Michael Wieck-Sosa

Professor Fu

Numerical Analysis

December 8, 2019

Reproduction of *Scalable Approximations for Generalized Linear Problems*

Abstract

In this reproduction, I will give an overview of the Mathematical derivation and experimental results of a scalable algorithm that approximates the population risk minimizer in generalized linear problems introduced by Erdogdu et al. (2019). Typically, the empirical risk is minimized using an iterative algorithm. However, when the number of observations grows much larger than the dimension of the feature space, these iterative algorithms become intractable. First, I will show that the true minimizer of the population risk is approximately proportional to the ordinary least squares estimator. Second, I will offer a limited experimental comparison of the proposed algorithm with a batch gradient descent algorithm. The experimental results are mostly consistent with the results of Erdogdu et al. (2019), which claim that the accuracy of the proposed algorithm is the same as the empirical risk minimizer. Lastly, and most interestingly, by analyzing the proposed scalable algorithm, we see it is computationally cheaper than classical batch methods such as gradient descent at least a factor of O(p).

Introduction

The topic of generalized linear models is well studied and has many examples such as binary classification (Buja et al., 2005; Reid and Williamson, 2010) and generalized linear models. The generalized linear models discussed in this paper are Poisson regression, logistic regression, and ordinary least squares (Nelder and Baker, 1972; McCullagh and Nelder, 1989).

Currently, these methods for stochastic optimization problems are popular because of their practicality in various industrial and technological settings.

Now, I will introduce the problem. Consider the stochastic optimization problem

$$minimize_{\beta \, \varepsilon \, \Re^p} \; := \; E[\psi(<x, \beta>) - y < x, \beta >]$$

where the function $\psi : \Re \rightarrow \Re$ is a nonlinear function, $y \,\varepsilon\, Y \subset \Re$ is the response variable, $x \,\varepsilon\, X \subset \Re^p$ is the predictor, and the expectation over the joint probability distribution *(y,x)*. This representation of a generalized linear problem is commonly used in the literature. Because the distribution of (*y,x*) is unknown, the empirical risk is approximated as

$$minimize_{\beta \varepsilon \Re^p} \; \widehat{\Re}(\beta) := \frac{1}{n} \sum_{i=1}^{n} \psi(<x_i, \beta>) - y < x_i, \beta > .$$

In the case of generalized linear models, this is equal to the maximum likelihood estimation. Second, in the case of binary classification, it is called the surrogate loss minimization. The gradient descent algorithm, which will be compared to the proposed algorithm in the experimental results section, is the first order approximation of the non-linear risk. It has a slow convergence rate, compared to other second-order methods such as the Newton-Raphson method, which is the most popular optimization method to compute the maximum likelihood estimation. However, the gradient descent algorithm has a O(np) per-iteration cost. While it is less computationally expensive than some other second-order optimization methods, it is still impractical for problems with a large number of observations.

As they are not crucial for this discussion or used in the experimental results section, the second-order methods will be glossed over. I will give their per-iteration cost, to demonstrate how they are not practical for large scale problems, and an explanation of why the computation has the per-iteration cost that it does.

Method	Per-iteration cost	Convergence rate	Bound of computation
Newton-Raphson	$O(np^2)$	Locally quadratic	-Scale gradient by inverse of Hessian matrix evaluated at

(McCullagh and Nelder, 1989; Buja et al., 2005).			each iteration.
Newton-Stein (Erdogdu, 2015, 2016	$O(np)$	Near quadratic	-Uses Stein's lemma and subsampling to efficiently estimate the Hessian matrix
Broyden-Fletcher-Goldfarb-Shanno (BFGS) (Nesterov, 2004)	$O(np)$	Locally super-linear	-Gradient scaled by a matrix found by using information from previous gradient computations
Limited memory BFGS Bishop (1995)	$O(np)$	Locally super-linear	-Only uses recent iterations and gradients to approximate the Hessian matrix more efficiently
Gradient descent	$O(np)$	Linear under certain assumptions	-Takes a step in the opposite direction of the gradient
Accelerated gradient descent (Nesterov, 1983)	$O(np)$	Linear under certain assumptions	-Uses an additional momentum term

Now, I will introduce the proposed approach to minimize the stochastic optimization problem

$$minimize_{\beta \, \varepsilon \, \Re^p} \; := \; E[\psi(< x, \beta >) - y < x, \beta >] \; .$$

Let β^{pop} be the true minimizer of the population risk, and let β^{ols} be the ordinary least squares coefficients calculated as $\beta^{ols} = E[xx^T]^{-1}E[xy]$. Under certain models, the population parameters are proportional to the OLS parameters $\beta^{pop} \; \alpha \; \beta^{ols}$. The relationship $\beta^{pop} \; \alpha \; \beta^{ols}$ implies that because the parameters are proportional, all that must be done is to find the scaling factor to scale the ordinary least squares coefficients. Computationally speaking, this is much cheaper than iteratively minimizing the empirical risk using the methods discussed above, particularly when the number of observations grows large.

So far, only the earlier version of the paper, Erdogdu et al. (2016), has considered the relationship $\beta^{pop} \; \alpha \; \beta^{ols}$ in the context of optimization, making this approach wholly new. However, this relationship is well-known and has been well-studied in the field of Statistics for decades. The relationship $\beta^{pop} \; \alpha \; \beta^{ols}$ was first noted by Fisher in his discussion of logistic regression (Fisher, 1936) and Brillinger's discussion about Gaussian models (Brillinger, 1982). Later on, the relationship $\beta^{pop} \; \alpha \; \beta^{ols}$ has lead to techniques in dimensionality reduction (Li, 1991; Li and Dong, 2009) and model misspecification problems (Li and Duan, 1989).

Mathematical Theory

Now, I will demonstrate the proof that β^{pop} is approximately proportional to β^{ols} in the random design setting, which is used for the proposed algorithm. That is, I will prove

$$|| \, \beta^{pop} \; - \; c_{\psi} \, x \, \beta^{ols} ||_{\infty} \; \leq \; c \, \frac{||\beta^{pop}||_{\infty}}{\sqrt{p}}$$

for some $c_{\psi} \varepsilon \Re$ that depends on ψ.

First, assume a random design setting with independent and identically distributed pairs $(y_i, x_i) \; i \; = 1, ..., n$ where y is the response variable and x is the vector of predictors. Second, no assumption is made about the joint distribution $(y_i, x_i) \; i \; = 1, ..., n$, so it is arbitrary.

Now, consider the population coefficients

$$\beta^{pop} = argmin_{\beta \, \varepsilon \, \Re^p} \; E[\psi(< x_i, \beta >) - y_i < x_i, \beta >] \; .$$

In the context of generalized linear models, ψ is the cumulant generating function for $y_i \mid x_i$ .so the problem can be understood as a standard generalized linear model with the link function and the regression coefficients β^{pop} . It is assumed that ψ is smooth. I will consider the following generalized linear models with the generating functions:

- Poisson regression: $\psi(w) = e^w$
- Linear Regression (OLS) : $\psi(w) = w^2 / 2$
- Logistic Regression: $\psi(w) = log\{1 + e^w\}$

Next, consider the OLS estimator $\beta^{ols} = (X^T X)^{-1} X^T y$ where $X = (x_1, ..., x_n)^T$ is the n x p matrix and $y = (y_1, ..., y_n)^T \varepsilon \, \Re^n$. Assume that $x_i \sim N_p(0, \Sigma)$. Then we can write $\beta^{pop} = c_\psi \, \beta^{ols}$ where $c_\psi \varepsilon \Re$ is the fixed point of the mapping $z \rightarrow E[\psi^{(2)}(< x_i, \beta^{OLS} > z)]^{-1}$.

Proof: Consider the stationary point minimization of

$$\beta^{pop} = argmin_{\beta \, \varepsilon \, \Re^p} \; E[\psi(< x_i, \beta >) - y_i < x_i, \beta >] \; .$$

The stationary point minimization satisfies the normal equations

$$E[y_i, x_i] = E[x_i \psi^{(1)}(< x_i, \beta >)] \; for \; i \; = \; 1, ..., p \, .$$

Next, let the multivariate normal density with mean 0 and covariance matrix Σ be $\Phi(x \mid \Sigma)$.

Using the fact that $\dfrac{d \, \Phi(x \mid \Sigma)}{dx} \; = \; - \Sigma^{-1} x \Phi(x \mid \Sigma)$ and integration by parts, we obtain

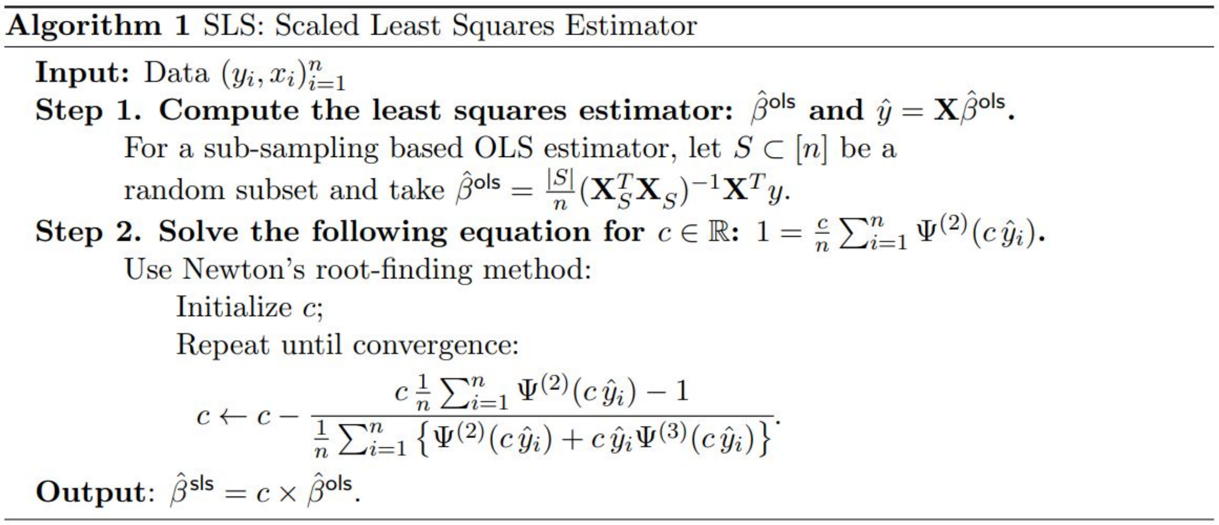
$$E[x_i \psi^{(1)}(< x_i, \beta >)] = \int x \, \psi^{(1)}(< x_i, \beta >) \, \Phi(x \mid \Sigma) \, dx$$

$$= \Sigma \, \beta \, E[\psi^{(2)}(< x_i, \beta >)] \; for \; i \; = \; 1, \; ... \; , \; p \, .$$

The final result that we can write $\beta^{pop} = c_\psi \, \beta^{ols}$ where $c_\psi \varepsilon \Re$ is the fixed point of the mapping $z \rightarrow E[\psi^{(2)}(< x_i, \beta^{OLS} > z)]^{-1}$ is obtained by multiplying both sides by Σ^{-1} and using the normal equations introduced above.

Numerical Method

In this paper, I will give an overview of the algorithm for the scaled least squares estimator reproduced from Erdogdu et al. (2019) and an implementation of the algorithm as a Python class.



```
Algorithm 1 SLS: Scaled Least Squares Estimator
  Input: Data $(y_i, x_i)_{i=1}^n$
  Step 1. Compute the least squares estimator: $\hat{\beta}^{\mathsf{ols}}$ and $\hat{y} = \mathbf{X}\hat{\beta}^{\mathsf{ols}}$.
      For a sub-sampling based OLS estimator, let $S \subset [n]$ be a
      random subset and take $\hat{\beta}^{\mathsf{ols}} = \frac{|S|}{n}(\mathbf{X}_S^T\mathbf{X}_S)^{-1}\mathbf{X}^T y$.
  Step 2. Solve the following equation for $c \in \mathbb{R}$: $1 = \frac{c}{n}\sum_{i=1}^n \Psi^{(2)}(c\,\hat{y}_i)$.
      Use Newton's root-finding method:
          Initialize $c$;
          Repeat until convergence:
              $c \leftarrow c - \dfrac{c\,\frac{1}{n}\sum_{i=1}^n \Psi^{(2)}(c\,\hat{y}_i) - 1}{\frac{1}{n}\sum_{i=1}^n \left\{\Psi^{(2)}(c\,\hat{y}_i) + c\,\hat{y}_i\Psi^{(3)}(c\,\hat{y}_i)\right\}}$.
  Output: $\hat{\beta}^{\mathsf{sls}} = c \times \hat{\beta}^{\mathsf{ols}}$.
```

The algorithm uses the mathematical results explained above to find the scaling factor c to satisfy the relationship $\beta^{pop} = c_\psi \beta^{ols}$ where $c_\psi \varepsilon \Re$ is the fixed point of the mapping $z \rightarrow E[\psi^{(2)}(< x_i, \beta^{OLS} > z)]^{-1}$. The algorithm has two main steps.

Step 1. The main computational cost of the algorithm is the computation for the ordinary least squares estimator in Step 1. Thus, a subsampling of the observations to estimate the covariance matrix can be used to cut down down the computational cost of the algorithm significantly. The properties of sub-sampling are studied by Vershynin, 2010; Dhillon et al., 2013; Erdogdu and Montanari, 2015; Pilanci and Wainwright, 2015; Roosta-Khorasani and Mahoney, 2016a,b. The computational cost of Step 1 is $O(|S|p^2 + p^3 + np) \approx O(pmax\{p^2log(p), n\}$.

Step 2. The second step of the algorithm consists of a root-finding method. In the Python implementation below, I use Newton's method, but other root finding methods will work. The per-iteration cost of this algorithm is $O(n)$ and it is able to obtain fast convergence rates of second order algorithms discussed previously, such as gradient descent. The Python implementation is in the appendix as Figure 1. Also, the classical batch gradient descent algorithm used to compare results is implemented in Python as Figure 2.

Numerical Results

The experimental comparisons in this reproduction are limited. For a more comprehensive comparison of the proposed scaled least squares estimator across various optimization algorithms and various algorithms see the paper Erdogdu et al. (2019). Erdogdu et al. (2019) explain that in the regime where the maximum likelihood estimator is expensive to compute ($n >> p >> 1$), the SLS is found much more efficiently while having the same accuracy. The limited experimental results obtained in this reproduction are mostly consistent with their results. That being said, there are some abnormalities caused by numerical errors that must be further investigated.

A preliminary investigation suggests that there are numerical errors because of large floating point operations resulting in the runtime errors "invalid value" or "overflow" encountered in double_scalars. In particular, this is because the range of the numpy double is $(-1.79769313486e+308\ 1.79769313486e+308)$ following IEEE floating point restrictions. The warnings are raised in the following parts of the Python implementation of the algorithm above:

- return np.exp(w) / (np.exp(w) + 1)**2
- c*self.yhat[i]*self.psi3(c*self.yhat[i]))
- c = c - self.f(c) / self.derivative(c)

After handling these errors, we can observe the consequences of these abnormalities by comparing the reproduction results to the plots reproduced from Erdogdu et al. (2019) in the appendix below. First, it must be stated that I am open to the possibility that there are mistakes in my implementation of the algorithm. However, beyond the abnormalities, the results remain consistent with Erdogdu et al. (2019), which is strange if my implementation is incorrect. This requires further investigation. Second, it is important to note that the performance of the algorithm does change to a moderate degree as the synthetic data and error distributions are implemented differently, so this could be the cause.[1] Therefore, it is possible that the synthetic data and error distributions I generated have caused these numerical errors. It would be best to ask the authors of the paper exactly how the synthetic datasets and error distributions were generated, as well as how the subsampling was done.

Following Erdogdu et al. (2019), I will show results of the proposed algorithm for the number of observations n $log_{10}(n) = \{3, 4, 5, 6\}$ and the dimension of the feature space $p = 200$. The design contexts that will be used are logistic regression and linear regression (OLS) with independent and identically distributed Gaussian design, and Poisson regression with independent and identically distributed Poisson design. In every case, the algorithm will only be compared to a batch gradient descent optimization algorithm that estimates the maximum likelihood estimator. The max number of iterations will be set to 1,000 and the tolerance will be set to 1e-5. Lastly, the synthetic data generated in Python is in the appendix as Figure 3. The results from Erdogdu et al. (2019) are reproduced in the appendix as Figure 4, 5, 6. My results are printed in the appendix are Figures 7, 8, 9 for logistic regression, Figures 10, 11, 12, for Poisson regression, and Figures 13, 14, 15 for linear regression.

Conclusion

This reproduction has given an overview of the Mathematical derivations and experimental results for the central numerical method proposed in in Erdogdu et al. (2019). Although the experimental reproduction is limited in that it only compares the proposed algorithm with a classical batch gradient descent algorithm, the results are mostly consistent, beyond the abnormalities caused by numerical errors. Future work going beyond this paper is using this approach to approximate the L2-regularized empirical risk minimizer in deep learning or other contexts in which regularization is crucial.

[1] Claim has not been supported with evidence, but this has been my experience when testing the algorithm.

References

Christopher M. Bishop. Neural Networks for Pattern Recognition. Oxford University Press, 1995.

David R Brillinger. A generalized linear model with ”Gaussian” regressor variables. In A Festschrift For Erich L. Lehmann, pages 97–114. CRC Press, 1982.

Andreas Buja, Werner Stuetzle, and Yi Shen. Loss functions for binary class probability estimation and classification: Structure and applications. 2005.

Paramveer Dhillon, Yichao Lu, Dean P Foster, and Lyle Ungar. New subsampling algorithms for fast least squares regression. In Advances in Neural Information Processing Systems, pages 360–368, 2013.

Naihua Duan and Ker-Chau Li. Slicing regression: a link-free regression method. The Annals of Statistics, pages 505–530, 1991.

Murat A Erdogdu. Newton-stein method: A second order method for glms via stein’s lemma. In Advances in Neural Information Processing Systems, pages 1216–1224, 2015.

Murat A. Erdogdu. Newton-stein method: an optimization method for glms via stein’s lemma. The Journal of Machine Learning Research, 17(1):7565–7616, 2016.

Murat A Erdogdu. Stein’s lemma and subsampling in large-scale optimization. PhD thesis, Stanford University, 2017.

Murat A Erdogdu and Andrea Montanari. Convergence rates of sub-sampled newton methods. In Advances in Neural Information Processing Systems, pages 3052–3060, 2015.

Murat A Erdogdu, Mohsen Bayati, and Lee H Dicker. Scaled least squares estimator for glms in large-scale problems. In Advances in Neural Information Processing Systems, 2016.

Murat A Erdogdu, Mohsen Bayati, and Lee H Dicker. Scalable Approximations for Generalized Linear Problems. In Journal of Machine Learning Research. 2019.

Ronald A. Fisher. The use of multiple measurements in taxonomic problems. Annals Eugenic, 7:179–188, 1936.

Bing Li and Yuexiao Dong. Dimension reduction for nonelliptically distributed predictors. The Annals of Statistics, pages 1272–1298, 2009.

Ker-Chau Li. Sliced inverse regression for dimension reduction. Journal of the American Statistical Association, 86(414):316–327, 1991.

Ker-Chau Li and Naihua Duan. Regression analysis under link violation. Annals of Statistics, 17:1009–1052, 1989.

Peter McCullagh and John A. Nelder. Generalized Linear Models. Chapman and Hall, 2nd edition, 1989.

John A Nelder and R. Jacob Baker. Generalized linear models. Wiley Online Library, 1972.

Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/k2). Soviet Math. Dokl., 27:372–376, 1983. Yurii Nesterov. Introductory Lectures on Convex Optimization: A Basic Course. Springer, 2004.

Mert Pilanci and Martin J Wainwright. Newton sketch: A linear-time optimization algorithm with linear-quadratic convergence. arXiv preprint arXiv:1505.02250, 2015.

Mark D Reid and Robert C Williamson. Composite binary losses. Journal of Machine Learning Research, 11(Sep):2387–2422, 2010.

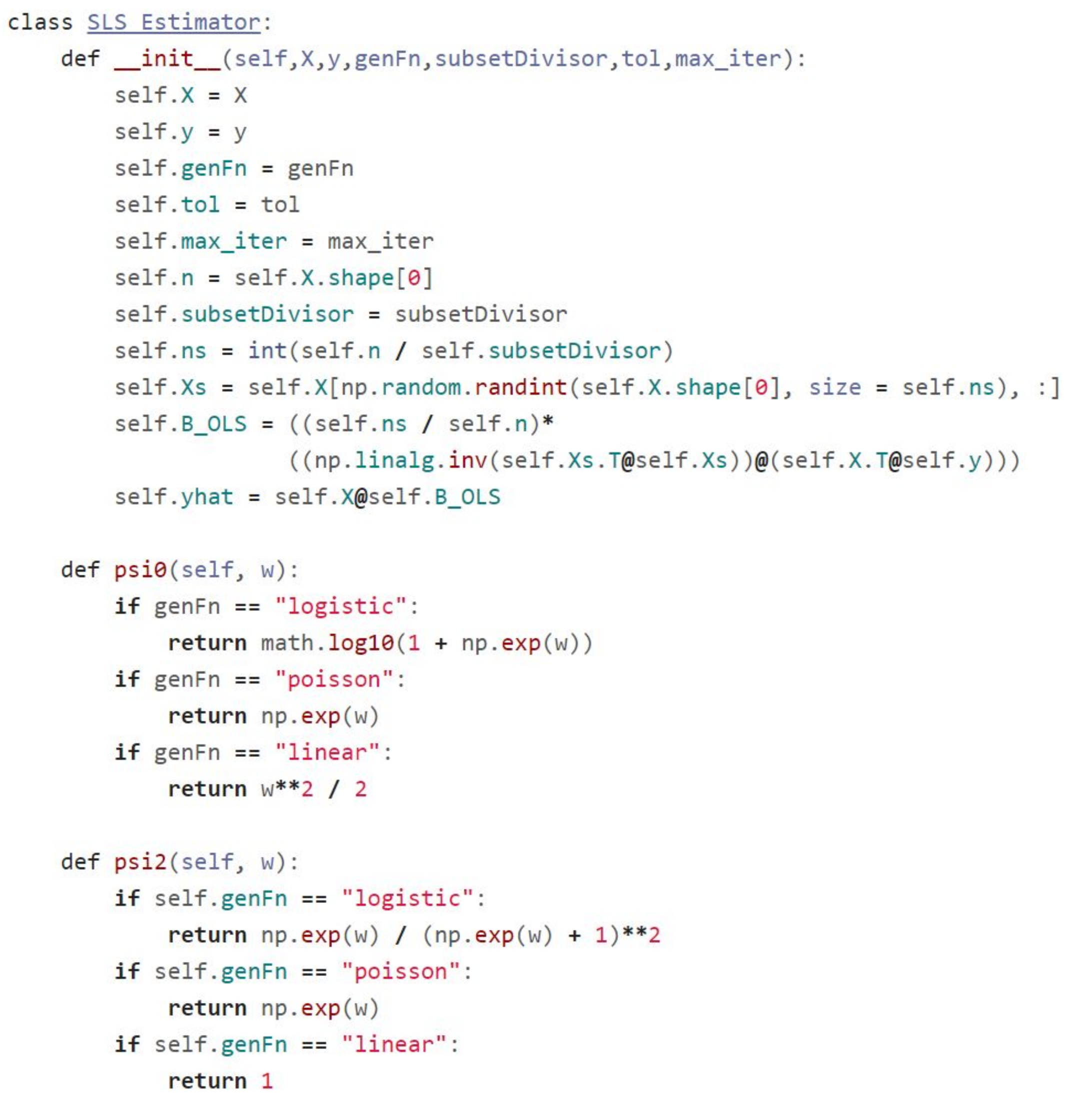
Farbod Roosta-Khorasani and Michael W Mahoney. Sub-sampled newton methods i: globally convergent algorithms. arXiv preprint arXiv:1601.04737, 2016a.

Farbod Roosta-Khorasani and Michael W Mahoney. Sub-sampled newton methods ii: Local convergence rates. arXiv preprint arXiv:1601.04738, 2016b.

Roman Vershynin. Introduction to the non-asymptotic analysis of random matrices, 2010. arXiv:1011.3027.

Appendix

Figure 1. Python Implementation of the Scaled Least Squares algorithm.



```
class SLS_Estimator:
    def __init__(self,X,y,genFn,subsetDivisor,tol,max_iter):
        self.X = X
        self.y = y
        self.genFn = genFn
        self.tol = tol
        self.max_iter = max_iter
        self.n = self.X.shape[0]
        self.subsetDivisor = subsetDivisor
        self.ns = int(self.n / self.subsetDivisor)
        self.Xs = self.X[np.random.randint(self.X.shape[0], size = self.ns), :]
        self.B_OLS = ((self.ns / self.n)*
                     ((np.linalg.inv(self.Xs.T@self.Xs))@(self.X.T@self.y)))
        self.yhat = self.X@self.B_OLS

    def psi0(self, w):
        if genFn == "logistic":
            return math.log10(1 + np.exp(w))
        if genFn == "poisson":
            return np.exp(w)
        if genFn == "linear":
            return w**2 / 2

    def psi2(self, w):
        if self.genFn == "logistic":
            return np.exp(w) / (np.exp(w) + 1)**2
        if self.genFn == "poisson":
            return np.exp(w)
        if self.genFn == "linear":
            return 1
```

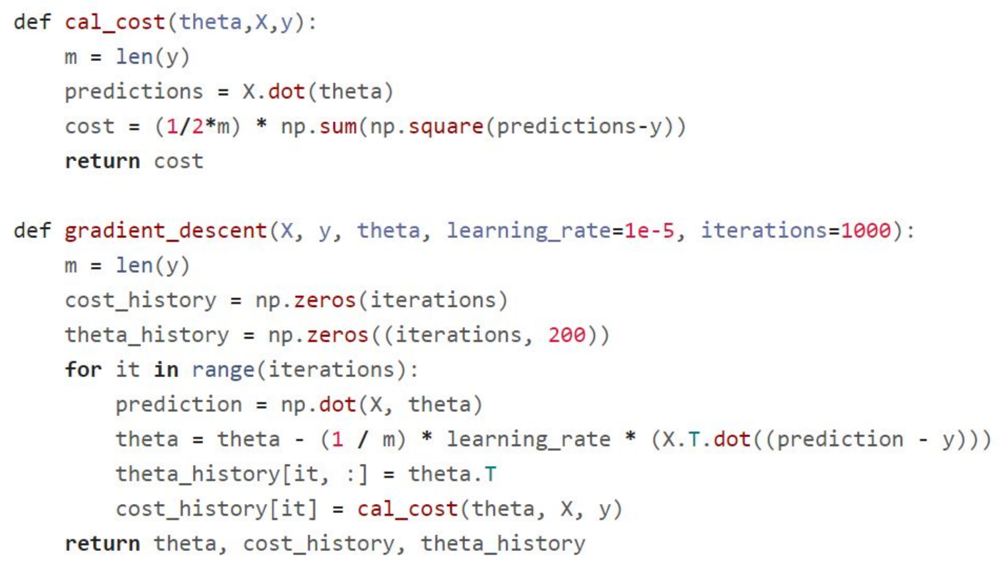
```
def psi3(self, w):
    if self.genFn == "logistic":
        return np.exp(w) / (np.exp(w) + 1)**3
    if self.genFn == "poisson":
        return np.exp(w)
    if self.genFn == "linear":
        return 0

def f(self, c):
    rsum = 0
    for i in range(self.n):
        rsum += self.psi2(c*self.yhat[i])
    return (c / self.n) * rsum - 1

def derivative(self, c):
    rsum = 0
    for i in range(self.n):
        rsum += (self.psi2(c*self.yhat[i])+
                c*self.yhat[i]*self.psi3(c*self.yhat[i]))
    return rsum / n

def getSLS_Estimator(self):
    c = 1 #init for newton's method
    it = 0
    while abs(self.f(c)) > self.tol and it < self.max_iter:
        c = c - self.f(c) / self.derivative(c)
        it += 1
    return c, self.B_OLS # B_SLS = c * B_OLS
```

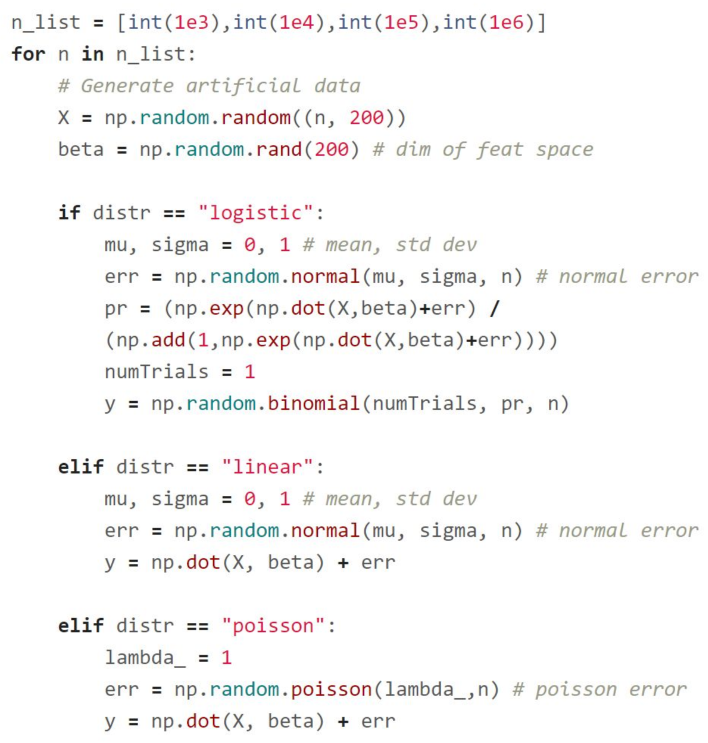
Figure 2. Python implementation of gradient descent algorithm.



```
def cal_cost(theta,X,y):
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/2*m) * np.sum(np.square(predictions-y))
    return cost

def gradient_descent(X, y, theta, learning_rate=1e-5, iterations=1000):
    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations, 200))
    for it in range(iterations):
        prediction = np.dot(X, theta)
        theta = theta - (1 / m) * learning_rate * (X.T.dot((prediction - y)))
        theta_history[it, :] = theta.T
        cost_history[it] = cal_cost(theta, X, y)
    return theta, cost_history, theta_history
```

Figure 3. Generation of synthetic data for experiments.



```
n_list = [int(1e3),int(1e4),int(1e5),int(1e6)]
for n in n_list:
    # Generate artificial data
    X = np.random.random((n, 200))
    beta = np.random.rand(200) # dim of feat space

    if distr == "logistic":
        mu, sigma = 0, 1 # mean, std dev
        err = np.random.normal(mu, sigma, n) # normal error
        pr = (np.exp(np.dot(X,beta)+err) /
        (np.add(1,np.exp(np.dot(X,beta)+err))))
        numTrials = 1
        y = np.random.binomial(numTrials, pr, n)

    elif distr == "linear":
        mu, sigma = 0, 1 # mean, std dev
        err = np.random.normal(mu, sigma, n) # normal error
        y = np.dot(X, beta) + err

    elif distr == "poisson":
        lambda_ = 1
        err = np.random.poisson(lambda_,n) # poisson error
        y = np.dot(X, beta) + err
```

Figure 4. Results comparing SLS and MLE from Erdogdu et al. (2019).

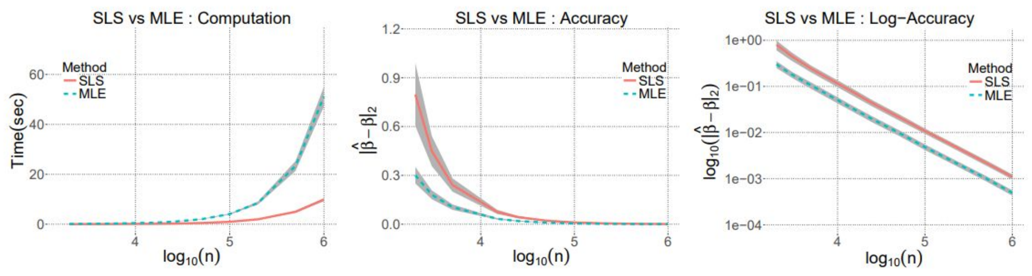


Figure 1: Logistic regression with iid standard Gaussian design. The left plot shows the computational cost (time) for finding the MLE and SLS as n grows and $p = 200$. The middle and the right plots depict the accuracy of the estimators in standard and log scales, respectively. In the regime where the MLE is expensive to compute, the SLS is found much more rapidly and has the same accuracy. R's built-in functions are used to find the MLE.

Figure 5. Results of logistic regression from Erdogdu et al. (2019).

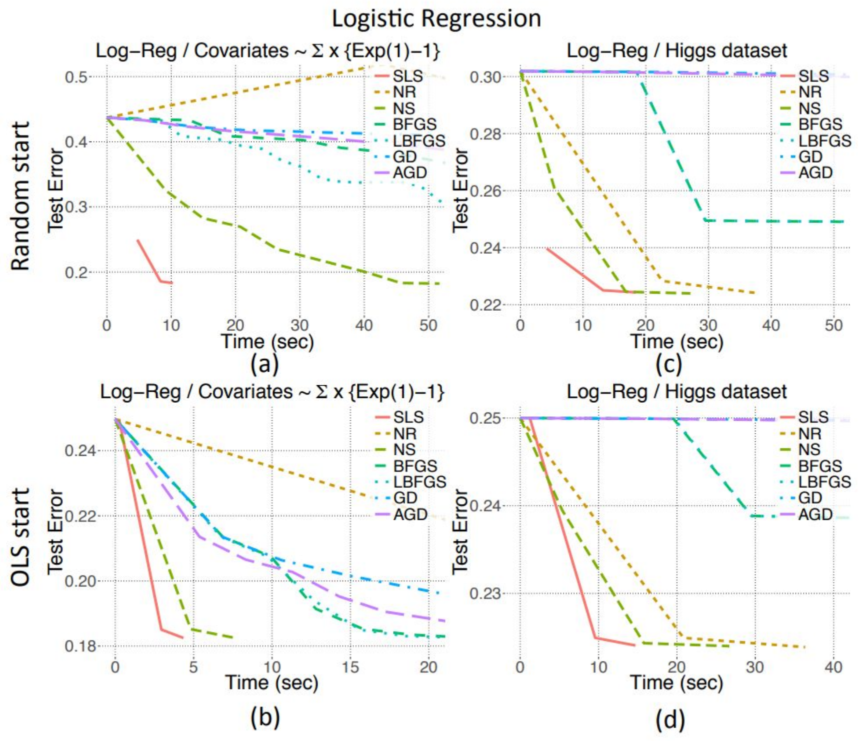


Figure 4: We compared the performance of SLS to that of MLE for the logistic regression problem on several datasets. MLE optimization is solved by various optimization algorithms. SLS is represented with red straight line. The details are provided in Table 2.

Figure 6. Results of Poisson regression from Erdogdu et al. (2019).

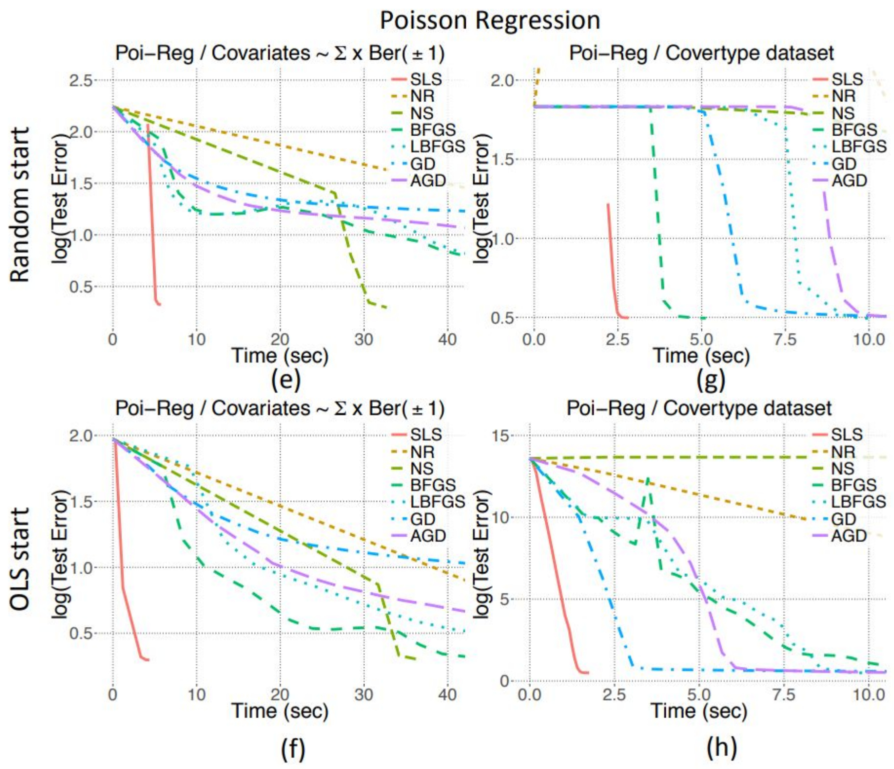


Figure 5: We compared the performance of SLS to that of MLE for the Poisson regression problem on several datasets. MLE optimization is solved by various optimization algorithms. SLS is represented with red straight line. Details are given in Table 2.

Figure 7. Reproduction results for logistic regression convergence. Note an abnormality caused by numerical errors due to overly large NumPy doubles at p = 100. Consistent with Erdogdu et al. (2019) except for the abnormality.

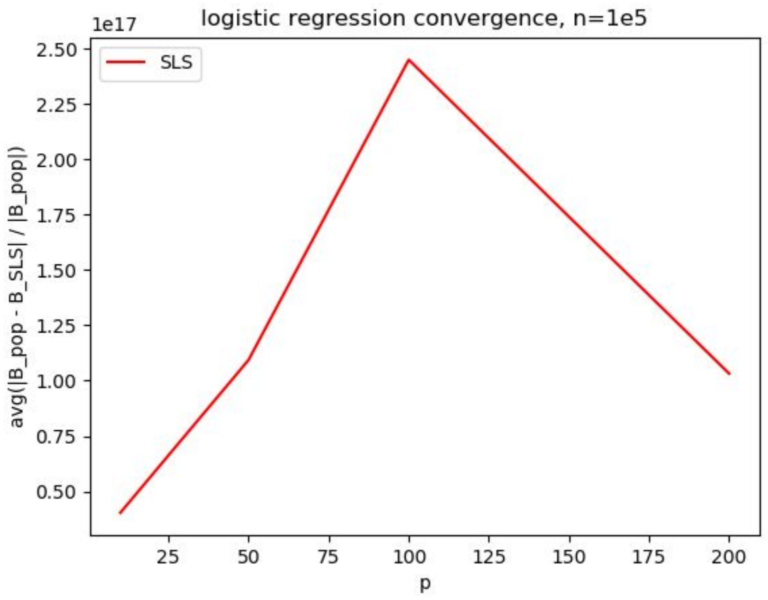


Figure 8. Reproduction results for logistic regression error. Note an abnormality caused by numerical errors due to overly large NumPy doubles at n = 1e6. Consistent with Erdogdu et al. (2019) except for the abnormality.

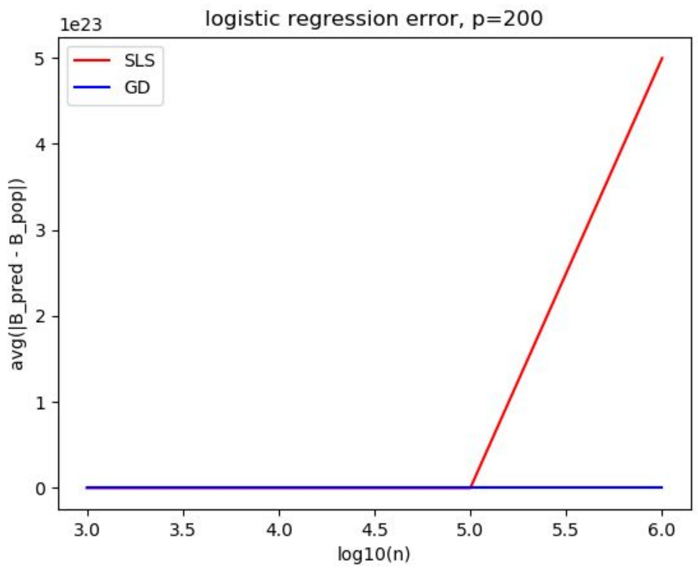


Figure 9. Reproduction results for logistic regression time. SLS is faster than the GD algorithm. Consistent with Erdogdu et al. (2019).

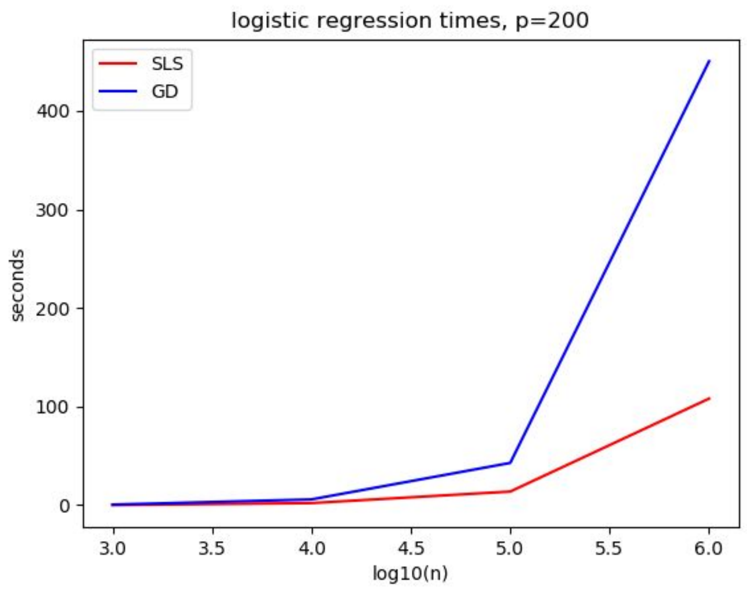


Figure 10. Reproduction results for Poisson regression convergence. Note an abnormality caused by numerical errors due to overly large NumPy doubles at p = 50. Consistent with Erdogdu et al. (2019) except for the abnormality.

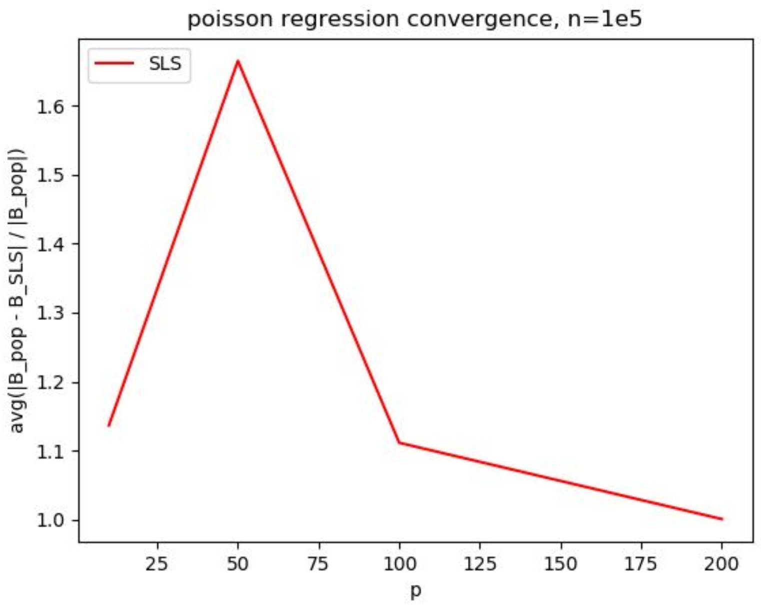


Figure 11. Reproduction results for Poisson regression error. Consistent with Erdogdu et al. (2019).

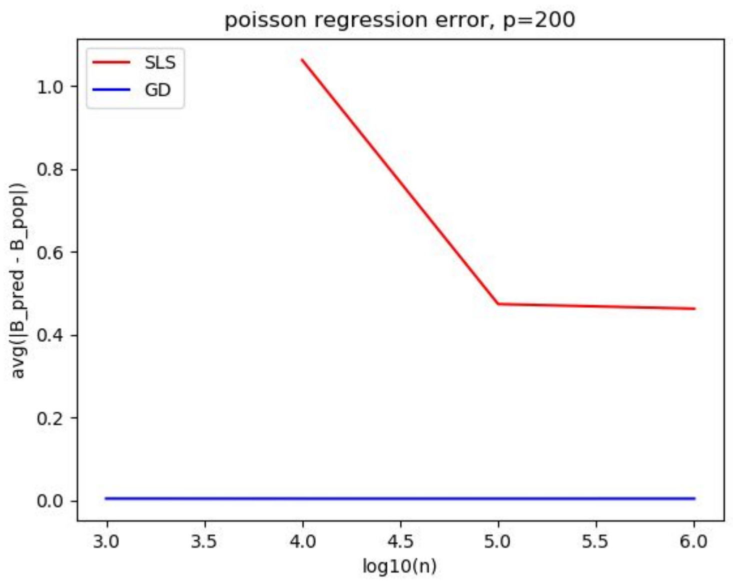


Figure 12. Reproduction results for Poisson regression time. These results are not consistent with Erdogdu et al. (2019).

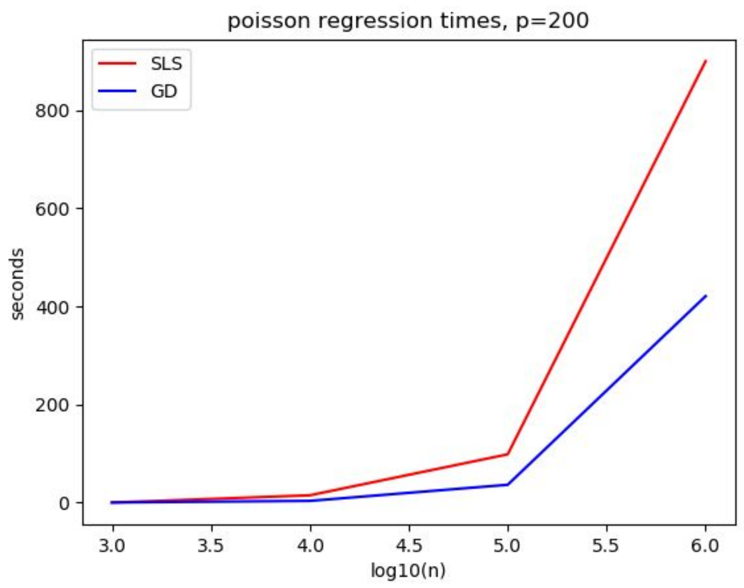


Figure 13. Reproduction results for linear regression convergence. The results are surprising because the coefficients do not converge. Result not shown by Erdogdu et al. (2019).

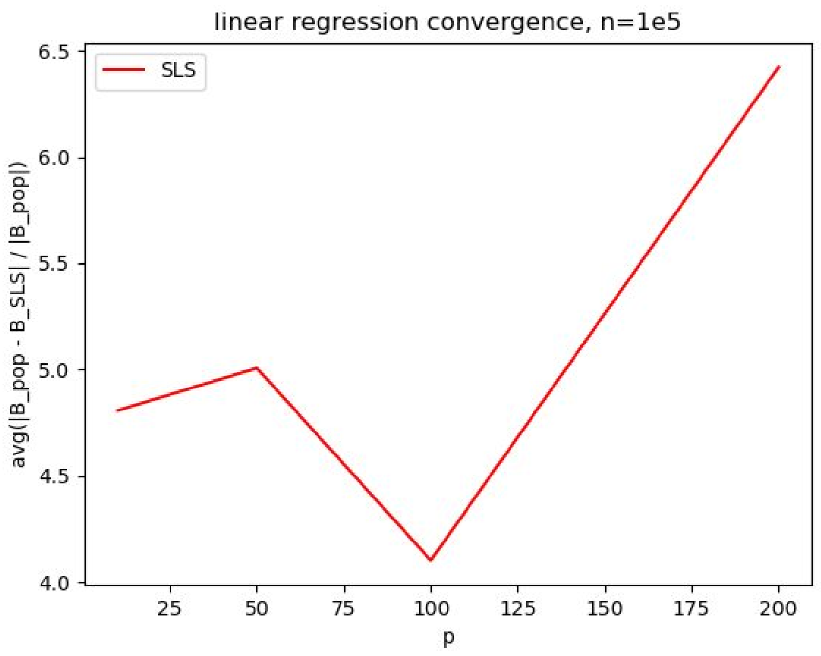


Figure 14. Reproduction results for linear regression error. We see an abnormality at n = 1e3 caused by numerical errors. Result not shown by Erdogdu et al. (2019).

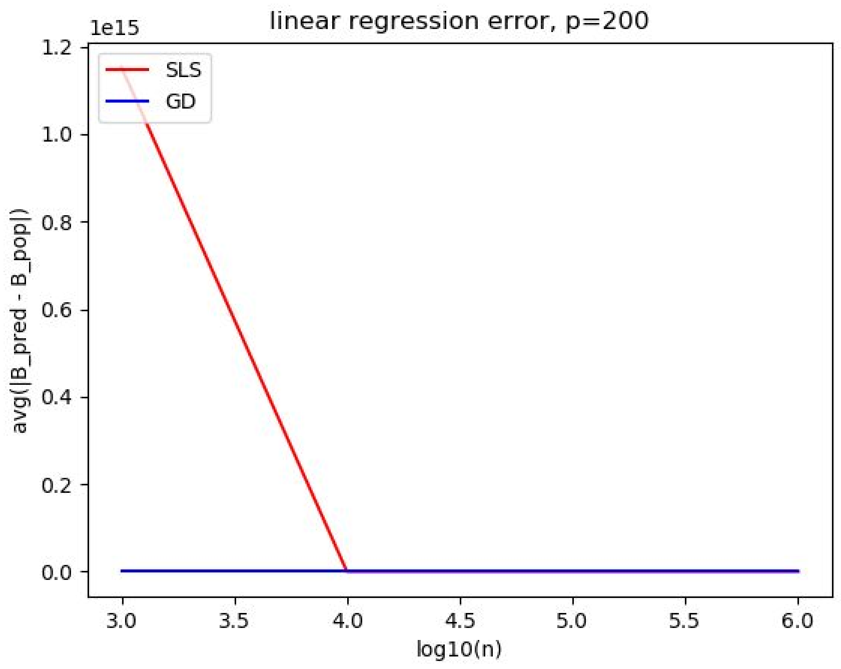
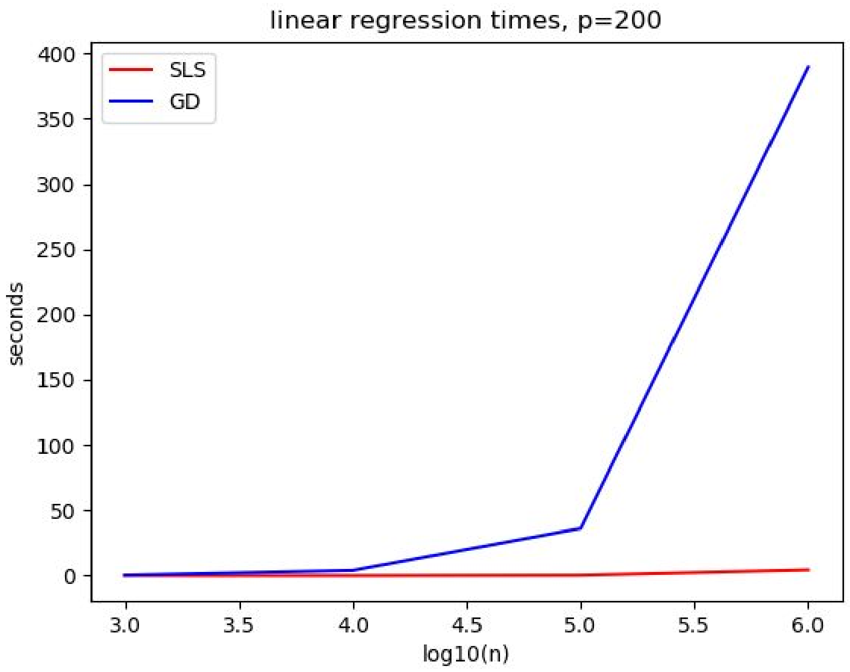


Figure 15. Reproduction results for linear regression time. Results consistent with other generating functions in the SLS is faster than GD. Result not shown by Erdogdu et al. (2019).


linear regression times, p=200
SLS
GD
seconds
log10(n)