

AN INTRODUCTION TO STOCHASTIC OPTIMIZATION IN MACHINE LEARNING

A Linear Regression Example

Student: Le Thi Minh Thao¹,
Supervisor: Prof. Le Thi Hoai An²

¹PUF class 2017 - 2018

²Université de Lorraine - Metz

Abstract. This paper provides an overview to the Stochastic approach in Optimization, using the specific example of Linear Regression. First, we'll talk about the role of Optimization in Machine Learning and formulate the Least Square problem. Then, we proceed by analyzing the 4 approaches of solving the Least Square problem, Batch-based and Stochastic-based approaches in First-Order and Second-Order methods. Finally, we'll implement a Linear Regression learner using Stochastic Gradient Descent, then apply the learner to the Wine Quality dataset [1] and evaluate its performance.

Part 1. Machine Learning and Optimization

A. What is Machine Learning?

Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel – 1959

One very good example of a problem perfect for Machine Learning is to build a self-driving car. If we have to explicitly program how a car would react to different situations on the road, then we must have a huge number of rules included and continuously updated. On the other hand, if we can somehow program the car the ability to learn those rules by itself, then the problem is much simpler.

Think about it, isn't it the human way to learn something? By looking at examples and deduce the rules for ourselves? That's why this field is called Machine Learning.

B. Elements of Machine Learning

The elements of Machine Learning are: *Data*, *Models* and *Algorithms*. First, Machine Learning is useless without *Data*. Since no data means no examples for the machine to learn and deduce the rules for itself. Second, *Model* is the mathematical representation of the rules our machines learnt from data, and *Algorithm* is the recipe we tell the machine to learn these representations. In short:

$$Rules \sim Model = Algorithm (Data)$$

Back to our example of building a self-driving car, the rules are whether our car should speed-up or slow down, or turn left or right, etc. under some specific conditions. The model for such rules is a Decision Tree, and we can tell the machine to learn that decision tree with the C4.5 algorithm.

C. The Mathematics of Machine Learning

So, what does a model look like? How a machine can learn and apply models? Basically, a model is a target function (f) that best “fits” our data. But what is “fit”?

Back to our self-driving car again. Imagine our lessons for the car (the data) are described as pairs of (X_i, Y_i) , where X_i is a specific driving condition (such as the road is slippery, a human is crossing, a car is slowing down, etc.) and Y_i is how the car would react to condition X_i . Then the target function (f) is the one that best approximates Y_i given X_i , that's why we say (f) best fits our data.

Mathematically stated, $f(X)$ is the function that minimizes the difference between $f(X = X_i)$ and Y_i for all pairs of (X_i, Y_i) in our dataset. And in order to learn that function, every Machine Learning algorithms must go with 2 things:

1. Representation of the Model, or the “general form” of (f) , such as Decision Tree, Neural Network, etc., without being specific.
2. An algorithm to find the explicit form of (f) , given its general form, by finding the specification that bests minimizes the difference of $f(X)$ and Y .

After finding (f) , our machine can simply use it by applying (f) on a new data X' to know its corresponding Y' . In our self-driving car example, that means applying (f) on a new driving condition to know what action to take.

One more thing, the second characteristic mentioned above is an Optimization problem.

D. The Role of Optimization

First, we recall the Standard Form of an Optimization problem:

Finding (one or more) minimizer of a function subject to some constraints

$$\operatorname{argmin}_x f_0(x)$$

subjected to

$$\begin{aligned} f_i(x) &\leq 0, \quad i = \{1, \dots, k\}, \\ h_j(x) &= 0, \quad j = \{1, \dots, l\}. \end{aligned}$$

Optimization is a big part of Machine Learning as almost every Machine Learning algorithm has an optimization algorithm at its core. For example:

Algorithm	The Optimization Problem
Linear SVM	$\min \sum_{i=1}^n \ w\ ^2 + C \sum_{i=1}^n \xi_i$ <p>such that $1 - y_i x_i^T w \leq \xi_i, \xi_i \geq 0, \forall i = \overline{1, n}$</p>
Maximum Likelihood	$\max_{\theta} \sum_{i=1}^n \log p_{\theta}(x_i)$
K – means	$\min_{\mu_1, \mu_2, \dots, \mu_k} J(\mu) = \sum_{j=1}^k \sum_{i \in C_j} \ x_i - \mu_j\ ^2$

But where do these problems come from. Back to our previous explanation of “General Form” and “Explicit Form”, let’s restate them mathematically.

Initially, most Machine Learning algorithms parameterize target function (f) , making the General Form. Then, they use some processes to turn those parameters into specific values, making the Explicit Form. These processes involve minimizing an “error” function, called the “Cost Function” (derived in order to minimize the difference between $f(X)$ and Y), making our optimization problem.

To understand this idea, let’s take a look at Linear Regression and formulate its optimization problem – the Least Square problem.

E. Linear Regression and the Least Square problem

The Linear Regression model is a model that assumes linear relationship between input variables (x) and single output variable (y), in order to predict the value of (y) using a linear combination of (x). The general form is in [2]:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n,$$

1. \hat{y} is the predicted value.
2. n is the number of features in data.
3. x_i is the i^{th} feature value.
4. θ_j is the j^{th} model parameter, including the bias term θ_0 , and the feature weights $\theta_1, \theta_2, \dots, \theta_n$.

Equation 1. Linear Regression model prediction

Or in vector form:

$$\hat{y} = h_{\theta}(x) = \theta^T x,$$

where x is a value-vector of a training instance.

Now we see the Linear Regression model, how do we train it?

First, we need a measure of how well (or poorly) the model fits our training data. That is

$$RMSE(X, h_{\theta}) = \sqrt{\frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2}.$$

Equation 2. Root Mean Square Error (RMSE) for m data points.

Next step, we need to find the value of θ that minimizes RMSE. In practice, it is simpler to minimize the Mean Square Error (MSE) instead of RMSE.

$$MSE(x, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2.$$

(We divide by m to avoid Big Number computations).

The problem of minimizing this function is called the **Least Square problem**, and it is a **solved** problem, where the best approximation for θ is

$$\hat{\theta} = (X^T X)^{-1} X^T y,$$

1. $\hat{\theta}$ is the value of θ that minimizes the cost function MSE,
2. X is the matrix of input variables x ,
3. y is the vector of target values y_1 to y_m .

Equation 3. The Normal Equation.

However, the computational complexity for finding the inverse of $X^T X$, which is an $n \times n$ matrix (n is the number of features), is typically about $O(n^3)$. That's why we need to find better ways to estimate the optimal value of θ for cases where there is large number of features. These methods are

1. **Gradient Descent**, also known as **First-Order method**,
2. **Newton Method**, also known as **Second-Order method**.

They'll be discussed in the next part, and even better, we'll further optimize the performance by looking at the **Batch vs Stochastic** approaches for each of them.

Part 2. Overview of Optimization Approaches

A. Convergence Rates

When talking about the speed of an Optimization algorithm, we use the concept of “Convergence Rate”. Simply put, it is how fast an algorithm reaches, or converges, to the optimal point of a problem.

Mathematically stated, ([3])

Assume that the sequence x_k is converging to x^* . We say x_k converges to x^* with rate r and rate constant $C < +\infty$ if

$$\lim_{k \rightarrow \infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^r} = C,$$

and r is the largest rate for which this is the case.

Basically, there are two classes of convergence rate:

1. *Linear convergence rate*, $r = 1$: Achieved by algorithms that only consider the first derivatives, cheaper in computation but run slow in practice.
2. *Quadratic convergence rate*, $r = 2$: Only achieved by algorithms that consider both first and second derivatives, costly in computation but run fast in practice.

B. The Convexity Requirements

Recall the fact that we use algorithms (instead of direct computations) because the high cost of computing things directly. But there is a trade-off: **Not everything can be found by algorithms.**

In fact, the generic solution scheme of every algorithm is:

1. Given *current feasible solution*, go over to a feasible solution in the *neighborhood* that has a better objective value.
2. **REPEAT** until *current solution* is the best solution within the *neighborhood*.

For this scheme to lead us to an optimal solution, irrespective of the starting point, we require that:

- i. The feasible region must be convex.
- ii. The objective function must be convex.

Why we have these requirements?

1. Convex feasible region: If – given the *current solution* – an optimal solution lies in a particular direction, then we can make a step into that direction **without leaving the feasible region.**

2. Convex objective function: If – given the *current solution* – an optimal solution lies in a particular direction, then the **objective function improves** when we take a step into that direction.

Back to our Least Square problem. Is the feasible region convex? Is the objective function convex? Can it be solved algorithmically?

The Least Square Problem Find θ that minimizes

$$MSE = \frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2$$

for all $\theta \in \mathbb{R}^n$.

1. First, the feasible region, \mathbb{R}^n , is a convex set \Rightarrow We have a convex feasible region.
 2. Second, $MSE = \frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2$ is a convex function because:
 - $f(x) = x^2$ is convex and $\theta^T x_i - y_i$ is affine $\forall i = \overline{1, n} \Rightarrow (\theta^T x_i - y_i)^2$ is convex $\forall i = \overline{1, n}$ (by the rule of Composition).
 - MSE is a sum of $(\theta^T x_i - y_i)^2 \Rightarrow$ MSE is a convex function.
- In other words, it is safe to solve our Least Square problem algorithmically.

C. The First-Order Method – Gradient Descent

Now, let's talk about the first way to solve the Least Square problem, “Gradient Descent” (also known as First-order method).

To get the intuition of this method, let's think about the gradient at one random input of our MSE function. The direction of this gradient is proven to be the direction that increases MSE. Therefore, by taking a small enough step in the opposite direction, we have a new input that decreases the value of MSE. And if we continue doing this, we'll eventually end up at the minimizer of MSE.

Now, in order to prove this intuition, we'll prove these two properties apply in the case of our MSE function:

1. Continuously taking small enough steps into the direction of the steepest descent will eventually reaches global minimum.
2. This algorithm has linear convergence rate.

Proof.

Point 1. *We'll eventually reaches the global minimum.*

Consider the First-order Taylor approximation at x_k (our input at k -th iteration), we have

$$f(x) = f(x_k) + \nabla f(x_k)^T (x - x_k) + o(x^2). \quad (1)$$

By taking a small enough step into the opposite direction of $\nabla f(x_k)$, we have $x_{k+1} = x_k - \eta \nabla f(x_k)$, where $\eta > 0$. Plug it into (1), we've got

$$\begin{aligned} f(x_{k+1}) &= f(x_k) + \nabla f(x_k)^T [-\eta \nabla f(x_k)] + o(x^2) \\ &= f(x_k) - \eta \nabla f(x_k)^T \nabla f(x_k) + o(x^2) \leq f(x_k), \end{aligned}$$

since $\nabla f(x_k)^T \nabla f(x_k) \geq 0$. Continue doing this, we'll eventually reach the local minimum of the “neighborhood” where we initially start. Since our objective function is convex, this also is the global minimum.

Point 2. *This algorithm has linear convergence rate.* The main idea of the proof is to consider the following theorem [4]:

Theorem. Let f be a function that satisfies:

1. f is convex and finite for all x ,
2. A finite optimal value x^* exists,
3. $\nabla f(x)$ is Lipschitz continuous with constant L (that is, there exists L such that $\|\nabla f(x_1) - \nabla f(x_2)\| \leq L \|x_1 - x_2\|, \forall x_1, x_2$).

Then, if we run Gradient Descent for k iterations with fixed step size $\lambda \leq 1/L$, we'll reach x_k such that

$$f(x_k) - f(x^*) \leq \frac{\|x_1 - x^*\|}{2\lambda k}.$$

First, the MSE function is convex and therefore satisfies **1**).

Second, the property of having finite optimal points depends on the dataset X we take. In practice, many implementations of OLS will output NA (not available due to singularities) in those bad dataset. For the sake of simplicity, we just assume that it has finite optimal points, satisfied **2**).

Third, we have the gradient of the MSE function is

$$\begin{bmatrix} \sum_{i=1}^m (\theta^T x_i - y_i) x_{i,0} \\ \sum_{i=1}^m (\theta^T x_i - y_i) x_{i,1} \\ \vdots \\ \sum_{i=1}^m (\theta^T x_i - y_i) x_{i,n} \end{bmatrix}.$$

Consider the expression $\|\nabla \text{MSE}(\theta_1) - \nabla \text{MSE}(\theta_2)\|$, we have

$$\begin{aligned}
 \|\nabla MSE(\theta_1) - \nabla MSE(\theta_2)\| &= \left\| \begin{bmatrix} (\theta_1^T - \theta_2^T) \sum_{i=1}^m x_i x_{i,0} \\ (\theta_1^T - \theta_2^T) \sum_{i=1}^m x_i x_{i,1} \\ \vdots \\ (\theta_1^T - \theta_2^T) \sum_{i=1}^m x_i x_{i,n} \end{bmatrix} \right\| \\
 &= \left\| (\theta_1^T - \theta_2^T) \begin{bmatrix} \sum_{i=1}^m x_i x_{i,0} \\ \sum_{i=1}^m x_i x_{i,1} \\ \vdots \\ \sum_{i=1}^m x_i x_{i,n} \end{bmatrix} \right\| \leq \|\theta_1^T - \theta_2^T\| \left\| \begin{bmatrix} \sum_{i=1}^m x_i x_{i,0} \\ \sum_{i=1}^m x_i x_{i,1} \\ \vdots \\ \sum_{i=1}^m x_i x_{i,n} \end{bmatrix} \right\|.
 \end{aligned}$$

Let $L := \left\| \begin{bmatrix} \sum_{i=1}^m x_i x_{i,0} \\ \sum_{i=1}^m x_i x_{i,1} \\ \vdots \\ \sum_{i=1}^m x_i x_{i,n} \end{bmatrix} \right\|$ be a constant, we have

$$\|\nabla MSE(\theta_1) - \nabla MSE(\theta_2)\| \leq \|\theta_1^T - \theta_2^T\| L = L \|\theta_1 - \theta_2\|,$$

which makes the gradient of MSE Lipschitz continuous with respect to θ , satisfied **3**). Hence, by the theorem, at k -th iteration, we'll have

$$f(x_k) - f(x^*) \leq \frac{\|x_1 - x^*\|}{2\lambda k},$$

making this algorithm has the convergence rate of $O(\frac{1}{k})$, a linear convergence rate. \square

Below is the detailed algorithm:

1. **Initialization:** Start with an initial point θ_1 and set $k = 1$.
2. **Descent direction:** Calculate $d_k = -\nabla f(\theta_k)$.
3. **Line search:** Find the next point $\theta_{k+1} = \min_{\lambda \geq 0} f(\theta_k + \lambda d_k)$.
4. **Termination:** If $\nabla f(\theta_{k+1}) \approx 0$, stop. Otherwise, set $k = k + 1$ and back to **2**.

In step 3 (“line search”), we can see a new parameter λ , this is called the learning rate. Intuitively, this represents how far we want to step in such direction. If we step too close, it's slow. But if we step too far, our path may fluctuate around the minimizer, and therefore lengthen the run time.

To fix this, we flexibly pick λ (instead of letting it be a fix number). At each iteration, we pick the learning rate (among a set of pre-defined learning rates) that minimizes our objective function. This guarantees our algorithm will stop after a finite steps.

What are the advantages and disadvantages of this method?

Advantages: Only requires gradient information \rightarrow Fast and need less storage.

Disadvantages: Only achieve linear convergence.

Next, we'll discuss the second solution to the Least Square problem – “**Newton Method**”, also known as “**Second-Order Method**”.

D. The Second-Order Method – Newton Method

The main motivation behind this Newton method is to improve the convergence rate. And to do this, instead of taking steps into the *direction of the steepest descent* as in Gradient Descent, we take steps into the *direction of the minimum* of our objective function. That is, intuitively, the direction which points straight towards our local minimum (and global minimum if our function is convex).

To prove that this idea works for our MSE function, we'll prove two things:

1. Continuously taking small enough steps into the direction of the minimum will eventually reaches global minimum.
2. This algorithm has quadratic convergence rate.

Proof.

Point 1. *We'll eventually reaches the global minimum.*

Consider the second-order Taylor approximation at x_k (our input at k -th iteration), we have

$$f(x) = f(x_k) + \nabla f(x_k)^T (x - x_k) + \frac{1}{2}(x - x_k)^T H(x_k) (x - x_k) + o(x^3). \quad (2)$$

By taking a small enough step into the direction of the minimum, we have $x_{k+1} = x_k - \eta \cdot H^{-1} \cdot \nabla f(x_k)$, where $\eta > 0$ is the learning rate and $H^{-1}(x_k)$ is the inverse of Hessian matrix at x_k . Plug it into (2), we have

$$\begin{aligned} f(x_{k+1}) &= f(x_k) + \nabla f(x_k)^T [-\eta H^{-1}(x_k) \nabla f(x_k)] \\ &\quad + \frac{1}{2} [-\eta H^{-1}(x_k) \cdot \nabla f(x_k)]^T H(x_k) [-\eta H^{-1}(x_k) \cdot \nabla f(x_k)] + o(x^3) \\ &= f(x_k) - \eta \nabla f(x_k)^T H^{-1}(x_k) \nabla f(x_k) + \frac{1}{2} \eta^2 \nabla f(x_k)^T H(x_k)^{-1} \nabla f(x_k) + o(x^3) \\ &= f(x_k) - \left(\eta - \frac{1}{2} \eta^2 \right) \nabla f(x_k)^T H^{-1}(x_k) \nabla f(x_k) + o(x^3). \end{aligned} \quad (3)$$

First, we can choose η small enough so that

$$\left(\eta - \frac{1}{2}\eta^2\right) \geq 0.$$

Second, $\nabla f_k^T \cdot H^{-1} \cdot \nabla f_k$ is a quadratic form with H^{-1} as coefficients. In order to prove that $\nabla f_k^T \cdot H^{-1} \cdot \nabla f_k \geq 0$, we'll prove that H^{-1} is semi-positive definite. Indeed, since $H(\theta)$ is the Hessian matrix of the MSE function, we have $H(\theta) \succeq 0$, which makes

$$u^T H(\theta) u \geq 0, \forall u.$$

Let $u = vH^{-1}(\theta)$, we have

$$(H(\theta)^{-1}v)^T H(\theta) (H(\theta)^{-1}v) \geq 0, \forall v \Rightarrow v^T H^{-1}(\theta) v \geq 0, \forall v,$$

which implies $H^{-1}(\theta)$ is semi-positive definite. Altogether, we have (3) implies $f(x_{k+1}) \leq f(x_k)$.

Continue doing this, we'll eventually reach the local minimum of the “neighborhood” where we initially start. Since our objective function is convex, this also is the global minimum. \square

Point 2. *This algorithm has quadratic convergence rate.* The proof for this property is outlined in [5], using Theorem 18. Below is the detailed algorithm:

1. **Initialization:** Start with an initial point x_1 and set $k = 1$.
2. **Descent direction:** Calculate $d_k = -H(\theta_k)^{-1} \cdot \nabla f(\theta_k)$.
3. **Line search:** Find the next point $\theta_{k+1} = \min_{\lambda \geq 0} f(\theta_k + \lambda d_k)$.
4. **Termination criterion:** If $\nabla f(\theta_{k+1}) \approx 0$, stop. Otherwise, set $k = k + 1$ and go to **2**.

In step 3 (“line search”), we also search for the learning-rate λ (among a pre-defined set of learning rates) that minimizes our objective function. This guarantees our algorithm will stop after a finite steps (as in the case of Gradient Descent). What are the advantages and disadvantages of this method?

Advantages: Achieves quadratic convergence rate.

Disadvantages: Requires both derivative and Hessian information, making each iteration very costly, and needs more storage.

Now we have discussed the two methods (algorithms) for solving the Least Square problem. One question arises: *What if we have too many data? Isn't the function will then be very complex?*

This question is very genuine. Recalling our MSE function as

$$MSE = \frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2,$$

and suppose $m = 10^6$ and $n = 100$ (this happens very often in practice), it's not difficult to see that computing the Gradient (and the Hessian) information for this function is very computationally expensive.

The original approach we just take is called the **Batch approach** (because we take the whole dataset into our objective function), which will run very slow in big dataset (often the case in practice). To deal with this problem, we'll use the **Stochastic approach** discussed in the next section.

E. The Batch approach vs The Stochastic approach

As stated, both the First-order and Second-order methods described above use a **Batch approach**. That is, to put all points of the dataset (as "batch" of data) into the objective function, then compute this "big" function's Gradient, as well as Hessian, to gradually "converge" at the optimal point after each iteration. This is a very natural approach, but runs slow in practice. Instead of computing the Gradient and Hessian information of this big objective function, why not approximate the true Gradient and Hessian information using several "mini-Gradient" and "mini-Hessian" information, each produced by just one data point in the dataset? Isn't it cheaper to compute, and run faster in practice?

Instead of computing the Gradient and Hessian information of this big objective function, why not approximate the true Gradient and Hessian information using several "mini-Gradient" and "mini-Hessian" information, each produced by **just one data point in the dataset**? Isn't it cheaper to compute, and run faster in practice?

This is the main idea behind the **Stochastic approach**, and below is the intuition.

We build the objective function and optimize it by:

1. Building several "mini" objective functions, each uses just one data point in the dataset. For example, $MSE_i = (\theta^T x_i - y_i)^2$, where x_i, y_i is one instance of the dataset.
2. At each iteration, compute the Gradient and Hessian information of these "mini" objective functions.
3. Find a way to "aggregate" these "mini-Gradient" and "mini-Hessian" information, so that at the end of each iteration, parameter θ moves closer to the optimal point (as in the case of using the true Gradient and Hessian information).

The detailed Algorithm and Mathematics formality will be discussed later. At this time, the key idea is that we escape low performance by approximating the true Gradient and Hessian information by aggregating many single-data-point Gradient and Hessian's information. This aggregation step is the key to the correctness of the Algorithm:

- If done wrong, we'll never reach the optimal point, as we wrongly approximate the true Gradient/Hessian information.
- But if done right, in the long run, we'll reach the optimal point as in the Batch case.

E.1. The Stochastic approach for First Order method (Stochastic Gradient Descent - SGD)

The way we do aggregation in SGD is by “random shuffling” the dataset before running each iteration, then update the parameters sequentially along this shuffled set. This way, we avoid repeating the update cycles (hence, increase the randomness nature), yet still give each instance a chance to contribute to the update of our parameters.

Here is the outline of the Algorithm for the case of optimizing our MSE function:

1. Start with an initial point θ_1 and set $k = 1$.
2. Run iteration k :
 - Random shuffling the dataset.
 - For each instance i in the dataset:
 - Compute the Gradient of $(\theta^T x_i - y_i)^2$ which is $x_i^T (\theta^T x_i - y_i)$.
 - Pick a suitable learning rate $(*)$
 - Update θ_k
 - **If acceptable minimum cost is reached:** Stop.
 - **Else:** Set $k = k + 1$ and continue.

Remark.

- Obviously, SGD runs much faster than Batch Gradient Descent, since it has very little data to manipulate at every iteration (only 1 data point, at random). This property also makes SGD possible to run on huge training sets, since only one instance needs to be in memory at each time.
- On the other hand, the stochastic nature makes the path towards global minimum not as direct as in Batch Gradient Descent. However, it has been shown that SGD almost surely converges to the global minimum if our cost function is convex (or pseudo-convex) [6], which is the case of our MSE function.
- One way to improve the convergence of SGD is to gradually reduce the learning rate, as pointed out in $(*)$. The steps start out large (which helps make quick progress and avoid local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.

For example, we can choose this “adaptive learning” rate as

$$\eta_{t+1} = \frac{\eta_t}{1 + t \cdot d},$$

where t is the order of our previous parameter update, and d is a fixed constant that helps shrink the learning rate over time.

- When the objective function is not convex (or pseudo-convex), we face a trade-off between “**Accuracy**” and “**Performance**” when using SGD. The cost function will bounce up and down, **decreasing only on average**. Over time, it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. So once the algorithm stop, the final parameters are good, but not optimal.

E.2. The Stochastic approach for Second Order method

At this point, it's tempting to think that the Stochastic approach in Second-order methods is simply adding the Hessian's inverse information at each "stochastic" parameter update. It isn't. The computation of the Hessian and the Hessian's inverse is very expensive, even when we use single-data-point objective functions. Moreover, when looking at an "outer" level, we must do this costly computation for each data point in the shuffled dataset, further increases the computational cost.

Until 2015, attempts to bring Stochastic ideas to Second-order methods weren't really successful (in compare to the wide adoption of Stochastic First-order methods). These attempts either:

1. Still require inversion of matrices, which is costly.
2. Or introduce dimension reduction techniques that can potentially distort the spectral information, reducing accuracy.

In 2016, [7] introduced an algorithm called **Linear-time Stochastic Second-order Algorithm**, or **LiSSA**, which inherits fast Second-order convergence rates while only requires linear-time per-iteration cost. Experimentally, LiSSA has achieved very good results:

- Linear convergence rate comparable to the class of fast First-order methods, such as SAGA and SVRG, but runs faster than most of them.
- Though not comparable (in convergence rate) to traditional Newton's method, it is much faster (in running time) than Newton's method and many other Second-order methods.

These experimental results, as well as LiSSA's applicability in the case of our MSE function, will be discussed in at the end of this section. This time, let's focus on the key idea behind the LiSSA algorithm – **using Taylor series to approximate the Hessian's inverse information**.

First, let's recall two facts:

1. What we need at each iteration in the traditional Newton's method is the Hessian's inverse information, not the Hessian's information. We just have to compute the Hessian because it's the common way to find the Hessian's inverse.
2. The main idea behind the Stochastic approach is to approximate the true First-order and Second-order's information using an aggregation of single-data-point's information, produces an accurate descent direction in the long run.

Second, bases on these two facts, LiSSA tries to estimate the Hessian's inverse using some aggregation of single-data-point Hessian. In details:

1. LiSSA estimates the Hessian's inverse using this well-known fact:

$$A^{-1} = \sum_{i=0}^{\infty} (I - A)^i$$

for all matrix A such that $\|A\| \leq 1$ and $A \succ 0$.

Equation 4. Taylor series expansion of the matrix inverse.

Under some regularity conditions of the objective function, we can re-write this equation as follow:

$$H_j^{-1} = \sum_{i=0}^j (I - H)^i,$$

where j is the number of terms in the Taylor series, the bigger, the better.

Equation 5. Estimating the Hessian's inverse with Taylor series.

2. In **Equation 5**, we utilize the full Hessian information H , which is very expensive to compute. To avoid this complexity, we utilize the Stochastic approach by taking an independent and unbiased sample $\{X_1, X_2, \dots, X_j\}$ of the Hessian H . That is:

$$H_j^{-1} = \sum_{i=0}^j (I - X_i)^i,$$

where $X_i = H_i$ is the Hessian information at one randomly chosen data point.

Equation 6. Stochastic estimation for the Hessian's inverse.

This aggregation, in the long run, will correctly estimate the true Hessian's inverse. Indeed, let's rephrase this idea formally in a recursive form, then shortly prove it.

Definition. (Hessian's Inverse Estimator)

Given j independent and unbiased samples $\{X_1, X_2, \dots, X_j\}$ of the Hessian H , define

$$\{\widetilde{H^{-1}_0}, \widetilde{H^{-1}_1}, \dots, \widetilde{H^{-1}_j}\}$$

recursively as follow:

$$\begin{cases} \widetilde{H^{-1}_0} = I \\ \widetilde{H^{-1}_{k+1}} = I + (I - X_k) \cdot \widetilde{H^{-1}_k} \end{cases}$$

for $k = 1, 2, \dots, j - 1$.

Remark.

1. This definition is equivalent to **Equation 6**, but is written in a recursive way which is suitable for stating algorithmically.
2. It's worth noting that the bigger j , the better the estimation as more Taylor terms are considered.
3. To prove that this estimation is correct, it's sufficient to prove that it is unbiased. In fact, base on the definition, we can see that

$$E[\widetilde{H^{-1}_j}] = H_j^{-1},$$

and since $H_j^{-1} \rightarrow H^{-1}$ as $j \rightarrow \infty$, we have $E[\widetilde{H^{-1}_j}] = H^{-1}$ as $j \rightarrow \infty$, proving its unbiasedness and further affirm **Remark 2**.

4. We can extend this definition to include the Gradient and hence completes the computation for the Descent Direction. Multiplying both sides with the Gradient's information of the previous iteration $\nabla f(x_t)$, we have

$$\begin{aligned}\widetilde{H^{-1}}_{k+1} &= I + (I - X_k) \cdot \widetilde{H^{-1}}_k \\ \Rightarrow \widetilde{H^{-1}}_{k+1} \cdot \nabla f(x_t) &= \nabla f(x_t) + (I - X_k) \cdot \widetilde{H^{-1}}_k \cdot \nabla f(x_t).\end{aligned}$$

Let

$$Z_k = \widetilde{H^{-1}}_k \cdot \nabla f(x_t),$$

because $\nabla f(x_t)$ is fixed despite how “deep” we expand the Taylor series, we can re-write the recursive equation as:

$$\begin{cases} Z_0 = \nabla f(x_t) \\ Z_{k+1} = \nabla f(x_t) + (I - X_k) \cdot Z_k \end{cases},$$

for $k = 1, 2, \dots, j - 1$.

Remark 4 builds a recursive equation that is key to the LiSSA algorithm, for two reasons: 1. It is easy to state algorithmically, and has linear computation time in the case of Generalized Linear Models. 2. It is also an unbiased estimator of the Descent Direction as:

$$E[Z_j] = E[\widetilde{H^{-1}}_j \cdot \nabla f(x_t)] = H_j^{-1}(x_t) \cdot \nabla f(x_t).$$

(using **Remark 3**)

And since

$$H_j^{-1}(x_t) \cdot \nabla f(x_t) \rightarrow H^{-1}(x_t) \cdot \nabla f(x_t)$$

as $j \rightarrow \infty$, we have

$$E[Z_j] = H^{-1}(x_t) \cdot \nabla f(x_t).$$

Below is the LiSSA algorithm, as used in the case of our MSE function.

INPUTS:

1. $f(\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T x_i - y_i)^2$: Our objective function – MSE.
2. T : The number of epochs we use to estimate θ .
3. S_1 : The number of Descent Direction estimators we want to compute.
4. S_2 : The number of Taylor expansion terms we want to use.
5. T', η : Parameters for “warm-up” Gradient Descent.

LiSSA Algorithm:

1. **Initialization:** Do **GradientDescent** for $f(\theta)$, $T', \eta \rightarrow \theta_1$
2. For $k = 1$ to T do:
 - a) **Compute Descent Direction estimator:**
 - [Loop 1] For $i = 1$ to S_1 do:
 - Base Case for Estimator i-th:** $X_{i,0} = \nabla f(\theta_k)$.
 - [Loop 2] For $j = 1$ to S_2 do:
 - Sample $\widetilde{H}_{i,j}(\theta_k)$ uniformly from $\{H_t(\theta_k) \mid t = \overline{1, m}\}$,
 - $X_{i,j} = \nabla f(\theta_k) + [I - \widetilde{H}_{i,j}(\theta_k)] X_{i,j-1}$.
 - [End Loop 2]
 - [End Loop 1]

- b) **Average of Descent direction estimators:** $X_k = \frac{1}{S_1} \sum_{i=1}^{S_1} X_{i,S_2}$.
- c) **Update Parameter:** $\theta_{k+1} = \theta_k - X_k$.

OUTPUTS: θ_{T+1}

Algorithm Remark:

1. The main process – computing Descent Direction estimator – is carried out in **Step 2 – Loop 2**.
2. In **Step 2 – Loop 1**, we compute S_1 number of Descent Direction estimators in order to build a more accurate estimator of $H^{-1}(\theta_k)$, by averaging these estimators.
3. The GradientDescent sub-routine carried out at **Step 1** is called the “Warm-up” step. This is to make the initial estimator of θ accurate “enough” so that LiSSA can converge in linear time.

The definition of “accurate enough” is stated rigorously in **Theorem 3.3** of [7]. In the scope of this paper, we just need to know that such sub-routine is proved to not affect the overall convergence rate of LiSSA.

4. On the convergence rate of LiSSA, under some regularity conditions of the objective function stated in **Section 2.1** of [7], which Generalized Linear Model is proved to satisfy, and suitable input parameters are taken, we have:

$$\|\theta_{k+1} - \theta^*\| \leq \frac{\|\theta_k - \theta^*\|}{2}$$

by **Theorem 3.3** of [7], this is a linear convergence rate. In other words, in the case of our MSE function, the objective function of Linear Regression, LiSSA has linear convergence rate.

To conclude the theoretical **Part 2** of this paper, we’ll show and discuss some very good results of this state-of-the-art LiSSA algorithm.

Figure 1 illustrates LiSSA superior convergence rate in compare to SAGA and SVRG, one of the best First-order methods in terms of convergence rate.

Figure 2 illustrates LiSSA superior running time in compare to Newton’s Method and NewSamp, two very popular Second-order methods, despite having lower convergence rate.

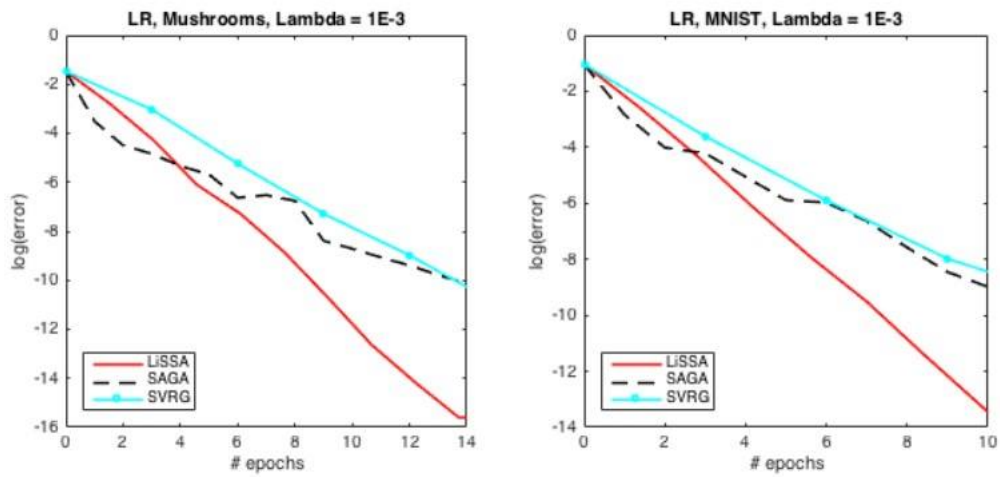


Figure 1. LiSSA vs SAGA vs SVRG in convergence rate.

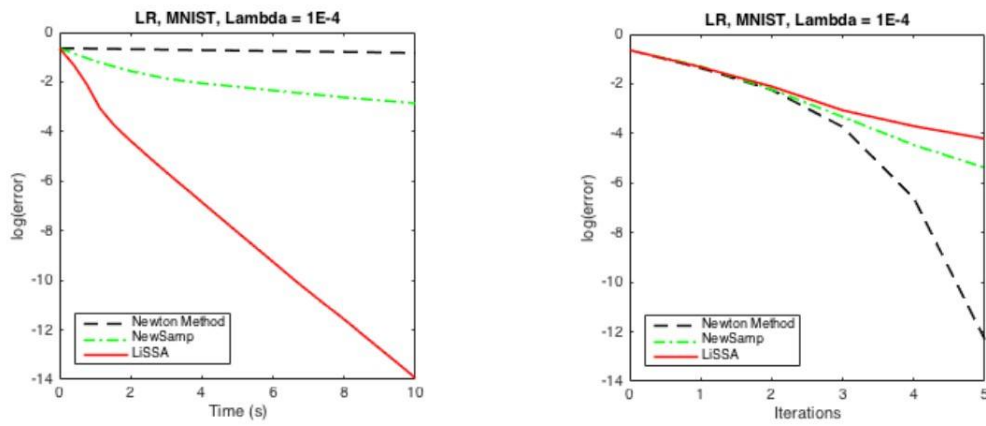


Figure 2. LiSSA vs Newton Method vs NewSamp on the MINST dataset.

Part 3. Implement a Linear Regression Learner

A. Overview

In this section, we will build a linear regression learner using Stochastic Gradient Descent, and apply it on the Wine Quality dataset to measure its performance.

About the Wine Quality dataset

From the “Wine Quality Dataset” with given chemical measures of each wine, we want to build a model that can give predictions for the quality of White Wine. This can be framed as a regression problem. There are 4,898 observations with 11 input variables and 1 output variable.

The input variables are:

1. Fixed acidity.
2. Volatile acidity.
3. Citric acid.
4. Residual sugar.
5. Chlorides.
6. Free sulfur dioxide.
7. Total sulfur dioxide.
8. Density.
9. pH.
10. Sulphates.
11. Alcohol.

And the output variable is, **Quality** (score between 0 and 10).

About the Least Square problem for this data-set

Our objective function is Root Mean Square Error

$$MSE = \frac{1}{4898} \sum_{i=1}^{4898} (\theta^T \cdot x_i - y_i)^2,$$

with:

- θ is the coefficient vector;
- $x_i, i = \overline{1, 4898}$ are the input variables vectors, represented by the first 11 columns of each row;
- $y_i, i = \overline{1, 4898}$ are the output variables, represented by the last column of each row.

About the Program

Our program goes with a .csv file named: “**winequality-white.csv**”.

First, the dataset is loaded using the “**load_csv()**” function. Then:

1. “**str_column_to_float()**” transforms the data from String to Numeric.
2. “**dataset_minmax()**” and “**normalize_dataset()**” is used to normalized all values, making the model faster to train.

We’ll use k-fold cross validation to estimate the performance of our model. The performance metrics is RMSE, averaged across all k folds. These functionalities is coded in “**cross_validation_split()**”, “**rmse_metric()**” and “**evaluate_algorithm()**” functions. Finally, at the core of our program, we will use “**predict()**”, “**coefficients_sgd()**” and “**linear_regression_sgd()**” functions to train the model.

B. Main Components and Algorithms

B.1. FUNCTION “**predict()**”

Goal of the function. Make prediction from an input vector using the trained Linear Regression model.

INPUT. Input vector x and a Linear Regression model represented by its coefficient vector θ .

OUTPUT. Predicted output \hat{y} .

Algorithm.

- Initially, set $\hat{y} = \text{coefficient}(0)$.
- Calculate $\hat{y} = \hat{y} + \sum_{i=0}^{10} \text{coefficients}(i+1) \times x(i)$.

B.2. FUNCTION “**coefficients_sgd()**”

Goal of the function. Using SGD to compute the coefficients of the Linear Regression Model.

INPUT. Training dataset, learning rate, the number of epochs.

OUTPUT. Coefficients of the Linear Regression model.

Algorithm.

- Initialize θ (the coefficients) to be Zero vector.
- While the number of executed epochs is in range, for each epoch: Set MSE equals 0. For each data point x_i in the training dataset:
 - Calculate \hat{y}_i using **predict()** with current coefficients and input x_i .
 - Calculate the error at data-point x_i : $\hat{y}_i - y_i$.
 - Update the MSE.
 - Update the coefficients $\theta \leftarrow \theta - \text{learning_rate} \cdot \text{error} \cdot x_i$.

B.3. FUNCTION “linear_regression_sgd()”

Goal of the function. Build Linear Regression model on the Training set. Making predictions on the Test set.

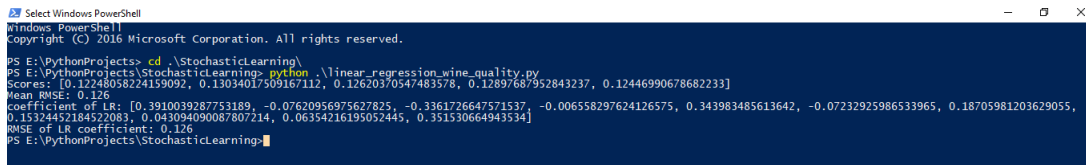
INPUT. Train dataset, Test dataset, learning rate, number of training epochs.

OUTPUT. Predictions on the Test dataset. *Algorithm.*

- Use function “**coefficients_sgd()**” with inputs: train_dataset, learning_rate and training_epochs to build the Linear Regression model.
- Use the trained model’s coefficients and function “**predict()**” with input test_dataset to output predictions.

C. Output and Conclusions

Run the program **linear_regression_wine_quality.py**, we observe the output as follow



```

Select Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS E:\PythonProjects> cd .\StochasticLearning\
PS E:\PythonProjects\StochasticLearning> python .\linear_regression_wine_quality.py
Scores: [0.12248058224159092, 0.13034017509167112, 0.12620370547483578, 0.12897687952843237, 0.12446990678682233]
Mean RMSE: 0.126
Coefficient of LR: [0.3910039287753189, -0.07620956975627825, -0.3361726647571537, -0.006558297624126575, 0.343983485613642, -0.07232925986533965, 0.18705981203629055,
0.15324452184522083, 0.043094090087807214, 0.06354216195052445, 0.351530664943534]
RMSE of LR coefficient: 0.126
PS E:\PythonProjects\StochasticLearning>

```

Let’s describe the outputs in greater detail.

RESULT 1. RMSE for each of the 5 cross–validations:

$$\begin{bmatrix} \text{1st_fold} \\ \text{2nd_fold} \\ \text{3rd_fold} \\ \text{4th_fold} \\ \text{5th_fold} \end{bmatrix} = \begin{bmatrix} 0.12 \\ 0.13 \\ 0.13 \\ 0.13 \\ 0.12 \end{bmatrix}$$

RESULT 2. Average RMSE: The average RMSE is 0.126. This is lower than the baseline value of 0.148.

RESULT 3. Best estimation of the coefficients:

$$[\text{bias_term } 0.39 \quad -0.08 \quad -0.34 \quad 0.01 \quad 0.34 \quad -0.07 \quad 0.19 \quad 0.15 \quad 0.04 \quad 0.06 \quad 0.35]$$

The RMSE of this coefficient vector is 0.126.

Part 4. Final Conclusions

The final section of this paper is considering what this paper has got and what I have got.

What did this paper get?

This paper supported us an overview to Stochastic approach in Optimization through the specific example of Linear Regression.

We started from basic concepts in Machine Learning and Least Square problems. These have been demonstrated intuitively through the series of practical examples.

The problem of Least Square led to useful and popular methods: First-order (Gradient Descent) and Second-order (Newton method), which use the idea of “**following the opposite direction of gradient**”.

Most important, we provided the **Stochastic approach** for both First-order and Second-order, in order to reduce the number of operations and timing.

Besides giving the algorithms, this paper also clarified the intuitions behind and the proves for the algorithms with both of advantages and disadvantages of each.

Finally, an example with real data-set is used to illustrate for Stochastic Gradient Descent.

What did I get?

Machine Learning is a really strange field to me. By doing this project I did learn a lot.

- What is Machine Learning? Purpose, examples and basic concepts in it.
- The role of Optimization in Machine Learning, especially Least Square Problem.
- Two related domains is convergence rate and convexity, respectively, one is used in estimating an algorithm and the other guarantees the working of algorithms.
- How to solve the problem of Least Square? An useful idea to solve this problem is following the opposite direction of Gradient, which presents in First-order and Second-order, respectively gets the linear convergence and quadratic convergence.
- However, to deal with problems of big data, we use the Stochastic approach which base on the idea of following the opposite direction of Gradient at one data point.
- How to program and solve a problem in practice.

Altogether, this project can bring a lot of things for readers and author, from the intuitions, algorithms to proves and practice.

References

- [1] <http://archive.ics.uci.edu/ml/datasets/Wine+Quality>
- [2] Aurélien Géron, Hands – On Machine Learning with Scikit – Learn & TensorFlow, First Edition, *O'Reily Media, Inc.*, March 2017.
- [3] Ryan Duy Luong – Lecture of “Optimisation and Decision Models: Nonlinear Programming”, *JVN Institute*, Fall 2017.
- [4] Ryan Tibshirani, 10 – 725: Optimization, Lecture 6: September 12, *Carnegie Mellon University, School of Computer Science*, Fall 2013.
- [5] Robert M. Freund, “Newton’s Method for Unconstrained Optimization”, *Massachusetts Institute of Technology*, February 2004.
- [6] Léon Bottou, ”Online Algorithms and Stochastic Approximations”, Online Learning and Neural Networks, *Cambridge University Press*, 1998.
- [7] Naman Agarwal, Brian Bullins, Elad Hazan, *Second – Order Stochastic Optimization for Machine Learning in Linear Time*, arXiv:1602.03943v5 [stat.ML], 30 Nov 2017.