logistic regression python solvers' defintions

Ask Question

11

3

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? If not, any related links or reading materials are much appreciated too.

Thanks a lot in advance.

python-3.x scikit-learn logistic-regression

edited Jul 28 '16 at 15:10

asked Jul 28 '16 at 15:02

user3436204
73 2 7

3 You find

user-guide – sascha Jul 28 '16 at 15:19

thanks a lot!!! – user3436204 Jul 29 '16 at 19:45

1 Answer

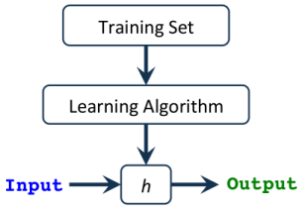
5

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

Introduction

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

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

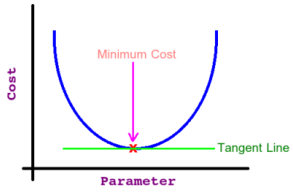


Our task is to find the

the ***least error*** in predicting the output. We call this error a ***Cost or Loss Function*** and apparently our goal is to ***minimize*** it in order to get the best predicted output!

One more thing to recall, that 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 very important) .

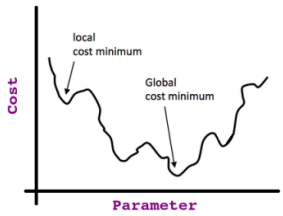
So if we start at any point in that curve and if we keep taking the derivative (i.e. tangent line) of each point we stop at, we will end up at what so called the **Global Optima** as shown in this image:



If we take the partial derivative at 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 *Convex* Cost Function, but if we

consider this non-convex function:



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

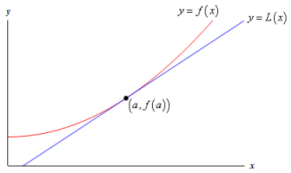
Side Note: The above mentioned intuition also related to the Gradient Descent Algorithm (see later).

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:

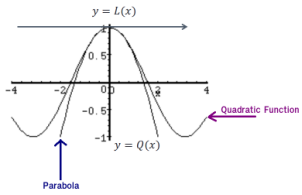


and the function have nearly the same graph. On occasion we will use the tangent line, `L(x)` , as an approximation to the function, `f(x)` , near `x=a` . In these cases we call the tangent line the linear approximation to the function at `x=a` .

Quadratic Approximation:

Same like linear approximation but this time we are dealing with a curve but we **cannot** find the point near to **0** by using the tangent line.

Instead, we use a **parabola** (*which is a curve where any point is at an equal distance from a fixed point or a fixed straight line*), like this:



And in order to fit a good parabola, both parabola and quadratic function should have same value, same first derivative, AND second derivative, ... the formula will be (*just out of curiosity*):

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

Comparison between the methods

1. Newton's Method

Recall the motivation for 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. A 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 nxn*).

Moreover, the geometric interpretation of Newton's method is that at each iteration one approximates `f(x)` by a quadratic function around `xn`, and then takes a step towards the maximum/minimum of that quadratic function

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

it stores only a few vectors that represent the approximation implicitly.

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

3. A Library for Large Linear Classification:

It's a linear classification that supports logistic regression and linear support vector machines (*A linear classifier achieves this by making a classification decision based on the value of a linear combination of the characteristics i.e feature value*).

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

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 is used by default for historical reasons.

4. Stochastic Average Gradient:

SAG method

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. Its memory cost of `O(N)`, which can make it impractical for large N (*because it remembers the most recently computed values for approx. all gradients*).

5. SAGA:

The SAGA solver is a *variant* of SAG that also supports the non-smooth *penalty=l1* option (i.e. L1 Regularization). This

also suitable ***very Large*** dataset.

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

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"

edited Sep 18 at 19:07

answered Sep 18 at 14:05

Yahya
3,519 2 8 28