UNDER REVIEW

RESCIENCE C

Letter / Machine Learning

Eliminating the Variability of Cross-Validation Results with LIBLINEAR due to Randomization and Parallelization

Vladimir Sukhoy[1, ID] and Alexander Stoytchev[1, ID]

[1]Iowa State University, Department of Electrical and Computer Engineering, Ames, IA 50011, USA

Edited by
(Editor)

Received
15 July 2019

Published
–

DOI
–

Cross-validation is the gold standard for evaluating machine learning algorithms or fine-tuning their parameters. The results of this technique, however, are not always reproducible and may depend on the computing platform and the number of parallel threads, especially if the underlying learning algorithm uses a pseudo-random number generator (PRNG). This paper gives a recipe for solving these reproducibility problems and applies it to LIBLINEAR[1], a popular software library that implements randomized learning algorithms based on support vector machines[2]. The proposed approach solves these problems by using a cross-platform PRNG and by making the PRNG state private in each thread. The cross-validation results obtained with the modified version of LIBLINEAR are the same across platforms. Furthermore, the parallelized cross-validation results are no longer affected by random fluctuations arising from the sharing of the PRNG state by parallel threads.

1 Introduction

The reproducibility of machine learning results has been questioned in the top scientific journals[3,4,5,6]. Similar issues have been brought up in the Artificial Intelligence[7,8] and Machine Learning[9] communities. According to Henderson et al.[7], reproducibility problems can be *extrinsic*, i.e., unrelated to the algorithms or their implementations, or *intrinsic*, i.e., directly associated with the algorithms, their implementations, or the computing environments in which they run. This paper describes a technique that eliminates two intrinsic problems that may impede the reproducibility of cross-validation results for randomized learning algorithms. The first problem stems from using a platform-dependent pseudo-random number generator (PRNG). The second problem is due to sharing of the PRNG state by parallel threads.

Cross-validation (CV) is a statistical technique that is often used to evaluate machine learning algorithms or to select their parameters (see Figure 1). It assigns data instances into N folds and then picks K folds for testing and $N-K$ folds for training. This process is repeated R times and the results are averaged. Typically, $K=1$ and $R=N$. Because a PRNG is often used to assign instances to folds, the results may depend on the platform (i.e., OS, programming language, compiler, runtime libraries, etc.). It is straightforward to parallelize CV by distributing the R repetitions over T threads, but the results can be affected by PRNG state sharing. In particular, this is true for algorithms implemented in C or C++ that use the standard function `rand()`. Furthermore, `rand()` may not even be thread-safe (POSIX 7, 2018, p. 1767).

The proposed technique solves the first reproducibility problem by replacing a platform-dependent PRNG with a cross-platform PRNG. In the experiments, we used the SIMD-oriented Fast Mersenne Twister (SFMT) library[11], but the approach should work with

Correspondence should be addressed to Vladimir Sukhoy (sukhoy@iastate.edu)
The authors have declared that no competing interests exists.
Code is available at https://github.com/sukhoy/cvrep.

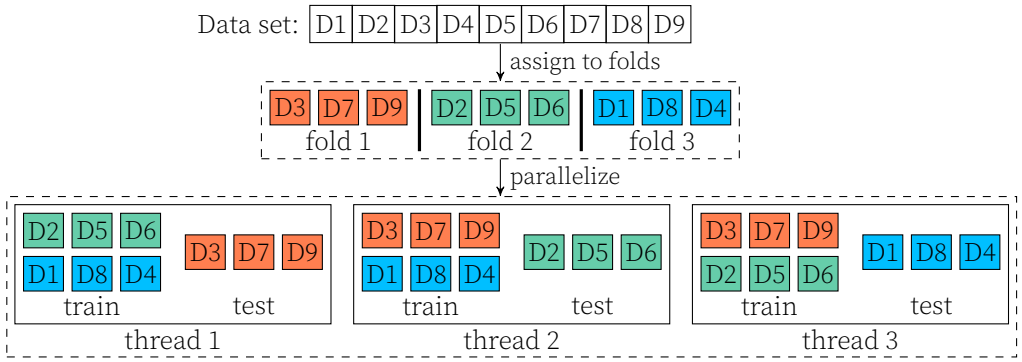

Figure 1. Visualization of parallelized cross-validation. In this case, there are 9 data instances that are randomly assigned to 3 folds, which is indicated using colors. Three parallel threads are used for the evaluation. The first thread uses folds 2 and 3 for training and fold 1 for testing. The second thread uses folds 1 and 3 for training and fold 2 for testing. The third thread uses folds 1 and 2 for training and fold 3 for testing. The results from all three threads are averaged.

any cross-platform PRNG. The second reproducibility problem was solved by re-seeding the PRNG in each thread before each repetition and holding the PRNG state variables in thread-local storage (TLS). These modifications do not require synchronizing the PRNG calls in parallel threads. That is, they don't reduce the scalability of parallelized CV.

This technique was evaluated by modifying LIBLINEAR[1], which is a popular library that implements randomized learning algorithms based on linear support vector machines[2]. The original library is affected by both reproducibility problems described above. The experiments showed that the results obtained with the modified version of the library remained reproducible across platforms and compilers. Furthermore, the results were not affected by the number of parallel threads used during the CV.

2 A Recipe for Reproducible Parallelizable Cross-Validation

The choice of a PRNG may affect the results of a parallelized CV when it is used with a randomized learning algorithm. More specifically, this choice affects: 1) the generation of CV folds; and 2) the PRNG output used by the learning algorithm in each thread. In both cases, the following recipe makes the results reproducible:

1. Use reproducible CV folds. This can be achieved by using a predetermined assignment of instances to folds or by using a known random seed for a randomized assignment.

2. Use thread-local storage (TLS) to hold the PRNG state in each thread without sharing it with any other thread. Alternatively, each thread can allocate the memory for its PRNG state dynamically, so that it is not shared with any other thread.

3. Re-seed the PRNG in each thread before processing each CV repetition. That is, if a thread runs more than one CV repetition, then use a known random seed to initialize the thread's PRNG before processing each repetition. The simplest approach is to use the same random seed in all cases. Another possibility is to determine the random seed from the data, e.g., by deriving the seed from the value of a cryptographic hash function applied to the data in the test fold. Yet another possibility is to use a PRNG that is *independent* of other PRNGs for each CV repetition[12], which prevents exhausting the state space when all PRNGs are structurally the same and only their seeds are different.

The next section describes how to apply this recipe to the LIBLINEAR library.

3 Modifications for LIBLINEAR

This section describes how to patch LIBLINEAR (version 2.21) to ensure CV reproducibility. This version of the patch assumes that a modern compiler is used (i.e., GCC 4.2 and later or LLVM/Clang 3.9 and later). The patch replaces all calls to `rand()` with a different PRNG based on the SFMT library[11]. The first five steps ensure that the cross-validation results are reproducible across platforms. The last step makes it possible to parallelize the cross-validation using multiple threads while preserving reproducibility.

1. Create a sub-directory called `SFMT` in the top-level LIBLINEAR directory and unpack the SFMT library there (we used SFMT v. 1.5.1). Then, compile it as follows:

   ```
   $ cc -c -fPIC -DSFMT_MEXP=19937 SFMT.c
   ```

2. Extend LIBLINEAR's `Makefile` to link it with SFMT by inserting the following line before the line that starts with `all:`

   ```
   override LIBS += SFMT/SFMT.o
   ```

3. Redirect all `rand()` calls to its SFMT-based replacement using the C/C++ preprocessor so that each thread uses a local private PRNG state. This can be done by inserting the following snippet after all `#include` directives at the beginning of `linear.cpp`:

   ```
   #include "SFMT/SFMT.h"
   #define rand sfmt_random
   #define RAND_MAX 0x7fffffff
   static __thread sfmt_t sfmt = {};
   static const int default_sfmt_seed = 1234;
   static inline int sfmt_random() {
       return sfmt_genrand_uint32(&sfmt) % RAND_MAX;
   }
   void seed_liblinear_PRNG(int seed) {
       sfmt_init_gen_rand(&sfmt, seed);
   }
   static void __attribute__((constructor)) seed_sfmt_startup() {
       seed_liblinear_PRNG(default_sfmt_seed);
   }
   ```

4. Add a function that initializes the random seed to LIBLINEAR's interface by inserting the following line in `linear.h`:

   ```
   void seed_liblinear_PRNG(int seed);
   ```

   This change is useful when LIBLINEAR is used as a library by another application.

5. To ensure that CV results are reproducible even with parallel processing, the PRNG should be re-seeded before processing each repetition. To achieve this, modify the last for-loop in the function `cross_validation()` in `linear.cpp` as follows:

   ```
   for(i=0;i<nr_fold;i++)
   {
           seed_liblinear_PRNG(default_sfmt_seed);
   ```

6. To enable parallel processing, use OpenMP (OpenMP v.3, 2008) to distribute the fold combinations to the worker threads by adding the following `#pragma` option before the same for-loop as in the previous step:

   ```
   #pragma omp parallel for
   for(i=0;i<nr_fold;i++)
   ```

   To enable OpenMP it may be necessary to modify the compilation options, e.g., by adding `-f openmp` to the `CFLAGS` variable.

OS	Original Version			Modified Version		
	5 Folds	10 Folds	20 Folds	5 Folds	10 Folds	20 Folds
Linux	96.858	96.883	96.952	**96.912**	**96.903**	**96.932**
macOS	96.883	96.828	96.927	**96.912**	**96.903**	**96.932**
Windows	96.858	96.749	96.799	**96.912**	**96.903**	**96.932**

Table 1. CV accuracy for LIBLINEAR on `rcv1_train` (in %), shown for 3 platforms.

OS	Original Version with Parallelization			Modified Version with Parallelization		
	5 Folds	10 Folds	20 Folds	5 Folds	10 Folds	20 Folds
Linux	96.861 (0.0062)	96.873 (0.0037)	96.951 (0.0082)	**96.912 (0)**	**96.903 (0)**	**96.932 (0)**
macOS	96.883 (0.0037)	96.830 (0.0035)	96.923 (0.0031)	**96.912 (0)**	**96.903 (0)**	**96.932 (0)**
Windows	96.863 (0)	96.759 (0)	96.804 (0)	**96.912 (0)**	**96.903 (0)**	**96.932 (0)**

Table 2. Means and standard deviations of the CV accuracies for 10 independent runs on `rcv1_train` (in %), shown for parallelized versions of LIBLINEAR on 3 platforms.

4 Results

Table 1 compares the cross-validated accuracies for the `rcv1_train` data set[14]. These results were obtained using LIBLINEAR-2.21 and its patched version described in the previous section (excluding step 6). The results are shown for three platforms: 1) Linux (32 cores, RedHat 4.4.7-18 with GCC 4.4.7), 2) macOS X (4 cores, version 10.13.6 with Xcode 10.1), and 3) Windows (2 cores, version 10 with Visual Studio 2017). The results (shown in **bold**) imply that the modified version of the library produced the same cross-validation results on all three platforms.

Table 2 shows the accuracy statistics for the parallelized CV. To enable parallel processing, the original version of LIBLINEAR was patched using only step 6 from Section 3; the PRNG remained unchanged. The modified version was patched using all six steps from Section 3 and the results with this version are shown in **bold**. The table shows that random fluctuations are introduced by parallelization and that the proposed technique eliminates them (i.e., the standard deviation is zero). The results in both tables were obtained using the following command line:

```
$ train -c 4 -e 0.1 -v <n_folds> rcv1_train.binary
```

where `<n_folds>` specified the number of folds, i.e., 5, 10, or 20.

On Windows, the PRNG state in Visual Studio is already thread-local, which prevented fluctuations (see the last row of Table 2). Without re-seeding the PRNGs, however, the results still depend on the number of threads T. To show this, we performed another experiment on Windows that used only step 6 from Section 3 and also varied T from 1 to the number of folds N, where N was set to 5, 10, and 20. The results in terms of average and standard deviation (in %) were as follows: 96.860 (0.0044) for 5-fold CV; 96.760 (0.0068) for 10-fold CV, and 96.800 (0.0043) for 20-fold CV. Table 2 shows that the results with the modified version (i.e., with all six steps) depend on N but not on T.

5 Conclusion

This paper described a technique that solves intrinsic reproducibility problems of randomized learning algorithms that stem from: 1) using a platform-dependent PRNG; and 2) sharing the PRNG state across parallel threads. A recipe for patching parallelized cross-validation was described for LIBLINEAR, which is a popular machine learning library. After applying this patch to LIBLINEAR, the CV results became reproducible on three different platforms, i.e., Linux, macOS, and Windows, because the random fluctuations arising from PRNG state sharing were eliminated.

References

1. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. "LIBLINEAR: A library for large linear classification." In: **Journal of Machine Learning Research** 9 (2008), pp. 1871–1874.
2. V. Vapnik. **Statistical Learning Theory**. New York: Wiley, 1998.
3. N. Barnes. "Publish your computer code: It is good enough." In: **Nature** 467.7317 (2010), p. 753.
4. R. Peng. "Reproducible research in computational science." In: **Science** 334.6060 (2011), pp. 1226–1227.
5. D. Ince, L. Hatton, and J. Graham-Cumming. "The case for open computer programs." In: **Nature** 482.7386 (2012), p. 485.
6. M. Hutson. "Artificial Intelligence faces reproducibility crisis." In: **Science** 359.6377 (2018), pp. 725–726.
7. P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. "Deep reinforcement learning that matters." In: **Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence**. 2018, pp. 3207–3214.
8. O. Gundersen and S. Kjensmo. "State of the art: Reproducibility in Artificial Intelligence." In: **Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence**. 2018, pp. 1644–1651.
9. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. "Hidden technical debt in machine learning systems." In: **Advances in Neural Information Processing Systems**. 2015, pp. 2503–2511.
10. POSIX 7. IEEE Computer Society and The Open Group. **IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7, Version 1003.1-2017**. Jan. 2018.
11. M. Saito and M. Matsumoto. "SIMD-oriented Fast Mersenne Twister: A 128-bit pseudorandom number generator." In: **Monte Carlo and Quasi-Monte Carlo Methods 2006**. Ed. by A. Keller et al. Errata: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/errata.pdf. Berlin, Germany: Springer, 2008, pp. 607–622.
12. M. Matsumoto and T. Nishimura. "Dynamic creation of pseudorandom number generators." In: **Monte Carlo and Quasi-Monte Carlo Methods** 2000 (1998), pp. 56–69.
13. OpenMP v.3. OpenMP Architecture Review Board. **OpenMP Application Program Interface Version 3.0**. May 2008.
14. D. Lewis, Y. Yang, T. Rose, and F. Li. "RCV1: A new benchmark collection for text categorization research." In: **Journal of Machine Learning Research** 5 (2004), pp. 361–397.