GENO – Optimization for Classical Machine Learning Made Fast and Easy

Sören Laue,[1,2] **Matthias Mitterreiter,**[1] **Joachim Giesen**[1]
[1]Friedrich-Schiller-Universität Jena
Faculty of Mathematics and Computer Science
Ernst-Abbe-Platz 2
07743 Jena, Germany
Friedrich-Schiller-University Jena
[2]Data Assessment Solutions GmbH, Hannover

Abstract

Most problems from classical machine learning can be cast as an optimization problem. We introduce GENO (GENeric Optimization), a framework that lets the user specify a constrained or unconstrained optimization problem in an easy-to-read modeling language. GENO then generates a solver, i.e., Python code, that can solve this class of optimization problems. The generated solver is usually as fast as hand-written, problem-specific, and well-engineered solvers. Often the solvers generated by GENO are faster by a large margin compared to recently developed solvers that are tailored to a specific problem class.

An online interface to our framework can be found at http://www.geno-project.org.

1 Introduction

Most problems from classical machine learning like support vector machines, robust regression, elastic net regression, sparse PCA, matrix or tensor factorization, etc. can be cast as optimization problems. Each new machine learning problem gives rise to a new optimization problem for which a new problem-specific solver is being implemented manually. However, designing and implementing optimization algorithms is still a time-consuming and error-prone task.

Here, we demonstrate GENO (GENeric Optimization), an optimization framework that allows to state constrained and unconstrained optimization problems in an easy-to-read modeling language. From the problem formulation that the user can specify an optimizer is automatically generated.

Contrary to common belief, we could show (Laue, Mitterreiter, and Giesen 2019) that the solvers generated by GENO are (1) as efficient as well-engineered, specialized solvers like LIBSVM, LIBLINEAR, or glmnet, (2) more efficient by a decent margin than recent state-of-the-art, problem-specific solvers that have been published at ICML and NeurIPS within the last few years, and (3) orders of magnitude more efficient than classical modeling language plus solver approaches like CVXPY paired with solvers like Gurobi or Mosek.

Related work. Classical machine learning is typically served by toolboxes like scikit-learn (Pedregosa and others 2011), Weka (Frank, Hall, and Witten 2016), and MLlib (Meng and others 2016). These toolboxes mainly serve as wrappers for a collection of well-engineered implementations of standard solvers like LIBSVM or glmnet. A disadvantage of the toolbox approach is a lacking of flexibility. A slightly changed model, for instance by adding a non-negativity constraint, might already be missing in the framework.

Modeling languages provide more flexibility since they allow to specify problems from large problem classes. Popular modeling languages for optimization are CVX (Grant and Boyd 2008) for MATLAB and its Python extension CVXPY (Diamond and Boyd 2016), and JuMP (Dunning, Huchette, and Lubin 2017) which is bound to Julia. All these languages take an instance of an optimization problem and transform it into some standard form of a linear program (LP), quadratic program (QP), second-order cone program (SOCP), or semi-definite program (SDP). The transformed problem is then addressed by solvers for the corresponding standard form. However, the transformation into standard form can be inefficient, because the formal representation in standard form can grow substantially with the problem size. This representational inefficiency directly translates into computational inefficiency.

GENO differs from the standard modeling language plus solver approach by a much tighter coupling of the language and the solver. GENO does not transform problem instances but whole problem classes, including constrained problems, into a very general standard form. Since the standard form is independent of any specific problem instance it does not grow for larger instances. GENO does not require the user to tune parameters and the generated code is highly efficient.

2 The GENO Pipeline

GENO features a modeling language and a solver that are tightly coupled. The modeling language allows to specify a whole class of optimization problems in terms of an objective function and constraints that are given as vectorized linear algebra expressions. Neither the objective function nor the constraints need to be differentiable. Non-differentiable

```
parameters
    Matrix A symmetric
variables
    Vector x
min
    -x'*A*x / (x'*x)
```

```
parameters
     Matrix A
     Vector b
variables
     Vector x
min
     norm2(A*x - b)^2
st
     x >= 0
```

```
parameters
     Matrix A
     Vector b
variables
     Vector x
min
     norm1(x)
st
     A*x == b
     sum(x) == 1
     x >= 0
```

Figure 1: A few optimization problems formulated in the GENO modeling language. The problem on the left is an unconstrained optimization problem that computes the Rayleigh quotient, the problem in the middle is the non-negative least squares problem, and the problem on the right shows an ℓ_1-norm minimization problem from compressed sensing over the unit simplex.

problems are transformed into constrained, differentiable problems. A general purpose solver for constrained, differentiable problems is then instantiated with the objective function, the constraint functions and their respective gradients. The gradients are computed by the matrix and tensor calculus algorithm (Laue, Mitterreiter, and Giesen 2018) and its extension (Laue, Mitterreiter, and Giesen 2020).

Generating a solver takes only a few milliseconds. An interface to the GENO framework can be found at `http://www.geno-project.org`.

Modeling Language

A GENO specification has four blocks, see Figure 1 for some examples: (1) Declaration of the problem parameters that can be of type *Matrix*, *Vector*, or *Scalar*, (2) declaration of one or more optimization variables that also can be of type *Matrix*, *Vector*, or *Scalar*, (3) specification of the objective function in a MATLAB-like syntax, and finally (4) specification of the constraints, also in a MATLAB-like syntax that supports the following operators and functions: `+, -, *, /, .*, ./, ^, .^, log, exp, sin, cos, tanh, abs, norm1, norm2, sum, tr, det, inv`. Matrices can be general dense matrices or sparse, symmetric, or positive semidefinite.

Note that in contrast to instance-based modeling languages like CVXPY no dimensions need to be specified. Also, the specified problems do not need to be convex. In case the problem is non-convex, a local optima will be returned.

Generic Optimizer

At its core, GENO's generic optimizer is a solver for unconstrained, smooth optimization problems. Thus, we implemented the L-BFGS-B quasi-Newton algorithm that can also handle box constraints on the variables. It scales very well to problems involving millions of variables. This solver is then extended to handle also non-smooth and constrained problems using an Augmented Lagrangian approach.

While GENO relies only on this optimization algorithm combination, it is surprising that the automatically generated solvers are usually as fast as well-engineered, specialized solvers, more efficient by a good margin than many recent state-of-the-art, problem specific solvers, and orders of magnitude faster than classical modeling language plus solver approaches.

More technical details and experiments on a diverse set of classical machine learning problems can be found in the full paper (Laue, Mitterreiter, and Giesen 2019).

Acknowledgments

Sören Laue has been funded by Deutsche Forschungsgemeinschaft (DFG) under grant LA 2971/1-1.

References

Diamond, S., and Boyd, S. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *JMLR* 17(83):1–5.

Dunning, I.; Huchette, J.; and Lubin, M. 2017. JuMP: A modeling language for mathematical optimization. *SIAM Review* 59(2):295–320.

Frank, E.; Hall, M. A.; and Witten, I. H. 2016. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, fourth edition.

Grant, M., and Boyd, S. 2008. Graph implementations for nonsmooth convex programs. In Blondel, V.; Boyd, S.; and Kimura, H., eds., *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences. 95–110.

Laue, S.; Mitterreiter, M.; and Giesen, J. 2018. Computing higher order derivatives of matrix and tensor expressions. In *NeurIPS*.

Laue, S.; Mitterreiter, M.; and Giesen, J. 2019. GENO – GENeric Optimization for Classical Machine Learning. In *NeurIPS*.

Laue, S.; Mitterreiter, M.; and Giesen, J. 2020. A simple and efficient tensor calculus. In *AAAI*.

Meng, X., et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17(1):1235–1241.

Pedregosa, F., et al. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12:2825–2830.