Modern Optimization Techniques

Lucas Rego Drumond

Information Systems and Machine Learning Lab (ISMLL)
Institute of Computer Science
University of Hildesheim, Germany

Newton's Method part II

Outline

Outline

Example II - The Logistic Regression

The logistic regression learning problem is

$$\text{minimize} \quad -\sum_{i=1}^{m} y_i \log \sigma(\mathbf{x}^T \mathbf{a_i}) + (1 - y_i) \log(1 - \sigma(\mathbf{x}^T \mathbf{a_i}))$$

$$A_{m,n} = \begin{pmatrix} 1 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 1 & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a_{m,1} & a_{m,2} & a_{m,3} & a_{m,4} \end{pmatrix} \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The Logistic Regression

First we need to compute the gradient of our objective function:

$$\text{minimize} \quad -\sum_{i=1}^{m} y_i \log \sigma(\mathbf{x}^T \mathbf{a_i}) + (1 - y_i) \log(1 - \sigma(\mathbf{x}^T \mathbf{a_i}))$$

$$\begin{aligned}
\frac{\partial f_0}{\partial \mathbf{x}_k} &= -\sum_{i=1}^{m} y_i \frac{1}{\sigma(\mathbf{x}^T \mathbf{a_i})} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(1 - \sigma(\mathbf{x}^T \mathbf{a_i})\right) a_{ik} \\
&\quad -(1 - y_i) \frac{1}{1 - \sigma(\mathbf{x}^T \mathbf{a_i})} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(1 - \sigma(\mathbf{x}^T \mathbf{a_i})\right) a_{ik} \\
&= -\sum_{i=1}^{m} y_i a_{ik} \left(1 - \sigma(\mathbf{x}^T \mathbf{a_i})\right) - (1 - y_i) a_{ik} \sigma(\mathbf{x}^T \mathbf{a_i}) \\
&= -\sum_{i=1}^{m} a_{ik} \left(y_i - \sigma(\mathbf{x}^T \mathbf{a_i})\right)
\end{aligned}$$

The Logistic Regression

$$\frac{\partial f_0}{\partial \mathbf{x}_k} = -\sum_{i=1}^{m} a_{ik} \left(y_i - \sigma(\mathbf{x}^T \mathbf{a_i}) \right)$$

Now we need to compute the Hessian matrix:

$$\frac{\partial^2 f_0}{\partial \mathbf{x}_k \partial \mathbf{x}_j} = -\sum_{i=1}^{m} -a_{ik} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(1 - \sigma(\mathbf{x}^T \mathbf{a_i}) \right) a_{ij}$$

$$= -\sum_{i=1}^{m} a_{ik} a_{ij} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(\sigma(\mathbf{x}^T \mathbf{a_i}) - 1 \right)$$

The Hessian H is an $n \times n$ matrix such that:

$$H_{k,j} = -\sum_{i=1}^{m} a_{ik} a_{ij} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(\sigma(\mathbf{x}^T \mathbf{a_i}) - 1 \right)$$

The Logistic Regression

So we have our gradient $\nabla f_0 \in \mathbb{R}^n$ such that

$$\nabla_{\mathbf{x}_k} f_0 = -\sum_{i=1}^{m} a_{ik} \left(y_i - \sigma(\mathbf{x}^T \mathbf{a_i}) \right)$$

And the Hessian $H \in \mathbb{R}^{n \times n}$:

$$H_{k,j} = -\sum_{i=1}^{m} a_{ik} a_{ij} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(\sigma(\mathbf{x}^T \mathbf{a_i}) - 1 \right)$$

the newton update rule is:

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \mu H^{-1} \nabla f_0$$

Newton's Method for Logistic Regression - Considerations

Biggest problem:

How to efficiently compute H^{-1} for:

$$H_{k,j} = -\sum_{i=1}^{m} a_{ik} a_{ij} \sigma(\mathbf{x}^T \mathbf{a_i}) \left(\sigma(\mathbf{x}^T \mathbf{a_i}) - 1 \right)$$

Considerations:

- H is symmetric: $H_{k,j} = H_{j,k}$

Outline

Matrix Inversion

Disclaimer: Never attempt to invert a matrix unless this is your last resort!

Given a matrix $A \in \mathbb{R}^{n \times n}$, its inverse A^{-1} is a matrix such that:

$$AA^{-1} = \mathbf{I}$$

Where:

- $\mathbf{I}$ is the identity matrix
- If no such matrix A^{-1} exists A is called a *singular* matrix or non-invertible

Matrix Inversion - Easy cases

Small Matrices:

For $A \in \mathbb{R}^{n \times n}$ with $n = 2$ or $n = 3$ it is still possible to compute A^{-1}

Orthogonal Matrices:

If $A \in \mathbb{R}^{n \times n}$ is Orthogonal then $A^T A = \mathbf{I}$ which means that $A^{-1} = A^T$

Matrix Inversion - Easy cases

Diagonal Matrices:

If $A \in \mathbb{R}^{n \times n}$ is diagonal, i.e. $A_{ij} = 0$ for all $i \neq j$, its inverse is a matrix A^{-1} such that

$$(A^{-1})_{i,i} = \frac{1}{A_{i,i}}$$

$$A = \begin{pmatrix} A_{1,1} & 0 & 0 & \cdots & 0 \\ 0 & A_{2,2} & 0 & \cdots & 0 \\ 0 & 0 & A_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & A_{n,n} \end{pmatrix} \quad A^{-1} = \begin{pmatrix} \frac{1}{A_{1,1}} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{A_{2,2}} & 0 & \cdots & 0 \\ 0 & 0 & \frac{1}{A_{3,3}} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{1}{A_{n,n}} \end{pmatrix}$$

Matrix Inversion - Conclusions

We can compute the inverse of the Hessian if it is:

- Low dimensional (2×2 or 3×3)
- Diagonal
- Orthogonal

What to do if that is not the case?

Avoiding the Inversion of the Hessian

Our goal is to compute the Newton Step:

$$\Delta \mathbf{x} = -\nabla^2 f_0(\mathbf{x})^{-1} \nabla f_0(\mathbf{x})$$

But we know:

- The gradient $\nabla f_0(\mathbf{x})$
- The Hessian $\nabla^2 f_0(\mathbf{x})$

So we can rearrange the Step equation:

$$\Delta \mathbf{x} = -\nabla^2 f_0(\mathbf{x})^{-1} \nabla f_0(\mathbf{x})$$
$$\nabla^2 f_0(\mathbf{x}) \Delta \mathbf{x} = -\nabla f_0(\mathbf{x})$$

Avoiding the Inversion of the Hessian

$$\nabla^2 f_0(\mathbf{x})\Delta\mathbf{x} = -\nabla f_0(\mathbf{x})$$

From this we know that the Newton step $\Delta\mathbf{x}$ is the solution to a linear system of equations:

$$A\mathbf{x} = \mathbf{b}$$

Where:

- $\mathbf{b}$ is the negative gradient $-\nabla f_0(\mathbf{x})$
- A is the Hessian $\nabla^2 f_0(\mathbf{x})$
- $\mathbf{x}$ is the Newton step $\Delta\mathbf{x}$

We need to know how to efficiently solve linear systems of equations!

Outline

Linear Systems of Equations

Given $A \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^{n \times 1}$, find $\mathbf{x} \in \mathbb{R}^{n \times 1}$ such that:

$$A\mathbf{x} = \mathbf{b}$$

A general method to find $\mathbf{x}$ such that $A\mathbf{x} - \mathbf{b} = 0$ is to solve:

$$\min_{\mathbf{x}} ||A\mathbf{x} - \mathbf{b}||_2^2$$

Depending of how A looks like there can be specific algorithms for solving the system:
For A **diagonal**:

$$\mathbf{x} = A^{-1}\mathbf{b} = \left(\frac{b_1}{A_{1,1}}, \frac{b_2}{A_{2,2}}, \ldots, \frac{b_n}{A_{n,n}} \right)$$

Linear Systems of Equations

For A **orthogonal**:

$$\mathbf{x} = A^{-1}\mathbf{b} = A^{T}\mathbf{b}$$

A special case of orthogonal matrices are **permutation matrices**:

Be $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ a permutation of $(1, 2 \ldots, n)$,

$$A_{i,j} = \begin{cases} 1 & \text{if } j = \pi_i \\ 0 & \text{otherwise} \end{cases}$$

Forward Substitution

For A **lower triangular**, i.e. $A_{i,j} = 0$ for all $i < j$:

$$
\begin{aligned}
x_1 &= \frac{b_1}{A_{1,1}} \\
x_2 &= \frac{b_2 - A_{2,1}x_1}{A_{2,2}} \\
x_3 &= \frac{b_3 - A_{3,1}x_1 - A_{3,2}x_2}{A_{3,3}} \\
&\vdots \\
x_n &= \frac{b_n - A_{n,1}x_1 - A_{n,2}x_2 - \ldots - A_{n,n-1}x_{n-1}}{A_{n,n}}
\end{aligned}
$$

This method is called **forward substitution**

Backward Substitution

For A **upper triangular**, i.e. $A_{i,j} = 0$ for all $i > j$:

$$
\begin{aligned}
x_n &= \frac{b_n}{A_{n,n}} \\
x_{n-1} &= \frac{b_{n-1} - A_{n-1,n} x_n}{A_{n-1,n-1}} \\
&\vdots \\
x_1 &= \frac{b_1 - A_{1,2} x_2 - A_{1,3} x_3 - \ldots - A_{1,n} x_n}{A_{1,1}}
\end{aligned}
$$

This method is called **backward substitution**

Factor and solve method

We can represent the matrix A as a product of different matrices $U_1, U_2, \ldots, U_p$:

$$A = U_1 U_2 \ldots U_p$$

Then we can solve the system by computing:

$$\mathbf{x} = A^{-1}\mathbf{b} = U_p^{-1} \ldots U_2^{-1} U_1^{-1}\mathbf{b}$$

Which boils down to solving p equations:

$$U_1 \mathbf{z}_1 = \mathbf{b}$$
$$U_2 \mathbf{z}_2 = \mathbf{z}_1$$
$$\vdots$$
$$U_p \mathbf{x} = \mathbf{z}_{p-1}$$

If U_i is diagonal, othogonal, lower or upper-triangular, the equations above are "easy" to solve!

LU Factorization

If A is nonsingular it can be factorized as:

$$A = PLU$$

Where:

- P is a permutation matrix
- L is Lower triangular
- U is upper Triangular

We can solve $A\mathbf{x} = \mathbf{b}$ for A nonsingular as follows:

1. LU-Factorize A such that $A = PLU$
2. Solve $P\mathbf{z}_1 = \mathbf{b}$
3. Solve $L\mathbf{z}_2 = \mathbf{z}_1$ with Forward Substitution
4. Solve $U\mathbf{x} = \mathbf{z}_2$ with Backward Substitution

Other Factorizations

Cholesky:

- For positive definite A
- $A = LL^T$
- L is lower triangular

LDLT:

- nonsingular symmetric A
- $A = PLDL^T P^T$
- L is lower triangular, P is a permutation matrix, D is block diagonal

Outline

Newton's method

The Newton's method can be then rewritten without the inverse of the Hessian as the follows:

Repeat until convergence:

1. Solve $\nabla^2 f_0(\mathbf{x})\Delta\mathbf{x} = -\nabla f_0(\mathbf{x})$ for $\Delta\mathbf{x}$

2. Get step size μ (line search)

3. Update $\mathbf{x}$: $\mathbf{x} \leftarrow \mathbf{x} + \mu\Delta\mathbf{x}$

Newton's method

1: **procedure** NEWTONS METHOD
input: f_0,
2: Get initial point $\mathbf{x}$
3: **repeat**
4: $\Delta\mathbf{x} \leftarrow$ Solve $\nabla^2 f_0(\mathbf{x})\Delta\mathbf{x} = -\nabla f_0(\mathbf{x})$
5: Get Step Size μ
6: $\mathbf{x} \leftarrow \mathbf{x} + \mu\Delta\mathbf{x}$
7: **until** convergence
8: **return** $\mathbf{x}$, $f_0(\mathbf{x})$
9: **end procedure**

Quasi-Newton Methods

Solving one linear system of equations per update can be infeasible for large scale problems

The same holds for computing and storing the second derivatives

Quasi-newton Methods replace the Newton update:

$$\Delta^{\text{Newton}}\mathbf{x} = -\nabla^2 f_0(\mathbf{x})^{-1}\nabla f_0(\mathbf{x})$$

with an approximation

$$\Delta^{\text{QN}}\mathbf{x} = -H^{-1}\nabla f_0(\mathbf{x})$$

where $H \succ 0$ is an approximation of the Hessian at $\mathbf{x}$

Quasi-Newton Method

1: **procedure** QUASI-NEWTON METHOD
input: f_0
2: Get initial point $\mathbf{x}^{(0)}$
3: $t \leftarrow 0$
4: **repeat**
5: Compute $H^{(t)^{-1}}$
6: $\Delta\mathbf{x} \leftarrow -H^{(t)^{-1}} \nabla f_0(\mathbf{x}^{(t)})$
7: Get Step Size μ
8: $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x} + \mu\Delta\mathbf{x}$
9: $t \leftarrow t + 1$
10: **until** convergence
11: **return** $\mathbf{x}$, $f_0(\mathbf{x})$
12: **end procedure**

Methods differ in how they perform line 4 (compute H^{-1})

Broyden-Fletcher-Goldfarb-Shanno (BFGS)

Be:

- $\mathbf{s} = \mathbf{x}^{(t)} - \mathbf{x}^{(t-1)}$
- $\mathbf{g} = \nabla f(\mathbf{x}^{(t)}) - \nabla f(\mathbf{x}^{(t-1)})$

We can update $H^{(t)}$:

$$H^{(t)} = H^{(t-1)} + \frac{\mathbf{g}\mathbf{g}^T}{\mathbf{g}^T\mathbf{s}} - \frac{H^{(t-1)}\mathbf{s}\mathbf{s}^T H^{(t-1)}}{\mathbf{s}^T H^{(t-1)}\mathbf{s}}$$

or we can update the inverse directly:

$$H^{(t)^{-1}} = \left(\mathbf{I} - \frac{\mathbf{s}\mathbf{g}^T}{\mathbf{g}^T\mathbf{s}}\right) H^{(t-1)^{-1}} \left(\mathbf{I} - \frac{\mathbf{g}\mathbf{s}^T}{\mathbf{g}^T\mathbf{s}}\right) + \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{g}^T\mathbf{s}}$$

Limited Memory BFGS (L-BFGS)

Although BFGS may be faster to compute, we still need to store H

L-BFGS solves this by storing the r most recent values of $\mathbf{s}$ and $\mathbf{g}$:

For $j = t - r, t - r + 1, t - r + 2, \ldots, t$:

$$\mathbf{s}^{(j)} = \mathbf{x}^{(j)} - \mathbf{x}^{(j-1)}$$
$$\mathbf{g}^{(j)} = \nabla f(\mathbf{x}^{(j)}) - \nabla f(\mathbf{x}^{(j-1)})$$

At each epoch t, $H^{(t)^{-1}}$ is computed recursively:
For $j = t - r, t - r + 1, t - r + 2, \ldots, t$:

$$H^{(j)^{-1}} = \left(\mathbf{I} - \frac{\mathbf{s}^{(j)}\mathbf{g}^{(j)^T}}{\mathbf{g}^{(j)^T}\mathbf{s}^{(j)}}\right) H^{(j-1)^{-1}} \left(\mathbf{I} - \frac{\mathbf{g}^{(j)}\mathbf{s}^{(j)^T}}{\mathbf{g}^{(j)^T}\mathbf{s}^{(j)}}\right) + \frac{\mathbf{s}^{(j)}\mathbf{s}^{(j)^T}}{\mathbf{g}^{(j)^T}\mathbf{s}^{(j)}}$$

with $H^{(t-r)^{-1}} = \mathbf{I}$