Evaluating performance of various optimization algorithms on a binary classification problem

Achyuth Nandikotkur

January 15, 2022

Abstract

Machine learning optimization is the process of adjusting the chosen model parameters, by minimizing a cost function that represents the error between the model output and the true label. To accomplish this, several optimization algorithms were taught during the course of ECE-503 and I was able to successfully apply them to a classification problem called Glasses or No Glasses. In this report, I will present the results and also draw a contrast between these various algorithms based on their accuracy and training times.

1 Introduction

The objective of my project is to determine if a person in a picture is wearing glasses or not. This evidently is a two-class classification problem that can be modeled using the logistic regression classifier. The cost function for the same is shown below

$$e(\hat{\boldsymbol{w}}) = \frac{1}{P}\sum_{p=1}^{P} log(1 + e^{-y_p \hat{\boldsymbol{w}}^T X_p}) \ - \ (1.0)$$

P = no. of samples; $X_p = p^{th}$ sample; y_p = true label of p^{th} sample

The original dimension of the images used in this project is 1024 x 1024. However, the data set also provided latent vector representations of these images, which I ultimately ended up using for minimization. The dataset originally contained a total of 4500 samples, and I split them into training and testing datasets using the 70-30 split ratio which resulted in 3150 samples for training and 1350 for testing. Subsequently, this training data set was used to minimize the above logistic regression cost function using the following optimization algorithms

- CG (Conjugate Gradient Descent).
- BFGS (Broyden–Fletcher–Goldfarb–Shanno)
- ML-BFGS (Memory-Less BFGS)
- Netwon's method.

1.0.1 Logistic Regression

Unlike regression, in a classification task, the labels in the data set are discrete, signifying certain classes or categories, and the learning aims to infer a class for a previously unseen input sample. We begin with a method for two-class (also called binary) classification problems. Consider a data set (x_p, y_p), p = 1,2,3...,P where each input sample x_p is associated with a label y_p that signifies which of the two classes, denoted by P and N, sample x_p belongs to. As such, label y_p is signed to value 1 if $x_p \in$ P, or assigned to value -1 if $x_p \in$ N. The core of a classification algorithm is to construct a classifier that takes an input x and infers its class according to

$$\tilde{y} = \text{sign}\left(\boldsymbol{w}^T\boldsymbol{x} + b\right) = \text{sign}\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}\right) \ - \ (1.1)$$

where $\hat{\boldsymbol{w}}$ and $\hat{\boldsymbol{x}}$ are defined as usual, and the sign function is defined by

$$\text{sign}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0 \end{cases}$$

Thus classification is performed as follows:

$$\begin{array}{ll} \boldsymbol{x} \in \boldsymbol{Q} & \text{if } \boldsymbol{w}^T\boldsymbol{x} + b > 0 \\ \boldsymbol{x} \in \mathcal{N} & \text{if } \boldsymbol{w}^T\boldsymbol{x} + b < 0 \end{array}$$

Needless to say, for the classifier to work, its weight $\boldsymbol{w}$ and bias b will have to be tuned to fit the given data. On comparing with the regression model that predicts a continuous output for an input $\boldsymbol{x}$, the nonlinear operation sign$(\cdot)$ in classifier (1.1) takes into account the specific form of binary label $\tilde{y} \in \{1, -1\}$. As explained below, the involvement of nonlinear and non-smooth function sign$(\cdot)$ in the classifier has a major impact on how parameter $\hat{\boldsymbol{w}}$ is to be trained using the given data. The first step in developing an effective training technique for tuning model (1.1) is to relax (approximate, that is) non-smooth sign(z) with a smooth function

$$\tanh(z) = 2\sigma(z) - 1 = \frac{2}{1 + e^{-z}} - 1 \; - \; (1.2)$$

where tanh$(\cdot)$ stands for hyperbolic-tangent which is closely related to the logistic sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \; - \; (1.3)$$

Now an L_2-loss based on smooth approximation of sign$(\cdot)$ for two-class classification can be constructed as

$$E_2(\hat{w}) = \frac{1}{P} \sum_{p=1}^{P} \left[\tanh\left(\hat{w}^T \hat{x}_p\right) - y_p\right]^2 \; - \; (1.4)$$

In principle, parameter $\hat{\boldsymbol{w}}$ can be properly turned by minimizing $E_2(\hat{\boldsymbol{w}})$ with respect to $\hat{\boldsymbol{w}}$. The minimization is however technically challenging because function $E_2(\hat{w})$ is nonconvex and hence admits local solutions of inferior quality. An alternative loss that takes the advantage of having specific discrete labels can be constructed based on the concept of log error. The log error for an individual input sample $\hat{x}_p$ is defined by

$$\begin{array}{ll} -\log\left(\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)\right) & \text{if } y_p = 1 \; - \; (1.5) \\ -\log\left(1 - \sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)\right) & \text{if } y_p = -1 \; - \; (1.6) \end{array}$$

Notice first that because the logistic sigmoid $\sigma(z)$ (see (1.3)) is always strictly between 0 and 1 , the log error is always positive. To see what the log error stands for, consider a data pair $(\boldsymbol{x}_p, y_p)$ with $y_p = 1$, the log error for this data pair is computed using (1.5). An adequately tuned $\hat{\boldsymbol{w}}$ in this case would yield $\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p > 0$ which implies $\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right) > 0.5$, and a more positive $\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p$ would yield a $\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)$ closer to 1 and hence a smaller log error. On the other hand, an inadequately tuned $\hat{\boldsymbol{w}}$ would produce a $\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p < 0$ and hence $\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right) < 0.5$, and a more negative $\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p$ would yield a $\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)$ closer to 0 and hence a larger log error. For a data pair $(\boldsymbol{x}_p, y_p)$ with $y_p = -1$, the log error is computed using (1.6). A similar analysis shows that an adequately tuned $\hat{\boldsymbol{w}}$ also leads to a smaller log error relative to an inadequately tuned $\hat{\boldsymbol{w}}$. Therefore we conclude that the log error in (1.6) may serve to measure the performance of model (1.1) at point $\hat{x}_p$.

Furthermore, by using the fact $1 - \sigma(z) = \sigma(-z)$ to the item in (1.6), the log error in (1.6) can be expressed as

$$\begin{cases} -\log\left(\sigma\left(\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)\right) & \text{if } y_p = 1 \\ -\log\left(\sigma\left(-\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)\right) & \text{if } y_p = -1 \end{cases}$$

which can in turn be combined into a single expression as

$$-\log\left(\sigma\left(y_p\hat{\boldsymbol{w}}^T\hat{\boldsymbol{x}}_p\right)\right) \; - \; (1.7)$$

In conjunction with (1.3) the log error in (1.7) can be written as

$$\log\left(1+e^{-y_p\hat{x}_p^\top\hat{w}}\right)$$

Therefore, it makes sense to tune the model parameter $\hat{\boldsymbol{w}}$ by minimizing the average of the above point-wise log error over all data points, which is given by

$$E_L(\hat{w})=\frac{1}{P}\sum_{p=1}^{P}\log\left(1+e^{-y_p\hat{x}_p^T\hat{w}}\right)$$

The gradient and Hessian of $E_L(\hat{\boldsymbol{w}})$ are given by

$$\nabla E_L(\hat{\boldsymbol{w}})=\frac{1}{P}\sum_{p=1}^{P}\nabla\log\left(1+e^{-y_p\hat{\boldsymbol{x}}_{\hat{w}}^T\hat{\boldsymbol{\omega}}}\right)=-\frac{1}{P}\sum_{p=1}^{P}\frac{y_p\hat{\boldsymbol{x}}_p}{1+e^{y_p\hat{\boldsymbol{x}}_p^T\hat{\boldsymbol{w}}}}$$

and

$$\nabla^2 E_L(\hat{\boldsymbol{w}})=\frac{1}{P}\sum_{p=1}^{P}\nabla^2\log\left(1+e^{-y_p\hat{x}_p^T\hat{w}}\right)=\frac{1}{P}\sum_{p=1}^{P}\frac{e^{y_p\hat{x}_p^T\hat{w}}}{\left(1+e^{y_p\hat{x}_p^T\hat{w}}\right)^2}\hat{\boldsymbol{x}}_p\hat{\boldsymbol{x}}_p^T$$

1.0.2 Conjugate Gradient Descent

Conjugate-direction methods are developed for quadratic optimization problems and then extended to non-quadratic optimization problems. A class of especially effective conjugate-direction methods for non-quadratic optimization is that of conjugate gradient (CG) methods. The search direction of a CG algorithm assumes the form

$$dk=-\nabla f(X_k)+\beta_k d_{k-1}\ -\ (1.8)$$

where the initial β_k is set to $\beta_0=0$ which implies that the initial search direction d_0 is simply a GD direction $d_0=-\nabla f(X_k)$. However B_k for k $>$ 0 are nonzero and good choices of B_k are the results of many years of research. In this project, I have used the Polak-Ribière-Polyak-plus (PRP+) method to determine the search direction.

$$\beta_k^{PRP+}=max(\beta_k^{PRP},0)\ -\ (1.9)$$

$$\beta_k^{PRP}=\frac{\nabla f(x_k^T)\gamma_{k-1}}{||\nabla f(x_{k-1})||_2^2}\ -\ (1.10)$$

1.0.3 Newton

A second-order method known as the Newton (also known as the Newton-Raphson) method can be deduced using the quadratic approximation based on the Taylor series:

$$f(\boldsymbol{x}+\boldsymbol{\delta})\approx f(\boldsymbol{x})+\nabla f(\boldsymbol{x})^T\boldsymbol{\delta}+\frac{1}{2}\boldsymbol{\delta}^T\nabla^2 f(\boldsymbol{x})\boldsymbol{\delta}\ -\ (1.11)$$

It is important to regard all component functions in (1.11) as functions of $\boldsymbol{\delta}$ and regard $\boldsymbol{x}$ as a temporarily fixed point (that represents the current design waiting for your improvement!). The idea of Newton algorithm is minimizing the quadratic function on the right-hand side of (1.11) rather than minimizing the original function of the left-hand side of (1.11). The technique obviously makes sense because in practice minimizing a quadratic function is much easier than minimizing the original function while (1.11) indicates that these two pieces are actually pretty close to each other as long as $\boldsymbol{\delta}$ is small.

Let us consider a general non-quadratic function $f(x)$ that is being minimized, and suppose we are in the iteration of Newton algorithm to update the current iterate $\boldsymbol{x}_k$ to the next iterate $\boldsymbol{x}_{k+1}$. The quadratic approximation (1.11) then becomes

$$f\left(\boldsymbol{x}_k+\boldsymbol{\delta}\right)\approx f\left(\boldsymbol{x}_k\right)+\nabla f\left(\boldsymbol{x}_k\right)^T\boldsymbol{\delta}+\frac{1}{2}\boldsymbol{\delta}^T\nabla^2 f\left(\boldsymbol{x}_k\right)\boldsymbol{\delta}\ -\ (1.12)$$

By setting the gradient of the quadratic function on the right-hand side of (1.12) to zero, we are led to the system of linear equations

$$\nabla^2 f\left(x_k\right) \delta = -\nabla f\left(x_k\right)$$

$$d_k = [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

whose solution, denoted by $\boldsymbol{d}_k$, is the stationary point of that quadratic function. We call $\boldsymbol{d}_k$ the Newton direction at point $\boldsymbol{x}_k$.

1.0.4 BFGS (Broyden–Fletcher–Goldfarb–Shanno)

One may take a unifying look at the basic GD and Newton algorithms by expressing their search directions as

$$d_k = -Sk\nabla f(x_k) \, - \, (1.13)$$

where

$$S_k = I \; for \; G.D \, - \, (1.14)$$

$$S_k = [\nabla^2 f(x_k)]^{-1} \nabla f(x_k) \; for \; Newton \, - \, (1.15)$$

Quasi-Newton algorithms are developed to provide convergent rates comparable with that of Newton algorithm with reduced complexity, especially for problems of median and large scales. The distinction between Newton and quasi-Newton algorithms is that essentially Newton algorithm requires evaluating inverse of the Hessian while quasi-Newton algorithms are based only on gradient information and do not require explicit evaluation of the Hessian and its inverse. Quasi-Newton algorithms follow the general algorithmic structure described earlier, where search direction dk is evaluated using (1.13) where matrix Sk is updated in each iteration using gradient information. The most well-known quasi-Newton algorithm is the one known as BFGS algorithm, developed by Broyden, Fletcher, Goldfarb, and Shanno in late 1960s and early 1970s.Matrix Sk is usually initiated as $S_0 = I$. In its k^{th} iteration, the BFGS algorithm updates matrix S_k in a couple of simple steps:

- Compute $\delta_k = x_{k+1} - x_k, \gamma_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.
- Update S_k to

$$\boldsymbol{S}_{k+1} = \boldsymbol{S}_k + \left(1 + \frac{\boldsymbol{\gamma}_k^T \boldsymbol{S}_k \boldsymbol{\gamma}_k}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}\right) \frac{\boldsymbol{\delta}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} - \frac{\boldsymbol{\delta}_k \boldsymbol{\gamma}_k^T \boldsymbol{S}_k + \boldsymbol{S}_k \boldsymbol{\gamma}_k \boldsymbol{\delta}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k} \, - \, (1.16)$$

1.0.5 Memoryless BFGS

For large-scale problems, producing, storing, and manipulation of matrix Sk (see (1.16)), which is usually not sparse, can be problematic. The memoryless BFGS algorithm deals with this difficulty by replacing (1.16) with a simplified updating formula that reduces the computation of the search direction $-S_{k+1}\nabla f(x_{k+1})$ to a few inner products.

- **Step 1**: Compute

$$\rho_k = \frac{1}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}, t_k = \boldsymbol{\delta}_k^T \nabla f\left(\boldsymbol{x}_{k+1}\right), \text{ and } \boldsymbol{q}_k = \nabla f\left(\boldsymbol{x}_{k+1}\right) - \rho_k t_k \boldsymbol{\gamma}_k$$

- **Step 2**: Compute search direction

$$\boldsymbol{d}_{k+1} = \rho_k \left(\boldsymbol{\gamma}_k^T \boldsymbol{q}_k - t_k\right) \boldsymbol{\delta}_k - \boldsymbol{q}_k$$

2 Formulating the optimization problem

In this project, the logistic regression classifier described in 1.1 is used for the prediction of the class for a certain sample x_p. However, the cost function (see 1.17) for the same is described in terms of log(error) because equation 1.4 is non-convex which could lead to the optimization algorithm converging to a local minimum. The following cost function can be minimized using the C.G, Newton, BFGS, and ML-BFGS algorithms.

$$e(\hat{\boldsymbol{w}}) = \frac{1}{P}\sum_{p=1}^{P} log(1 + e^{-y_p \hat{\boldsymbol{w}}^T X_p}) \; - \; (1.17)$$

The data set contains latent vectors that represent images, and our objective is to minimize the cost function such that a hyper-plane is fit through the datapoints. In fact, in this report, we will be using the aforementioned algorithms to minimize the above cost function and analyze, compare their results.

3 Solution Method

The logistic regression cost function and its gradient, hessian need to be abstracted into their own functions in MATLAB. To accomplish this, I have leveraged the code provided on the course website and built on top of it. As part of the solution, we minimize the above cost function over the 3500 training samples using the mentioned algorithms. Once the cost function is successfully minimized, we get the optimal weights vector $\hat{w}^*$ which can be used to classify the 1500 test samples in the following way

$$\begin{array}{ll} \boldsymbol{x} \in \boldsymbol{P} & \text{if } \boldsymbol{w}^{*T}\boldsymbol{x} + b > 0 \\ \boldsymbol{x} \in \mathcal{N} & \text{if } \boldsymbol{w}^{*T}\boldsymbol{x} + b < 0 \end{array}$$

Subsequently, we can use the true labels from the 1500 test labels to calculate the accuracy and training time for various algorithms.

3.1 Matlab Code for $E_L(\hat{\boldsymbol{w}})$

In this section, we see the corresponding matlab code for the cost function mentioned in the equation 1.17.

$$e(\hat{\boldsymbol{w}}) = \frac{1}{P}\sum_{p=1}^{P} log(1 + e^{-y_p \hat{\boldsymbol{w}}^T X_p})$$

```
% f_LRBC.m
function f = f_LRBC(w,X)
P = size(X,2);
f = sum(log(1+exp(-X'*w)))/P;
```

3.2 Matlab Code for $\nabla E_L(\hat{\boldsymbol{w}})$

In this section, we see the corresponding matlab code for the gradient of the cost function mentioned in the equation 1.17.

$$\nabla E_L(\hat{\boldsymbol{w}}) = \frac{1}{P}\sum_{p=1}^{P} \nabla \log\left(1 + e^{-y_p \hat{\boldsymbol{x}}_{\hat{w}}^T \hat{\boldsymbol{\omega}}}\right) = -\frac{1}{P}\sum_{p=1}^{P} \frac{y_p \hat{\boldsymbol{x}}_p}{1 + e^{y_p \hat{\boldsymbol{x}}_p^T \hat{\boldsymbol{w}}}}$$

```
% g_LRBC.m
function g = g_LRBC(w,X)
P = size(X,2);
q1 = exp(X'*w);
q = 1./(1+q1);
g = -(X*q)/P;
```

3.3 Matlab Code for $\nabla^2 E_L(\hat{\boldsymbol{w}})$

In this section, we see the corresponding matlab code for the hessian of the cost function mentioned in the equation 1.17.

$$\nabla^2 E_L(\hat{\boldsymbol{w}}) = \frac{1}{P}\sum_{p=1}^{P} \nabla^2 \log\left(1 + e^{-y_p \hat{x}_p^T \hat{w}}\right) = \frac{1}{P}\sum_{p=1}^{P} \frac{e^{y_p \hat{x}_p^T \hat{w}}}{\left(1 + e^{y_p \hat{x}_p^T \hat{w}}\right)^2} \hat{\boldsymbol{x}}_p \hat{\boldsymbol{x}}_p^T$$

```
% h_LRBC.m
function H = h_LRBC(w,X)
[N1,P] = size(X);
q1 = exp(X'*w);
q = q1./((1+q1).^2);
H = zeros(N1,N1);
for p = 1:P
    xp = X(:,p);
    H = H + q(p)*(xp*xp');
end
H = H/P;
```

3.4 Using Conjugate Gradient Descent

As seen in equation 1.8, the direction of the conjugate gradient descent method depends on the gradient of the cost function, B_k and d_{k-1}. Furthermore, the algorithm for the same is described below

- **Step 1** Input an initial point x_0, and a convergence tolerance ε. Set $k = 0$. Compute $\nabla f(x_0)$ and set $d_0 = -\nabla f(x_0)$.
- **Step 2** Find α_k, the value of α that minimizes $f(x_k + \alpha d_k)$ using BTLS. Update $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$.
- **Step 3** Compute $\nabla f(x_{k+1})$. If $\|\nabla f(x_{k+1})\|_2 < \varepsilon$, output x_{k+1} as the solution and stop; otherwise, set $k := k+1$ and go to Step 4.
- **Step 4** Compute search direction

  $$\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k) + \beta_k \boldsymbol{d}_{k-1}$$

  where β_k is calculated using (2.2c), and repeat from Step 2.

The above algorithm has been translated into code and abstracted into the following function.

Matlab Code

```
% To implement CG algorithm with Polak-Ribiere-Polyak-(plus)'s beta.
% Example:
% [xs,fs,k] = cg('f_rosen','g_rosen',[-1;-1],1e-6);
function [Ws,fs,k] = cg(fname,gname,wk,X,y,epsi)
format compact
format long
[N,P] = size(X);
ind = 1:1:P;
indp = find(y > 0);
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];
```

```
n = length(wk);
k = 0;
gk = feval(gname,wk, Xw);
dk = -gk;
er = norm(gk);
while er >= epsi
    ak = bt_lsearch2019(wk,dk,fname,gname, Xw);
    wk_new = wk + ak*dk;
    gk_new = feval(gname,wk_new, Xw);
    gmk = gk_new - gk;
    bk = max((gk_new'*gmk)/(gk'*gk),0);
    dk = -gk_new + bk*dk;
    wk = wk_new;
    gk = gk_new;
    er = norm(gk);
    k = k + 1;
end
% disp('solution:')
Ws = wk;
disp('objective function at solution point:')
fs = feval(fname,Ws, Xw);
format short
disp('number of iterations at convergence:')
k
```

3.5 Using Newton's Method

Newton's algorithm for minimization of the cost function is shown below:

- **Step 1** Input x_0 and initialize the tolerance ε. Set $k = 0$.
- **Step 2** Compute $\nabla f\left(x_k\right)$ and $\nabla^2 f\left(x_k\right)$. If $\nabla^2 f\left(x_k\right)$ is not positive definite, modify it to become positive definite (see below).
- **Step 3** Compute $\boldsymbol{d}_k$ by solving the linear system of equations.
- **Step 4** Using BTLS to find α_k, the value of α that minimizes $f\left(x_k + \alpha d_k\right)$.
- **Step 5** Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$.
- **Step 6** If $\|\alpha_k \boldsymbol{d}_k\| < \varepsilon$, then output $\boldsymbol{x}^* = \boldsymbol{x}_{k+1}$ and stop; otherwise, set $k = k + 1$ and repeat from Step 2.

Modification of the Hessian If the Hessian is not positive definite in an iteration of Newton algorithm, it is forced to become positive definite in Step 2. One approach to modifying $\nabla^2 f\left(x_k\right)$ is to replace it by

$$\hat{\boldsymbol{H}}_k = \frac{\nabla^2 f\left(\boldsymbol{x}_k\right) + \beta \boldsymbol{I}_N}{1+\beta}$$

The above algorithm has been translated into code, and abstracted into a function called $LRBC_Newton.m$

LRBC_Newton.m

```
function [ws,C2] = LRBC_newton(X,y,K)
y = y(:)';
[N,P] = size(X);
N1 = N + 1;
Ine = 1e-10*eye(N1);
ind = 1:1:P;
indp = find(y > 0);
```

```
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];
P1 = length(indp);
k = 0;
wk = zeros(N1,1);
while k < K
  gk = feval('g_LRBC',wk,Xw);
  Hk = feval('h_LRBC',wk,Xw) + Ine;
  dk = -Hk\gk;
  ak = bt_lsearch2019(wk,dk,'f_LRBC','g_LRBC',Xw);
  wk = wk + ak*dk;
  k = k + 1;
  fk = feval('f_LRBC',wk, Xw);
  fprintf('iter %1i: loss = %8.2e\n',k,fk)
end
ws = wk;
yt = sign(ws'*[Xp Xn]);
er = abs(y-yt)/2;
erp = sum(er(1:P1));
ern = sum(er(P1+1:P));
C2 = [P1-erp ern; erp P-P1-ern];
```

3.6 Using BFGS Method

In the Newton's method it is important to note that the storage and calculation of Hessian and its inverse is a computationally expensive task. From a quick research, I have found out that the time complexity of Matrix Inversion is $O(n^2)$ at the least. This implies that with the increase in the number of features and samples, the complexity for the calculation of inverse increases largely. This drawback, along with the need to check positive definiteness of hessian at every step, led to the development of quasi-newton algorithms. The BFGS direction described in equation 1.13 contains S_k which is trying to approximate the Hessian and in no more than n iterations it converges to H^{-1}.

- **Step 1** Input $\boldsymbol{x}_0$ and initialize the tolerance ε. Set $k = 0$ and $\boldsymbol{S}_0 = \boldsymbol{I}$. Compute $\nabla f(\boldsymbol{x}_0)$.
- **Step 2** Set $\boldsymbol{d}_k = -\boldsymbol{S}_k \nabla f(\boldsymbol{x}_k)$. Find α_k, the value of α that minimizes $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ using BTLS.
- **Step 3** Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$ and compute $\boldsymbol{\delta}_k$.
- **Step 4** If $\|\boldsymbol{\delta}_k\|_2 < \varepsilon$, output $\boldsymbol{x}^* = \boldsymbol{x}_{k+1}$ and stop; otherwise proceed.
- **Step 5** Compute $\nabla f(x_{k+1})$ and γ_k (see section 1.0.4). Compute S_{k+1}. Set $k = k+1$ and repeat from Step 2 .

The above algorithm has been translated into code and abstracted into the following function

Matlab Code

```
function [Ws,fs,k] = bfgs(fname,gname,wk, X,y, iter)
format compact
format long

[N,P] = size(X);
ind = 1:1:P;
indp = find(y > 0);
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
```

```
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];

n = length(wk);
I = eye(n);
k = 1;
Sk = I;
fk = feval(fname,wk,Xw);
gk = feval(gname,wk,Xw);
dk = -Sk*gk;
ak = bt_lsearch2019(wk,dk,fname,gname,Xw);
dtk = ak*dk;
wk_new = wk + dtk;
fk_new = feval(fname,wk_new,Xw);

while k < iter
      gk_new = feval(gname,wk_new,Xw);
      gmk = gk_new - gk;
      D = dtk'*gmk;
      if D <= 0
         Sk = I;
      else
         sg = Sk*gmk;
         sw0 = (1+(gmk'*sg)/D)/D;
         sw1 = dtk*dtk';
         sw2 = sg*dtk';
         Sk = Sk + sw0*sw1 - (sw2'+sw2)/D;
      end
      fk = fk_new;
      gk = gk_new;
      wk = wk_new;
      dk = -Sk*gk;
      ak = bt_lsearch2019(wk,dk,fname,gname,Xw);
      dtk = ak*dk;
      wk_new = wk + dtk;
      fk_new = feval(fname,wk_new,Xw);
      k = k + 1;
      fprintf('iter %1i: loss = %8.2e\n',k,fk_new)
end
disp('solution:')
Ws = wk_new
disp('objective function at solution point:')
fs = feval(fname,Ws,Xw)
format short
disp('number of iterations at convergence:')
k
```

3.7 Memoryless BFGS

For large-scale problems, producing, storing, and manipulation of matrix Sk, which is usually not sparse, can be problematic. The memoryless BFGS algorithm deals with this difficulty by replacing (1.16) with a simplified updating formula that reduces the computation of the search direction to a few inner products. The algorithm for the same is described below

- **Step 1**: Compute

$$\rho_k = \frac{1}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}, t_k = \boldsymbol{\delta}_k^T \nabla f\left(\boldsymbol{x}_{k+1}\right), \text{ and } \boldsymbol{q}_k = \nabla f\left(\boldsymbol{x}_{k+1}\right) - \rho_k t_k \boldsymbol{\gamma}_k$$

- **Step 2**: Compute search direction

$$\boldsymbol{d}_{k+1} = \rho_k \left(\boldsymbol{\gamma}_k^T \boldsymbol{q}_k - t_k\right) \boldsymbol{\delta}_k - \boldsymbol{q}_k$$

The above algorithm has been translated into code and abstracted into the following function

Matlab Code

```
function [Ws,f] = bfgsML(Dtr,fname,gname,mu,K,iter)
N1 = size(Dtr,1);
muK = [mu K];
W0 = zeros(N1,K);
k = 1;
xk = W0(:);
disp(muK)
fk = feval(fname,xk,Dtr,muK);
f = fk;
fprintf('iter %1i: loss = %8.2e\n',k-1,fk)
gk = feval(gname,xk,Dtr,muK);
dk = -gk;
ak = bt_lsearch2019(xk,dk,fname,gname,Dtr,muK);
dtk = -ak*gk;
xk_new = xk + dtk;
fk = feval(fname,xk_new,Dtr,muK);
f = [f; fk];
fprintf('iter %1i: loss = %8.2e\n',k,fk)
while k < iter
  gk_new = feval(gname,xk_new,Dtr,muK);
  gmk = gk_new - gk;
  gk = gk_new;
  rk = 1/(dtk'*gmk);
  if rk <= 0
     dk = -gk;
  else
     tk = dtk'*gk;
     qk = gk - (rk*tk)*gmk;
     bk = rk*(gmk'*qk - tk);
     dk = bk*dtk - qk;
  end
  xk = xk_new;
  ak = bt_lsearch2019(xk,dk,fname,gname,Dtr,muK);
  dtk = ak*dk;
  xk_new = xk + dtk;
  fk = feval(fname,xk_new,Dtr,muK);
  f = [f; fk];
  k = k + 1;
  fprintf('iter %1i: loss = %8.2e\n',k,fk)
end
Ws = reshape(xk_new,N1,K);
fprintf('final loss = %8.2e\n',fk)
```

4 Results

I was able to successfully apply the various algorithms to minimize the cost function and obtained the following results

4.1 Results from C.G

```
epsilon = 0.001
iter 1: loss = 3.29e-01
iter 2: loss = 2.79e-01
iter 3: loss = 2.45e-01
iter 4: loss = 2.19e-01
iter 5: loss = 1.99e-01
....
iter 1922: loss = 2.06e-03
iter 1923: loss = 2.06e-03
number of iterations at convergence: 1923

Training time for logistic regression cost function with CG (1923 iterations): 35.255791 seconds
   876    21
     0   453

Accuracy: 98.444444
```

Iterations	1923
Accuracy	98.444444%
Training Time (in sec)	35.255791

4.2 Results from Newton

```
iter 1: loss = 1.66e-01
iter 2: loss = 6.63e-02
iter 3: loss = 2.71e-02
iter 4: loss = 1.09e-02
iter 5: loss = 4.29e-03


Training time for logistic regression cost function with Newton (5 iterations): 7.438864 seconds
   876    15
     0   459


Accuracy: 98.888889
```

Iterations	5
Accuracy	98.888889%
Training Time (in sec)	7.438864

4.3 Results from BFGS

```
iter 2: loss = 2.65e-01
iter 3: loss = 1.47e-01
iter 4: loss = 9.94e-02
iter 5: loss = 6.67e-02
iter 6: loss = 4.69e-02
iter 7: loss = 3.35e-02
iter 8: loss = 2.43e-02
iter 9: loss = 1.78e-02
```

```
....
iter 126: loss = 7.05e-20
iter 127: loss = 7.05e-20
iter 128: loss = 7.05e-20
iter 129: loss = 7.05e-20
iter 130: loss = 7.05e-20

objective function at solution point:
fs =
     7.049035076985120e-20
number of iterations at convergence:
k =
   130


Training time for logistic regression cost function with BFGS (130 iterations): 15.280920 seconds
   870    26
     6   448


Accuracy: 97.629630
```

Iterations	130
Accuracy	97.629630%
Training Time (in sec)	15.280920

4.4 Results from ML-BFGS

```
iter 0: loss = 6.93e-01
iter 1: loss = 2.95e-01
iter 2: loss = 2.15e-01
iter 3: loss = 1.10e-01
iter 4: loss = 8.49e-02
iter 5: loss = 6.04e-02
iter 6: loss = 4.71e-02
iter 7: loss = 4.01e-02
iter 8: loss = 3.73e-02
iter 9: loss = 3.63e-02
iter 10: loss = 3.59e-02
...
iter 127: loss = 3.06e-02
iter 128: loss = 3.06e-02
iter 129: loss = 3.05e-02
iter 130: loss = 3.05e-02
final loss = 3.05e-02

Training time for softmax regression cost function with ML-BFGS (130 iterations): 9.390939 seconds
   452     0
    22   876


Accuracy: 98.370370
```

Iterations	130
Accuracy	98.370370%
Training Time (in sec)	9.390939

4.5 Comparison of Results

In this section we compare the results obtained from using C.G, Newton, BFGS and ML-BFGS algorithms towards minimizing the logistic regression cost function.

Algorithm	**Iterations**	**Accuracy**	**Training Time (in sec)**
Conjugate Gradient	1923	98.444444%	35.255791
Newton's Method	5	98.888889%	7.438864
BFGS	130	97.629630%	15.280920
ML-BFGS	130	98.370370%	9.390939

4.5.1 Newton's Method

Newton's algorithm performs the best among all the other algorithms in terms of both accuracy and training time. Although it performed well in this case, it is important to note that this method inherently has several disadvantages, shown below

- Positive definiteness of the Hessian must be checked at every step.
- If 'H' is not positive definite, then it must be modified to be P.D.
- Calculation of Hessian Inverse has a time complexity of $O(n^2)$, which implies that with an increase in the number of features and samples the training time will get slower.
- Newton's method is also shown to get attracted to saddle points.

4.5.2 Conjugate Gradient Descent

Conjugate Gradient performed the next best in terms of accuracy, but had the worst training time. The results in the course notes concurs with my result. C.G took about 1923 iterations to reach the minimizer and a total of 35.255791 seconds. CG has guaranteed convergence, with the absolute worst case number of iterations being at most the number of rows or columns of the matrix.

4.5.3 BFGS

BFGS method took about 15.280920 seconds and 130 iterations to converge to the minimizer and attained an accuracy of over 97.629620% on the test samples. It is clear that calculation and storing of S_k, which in this case is a (513x513) matrix has caused each iteration to be slower when compared to ML_BFGS.

4.5.4 ML-BFGS

ML-BFGS performed the best after newton in terms of both accuracy and training time. It took about 130 iterations and a training time of 9.390939 seconds to converge to the minimizer and attained an decent accuracy of 98.370370% on the test samples. Not having to store the previous value of S_k has clearly led to each iteration being faster thus leading to a faster convergence.

References

[1] Optimization for Machine Learning - ECE403/503 Course Notes by Professor. Wu-Sheng Lu.

[2] Optimization for Machine Learning - ECE403/503 Lecture Notes by Lei Zhao.

[3] Optimization for Machine Learning - Laboratory Manual by Professor. Wu-Sheng Lu.

[4] Dataset: T81-855: Applications of Deep Learning at Washington University in St. Louis.

[5] Dauphin, Yann & Pascanu, Razvan & Gulcehre, Caglar & Cho, Kyunghyun & Ganguli, Surya & Bengio, Y.. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. 2933-2941.

[6] Pascanu, Razvan & Dauphin, Yann & Ganguli, Surya & Bengio, Y.. (2014). On the saddle point problem for non-convex optimization.

[7] An Introduction to MATLAB by David F. Griffiths.

[8] Scikit learn documentation for logistic regression - sklearn.linear_model.LogisticRegression

Acknowledgements

I like to acknowledge the constant support provided by instructor Lei Zhao in the form of delivering quality lectures, hosting several office hour sessions, and responding to countless emails of mine, without which this couldn't have been possible.

Appendices

MATLAB Code

```
clear;
clc;
K = 2;
global train
train = load('train.mat').M1';

% Extract training data
train_samples = train(1:512, 1:3150);
train_labels = train(513, 1:3150);

% Extract testing data
test_samples = train(1:512, 3151:4500);
test_labels = train(513, 3151:4500);

Dte = [test_samples; ones(1, 1350)];

%%%%%%%%%%%%%%%% CG Method %%%%%%%%%%%%%%%
useCG(train_samples, train_labels, test_labels, Dte);

%%%%%%%%%%%%%%%% BFGS Method %%%%%%%%%%%%%%%
useBFGS(train_samples, train_labels, test_labels, Dte);

%%%%%%%%%%%%%%%% MLBFGS Method %%%%%%%%%%%%%%%%
useMLBFGS(Dte);

%%%%%%%%%%%%%%%% Newton Method %%%%%%%%%%%%%%%%
useNewton(train_samples, train_labels, test_labels, Dte)

function useCG(train_samples, train_labels, test_labels, Dte)
    train_labels(train_labels == 0)= -1;
    test_labels(test_labels == 0)= -1;

    W0 = zeros(1,513)';
    tic;
    [Ws, f, k] = cg('f_LRBC', 'g_LRBC', W0, train_samples, train_labels, 0.001);
    fprintf("\n\nTraining time for logistic regression cost function
    with CG (%d iterations): %f seconds\n", k, toc);
    displayResultGeneric(Dte, Ws, test_labels);
end

function useBFGS(train_samples, train_labels, test_labels, Dte)
    train_labels(train_labels == 0)= -1;
    test_labels(test_labels == 0)= -1;

    W0 = zeros(1,513)';
    tic;
    [Ws, f, k] = bfgs('f_LRBC', 'g_LRBC', W0, train_samples, train_labels, 130);
    fprintf("\n\nTraining time for logistic regression cost function
    with BFGS (%d iterations): %f seconds\n", k, toc);

    normw = norm(Ws(1:512));
    for k = 1:513
        Ws(k) = Ws(k)/normw;
    end

    displayResultGeneric(Dte, Ws, test_labels);
end

function useMLBFGS(Dte)
    global train
    train_samples = train(1:512, 1:3150);
    train_labels = 1 + train(513, 1:3150);
    Dtr = [train_samples; train_labels];
    test_labels = 1  + train(513, 3151:4500);

    tic;
    [Ws, f]= SRMCC_bfgsML(Dtr, 'f_SRMCC', 'g_SRMCC', 0, 2, 130);

    normw = norm(Ws(1:512));
    for k = 1:513
        Ws(k) = Ws(k)/normw;
    end

    fprintf("\n\nTraining time for softmax regression cost function with
    ML-BFGS (%d iterations): %f seconds\n",130, toc);

    displayResult(Dte, Ws, 2, test_labels);
end

function useNewton(train_samples, train_labels, test_labels, Dte)
    train_labels(train_labels == 0)= -1;
    test_labels(test_labels == 0)= -1;

    tic;
    [Ws, f] = LRBC_newton(train_samples, train_labels, 5);
    fprintf("\n\nTraining time for logistic regression cost function
    with Newton (%d iterations): %f seconds\n", 5, toc);

    displayResultGeneric(Dte, Ws, test_labels);
end

function displayResult(Data_Matrix, Ws, K, ytest)
    [~, ind_pre] = max((Data_Matrix' * Ws)');

    C = zeros(K,K);
    for j = 1:K
        ind_j = find(ytest == j);
        for i = 1:K
            ind_pre_i = find(ind_pre == i);
            C(i,j) = length(intersect(ind_j,ind_pre_i));
        end
    end

    disp(C)
    accuracy = sum(diag(C))/(sum(C, 'all')) * 100;
    fprintf("\n\nAccuracy: %f\n", accuracy);
end

function displayResultGeneric(Data_Matrix, Ws, ytest)
```

```
    C = zeros(2,2);
    values = sign(Ws' * Data_Matrix);

    for value = 1 : length(values)
        if(ytest(value) == 1 && values(value) == 1)
            C(1, 1) = C(1, 1) + 1;
        elseif(ytest(value)== 1 && values(value) == -1)
            C(2, 1) = C(2, 1) + 1;
        elseif(ytest(value) == -1 && values(value) == 1)
            C(1, 2) = C(1, 2) + 1;
        else
            C(2, 2) = C(2, 2) + 1;
        end
    end

    disp(C)
    accuracy = sum(diag(C))/(sum(C, 'all')) * 100;
    fprintf("\n\nAccuracy: %f\n", accuracy);
end

%%%%%%%%%%%%%% f_LRBC.m %%%%%%%%%%%%%%
function f = f_SRMCC(x,D,muK)
mu = muK(1);
K = muK(2);
[N1,P] = size(D);
Xh = [D(1:N1-1,:); ones(1,P)];
y = D(N1,:);
W = reshape(x,N1,K);
f = 0;
for p = 1:P
    xp = Xh(:,p);
    t0 = sum(exp(xp'*W));
    tp = exp(xp'*W(:,y(p)));
    f = f + log(tp/t0);
end
xw1 = W(:);
f = -f/P + 0.5*mu*(xw1'*xw1);


%%%%%%%%%%%%%% g_LRBC.m %%%%%%%%%%%%%%
function g = g_LRBC(w,X)
P = size(X,2);
q1 = exp(X'*w);
q = 1./(1+q1);
g = -(X*q)/P;

%%%%%%%%%%%%%% h_LRBC.m %%%%%%%%%%%%%%
function H = h_LRBC(w,X)
[N1,P] = size(X);
q1 = exp(X'*w);
q = q1./((1+q1).^2);
H = zeros(N1,N1);
for p = 1:P
    xp = X(:,p);
    H = H + q(p)*(xp*xp');
end
```

```
H = H/P;

%%%%%%%%%%%%% cg.m %%%%%%%%%%%%%
% To implement CG algorithm with Polak-Ribiere-Polyak-(plus)'s beta.
% Example:
% [xs,fs,k] = cg('f_rosen','g_rosen',[-1;-1],1e-6);
function [Ws,fs,k] = cg(fname,gname,wk,X,y,epsi)
format compact
format long
[N,P] = size(X);
ind = 1:1:P;
indp = find(y > 0);
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];

n = length(wk);
k = 0;
gk = feval(gname,wk, Xw);
dk = -gk;
er = norm(gk);

while er >= epsi
    ak = bt_lsearch2019(wk,dk,fname,gname, Xw);
    wk_new = wk + ak*dk;
    gk_new = feval(gname,wk_new, Xw);
    gmk = gk_new - gk;
    bk = max((gk_new'*gmk)/(gk'*gk),0);
    dk = -gk_new + bk*dk;
    wk = wk_new;
    gk = gk_new;
    er = norm(gk);
    k = k + 1;
    fk_new = feval(fname,wk,Xw);
    fprintf('iter %1i: loss = %8.2e\n',k, fk_new)
end
% disp('solution:')
Ws = wk;
disp('objective function at solution point:')
fs = feval(fname,Ws, Xw);
format short
disp('number of iterations at convergence:')
k

%%%%%%%%%%%%% LRBC_newton.m %%%%%%%%%%%%%
function [ws,C2] = LRBC_newton(X,y,K)
y = y(:)';
[N,P] = size(X);
N1 = N + 1;
Ine = 1e-10*eye(N1);
ind = 1:1:P;
indp = find(y > 0);
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
```

```
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];
P1 = length(indp);
k = 0;
wk = zeros(N1,1);
while k < K
  gk = feval('g_LRBC',wk,Xw);
  Hk = feval('h_LRBC',wk,Xw) + Ine;
  dk = -Hk\gk;
  ak = bt_lsearch2019(wk,dk,'f_LRBC','g_LRBC',Xw);
  wk = wk + ak*dk;
  k = k + 1;
  fk = feval('f_LRBC',wk, Xw);
  fprintf('iter %1i: loss = %8.2e\n',k,fk)
end
ws = wk;
yt = sign(ws'*[Xp Xn]);
er = abs(y-yt)/2;
erp = sum(er(1:P1));
ern = sum(er(P1+1:P));
C2 = [P1-erp ern; erp P-P1-ern];

%%%%%%%%%%%%%%%%%% bfgs.m %%%%%%%%%%%%%%%%%%%
function [Ws,fs,k] = bfgs(fname,gname,wk, X,y, iter)
format compact
format long
[N,P] = size(X);
ind = 1:1:P;
indp = find(y > 0);
indn = setdiff(ind,indp);
Xh = [X; ones(1,P)];
Xp = Xh(:,indp);
Xn = Xh(:,indn);
Xw = [Xp -Xn];
n = length(wk);
I = eye(n);
k = 1;
Sk = I;
fk = feval(fname,wk,Xw);
gk = feval(gname,wk,Xw);
dk = -Sk*gk;
ak = bt_lsearch2019(wk,dk,fname,gname,Xw);
dtk = ak*dk;
wk_new = wk + dtk;
fk_new = feval(fname,wk_new,Xw);

while k < iter
      gk_new = feval(gname,wk_new,Xw);
      gmk = gk_new - gk;
      D = dtk'*gmk;
      if D <= 0
         Sk = I;
      else
         sg = Sk*gmk;
         sw0 = (1+(gmk'*sg)/D)/D;
         sw1 = dtk*dtk';
         sw2 = sg*dtk';
         Sk = Sk + sw0*sw1 - (sw2'+sw2)/D;
      end
      fk = fk_new;
      gk = gk_new;
      wk = wk_new;
      dk = -Sk*gk;
      ak = bt_lsearch2019(wk,dk,fname,gname,Xw);
      dtk = ak*dk;
      wk_new = wk + dtk;
      fk_new = feval(fname,wk_new,Xw);
      k = k + 1;
      fprintf('iter %1i: loss = %8.2e\n',k,fk_new)
end
disp('solution:')
Ws = wk_new
disp('objective function at solution point:')
fs = feval(fname,Ws,Xw)
format short
disp('number of iterations at convergence:')
k

%%%%%%%%%%%%% SRMCC_bfgsML.m %%%%%%%%%%%%%
function [Ws,f] = SRMCC_bfgsML(Dtr,fname,gname,mu,K,iter)
N1 = size(Dtr,1);
muK = [mu K];
W0 = zeros(N1,K);
k = 1;
xk = W0(:);
disp(muK)
fk = feval(fname,xk,Dtr,muK);
f = fk;
fprintf('iter %1i: loss = %8.2e\n',k-1,fk)
gk = feval(gname,xk,Dtr,muK);
dk = -gk;
ak = bt_lsearch2019(xk,dk,fname,gname,Dtr,muK);
dtk = -ak*gk;
xk_new = xk + dtk;
fk = feval(fname,xk_new,Dtr,muK);
f = [f; fk];
fprintf('iter %1i: loss = %8.2e\n',k,fk)
while k < iter
  gk_new = feval(gname,xk_new,Dtr,muK);
  gmk = gk_new - gk;
  gk = gk_new;
  rk = 1/(dtk'*gmk);
  if rk <= 0
     dk = -gk;
  else
     tk = dtk'*gk;
     qk = gk - (rk*tk)*gmk;
     bk = rk*(gmk'*qk - tk);
     dk = bk*dtk - qk;
  end
  xk = xk_new;
  ak = bt_lsearch2019(xk,dk,fname,gname,Dtr,muK);
  dtk = ak*dk;
  xk_new = xk + dtk;
  fk = feval(fname,xk_new,Dtr,muK);
  f = [f; fk];
  k = k + 1;
  fprintf('iter %1i: loss = %8.2e\n',k,fk)
end
Ws = reshape(xk_new,N1,K);
fprintf('final loss = %8.2e\n',fk)

%%%%%%%%%%%% bt_lsearch2019.m %%%%%%%%%%%%
function a = bt_lsearch2019(x,d,fname,gname,p1,p2)
rho = 0.1;
gma = 0.5;
x = x(:);
d = d(:);
a = 1;
xw = x + a*d;
parameterstring ='';
if nargin == 5
   if ischar(p1)
      eval([p1 ';']);
   else
      parameterstring = ',p1';
   end
end
if nargin == 6
   if ischar(p1)
      eval([p1 ';']);
   else
      parameterstring = ',p1';
   end
   if ischar(p2)
      eval([p2 ';']);
   else
      parameterstring = ',p1,p2';
   end
end
eval(['f0 = ' fname '(x' parameterstring ');']);
eval(['g0 = ' gname '(x' parameterstring ');']);
eval(['f1 = ' fname '(xw' parameterstring ');']);
t0 = rho*(g0'*d);
f2 = f0 + a*t0;
er = f1 - f2;
while er > 0
     a = gma*a;
     xw = x + a*d;
     eval(['f1 = ' fname '(xw' parameterstring ');']);
     f2 = f0 + a*t0;
     er = f1 - f2;
end
if a < 1e-5
   a = min([1e-5, 0.1/norm(d)]);
end
```