A Generic Tool for Performance Evaluation of Supervised Learning Algorithms

Ron Begleiter
ronbeg@cs.technion.ac.il

Ran El-Yaniv
rani@cs.technion.ac.il

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract

We provide a general purpose Matlab tool for performance evaluation of supervised learning algorithms. Given any existing implementation of a classifier learning algorithm, our Generic-CV tool operates the algorithm, tunes its hyper-parameters and evaluates its performance on any given dataset. The tool is generic in the sense that it easily accommodates new learning algorithms, without any need for programming, and can operate Matlab implementations as well as ‘executables’. In addition, we provide a distributed version of our tool that can utilize multiple computers and in principle can speed up the enormous running time required for performance evaluation by several orders of magnitude.

1. Introduction

Evaluating the performance of a learning algorithm on a dataset is one of the most important and frequent tasks in empirical machine learning research. Up to fairly recently, a common practice was to evaluate learning algorithms using some predefined hyper-parameter values (e.g., using some ‘default’ or other arbitrary values based on ‘prior knowledge’). One reason for this bad practice was lack of sufficient computing power that would allow a more systematic and statistically unbiased search of hyper-parameter space.

The availability of significantly stronger computers paved the way to more systematic hyper-parameter selection techniques that can lead to more objective performance evaluation as well as significantly better performance. Indeed, to date the common standard in experimental machine learning protocols is to apply some type of search routine to seek suitable hyper-parameter values, in a statistically unbiased manner, as far as possible. The cost of such search routines, which are typically based on cross-validation, can be enormous in terms of computation time.

Consider the k-fold cross-validation experimental protocol (see, e.g., Duda et al., 2001, Chapter 9.6.2 and Hastie et al., 2001, Chapter 7.10). A few toolboxes exist that include implementations of the k-fold cross-validation protocol. For example, Weka (Witten and Frank, 1999), SPIDER (Weston et al., 2003), and Elephant (Gawande et al., 2007). However, prospective users have to acquire a fairly deep knowledge in their architecture before adding new learning algorithm implementations into them. Our Generic-CV tool aims to

1. Consider any variation of this protocol.
fill this gap by providing an easy to use plug-and-play classifier assessment solution for the practitioner.

The design of Generic-CV focuses on generality and usability. Generic-CV works with any learning algorithm implementation written in Matlab or alternatively ‘executables’ that are launched via the operating-system’s command-line. Plugging-in a new algorithm implementation into Generic-CV is very easy and does not require any programming knowledge.

In addition, we provide a distributed implementation of Generic-CV for speeding up the runtime consumption of $k$-fold cross-validation. This Distributed Generic-CV is built on top of the Condor high throughput computing platform (Litzkow et al., 1988). Our experiments suggest that Distributed Generic-CV can provide significant runtime speedup.\footnote{We achieved up to an 11.3 speedup factor using scarce lab resources of 30 machines; see Chapter 4 for more details.}

In the next section we briefly discuss the $k$-fold cross-validation protocol. Then we show an example for evaluating the performance of a ready-made classifier implementation using our Generic-CV tool. Finally, the Distributed Generic-CV variant is discussed in Section 4.

2. The k-Fold Cross-Validation Protocol

We consider a classifier learning algorithm $A_\theta$ (e.g., SVM), where $\theta$ is the vector of its hyper-parameters (e.g., for SVM $\theta$ includes the choice of the kernel and its parameters). The output $A_\theta(D)$ of the algorithm, applied on a dataset $D$, is a classifier. Given a set $\Theta$ of possible hyper-parameter assignments, we would like to evaluate the error rate of the best achievable classifier $A_{\theta^*}(D)$ where $\theta^* \in \Theta$ is the best assignment for $D$.

To this end, we apply the following $k$-fold cross-validation procedure $CV(k, A, D, \Theta, n)$, which returns the average error of $k$ classifiers that were learned by the algorithm, each over a different fold and optimized over $\Theta$ using a certain hyper-parameter search strategy. The procedure consists of the following steps:

1. **Initialize data splits.** Randomly split the data into $k$ non-overlapping (roughly) equally sized random subsets $D_i$, $i = 1, \ldots, k$. This is usually done by drawing uniformly at random a single permutation over the indices of the data points.

2. **Train classifiers over folds.** For each $i = 1, \ldots, k$ generate a classifier $C_i = A_{\theta_i}(D \setminus D_i)$, where $\theta_i$ is optimized over $\Theta$ using a search strategy (described below).

3. **Test classifiers and compute statistics.** For each $i = 1, \ldots, k$ apply $C_i$ over all points in $D_i$ and compute its error rate. Calculate and output desired statistics (average error, standard deviation and standard error of the mean).

In Step 2, for each fold we search $\Theta$ for the best $\theta_i$ for this fold. Several strategies can be used for this search but all of them are based on the following validation step. For each candidate $\theta \in \Theta$, calculate the average error obtained by the learning algorithm $A_\theta$ over $n$ data splits (folds) of $D \setminus D_i$. These $n$ splits are generated by partitioning $D \setminus D_i$ uniformly at random to $n$ parts, $V_i$, of (roughly) equal size. The error rates are obtained by training $A_\theta$ over $D \setminus \{D_i \cup S_i\}$ and testing the resulting classifier over $V_i$.\footnote{We achieved up to an 11.3 speedup factor using scarce lab resources of 30 machines; see Chapter 4 for more details.}
We implemented three different search strategies. The first is an exhaustive search of a finite set of candidate hyper-parameter values. The set is typically a (finite) grid over some subspace of all possible assignments. This strategy is called the grid method. The second strategy, called the simplex method, sequentially and adaptively searched for a good \( \theta \) using a gradient technique (see Press et al., 2007, Chapter 10.5 for details). The third strategy is a simple greedy line search applied to each dimension of \( \theta \) independently.

3. Generic-CV: Usage Example

This section explains the essential steps for adding a new learning algorithm into the Generic-CV tool. For example, assume one would like to evaluate the performance of the libsvm SVM implementation (Chang and Lin, 2001). This can be done as follows.\(^3\)

1. Download the libsvm executables into the Generic-CV root directory.
2. Launch the Generic-CV wizard for adding a new classifier.
3. Provide the commands for training a classifier and predicting labels.\(^4\) As a result the wizard will accommodate libsvm within the tool and generate a corresponding script, \texttt{libsvm.cv.m}, for running a cross-validation experiment with libsvm.
4. You are now ready to experiment with libsvm. Just pick a dataset, define the hyper-parameter search space, and launch the \texttt{libsvm.cv} running script.

4. Distributed Implementation for Faster Evaluations

The runtime of \( k \)-fold cross-validation with the hyper-parameter search procedure is (usually) very high and depends heavily on the number of folds \( (k \text{ and } n) \), size of the data \( |D| \), runtime complexity of algorithm \( A \), and size of the hyper-parameter search space \( |\Theta| \). Thus, a distributed execution of the procedure can significantly speed up the running time.

A couple of previous works by Celis and Musicant (2002) and Khoussainov et al. (2004) added some distributed capabilities to the Weka tool. However, both solutions parallelize only the cross-validation folds and use a proprietary distributed computing solution. (Also, it is a Weka enhancement and thus not “generic,” as is our solution.)

We developed a distributed version of the Generic-CV. The Distributed Generic-CV is built on top of the Condor high throughput computing platform (Litzkow et al., 1988).\(^5\) We decided to parallelize the hyper-parameter evaluation steps within the grid search. Usually, the size of the grid \( |\Theta| \) is much larger than the number of folds \( k \). Hence, this choice would potentially significantly accelerate the parallelization of the cross-validation folds.

Figure 1 depicts a runtime comparison of Distributed Generic-CV on a ‘grid’ consisting of 30 computers vs. Generic-CV (which runs on a single computer).\(^6\) Each bar in the figure corresponds to a single run of a 5-fold cross-validation with a 35 hyper-parameter search space.

---

\(3\) See user’s manual for an illustrated example.

\(4\) For example “libsvm_train -b 1 -t 2 -c %f -g %f data model” (where, “%f” indicates any possible value).

\(5\) Condor is a standard free-to-use distributed high throughput computing platform.

\(6\) The dataset is defined in Chapelle et al. (2006) as the TEXT dataset.
combination grid and SVM algorithm on a textual dataset with 1500 instances. Clearly, the runtime of Distributed Generic-CV depends on the amount of available resources (e.g., number of free computers and the load on the network traffic). Thus, we ran the same experiment 10 times in different occasions. Running the same experiment with Generic-CV on a single computer\textsuperscript{7} takes 169 minutes (depicted as a dashed line). Observe that the worst distributed run takes 54 minutes and the fastest 15 minutes. On average Distributed Generic-CV is six times faster than Generic-CV. Clearly, one should expect that a larger grid of computers with fewer users would give more significant speedup.

**References**


\textsuperscript{7} With similar resources as the ones used in the distributed runs.