

# SOL: A Large Scale Sparse Online Learning Library

Version 0.1.0

January 17, 2014

## **Abstract**

SOL is an open-source library for large-scale sparse online learning, which consists of a family of efficient and scalable sparse online learning algorithms for large-scale online classification tasks. We have offered easy-to-use command-line tools and examples for users and developers. We also have made comprehensive documents available for both beginners and advanced users. SOL is not only a machine learning tool, but also a comprehensive experimental platform for conducting large scale sparse online learning research.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Problem Formulation . . . . .	3
1.2	Summary of Main Algorithms . . . . .	3
1.3	Main Features . . . . .	4
<b>2</b>	<b>Installation</b>	<b>4</b>
2.1	Installation on Linux . . . . .	4
2.2	Install on Windows . . . . .	5
<b>3</b>	<b>How to Use the SOL Package</b>	<b>6</b>
3.1	Data Formats and Folders . . . . .	7
3.2	Command Line . . . . .	7
3.3	Use Command Line . . . . .	8
3.4	Dynamic & Static Library Interface . . . . .	10
3.5	Use Dynamic & Static Library on Windows . . . . .	11
3.6	Use Dynamic & Static Library on Linux . . . . .	13
3.7	Use Scripts in experiments . . . . .	13
3.8	Parameter Settings . . . . .	15
<b>4</b>	<b>System Framework</b>	<b>15</b>
4.1	DataSet . . . . .	16
4.2	Loss Function . . . . .	17
4.3	Optimizers . . . . .	18
4.4	How to Add New Algorithms . . . . .	20
4.5	Extend DataReader . . . . .	20
4.6	Extend Loss Functions . . . . .	20
4.7	Implement your own algorithms . . . . .	20
<b>5</b>	<b>Documentation</b>	<b>20</b>
<b>6</b>	<b>Conclusions, Revision, and Citation</b>	<b>21</b>

# 1 Introduction

Sparse online learning represents a family algorithms which try to explore structure of features and learn a sparse model. It is valuable in handling extremely large scale high dimensional datasets, which cannot be loaded into memory. The problem has been widely studied in batch learning, which is effective but not scalable to large scale high dimensional data. Over the past years, a variety of online learning algorithms have been proposed to explore sparsity and maintain the benefits of online algorithms. However, there is very few comprehensive libraries which include most of the state-of-the-art algorithms for researchers to make easy side-by-side comparisons and for developers to explore their various applications.

In this work, we develop SOL, an easy-to-use sparse online learning tool that consists of a family of existing and recent state-of-the-art sparse online learning algorithms for large-scale sparse online classification tasks. SOL enjoys significant advantages for massive-scale classification in the era of big data nowadays, especially in efficiency, scalability, and adaptability.

## 1.1 Problem Formulation

The focus of sparse online learning is an algorithmic framework for regularized convex programming to the following minimization problem:

$$\min_w f(w) \quad s.t. \quad r(w) < R, \quad (1)$$

where both  $f(w)$  and  $r(w)$  are convex bounded below functions. Often,  $f(w)$  is an empirical loss and takes the form of  $\sum_{i=1}^t l_i$  for a sequence of loss functions  $l_i$ .  $r(w)$  is a regularization term and often takes the form of  $L1$  ( $r(w) = |w|$ ) or  $L0$  ( $r(w) = |w|_0$ ) norm.  $R$  is a pre-defined threshold to control sparsity of the model. With lagrangian multiplier method, equation (1) can be reformulated as:

$$\min_w f(w) + \lambda r(w) \quad (2)$$

## 1.2 Summary of Main Algorithms

Table 1 gives a summary of the family of implemented algorithms in this software package.

Table 1: Summary of the Implemented Algorithms.

Problem Type	Methodology	Algorithm	Description
Binary	First-Order	SGD	Stochastic Gradient Descent [5]
		STG	Truncated sparse online learning [4]
		FOBOS	Forward backward splitting [3]
		RDA	Regularized dual averaging [7]
	Classification	Ada-FOBOS	Adaptive FOBOS [2]
		Ada-RDA	Adaptive regularized dual averaging [2]
	Second-Order	AROW	Confidence weighted learning [1]
		AROW-TG	Confidence weighted truncated gradient
		AROW-DA	Confidence weighted dual averaging
		AROW-FS	Confidence weighted feature selection
		SCW	Soft confidence weighted learning [6]
		SCW-RDA	Soft confidence weighted dual averaging

## 1.3 Main Features

- **Comprehensiveness:** A family of existing sparse online binary algorithms have been implemented in this software package;
- **Extendibility:** One can easily implement a new algorithm by only inheriting from a base class and implementing the key updating virtual functions.
- **Scalability:** The library can handle million and even billion scale dataset on a single PC.
- **Usability:** It is easy to use the main functions to evaluate one algorithm by comparing with all the algorithms;

## 2 Installation

SOL features a very simple installation procedure. No third-party dependencies are required. The project is managed by *CMake*. There exists a *CMakeLists.txt* in the root directory of SOL. Note that all the following are tested on *CMake 2.8*. Lower versions of cmake may work, but are not ensured.

### 2.1 Installation on Linux

The following steps have been tested for Ubuntu 12.04 but should work with other distros as well.

#### Required Packages

- GCC 4.6.2 or later. This can be installed with

```
sudo apt-get install build-essential
```

- CMake 2.8 or higher
- Python 2.7

#### Build from source

1. Create a temporary directory, which we denote as (*<cmake\_binary\_dir>*), where you want to put the generated Makefiles, project files, as well the object files and output binaries.
2. Enter the (*<cmake\_binary\_dir>*) and type

```
cmake [<some optional parameters>] <path to SOL source directory>
```

For example

```
cd SOL
mkdir build
cd build
cmake -DCMAKE_INSTALL_PATH=/usr/local/ ..
```

3. Enter the created temporary directory (*<cmake\_binary\_dir>*) and proceed with:

```
make
make install
```

By default, SOL is installed in the directory *<cmake\_source\_dir>/install*.

## 2.2 Install on Windows

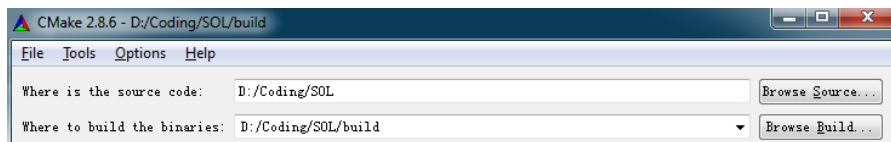
The following steps have been tested on Visual Studio 2010 and Visual Studio 2012 on Windows 7 SP1. Nevertheless but should work with any other modern versions of Visual Studio and Windows OS.

### Required Packages

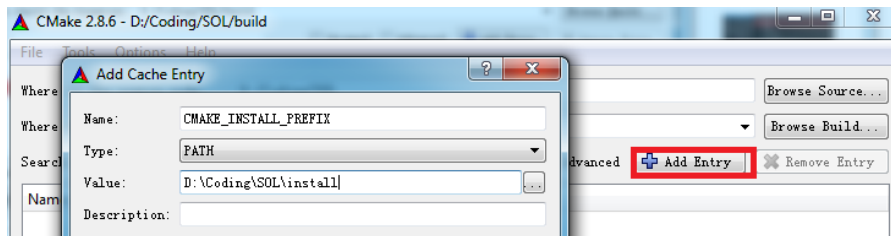
- Visual Studio 2010, 2012, or higher
- CMake 2.8 or higher
- Python 2.7

### Build from source

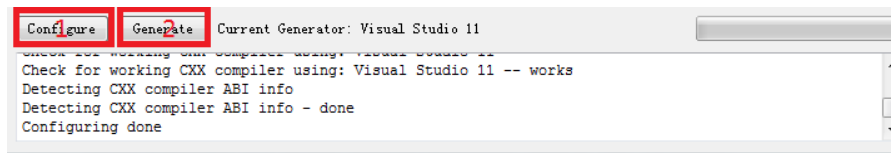
1. Create a temporary directory, which we denote as ( $\langle cmake\_binary\_dir \rangle$ ), where you want to put the generated project files as well the object files and output binaries.
2. Install with CMake GUI.
  - (a) Open *cmake-gui.exe*, set where is the source code and where to build the binaries ( $\langle cmake\_binary\_dir \rangle$ ).



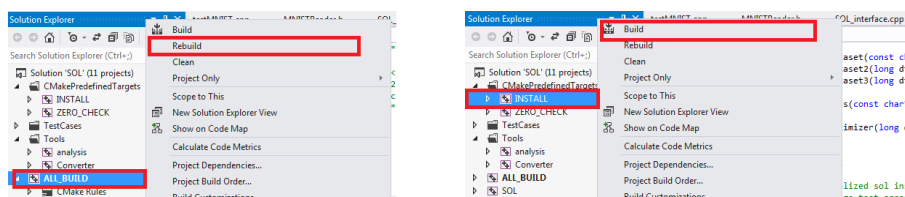
- (b) Click *Add Entry* and set install path. Otherwise, SOL will be install in ( $\langle cmake\ source\ directory \rangle /install$ ).



- (c) Click *Configure* and select compiler.
- (d) After finish configuration, click *Generate*.



- (e) Open *SOL.sln*, Rebuild all the *ALL\_BUILD* project and then build *INSTALL* project.



3. Install from command line.

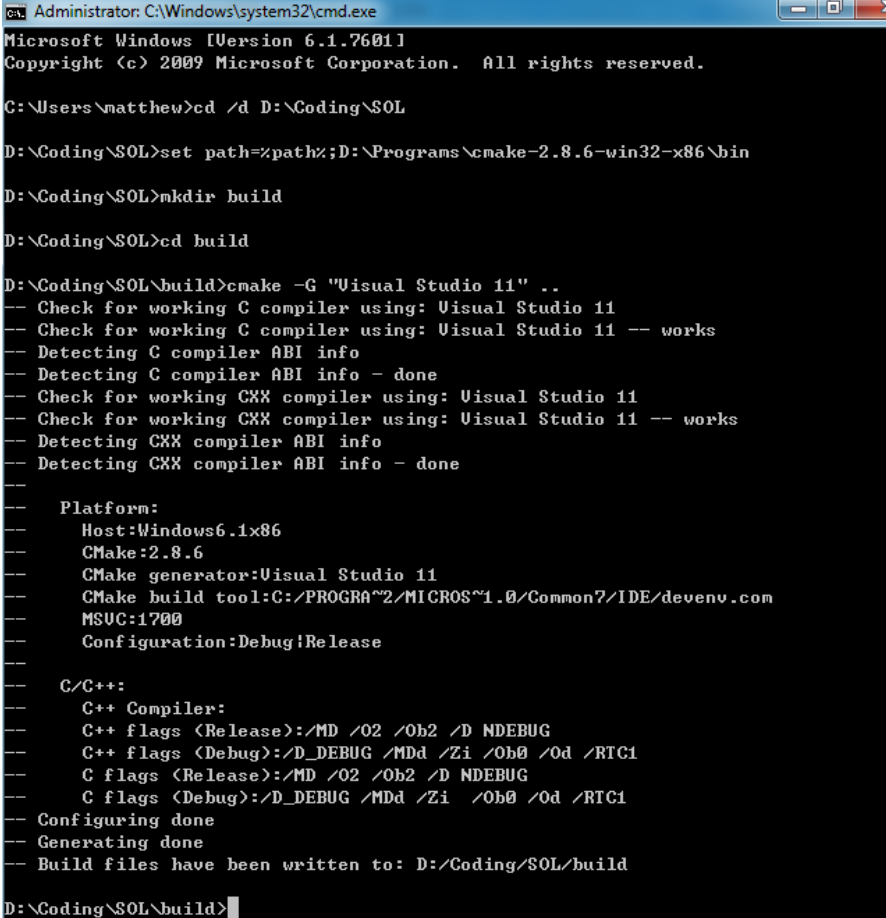
Before this step, you should make sure that cmake is in the environment path or set environment path manually as step (c) shows.

- (a) Search *cmd* in Start Menu and open it
- (b) enter (<*cmake\_binary\_dir*>)
- (c) Add cmake to your environment path by typing:

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

- (d) generate Visual Studio Projects. Example code for Visual Studio 2010 and 2012 are as the following shows:

```
#Generate Visual Studio 2010 Projects
cmake -G "Visual Studio 10" <path to SOL source directory>
#Generate Visual Studio 2012 Projects
cmake -G "Visual Studio 11" <path to SOL source directory>
```



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\matthew>cd /d D:\Coding\SOL

D:\Coding\SOL>set path=%path%;D:\Programs\cmake-2.8.6-win32-x86\bin

D:\Coding\SOL>mkdir build

D:\Coding\SOL>cd build

D:\Coding\SOL\build>cmake -G "Visual Studio 11" ..
-- Check for working C compiler using: Visual Studio 11
-- Check for working C compiler using: Visual Studio 11 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler using: Visual Studio 11
-- Check for working CXX compiler using: Visual Studio 11 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
--
-- Platform:
--   Host: Windows6.1x86
--   CMake: 2.8.6
--   CMake generator: Visual Studio 11
--   CMake build tool: C:/PROGRA~2/MICROS~1.0/Common7/IDE/devenv.com
--   MSVC: 1700
--   Configuration: Debug|Release
--
-- C/C++:
--   C++ Compiler:
--   C++ flags (Release): /MD /O2 /Ob2 /D NDEBUG
--   C++ flags (Debug): /D_DEBUG /MDd /Zi /Ob0 /Od /RTC1
--   C flags (Release): /MD /O2 /Ob2 /D NDEBUG
--   C flags (Debug): /D_DEBUG /MDd /Zi /Ob0 /Od /RTC1
-- Configuring done
-- Generating done
-- Build files have been written to: D:/Coding/SOL/build

D:\Coding\SOL\build>
```

- (e) Open *SOL.sln*, Rebuild all the *ALL\_BUILD* project and then build *INSTALL* project.

### 3 How to Use the SOL Package

The current version of SOL package are written in C++, with some python scripts. The library is packed into an executable, a dynamic library, and a static library. Some test cases are provided as examples.

### 3.1 Data Formats and Folders

Types of data formats supported by this software package are “libsvm” data format (commonly used in LIBSVM and SVM-light), and a binary format defined by ourselves. Features in the “libsvm” data files should all be numeric and labels are restricted to +1, -1, 1 for binary classification.

- **Format of binary format**

The binary format is for fast loading and processing. It is used to cache datasets. Each sample in binary format is comprised of the following items in sequence:

- **label:** sizeof(char)
- **feature number:** sizeof(char)
- **max index:** sizeof(int32\_t)
- **length of compressed index:** sizeof(unsigned int)
- **compressed index:** dynamic
- **features:** feature number \* sizeof(float)
- **sum of feature square:** sizeof(float)

- **Contents of the library:**

- **data:** example datasets
- **doc:** documentation of the library
- **exp:** python and matlab scripts for experiments, including cross validation and performance evaluation
- **src:** source code of the library
- **test:** test code and example use of the library
- **tools:** python scripts to pre-process datasets
- CMakeLists.txt

### 3.2 Command Line

Running SOL without any arguments or with ‘-help/-h’ will produce a message which briefly explains each argument. Below arguments are grouped according to their function.

#### 1. Input Options

```
-i arg :      training file name
-c arg :      cached training file name
-t arg :      test file name
-tc arg :     cached test file name
-or arg :     output readable model
-dt arg :     dataset type format, by default is libsvm
-bs arg :     number of chunks for buffering, default is 2
```

#### 2. Loss Functions

```
-loss arg :      loss function type
```

supported loss functions:

```
Hinge      :      hinge loss
Logit       :      logistic loss
Square      :      square loss
SquareHinge :      squared hinge loss
```

#### 3. Learning Rate Setting:

Learning rate in our library is as the following equation shows:

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

```

-eta      arg : learning rate
-power_t  arg : power t of decaying learning rate
-t0       arg : initial iteration number

```

#### 4. Algorithms and parameters

```

-opt      arg : optimization method:
            SGD, STG, RDA, RDA_E, FOBOS Ada-RDA, Ada-FOBOS,
            AROW, AROW-TG, AROW-DA AROW-FS, SCW, SCW-RDA
-l1       arg : L1 regularization parameter
-norm     : whether normalize the data, default is false
-passes   arg : number of passes
-delta    arg : delta in Adaptive algorithms (Ada-)
-grou     arg : gamma times rou in enhanced RDA (RDA_E)
-k        arg : number of k in truncated gradient descent or
            feature selection
-phi      arg : phi in SCW
-r        arg : r in Confidence weighted algorithms

```

Table 2 shows the algorithms and their correspondent parameters.

Table 2: Comparison of sparse online learning algorithms

Algorithm	Parameters	Meaning
SGD	-eta	learning rate
FOBOS	-eta	learning rate
STG	-eta	learning rate
	-k	truncate the weight vector every k steps
Ada-FOBOS	-eta	learning rate
	-delta	parameter to ensure positive-definite property of the adaptive weighting matrix
AROW-TG	-r	parameter of passive-aggressive update trade-off
RDA	-eat	learning rate
	-grou	decreased L1 regularization parameter
RDA_E	-eta	learning rate
Ada-RDA	-eta	learning rate
	-delta	parameter to ensure positive-definite property of the adaptive weighting matrix
AROW-DA	-r	parameter of passive-aggressive update trade-off
AROW-FS	-k	no default
SCW	-phi	probability parameter in SCW
	-r	parameter of passive-aggressive update trade-off
SCW-RDA	-phi	probability parameter in SCW
	-r	parameter of passive-aggressive update trade-off

### 3.3 Use Command Line

In this section, we show how to use the command line from terminals. We provide an example to show how to use SOL and explain the details of how SOL works. The dataset we use is rcv1.

Command for training is as the following.



```
./SOL -i data/rcv1/rcv1.train -c data/rcv1/rcv1.train_cache
      -t data/rcv1/rcv1.test -tc data/rcv1/rcv1.test_cache
```

For meaning of the input parameters, please refer to the explanation above. Output of the above command will be:

```
Algorithm: SGD

Learning Rate: 10
Initial t : 1
Power t : 0.5
lambda : 0

Iterate No.          Error Rate
2                    0.500000
4                    0.500000
8                    0.500000
16                   0.437500
32                   0.531250
64                   0.515625
128                  0.515625
256                  0.417969
512                  0.341797
1024                 0.252930
2048                 0.182617
4096                 0.148193
8192                 0.122681
16384                0.102661
32768                0.090027
65536                0.079849
131072               0.072670
262144               0.066559
524288               0.061661
data number: 781265
Learn error rate: 5.97 +/- 0.00 %
Test error rate: 5.70 %
Sparsification Rate: 19.65 %
Learning time: 21.154 s
Test time: 0.530 s
```

### Illustrations:

- *Algorithm*: this line is name of the algorithm applied to the dataset. By default, the algorithm is SGD. Select different algorithm by *-opt <algorithm name>*.
- *Learning Rate*:  $\eta_0$  in equation (3). Change this value by *-eta <val>*.
- *Power t*:  $p$  in equation (3). Change this value by *-power\_t <val>*.
- *lambda*:  $\lambda$  in equation (2). Change this value by *-l1 <val>*. Note that this value will have no effect on SGD.
- *data number*: number of samples in training data.
- *Learn error rate*: learning error rate at the last iteration.
- *Test error rate*: test error rate on the test data.
- *Sparsification rate*: ratio of zero elements in the learned model.
- *Learning time*: time to train the model
- *Test time*: time to test the model.

To show how  $\lambda$  will effect sparsity and accuracy, we set the algorithm to STG, and change the value of  $\lambda$  as table 3.3 shows. The command line is:

```
./SOL -i data/rcv1/rcv1.train -c data/rcv1/rcv1.train_cache
      -t data/rcv1/rcv1.test -tc data/rcv1/rcv1.test_cache
      -opt STG -l1 lambda
```

$\lambda$	Learning error rate	test error rate	sparsity
0	5.97	5.7	19.65
1e-8	5.97	5.69	19.7
1e-7	5.98	5.7	20.09
1e-6	5.98	5.7	30.09
1e-5	6.1	5.84	67.56
1e-4	7.23	7.24	90.41
1e-3	15.54	18.34	97.71
1e-2	41.7	42.65	99.85
1e-1	47.51	46.59	99.96
1	47.51	46.59	99.96

Table 3: Effect of sparsity on learning error rate and test error rate

### 3.4 Dynamic & Static Library Interface

Interfaces of dynamic/static library are comprised of three parts: initialization, training/testing, and finalization.

#### Initialization of IO

Initialization of IO means to create a object of dataset. Users can use path to a libsvm data file to initialize a dataset (the “*sol\_init\_dataset*”). Or users can customize a reader and use it as the parameter (“*sol\_init\_dataset2*” and “*sol\_init\_dataset3*”). The parameter “*cache\_filename*” is for cached data files. The program will cache the dataset to the specified file, or the program will use the cached data if it already exist. In case when users do not want to cached files, set it to an empty string “”.

```
long sol_init_dataset (const char* filename, const char* cache_filename,
                     const char* dt_type, int passNum, int buf_size);
long sol_init_dataset2(long dt_reader, int passNum, int buf_size);
long sol_init_dataset3(long dt_reader, const char* cache_filename,
                     int passNum, int buf_size);
```

Note that the type “*long*” here is actually a pointer.

#### Initialization of Loss Functions

Loss functions can either be initialized from the library, or be user-customized. The api provided by the library is

```
long sol_init_loss(const char* loss_type);
```

“*loss\_type*” can be “Hinge”, “Logit”, or “SquaredHinge”.

User-customized loss functions can also be used, as long as it is inherited from the base Class “*LossFunction*”. Please refer to section 4.2 and section 4.6 for more details about how to implement customized loss functions.

#### Initialization of Optimizers

```
long sol_init_optimizer(long dataset, long loss_func,
                      int argc, const char** args);
```

“*dataset*”, “*loss\_func*” are initialized as mentioned above. As the third and fourth parameters, it is the same as mentioned in **Command Line**.

#### Training/Testing

Parameters to train or test the model is simply the initialized optimizer. Meaning of the rest parameters are:

```
void sol_train( long optimizer, float* l_errRate, float* var_errRate,
float* sparse_rate, float* time_cost);
void sol_test ( long optimizer, long test_dataset,
float* t_errRate, float* time_cost);
```

- *l\_errRate*: learning error rate
- *var\_errRate*: variance learning error rate, not valid at the moment
- *sparse\_rate*: sparsification rate of the trained model
- *t\_errRate*: test error rate
- *time\_cost*: time cost of training/testing
- *test\_dataset*: test dataset, initialized the same as mentioned above.

## Finalization

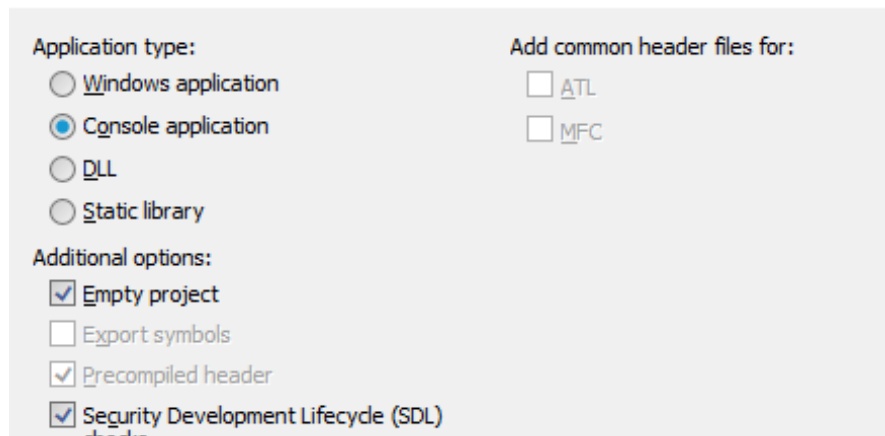
Finalization is to release the initialized resources. The functions are:

```
void sol_release_dataset(long dataset);
void sol_release_loss(long loss_func);
void sol_release_optimizer(long optimizer);
```

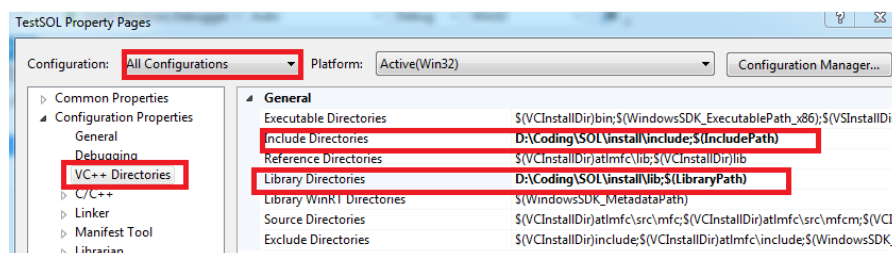
## 3.5 Use Dynamic & Static Library on Windows

SOL can be linked as a dynamic library. It is named as “SOLdll.dll” on Windows. Interface of the library is in “src/SOL\_interface.h”. Before going on, please make sure the directory “<sol\_install\_path>/bin” is in your system environment path. Otherwise, you may encounter the “dll not found” error. In the following, I will show an example to use SOL as dynamic or static libraries on Visual Studio 2012.

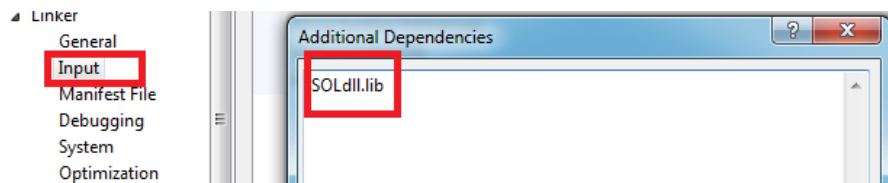
1. Create an empty Win32 C/C++ project.



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 *SOLdll.lib* for dynamic library or *SOLstatic.lib* for static library. In *Debug* mode, add *SOLdll.d* for dynamic library or *SOLstaticd.lib* for static library.



4. Test the program. Add a new source file and paste the following code to the file. You can copy the code from *test/testDll/testDll.cpp*. Note that you may need to change the paths to training/test file.

```
#include "SOLdll.h"

#include <iostream>
#include <cstring>
#include <vector>
#include <cstdio>

using namespace std;
void release(long &dataset){
    sol_release_dataset(dataset);
    dataset = 0;
}
void release(long &dataset, long &loss_func){
    sol_release_dataset(dataset);
    sol_release_loss(loss_func);
    dataset = loss_func = 0;
}
void release(long &dataset, long &loss_func, long &opti){
    sol_release_dataset(dataset);
    sol_release_loss(loss_func);
    sol_release_optimizer(opti);
    dataset = loss_func = opti = 0;
}

int main(int argc, const char** args){
    printf("example: testDll [train_file test_file]\n");
    const char* train_file = "../data/a7a/a7a";
    const char* test_file = "../data/a7a/a7a.t";
    if(argc == 3){
        train_file = args[1];
        test_file = args[2];
    }

    vector<const char*> args_vec;
    args_vec.push_back("-opt");
    args_vec.push_back("AROW");

    long dataset = sol_init_dataset(train_file, "", "libsvm", -1, -1);
    if (dataset == 0) {
        release(dataset);
        return -1;
    }
    long loss_func = sol_init_loss("hinge");
    if (loss_func == 0){
        release(dataset, loss_func);
        return -1;
    }
    long opti = sol_init_optimizer(dataset, loss_func,
```

```

        args_vec.size(), &(args_vec[0]));
    if (opti == 0){
        release(dataset, loss_func, opti);
        return -1;
    }

    float l_err(0), var_err(0), sparse_rate(0), time_cost(0);
    sol_train(opti,&l_err, &var_err, &sparse_rate, &time_cost);

    long t_dataset = sol_init_dataset(test_file, "", "libsvm", -1, -1);
    if (t_dataset != 0){
        float t_errRate(0), t_cost(0);
        sol_test(opti, t_dataset, &t_errRate, &t_cost);
        sol_release_dataset(t_dataset);
        printf("Learn error rate: %.2f +/- %.2f %%\n",
            l_err * 100, var_err * 100);
        printf("Test error rate: %.2f %%\n", t_errRate * 100);
        printf("Sparsification Rate: %.2f %%\n", sparse_rate * 100);
        printf("Learning time: %.3f s\n", time_cost);
        printf("Test time: %.3f s\n", t_cost);
    }
    else{
        printf("Learn error rate: %.2f +/- %.2f %%\n",
            l_err * 100, var_err * 100);
        printf("Sparsification Rate: %.2f %%\n", sparse_rate * 100);
        printf("Learning time: %.3f s\n", time_cost);
    }
    release(dataset, loss_func, opti);
    return 0;
}

```

### 3.6 Use Dynamic & Static Library on Linux

On linux, users need to set the include path, library link path, and linked libraries in the *makefile*.

1. Include path:

```
-I <sol install path>/include
```

2. Link path and library

#### Static Library

```
-L <sol install path>/lib -lsolstatic
```

#### Dynamic Library

```
-L <sol install path>/bin -lsoldll
```

### 3.7 Use Scripts in experiments

In the “*exp*” folder, there are some scripts to help compare different algorithms.

#### Required Packages

- Python 2.7
- MSYS or Cygwin(on Windows), must in the environment path
- *exe\_path.py*

This file defines the path to executables. Users may need to change the path according to their installation.

- *dataset.py*

This file defines paths to dataset. Users can add and delete datasets as needed in function “*get\_file\_name*” and “*get\_model\_param*”. Note the “*rootDir*” at the beginning of the file. Users need to change it to the root directory of datasets.

- *CV.py*

This script is used to do cross validation. Usage of the script is:

```
python CV.py dataset_name algorithm fold_num param start:step:end
```

**Parameters:**

```
dataset_name      : name of the dataset defined in dataset.py
algorithm         : name of the algorithm
fold_num         : number of folds to do cross validation
param            : parameter to be cross validated. Note that this
                  takes the form “-eta” as command line.
start:step:end    : search space for the parameter.
```

**Notes:**

- *param* and *start:step:end* can appear more than one time. For example:

```
python CV.py rcv1 Ada-FOBOS 10 -eta 0.5:2:256 -delta 0.03125:2:32
```

- Users may need to set the extra command in *CV.py*. Currently, it is:

```
-loss Hinge -norm
```

- *batch\_cv.py*

Scripts to do cross validation in batch. What users need to do is to set the algorithm list “*opt\_list*”, dataset list “*ds\_list*”, and folder number “*fold\_num*”.

- *l1\_def.py*

This file defines different sparse regularization parameters to different dataset and to different algorithms. Users may need to modify these files to get a full view of how sparsity will effect accuracy.

- *demo.py*

This script evaluates how sparsity will affect accuracy. Users need to set the algorithm list “*opt\_list*”, dataset list “*ds\_list*”, number of times to randomize the dataset. Advanced users can check other settings.

- *draw.m*

Matlab scripts to draw how test error rate is affected by sparsity.

- *draw\_convergence.m*

Matlab scripts to draw how convergence with iterations.

- *draw\_time.m*

Matlab scripts to draw how training time is affected by sparsity.

In the following, we show how sparsity will affect test error rate, convergence , and training time for different algorithms. First, we obtain the parameters by cross validation. Parameter setting in *batch\_cv.py* is as the following shows:

```
opt_list = [ 'RDA', 'Ada-RDA', 'CW-RDA' ]
ds_list = [ 'rcv1' ]
fold_num = 5

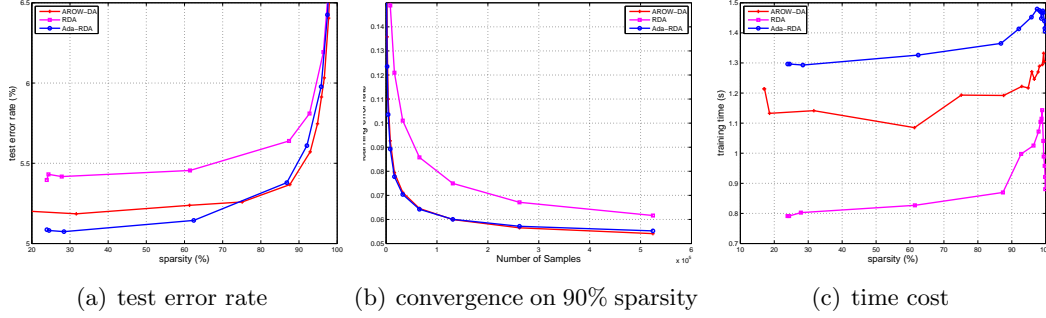
eta_search = '0.5:2.0:512'
delta_search = '0.03125:2:32'
r_search = delta_search
```

Then we evaluate effect of sparsity.

```
opt_list = ['RDA', 'Ada-RDA', 'CW-RDA']
ds_list = ['rcv1']
#number of times to randomize a dataset for averaged results
rand_num = 10
#extra command sent to SOL
extra_cmd = ' -loss Hinge -norm '
```

Results are as the following shows:

Figure 1: Example of effect of sparsity



### 3.8 Parameter Settings

Setting proper parameters plays a nontrivial role in affecting the empirical performance of different sparse online learning algorithms. Table 4 gives a summary of parameters and their default settings by different sparse online learning algorithms. Note that we do not include sparsity parameter  $\lambda$  in this table.

To enable fair side-by-side comparisons between different algorithms, there are two choices:

- Enable the learn best parameter setting “-lbp” in the command line. This scheme selects best parameters one after another if there are more than one parameters.
- Use the provided scripts to do cross validation.

Table 4: Comparison of sparse online learning algorithms

Algorithm	Norm	Type	Parameters
SGD	NA	First Order	$\eta = 10$
FOBOS	L1	First Order	$\eta = 10$
STG	L1	First Order	$\eta = 10, k = 10$
ADA-FOBOS	L1	Second Order	$\eta = 10, \delta = 10$
AROW-TG	L1	Second Order	$r = 1$
RDA	L1	First Order	$\eta = 10$
ADA-RDA	L1	Second Order	$\eta = 10, \delta = 10$
AROW-DA	L1	Second Order	$r = 1$
AROW-FS	L0	Second Order	$k$ , no default

## 4 System Framework

The design principle is to keep the package simple, easy to read and extend. All codes follow the C++99 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. We choose python to compare different algorithms as it is more easy to read and it does not affect the performance of algorithms themselves.

In general, SOL is written in a modular way, in which one can easily develop a new algorithm and make side-by-side comparisons with the existing ones in the package. Thus, we hope that SOL is not only a machine learning tool, but also a comprehensive experimental platform for conducting sparse online learning research.

Figure 2 shows the framework of the system.

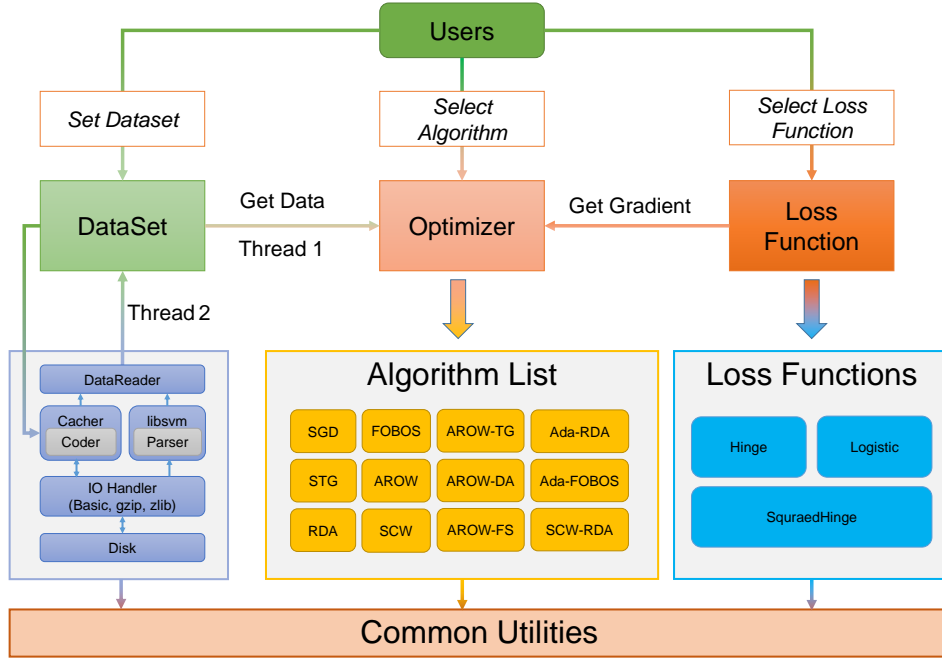


Figure 2: Framework of SOL

## 4.1 DataSet

DataSet is in charge of transferring data from disk to optimizers efficiently. The three major functions are: parsing the original dataset, caching data, and a common interface with optimizers.

### Loading Data

It requires dataset readers to parse different formats of data. By default, we support *libsvm* and the cached *binary* format. Base class to parse a dataset is *DataReader* (in *DataReader.h*), in which we define the interfaces to load a dataset file correctly. The interfaces are:

- *OpenReading*:

```
virtual bool OpenReading() = 0;
```

Open a dataset file to load data. Note that we do not specify the source of dataset (like a file path name). It is specified in the constructor to support diverse data sources. Source of data can be files, TCP, etc. The open operation can go beyond opening a local file. It can also open a socket listener.



The function returns true if everything is ok.

- *GetNextData*:

```
virtual bool GetNextData(DataPoint<FeatType, LabelType> &data) = 0;
```

Get a new data point from the source. The parameter is the variable to place the obtained data.

The function returns true if everything is ok.

- *Rewind*:

```
virtual void Rewind() = 0;
```

Rewind the data source to the beginning

- *Close*:

```
virtual void Close() = 0;
```

Close the data source when loading is finished.

- *Good*:

```
virtual bool Good() = 0;
```

Test if the data reader is ok. Note that end of file(eof) is **NOT** regared as **Bad** status.

## 4.2 Loss Function

At the moment, we provide a base class (purely virtual class) for loss functions and three inherited classes(*HingleLoss*, *Logit*, and *SquaredHinge*). The interfaces are:

- *IsCorrect*:

```
virtual inline bool IsCorrect(LabelType label, float predict);
```

This function is implemented in the base class to justify whether a prediction is correct for binary classification problems. We assign the virtual property to it for the extensibility to multi-class or regression problems.

- *GetLoss*:

```
virtual float GetLoss(LabelType label, float predict) = 0;
```

Get the loss of the current prediction

- *GetGradient*:

```
virtual float GetGradient(LabelType label, float predict) = 0;
```

Get the gradient of the loss function at the current data point. Note that we do not calculate the exact gradient here. 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.

## 4.3 Optimizers

Optimizers are the sparse online learning algorithms. The base class Optimizer implements the details of how a classification model works, including interacting with a dataset, training the model, updating the model, test the model, and some other auxiliary functions. It also define the interfaces for different learning algorithms(the virtual functions).

### Details of Optimizer class

#### 1. Variables:

```
eta0    : initial learning rate
eta      : learning rates. This variable is set to eta0 at beginning.
Different algorithms can set the value on the fly.
id_str   : a string to describe the algorithm, need to be assigned a value
when an algorithm is constructed
lambda   : The L1 regularization parameter.
dataSet  : object of training data.
lossFunc : user specified loss function
weightVec : The weight vector of the linear model.
weightDim : current dimension of the weight vector. Note that
this variable is changing with training data coming in.
initial_t : initial iteration number, this is the variable to avoid
large learning rates at the beginning
curlterNum : current iteration number
sparse_soft_thresh : threshold below which a weight is regarded as zero.
```

#### 2. Constructor:

```
Optimizer(DataSet<FeatType, LabelType> &dataSet,
LossFunc<FeatType, LabelType> &lossFunc);
```

When initializing an optimizer, users need to assign the data set and the loss function.

#### 3. Training:

Following are the functions to learn a model.

- **Learn:** public function called by users to learn a model.

```
float Learn(int numOfTimes = 1);
```

Learn a model and return the average error rate. Note that the input parameter is only used for those dataset that can be randomized, which is not available at the moment.

```
float Learn(float &aveErrRate, float &varErrRate,
float& sparseRate, int numOfTimes = 1);
```

Learn a model and return the average error rate.

#### Parameters:

```
aveErrRate: average error rate
varErrRate: variance of the average error rate,
only valid when dataset can be randomized
sparesRate: sparsification rate of the linear model
numOfTimes: number of times to learn the model with
randomized dataset, not available at the moment
```

- **BeginTrain:**

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

Reset the optimizer to the initialization status of training

**Note:** If user-customized algorithm contains some new parameters that need to be reset, users should call this base function explicitly in their inherited function to ensure the model is reset correctly.

- **Train:**

```
float Train();
```

Train the model and return the learning error rate.

- **EndTrain:**

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

Called when training is finished.

- **Predict:**

```
float Predict(DataPoint<FeatType, LabelType> &data);
```

Predict the label of the input data point.

- **Update model:**

```
virtual float UpdateWeightVec(  
const DataPoint<FeatType, LabelType> &x) = 0;
```

The core function of learning. This function is called each time a new data point comes to update the model.

**x:** the new data that comes in

The function returns prediction of the input data **x**.

#### 4. Test

```
float Test(DataSet<FeatType, LabelType> &testSet);
```

Test the performance of the given dataset. Return the test error rate.

#### 5. Auxiliary Function

- **Set Parameters:**

```
void SetParameter(float lambda = -1, float eta0 = -1);
```

Set the learning rate and L1 regularization parameter. -1 means no change.

- **Learn best parameter:**

```
virtual void BestParameter();
```

Learn the best learning rate by default. It can be inherited and learn other parameters to satisfy the requirements of different algorithms.

- **get sparse rate:**

```
float GetSparseRate(int total_len = 0);
```

Get the sparse rate of the model. The total\_len is the dimension of the input data. If not assigned by users, the largest index of features will be used.

- **update dimension of data:**

```
virtual void UpdateWeigthSize(int newDim);
```

Update the dimension of the weight vector. As we are learning on sparse data online, we do not know the dimension of the input data. So the weight vector needs to be resized on the fly. Note that inherited algorithms need to overridden this function to resize their own dimension-related members and call the base one explicitly to resize the weight vector.

- **print algorithm info**

```
void PrintOptInfo() const;
```

Print the optimization information.

- **Id\_Str**

```
const string& Id_Str() const;
```

Get the identity string of the optimizer.

## 4.4 How to Add New Algorithms

A salient property of this library is that it provides a fairly easy-to-use testbed to facilitate sparse 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 two 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., ( $l_t > 0$ ).
- How to perform the update on the classifier (i.e., the weight vector  $\mathbf{w}$ ) whenever the condition is satisfied? For example, Perceptron updates  $w = w + y_t * x_t$ ;
- Are there some parameters in your new algorithm? If so, you need to do some initializations, including (i) modify the "init\_params.h" file by adding the initialization; (ii) override the "BestParameter.h" to learn best parameter as needed.

## 4.5 Extend DataReader

For a specific format of data, we only need to inherit from the *DataReader* class and implement the above interfaces. It will work when you assign the customized data reader to the dataset.

The file *libsvmread.h* and *MNISTReader.h* (in "testMNIST" folder) can be regarded as an example to extend *DataReader*.

## 4.6 Extend Loss Functions

To implement a new loss function, users only need to inherit from the class "*LossFunction*", and implement the pure virtual functions. The files *HingeLoss.h*, *LogisticLoss.h*, and *SquareHingeLoss.h* are three examples to extend the base class.

## 4.7 Implement your own algorithms

To implement a specific learning algorithm, users only need to inherit from the base class *Optimizer*, and implement the pure virtual function *UpdateWeightVec*. Whether other virtual functions need to be override depends on the specific algorithm. Take *STG* for example, it has to maintain a time stamp vector. So it override the *BeginTrain* and *UpdateWeightSize* functions to initialize and resize the time stamp vector. It needs to shrink weight vectors at the end of the training. So *EndTrain* is overridden.

Check the implemented algorithms to explore more details of how to extend the optimizers.

# 5 Documentation

The SOL package comes with comprehensive documentation. The README file describes the setup and usage. Users can read the "Quick Start" section to begin shortly. All the functions and related data structures are explained in detail. If the README file does not give the information users want, they can also check this document and the software manual.

## 6 Conclusions, Revision, and Citation

SOL is an easy-to-use open source package for efficient and scalable sparse online linear classification. It is currently one of comprehensive online learning software packages that include the largest number of diverse sparse online learning algorithms for sparse online classification. SOL is still being improved by improvements from practical users and new research results. The ultimate goal is to make easy learning with massive-scale data streams to tackle the emerging grand challenge of big data mining.

### Revision History

- Version 0.1.0 was released on January 10th, 2014. This version mainly includes C++ implementation for binary classification.

Welcome to send us your suggestions/corrections for SOL by the following email:

`chhoi@ntu.edu.sg`

### Citation

In citing SOL in your papers, please use the following reference:

### References

- [1] K. Crammer, A. Kulesza, and M. Dredze. Adaptive regularization of weight vectors. *Machine Learning*, pages 1–33, 2009.
- [2] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 999999:2121–2159, 2011.
- [3] J. Duchi and Y. Singer. Efficient online and batch learning using forward backward splitting. *The Journal of Machine Learning Research*, 10:2899–2934, 2009.
- [4] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. *The Journal of Machine Learning Research*, 10:777–801, 2009.
- [5] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [6] J. Wang, P. Zhao, and S. C. Hoi. Exact soft confidence-weighted learning. *arXiv preprint arXiv:1206.4612*, 2012.
- [7] L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of Machine Learning Research*, 9999:2543–2596, 2010.