$ variables (see ADMM form in eq. (5)). The resulting sub-problems are easier to tackle and can readily be solved in a distributed fashion. The ADMM updates at iteration $k+1$ are:

$$\begin{aligned}
(\mathbf{w}-\text{step}): \quad & \mathbf{w}^{k+1} = \arg\min_{\mathbf{w}} \quad \frac{1}{N}\sum_{i=1}^{N} L(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i) \quad + \quad \frac{\rho}{2}\|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|^2 \qquad (6) \\
(\mathbf{z}-\text{step}): \quad & \mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} \quad \lambda R(\mathbf{z}) \quad + \quad \frac{\rho}{2}\|\mathbf{w}^{k+1} - \mathbf{z} + \mathbf{u}^k\|^2 \\
(\mathbf{u}-\text{step}): \quad & \mathbf{u}^{k+1} = \mathbf{u}^k + \mathbf{w}^{k+1} - \mathbf{z}^{k+1}.
\end{aligned}$$

These updates are guaranteed to converge (Boyd et al., 2011). As shown in (4), the ADMM updates alternate between two smaller sub-problems in $\mathbf{w}$ and $\mathbf{z}$. For the algorithms in Table 1, the $\mathbf{z}$-update has a closed form solution and can be solved using a soft-thresholding operation (which involves $O(D)$ complexity (Boyd et al., 2011)). The bottleneck (due to big-data) is handled in the $\mathbf{w}$-step. However, now the $\mathbf{w}$-update is a conjunction of $L(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$ and a simple ℓ^2-regularizer (rather than a possibly more complicated function $R(\mathbf{z})$). This

Table 1: Machine Learning Algorithms

Methods	**Loss Function** $L(f_{\mathbf{w}}(\mathbf{x}, y))$	**Regularizer** $R(\mathbf{w})$
Classification , $y \in \{-1, +1\}$		Elastic-net :
Logistic	$\frac{1}{N}\sum_i \log(1 + e^{-y_i \mathbf{w}^\top \mathbf{x}_i})$	$\sum_{j=1}^{D} \delta_j \{\alpha \lvert w_j \rvert + (1-\alpha)\frac{w_j^2}{2}\}$
LS-SVM	$\frac{1}{2N}\sum_i (1 - y_i \mathbf{w}^\top \mathbf{x}_i)^2$	with, $\alpha \in [0, 1]$
Squared Hinge	$\frac{1}{2N}\sum_i \max\big(1 - y_i \mathbf{w}^\top \mathbf{x}_i, 0\big)^2$	
Regression , $y \in \Re$		Group :
Linear	$\frac{1}{2N}\sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)^2$	$\sum_{g \in G} \delta_g \lVert \mathbf{w}_g \rVert$
Huber	$\frac{1}{N}\sum_{i=1}^{N} \begin{cases} \frac{1}{2}(y_i - \mathbf{w}^T \mathbf{x}_i)^2 & ;\text{if } \lvert y_i - \mathbf{w}^T \mathbf{x} \rvert \le \mu \\ \mu \lvert y_i - \mathbf{w}^T \mathbf{x} \rvert - \frac{1}{2}\mu^2 & ;\text{otherwise} \end{cases}$	G := Groups
Pseudo-Huber	$\frac{1}{N}\sum_i \sqrt{\mu^2 + (y_i - \mathbf{w}^\top \mathbf{x}_i)^2} - \mu$	

can be easily solved using Newton updates. The bottleneck on constructing the Hessian and gradient is tackled by using the map-reduce paradigm (as introduced in (Chu et al., 2007)), and is implemented in the Spark computing framework. Further implementation details are available in (Dhar et al., 2015).

2.3 The ADMM update steps

2.3.1 Regularization Functions (z-update)

Elastic-Net In this work we introduce a modified form of the elastic regularizer. This form is more generic, and can control the regularization in the bias space i.e. by setting $\delta_{D+1} = 0$. The $\mathbf{z}$-update is given as:

$$\mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} \; \lambda \sum_j \delta_j \left\{ \alpha \, |z_j| + (1-\alpha) \cdot \frac{z_j^2}{2} \right\} + \frac{\rho}{2} \|\mathbf{w}^{k+1} - \mathbf{z} + \mathbf{u}^k\|_2^2 \tag{7}$$

$$\Rightarrow \forall j = 1 \ldots D; \quad z_j^{k+1} = \frac{S_{\kappa_j}(w_j^{k+1} + u_j^k)}{1 + \lambda\delta_j(1-\alpha)/\rho},$$

where $\kappa_j = \lambda\delta_j\alpha/\rho$ and $S_\kappa(t) = \big(1 - \frac{\kappa}{|t|}\big)_+ t$ is the soft-thresholding operator (Boyd et al., 2011). In simple terms this is given as:

$$z_j^{k+1} = \begin{cases} \{\rho(w_j + u_j) - \lambda\delta_j\alpha\}/\{\lambda\delta(1-\alpha) + \rho\} & ;\text{if} \quad \rho(w_j + u_j) \ge \lambda\delta_j\alpha \\ \{\rho(w_j + u_j) + \lambda\delta_j\alpha\}/\{\lambda\delta(1-\alpha) + \rho\} & ;\text{otherwise.} \end{cases} \tag{8}$$

Group-regularizer The $\mathbf{z}$-update is given as:

$$\mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} \ \lambda \sum_{g \in G} \delta_g \|\mathbf{w}_g\| + \frac{\rho}{2} \|\mathbf{w}^{k+1} - \mathbf{z} + \mathbf{u}^k\|_2^2 \tag{9}$$

$$\Rightarrow \mathbf{z}_g^{k+1} = \quad \max\left(\|\mathbf{w}_g + \mathbf{u}_g\| - \frac{\lambda \delta_g}{\rho}, 0 \right) \cdot \frac{(\mathbf{w}_g + \mathbf{u}_g)}{\|\mathbf{w}_g + \mathbf{u}_g\|}.$$

2.3.2 Loss Functions (Classification) w-update

This section discuss the algorithm and the implementation details of the different loss functions.

Logit loss The $\mathbf{w}$-update for the logit loss is

$$\mathbf{w}^{k+1} = \arg\min_{\mathbf{w}} \ \frac{1}{N} \sum_{i=1}^{N} \log(1 + e^{-y_i(\mathbf{w}^T \mathbf{x}_i)}) + \frac{\rho}{2} \|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2. \tag{10}$$

The $\mathbf{w}$-problem is solved through Newton updates, as shown in Algorithm 1.

Algorithm 1: Iterative Algorithm for $\mathbf{w}^{k+1}$ (Logit Loss)

```
Input: $\mathbf{w}^k$, $\mathbf{z}^k$, $\mathbf{u}^k$
Output: $\mathbf{w}^{k+1}$
initialize $\mathbf{s}^{(0)} \leftarrow \mathbf{w}^k$, $j \leftarrow 0$ ;
while not converged until MAX_INNER_ITER do
    $p_i^{(j)} \leftarrow 1/(1 + e^{-\mathbf{x}_i^\top \mathbf{v}^{(j)}})$;
    $P^{(j)} \leftarrow \frac{1}{N}\sum_i H_i + \rho \mathbf{I}_D$;   where $\boxed{H_i = p_i^{(j)}(1 - p_i^{(j)})\mathbf{x}_i\mathbf{x}_i^\top}$ (distributed) ;
    $\mathbf{q}^{(j)} \leftarrow -\frac{1}{N}\sum_i \mathbf{v}_i + \rho(\mathbf{s}^{(j-1)} - \mathbf{z}^k + \mathbf{u}^k)$;   $\boxed{\mathbf{v}_i = y_i(1 - p_i^{(j)})\mathbf{x}_i}$ (distributed);
    $\mathbf{s}^{(j+1)} \leftarrow \mathbf{s}^{(j)} - (P^{(j)})^{-1}\mathbf{q}^{(j)}$;
    $j \leftarrow j + 1$;
return $\mathbf{w}^{k+1} \leftarrow \mathbf{s}^*$;
```

Least Squares (classification) The $\mathbf{w}$-update is given as,

$$\begin{aligned} \mathbf{w}^{k+1} &= \arg\min_{\mathbf{w}} \quad \frac{1}{2N} \sum_{i=1}^{N} (1 - y_i(\mathbf{w}^T \mathbf{x}_i))^2 + \frac{\rho}{2} \|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2 \\ &= \left(\frac{1}{N} \sum_{i=1}^{N} H_i + \rho \mathbf{I}_D \right)^{-1} \left[\frac{1}{N} \sum_{i}^{N} \mathbf{v}_i + \rho(\mathbf{z}^k - \mathbf{u}^k) \right] \\ &= P^{-1}\mathbf{q}, \end{aligned} \tag{11}$$

where $H_i = \mathbf{x}_i \mathbf{x}_i^T$ and $\mathbf{v}_i = \mathbf{x}_i y_i$.

Squared Hinge The $\mathbf{w}$-update is given as,

$$\mathbf{w}^{k+1} = \arg\min_{\mathbf{w}} \quad \frac{1}{2N} \sum_{i=1}^{N} \max\Big(1 - y_i \mathbf{w}^\top \mathbf{x}_i, 0\Big)^2 + \frac{\rho}{2} \|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2. \tag{12}$$

The $\mathbf{w}$-problem is solved through Newton updates, as shown in Algorithm 2.

```
Algorithm 2: Iterative Algorithm for $\mathbf{w}^{k+1}$ (Square Hinge)
  Input: $\mathbf{w}^k$, $\mathbf{z}^k$, $\mathbf{u}^k$
  Output: $\mathbf{w}^{k+1}$
  initialize $\mathbf{s}^{(0)} \leftarrow \mathbf{w}^k$, $j \leftarrow 0$ ;
  while not converged until MAX_INNER_ITER do
      $P^{(j)} \leftarrow \frac{1}{N} \sum_i H_i + \rho \mathbf{I}_D$;
      $\mathbf{q}^{(j)} \leftarrow \frac{1}{N} \sum_i \mathbf{v}_i + \rho(\mathbf{s}^{(j-1)} - \mathbf{z}^k + \mathbf{u}^k)$ ;
      $\mathbf{s}^{(j+1)} \leftarrow \mathbf{s}^{(j)} - (P^{(j)})^{-1} \mathbf{q}^{(j)}$;
      $j \leftarrow j + 1$;
  return $\mathbf{w}^{k+1} \leftarrow \mathbf{s}^*$;
```

Here,

$$H_i = \begin{cases} \mathbf{x}_i \mathbf{x}_i^T & ;\text{if} \quad y_i \mathbf{w}^\top \mathbf{x}_i \leq 1 \\ \mathbf{0}_{D \times D} & ;\text{otherwise} \end{cases}$$

$$\mathbf{v}_i = \begin{cases} -y_i(1 - y_i \mathbf{w}^\top \mathbf{x}_i)\mathbf{x}_i & ;\text{if} \quad y_i \mathbf{w}^\top \mathbf{x}_i \leq 1 \\ \mathbf{0}_{D \times 1} & ;\text{otherwise.} \end{cases}$$

2.3.3 Loss Functions (Regression)

Least Squares (regression) The $\mathbf{w}$-update is given as

$$\begin{aligned} \mathbf{w}^{k+1} &= \arg\min_{\mathbf{w}} \quad \frac{1}{2N} \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \frac{\rho}{2} \|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2 \qquad (13) \\ &= \left(\frac{1}{N} \sum_{i=1}^{N} H_i + \rho \mathbf{I}_D \right)^{-1} \left[\frac{1}{N} \sum_{i}^{N} \mathbf{v}_i + \rho(\mathbf{z}^k - \mathbf{u}^k) \right] \\ &= P^{-1} \mathbf{q}, \end{aligned}$$

where $H_i = \mathbf{x}_i \mathbf{x}_i^T$ and $\mathbf{v}_i = \mathbf{x}_i y_i$.

Huber The $\mathbf{w}$-update is given as,

$$\mathbf{w}^{k+1} = \underset{\mathbf{w}}{\arg\min} \quad \frac{1}{N}\sum_{i=1}^{N} \begin{cases} \frac{1}{2}(y_i - \mathbf{w}^T\mathbf{x}_i) & ;\text{if } |y_i - \mathbf{w}^T\mathbf{x}| \leq \mu \\ \mu|y_i - \mathbf{w}^T\mathbf{x}| - \frac{1}{2}\mu^2 & ;\text{otherwise.} \end{cases} + \frac{\rho}{2}\|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2 \tag{14}$$

The $\mathbf{w}$-problem is solved through Newton updates, as shown in Algorithm 3.

Algorithm 3: Iterative Algorithm for $\mathbf{w}^{k+1}$ (Huber)

```
Input: $\mathbf{w}^k$, $\mathbf{z}^k$, $\mathbf{u}^k, \mu$
Output: $\mathbf{w}^{k+1}$
squa initialize $\mathbf{s}^{(0)} \leftarrow \mathbf{w}^k, j \leftarrow 0, \hat{\mu}_j \leftarrow 1$ ;
while not converged until MAX_INNER_ITER do
    $P^{(j)} \leftarrow \frac{1}{N}\sum_i H_i + \rho\mathbf{I}_D$ where $H_i = \mathbb{1}(|y_i - \mathbf{w}^T\mathbf{x}_i| \leq \mu)(\mathbb{1}(\cdot) :=$ Indicator Function);
    $\mathbf{q}^{(j)} \leftarrow \frac{1}{N}\sum_i \mathbf{v}_i + \rho(\mathbf{s}^{(j-1)} - \mathbf{z}^k + \mathbf{u}^k)$ where $\mathbf{v}_i = -\max(-\mu, \min(\mu, y_i - \mathbf{w}^T\mathbf{x}_i))$ ;
    $\mathbf{s}^{(j+1)} \leftarrow \mathbf{s}^{(j)} - (P^{(j)})^{-1}\mathbf{q}^{(j)}$;
    if $\hat{\mu}_j \neq \mu$ then
        $\hat{\mu}_{j+1} \leftarrow \hat{\mu}_j/2$;
        $j \leftarrow j + 1$;
return $\mathbf{w}^{k+1} \leftarrow \mathbf{s}^*$;
```

Pseudo-Huber The $\mathbf{w}$-update is given as,

$$\mathbf{w}^{k+1} = \underset{\mathbf{w}}{\arg\min} \quad \frac{1}{N}\sum_{i=1}^{N} \sqrt{\mu^2 + (y_i - (\mathbf{w}^T\mathbf{x}_i))^2} - \mu + \quad \frac{\rho}{2}\|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2 \tag{15}$$

The $\mathbf{w}$-problem is solved through Newton updates, as shown in Algorithm 4.

Algorithm 4: Iterative Algorithm for $\mathbf{w}^{k+1}$ (Pseudo-Huber)

```
Input: $\mathbf{w}^k$, $\mathbf{z}^k$, $\mathbf{u}^k, \mu$
Output: $\mathbf{w}^{k+1}$
initialize $\mathbf{s}^{(0)} \leftarrow \mathbf{w}^k, j \leftarrow 0, \hat{\mu}_j \leftarrow 1$ ;
while not converged until MAX_INNER_ITER do
    $P^{(j)} \leftarrow \frac{1}{N}\sum_i H_i + \rho\mathbf{I}_D$ where $H_i = \{\mu^2/[\mu^2 + (y_i - \mathbf{w}^T\mathbf{x}_i)^2]^{3/2}\} \cdot \mathbf{x}_i\mathbf{x}_i^T$;
    $\mathbf{q}^{(j)} \leftarrow \frac{1}{N}\sum_i \mathbf{v}_i + \rho(\mathbf{s}^{(j-1)} - \mathbf{z}^k + \mathbf{u}^k)$ ;
            where, $\mathbf{v}_i = -\{(y_i - \mathbf{w}^T\mathbf{x}_i)/[\mu^2 + (y_i - \mathbf{w}^T\mathbf{x}_i)^2]^{1/2}\}\mathbf{x}_i$;
    $\mathbf{s}^{(j+1)} \leftarrow \mathbf{s}^{(j)} - (P^{(j)})^{-1}\mathbf{q}^{(j)}$;
    if $\hat{\mu}_j \neq \mu$ then
        $\hat{\mu}_{j+1} \leftarrow \hat{\mu}_j/2$;
        $j \leftarrow j + 1$;
return $\mathbf{w}^{k+1} \leftarrow \mathbf{s}^*$;
```

As shown above, all the $\mathbf{w}$ sub-problems involve solving a linear system $P\mathbf{w} = \mathbf{q}$, which involves worst-case $O(D^3)$ complexity. However. the bottle-neck for big-data is the construction of $P, \mathbf{q}$. In this work, we perform this by distributing the computation of $H_i, \mathbf{v}_i$'s using map-reduce.

Remarks:

1. For Least-Square loss the **w**-update provides a closed form solution and can be obtained in a single step. In fact, this is an advantage for all quadratic loss functions as previously noted in (Boyd et al., 2011).
2. For the Huber/Pseudo-Huber loss functions simple application of Newton updates may lead to divergence of the solution. This however is a well-studied phenomenon. There is a huge amount of research on the convergence of ℓ^2-regularized Huber loss. In fact, we adapt the algorithm originally introduced by Huber and solve the current sub-problem. This is presented in Algorithms 3 and 4.

3. ADMML Toolkit Usage

Next we discuss the **A**lternating **D**irection method of **M**ultipliers based **M**achine **L**earning (ADMML) toolkit usage.

3.1 Installation and Configuration

The ADMML toolkit is available at `https://github.com/DL-Benchmarks/ADMML`. The details of the tools installation and configuration are provided next.

3.1.1 Local or Cluster

Dependencies

Apache Spark: Requires Apache Spark 2.0.2 or higher. (tested on version 2.0.2)

Numpy. (tested on version 1.10.4)

setuptools. (tested on version 34.4.1)

Installation

1. Clone the repository (`git clone https://github.com/DL-Benchmarks/ADMML.git`)
2. Navigate to the ADMML folder, i.e. the folder which contains setup.py
3. Build the .egg file (`python setup.py bdist_egg`)
    - For Python 2.7, the output file is named `admml-0.1-py2.7.egg`, and this is the name we use throughout the documentation. If you are using a different Python version, you'll need to modify this filename accordingly.

4. Launch Pyspark and distribute the .egg file to all the cluster nodes for the pyspark context:
   `sc.addPyFile('<ADMML folder absolute path>/dist/admml-0.1-py2.7.egg')`

3.1.2 Amazon Web Services

Amazon Web Services (AWS) (`https://aws.amazon.com`) is one of the most commonly used cloud computing services. The ADMML toolkit can be easily configured for the AWS platform as discussed next,

1. Create an EMR cluster from the AWS console (`https://console.aws.amazon.com/elasticmapreduce`).
   - For detailed instructions, see “Create a Cluster With Spark” (`http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-launch.html`).
2. Once logged in to the instance, follow the steps in Section 3.1.1.
3. Set Spark related environment variables on AWS:

```
export SPARK_HOME=/usr/lib/spark
export HADOOP_CONF_DIR=/etc/hadoop/conf
```

3.2 APIs

The code is organized as shown in Figure 2. The current code allows access to the entire spectrum of development to Python users.

3.2.1 admml.admml Module

ADMM CORE ALGORITHM

`admml.admml.ADMM`(*trndata*, *algoParam*)

This is the main engine that implements the ADMM steps.

Parameters

trndata : RDD

RDD with each entry as numpy array [y, **x**]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

z : numpy.ndarray

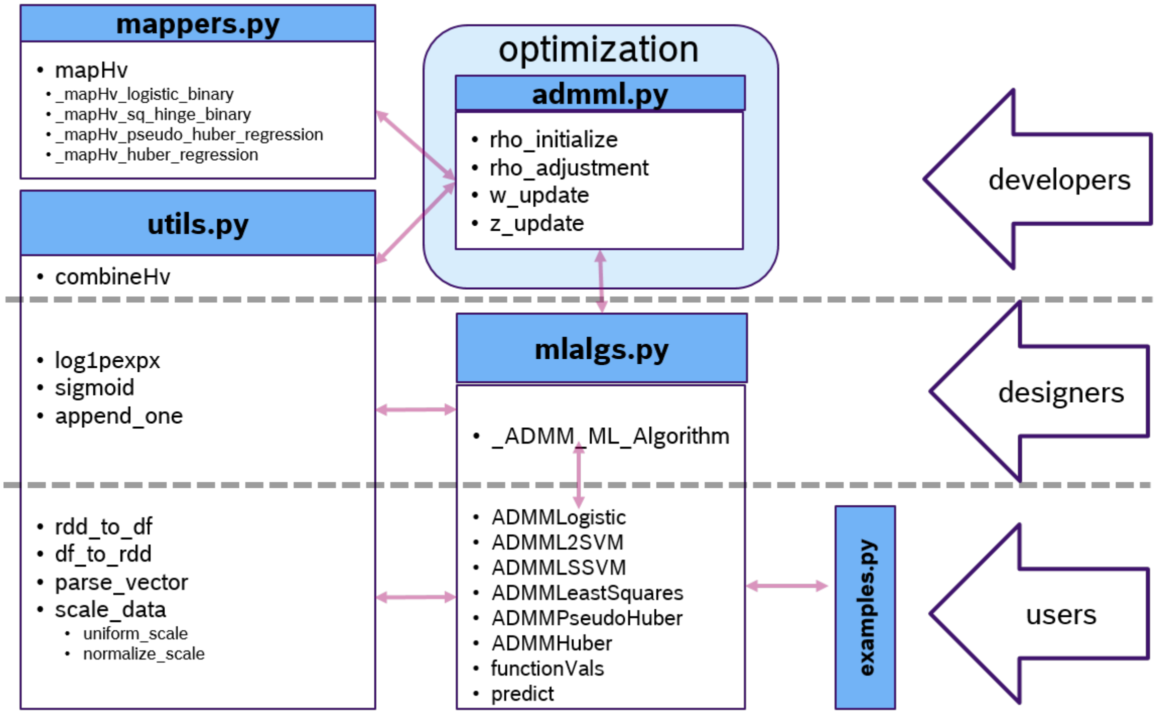

Figure 2: ADMML tool code organization.

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

3.2.2 `admml.examples` Module

Basic Examples

`admml.examples.example`(*trndata*, *tstdata*, *example=1*)

This module shows three basic examples to use the tool.

Parameters

trndata : RDD

This is the training data on which the models are trained. This is an RDD of numbers only created by reading a comma separated file through `sc.textFile()`. The 1st field is y-labels i.e. `trndata = sc.textFile('<URI>\trndata.csv')`

tstdata : RDD

This is the test data on which the models are scored. This is an RDD of numbers only created by reading a comma separated file through `sc.textFile()`. The 1st field is y-labels. i.e. `trndata = sc.textFile('<URI>\tstdata.csv')`

example : int {1,2,3}

1 = L2-Linear Regression, 2 = L2-Logistic Regression, 3 = Group - Linear Regression

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

3.2.3 `admml.mappers` Module

This module contains the mappers to compute the Hessian/gradients in a distributed fashion

`admml.mappers.mapHv`(*algoParam*, *vector*, *w*, *mu*)

Map function to compute the appropriate Hessian/Gradient based on `algoParam['LOSS']` and `algoParam['PROBLEM']`.

Parameters

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as: `algoParam = setDefaultAlgoParams()`. Here `algoParam['LOSS']` and `algoParam['PROBLEM']` should be specified

vector : numpy.ndarray

Data sample (a row of data matrix) which contributes to Hessian/Gradient

w : numpy.ndarray

weight vector at the current internal newtonian iteration.

mu : float

The μ value only for Huber/Pseudo-Huber Loss. Else its None.

Returns

H_i : numpy.ndarray

The i-th samples contribution to the Hessian

v_i : numpy.ndarray

The i-th samples contribution to the gradient

3.2.4 `admml.mlalgs` Module

ML Algorithms

`admml.mlalgs.ADMMHuber`(*trndata*, *algoParam*)

Solves the (Elastic-Net + Group) Huber Regression problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.ADMML2SVM`(*trndata*, *algoParam*)

Solves the (Elastic-Net + Group) L2-SVM classification problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration

- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.ADMMLSSVM(`*trndata*, *algoParam*`)`

Solves the (Elastic-Net + Group) LS-SVM classification problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.ADMMLeastSquares(`*trndata*, *algoParam*`)`

Solves the (Elastic-Net + Group) Linear Regression problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.ADMMLogistic`(*trndata*, *algoParam*)

Solves the (Elastic-Net + Group) Logistic Regression problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.ADMMPseudoHuber`(*trndata*, *algoParam*)

Solves the (Elastic-Net + Group) Pseudo-Huber Regression problem

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

output : dict

Dictionary of the algorithm outputs

- `W_k`: each row correspond to w vectors at k_th ADMM iteration
- `norm_pri`: ℓ^2-norm of primal residual
- `norm_dual`: ℓ^2-norm of dual residual
- `rel_norm_pri`: relative norm of primal residual
- `rel_norm_dual`: relative norm of dual residual

fval : numpy.float64

Function Value at the optimal solution

`admml.mlalgs.functionVals`(*trndata*, *w*, *algoParam*)

Provides function values for the supported algorithms at specified w-values.

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

fval :numpy.float64

Function Value at the specified w-value

`admml.mlalgs.predict`(*tstdata*, *w*, *algoParam*)

Predict on Future Test Data

Parameters

tstdata : RDD

RDD with each entry as numpy array [y,x]

w : numpy.ndarray

Weight vector [w,b] (bias term appended to end)

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as:
`algoParam = setDefaultAlgoParams()`

Returns

error_rate : numpy.float64

Misclassification Error Rate (binary classification) Mean Square Error (regression)

`admml.mlalgs.setDefaultAlgoParams`()

This module creates a default template of the algorithm parameters.

Parameters

algoParam : dict

a dictionary with keys and values (default values are in bold):

- **PROBLEM: binary**,regression
- **REG: elasticnet**, group
- **LOSS: logistic** (classification), sq_hinge (classification), leastsq (classification/regression), huber (regression), pseudo_huber (regression)
- **LAMBDA: 1.0** , (Regularization parameter)
- **ALPHA: 0.0**, (Elastic-Net parameter)
- **MU: 0.1**, ((Pseudo)-Huber Threshold parameter)
- **SCALE: Uniform**, Normalize, None
- **RHO: 1.0**, (ADMM augmented lagrangian penalty term)
- **RHO_INITIAL: 0**, 1 (0 = Constant (RHO = LAMBDA) , 1 = Goldstein (pending))
- **RHO_ADAPTIVE_FLAG: False**, True
- **MAX_ITER: 100**, (Max. iterations for ADMM updates)
- **PRIM_TOL: 1e-4**, (Error tolerance of relative primal residual)
- **DUAL_TOL: 1e-4**, (Error tolerance of relative dual residual)
- **MAX_INNER_ITER: 10**, (Max. iteration for internal newton updates)
- **INNER_TOL: 1e-6**, (Relative error tolerance of internal iterations)
- **VERBOSE: 0** (no print), 1 (print)

3.2.5 admml.utils Module

This module contains various helper functions used in ADMM

admml.utils.append_one(*vector*)

Appends a column of ones to the end of the vector. This takes care of bias term.

Parameters

vector : numpy.ndarray

input

Returns

numpy.ndarray

the input vector appended with one – [input,1]

`admml.utils.combineHv`(*x1*, *x2*)

Combine/Reduce function for the mappers

Parameters

x1 : list

element1

x2 : list

element2

Returns

tuple

`(x1[0]+x2[0], x1[1]+x2[1])`

`admml.utils.df_to_rdd`(*line*)

Converts a pyspark `Row` element to numpy array

Parameters

line : pyspark.sql.types.Row

A line dataframe.rdd

Note: The dataframe should contain only numbers. Also the method can be invoked on `dataframe.rdd` as `dataFrame` objects have no attribute map

---

Returns

numpy.ndarray

A numpy array representation of the data contained in `line`

`admml.utils.log1pexpx`(*x*)

Transformation $\mathbf{x} \mapsto \log(\mathbf{1}+\mathbf{x})$

Parameters

x : numpy.ndarray

input

Returns

numpy.ndarray

$\log(\mathbf{1}+\mathbf{x})$

`admml.utils.normalize_scale`(*vector*, *col_mean*, *col_var*)

Scale the features to a normal distribution

Parameters

vector : numpy.ndarray

A vector (numpy array) representation of a row of the data

col_mean : numpy.ndarray

Mean over all the columns of data

col_var : numpy.ndarray

Variance over all the columns of data

Returns

numpy.ndarray

Normally scaled representation of the vector.

`admml.utils.parse_vector`(*line*)

Read a line from a CSV file into a numpy array

Parameters

line : str

A line from a CSV file.

Note: The CSV file should contain only numbers

Returns

numpy.ndarray

A numpy array representation of the data contained in `line`

`admml.utils.rdd_to_df`(*line*)

Converts a numpy array to Row (pyspark.sql.types.Row)

Parameters

line : numpy.ndarray

A numpy array where the first element is the label

Returns

pyspark.sql.types.Row

A `Row` with label and features attributes compatible with pyspark.ml libraries `admml.utils.scale_data`(*trndata*, *tstdata*, *algoParam*)

Scales the data as specified by the `algoParam['SCALE']`.

The `tstdata` is scaled in the same range as `trndata`.

Parameters

trndata : RDD

RDD with each entry as numpy array [y,x]

tstdata : RDD

RDD with each entry as numpy array [y,x]

algoParam : dict

This contains the algorithm parameters for ML/ADMM algorithm. The default can be set as: `algoParam = setDefaultAlgoParams()` Here `algoParam['SCALE']` is either `'Uniform'` or `'Normalize'`

Returns

trndata : RDD

the scaled training data

tstdata : RDD

the scaled test data

`admml.utils.sigmoid`(x)

Transformation to $\mathbf{x} \mapsto sigmoid(\mathbf{x})$

Parameters

x : numpy.ndarray

input

Returns

numpy.ndarray

$sigmoid(\mathbf{x})$

`admml.utils.uniform_scale`(*vector*, *col_min*, *col_max*)

Scale the features uniformly in the range of [0,1]

Parameters

vector : numpy.ndarray

A vector (numpy array) representation of a row of the data

col_min : numpy.ndarray

Min values over all the columns of data

col_max : numpy.ndarray

Max values over all the columns of data

Returns

numpy.ndarray

Uniformly scaled representation of the vector.

3.3 Examples

Here we provide some simple examples to use the toolkit.

Elastic-Logistic Regression Our first example involves training an elastic-net logistic regression model.

```
# Distribute the ADMML package in all Spark Nodes
sc.addPyFile('<local dir>/ADMML/dist/admml-0.1-py2.7.egg')

# Import the ADMML libraries
import admml.utils as utils
import admml.mlalgs as ml
import numpy as np

# Load Training data in .csv only
trndata = sc.textFile('<URI_of_a_CSV_data>')

# Convert into RDD of numpy array
trndata = trndata.map(utils.parse_vector)

#Set default algorithm parameters
# {'ALPHA': 0.0,
# 'D': 0,
# 'DUAL_TOL': 0.0001,
# 'EIG_VALS_FLAG': 0,
# 'INNER_TOL': 1e-06,
# 'K': 1,
# 'LAMBDA': 1.0,
# 'LOSS': 'logistic',
# 'MAX_INNER_ITER': 10,
# 'MAX_ITER': 100,
# 'MU': 0.1,
# 'MU_MAX': 1.0,
# 'N': 0,
# 'PRIM_TOL': 0.0001,
# 'PROBLEM': 'binary',
# 'REG': 'elasticnet',
# 'RHO': 1.0,
# 'RHO_ADAPTIVE_FLAG': False,
# 'RHO_INITIAL': 0,
# 'SCALE': 'Uniform',
# 'VERBOSE': 0}
algoParam = ml.setDefaultAlgoParams()


# Change the default parameters.
algoParam['LAMBDA'] = 0.1   # Set Regularization Parameter
algoParam['ALPHA'] = 0.25       # Set the Elastic-Net parameter

#Train the algorithm
w, output, fval = ml.ADMMLogistic(trndata,algoParam)

# Predict (Training Error:: For binary 0/1 error.)
train_err = ml.predict(trndata, w, algoParam)
```

Group Regularized-Linear Regression Our next example shows how to train a group-regularized Least Square regression.

```
# Distribute the ADMML package in all Spark Nodes
sc.addPyFile('<local dir>/ADMML/dist/admml-0.1-py2.7.egg')

# Import the ADMML libraries
import admml.utils as utils
import admml.mlalgs as ml
import numpy as np

# Load Training data in .csv only
trndata = sc.textFile('<URI_of_a_CSV_Traindata>')
tstdata = sc.textFile('<URI_of_a_CSV_Testdata>')

# Convert into RDD of numpy array
trndata = trndata.map(utils.parse_vector)
tstdata = tstdata.map(utils.parse_vector)

# set group regularizer for linear regression
algoParam = ml.setDefaultAlgoParams()
algoParam['PROBLEM'] = 'regression'
algoParam['REG'] = 'group'
```

```
# Change algorithm parameters
algoParam['LAMBDA'] = 0.1    # Set Regularization Parameter
#set group weights
algoParam['G'] = np.array([1.0, 1.0, 1.0, 0]/np.sqrt(3))
#set group membership
algoParam['DELTA'] = np.array([0, 1, 2, 0, 1, 2, 0, 1, 2, 3])


# SET OPTIMIZATION STOPPING CRITERIA (OPTIONAL)
algoParam['DUAL_TOL'] = 1e-3
algoParam['PRIM_TOL'] = 1e-3
algoParam['MAX_ITER'] = 10

algoParam['VERBOSE'] = 1        # PRINT OUT


#Scale the data in [0,1] Uniformly
trn_sc, tst_sc = utils.scale_data(trndata,tstdata,algoParam)

#Train the algorithm
w, output, fval = ml.ADMMLeastSquares(trn_sc,algoParam)

# Predict (Test/Training Error:: For mean squared error)
test_err = ml.predict(tst_sc, w, algoParam)
train_err = ml.predict(trn_sc, w, algoParam)
```

L2-Huber regression Our final example shows how to train an ℓ^2-regularized Huber regression by directly calling the ADMM API.

```
# Distribute the ADMML package in all Spark Nodes
sc.addPyFile('<local dir>/ADMML/dist/admml-0.1-py2.7.egg')

# Import the ADMML libraries
import admml.utils as utils
import admml.mlalgs as ml
import admml.admml as admml
import numpy as np

# Load Training data in .csv only
trndata = sc.textFile('<URI_of_a_CSV_Traindata>')

# Convert into RDD of numpy array and append ones
trndata = trndata.map(utils.parse_vector)
trndata = trndata.map(utils.append_one)
```

```
algoParam = ml.setDefaultAlgoParams()
algoParam['N'] = trndata.count()
algoParam['D'] = trndata.first().size-1  # Data format: [y,X,1]

# set L2 regularizer
algoParam['PROBLEM'] = 'regression'
algoParam['LOSS'] = 'huber'
algoParam['MU'] = 0.05 # Huber parameter
algoParam['DELTA'] = np.append(np.ones(algoParam['D']), 0.)

algoParam['LAMBDA'] = 0.1 # REGULARIZATION PARAMETER
algoParam['ALPHA'] = 0.0 # L2

w, output = admml.ADMM(trndata, algoParam)
```

Additional examples are also available in the **examples.py** file. Experimental results using this toolkit are available at: (Dhar et al., 2015; Dhar and Shah, 2017).

4. Future Directions

Future versions of this toolkit shall include,

- Initial selection of ρ and its adaptive update for improved convergence.
- Consensus ADMM for better coverage of discontinuous loss functions like SVM, α-trimmed functions, etc.
- Multiclass algorithms like multinomial logistic regression, multi-class SVM, etc.

Acknowledgement

We thank Dr. Congrui Yi for the work done during his 2014 internship at Bosch (Dhar et al., 2015) and Dr. Goutham Kamath for the work during his 2016 internship (Kamath, 2016). Both of these works were precursors to this toolkit. Finally we thank Dr. Naveen Ramakrishnan for his inputs on the algorithm designs.

References

Dimitri P Bertsekas. *Nonlinear Programming*. Athena scientific Belmont, 1999.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2): 285–296, 2010.

Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

Sauptik Dhar and Mohak Shah. ADMM based scalable machine learning on apache spark. `https://spark-summit.org/2017/events/admm-based-scalable-machine-learning-on-apache-spark/`, 2017. [Online; accessed 02-July-2017].

Sauptik Dhar, Congrui Yi, Naveen Ramakrishnan, and Mohak Shah. ADMM based scalable machine learning on Spark. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1174–1182. IEEE, 2015.

Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1):17–40, 1976.

Roland Glowinski and A Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 9(R2):41–76, 1975.

Apache Hadoop. `http://hadoop.apache.org/`. [Online; accessed 02-July-2017].

Wilmer Henao. *An L-BFGS-B-NS optimizer for non-smooth functions.* PhD thesis, Masters thesis, Courant Institute of Mathematical Science, New York University, 2014.

Goutham Kamath. Scalable machine learning on spark for multiclass problems. Baylearn 2016 `https://www.researchgate.net/publication/309858130_Scalable_Machine_Learning_on_Spark_for_multiclass_problems`, 2016. [Online; accessed 02-July-2017].

LinkedIn. Photon machine learning (photon ML). `https://github.com/linkedin/photon-ml`, 2016. [Online; accessed 02-July-2017].

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

Apache Spark. `https://spark.apache.org/`. [Online; accessed 02-July-2017].