Journal of Machine Learning Research 18 (2017) 1-5 Submitted 3/16; Revised 3/17; Published 4/17

JSAT: Java Statistical Analysis Tool, a Library for Machine Learning

Edward Raff RAFF.EDWARD@GMAIL.COM
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Editor: Geoff Holmes

Abstract

Java **S**tatistical **A**nalysis **T**ool (JSAT) is a Machine Learning library written in pure Java. It works to fill a void in the Java ecosystem for a general purpose library that is relatively high performance and flexible, which is not adequately fulfilled by Weka (Hall et al., 2009) and Java-ML (Abeel et al., 2009). Almost all of the algorithms are independently implemented using an Object-Oriented framework. JSAT is made available under the GNU GPL license here: `https://github.com/EdwardRaff/JSAT`.

Keywords: java, machine learning, open source, java library, machine learning software.

1. Introduction

There exist relatively few general purpose Machine Learning libraries for use in the Java ecosystem. Weka is one of the only such libraries in this domain, and one of the oldest and most used in general. It seems that more modern and efficient libraries, such as Scikit-learn (Pedregosa et al., 2011), have been developed in every language but Java. While a number of distributed libraries, such as MLlib (Meng et al., 2016), exist in the Java ecosystem, their focus precludes the inclusion of many algorithms that do not scale out. These tools may add needless overhead and complexity for datasets and tasks that can be solved with a single machine. This leaves a void in the Java ecosystem for a tool that is both easy for developers, relatively fast, and allows more flexibility for researchers to develop and compare new algorithms.

JSAT attempts to fill this void. It is written in Java 6 and has no dependencies, making it easy to integrate into any Java project without conflict. Many common "bread and butter" algorithms, such as k-means, Random Forest, and SVMs are implemented with multiple different solvers. In every case one or more classical implementations is included, such as Platt's SMO algorithm for SVMs, that can be used as a baseline for correctness and comparison for researchers developing new solutions and to ensure performance is respectable compared to common baselines. Popular and more efficient algorithms, such as the algorithms for Linear-SVM and L1 regularized Logistic-Regression used in LIBLINEAR (Fan et al., 2008) are provided—as well as less common but advantageous solutions, such as accelerated k-means algorithms. JSAT is not limited to exact solvers like SMO, and includes many faster approximations such as the Projectron and BSGD.

Collectively, JSAT includes numerous algorithms for classification and regression (over 70), clustering (18), feature selection and engineering (over 20), visualization (5), and the

tools for implementing them. These counts do not include cases where many algorithms are covered by a single class. For example, the `NNChainHAC` class provides $O(n^2)$ hierarchical clustering for any Lance Williams dissimilarity. This also includes a number of algorithms that are useful but not widely available—be it in Java or any other language. Some examples include a multi-class generalization of the popular Passive Aggressive classifier and the Adaptive Multi-Hyperplane Machine (AMM). Almost all code in JSAT has been implemented independently by the author based on the original papers, rather than porting other implementations. The three primary interfaces in JSAT are `Classifier`, `Regressor`, and `Clusterer` for the three primary tasks JSAT supports. Each interface defines the API for learning and inference, with extension interfaces for specializes abilities, such as an `UpdateableClassifier` for online learning. JSAT also has interfaces related to warm starting, change detection, text processing, building/querying metric spaces, loss functions, and optimization. JSAT comes out to about 202k lines of code, compared to 707k for Weka and just under 190k lines for Scikit-learn.[1]

2. Features

JSAT has a number of features in its implementation and design to try and balance ease of use for users, and the development of new algorithms by researchers. Both of these tasks benefit from a certain level of performance in terms of run-time, which JSAT strives to achieve for all algorithms. A small benchmark on MNIST is given in Table 1, where JSAT is compared against the same algorithm in other libraries, with JSAT adjusted to match the default parameters used by others. Error rate was measured from one run on the standard training and testing split of MNIST. JSAT's performance is generally better than Weka and BudgetedSVM (Djuric et al., 2013), and more mixed when compared to LIBLINEAR. That the latter two are written in C/C++ is important to demonstrate that Java code can perform on a similar level while offering the benefits of the Java platform and ecosystem. JSAT's large collection of algorithms also allows for better efficiency by selecting the most appropriate algorithm. For example, when using the accelerated k-means algorithm from Elkan (2003), JSAT becomes 170 times faster with respect to Weka's implementation. All of these numbers are for single-threaded executions, and JSAT supports multi-threaded execution of many algorithms, more so than Weka and many other libraries.

	Weka						*LIBLINEAR*		*BudgetedSVM*	
	Platt SMO	C45	RF	1-NN	LR	Lloyd's k-means	SVM by DCD	newGLMNET	AMM	BSGD
Other Time	7904	303	143	2537	3301	1011	161	14.5	59.5	160
JSAT Time	973	139	125	691	914	36	71	29.2	9.7	64
Other Error	1.55%	11.1%	3.26%	3.09%	8.21%	—	8.30%	8.35%	37.2%	21.8%
JSAT Error	1.56%	11.5%	4.19%	3.09%	7.76%	—	9.21%	8.53%	5.02%	10.5%

Table 1: Training time (in seconds) on MNIST

2.1 For Researchers

JSAT includes a wide breadth of algorithms, allowing for comparison against many other approaches in a unified framework. Comparisons also become more meaningful as all al-

1. Excluding comments and blanks lines, based on `https://www.openhub.net`

gorithms are on an equal footing of speed, avoiding the need for alternative comparison metrics when using multiple libraries. Compared to Python, developing in Java can be a benefit for any algorithm that is not easy to implement in a vectorized approach, which is necessary for performance in most interpreted languages.

For high level implementations, JSAT includes the L-BFGS and OWL-QN solvers, requiring only the implementation of the desired objective function and its gradient to be used. It's also common to implement one's algorithm via SGD for efficiency, and JSAT provides common variants such as Nestrov Momentum SGD, Adam, and RMSprop.

When using lower level linear algebra constructs, JSAT continues to use an Object-Oriented approach. This allows for simple and common tricks to be added to an implementation by changing only the initialization of the object. For example the `ScaledVector` object allows one to implement the common trick for updating a vector by a scalar in constant time presented in Shalev-Shwartz et al. (2007). This works with both sparse and dense vectors. Similar objects exist for a vector shifted by a constant, keeping track of the 2-norm of a vector, or implicitly constructing the degree-2 polynomial expansion of the vector.

We also borrow the concept of the `kcentroid` from Dlib (King, 2009), called `KernelPoint` in JSAT. This object represents the result of performing math in the space induced by using the kernel trick. The object itself approximates the result with a compact solution by projecting new vectors to a basis set. JSAT extends it to support multiple strategies, such as the specialized merged result for the RBF kernel from Wang et al. (2010). This allows for a single piece of code to switch between multiple different approximation methods for implementing a kernelized algorithm. A `KernelPoints` further extends the concept to multiple kernelized weight vectors backed by a single set of basis vectors, which is useful for multi-class algorithms.

At the lowest level, JSAT includes interfaces for continuous and discrete probability distributions, and implementations of more specialized math functions such as the digamma function. Faster, but less accurate, implementations of common math functions are provided as well so authors may optimize their algorithms further. Useful tools are included at this level, such as a generic object for implementing functions via continued fractions.

2.2 For Developers / Users

Building Machine Learning-based solutions often revolves around parameter tuning and trying multiple different algorithms to see which obtains the best performance. Based on this fact, JSAT attempts to make this process simpler for both novices as well as experts. All data in JSAT is represented with a `DataSet` class, of which specialized instances exist for label-less, classification, and regression datasets—making the goal explicit by the type system. The creation of test and validation sets are simply one function call, for example `List<DataSet> splits = dataset.randomSplit(0.7, 0.2, 0.1)` will split a dataset into 3 subsets, with 70% for training, 20% for validation, and 10% for testing. Any number of ratios could be given, and it would return as many sets. It can also be used for selecting a random subset by passing in just one value less than 1.

Explicit classification and regression model evaluation objects exist to either use pre-selected test sets or perform cross validation. Each support the use of an arbitrary number of evaluation metrics for their respective types, which can be passed in as objects. This

includes common metrics such as AUC, F_β score, Precision, Recall, and others. These metrics, as well as many of the algorithms in JSAT, support weighted data instances.

```
ClassificationDataSet dataset = LIBSVMLoader.loadC(new
    File("diabetes.libsvm"));
List<ClassificationDataSet> splits = dataset.randomSplit(0.75, 0.25);
ClassificationDataSet train = splits.get(0), test = splits.get(1);
PlattSMO model = new PlattSMO(new RBFKernel());//have data, now pick model
RandomSearch search = new RandomSearch((Classifier)model, 3);
search.autoAddParameters(train); //automatically find parameters to tune
search.trainC(train);//build model & tune parameters
ClassificationModelEvaluation cme = new
    ClassificationModelEvaluation(search.getTrainedClassifier(), train);
cme.evaluateTestSet(test);//evaluate tuned model
System.out.println("Tuned Error rate: " + cme.getErrorRate());
```

Listing 1: Loading a dataset and performing a parameter search in JSAT

Built upon these is a framework for model evaluation, currently supporting the classic Grid-Search approach and the Random-Search method (Bergstra and Bengio, 2012). Setting the search parameters and their ranges is both cumbersome from a coding standpoint, and difficult for novice users. To alleviate this issue, every object with tunable parameters in JSAT may specify a distribution for each parameter it wants included in the search. This is done with respect to an input dataset, and the methods are discovered through Java's reflection mechanism. This allows JSAT to automatically populate the search parameters with reasonable values, even when the user may not have properly normalized their data. An abridged example of this is provided in Listing 1. In this example, the RBF kernel will automatically be searched over a wider space since the data has not been normalized into the $[-1, 1]$ range, a common issue overlooked by beginners (Hsu et al., 2003). It also has the benefit of being concise, and does not need to be changed when the algorithm used changes. When desired, the user can manually specify the searched parameters and values.

The GridSearch object also has additional logic for models that can be warm-started in their training. The warm-start interface indicates the main parameter that warm-starts should be based off of—automatically making use of this functionality for faster training. This is also helpful for building regularization paths, as is common with L_1 regularized models (Schmidt et al., 2009). This is an improvement over many packages that support warm-starting for some models, but require explicit cross validation objects for each algorithm.

3. Conclusion and Future Plans

JSAT has a wide breadth in the algorithms and tools it makes available to users and researchers, and fills a vital performance need while maintaining a relatively small code base. Future development goals are to further refine the API for consistency and ease of use. A move to Java 8 is planned, and will be used to perform more significant refactoring.

References

T Abeel, Y Peer, and Y Saeys. Java-ML: A Machine Learning Library. *Journal of Machine Learning Research*, 10:931–934, 2009. ISSN 15324435.

J Bergstra and Y Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. ISSN 1532-4435.

N Djuric, L Lan, S Vucetic, and Z Wang. BudgetedSVM: A Toolbox for Scalable SVM Approximations. *Journal of Machine Learning Research*, 14:3813–3817, 2013.

C Elkan. Using the Triangle Inequality to Accelerate k-Means. In *International Conference on Machine Learning (ICML)*, pages 147–153. AAAI Press, 2003.

R Fan, K Chang, C Hsieh, X Wang, and C Lin. LIBLINEAR: A Library for Large Linear Classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

M Hall, E Frank, G Holmes, B Pfahringer, P Reutemann, and I Witten. The WEKA Data Mining Software: An Update Mark. *ACM SIGKDD Explorations Newsletter*, 11 (1):10–18, nov 2009. ISSN 19310145. doi: 10.1145/1656274.1656278.

C Hsu, C Chang, and C Lin. A Practical Guide to Support Vector Classification. Technical report, National Taiwan University, 2003.

D. E. King. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009. ISSN 15324435. doi: 10.1145/1577069.1755843.

X Meng, J Bradley, B Yavuz, E Sparks, S Venkataraman, D Liu, J Freeman, DB Tsai, M Amde, S Owen, D Xin, R Xin, M J. Franklin, R Zadeh, M Zaharia, and A Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34): 1–7, 2016. URL `http://jmlr.org/papers/v17/15-237.html`.

F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

M Schmidt, G Fung, and R Rosaless. Optimization Methods for L1-Regularization. Technical report, University of British Columbia, 2009.

S Shalev-Shwartz, Y Singer, and N Srebro. Pegasos : Primal Estimated sub-GrAdient SOlver for SVM. In *International Conference on Machine Learning (ICML)*, pages 807–814, New York, NY, 2007. ACM. doi: 10.1145/1273496.1273598.

Z Wang, K Crammer, and S Vucetic. Multi-class pegasos on a budget. In *International Conference on Machine Learning (ICML)*, pages 1143–1150, 2010.