

Logistic regression python solvers' definitions

Asked 8 years, 4 months ago Modified 2 years, 2 months ago

Viewed 145k times



164



I am using the logistic regression function from sklearn, and was wondering what each of the solver is actually doing behind the scenes to solve the optimization problem.

Can someone briefly describe what "newton-cg", "sag", "lbfgs" and "liblinear" are doing?

python

python-3.x

scikit-learn

logistic-regression

Share

Improve this question

Follow

edited Jun 10, 2021 at 15:18



J. Dionisio

170 ● 1 ● 15

asked Jul 28, 2016 at 15:02



Clement

1,790 ● 3 ● 12 ● 10

-
- 4 You find introductions and references to original papers within the [user-guide](#) – [sascha](#) Jul 28, 2016 at 15:19
-



325

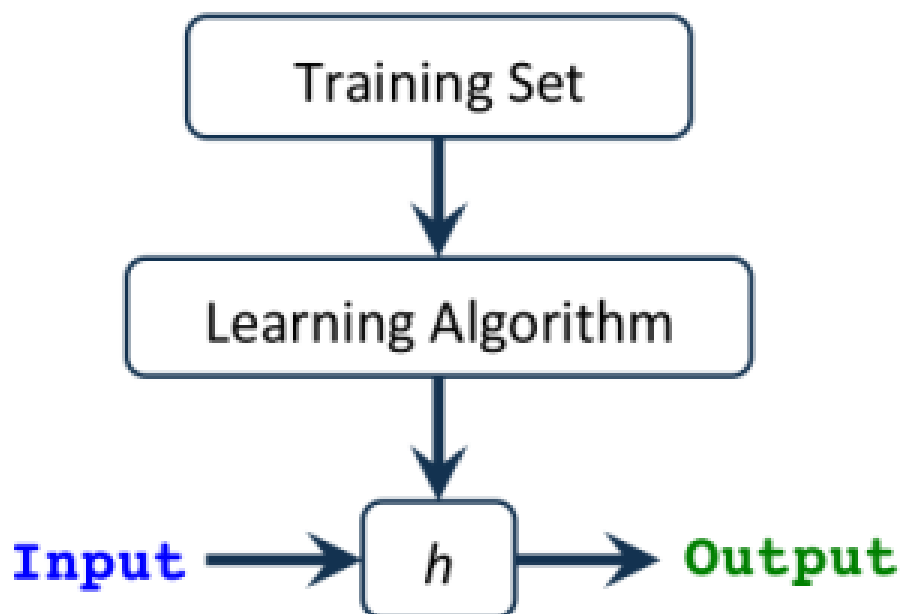


Well, I hope I'm not too late for the party! Let me first try to establish some intuition before digging into loads of information (**warning**: *this is not a brief comparison, TL;DR*)

Introduction

A hypothesis $h(x)$, takes an *input* and gives us the *estimated output value*.

This hypothesis can be as simple as a one-variable linear equation, .. up to a very complicated and long multivariate equation with respect to the type of algorithm we're using (e.g. *linear regression, logistic regression..etc*).

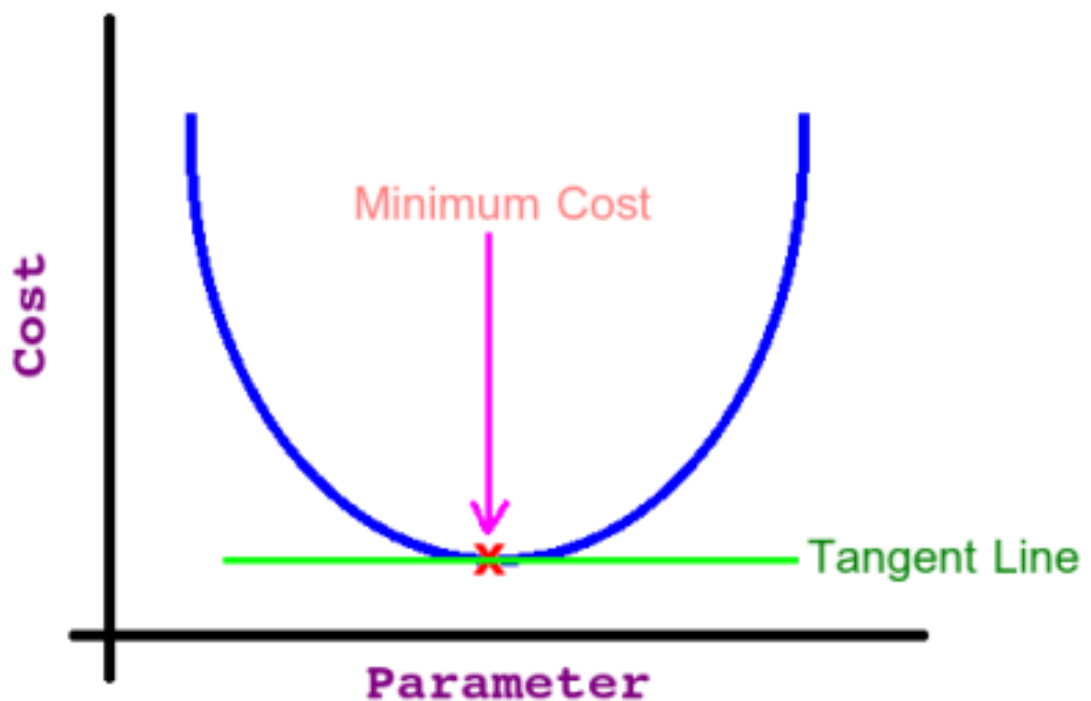


Our task is to find the **best Parameters** (a.k.a Thetas or Weights) that give us the **least error** in predicting the output. We call the function that calculates this error a

Cost or Loss Function, and apparently, our goal is to **minimize** the error in order to get the best-predicted output!

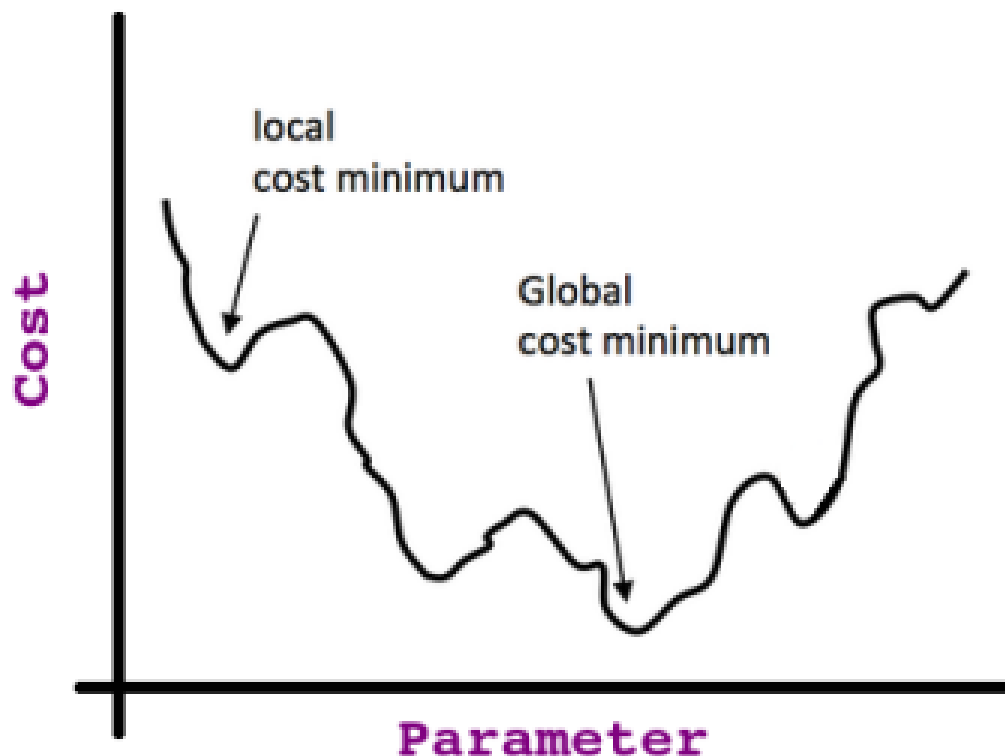
One more thing to recall is, the relation between the parameter value and its effect on the cost function (i.e. the error) looks like a **bell curve** (i.e. **Quadratic**; recall this because it's important).

So if we start at any point in that curve and keep taking the derivative (i.e. tangent line) of each point we stop at (*assuming it's a univariate problem, otherwise, if we have multiple features, we take the partial derivative*), we will end up at what so-called the **Global Optima** as shown in this image:



If we take the partial derivative at the minimum cost point (i.e. global optima) we find the **slope** of the tangent line = **0** (then we know that we reached our target).

That's valid only if we have a *Convex* Cost Function, but if we don't, we may end up stuck at what is called **Local Optima**; consider this non-convex function:



Now you should have the intuition about the heck relationship between what we are doing and the terms: *Derivative, Tangent Line, Cost Function, Hypothesis ..etc.*

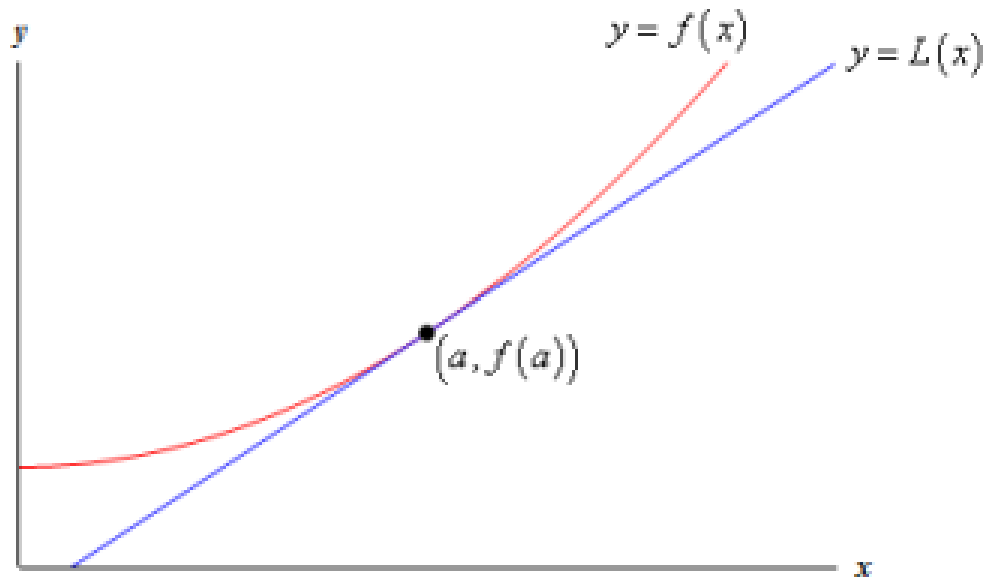
Side Note: The above-mentioned intuition is also related to the Gradient Descent Algorithm (see later).

Background

Linear Approximation:

Given a function, $f(x)$, we can find its tangent at $x=a$. The equation of the tangent line $L(x)$ is: $L(x)=f(a)+f'(a)(x-a)$.

Take a look at the following graph of a function and its tangent line:

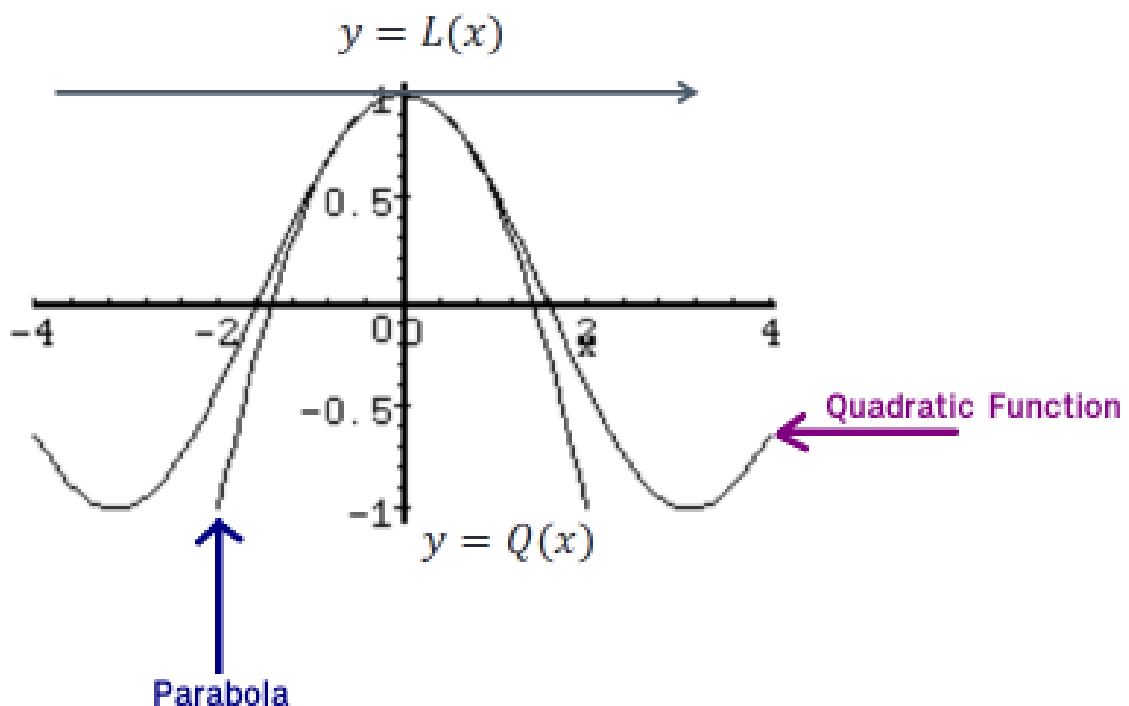


From this graph we can see that near $x=a$, the tangent line and the function have *nearly* the same graph. On occasion, we will use the tangent line, $L(x)$, as an approximation to the function, $f(x)$, near $x=a$. In these cases, we call the tangent line the "*Linear Approximation*" to the function at $x=a$.

Quadratic Approximation:

Same as a linear approximation, yet this time we are dealing with a curve where we *cannot* find the point near to **0** by using only the *tangent line*.

Instead, we use the **parabola** as it's shown in the following graph:



In order to fit a good parabola, both parabola and quadratic function should have the same **value**, the same **first derivative**, AND the same **second derivative**. The formula will be (*just out of curiosity*): $Q_a(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2$

Now we should be ready to do the comparison in detail.

Comparison between the methods

1. Newton's Method

Recall the motivation for the gradient descent step at x : we minimize the quadratic function (i.e. Cost Function).

Newton's method uses in a sense a **better** quadratic function minimisation. It's better because it uses the quadratic approximation (i.e. first AND second partial derivatives).

You can imagine it as a twisted Gradient Descent with the Hessian (*the Hessian is a square matrix of second-order partial derivatives of order $n \times n$*).

Moreover, the geometric interpretation of Newton's method is that at each iteration one approximates $f(x)$ by a quadratic function around x_n , and then takes a step towards the maximum/minimum of that quadratic function (in higher dimensions, this may also be a *saddle point*). Note that if $f(x)$ happens to be a quadratic function, then the exact extremum is found in one step.

Drawbacks:

1. It's computationally **expensive** because of the Hessian Matrix (i.e. second partial derivatives calculations).
2. It attracts to **Saddle Points** which are common in multivariable optimization (i.e. a point that its partial derivatives disagree over whether this input should be a maximum or a minimum point!).

2. Limited-memory Broyden–Fletcher–Goldfarb–Shanno Algorithm:

In a nutshell, it is an analogue of Newton's Method, yet here the Hessian matrix is **approximated** using updates specified by gradient evaluations (or approximate gradient evaluations). In other words, using estimation to the inverse Hessian matrix.

The term Limited-memory simply means it stores only a few vectors that represent the approximation implicitly.

If I dare say that when the dataset is ***small***, L-BFGS relatively performs the best compared to other methods especially because it saves a lot of memory, however, there are some “*serious*” drawbacks such that if it is unsafeguarded, it may not converge to anything.

Side note: This solver has become the default solver in sklearn LogisticRegression since version 0.22, replacing LIBLINEAR.

3. A Library for Large Linear Classification:

It's a linear classification that supports logistic regression and linear support vector machines.

The solver uses a Coordinate Descent (CD) algorithm that solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes.

LIBLINEAR is the winner of the ICML 2008 large-scale learning challenge. It applies *automatic parameter selection* (a.k.a L1 Regularization) and it's recommended when you have high dimension dataset (*recommended for solving large-scale classification problems*)

Drawbacks:

1. It may get stuck at a *non-stationary point* (i.e. non-optima) if the level curves of a function are not

smooth.

2. Also cannot run in parallel.
3. It cannot learn a true multinomial (multiclass) model; instead, the optimization problem is decomposed in a “one-vs-rest” fashion, so separate binary classifiers are trained for all classes.

Side note: According to Scikit Documentation: The “liblinear” solver was the one used by default for historical reasons before version 0.22. Since then, the default use is Limited-memory Broyden–Fletcher–Goldfarb–Shanno Algorithm.

4. Stochastic Average Gradient:

The SAG method optimizes the sum of a finite number of smooth convex functions. Like stochastic gradient (SG) methods, the SAG method's iteration cost is independent of the number of terms in the sum. However, by *incorporating a memory of previous gradient values, the SAG method achieves a faster convergence rate than black-box SG methods.*

It is **faster** than other solvers for *large* datasets when both the number of samples and the number of features are large.

Drawbacks:

1. It only supports **L2** penalization.

2. This is not really a drawback, but more like a comparison: although SAG is suitable for large datasets, with a memory cost of $O(N)$, it can be less practical for very large N (*as the most recent gradient evaluation for each function needs to be maintained in the memory*). This is usually not a problem, but a better option would be SVRG [1](#), [2](#) which is unfortunately not implemented in scikit-learn!

5. SAGA:

The SAGA solver is a *variant* of SAG that also supports the non-smooth *penalty L1* option (i.e. L1 Regularization). This is therefore the solver of choice for **sparse** multinomial logistic regression. It also has a better theoretical convergence compared to SAG.

Drawbacks:

1. This is not really a drawback, but more like a comparison: SAGA is similar to SAG with regard to memory cost. That's it's suitable for large datasets, yet in edge cases where the dataset is very large, the SVRG [1](#), [2](#) would be a better option (unfortunately not implemented in scikit-learn)!

Side note: According to Scikit Documentation: The SAGA solver is often the best choice.

Please note the attributes "Large" and "Small" used in Scikit-Learn and in this comparison are relative. AFAIK, there is no universal unanimous and accurate definition of the dataset boundaries to be considered as "Large", "Too Large", "Small", "Too Small"...etc!

Summary

The following table is taken from [Scikit Documentation](#)

Case	Solver
L1 penalty	"liblinear" or "saga"
Multinomial loss	"lbfgs", "sag", "saga" or "newton-cg"
Very Large dataset (n_samples)	"sag" or "saga"

Updated Table from the same link above (accessed 02/11/2021):

	Solvers				
Penalties	'liblinear'	'lbfgs'	'newton-cg'	'sag'	'saga'
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty ('none')	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

Share Improve this answer

edited Oct 11, 2022 at 9:31

Follow

answered Sep 18, 2018 at 14:05



Yahya

14k ● 6 ● 32 ● 43

-
- 1 Minor question, what is approx short for in "approx. all gradients"? – [stannius](#) Jul 10, 2019 at 22:15
 - 2 How do you decide whether your data is "small" enough for lbfgs with logistic regression? – [michen00](#) Jan 6, 2022 at 5:50
 - 3 In addition this terrific answer, Stochastic Gradient Descent also possible by using the solver: `SGDClassifier(loss='log')`. Specifying `loss="log"` forces the use of logistic regression. – [Joe Golton](#) Mar 2, 2022 at 16:52
-
- 1 I would suggest to remove the drawback about memory cost for SAG. It creates a confusion when compared to SAGA. It's true that the memory cost is $O(n)$. But 1) SAGA has similar memory cost. 2) The storage requirement can be improved by minibatches for dense matrices (as per SAG paper). 3) Later in the summary it is written that both SAG and SAGA are suitable for large datasets, which contradicts this SAG drawback – [Nikolay Zakirov](#) Oct 11, 2022 at 4:07
-
- 1 @NikolayZakirov Thanks for the feedback. Yes, you are correct, but it's more about my wording of the problem. Let me update it! – [Yahya](#) Oct 11, 2022 at 8:54
-



I would like to add my two cents to the terrific answer given by [Yahia](#)

8



My goal is to establish intuition how to get from full gradient descent method to SG then to SAG and then to SAGA.



On the Stochastic Gradient (SG) methods.



SG takes advantage of the fact that commonly used loss functions can be written as a sum of per-sample loss functions $f(w) = \frac{1}{n} \sum f_i(w)$, where w is the weight vector being optimized. The gradient vector then is written as a sum of per-sample gradient vectors:

$$\nabla f(w) = \frac{1}{n} \sum \nabla f_i(w).$$

E.g. least square error has this form

$$\frac{1}{n} \sum \{(x_i^T w - y_i)^2\}, \text{ where } x_i \text{ are features of } i\text{-th sample and } y_i \text{ the } i\text{-th ground truth value (target, dependent variable).}$$

And the logistic regression loss has this form (in notation 2)

$$\frac{1}{n} \sum_{j=1}^n \log(1 + e^{-y_j(x_j^T w)}).$$

SG

The main idea of stochastic gradient that instead of computing the gradient of the whole loss function, we can compute the gradient of f_i , the loss function for a single random sample and descent towards that sample gradient direction instead of full gradient of $f(x)$. This is much faster. The reasoning is that uniformly randomly chosen sample gradient represents an unbiased estimate of the gradient of the whole loss function.

In practice, SG descent has worse convergence rate $O(1/\sqrt{k})$ than full gradient descent $O(1/k)$ where k is the number of iterations. But it has faster

convergence in terms of number of flops (simple arithmetic operations) as each iteration requires computation of only one gradient instead of n . It also suffers from high variance (indeed we may not necessarily descent when picking random i , we may as well ascent)

SAG

SAG achieves convergence rate of full gradient descent $\mathcal{O}(1/k)$ without making each iteration more expensive in flops compared to SG (if only by a constant).

SAG algorithm minimizing $f(w)$ is straightforward (for dense matrices of features).

At step 0 pick a point w_0 (leaving aside how you pick it). Initialize with 0 memory cells y_i for saving gradients of f_i at later steps.

At step k update weights with an average of lagged gradients taken from the memory cells (lagged as they are not updated at every step):

$$w_{k+1} = w_k - \frac{\alpha_k}{n} \sum_{i=1}^n y_i$$

Pick uniformly randomly index i_k from $1..n$ and update only one single memory cell y_{i_k}

$$y_i = \begin{cases} \nabla f_i(w_k) & \text{if } i = i_k \\ y_i & \text{otherwise} \end{cases}$$

It seems that we're computing the whole sum of lagged gradients at each step but the nice part is that we can store the cumulative sum as a variable and make a cheap update to it at every step.

We may rewrite the update step a little

$$w_{k+1} = w_k - \alpha_k \left(\frac{1}{n} \sum_{i=1}^n \nabla f_{i_k}(w_k) - y_{i_k}^k \right) + \frac{1}{n} \sum_{i=1}^n y_{i_k}^k$$

and see that the sum $\frac{1}{n} \sum_{i=1}^n y_{i_k}^k$ is updated by the amount

$$\frac{1}{n} \sum_{i=1}^n \nabla f_{i_k}(w_k) - y_{i_k}^k$$

However, when we do this descent step we're not anymore going in a direction of an unbiased estimate of the full gradient at step k . We're going in a direction of a reduced variance estimate (in part because we're making a small step) but biased. I think this is an important and beautiful thing to understand so I will cite an excerpt from SAGA paper:

Suppose that we want to use Monte Carlo samples to estimate EX and that we can compute efficiently EY for another random variable Y that is highly correlated with X . One variance reduction approach is to use the following estimator θ as an approximation to EX : $\theta := \alpha(X - Y) + EY$, for a step size $\alpha \in [0, 1]$. We have that $E\theta$ is a convex combination of EX and EY : $E\theta = \alpha EX + (1 - \alpha)EY$. The standard

variance reduction approach uses $\alpha = 1$ and the estimate is unbiased $E\theta = EX$. The variance of θ is: $\text{Var}(\theta) = \alpha^2[\text{Var}(X) + \text{Var}(Y) - 2 \text{Cov}(X, Y)]$, and so if $\text{Cov}(X, Y)$ is big enough, the variance of θ is reduced compared to X , giving the method its name. By varying α from 0 to 1, we increase the variance of θ towards its maximum value (which usually is still smaller than the one for X) while decreasing its bias towards zero.

So we applied a more or less standard variance reduction approach to get from SG to SAG. The variance reduction constant α is equal to $1/n$ in SAG algorithm. If Y is the randomly picked $y^k_{i_k}$, X is the $\nabla f_{i_k}(w_k)$, the update

$$w_{k+1} = w_k - \alpha_k \left(\nabla f_{i_k}(w_k) - y_{i_k}^k \right) + \frac{1}{n} \sum_{i=1}^n y_i^k$$


uses the estimate of full gradient in the form $1/n \cdot (X - Y) + EY$

We mentioned that SG suffers from high variance. So we may say that SAG is SG with a clever method of variance reduction applied to it. I don't want to diminish the significance of the findings - picking suitable Y random variable is not simple. Now we can play with variance reduction constants. What if we take the variance reduction constant of 1 and therefore use an unbiased estimate of the full gradient?

SAGA

This is the main idea of SAGA. Take SAG algorithm and apply unbiased estimate of full gradient with variance reduction constant $\alpha=1$.

The update step gets bigger and becomes



$$w_{k+1} = w_k - \alpha_k (\nabla f_{i_k}(w_k) - y_{i_k}^k + \frac{1}{n} \sum_{i=1}^n y_i^k)$$

Due to lack of bias the proof of convergence becomes simple and has better constants than in SAG case. It also allows for additional trick allowing for l1 regularization. What I mean is proximal operator.


Proximal gradient descent step in SAGA




If you don't need l1 regularisation you can skip this part as there is whole [mathematical theory](#) on proximal operators.

Proximal operator is a generalization of gradient descent in some sense. (Operator is just a function from a vector into a vector. Gradient is an operator for example)



$$\text{prox}_h(v) = \operatorname{argmin}_u (h(u) + \frac{1}{2} \|u - v\|^2)$$



where $h(u)$ is a continuous convex function.


In other words it is same as finding minimum of $h(u)$ but also getting penalized for going too far from the initial point v . Proximal operator is a function from  \mathbb{R}^n to


 \mathbb{R}^n (vector to vector, just like gradient) parametrized by $h(x)$. It is non-expansional (i.e distance between x and y does not get bigger after applying proximal operator to x and y). Its' fixed point ( $\text{prox}_h(x) = x$) is the solution of the optimization problem. Proximal operator applied iteratively actually converges to its fixed point (although this is generally not true for non-expansive operators, i.e. not true for rotation). So most simple algorithm to find minimum using proximal operator is just applying the operator multiple times  $x_{k+1} = \text{prox}_{\alpha h}(x_k)$. And this is similar to gradient descent in some sense. Here is why:


Suppose a differentiable convex function h and instead of gradient descent update a similar backward Euler update:

 $x_{k+1} = x_k - \alpha \nabla h(x_{k+1})$. This update can be viewed as a proximal operator update

 $x_{k+1} = \text{prox}_{\alpha h}(x_k)$, since for proximal operator we need to find  x_{k+1} minimizing

 $\alpha h(x_{k+1}) + \frac{1}{2} \|x_{k+1} - x_k\|^2$ or find

 x_{k+1} such that

 $\alpha \nabla h(x_{k+1}) + x_{k+1} - x_k = 0$ so

 $x_{k+1} = x_k - \alpha \nabla h(x_{k+1})$

Ok why even consider changing one minimization problem by another (computing proximal operator is a minimization problem inside a minimization problem). The answer is for most common loss functions proximal operator either has a closed form or has efficient approximation method. Take l_1 regularizer. Its proximal

operator is called soft-thresholding operator and it has a simple form (I tried to insert it here but failed).

Now back to SAGA. Assume we minimize $g(x) + h(x)$ where $g(x)$ is a smooth convex function and $h(x)$ is a non-smooth convex function (e.g. l_1 regularization) but for which we are able to efficiently compute the proximal operator. So the algorithm could first make a gradient descent step for g to reduce g and then apply the proximal operator of h to the result to reduce h . This is the additional trick in SAGA and it is called **proximal gradient descent**.

Why SAG and SAGA are well suited for very large dataset

Here I am not sure what sklearn authors meant. Making a guess - very large dataset probably means that the feature matrix is sparse (has many 0).

Now let's consider a linearly-parameterized loss function

$$f(w) = \frac{1}{n} \sum f_i(w) = \frac{1}{n} \sum l_i(x_i^T w)$$

. Each sum term has a special form $f_i(w) = l_i(x_i^T w)$

. l_i is a function of a single variable. Notice that both cross entropy loss and least square loss have this form.

By chain rule

$$\nabla f_i(w) = \begin{bmatrix} x_i^T \nabla l_i(x_i^T w) & \dots & x_i^n \nabla l_i(x_i^T w) \end{bmatrix}$$

So it's evident that the gradient is also sparse.

SAG (and SAGA) apply a clever trick for sparse matrices. The idea is that weight vector does not need to be updated in every index at every step. Update may be skipped for the indices of the weight vector that are sparse in the current randomly chosen sample x_i at the step k .

There are other clever tricks in SAG and SAGA. But if you made it so far I invite you to look at the original papers [1](#) and [2](#). They are well written.

Share Improve this answer

answered Oct 12, 2022 at 9:22

Follow



Nikolay Zakirov


1,585 ● 10 ● 18

I've been trying to understand why SAG/SAGA offer competitive convergence rates relative to typical GD and SGD. Assuming you start with every loss gradient for every observation initialized at zero, won't the first updates involve very tiny movements, compared to SGD? I'm assuming that's the case because we're dividing the gradient sum by the number of observations at each weight update, and shouldn't most of the gradients be zeroed until we've performed a sufficiently large number of iterations? Yet sklearn's implementation seems to converge pretty quickly. – [hillard28](#)
Jan 16, 2023 at 21:06 ✎

-
- 1 there are two things. 1) to make convergence faster in the first epoch you divide not by the total number of observations n but by the number of already seen observations n_{seen} 2) (not present in sklearn but you can see it in Defazio's own implementation github.com/adefazio/point-saga/blob/master/optimization/...) in the first epoch you don't randomize, you calculate all the gradients – [Nikolay Zakirov](#)
Jan 17, 2023 at 1:58 ✎
-

Thanks! As I understand, the former is what is used by SAG (but can also be used by SAGA), right? I've been thinking about this in the context of sklearn's logistic regression, where I'm a bit fuzzy on how an epoch and an iteration might be differentiated. Is an iteration (specified by sklearn's `max_iter`) really more like an epoch, where within each epoch we randomly sample observations (e.g. SAG/SAGA total updates = `max_iter` * number of observations)? Thanks for sharing the github link by the way. Going through that as well as sklearn's code but stumbling a bit with the Cython syntax.

– [hillard28](#) Jan 17, 2023 at 4:55

- 1 in sklearn implementation SAGA is a part of SAG algorithm (here [github.com/scikit-learn/scikit-learn/blob/main/sklearn/...](https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/)). epoch is here in sklearn: `for n_iter in range(max_iter):` and iteration here: `for sample_itr in range(n_samples):` – [Nikolay Zakirov](#) Jan 17, 2023 at 5:11 

Got it! Those loops in the sklearn documentation (and I finally noticed it in Defazio's implementation too) were exactly why I asked! So, it sounds like when sklearn reports 100 iterations were performed, what's happening under the hood are $100 * n_samples$ (or total observations) iterations of randomly selecting single samples and adjusting. Interesting. I need to do a bit more digging to wrap my head around how computational complexity is still reduced against GD, since in theory sounds like each iteration calculates a gradient per sample (if sampled once). This was super helpful! – [hillard28](#) Jan 17, 2023 at 19:10



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.

