Exercise 1

Bowen Hua

September 11, 2017

1 Linear Regression

1.1 (A)

Matrix form:

$$\hat{\beta} = \arg\min_{\beta \in \mathcal{R}^P} \frac{1}{2}(y - X\beta)^T W (y - X\beta).$$

Since W is symmetric and positive semidefinite, this is a convex optimization problem. Its first-order optimality condition is necessary and sufficient:

$$X^T W (y - X\beta) = \mathbf{0}.$$

1.2 (B)

1.2.1 Method 1: direct inversion

Described in the problem statement:

$$\beta = (X^T W X)^{-1} X^T W y.$$

To exploit sparsity of W, we use broadcasting in Python instead of matrix multiplication to compute operations w.r.t. W matrix. This method has a complexity of $O(np^2)$.

1.2.2 Method 2: pseudoinverse

Since the weight matrix W is diagonal, we can define $W^{\frac{1}{2}}$ to be a diagonal matrix where the i-th diagonal element equals $\sqrt{w_i}$. Now we can re-write the optimality condition:

$$\beta = [(W^{\frac{1}{2}}X)^T (W^{\frac{1}{2}}X)]^{-1} (W^{\frac{1}{2}}X)^T y,$$

which we re-write as

$$\beta = (W^{\frac{1}{2}}X)^{\dagger} W^{\frac{1}{2}} y,$$

where $(W^{\frac{1}{2}}X)^{\dagger}$ is the pseudoinverse of $W^{\frac{1}{2}}X$.

We first "preprocess" the feature matrix X and y by multiplying $W^{\frac{1}{2}}$ on the left.

We then compute the pseudoinverse through computing SVD of $W^{\frac{1}{2}}X$. This method is numerically more stable than the direct inverse method. (There could be correlation between our observations X. Therefore we care about numerical stability.)

Pseudocode for pseudoinverse of matrix A:

$(U, \Sigma, V) = \texttt{svd}(A)$
for Σ_{ii} in Σ: `#traverse through the diagonal elements`
  if $\Sigma_{ii} \neq 0$:
    $\Sigma_{ii} = 1/\Sigma_{ii}$
return $V^T \Sigma U^T$

In the Python code, we call `numpy.linalg.pinv` to perform the pseudoinverse. This method also has a complexity of $O(np^2)$.

In addition, this is the implementation of `scikit-learn`.

1.2.3 Method 3: Cholesky decomposition

We can use Cholesky decomposition on the matrix $X^T W X$. Now we have $LL^T \beta = D$. Then we can obtain β by solving two linear systems.

This method also has a complexity of $O(np^2)$.

Pseudocode for Cholesky-decomposition-based method:

Let $C = X^T W X$, $d = X^T W y$.
Compute Cholesky decomposition $C = LL^T$.
Solve for α in $L\alpha = d$.
Solve for β in $L^T \beta = \alpha$.

1.3 (C)

I coded the three methods in Python, using the `numpy` package. The codes can be find in my GitHub.

The results are summarized here:

Table 1: CPU Times (s) for Three Methods of Weighted Least Squares

(n, p)	Method 1	Method 2	Method 3
$(2000, 50)$	0.001	0.007	0.001
$(1000, 1000)$	0.122	0.346	0.064
$(20000, 50)$	0.012	0.034	0.005
$(50000, 50)$	0.028	0.118	0.013
$(5000, 5000)$	6.402	37.00	4.462

We can see that:

- Method 3 (Choleskey) consistently performs better than Method 1 (Direct Inverse), which is faster than Method 2 (pseudoinverse).
- When X is close to a square matrix, the performance of Method 3 is way worse than the other two methods.

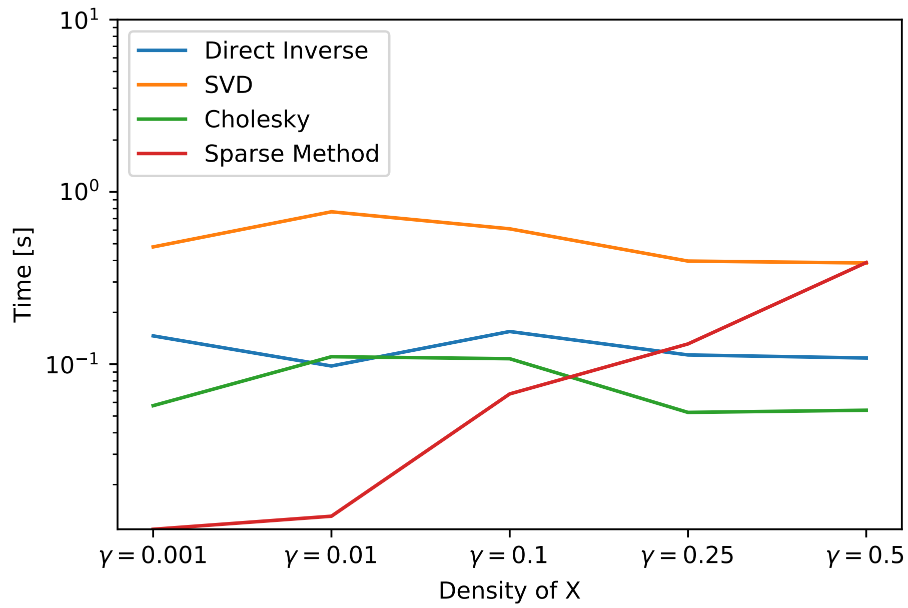
1.4 (D)

We use `scipy.sparse` in Python as our tool for sparse matrix operations. In particular, we use `scipy.sparse.linalg.lsqr` to solve the sparse least square problem:

$$\beta = (W^{\frac{1}{2}}X)^{\dagger}W^{\frac{1}{2}}y.$$

We generate random X matrices of size $(200000, 50)$, with different density. We name the sparse method as Method 4. The results are summarized as follows:

Figure 1: CPU Times (s) of Four Methods for the Sparse Matrix



We can see that:

- Method 1–3 do not exploit sparsity. Their CPU times do not change with density.
- When density is relatively small, the sparse method has an advantage over all the other three method. When density is relatively large, the sparse method is slower than Method 1 and 3, possibly due to overhead of the sparse data structure.

2 Generalized Linear Models

2.1 (A)

We have $y_i \sim \text{Binomial}(m_i, w_i)$, where

$$w_i = \frac{1}{1 + \exp(-x_i^T\beta)},$$

$$1 - w_i = \frac{\exp(-x_i^T\beta)}{1+\exp(-x_i^T\beta)}.$$

The negative log likelihood function is:

$$\begin{aligned}
\ell(\beta) &= -\log\left\{\prod_{i=1}^{N} p(y_i|\beta)\right\} && (1)\\
&= -\log\left\{\prod_{i=1}^{N}\binom{m_i}{y_i}(w_i)^{y_i}(1-w_i)^{m_i-y_i}\right\} && (2)\\
&= -\left\{\sum_{i=1}^{N}\left(\log\binom{m_i}{y_i} + y_i\log(w_i) + (m_i-y_i)\log(1-w_i)\right)\right\} && (3)
\end{aligned}$$

We have:

$$\begin{aligned}
\nabla \log w_i &= -\nabla\log(1+\exp(-x_i^T\beta)) && (4)\\
&= \frac{x_i\exp(-x_i^T\beta)}{1+\exp(-x_i^T\beta)} && (5)\\
&= x_i(1-w_i) && (6)
\end{aligned}$$

$$\begin{aligned}
\nabla\log(1-w_i) &= \nabla(-x_i^T\beta) - \nabla\log(1+\exp(-x_i^T\beta)) && (7)\\
&= -x_i + x_i(1-w_i) && (8)\\
&= -x_iw_i && (9)
\end{aligned}$$

Therefore, the gradient of the loss function is:

$$\begin{aligned}
\nabla\ell(\beta) &= \sum_{i=1}^{N}(y_ix_i(1-w_i) - (m_i-y_i)x_iw_i) && (10)\\
&= \sum_{i=1}^{N}(m_iw_i - y_i)x_i && (11)\\
&= X^T(m\circ w - y), && (12)
\end{aligned}$$

where operator $\circ$ denotes element-wise product.

2.2 (B)

We use the data wdbc.csv from course website. We use the first 10 features in our model. We add a column of ones into X matrix to represent the intercept term.

To alleviate the numerical issue brought by having a very large w_i value, we first scale the X data using `scikit-learn preprocessor`.

We perform a naive gradient descent with a fixed step length of 0.005. We compute 500 iterations. The codes are shown in `logit.py` and the loss function value as a function of iteration count is shown below:

Figure 2: Loss function value as a function of iteration count