LIBSOL: A Library for Scalable Online Learning Algorithms

July 26, 2016

Abstract

LIBSOL is an open-source library for scalable online learning with high-dimensional data. The library provides a family of regular and sparse online learning algorithms for large-scale binary and multi-class classification tasks with high efficiency, scalability, portability, and extensibility. We provide easy-to-use command-line tools, python wrappers and library calls for users and developers, and comprehensive documents for both beginners and advanced users. LIBSOL is not only a machine learning toolbox, but also a comprehensive experimental platform for online learning research. Experiments demonstrate that LIBSOL is highly efficient and scalable for large-scale learning with high-dimensional data.

Contents

1 Introduction

In many big data applications, data is large not only in sample size, but also in feature/dimension size, e.g., web-scale text classification with millions of dimensions. Traditional batch learning algorithms fall short in low efficiency and poor scalability, e.g., high memory consumption and expensive re-training cost for new training data. Online learning represents a family of efficient and scalable algorithms that sequentially learn one example at a time. Some existing toolbox, e.g., LIBOL [9], allows researchers in academia to benchmark different online learning algorithms, but it was not designed for practical developers to tackle online learning with large-scale high-dimensional data in industry.

In this work, we develop LIBSOL as an easy-to-use scalable online learning toolbox for large-scale binary and multi-class classification tasks. It includes a family of ordinary and sparse online learning algorithms, and is highly efficient and scalable for processing high-dimensional data by using (i) parallel threads for both loading and learning the data, and (ii) specially designed data structure for high-dimensional data. The library is implemented in standard C++ with the cross platform ability and there is no dependency on other libraries. To facilitate developing new algorithms, the library is carefully designed and documented with high extensibility. We also provide python wrappers to facilitate experiments and library calls for advanced users. LIBSOL is available at `http://libsol.stevenhoi.org`.

1.1 Online Learning For Binary Linear Classification

Online learning operates sequentially to process one example at a time. Without loss of generality, we first investigate the problem of online feature selection for binary classification tasks. We will extend the solution to multi-class settings in the next section. Consider $\{(\mathbf{x}_t, y_t)|t \in [1, T]\}$ be a sequence of training data examples, where $\mathbf{x}_t \in \mathbb{R}^d$ is a d-dimensional vector, $y_t \in \{+1, -1\}$. As Algorithm 1 shows, at each time step t, the learner receives an incoming example $\mathbf{x}_t$ and then predicts its class label $\hat{y}_t$:

$$\hat{y}_t = sgn(\mathbf{w}_t \cdot \mathbf{x})$$

Afterward, the true label y_t is revealed and the learner suffers a loss $l_t(y_t, \hat{y}_t)$, For example, the hinge loss and logistic loss are commonly used for binary classification:

$$l_t(y_t, \hat{y}_t) = \begin{cases} max(0, 1 - y_t \cdot \hat{y}_t) & \text{Hinge loss} \\ \log\left(1 + e^{-y_t \cdot \hat{y}_t}\right) & \text{Logistic Loss} \end{cases}$$

In real applications, the data dimension d can be very large. To learn a compact model with only meaningful features, many works have been conducted as called sparse online learning. The learner suffers a regularization term to induce sparsity for the learned model $\mathbf{w}$, which is $L1$ in most cases.

$$l_t(y_t, \hat{y}_t) = \begin{cases} max(0, 1 - y_t \cdot \hat{y}_t) + \lambda \|\mathbf{w}_t\|_1 & \text{Hinge loss} \\ \log\left(1 + e^{-y_t \cdot \hat{y}_t}\right) + \lambda \|\mathbf{w}_t\|_1 & \text{Logistic Loss} \end{cases}$$

At the end of each learning step, the learner decides when and how to update the model. The update function is often dependent on gradient of the weights $\mathbf{w}$ with respect to loss. $\mathbf{g}_t = \frac{\partial l_t}{\partial \mathbf{w}_t}$.

Algorithm 1: LIBSOL: Online Learning Framework for Linear Classification

```
Initialize: w_1 = 0;
for t in {1,..., T} do
    Receive x_t ∈ R^d, predict ŷ_t, receive true label y_t;
    Suffer loss l_t(y_t, ŷ_t);
    if l_t(y_t, ŷ_t) then
        w_{t+1} ← update(w_t);
    end
end
```

1.2 Online Learning For Multi-Class Linear Classification

In the multi-class setting, each training example is associated with a label $y \in \{0, 1, \ldots, C-1\}$ for C classes. We adopt the one-vs-the-rest strategy to extend the binary online learning algorithm to the multi-class setting. We introduce a new label-dependent feature,

$$\psi(\mathbf{x}, i) = [\mathbf{0}^T, \ldots, \mathbf{x}^T, \ldots, \mathbf{0}^T]^T,$$

where only the i-th position $\psi(\mathbf{x}, i)$ is $\mathbf{x}$ and the others are $\mathbf{0}$, $(\mathbf{0}, \mathbf{x} \in \mathbb{R}^d)$. At each step, the classifier receives a new example $\mathbf{x}_t$ and predicts the label

$$\hat{y_t} = \arg\max_{i=0}^{C-1} \mathbf{w}_t \cdot \psi(\mathbf{x}, i), \quad \mathbf{w}_t \in \mathbb{R}^{d*C}$$

The loss function is as follows:

$$l_t(y_t, \hat{y}_t) = \begin{cases} max(0, 1 - \mathbf{w}_t \cdot \Delta\psi_t) & \text{Hinge loss} \\ \log\left(1 + e^{-\mathbf{w}_t \cdot \Delta\psi_t}\right) & \text{Logistic Loss,} \end{cases}$$

where $\Delta\psi_t$ is dependent on the multi-class updating strategy.

For max-score multi-class update,

$$\Delta\psi_t = \psi(\mathbf{x}_t, y_t) - \psi(\mathbf{x}_t, \arg\max_{i=0, i \neq y_t}^{C-1} \mathbf{w}_t \cdot \psi(\mathbf{x}, i)) \tag{1}$$

For uniform multi-class update, let

$$E_t \quad = \quad \{i \neq y_t : \boldsymbol{\mu}_t \cdot \psi(\mathbf{x}_t, i) \geq \boldsymbol{\mu}_t \cdot \psi(\mathbf{x}_t, y_t)\}.$$

We have

$$\Delta\psi_t = \Sigma_{i=1}^{k} \alpha_{t,i} \psi(\mathbf{x}_t, i), \quad \alpha_{t,i} = \begin{cases} -1/|E_t| & i \in E_t \\ 1 & \text{if} \quad i = y_t \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

1.3 Summary of Main Algorithms

The goal of our work is to implement most state-of-the-art online learning algorithms to facilitate research and application purposes on the real world large-scale high dimensional data. Especially, we include sparse online learning algorithms which can effectively learn important features from the high dimensional real world data [10]. We provide algorithms for both binary and multi-class problems. These algorithms can also be classified into first order algorithms [12] and second order algorithms [4] from the model's perspective. The implemented algorithms are listed in table 1.

Type	Methodology	Algorithm	Description
Online Learning	First Order	Perceptron [11]	The Perceptron Algorithm
		OGD [13]	Online Gradient Descent
		PA [2]	Passive Aggressive Algorithms
		ALMA [8]	Approximate Large Margin Algorithm
		RDA [12]	Regularized Dual Averaging
	Second Order	SOP [1]	Second-Order Perceptron
		CW [5]	Confidence Weighted Learning
		ECCW [3]	Exactly Convex Confidence Weighted Learning
		AROW [4]	Adaptive Regularized Online Learning
		Ada-FOBOS [6]	Adaptive Gradient Descent
		Ada-RDA [6]	Adaptive Regularized Dual Averaging
Sparse Online Learning	First Order	STG [10]	Sparse Online Learning via Truncated Gradient
		FOBOS-L1 [7]	$l1$ Regularized Forward Backward Splitting
		RDA-L1 [12]	Mixed $l1/l_2^2$ Regularized Dual Averaging
		ERDA-L1 [12]	Enhanced $l1/l_2^2$ Regularized Dual Averaging
	Second Order	Ada-FOBOS-L1 [6]	Ada-FOBOS with $l1$ regularization
		Ada-RDA-L1 [6]	Ada-RDA with $l1$ regularization

Table 1: Summary of the implemented online learning algorithms in LIBSOL.

1.4 Main Features

The whole package is designed for high efficiency, scalability, portability, and extensibility.

- Efficiency: it is implemented in C++ and optimized to reduce time and memory cost.
- Scalability: Data samples are stored in a sparse structure. All operations are optimized around the sparse data structure.
- Portability: All the codes follow the C++11 standard, and there is no dependency on external libraries. We use "cmake" to organize the project so that users on different platforms can build the library easily. LIBSOL thus can run on almost every platform.
- Extensibility: (i) the library is written in a modular way, including **PARIO**(for PARallel IO), **Loss**, and **Model**. User can extend it by inheriting the base classes of these modules and implementing the corresponding interfaces; (ii) We try to relieve the pain of coding in C++ so that users can implement algorithms in a "Matlab" style. The code snippet in Figure 1 shows an example to implement the core function of the "*ALMA*" algorithm.

```
1  Vector<float> w; //weight vector
2  void Iterate(SVector<float> x, int y) {
3      float predict = dotmul(w, x);  //predict label with dot product
4      float loss = max(0, 1 - y * predict); //hinge loss
5      if (loss > 0) {
6          //non-zero loss, update the model
7          w = w + eta * y * x; //eta is the learning rate
8          float w_norm = Norm2(w); //calculate the L2 norm of w
9          if (w_norm > 1) w /= w_norm;
10     }
11 }
```

Code 1: Compact implementation of the core function of "*ALMA*" algorithm in LIBSOL.

```
Vector<float> w; //weight vector
void Iterate(SVector<float> x, int y) {
    //predict label with dot product
    float predict = 0;
    for (int i = 0; i < x.size(); ++i){
        predict += w[x.index(i)] * x.value(i);
    }
    float loss = max(0, 1 - y * predict); //hinge loss
    if (loss > 0) {
        //non-zero loss, update the model
        //eta is the learning rate
        for (int i = 0; i < x.size(); ++i) {
            w[x.index(i)] += eta * y * x.value(i);
        }
        //calculate the L2 norm of w
        float w_norm = 0;
        for (int i = 0; i < x.size(); ++i) {
            w_norm += x.value(i) * x.value(i);
        }
        w_norm = sqrtf(w_norm);
        if (w_norm > 1) {
            for (int i = 0; i < w.dim(); ++i){
                w[i] /= w_norm;
            }
        }
    }
}
```

Code 2: Traditional "C++"‘ style counterpart implementation of the above algorithm.

It's obvious that the traditional implementation is tedious. It's also much harder for readers to understand the code. What's more, users have to pay much more effort with much larger risk to make mistakes when coding in the traditional implementation.

2 Installation

LIBSOL features a very simple installation procedure. The project is managed by **CMake**. There exists a **CMakeLists.txt** in the root directory of LIBSOL. Note that all the following are tested on "*CMake 2.8*". Lower versions of cmake may work, but are not ensured.

2.1 Getting the code

The latest version of LIBSOL is always available via "github" by invoking one of the following:

```
# For the traditional ssh-based Git interaction:
$ git clone git://github.com/LIBOL/LIBSOL.git

# For HTTP-based Git interaction
$ git clone https://github.com/LIBOL/LIBSOL.git
```

2.2 Installation on Linux/Unix/MacOS

The following steps have been tested for Ubuntu 14.04, Centos 6.6 (with "devtoolset-2" installed for the latter one), and OS X 10.10, but should work with other Unix like distributions as well, as long as it provides a "C++11" compiler.

2.2.1 Required Packages

- g++($\geq$ 4.8.2) or clang++($\geq$ 3.3);
- CMake 2.8 or higher;
- Python 2.7 (required for python wrappers, otherwise optional)

2.2.2 Build from source

1. Navigate to the root directory of LIBSOL and create a temporary directory to put the generated project files, as well the object files and output binaries.

   ```
   $ cd LIBSOL && mkdir build && cd build
   ```

2. Generate and build the project files.

   ```
   $ cmake ..
   $ make -j
   $ make install
   ```

3. For Xcode users, the command is:

   ```
   $ cmake -G"Xcode" ..
   ```

By default, LIBSOL will be installed in the directory "*<LIBSOL>/dist*". If you want to change the installation directory, set the "PREFIX" variable when using "cmake".

```
$ cmake -DPREFIX=/usr/local ..
```

2.3 Install on Windows

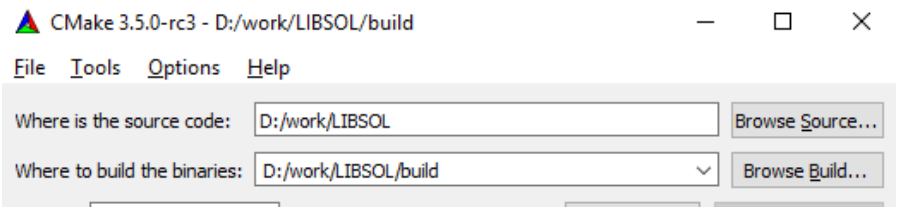
The following steps have been tested on Visual Studio 2013 and Visual Studio 2015. Lower versions of Visual Studio do not provide full support of C++11 features.

2.3.1 Required Packages

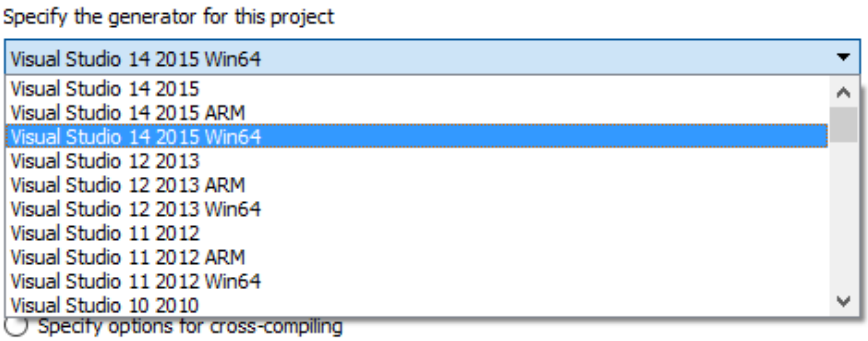
- Visual Studio 2013, 2015, or higher
- CMake 2.8 or higher
- Python 2.7 (required if you want to use the python wrappers)

2.3.2 Build from source

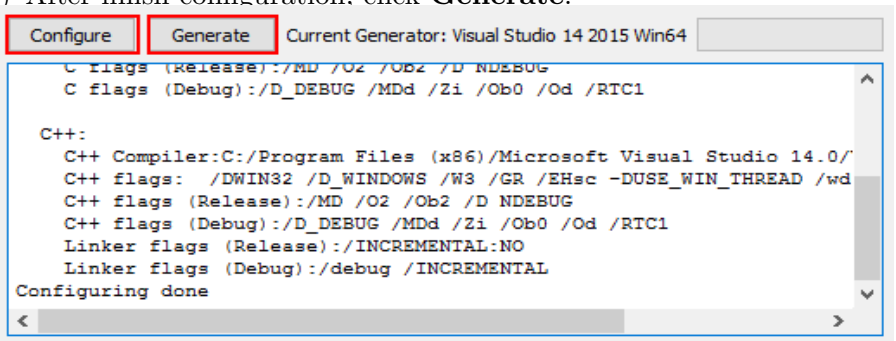
1. Navigate to the root directory of LIBSOL and create a temporary directory to put the generated project files, as well the object files and output binaries. Then follow either Step 2 or Step 3.
2. Install with CMake GUI.
   (a) Open **cmake-gui.exe**, set "where is the source code" and "where to build the binaries".

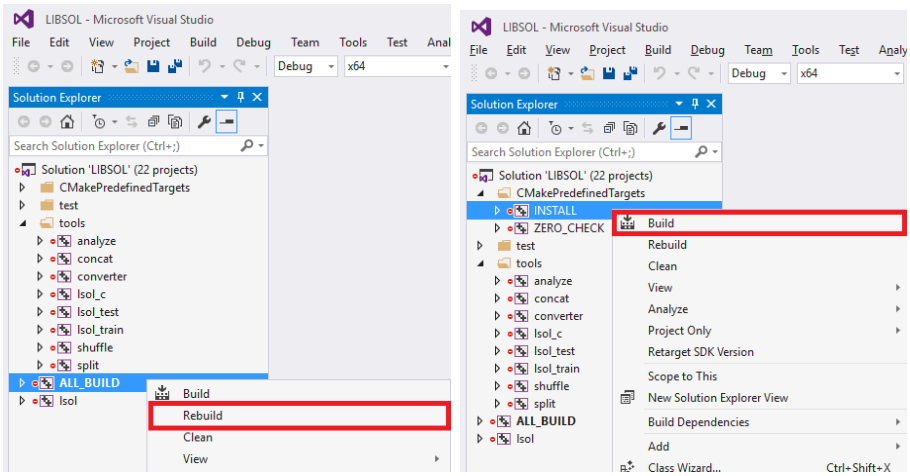


   (b) Click **Configure** and select compiler.



   (c) After finish configuration, click **Generate**.





(d) Open **LIBSOL.sln**, Rebuild the **ALL_BUILD** project and then build the **INSTALL** project.

3. Install from command line. Before this step, you should make sure that cmake is in the environment path or set environment path manually as step (c) shows.

   (a) Search **cmd** in Start Menu and open it.

   (b) Navigate to the root directory of LIBSOL and create a temporary directory to put the generated project files, as well the object files and output binaries.

   ```
   $ cd LIBSOL && mkdir build && cd build
   ```

   (c) If cmake is not in environment path, add by executing the following command:

   ```
   $ set path=<path_to_cmake>;%path%
   ```

   (d) Generate Visual Studio Projects. Example code for Visual Studio 2013, 2015 and their **64-bit** versions are as the following shows:

   ```
   # Generate Visual Studio 2013 Projects
   $ cmake -G ''Visual Studio 12 2013'' ..
   # Generate 64-bit Visual Studio 2013 Projects
   $ cmake -G ''Visual Studio 12 2013 Win64'' ..
   # Generate Visual Studio 2015 Projects
   $ cmake -G ''Visual Studio 14 2015'' ..
   # Generate 64-bit Visual Studio 2015 Projects
   $ cmake -G ''Visual Studio 14 2015 Win64'' ..
   ```

   (e) Open **LIBSOL.sln**, Rebuild **ALL_BUILD** project and then build **INSTALL** project.

2.4 Install Python Wrappers

To install python wrappers:

```
$ cd LIBSOL/python && pip install -r requirements.txt
```

3 Command Line Tools

LIBSOL provides a set of useful command line tools related to data processing and model training/testing.

3.1 Data Formats and Preprocessing Tools

Data formats supported by this software package are "svmlight" (commonly used in LIBSVM and LIBLINEAR), "csv", and a binary format defined by ourselves. Labels and features in data files should all be numeric.

1. **The Binary Format.** The binary format is for fast loading and processing. It is used to cache datasets, like in cross-validation procedures. Each sample in binary format is comprised of the following items in sequence:
   - **label**: *sizeof(label_t)*, *label_t* is *int32_t* by default;
   - **feature number**: *sizeof(size_t)*;
   - **length of compressed index**: *sizeoof(size_t)*;
   - **compressed index**: *sizeof(char)* * **length of compressed index**;
   - **features**: *sizeof(float)* * **feature number**;
2. **Data Preprocessing Tools** The library provides some tools to help users preprocess datasets, like analyzing, splitting, shuffling , etc. Note that the tools support the data formats as mentioned above.
   - **analyze**: analyze the data number, feature number, feature dimension, nonzero feature number, class number, feature sparsity of datasets.
   - **concat**: concatenate several data files into one.
   - **converter**: convert data from one format to another.
   - **shuffle**: shuffle the order of data samples.
   - **split**: split one data file into several parts.

   The detailed input parameters for the tools can be obtained by running the tool without options or with "-h" or "–help" option.

3.2 Training Tool

The training tool is **libsol_train**. Running **libsol_train** without any arguments or with "–help/-h" will produce a message which briefly explains each argument.

The command to call **libsol_train** is:

```
$ libsol_train [options] ... train_file [model_file]
```

Options include "general options", "io options" and "model options".

1. **General Options**

```
-h arg :    show all help information;
-s arg :    show list of information specified by arg.
```

The "-s" option is to help users know what the library can do (what kind of algorithms are implemented, what kind of data formats are supported, what kind of loss functions are implemented, etc.) without checking the code. The available arguments include "reader", "writer", "model", and "loss". For example, running with "model" will show users the available algorithms as follows:

```
$ libsol_train -s model
[output is:]
ada-fobos-l1:   Adaptive Subgradient FOBOS with l1 regularization
ada-fobos:      Adaptive Subgradient FOBOS
ada-rda-l1:     Adaptive Subgradient RDA with l1 regularization
ada-rda:        Adaptive Subgradient RDA
alma2:  Approximate Large Margin Algorithm with norm 2
arow:   Adaptive Regularization of Weight Vectors
cw:     confidence weighted online learning
eccw:   exact convex confidence weighted online learning
erda-l1:        mixed l1-l2^2 enhanced regularized dual averaging
fobos-l1:       Forward Backward Splitting l1 regularization
ogd:    Online Gradient Descent
pa1:    Online Passive Aggressive-1
pa2:    Online Passive Aggressive-2
pa:     Online Passive Aggressive
perceptron:     perceptron algorithm
rda-l1: mixed l1-l2^2 regularized dual averaging
rda:    l2^2 regularized dual averaging
sop:    second order perceptron
stg:    Sparse Online Learning via Truncated Gradient
```

And Running with "reader" will show users the data readers for all support data formats:

```
$ libsol_train -s reader
bin:    binary format data reader
csv:    csv format data reader
svm:    libsvm format data reader
```

2. **IO Options**

```
-f  arg :    dataset format ('svm'[default], 'bin', or 'csv')
-c  arg :    nubmer of classes (default=2)
-p  arg :    number of passes to go through the data (default=1).
-d  arg :    dimension of the data.
```

Note that the IO options are not required. For the "-d" option, if not specified, the tool will learn the dimension by itself, with a little more memory cost.

3. **Model Options**

```
-a  arg   :    learning algorithm to use (see ``General Options'')
-m  arg   :    path to pre-trained model for finetuning
--params arg:    parameters for algorithms in the format
    ``param=val;param2=val2;...''
```

Some usefule parameters include:

loss=[string]

```
bool          :   1 if wrong predict, 0 if correct
hinge         :   hinge loss
maxscore-bool :   multi-class max-score bool loss
maxscore-hinge:   multi-class max-score hinge loss
uniform-bool  :   multi-class uniform bool loss
uniform-hinge :   multi-class uniform hinge loss
```

lambda=[float]

Regularization parameter for sparse online learning algorithms

norm=[string]

Normalize the data samples, the supported normalization method include:

```
l1 :  divide each feature by L1 norm
l2 :  divide each feature by L2 norm
```

eta=[float]

Learning rate for online algorithms.

For the **OGD** algorthm, learning rate is as the following equation shows:

$$\eta_t = \frac{\eta_0}{(t_0 + t)^p} \tag{3}$$

```
eta      arg :  initial learning rate
power_t  arg :  power t of decaying learning rate
t        arg :  initial iteration number
```

So the options can be:

```
$ libsol_train -a ogd --params "eta=0.1;power_t=1;t=100" data_path
```

Table 2 shows the algorithms and their correspondent parameters.

We provide an example to show how to use **libsol_train** and explain the details of how **libsol_train** works. The dataset we use will be "a1a" provided in the "data" folder.

The command for training wit default algorithm is as the following shows.

```
$ libsol_train data/a1a
```

Output of the above command will be:

```
Model Information:
{
    "clf_num" : 1,
    "cls_num" : 2,
    "loss" : "hinge",
    "model" : "ogd",
    "norm" : 0,
    "online" : {
        "bias_eta" : 0,
        "dim" : 1,
        "eta" : 1,
        "lazy_update" : "false",
        "power_t" : 0.5,
```

```
        "t" : 0
    }
}

Training Process....
Iterate No.             Error Rate              Update No.
2                       0.500000                1
4                       0.250000                1
8                       0.125000                1
16                      0.187500                3
32                      0.125000                8
64                      0.218750                19
128                     0.179688                41
256                     0.218750                95
512                     0.210938                179
1024                    0.205078                381
1605                    0.187539                567
training accuracy: 0.8125
training time: 0.031 seconds
model sparsity: 15.1260%
```

Illustrations:

- **Model Information**: Class number, classifier number, including specified algorithm, and detailed parameters as specified by "–params" option above.
- **Trainin Process**: The iteration information: the first column is number of processed data samples; the second column is the training error rate; the third column is number of updated times of the classifiers.
- **Summary**: The final training accuracy, time cost, and model sparsity.

By default, LIBSOL use the "OGD" algorithm to learn a model. If users want to try another algorithm ("AROW" for example) and save the learnt model to a file ("arow.model"):

```
$ libsol_train -a arow data/a1a arow.model
```

Each algorithm may have its own parameters as illustrated in "Model Options". The following command changes the default value of parameter "r" to "2.0":

```
$ libsol_train -a arow --params r=2.0 data/a1a arow.model
```

In some cases we want to finetune from a pre-trained model, the command is:

```
$ libsol_train -m arow.model data/a1a arow2.model
```

3.3 Test Tool

The test tool is **libsol_test**. The test command is:

```
$ libsol_test model_file data_file [predict_file]


model_file   :  model trained by libsol_train
data_file    :  path to test data
predict_file :  path to save the prediction results(optional)
```

Table 2: Algorithm specific Parameters

Algorithm	Parameters	Meaning
Ada-FOBOS	eta	learning rate
	delta	parameter to ensure positive-definite property of the adaptive weighting matrix
Ada-RDA	eta	learning rate
	delta	parameter to ensure positive-definite property of the adaptive weighting matrix
ALMA	alpha	final margin parameter $(1-\alpha)\gamma_\star$
	C	typically set to $\sqrt{2}$.
AROW	r	parameter of passive-aggressive update trade-off
CW	a	initial confidence
	phi	threshold of inverse normal distribution of the threshold-probability.
ECCW	a	initial confidence
	phi	threshold of inverse normal distribution of the threshold-probability.
OGD	eta	learning rate
	power_t	power to of decaying learning rate
PA1	C	passive-aggressive trade-off parameter
PA2	C	passive-aggressive trade-off parameter
RDA	sigma	coefficient of the proximal function
ERDA	sigma	coefficient of the proximal function
	rou	parameter for l1 penality in proximal function
SOP	a	parameter for positive definite normalization matrix

For exmaple, We can test with the learned model:

```
$ libsol_test arow.model data/a1a.t predict.txt
test accuracy: 0.8437
test time: 0.016 seconds
```

3.4 Python Wrapper

The library provides python wrappers for users. The similar command line toos are "**libsol_train.py**" and "**libsol_test.py**". The usage are almost exactly the same as "**libsol_train**" and "**libsol_test**", except that "**libsol_train.py**" provides the **cross validation** function. For example, if users want to do a 5-fold GridSearch Cross Validation in the range $[2^-5, 2^-4, ..., 2^4, 2^5]$ for parameter "r" of "AROW", the command and output will be:

```
$ python python/libsol_train.py -a arow --cv r=0.03125:2:32 -f 5 \
    data/a1a arow.model
cross validation parameters: [('r', 2.0)]
```

For advanced users, they can **import Model from lsol_core** to their own python scripts. The **Model** class provides the **train** and **test** functions for embedded usage of LBISOL.

4 Library Call

LIBSOL provides a dynamic library named "**lsol.dll**" on Windows or "**liblsol.so**" on Unix like systems. Interfaces of the library are comprised of three parts: **initialization**, **training/testing**, and **finalization**.

4.1 Initialization of IO

Initialization of IO means to create a data iterator. The API is:

```
void* lsol_CreateDataIter(int batch_size, int buf_size);
```

- **batch_size**: the number of data samples in each mini-batch while loading data;
- **buf_size**: number of mini-batches to cache while loading data.

With the created data iterator, users can load one or more data sequentially. The API is:

```
void* lsol_LoadData(void* data_iter, const char* path, const char*
    format, int pass_num);
```

- **data_iter**: the created data iterator;
- **path**: path the training data;
- **format**: format of the data ('svm', 'csv', 'bin', etc.);
- **pass_num**: number of passes to go through the data.

Note: **lsol_LoadData** can be called multiple times to add multiple training data. The data will be processed sequentially as they are added to the data iterator. This is in accordance with the target of online learning, where the data distribution may change over time.

4.2 Initialization of Model

Initialization of Model means to create a model instance. There are two ways: train from scratch and finetune from an existing model.

1. Train from Scratch

Create a new clean model for training from scratch:

```
void* lsol_CreateModel(const char* name, int class_num);
```

- **name**: name of the algorithm to be used;
- **class_num**: number of classes to train.

2. Finetune from Existing Model

Load an existing model for continuous training:

```
void* lsol_RestorModel(const char* model_path);
```

- **model_path**: path to existing model

3. Set Model Parameter

```
void* lsol_SetModelParameter(void* model, const char* param_name,
      const char* param_val);
```

- **model**: the model instance;
- **param_name**: name of the parameter ('loss', 'eta', etc.);
- **param_val**: value string of the parameter.

4.3 Training & Testing

The API to train a model is:

```
void lsol_Train(void* model, void* data_iter);
```

- **model**: the model instance;
- **data_iter**: data iterator to be used for training.

The API to test a model is:

```
void lsol_Test(void* model, void* data_iter, const char*
      output_path);
```

- **model**: the trained model instance;
- **data_iter**: data iterator to be used for testing.
- **output_path**: path to save the predicted results, if no need to save, set to NULL.

4.4 Finalization

Finalization is to release the initialized resources. The functions are:

```
void lsol_ReleaseDataIter(void** data_iter);
//save model to a file
void lsol_SaveModel(void* model, const char* model_path);
void lsol_ReleaseModel(void** model);
```

4.5 Use Library on Windows Visual Studio

LIBSOL can only be linked as a dynamic library to support the reflection of class names to class constructors. It is named as "lsol.dll" on Windows. Interface of the library is in **"lsol/c_api.h"**. Suppose LIBSOL is installed into **"<LIBSOL>/dist"**. The following shows an example to link LIBSOL to another project in Visual Studio 2015.

1. Create an empty Win32 C/C++ project.
2. Add **include** path and **library** path to the created project. **Properties** → **Select All Configurations** → **VC++ Directories**.
3. Add link libraries to the project. **Linker** → **Input** → **add the library**. In **Release** mode, add **lsol.lib**. In **Debug** mode, add **lsold.lib**.
4. Test the program. Add a new source file and use the above APIs to test the library. The **"LIBSOL/tools/lsol_c.cc"** is a good example to use the Library Calls.

4.6 Use Library on Linux/Unix

Suppose LIBSOL is installed into **"<LIBSOL>/dist"**. On linux, users need to set the include path, library link path, and linked libraries for "gcc/g++/clang/clang++" compilers.

1. Include path:

   ```
   -I <LIBSOL>/dist/include
   ```

   Then include "**lsol/c_api.h**" in your source files.

2. Link path and library

   ```
   -L <LIBSOL>/dist/bin -llsol
   ```

5 Design & Extension of the Library

The design principle is to keep the package simple, easy to read and extend. All codes follow the C++11 standard and need no external libraries. The reason to choose C++ is for feasibility and efficiency in handling large scale high dimensional data. The system is designed so that machine learning researchers can quickly implement a new algorithm with a new idea, and compare it with a large family of existing algorithms without spending much time and efforts in handling large scale data.

In general, LIBSOL is written in a modular way, including **PARIO**(for PARallel IO, whose key component is **DataIter**), **Loss**, and **Model**. User can extend it by inheriting the base classes of these modules and implementing the corresponding interfaces; Thus, we hope that LIBSOL is not only a machine learning tool, but also a comprehensive experimental platform for conducting online learning research. Figure 1 shows the framework of the system.

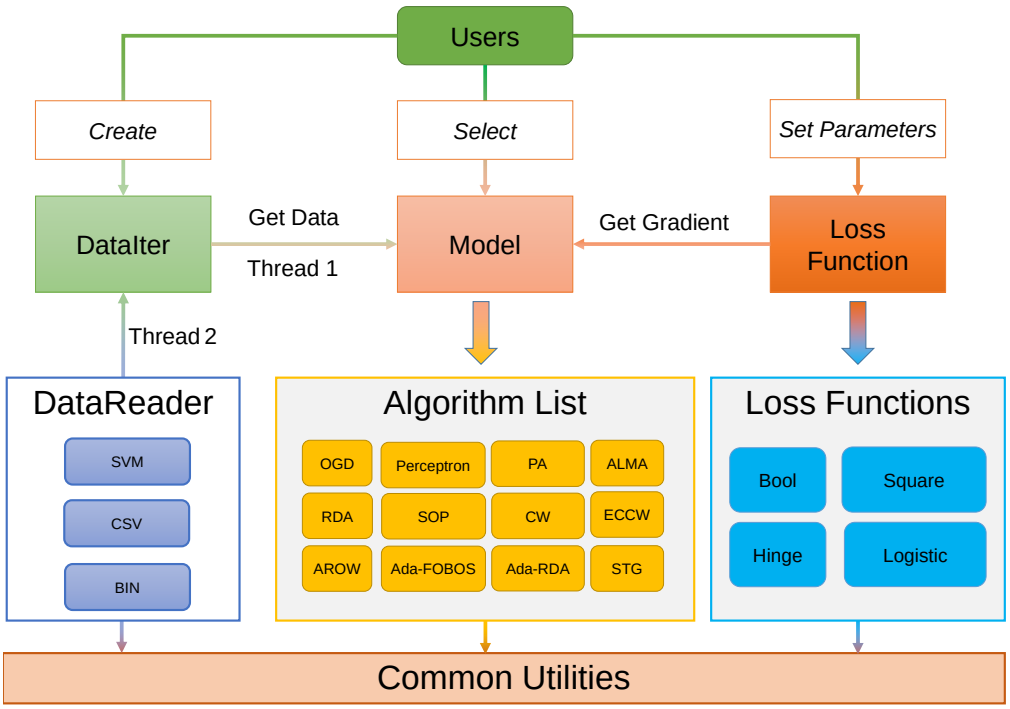


Figure 1: Framework of LIBSOL

5.1 How to Add New Algorithms

A salient property of this library is that it provides a fairly easy-to-use testbed to facilitate online learning researchers to develop their new algorithms and conduct side-by-side comparisons with the state-of-the-art algorithms on various datasets with various loss functions with the minimal efforts. More specifically, adding a new algorithm has to address three major issues:

- What is the condition for making an update? This is usually equivalent to defining a proper loss function (e.g., a hinge loss $l_t = \max(0, 1 - y_t * f_t)$) such that an update occurs wherever the loss is nonzero, i.e., $(l_t > 0)$. Please check details in Section 5.3.

- How to perform the update on the classifier (i.e.,the weight vector **w**) whenever the condition is satisfied? For example, Perceptron updates $w = w + y_t * x_t$; Please check details in Section 5.2.

- Are there some parameters in your new algorithm? If so and you want your algorithm can be serialized to and deserialized from files, you need to do some parameter setting (**SetParameter**), model serialization (**GetModelInfo**, **GetModelParam**) and deserialization (**SetModelParam**). Please check details in Section 5.2.

5.2 Model

Model is about all the learning algorithms. There are several child base classes for different kind of algorithms.

5.2.1 Model

This is the base class for all algorithms. It implements the main test strategy of algorithms and serialization/deserialization functions. The interfaces include:

- **Constructor**:

```
Model(int class_num, const std::string& type);
```

  Here the parameter **type** indicates whether the algorithm is an online algorithm, stochastic, or batch algorithm for future extension purposes.

- **Parameter Setting**[optional]:

```
virtual void SetParameter(const string& name, const string&  value);
```

  Each algorithm may have its own parameters with different names. This interface allows the new algorithms to parse their parameters easily. The function will throw an **invalid_argument** exception if a wrong parameter is set.

- **Training Initialization**[optional]:

```
virtual void BeginTrain();
```

  Some algorithms may need to do some initialization before training. If so, the algorithm can overload this function and place the initialization code here.

- **Training Finalization**[optional]:

```
virtual void EndTrain();
```

  Some algorithms may need to do some finalization after training. If so, the algorithm can overload this function and place the finalization code here. For example, for sparse online learning algorithms, they need to truncate weights after all iterations if they used the lazy-update strategy.

- **GetModelInfo**[optional]:

```
virtual void GetModelInfo(Json::Value& root) const;
```

This function is for the serialization of the model to model files. If th new algorithm contains some hyper-parameters, it should overload this function to serialize the hyper-parameter to a json object.

- **GetModelParameter**[optional]:

```
virtual void GetModelParam(ostream& os) const;
```

This function is for the serialization of the model to model files. If th new algorithm contains some other parameters, it should overload this function to serialize the parameters to an output stream. For example, for second order online learning algorithm, they often have another matrix about the second order information. While the base class only know there exists a weight vector, the algorithms should overload this interface and serialize the second order matrix by themselves.

- **SetModelParameter**[optional]:

```
virtual void SetModelParam(istream& os);
```

This function is for the deserialization of the model from model files (the inverse step of **GetModelParam**). Generally, if an algorithm overloads **GetModelParam**, it should also overload **SetModelParam**.

Users may be confused that why they do not need to overload **SetModelInfo**. The reason is that new algorithms have already provided the **SetParameter** interface for the setting of hyper-parameters. The base class will automatically call this function during deserialization.

5.2.2 Online Model

This is the base class for all online algorithms. It implements the main training strategy of online algorithms and defined some shared hyper-parameters or auxiliary parameters for online learning algorithms. The two interfaces that users may need to take care are:

- **Predict in training**[optional]:

```
label_t TrainPredict(const DataPoint& dp, float* predicts);
```

dp is the input data sample. **predicts** is the output prediction, with equal length to number of classifiers. The purpose of this interface is that some online algorithms need to do some calculation before prediction in each iteration, like the second order perceptron (SOP) algorithm. In other cases, there is no need to overload this interface.

- **Update dimension**[optional]:

```
void update_dim(index_t dim);
```

For online algorithms, data samples are processed one by one. So the model does not know the dimension of the whole dataset. When new data sample comes, this function is called to ensure that the dimension of the model will be updated. This function should be overloaded in the same case of **GetModelParam** and **SetModleParam**, where extra model parameters exist for algorithms.

5.2.3 Online Linear Model

This is the base class for all online linear algorithms. The only required interface here is the update function, which is the key algorithm for online linear algorithms.

- **Update function**[**required**]:

  ```
  void Update(const DataPoint& dp, const float* predicts, float loss);
  ```

  dp is the input data sample. **predicts** is the prediction on each classes. **loss** is the output of loss function.

5.3 Loss Function

At the moment, we provide a base class (purely virtual class) for loss functions and four inherited classes(*BoolLoss*, *HingeLoss*, *LogisticLoss*, and *SquareLoss*) for binary classification, as well as their max-score and uniform loss functions for multi-class classification. The interfaces are:

- **Loss**:

  ```
  virtual float loss(const DataPoint& dp, float* predicts,
          label_t predict_label, int cls_num);
  ```

  Get the loss for the current predictions. The first parameter is the data sample. The second is the prediction on each class. The third is the predicted class label, and the last one is the number of classes.

- **Gradient**:

  ```
  virtual float gradient(const DataPoint& dp, float* predicts,
          label_t predict_label, float* gradient, int cls_num);
  ```

  Get the gradient of the loss function at the current data point. Note that we do not calculate the exact gradient here. The dimension of the fourth parameter **gradient** is the number of classifiers. To linear classification problems, the gradients on different features share a same part. Take Hinge Loss for example:

  $$l(\vec{w}) = 1 - y\vec{w} \cdot \vec{x}$$

  The gradient is:

  $$l'(\vec{w}) = -y\vec{x}$$

  As a result, we only calculate the shared term $-y$ for the gradients of different features for efficiency concern. Users need to multiply the correspondent feature $x[i]$ in the optimization algorithms.

5.4 DataIter

DataIter is in charge of loading and transferring data from disk to models efficiently. The two major functions are: adding and data reader and providing parsed data.

5.4.1 Extension of Data Reader

Adding data reader means **DataIter** should allow users to add a reader to parse the source data files. Generally, users just need to specify the following function of **DataIter** to add a reader:

```
int AddReader(const std::string& path, const std::string& dtype,
    int pass_num = 1);
```

- **path**: path to the data file;
- **dtype**: type of the data file ("svm", "bin", "csv");
- **pass_num**: number of passes to go through the data.

By default, **dtype** supports "svm", "bin", and "csv". Extension of Data reader is to implement a reader for a new data type.

The new type of data reader should inherit from the base class **DataReader**, (in *include/lsol/pario/data_reader.h*) and implement the interfaces as follows:

- **Open data file**:

  ```
  virtual int Open(const std::string& path, const char* model=''r'');
  ```

  Open a data file to load data. The function returns **Status_OK**(0) if everything is ok.

- **Parse data**:

  ```
  virtual int Next(DataPoint& dst_data);
  ```

  Parse a new data point from the input source. The parameter is the place to store the parsed data.

  The function returns **Status_OK**(0) if everything is ok.

- **Rewind**[optional]:

  For some special data formats like csv, the first line is some meta data. In this case, Rewind should be inherited and overloaded.

  ```
  virtual void Rewind();
  ```

- **Close**[optional]:

  If the new reader needs to allocate some resources, it should overload the **Close** function to avoid memory leak.

  ```
  virtual voidnt Close();
  ```

5.5 Examples to add new algorithms

In this page, we use the "Ada-FOBOS" and "Ada-FOBOS-L1" algorithms to show how to extend the library to add new algorithms.

5.5.1 Ada-FOBOS and Ada-FOBOS-L1 Algorithms

The following steps show how Ada-FOBOS and Ada-FOBOS-L1 Algorithms work.

1. Input:
   - η: learning rate
   - δ: parameter to ensure positive-definite property of the adaptive weighting matrix
2. Procedures:
   (a) receive new example $\mathbf{x}_t$
   (b) predict results:

$$y_t = \mathbf{w}_t \cdot \mathbf{x}_t$$

   (c) suffer loss l_t
   (d) calculate gradient $\mathbf{g}_t$ with respect to $\mathbf{w}_t$
   (e) update the diagonal matrix H_t of the proximal function $\frac{1}{2}(\mathbf{x}_t^T H_t \mathbf{x}_t)$

$$H_t = \delta + diag(\Sigma_{i=1}^{t} g_i g_i^T)$$

   (f) update $\mathbf{w}_t$

$$\text{Ada-FOBOS} \quad : \quad \mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta \mathbf{g}_t}{H_t}$$

$$\text{Ada-FOBOS-L1} \quad : \quad \mathbf{w}_{t+1} = truncate(\mathbf{w}_t - \frac{\eta \mathbf{g}_t}{H_t}, \frac{\lambda \eta}{H_t})$$

5.5.2 Step by Step Implementation of Ada-FOBOS

1. **Loss Function**

   Ada-FOBOS uses "Hinge Loss" or "Logistic Loss" as the loss function. So we do not need to write new loss functions.

2. **Create File**

   Since Ada-FOBOS is an online linear algorithm, we first create a new file named **ada_fobos.h** in *include/lsol/model/olm*. In the header, we include the base class **OnlineLinearModel**. We need to define the constructor and destructor of the algorithm.

```
#include <lsol/model/online_linear_model.h>

class AdaFOBOS: public OnlineLinearModel {
    public:
    AdaFOBOS(int class_num);
    virtual ~AdaFOBOS();
};
```

3. **Update Function**

   Then we define the **UPDATE** function to update the model, which is the key of the algorithm. Note that **OnlineLinearModel** will predict on the incoming example and calculate the loss automatically.

```
class AdaFOBOS: public OnlineLinearModel {
    public:
    AdaFOBOS(int class_num);
    virtual ~AdaFOBOS();

    protected:
    virtual void Update(const pario::DataPoint& dp, const float*
          predict, float loss);
};
```

4. **Parameters**

   The algorithm needs the following parameters:

   - **eta**: learning rate, which is a shared parameter of online methods and is already defined in **OnlineModel**. Since Ada-FOBOS uses constant learning rate, we need to set its value in **SetParameter** function.
   - **delta**: we need to set its value in **SetParameter** function. Besides, we need serialize the parameter in **GetModelInfo** function. The desiralization is achieved by **SetParameter**.
   - **H**: we need to keep the diagonal matrix H_t (denoted as a vector since it's diagonal) for each class. We need to overload the **update_dim** to adaptively update the dimension of H_t. Besides, we need to overload **GetModelParam** and **SetModelParam** to serialize and deserialize H_t.

```
class AdaFOBOS: public OnlineLinearModel {
    public:
    AdaFOBOS(int class_num);
    virtual ~AdaFOBOS();

    virtual void SetParameter(const string& name, const string&
          value);

    protected:
    virtual void Update(const pario::DataPoint& dp,
          const float* predict, float loss);

    virtual void update_dim(index_t dim);
    virtual void GetModelInfo(Json::Value& root) const;
    virtual void GetModelParam(ostream& os) const;
    virtual int SetModelParam(istream& is);

    protected:
    float delta_;
    math::Vector<real_t>* H_;
};
```

5. **Implementation**

(a) **Constructor and Destructor**

In constructor, we need to allocate the space for $\mathbf{H}\_$. In destructor, we need to free the space of $\mathbf{H}\_$.

```
AdaFOBOS::AdaFOBOS(int class_num):
                OnlineLinearModel(class_num), delta_(10.f) {
    // clf_num_ is the actual number of classifiers
    // when class number is 2, clf_num_ = 1. Otherwise,
                clf_num_ = class_num
    this->H_ = new math::Vector<real_t>[this->clf_num_];
    for(int i = 0; i < this->clf_num_; ++i){
        this->H_[i].resize(this->dim_);
        this->H_[i] = this->delta_;    //initialize
    }
}

AdaFOBOS::~AdaFOBOS() {
    DeleteArray(this->H_);
}
```

(b) **Update Function**

We can implement the update function in almost the same way as the algorithm shows in a MATLAB style as follows:

```
void AdaFOBOS::Update(const pario::DataPoint& dp, const float*
                predicts, float loss) {
    const auto& x = dp.data();
    for (int c = 0; c < this->clf_num; ++c){
        if (g(c) == 0) continue; //no need to update

        // The gradient has already been calculated by the
        // parent class OnlineLinearModel, which is g(c) * x.
        // Please refer to the documentation for the reason
        // The following function update H_ incrementally
        // with new coming examples
        H_[c] = Sqrt(L2(H_[c] - delta_) + L2(g(c) * x)) +
                delta_;
        //update bias
        H_[c][0] = sqrtf(H_[c][0] - delta_) * (H_[c][0] -
                delta_) + g(c) * g(c)) + delta_;

        //update weight vector
        w(c) -= eta_ * g(c) * x / H_[c];
        //update bias
        w(c)[0] -= bias_eta() * g(c) / H_[c][0];
    }
```

(c) **Parameters**

- SetParameter & Update Dimension

```
void AdaFOBOS::SetParameter(const string& name,
                    const string& value) {
    if (name == "delta") {
        this->delta_ = stof(value);
        Check(delta_>0);
    }
    else if (name == "eta") {
```

```
            this->eta_ = stof(value);
            Check(delta_>0);
        }
        else {
            OnlineLinearModel::SetParameter(name, value);
        }
    }

    void  AdaFOBOS::update_dim(index_t dim) {
        if (dim > this->dim_) {
            real_t delta = this->delta_;
            for (int c = 0; c < this->clf_num_; ++c){
                this->H_[c].resize(dim); //resize to the new dim
                // we set the new value of the added dimension
                // (from dim_ to dim) to delta
                this->H_[c].slice_op([delta](real_t& val) {
                    val = delta;
                }
                , this->dim_);
            }
            OnlineLinearModel::update_dim(dim);
        }
```

- Serialization & Deserialization

```
    void AdaFOBOS::GetModelInfo(Json::Value root) {
        OnlineLinearModel::GetModelInfo(root);
        root["online"]["delta"] = this->delta_;
        root["online"]["eta"] = this->eta_;
    }

    void AdaFOBOS::GetModelParam(ostream&  os) {
        OnlineLinearModel::GetModelParam(os);
        for (int c = 0; c < this->clf_num_; ++c) {
            os<<"H["<<c<<"]: "<<this->H_[c]<<"\n";
        }
    }

    int AdaFOBOS::SetModeoParam(istream&  is) {
        OnlineLinearModel::SetModelParam(is);

        string line; //for the header "H[*]"
        for (int c = 0; c < this->clf_num_; ++c) {
            is >>line>>this->H_[c]; ;
        }
    }
    return Status_OK;
```

- Registration
  To make sure that the tool knows we add a new algorithm, we register the new algorithm

```
    RegisterModel(AdaFOBOS, "ada-fobos",
                  "Adaptive Subgradient FOBOS");
```

5.5.3 Step by Step Implementation of Ada-FOBOS-L1

The only difference between Ada-FOBOS-L1 and Ada-FOBOS is the update function. We can simply add a new parameter λ and then modify the **Update** function as required.

However, truncation of all dimension is not efficient when dimension is high and data is sparse. So researchers proposed the lazy update scheme. The learner truncates the weights only when needed, that is:

- when the input data requires that dimension;
- when the learning process finishes, the learner needs to truncate all weights.

The lazy udpate scheme is a common way for many sparse online learning algorithms, which is implemented as **LazyOnlineL1Regularizer** in this toolbox.

As a result, the difference between Ada-FOBOS-L1 and Ada-FOBOS now is to truncate weights when predicting on the current example, rather than the **Update** function.

```
class AdaFOBOS_L1: public AdaFOBOS {
    public:
    AdaFOBOS_L1(int class_num);

    // the learner needs to truncate all weights at the end of training
    virtual void EndTrain();
    protected:
    // the learner needs to truncate weights before prediction
    virtual label_t TrainPredict(const pario::DataPoint& dp,
      const float* predict);

    protected:
    LazyOnlineL1Regularizer l1_;
};
```

And the implementations:

```
AdaFOBOS_L1::AdaFOBOS_L1(int class_num): AdaFOBOS(class_num) {
    this->regularizer)_ = &l1_;
      //tell the parent classes that we have a regularizer
}

label_t AdaFOBOS_L1::TrainPredict(const pario::DataPoint& dp, float*
      predicts) {
    const auto& x = dp.data();

    real_t t = real_t(cur_iter_num_ - 1); //current iteration,
      since not updated, minus 1
    // truncate the weights
    for(int c = 0; c < this->clf_num_; ++c) {
        // the w(c).slice(x)
      means that we only need to truncate weights needed by x
        // this is a similar operation to numpy
      to index part of the elements in array
        w(c) = truncate(w(c).slice(x), (eta_ * l1_.lambda()) * (t -
      l1_.last_update_time()) / H_[c]);
        //truncate bias
        w(c)[0] = truncate(w(c)[0], bias_eta() * l1_.lambda() /
      H_[c][0]);
    }
```

```
    //normal prediction
    return OnlineLinearModel::TrainPredict(dp, predicts);
};

void
 AdaFOBOS_L1::EndTrain() {
    real_t t = cur_iter_num_;
    // truncate all weights
    for(int c = 0; c < this->clf_num_; ++c) {
        w(c) = truncate(w(c), (eta_ * l1_.lambda()) * (t -
      l1_.last_update_time()) / H_[c]);
        //truncate bias
        w(c)[0] = truncate(w(c)[0], bias_eta() * l1_.lambda() /
      H_[c][0]);
    }

    OnlineLinearModel::EndTrain();
};

RegisterModel
(AdaFOBOS_L1, "ada-fobos-l1",
    "Adaptive Subgradient FOBOS with L1 regularization");
```

References

[1] N. Cesa-Bianchi, A. Conconi, and C. Gentile. A second-order perceptron algorithm. *SIAM J. Comput.*, 34(3):640–668, Mar. 2005.

[2] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585, 2006.

[3] K. Crammer, M. Dredze, and F. Pereira. Exact convex confidence-weighted learning. In *Advances in Neural Information Processing Systems*, pages 345–352, 2008.

[4] K. Crammer, A. Kulesza, and M. Dredze. Adaptive regularization of weight vectors. *Machine Learning*, pages 1–33, 2009.

[5] M. Dredze, K. Crammer, and F. Pereira. Confidence-weighted linear classification. In *Proceedings of the 25th international conference on Machine learning*, pages 264–271. ACM, 2008.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[7] J. Duchi and Y. Singer. Efficient online and batch learning using forward backward splitting. *The Journal of Machine Learning Research*, 10:2899–2934, 2009.

[8] C. Gentile. A new approximate maximal margin classification algorithm. *J. Mach. Learn. Res.*, 2:213–242, Mar. 2002.

[9] S. C. Hoi, J. Wang, and P. Zhao. Libol: A library for online learning algorithms. *The Journal of Machine Learning Research*, 15(1):495–499, 2014.

[10] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. *The Journal of Machine Learning Research*, 10:777–801, 2009.

[11] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[12] L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of Machine Learning Research*, 9999:2543–2596, 2010.

[13] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. 2003.