Term Paper: How and Why to Use Stochastic Gradient Descent?

Minjie Fan
Department of Statistics
University of California, Davis

Abstract

Stochastic gradient descent is a simple but efficient numerical optimization method. It has been widely used in solving large-scale machine learning problems. This paper first shows how to implement stochastic gradient descent, particularly for ridge regression and regularized logistic regression. Then the pros and cons of the method are demonstrated through two simulated datasets. The comparison of stochastic gradient descent with a state-of-the-art method L-BFGS is also done.

1 Introduction

Let us consider the following scenario: $(\boldsymbol{X}_i, Y_i), i = 1, \cdots, n$ are samples from a population $(\boldsymbol{X}, Y)$, where Y is a scalar, $\boldsymbol{X}$ can be either a scalar or a vector, and n is the sample size. Usually Y is called the response and $\boldsymbol{X}$ is called the predictor. A parametric predictive function $f_{\boldsymbol{\theta}}$ is used to predict Y based on $\boldsymbol{X}$, where $\boldsymbol{\theta}$ is the parameter vector. The predictive accuracy is measured by the loss function $l(\widehat{Y}, Y) = l(f_{\boldsymbol{\theta}}(\boldsymbol{X}), Y)$. The parameter vector $\boldsymbol{\theta}$ is estimated by minimizing the risk function

$$E(l(\widehat{Y}, Y)) = \int_{\Omega} l(f_{\boldsymbol{\theta}}(\boldsymbol{X}), Y) dP,$$

where P is the underlying probability measure. However, since the distribution of $(\boldsymbol{X}, Y)$ is unknown in practice, we usually approximate the risk by the empirical one

$$E_n(l(\widehat{Y}, Y)) = \frac{1}{n} \sum_{i=1}^{n} l(f_{\boldsymbol{\theta}}(\boldsymbol{X}_i), Y_i) = \frac{1}{n} \sum_{i=1}^{n} l_i(\boldsymbol{\theta}).$$

This form of objective function, i.e., $\sum_i l_i(\boldsymbol{\theta})$, is very common in supervised learning. Additionally, the minimizer of the objective function is called an M-estimator.

Traditional numerical optimization methods, such as those we have learned in class: gradient descent, Newton's method and BFGS, can be used to minimize the objective function. For example, at each iteration, gradient descent updates $\boldsymbol{\theta}$ as follows (assuming that the gradient of l_i exists)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \sum_{i=1}^{n} \nabla l_i(\boldsymbol{\theta}^{(t)}),$$

where α_t is the step size (or called the learning rate). However, when the sample size is enormous, evaluating the gradient of the objective function, which is the sum of **all** the gradients of l_i, becomes time prohibitive. Given the limitation of the traditional methods, stochastic gradient descent (see Bottou (2012)) is proposed to speed up the computation through approximating the true gradient by the sum of a randomly selected subset of the gradients of l_i.

The contributions of this paper are: (i) showing how to implement stochastic gradient descent, particularly for ridge regression and regularized logistic regression; (ii) demonstrating the pros and

cons of stochastic gradient descent through comparing it with other traditional methods on simulated datasets.

The remainder of the paper is organized as follows. Section 2 introduces stochastic gradient descent. Its implementation on ridge regression and regularized logistic regression is shown in Section 3. We demonstrate stochastic gradient descent in Section 4 on simulated datasets. Section 5 concludes the paper.

2 Stochastic Gradient Descent

In its simplest form, stochastic gradient descent updates $\boldsymbol{\theta}$ as follows

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha_t \nabla l_i(\boldsymbol{\theta}^{(t)}),$$

where the index i is randomly selected at each iteration. In practice, we usually randomly shuffle the samples and then go through them sequentially. Note that the method can be straightforwardly extended to using the gradient of more than one sample, i.e., $\sum_{j=1}^{b} \nabla l_{i+j}(\boldsymbol{\theta}^{(t)})$, which is called mini-batch gradient descent. There are some advantages of mini-batching, such as more stable convergence and suitable for faster matrix operations and parallelization.

When the step size decreases according to $\sum_t \alpha_t^2 < \infty$ and $\sum_t \alpha_t = \infty$, e.g., $\alpha_t = \mathcal{O}(1/t)$, and under certain mild conditions, stochastic gradient descent converges almost surely to a local minimum, and even a global minimum for a convex objective function. As we have learned in class, under certain regularity conditions, gradient descent and Newton's method can achieve linear convergence and quadratic convergence, respectively. This is equivalent to that to achieve an accuracy of ϵ for the objective function, gradient descent takes $\mathcal{O}(\log(1/\epsilon))$ iterations, and Newton's method takes even fewer. However, stochastic gradient descent takes $\mathcal{O}(1/\epsilon)$ iterations, which seems to be exponentially worse than gradient descent. Nonetheless, when n is sufficiently large, assuming that the time complexity of calculating the gradient of one sample is a constant C, the total time complexity of stochastic gradient descent is $\mathcal{O}(C/\epsilon)$, which is smaller than that of gradient descent, $\mathcal{O}(nC\log(1/\epsilon))$.

3 Implementation: Ridge Regression and Regularized Logistic Regression

In statistics and machine learning, regularization is usually used to solve an ill-posed problem or to prevent overfitting. The latter is originated from minimizing the empirical risk instead of the true one. Although stochastic gradient descent is designed for datasets with a large sample size, a probably large number of features make it necessary to incorporate regularization into the empirical risk to reduce the generalization error. In the sequel, we shall present the implementation of stochastic gradient descent on ridge regression and regularized logistic regression with ℓ_2-norm. Note that the method is not restricted to these two problems. Other problems, such as Lasso (Shalev-Shwartz and Tewari, 2011), support vector machines (Menon, 2009) and neural networks (Bottou, 1991) can be solved by stochastic gradient descent or its variants.

3.1 Ridge Regression

Ridge regression is formulated as

$$(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) = \arg\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - \boldsymbol{x}_i^T \boldsymbol{\beta})^2 + \frac{\lambda}{2} \|\boldsymbol{\beta}\|_2^2,$$

where $\boldsymbol{x}_i^T$ is the i-th row vector of the $n \times p$ (p is the number of features) design matrix $\boldsymbol{X}$, $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_p)^T$, and λ is the tuning parameter that controls the regularization. It is clear that for all $\lambda > 0$, the objective function is strictly convex, which implies that its minimizer exists and is unique. We write the objective function as

$$(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) = \arg\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{n} \sum_{i=1}^{n} l_i(\boldsymbol{\beta}),$$

where $l_i(\boldsymbol{\beta}) = \frac{1}{2}(y_i - \beta_0 - \boldsymbol{x}_i^T\boldsymbol{\beta})^2 + \frac{\lambda}{2}\|\boldsymbol{\beta}\|_2^2$. The gradient of l_i is

$$\nabla l_i(\boldsymbol{\beta}) = (-y_i + \beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})(1, \boldsymbol{x}_i^T)^T + \lambda(0, \boldsymbol{\beta}^T)^T. \tag{1}$$

The minimizer of the objective function can be obtained by setting $\sum_i \nabla l_i(\boldsymbol{\beta})$ as 0. W.l.o.g., assuming $\beta_0 = 0$, the minimizer has the closed form

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_n)^{-1}\boldsymbol{X}^T\boldsymbol{y},$$

where $\boldsymbol{y} = (y_1, \cdots, y_n)^T$ is the vector of responses. However, the time complexity of inverting (or more exactly, calculating its Cholesky decomposition) a $p \times p$ matrix is $\mathcal{O}(p^3)$, which becomes time prohibitive when p is enormous. In comparison, the time complexity of stochastic gradient descent is $\mathcal{O}(p/\epsilon)$, which will be smaller than $\mathcal{O}(p^3)$ if the number of iterations is less than $\mathcal{O}(p^2)$. Note that here we do not consider the case when $\boldsymbol{X}$ is sparse.

3.2 Regularized Logistic Regression

Regularized logistic regression is formulated as

$$\begin{aligned}(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) &= \arg\min_{\beta_0, \boldsymbol{\beta}} -\frac{1}{n}\log(\text{likelihood function}) + \frac{\lambda}{2}\|\boldsymbol{\beta}\|_2^2 \\ &= \arg\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{n}\sum_{i=1}^{n}\left[\log\left\{1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})\right\} - y_i(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})\right] + \frac{\lambda}{2}\|\boldsymbol{\beta}\|_2^2.\end{aligned}$$

For all $\lambda > 0$, given that its Hessian matrix is positive definite, the objective function is strictly convex, which implies that its minimizer exists and is unique. We write the objective function as

$$(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) = \arg\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{n}\sum_{i=1}^{n} l_i(\boldsymbol{\beta}),$$

where $l_i(\boldsymbol{\beta}) = \log\left[1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})\right] - y_i(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta}) + \frac{\lambda}{2}\|\boldsymbol{\beta}\|_2^2$. The gradient of l_i is

$$\nabla l_i(\boldsymbol{\beta}) = [-y_i + S(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})](1, \boldsymbol{x}_i^T)^T + \lambda(0, \boldsymbol{\beta}^T)^T, \tag{2}$$

where $S(\cdot)$ denotes the sigmoid function, i.e.,

$$S(x) = \frac{\exp(x)}{1 + \exp(x)}.$$

An interesting observation is that the gradients (1) and (2) share the same form except that the sigmoid function in (2) is replaced by the identity function in (1).

3.3 Implementation

For both ridge regression and regularized logistic regression, the tuning parameter λ can be determined by cross-validation. In the sequel, we shall assume that λ is known and fixed since our focus is on minimizing the objective function.

As one of the disadvantages of stochastic gradient descent, choosing a proper step size is challenging. We simply use the form suggested by Bottou (2012)

$$\alpha_t = \frac{\alpha_0}{1 + \alpha_0\lambda t},$$

where t is the iteration number starting from 0 and α_0 is the initial step size, which can be tuned using a small subset of the dataset. See Schaul *et al.* (2013) for an adaptive method that does not need any manual tuning but compares favorably with an "ideal SGD".

By default, the initial value of $(\beta_0, \boldsymbol{\beta})$ is a zero vector. For a warm start, one may specify it as the estimate obtained by other methods using a small subset of the dataset. The total number of iterations is set as $1e6$ according to Pedregosa *et al.* (2011), which empirically ensures the convergence of stochastic gradient descent. All my codes are written in *Julia*. It is worth pointing out that there is only one *Julia* package available for stochastic gradient descent, which is called *SGDOptim*.

4 Simulation Study

This section applies stochastic gradient descent to two simulated datasets: one for ridge regression and the other for regularized logistic regression.

4.1 Simulated Dataset 1: Ridge Regression

We simulate data from a linear model with i.i.d. Gaussian random errors. The coefficient vector $(\beta_0, \boldsymbol{\beta}) = (3, -4, 5)$. The standard deviation of the random errors is 0.1. All the entries of the design matrix $\boldsymbol{X}$ are simulated from $\mathcal{N}(0, 1)$ independently. Since the aim of this simulation study is to check how stochastic gradient descent converges to the global minimum, we only specify n as 100. Given that the number of features is only 2, the tuning parameter λ is fixed at $1e{-}4$. By trial and error, we find $\alpha_0 = 0.3$ to be suitable. From Figure 1, we can see that the objective function moves close to the minimum very fast, and then oscillates around it. As the iteration number increases, the pronounced oscillation at the very beginning is gradually damped to be negligible. This confirms the aforementioned almost sure convergence of stochastic gradient descent. It is notable that the convergence to the minimum takes a very long time. In practice, one may use fewer iterations when the running time is the bottleneck.

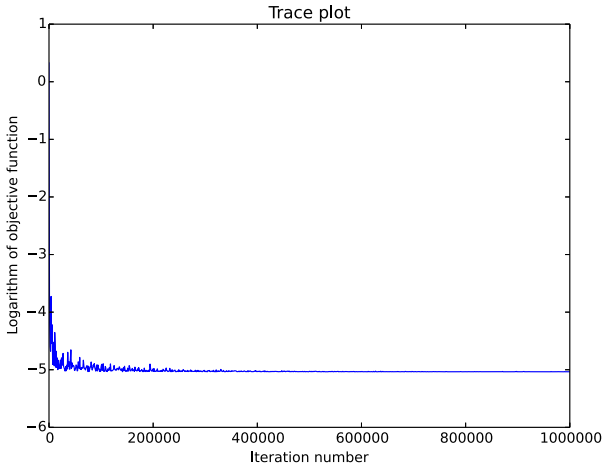


Figure 1: Trace plot of the logarithm of the objective function (only every 1000 iterations).

4.2 Simulated Dataset 2: Regularized Logistic Regression

We turn to compare the computational speed of stochastic gradient descent and L-BFGS on a simulated dataset from a logistic regression model. L-BFGS is a limited-memory version of BFGS. It achieves super-linear convergence with the cost per iteration smaller than Newton's method. Due to these advantages, it has been a popular algorithm for parameter estimation in machine learning. We use the *Julia* package *Optim* to run BFGS.

The experiment is done under a high-dimensional setting, where $n = 1e6$ and $p = 100$. All the entries of the coefficient vector and the design matrix are simulated from $\mathcal{N}(0, 1)$ independently. The tuning parameter λ is fixed at $1e - 4$. By trial and error, we find $\alpha_0 = 0.005$ to be suitable.

In terms of the computational speed, stochastic gradient descent takes 9.45 seconds, while L-BFGS takes 151.49 seconds, which is much slower than the former. Figure 2 contains the comparison among the true coefficients and the estimated coefficients by stochastic gradient descent and L-BFGS. The RMSE of these two estimated coefficients are almost the same, which are 0.04926 and 0.04924, respectively. Thus, stochastic gradient descent and L-BFGS achieve the equivalent accuracy but the former performs almost 20 times faster than the latter.

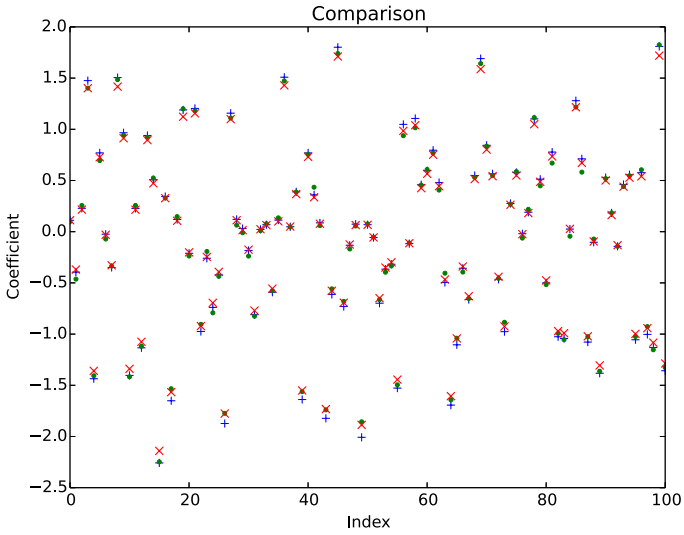


Figure 2: Comparison among the true coefficients (+) and the estimated coefficients by stochastic gradient descent (.) and L-BFGS (x).

5 Conclusion

In this paper, we have shown how to implement stochastic gradient descent. We have also demonstrated the superiority of stochastic gradient descent to other traditional methods for large-scale machine learning problems. It is efficient and easy to implement, but tuning the step size parameter needs some efforts.

References

Bottou, L. (1991) Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, **91**.

Bottou, L. (2012) Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, 421–436. Springer.

Menon, A. K. (2009) Large-scale support vector machines: algorithms and theory. *Research Exam, University of California, San Diego*, 1–17.

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

Schaul, T., Zhang, S. and Lecun, Y. (2013) No more pesky learning rates. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 343–351.

Shalev-Shwartz, S. and Tewari, A. (2011) Stochastic methods for ℓ_1-regularized loss minimization. *The Journal of Machine Learning Research*, **12**, 1865–1892.

In [9]:

```
# loading packages
using PyPlot
using Distributions
using Optim
```

In [10]:

```
function identity(x::Float64)
    # identity function
    return x
end
```

Out[10]:

```
identity (generic function with 1 method)
```

In [11]:

```
function sigmoid(x::Float64)
    # sigmoid function
    y = exp(x)/(1+exp(x))
    return y
end
```

Out[11]:

```
sigmoid (generic function with 1 method)
```

In [12]:

```
function cal_obj(X::Matrix{Float64}, y::Vector{Float64}, beta::Vector{Float64},
link_fun::Function, lambda::Float64)
    # calculate the objective function
    f = 0
    (n, p) = size(X)
    if link_fun==identity
        for i = 1:n
            lin_pred = beta[1]+dot(beta[2:end], vec(X[i, :]))
            f = f+(y[i]-lin_pred)^2
        end
        f = f/2
    else
        for i = 1:n
            lin_pred = beta[1]+dot(beta[2:end], vec(X[i, :]))
            f = f+log(1+exp(lin_pred))-y[i]*lin_pred
        end
    end
    f = f/n+lambda/2*norm(beta[2:end])^2
    return f
end
```

Out[12]:

```
cal_obj (generic function with 1 method)
```

In [13]:

```
function sgd(X::Matrix{Float64}, y::Vector{Float64}, beta_init::Vector{Float64},
link_fun::Function, lambda::Float64,
    gamma0::Float64, max_iter=1e6, eval=false)
    # stochastic gradient descent for ridge regression and regularized (l2-norm)
logistic regression
    # learning rate gamma_t = gamma0/(1+gamma0*lambda*t)
    # make sure beta_init will be unchanged
    beta = copy(beta_init)
    (n, p) = size(X)
    t = 0
    beta_new = zeros(Float64, p+1)
    max_epoch = round(Int, ceil(max_iter/n))
    # stores eval
    f = zeros(Float64, max_epoch*n)
    for epoch = 1:max_epoch
        # random permutation
        idx = collect(1:n)
        shuffle!(idx)
        for i = 1:n
            t = t+1
            # when t=1, gamma_t=gamma0
            gamma_t = gamma0/(1+gamma0*lambda*(t-1))
            err_term = -y[idx[i]]+link_fun(beta[1]+dot(beta[2:end], vec(X[idx[i]
, :])))
            beta_new[1] = beta[1]-gamma_t*err_term
            beta_new[2:end] = beta[2:end]-gamma_t*lambda*beta[2:end]-gamma_t*err
_term*vec(X[idx[i], :])
            if eval
                f[t] = cal_obj(X, y, beta_new, link_fun, lambda)
            end
            for j = 1:p+1
                beta[j] = beta_new[j]
            end
        end
    end
    if eval
        return beta, f
    else
        return beta
    end
end
```

Out[13]:

```
sgd (generic function with 3 methods)
```

Simulation 1: Ridge Regression

In [14]:

```
# the underlying model coefficients
beta = [3.0, -4.0, 5.0]
```

Out[14]:

```
3-element Array{Float64,1}:
  3.0
 -4.0
  5.0
```

In [15]:

```
# generate 100 sample features
n = 100
X = randn(n, 2)
```

Out[15]:

```
100x2 Array{Float64,2}:
 -0.583171    1.2449
 -0.971005   -0.0210151
  1.69764     1.69948
  0.959458   -0.728807
  0.0222367   0.300559
  0.876112   -0.802153
 -1.32335     2.16682
 -0.757819   -0.106327
  0.0150875  -0.00861295
  0.921816   -0.56862
  0.107026    0.598436
  2.07681    -0.895444
 -0.429312    0.879739
  ⋮
  0.629477    0.581746
  0.141066   -0.0319641
  2.19861     0.254433
 -1.06322    -0.780303
 -0.250349   -0.806091
  0.979702   -0.697077
 -1.9782     -0.105999
  0.323679   -0.0586646
  0.454688    0.387208
  1.45067     1.35517
 -0.89092     0.244778
 -0.0228052   0.942253
```

In [16]:

```
# generate the responses, adding some noise
sig = 0.1
y = beta[1]+X*vec(beta[2:end])+sig*randn(n)
```

Out[16]:

```
100-element Array{Float64,1}:
 11.6012
  6.85773
  4.79993
 -4.55376
  4.43267
 -4.49798
 19.1141
  5.50015
  2.77666
 -3.41908
  5.71159
 -9.81765
  9.1724
  ⋮
  3.39515
  2.50364
 -4.59379
  3.27736
  0.0707259
 -4.47656
 10.3562
  1.44704
  3.07295
  3.96811
  7.71308
  7.94608
```

In [46]:

```
beta_init = zeros(Float64, 3)
lambda = 1e-4
(beta_hat, f) = sgd(X, y, beta_init, identity, lambda, 0.3, 1e6, true)
```

Out[46]:

```
([3.0036104107126866,-4.000567013341443,5.0191882015680305],[1.3861,
1.07736,1.10218,0.558367,0.347497,0.460148,0.456272,0.400298,0.38383
3,0.229814  …  0.00651245,0.00651253,0.00651623,0.00651569,0.0065158
2,0.00651584,0.00651721,0.00651752,0.00651314,0.00650683])
```

In [47]:

```
beta_hat
```

Out[47]:

```
3-element Array{Float64,1}:
  3.00361
 -4.00057
  5.01919
```

In [48]:

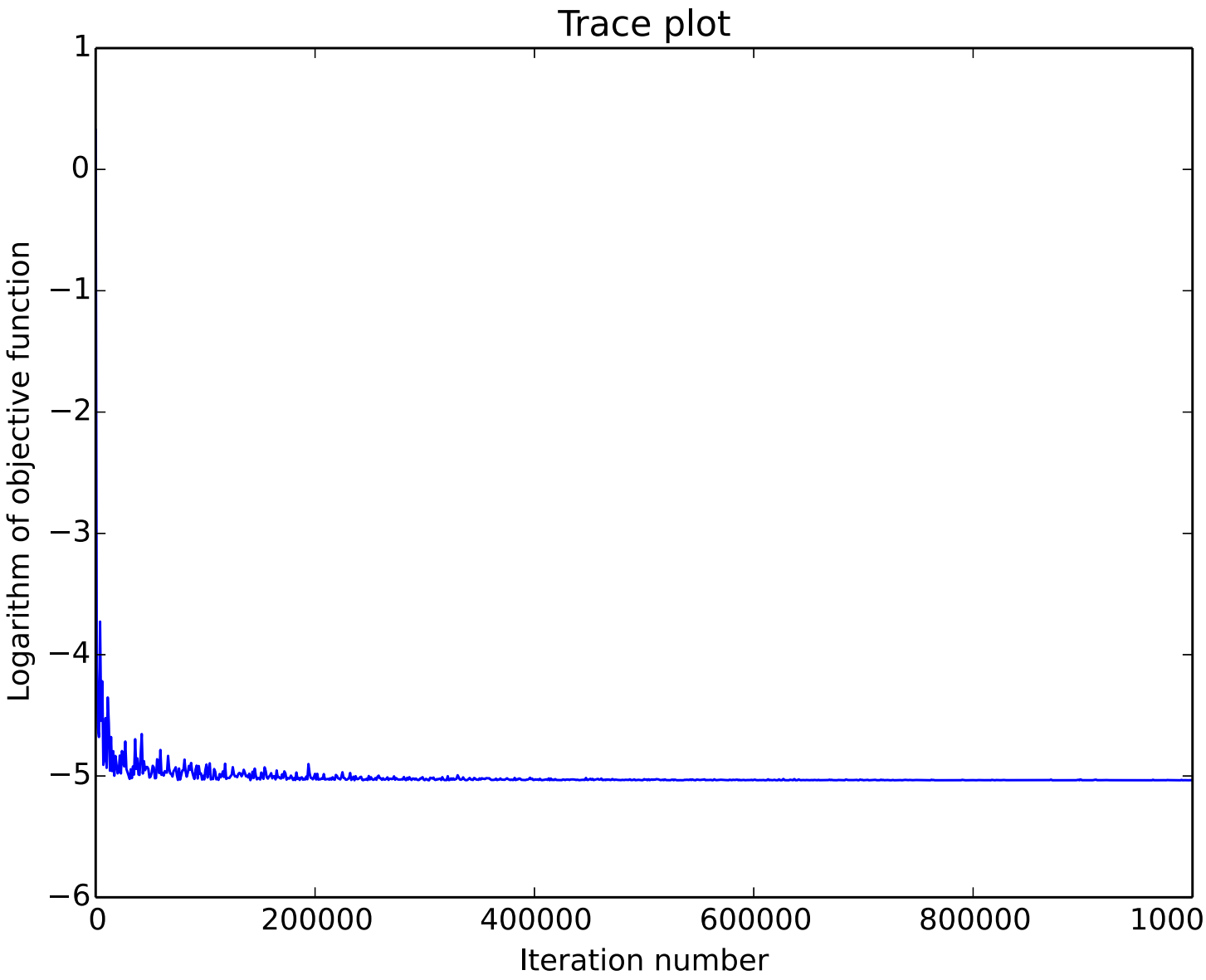
```
PyPlot.svg(true)
```

Out[48]:

```
true
```

In [54]:

```
plot(1:1000:1000000, log(f[1:1000:1000000]))
xlabel("Iteration number");
ylabel("Logarithm of objective function");
title("Trace plot");
savefig("trace_plot.eps")
```



Simulation 2: Logistic Regression

In [55]:

```
p = 100
beta = randn(p+1)
```

Out[55]:

```
101-element Array{Float64,1}:
  0.111365
 -0.39488
  0.228392
  1.47518
 -1.43635
  0.76896
 -0.024029
 -0.349056
  1.50484
  0.966028
 -1.40318
  0.227324
 -1.13426
  ⋮
 -1.38168
  0.528449
  0.182664
 -0.142819
  0.461183
  0.556181
 -1.05571
  0.576261
 -1.00278
 -1.13161
  1.80962
 -1.35791
```

In [56]:

```
n = 1000000
X = randn(n, p)
```

Out[56]:

```
1000000x100 Array{Float64,2}:
  1.61719    -1.65077      0.378564  …   0.80211      0.506168  -0.4
74577
  0.188394    0.323004     1.11977       1.68015      1.04115    0.7
62194
  0.279615   -1.01524      1.05282      -1.71778     -0.10364    0.7
65419
 -1.1565     -0.854652    -1.17783      -0.635115     0.345301  -2.1
8847
  1.32801     1.22534      0.526442     -0.187554    -1.04453   -0.5
27556
  2.16786     0.234011     1.18162   …  -0.839294    -0.089363  -1.7
6808
```

```
  0.1572     -0.439558    -0.648022     -2.15507      0.515458  -0.4
08599
 -1.23389     0.866207     0.313969      0.0811189    0.903874   1.3
6584
 -0.228185    0.0876585   -1.55348      -0.348265    -0.651498   0.1
19013
  0.41779     0.261066    -1.03086      -0.668181    -0.582049   0.0
27994
 -0.0838109   0.261878     0.804134  …   0.7902       0.506186  -1.2
0214
  1.07555     0.27597      0.981039     -0.440509    -0.525532   0.5
24447
 -0.579852   -0.00346613   1.54956      -0.416688    -1.42651    0.4
70442
  ⋮                                  ⋱
 -0.819617   -1.70523     -0.339743      0.691116    -0.19523    0.7
72021
  0.117846   -0.229756     1.05384      -0.521334     0.589964  -1.1
7779
  0.14268     1.06002      1.36355   …   0.441483     0.415082  -1.6
4918
 -0.34897    -1.37459      0.71608       2.42219      0.52555   -1.4
142
  0.340785   -0.696449    -1.01347      -0.00345933   0.776469  -0.7
58472
  0.126584   -1.83221      2.51231      -1.30306      1.11096    0.1
25271
  0.394158   -0.0757726   -0.536737      0.932371    -0.643962   1.2
616
  0.27297    -1.42776      1.31784   …   1.57487     -1.64266   -0.2
2704
  1.29409    -0.282394     0.216374      0.980583    -0.838287   0.4
7083
 -2.13803    -0.836115     0.266043      1.93147     -0.202607   0.8
94209
  1.93314     0.0399141    0.815048     -0.448002     0.790242   1.1
8888
 -0.955354    0.190652     1.43474      -1.16672      0.306917  -0.0
812812
```

In [57]:

```
lin_pred = beta[1]+X*vec(beta[2:end])
theta = zeros(n)
y = zeros(n)
for i = 1:n
    theta[i] = sigmoid(lin_pred[i])
    y[i] = rand(Bernoulli(theta[i]))
end
```

In [91]:

```
beta_init = zeros(Float64, p+1)
tic()
beta_hat = sgd(X, y, beta_init, sigmoid, lambda, 0.005)
toc();
```

```
elapsed time: 9.449696993 seconds
```

In [92]:

```
beta_hat
```

Out[92]:

```
101-element Array{Float64,1}:
  0.0985153
 -0.463078
  0.256002
  1.4018
 -1.40686
  0.694782
 -0.0716831
 -0.33605
  1.48509
  0.94303
 -1.41721
  0.255828
 -1.11927
  ⋮
 -1.36295
  0.518534
  0.183958
 -0.142634
  0.441405
  0.546126
 -1.02381
  0.604678
 -0.926639
 -1.15318
  1.82456
 -1.32623
```

Stochastic Gradient Descent vs L-BFGS

In [64]:

```
f_log_uni(beta) = cal_obj(X, y, beta, sigmoid, lambda)
```

Out[64]:

```
f_log_uni (generic function with 1 method)
```

In [65]:

```
function grd_log(X, y, beta, link_fun, lambda, g)
    (n, p) = size(X)
    for i = 1:p+1
        g[i] = 0
    end
    for i = 1:n
        err_term = -y[i]+link_fun(beta[1]+dot(beta[2:end], vec(X[i, :])))
        g[1] = g[1]+err_term
        g[2:end] = g[2:end]+lambda*beta[2:end]+err_term*vec(X[i, :])
    end
    for i = 1:p+1
        g[i] = g[i]/n
    end
    return 0
end
```

Out[65]:

```
grd_log (generic function with 1 method)
```

In [66]:

```
grd_log_uni(beta, g) = grd_log(X, y, beta, sigmoid, lambda, g)
```

Out[66]:

```
grd_log_uni (generic function with 1 method)
```

In [67]:

```
beta_init = zeros(Float64, p+1)
tic()
res = optimize(f_log_uni, grd_log_uni, beta_init, method = :l_bfgs)
toc();
```

```
elapsed time: 151.491975037 seconds
```

In [68]:

```
res
```

Out[68]:

```
Results of Optimization Algorithm
 * Algorithm: L-BFGS
 * Starting Point: [0.0,0.0, ...]
 * Minimum: [0.11265245979332834,-0.36984979396344736, ...]
 * Value of Function at Minimum: 0.141961
 * Iterations: 8
 * Convergence: true
   * |x - x'| < 1.0e-32: false
   * |f(x) - f(x')| / |f(x)| < 1.0e-08: false
   * |g(x)| < 1.0e-08: true
   * Exceeded Maximum Number of Iterations: false
 * Objective Function Calls: 33
 * Gradient Call: 33
```

In [69]:

```
beta_hat2 = res.minimum
```

Out[69]:

```
101-element Array{Float64,1}:
  0.112652
 -0.36985
  0.215302
  1.40181
 -1.35973
  0.727253
 -0.030653
 -0.327741
  1.41668
  0.913104
 -1.33958
  0.214068
 -1.07549
  ⋮
 -1.30586
  0.500618
  0.162869
 -0.131104
  0.439804
  0.526872
 -0.99848
  0.54044
 -0.941634
 -1.08322
  1.71919
 -1.28691
```

In [93]:

```
sqrt(mean((beta-beta_hat).^2))
```

Out[93]:

```
0.04926078945866942
```

In [71]:

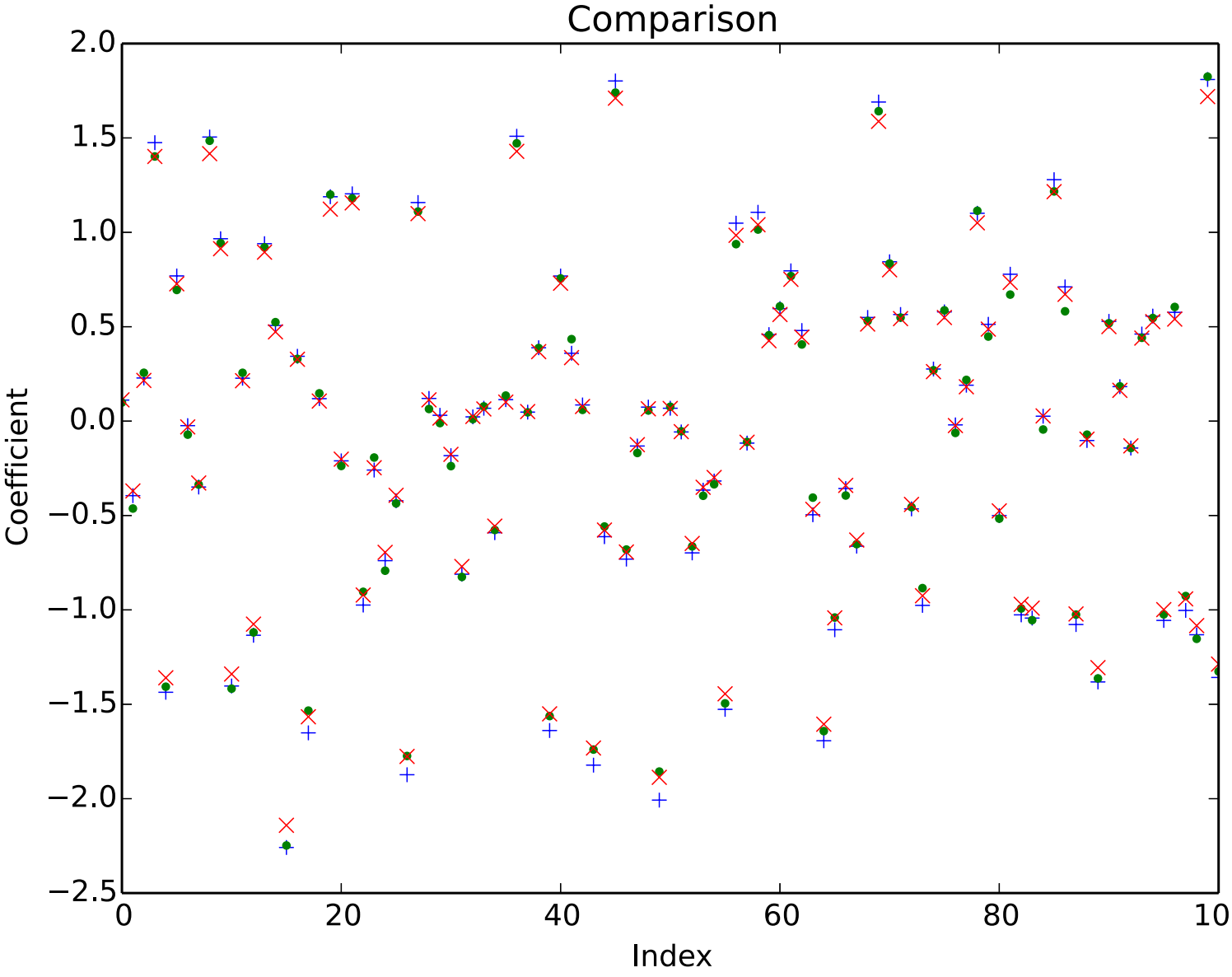
```
sqrt(mean((beta-beta_hat2).^2))
```

Out[71]:

```
0.04923705873902063
```

In [113]:

```
plot(beta, "+")
hold(true)
plot(beta_hat, ".")
plot(beta_hat2, "x")
xlabel("Index")
ylabel("Coefficient")
title("Comparison")
savefig("comp.eps")
```


Comparison
Coefficient
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
20
40
60
80
100
Index

In [ ]: