ThunderSVM Documentation

Release 0.1

Zeyi Wen

Jan 22, 2019

Contents

The mission of ThunderSVM is to help users easily and efficiently apply SVMs to solve problems. ThunderSVM exploits GPUs and multi-core CPUs to achieve high efficiency. Key features of ThunderSVM are as follows.

- Support all functionalities of LibSVM such as one-class SVMs, SVC, SVR and probabilistic SVMs.
- Use same command line options as LibSVM.
- Support Python, R and Matlab interfaces.
- Supported Operating Systems: Linux, Windows and MacOS.

Why accelerate SVMs: A survey conducted by Kaggle in 2017 shows that 26% of the data mining and machine learning practitioners are users of SVMs.

CHAPTER 1

More information about ThunderSVM

1.1 Getting Started with ThunderSVM

Here we provide a quick start tutorial for users to install ThunderSVM.

1.1.1 Prerequisites

- `cmake` 2.8 or above
- `gcc` 4.8 or above for Linux and MacOS
- `Visual C++` for Windows

If you want to use GPUs, you also need to install CUDA.

- CUDA 7.5 or above

1.1.2 Installation

If you don't have GPUs, please go to *Working without GPUs* in later section of this page.

Installation for Linux

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
```

- Build ThunderSVM

```
mkdir build
cd build
cmake ..
make -j
```

If `make -j` doesn't work, please use `make` instead.

Installation for MacOS

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
```

You need to Install `CMake` and `gcc` for MacOS. If you don't have Homebrew, here is its website.

```
brew install gcc
brew install cmake
```

- Build ThunderSVM. You can specify gcc as the compiler of cmake. ([path_to_g++] and [path_to_gcc] typically look like `/usr/local/bin/g++-7` and `/usr/local/bin/gcc-7`, respectively.).

```
# in thundersvm root directory
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=[path_to_g++] -DCMAKE_C_COMPILER=[path_to_gcc] -DUSE_
↪CUDA=ON -DUSE_EIGEN=OFF ..
make -j
```

Installation for Windows

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
```

- Create a Visual Studio project

```
mkdir build
cd build
cmake .. -DCMAKE_WINDOWS_EXPORT_ALL_SYMBOLS=TRUE -DBUILD_SHARED_LIBS=TRUE -G "Visual␣
↪Studio 14 2015 Win64"
```

You need to change the Visual Studio version if you are using a different version of Visual Studio. Visual Studio can be downloaded from this link. The above commands generate some Visual Studio project files, open the Visual Studio project to build ThunderSVM. Please note that CMake should be 3.4 or above for Windows.

Working without GPUs

If you don't have GPUs, ThunderSVM can run purely on CPUs. The number of CPU cores to use can be specified by the `-o` option (e.g., `-o 10`), and refer to Parameters for more information.

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
```

- Get Eigen Library. ThunderSVM uses Eigen for matrix calculation. To use Eigen, just initialize the submodule.

```
# in thundersvm root directory
git submodule init eigen && git submodule update
```

- Build without GPUs for Linux

```
# in thundersvm root directory
mkdir build && cd build && cmake -DUSE_CUDA=OFF -DUSE_EIGEN=ON .. && make -j
```

If `make -j` doesn't work, please simply use `make`. Now ThunderSVM will work solely on CPUs and does not rely on CUDA.

- Build without GPUs for MacOS

```
# in thundersvm root directory
mkdir build && cd build && cmake -DCMAKE_CXX_COMPILER=[path_to_g++] -DCMAKE_C_
↪COMPILER=[path_to_gcc] -DUSE_CUDA=OFF -DUSE_EIGEN=ON .. && make -j
```

- Build without GPUs for Windows

```
mkdir build
cd build
cmake .. -DCMAKE_WINDOWS_EXPORT_ALL_SYMBOLS=TRUE -DBUILD_SHARED_LIBS=TRUE -DUSE_
↪CUDA=OFF -DUSE_EIGEN=ON -G "Visual Studio 14 2015 Win64"
```

Then, you can open the generated the Visual Studio project file to build ThunderSVM.

1.1.3 Training SVMs

We show some concrete examples of using ThunderSVM. ThunderSVM uses the same command line options as LibSVM, so existing users of LibSVM can use ThunderSVM easily. For new users of SVMs, the *Parameters* page provides explanation for the usage of each option.

Training SVMs for Classification

In the following, we provide an example of using ThunderSVM for the MNIST dataset.

- Download the MNIST data set. The data set is available in this link.
- Decompress the data set. For Windows machines, you can decompress the data set using tools such as 7-Zip. For Unix based OSes, you can use

```
bunzip2 mnist.scale.bz2
```

- Install ThunderSVM. Instructions available in the previous sections of this page.
- Run ThunderSVM

```
./thundersvm-train -s 0 -t 2 -g 0.125 -c 10 mnist.scale svm.model
```

The meaning of each option can be found in the *Parameters* page. The training takes a while to complete. Once completed, you can see the classifier accuracy is 94.32%.

Training SVMs for Regression

The usage of other SVM algorithms (such as SVM regression) is similar to the above example. The key difference is the selection of the options. Let's take the `Abalone` data set as an example.

- Download the Abalone data set. The data set is available in this link.
- Install ThunderSVM. Instructions available in the previous sections of this page.

- Run ThunderSVM

```
./thundersvm-train -s 3 -t 2 -g 3.8 -c 1000 abalone_scale svm.model
```

The meaning of each option can be found in the *Parameters* page.

Interfaces

ThunderSVM provides Python, R and Matlab interfaces. You can find the instructions in the corresponding subdirectories on GitHub.

1.2 Introduction

In this page, we present Support Vector Machines (SVMs) and the Sequential Minimumal Optimazation (SMO) solver.

1.2.1 Support Vector Machines (SVMs)

SVMs have been used in various applications including spam filtering, document classification, network attack detection. SVMs have good generalization property via maximizing margin of the seperating hyperplane. We discuss SVMs for binary classification here, because other SVM training problems such as SVM regression and ν-SVMs can be converted into binary SVM training problems. The figure below shows an example of binary SVMs which have an optimal hyperplane seperating the training data set denoted by circles and squares. For non-linearly seperable data, SVMs use the kernel functions to map the data to a higher dimentional data space and also allows errors by introducing slack variables (see the *where* under the optimization problem below).

In the following, we describe the formal defination of SVMs. Specifically, a training instance x_i is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) instance is a training instance with the label of +1 (-1). Given a set $\mathcal{X}$ of n training instances, the goal of training SVMs is to find a hyperplane that separates the positive and the negative instances in $\mathcal{X}$ with the maximum margin and meanwhile, with the minimum misclassification error on the training instances. The training is equivalent to solving the following optimization problem:

$$\underset{\boldsymbol{w},\boldsymbol{\xi},b}{\text{argmin}} \quad \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

$$\text{subject to} \quad y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i - b) \geq 1 - \xi_i$$

$$\xi_i \geq 0,\ \forall i \in \{1, ..., n\}$$

where $\boldsymbol{w}$ is the normal vector of the hyperplane, C is the penalty parameter, $\boldsymbol{\xi}$ is the slack variables to tolerant some training instances falling in the wrong side of the hyperplane, and b is the bias of the hyperplane.

To handle the non-linearly separable data, SVMs use a mapping function to map the training instances from the original data space to a higher dimensional data space where the data may become linearly separable. The optimization problem above can be rewritten to a dual form where the dot products of two mapped training instances can be replaced by a kernel function which avoids explicitly defining the mapping functions as only dot products are involved. The

optimization problem in the dual form is shown as follows.

$$\max_{\boldsymbol{\alpha}} F(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\boldsymbol{\alpha}^{\boldsymbol{T}}\boldsymbol{Q}\boldsymbol{\alpha}$$

subject to

$$0 \leq \alpha_i \leq C, \forall i \in \{1, ..., n\}$$

$$\sum_{i=1}^{n} y_i \alpha_i = 0$$

where $F(\boldsymbol{\alpha})$ is the objective function; $\boldsymbol{\alpha} \in \mathbb{R}^n$ is a weight vector, where α_i denotes the *weight* of $\boldsymbol{x}_i$; C is the penalty parameter; $\boldsymbol{Q}$ is a positive semi-definite matrix, where $\boldsymbol{Q} = [Q_{ij}]$, $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function (e.g., Gaussian kernel, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp\{-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2\}$). All the kernel values together form an $n \times n$ kernel matrix.

1.2.2 Sequential Minimal Optimization (SMO)

The goal of the training is to find a weight vector $\boldsymbol{\alpha}$ that maximizes the value of the objective function $F(\boldsymbol{\alpha})$. Here, we describe a popular training algorithm, the Sequential Minimal Optimization (SMO) algorithm. It iteratively improves the weight vector until the optimal condition of the SVM is met. The optimal condition is reflected by an *optimality indicator vector* $\boldsymbol{f} = \langle f_1, f_2, ..., f_n \rangle$ where f_i is the optimality indicator for the i-th instance $\boldsymbol{x}_i$ and f_i can be obtained using the following equation: $f_i = \sum_{j=1}^{n} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - y_i$. In each iteration, the SMO algorithm has the following three steps:

Step 1: Search two extreme instances, denoted by $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, which have the maximum and minimum optimality indicators, respectively. It has been proven that the indexes of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by u and l respectively, can be computed by the following equations.

$$u = \text{argmin}_i\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{upper}\}$$
$$l = \text{argmax}_i\{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, \boldsymbol{x}_i \in \mathcal{X}_{lower}\}$$

where

$$\mathcal{X}_{upper} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3,$$
$$\mathcal{X}_{lower} = \mathcal{X}_1 \cup \mathcal{X}_4 \cup \mathcal{X}_5;$$

and

$$\mathcal{X}_1 = \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, 0 < \alpha_i < C\},$$
$$\mathcal{X}_2 = \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = 0\},$$
$$\mathcal{X}_3 = \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = C\},$$
$$\mathcal{X}_4 = \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = C\},$$
$$\mathcal{X}_5 = \{\boldsymbol{x}_i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = 0\};$$

and $\eta_i = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_i)$; f_u and f_l denote the optimality indicators of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, respectively.

Step 2: Improve the weights of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by α_u and α_l.

$$\alpha_l' = \alpha_l + \frac{y_l(f_u - f_l)}{\eta}$$
$$\alpha_u' = \alpha_u + y_l y_u(\alpha_l - \alpha_l')$$

where $\eta = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_l, \boldsymbol{x}_l) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_l)$. To guarantee the update is valid, when α_u' or α_l' exceeds the domain of [0, C], α_u' and α_l' are adjusted into the domain.

Step 3: Update the optimality indicators of all the instances. The optimality indicator f_i of the instance $\boldsymbol{x}_i$ is updated to f_i' using the following formula:

$$f_i' = f_i + (\alpha_u' - \alpha_u) y_u K(\boldsymbol{x}_u, \boldsymbol{x}_i) + (\alpha_l' - \alpha_l) y_l K(\boldsymbol{x}_l, \boldsymbol{x}_i)$$

SMO repeats the above steps until the optimal condition is met, i.e., $f_u \geq f_{max}$, where

$$f_{max} = max\{f_i | \boldsymbol{x}_i \in \mathcal{X}_{lower}\}$$

After the optimal condition is met, we obtain the $\boldsymbol{\alpha}$ values which corresponding to the optimal hyperplane and the SVM with these $\boldsymbol{\alpha}$ values is considered trained.

Prediction

After the training, the trained SVM is used to predict the labels of unseen instances. The label of an instance $\boldsymbol{x}_j$, denoted by y_j, is predicted by the following formula:

$$y_j = \text{sgn}(\sum_{i=1}^{n} y_i \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_j) + b)$$

where b is the bias of the hyperplane of the trained SVM. The training instances with their weights greater than zero are called *support vectors*, which are used to predict the labels of unseen instances.

1.2.3 Other SVM Training Problems

The other SVM training problems implemented in ThunderSVM can be modeled as binary SVM training problems. More specifically,

- SVM regression: The SVM regression training problem can be modeled as a binary SVM training problem, where each instance in the data set is duplicated, such that each instance is associated with two new training instances: one with label of +1 and the other with label of -1.
- Multi-class SVM classification: The multi-class SVM classsification problem can be decomposed into a few binary SVM training problems via pair-wise coupling (also known as one-vs-one decomposition).
- Probabilistic SVMs: Training probabilistic SVMs can be modeled as training binary SVMs and then using the decision values of the binary SVMs to fit a sigmoid function in order to obtain probabilities.
- ν-SVMs: Training ν-SVMs is also very similar to training binary SVMs. The key difference is that instead of using two training instances to improve the currently trained model, ν-SVMs use four training instances. Training ν-SVMs for regression (ν-SVR) is similar to training traditional SVMs for regression.

1.3 ThunderSVM Parameters

This page is for parameter specification in ThunderSVM. The parameters used in ThunderSVM are identical to LibSVM (except some newly introduced parameters), so existing LibSVM users can easily get used to ThunderSVM.

The command line options for ThunderSVM are shown below.

- -s: set the type of SVMs (default=0)
  - 0 – C-SVC

  - 1 – ν-SVC
  - 2 – one-class SVMs
  - 3 – ϵ-SVR
  - 4 – ν-SVR
- -t: set the type of kernel function (default=2)
  - 0 – linear: $\boldsymbol{x}_i^T \cdot \boldsymbol{x}_j$
  - 1 – polynomial: $(\gamma \boldsymbol{x}_i^T \cdot \boldsymbol{x}_j + r)^d$
  - 2 – radial basis function (RBF): $\exp(-\gamma ||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$
  - 3 – sigmoid: $tanh(\gamma \boldsymbol{x}_i^T \cdot \boldsymbol{x}_j + r)$
- -d: set the degree in kernel function (default=3)

- -g: set γ in kernel function (default=$\frac{1}{}$

The options in italic are not applicable for GPUs, and the alternative optimizations are implemented with automatically setting working set size.

1.4 ThunderSVM How To

This page is for key instructions of intalling, using and contributing to ThunderSVM. ThunderSVM has been used by many users, and everyone can contribute to ThunderSVM to make it better.

1.4.1 Install ThunderSVM

To use ThunderSVM, we need to install the following software:

- `cmake` 2.8 or above
- `gcc` 4.8 or above for Linux and MacOS; `Visual C++` for Windows

If you want to use GPUs, you also need to install CUDA.

- CUDA 7.5 or above

After installing the above software, you can start compiling ThunderSVM.

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
```

If you don't have a GPU or want to run ThunderSVM purely on CPUs, you need to add a submodule to ThunderSVM by the following command. Please refer to Working without GPUs for more information.

```
# in thundersvm root directory
git submodule init eigen && git submodule update
```

- Build the executable for Linux, use the following commands.

```
cd thundersvm
mkdir build
cd build
cmake ..
make -j
```

If `make -j` doesn't work, please simply use `make` instead.

- Build for MacOS, use the following commands (`[path_to_g++]` and `[path_to_gcc]` typically look like `/usr/local/bin/g++-7` and `/usr/local/bin/gcc-7`, respectively.).

```
# in thundersvm root directory
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=[path_to_g++] -DCMAKE_C_COMPILER=[path_to_gcc] -DUSE_
↪CUDA=ON -DUSE_EIGEN=OFF ..
make -j
```

- Build the executable for Windows, use the following example commands. You need to change the Visual Studio version if you are using a different version of Visual Studio.

```
mkdir build
cd build
cmake ..  -DCMAKE_WINDOWS_EXPORT_ALL_SYMBOLS=TRUE -DBUILD_SHARED_LIBS=TRUE -G "Visual␣
↪Studio 14 2015 Win64"
```

The above commands generate some Visual Studio project files, open the Visual Studio project in the `build` directory to start building ThunderSVM on Windows. Please note that CMake should be 3.4 or above for Windows.

You can now use ThunderSVM, and the options of executing ThunderSVM are available here. Please refer to Getting Started for some examples of training SVMs using ThunderSVM.

1.4.2 How can I do grid search?

Since ThunderSVM supports cross-validation, you can write a simple grid.sh script like the following one. Then run `sh grid.sh [dataset]`. You may modify the script to meet your needs. Indeed, ThunderSVM supports the same command line parameters as LIBSVM. So the script grid.py in LIBSVM can be used for ThunderSVM with minor modifications.

```
#!/usr/bin/bash
DATASET=$1
OPTIONS=
N_FOLD=5
for c in 1 3 10 30 100
do
  for g in 0.1 0.3 1 3 10
    do
      bin/thundersvm-train -c ${c} -g ${g} -v ${N_FOLD} ${OPTIONS} ${DATASET}
    done
done
```

1.4.3 Improve documentations

Most of the documents can be viewed on GitHub, although the documents look much better in Read the Doc. The HTML files of our documents are generated by Sphinx, and the source files of the documents are written using Markdown. In the following, we describe how to setup the Sphinx environment for ThunderSVM.

- Install sphinx

```
pip install sphinx
```

- Install Makedown Parser

```
pip install recommonmark
```

Note that `recommonmark` has a bug when working with Sphinx in some platforms, so you may need to hack into transform.py to fix the problem by yourself. You can find the instruction of hacking in this link.

- Install Sphinx theme

```
pip install sphinx_rtd_theme
```

- Generate HTML

  Go to the "docs" directory of ThunderSVM and run:

```
make html
```

At this point, make sure you have generated the documents of ThunderSVM. You can build the documents in your machine to see the outcome.

1.4.4 Contribute to ThunderSVM

You need to fetch the latest version of ThunderSVM before submitting a pull request.

```
git remote add upstream https://github.com/zeyiwen/thundersvm.git
git fetch upstream
git rebase upstream/master
```

Test ThunderSVM

We recommend our contributors using Linux as the development platform where ThunderSVM is relatively well tested.

Please note that `cmake .. [-D<options>=<args>]` produces a `CMakeCache.txt` file. You need to remove it or the whole `build` directory, before you can run a new `cmake` command with different configurations. Also please note that the test build will not generate binaries like `thundersvm-train`.

Build test on Linux

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
git submodule update --init src/test/googletest
```

- Build the binary for testing

```
mkdir build
cd build
cmake .. -DBUILD_TESTS=ON
make -j runtest
```

If `make -j runtest` doesn't work, please use `make runtest` instead. Make sure all the test cases pass. You can also add more test cases in the test module of ThunderSVM.

Build test on MacOS

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
git submodule init eigen && git submodule update
git submodule update --init src/test/googletest
```

- Build the binary for testing

```
mkdir build
cd build
cmake .. -DCMAKE_CXX_COMPILER=[path_to_g++]/usr/local/bin/g++-7 -DCMAKE_C_
↪COMPILER=[path_to_gcc] -DUSE_CUDA=OFF -DUSE_EIGEN=ON -DBUILD_TESTS=ON
make -j runtest
```

Where `[path_to_g++]` and `[path_to_gcc]` typically look like `/usr/local/bin/g++-7` and `/usr/local/bin/gcc-7`, respectively.

Build test on Windows

- Clone ThunderSVM repository

```
git clone https://github.com/zeyiwen/thundersvm.git
git submodule update --init src/test/googletest
```

- Build the binary for testing

```
mkdir build
cd build
cmake .. -DCMAKE_WINDOWS_EXPORT_ALL_SYMBOLS=TRUE -DBUILD_SHARED_LIBS=TRUE -DBUILD_
↪TESTS=ON -G "Visual Studio 14 2015 Win64"
cmake --build . --target runtest
```

You need to change the Visual Studio version if you are using a different version of Visual Studio. Visual Studio can be downloaded from this link. The above commands generate some Visual Studio project files, open the Visual Studio project to build ThunderSVM. Please note that CMake should be 3.4 or above for Windows.

1.5 Frequently Asked Questions (FAQs)

This page is dedicated to summarizing some frequently asked questions about ThunderSVM.

1.5.1 FAQs of users

- **How can I use the source code?** Please refere to *How To* page.
- **What is the data format of the input file?** ThunderSVM uses the LibSVM format.
- **Can ThunderSVM run on CPUs?** Yes. Please see *Working without GPUs*.
- **How can I do grid search?** Please refer to *How To* page.

1.5.2 FAQs of developers

- **Why not use shrinking?** Shrinking is used in ThunderSVM and is implemented by the working set size. We don't provide the shrinking option to users, because the traditional way of shrinking is inefficient on GPUs.

1.6 Related websites to ThunderSVMs

Other SVM libraries: