Comparative Analysis of Optimization Techniques for Logistic Regression

A Study on Mini-Batch, Stochastic, Batch Gradient Descent, and Newton's Method

Jeff Anderson, Anant Mashiana, Weilin Cheng, Xinhui Luo, Erick S. Arenas

March 2023

Abstract: The aim of this project is to optimize logistic regression by utilizing different machine learning algorithms. In general, logistic regression is a statistical model to classify and predict data. In our project, we mainly applied four methods, including Stochastic Gradient Descent, Mini-Batch, Full Batch, and Newton's Method. The results show that while implementing different algorithms to build a logistic regression model, the predictive accuracy and time consumption can vary drastically. We propose that Mini-Batch Gradient Descent provides the best balance between accuracy, computational speed, and consistency.

Contents

1 Background **1**
1.1 Logistic/Logit Model 1
1.2 Logistic Regression Classification 1

2 Methods **2**
2.1 Gradient Descent 2
2.1.1 Batch Gradient Descent 3
2.1.2 On-Line Stochastic Gradient Descent 3
2.1.3 Mini-Batch Gradient Descent 3
2.2 Newton's Method 4

3 Results – Time Analysis and Accuracy Analysis **4**

4 Conclusion **8**

5 Appendix **9**

6 Functions **11**

7 Reference **12**

8 Code Appendix - see next page

1 Background

1.1 Logistic/Logit Model

Logistic regression is a method of modeling the conditional probabilities of a binary output for a given linear combination of the covariates. It specifically arises when the output variable can be mapped to values of 0 or 1 and is more broadly speaking part of a larger class of models known as Generalized Linear models.

Generalized linear models encompass a wide array of models, the most common of which is perhaps the normal linear regression model. Agresti explains that there are three fundamental components to Generalized Linear models: 1) a random component which identifies the response variable Y and it probability distribution; 2) a linear predictor element which specifies the explanatory variables through a prediction equation that has a linear form; and 3) a Link function which specifies a function of $E(Y)$ that the GLM relates to the linear predictor. (Agresti, 66)

In the case of Logistic regression models – also known as logit models – the response variable Y is characterized by a binomial distribution, and the link function is the logit, or log odds function. The general form of the logit model that expresses the log odds of P $(Y = 1|x, w)$ where x is a column vector of covariates and w is a column vector of weights is:

$$Logit(P(Y = 1)) = x^T w \tag{1}$$

It is often more convenient to reformulate the equation above such that we are modeling the P(Y = 1) directly itself. This is accomplished by applying the inverse of the logit function, the logistic function, to each side of the equation. Doing so we have:

$$P(Y = 1) = logistic(x^T w) \tag{2}$$

Throughout the rest of the report the logistic function will be denoted by σ and referred to as the sigmoid function such that:

$$\sigma(z) = logistic(z) = 1/(1 + exp(-z)) \tag{3}$$

1.2 Logistic Regression Classification

Logistic regression may be readily modified into a classification algorithm by specifying some cut off threshold such that we classify:

$$\hat{Y} = \begin{cases} 1, & \sigma(x^T w) > 0.5 \\ 0, & otherwise \end{cases} \tag{4}$$

Thus in order to get a predicted value for our weighted linear combination of variables we need only pass it through our classifier function, the logistic function – which returns a probability – and then apply a decision rule to the resulting probability to determine a label classification.

However, the process for predicting class-labels is not so simple. The weights w in the linear combination of predictor variables are unknown parameters and must be estimated. Since the output is modeled by the binomial distribution, it belongs to the **exponential family of models**, which is known to have a **concave likelihood**. Common wisdom suggests then that we need only to maximize the log-likelihood of the data in order to ascertain the maximum likelihood estimate for our weight parameters. However, again, the process is not so straightforward. Attempts to maximize the log likelihood of the data lead to a transcendental equation, which means that it does not have a closed form solution. Thus it becomes necessary to approximate the MLE via iterative methods. Some of the most common approaches for approximating the MLE for logistic regression are Gradient Descent and Newton's Method.

Conditional Probability

$$\begin{aligned} P(Y=1 \mid \mathbf{X}=\mathbf{x}) &= \sigma\left(w^T\mathbf{x}\right) \\ P(Y=0 \mid \mathbf{X}=\mathbf{x}) &= 1-\sigma\left(w^T\mathbf{x}\right) \\ P(Y=y \mid X=\mathbf{x}) &= \sigma\left(w^T\mathbf{x}\right)^y \cdot \left[1-\sigma\left(w^T\mathbf{x}\right)\right]^{(1-y)} \end{aligned} \tag{5}$$

Likelihood of Data

$$\begin{aligned} L(w) &= \prod_{i=1}^{n} P\left(Y=y^{(i)} \mid X=\mathbf{x}^{(i)}\right) \\ &= \prod_{i=1}^{n} \sigma\left(w^T\mathbf{x}^{(i)}\right)^{y^{(i)}} \cdot \left[1-\sigma\left(w^T\mathbf{x}^{(i)}\right)\right]^{\left(1-y^{(i)}\right)} \end{aligned} \tag{6}$$

Log Likelihood of Data

$$LL(w) = \sum_{i=1}^{n} y^{(i)} \log\sigma\left(w^T\mathbf{x}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left[1-\sigma\left(w^T\mathbf{x}^{(i)}\right)\right] \tag{7}$$

2 Methods

It is useful to first note that both Gradient Descent and Newton's Method rely on an approximation of the Taylor expansion of polynomials. The Taylor expansion of a polynomial says that if f: R $\rightarrow$ R is infinitely differentiable at $x\epsilon R$ then the Taylor series for f at x is the following power series[8]:

$$f(x)+f'(x)\Delta x+f''(x)\frac{(\Delta x)^2}{2!}+...+f^k(x)\frac{(\Delta x)^k}{k!}+... \tag{8}$$

Now, assume Δ x is sufficiently small such that:

$$(\Delta x)^k \rightarrow 0 \ as \ k \rightarrow \infty \tag{9}$$

When this assumption holds, the higher-order terms of the Taylor expansion may be dropped while still retaining an adequate and meaningful approximation to the original function. Due to iterative methods being required, it is essential that these algorithms are implemented efficiently. Failure to do so can lead to drastically increased computation times in the "big data setting" as the iterative nature will compound any implementation inefficiencies present in the algorithms. The next sections will explore these iterative methods in greater detail.

2.1 Gradient Descent

The first method that was explored to find the optimal weights for the logistic regression classification algorithm was Gradient Descent. Gradient Descent is a widely used method that only uses up to the first-order derivative term of the Taylor Expansion. Gradient Descent is a general class of optimization algorithms that involves a simple series of steps toward the global minimum. Maximizing the log-likelihood has been re-framed as minimizing the loss function, which was taken to be the cross-entropy of the data at each iteration (see appendix for reformulation). Since the objective is to now minimize the cost function, the gradient of the loss function is taken at each step of the iteration as the gradient denotes the direction of the steepest ascent. With the gradient calculated, the algorithm takes a "step" in the direction opposite to the gradient, which is the direction that effectively minimizes the cost function as quickly as possible. Thus at each iteration of the algorithm (**):

(1) The gradient of the cost function is taken

(2) A "step" is taken in the direction opposite of the gradient

(3) The weight parameters are updated via:

$$W_{i+1} = W_i - \eta * \nabla L_{CE}(\hat{y}, y) \tag{10}$$

Where:

$$L_{CE}(\hat{y}, y) = -[ylog\sigma(w \cdot x + b) + (1 - y)log(1 - \sigma(w \cdot x + b))] \tag{11}$$

η: a learning rate that modulates the step size at each iteration

Thus beginning with some initialized set of starting weights, we progressively move in a direction opposite to the gradient of the loss function until the 2-norm of the gradient is less than some tolerance close to zero. When the 2-norm is close to zero, the algorithm will have approached or nearly approached a local minimum (within some tolerance). In the case of logistic regression, the cost function being used is the cross entropy of the data which is the additive inverse of the log-likelihood (up to a constant). Additionally, cross-entropy as a loss function will be convex and any minimum reached ought to be the global minimum.

The Different Formulations
Batch, On-line SGD, and Mini-Batch SGD stochastic Gradient Descent may be understood as slight reformulations of the generalized Gradient Descent process.

Let N : the number of observations
Let M : the Batch Size, a hyper-parameter

While each formulation follows the algorithm (**) outlined above, the three different formulations differ based on the specified batch size. The data is partitioned into random subsets each of size equal to **N/M**, and a single subset is considered at each iteration of the algorithm. Note also that a cyclic rule [7] was implemented for selecting subsets, meaning that instead of randomly sampling with replacement, the random subsets were sequentially cycled through. The effective differences between these slight reformulations may be summarized as follows.

2.1.1 Batch Gradient Descent

If M = N then this corresponds to Batch Gradient Descent. Batch Gradient Descent computes the gradient of the loss function with respect to the entire data set at each iteration. It provides an exact gradient and also results in a smooth convergence of the cost function. (**see Appendix, figure 5.9**) The theoretical time complexity for each iteration update is O(np).[7]

2.1.2 On-Line Stochastic Gradient Descent

Then if M = 1 this corresponds to On-Line Stochastic Gradient Descent. Stochastic Gradient Descent (also referred to as On-Line SGD) randomly samples a single observation at each iteration and computes the gradient of the loss function with respect to this single observation. The gradient that is computed will be an approximation of the overall gradient and the stochasticity of this process results in a more volatile convergence of the cost function. (**see Appendix, figure 5.8**) The theoretical time complexity for each iteration update is O(p).[7]

2.1.3 Mini-Batch Gradient Descent

If M = 2,..., N-1 then this corresponds to Mini-Batch Gradient Descent. Note however that Mini-Batch Gradient Descent should have a batch size much smaller than N -1. I.e. M ¡¡ N Mini-Batch is a compromise between On-line and Batch Gradient Descent. Instead of considering the entire set of observations or only a single observation, the gradient of the cost function is computed for a smaller subset of random observations at each iteration. This results in a better approximation for the overall gradient than On-Line SGD, but depending on the batch size selected, the convergence of the cost function will vary in smoothness. (**see Appendix, figure 5.10**)
The theoretical time complexity for each iteration update is O(pn) where $p = N/M$.[7]

2.2 Newton's Method

To optimize logistic regression, we decided to use a more complex iterative method to find the global minima of our cost function. Newton's Method used a similar approach to Gradient Descent by using an algorithm similar to Taylor expansion where for each iteration of the weights of our original guess we reduced by multiplying the Hessian inverse which is the second derivative of our cost function, with the first derivative of the cost function known as the gradient. The Hessian is a matrix that contains all the values of the second derivative applied to our original dependent variable.

$$H(w^{(t)}) = d^2L(w^{(t)}) = \sum_{n=1}^{m} x_i^T x_i(\sigma(w^{(t)}) * (1 - \sigma(w^{(t)})) \tag{12}$$

What this means is that we do not need to set a learning rate for our function because the Hessian serves that purpose. The advantage of using the Hessian comes in the ability of the convergence to happen at a quadratic rate. Meaning that it would most likely be the fastest of our methods.

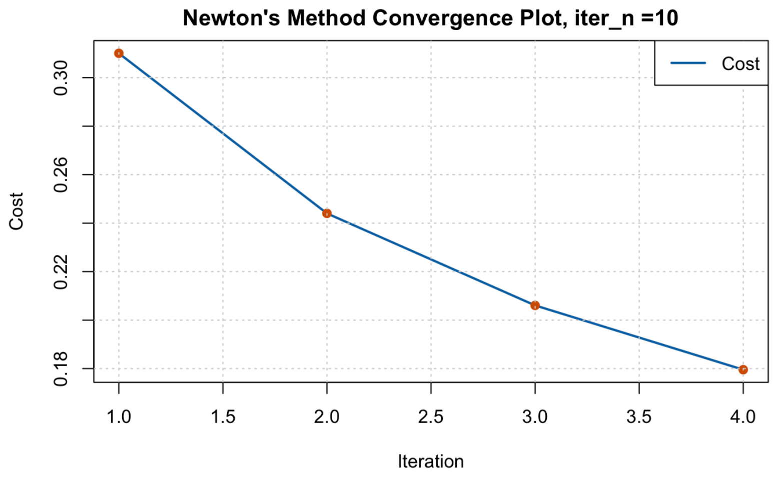

Figure 1: Newton's Method cost function

However, some disadvantages come in the fact that the second derivative reduces considerably each iteration matrix of the Hessian thus it would likely be close to zero that taking the inverse may be not possible. This is mostly due to a limitation of R. With the smallest number being 2.225074e-308 in R. Therefore the more interactions we have for our function the is more likely for the Hessian to reach this number. This most likely would cause the determinant to be zero which would make it impossible to compute the inverse of the Hessian.

We also want to think of the computational time complexity of taking the inverse of such a complex matrix. Taking the inverse of a matrix takes about $O(n^3)$ flops which are computationally expensive for larger datasets. Therefore, we wanted to think of an alternative to this issue. That is when we decide to also implement Cholesky decomposition. Instead of using the inverse of the Hessian itself, we end up taking the Cholesky decomposition of the Hessian matrix and calculating the inverse from that decomposition. Where taking the inverse of Cholesky decomposition gives us $O(n^3/3)$ flops which improve the speed slightly compared to inverting the Hessian itself.

3 Results – Time Analysis and Accuracy Analysis

In order to compare each of our methods to identify the most optimal one, we conducted several simulations to compare model accuracy and speed across methods. We did similar simulations with our

Mini-Batch Gradient Descent model as well to find the optimal batch size parameter, with learning rate and error tolerance parameters fixed.

Since our focus was developing logistic regression models instead of performing a data analysis study with a specific dataset, we decided that it was best to use randomly generated datasets. Using several different, randomly generated datasets allowed us to evaluate our model's performance on many different, large datasets. By making sure that it had utility on many different datasets, we were able to make sure that our model was not over-fitted to any specific datasets.

The datasets used for these simulations were randomly generated; we had 10,000 observations with 10 predictors and 10,000 observations of the corresponding response variable. The matrix of predictors was made up of floating point numbers between -10 and 10, randomly generated from the uniform distribution. The response vector consisted of 10,000 randomly generated 0s or 1s.

Each simulation involved calling the desired version of our logistic regression model-fitting function 100 times and recording either the model-fitting time or the prediction accuracy with a 75 percent/25 percent train-test split. Each iteration of a simulation involved generating a fresh dataset with the same parameters.

First, we fixed the learning rate and batch size and measured the speed and accuracy of our Mini-Batch logistic regression model to find the optimal batch size for a dataset with similar dimensions to ours.

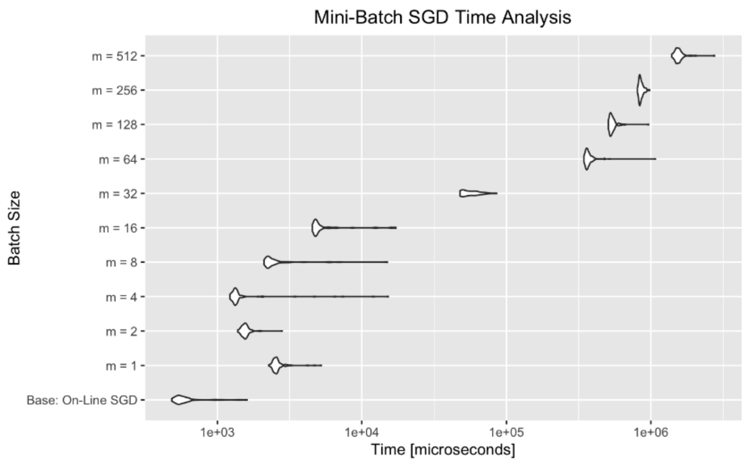

Figure 2: Computational speed for Mini-Batch SGD with different batch size

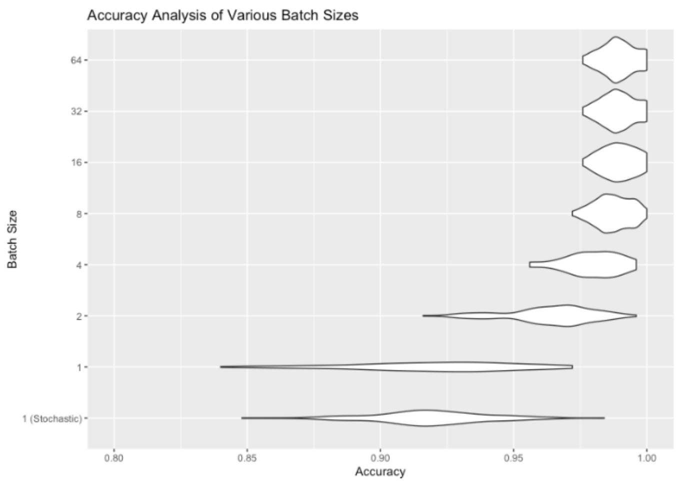

Figure 3: Accuracy for Mini-Batch SGD with different batch size

Figures 2 and 3 show how speed and accuracy respectively varied for our Mini-Batch logistic regression model. For a baseline comparison, both figures also include the results from our Stochastic Gradient Descent function at the bottom. Figure 2 shows a clear slowdown in computation time when batch size went above 16, with the biggest batch sizes of 256 and 512 being almost 2 magnitudes slower than the fastest batch sizes of 2 and 4. On the flip side, the accuracy plot in Figure 3 shows that when we increased batch size above 4, the accuracy wasn't improving at all because it was already nearly 100 percent at that point.

Seeing how smaller batch sizes are faster but larger ones are more accurate, the key takeaway from both of these plots is that we need to select a batch size that sufficiently balances speed and accuracy. For us, that batch size was 4. It was faster on average than any other batch size we tested and only slightly less accurate than the larger batch sizes. The simulations showed that a batch size of 4 provided the perfect balance between speed and accuracy for data with the dimensions ours had. For our other simulations to compare Mini-Batch Gradient Descent to other models, we decided to fix batch size at 4.

For someone using Mini-Batch Gradient Descent to fit a logistic regression model, it is advisable to conduct similar simulations to determine an optimal batch size for their data before fitting the model.

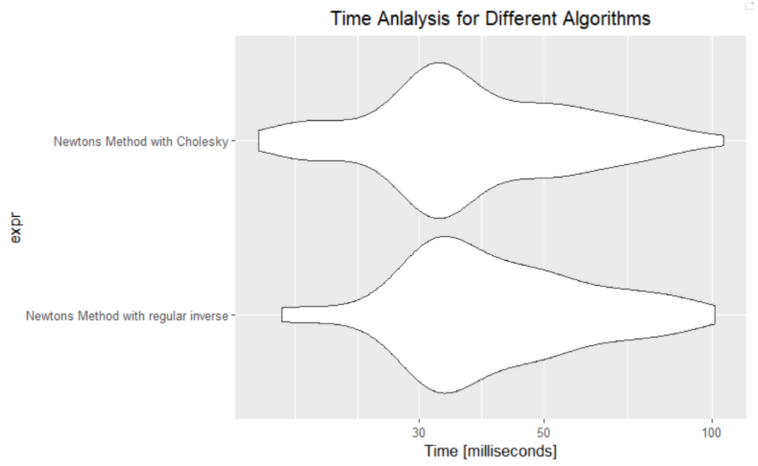

Figure 4: speed comparison with or without Cholesky for Newton's Method

Figure 4 shows that the method where we inverted the Hessian itself is slightly slower than the one where we used Cholesky decomposition on the Hessian before taking the inverse. The median time for the model with Cholesky decomposition was 41.08516 milliseconds, while the median time for the model with the regular inverse was 44.34981 milliseconds. However, the results for accuracy were similar between the two. Therefore, we stuck with the Cholesky method moving forward for comparison with the other methods.

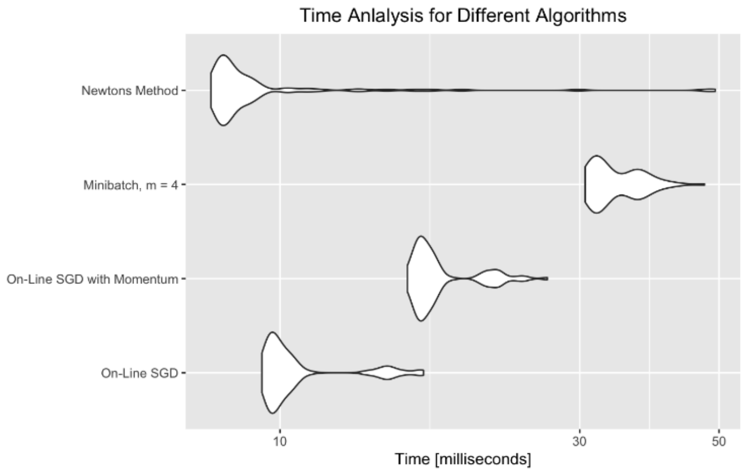

Figure 5

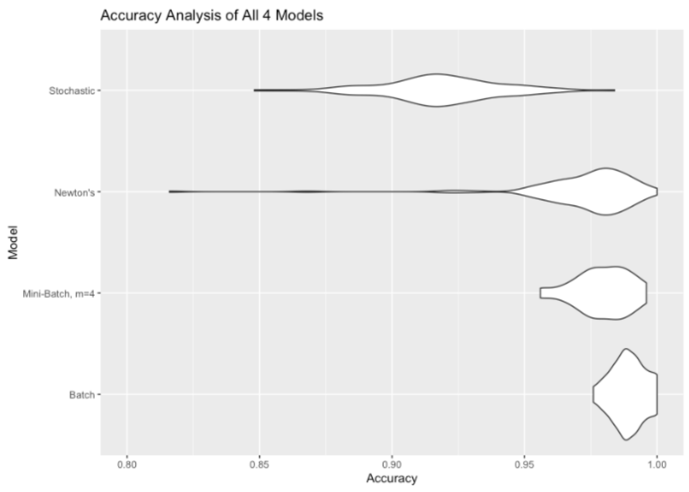

Figure 6

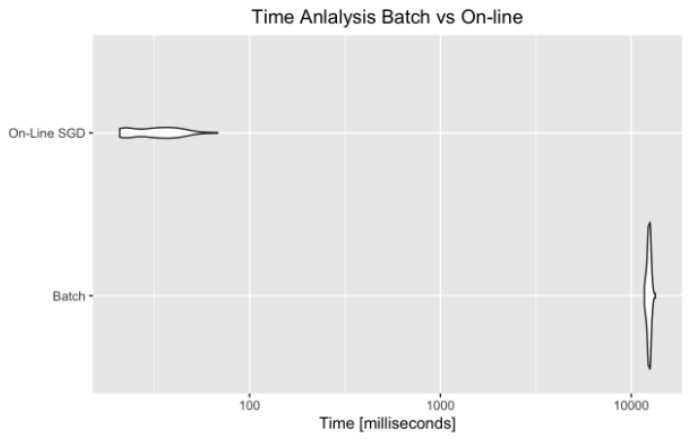

Figure 7

Figures 5 and 6 show the computation speed and accuracy of our various logistic regression models compared to one another. In figure 5, what stood out was how volatile Newton's Method ended up being. While the spread of computation time of the other three models was fairly compact, Newton's Method ended up being all over the place. Even though the majority of its 100 iterations had a fast computation time, the violin for Newton's Method had a long tail to its right, indicating that there were a handful of instances where it was extremely slow. When it came to accuracy as shown in Figure 5, Newton's Method behaved similarly. It was reasonably accurate most of the time, but it had a long tail to the left showing occasional instances of poor model accuracy. These two plots show that Newton's Method usually performed decently enough, but it was unpredictably either slow, inaccurate, or both, and this volatility/unpredictability showed us that it was too inconsistent to trust.
The other pattern to note from Figure 6 is the accuracy performance of Mini-Batch Gradient Descent compared to Full Batch Gradient Descent. Full Batch Gradient Descent was our most accurate model, but Mini-Batch came very close to matching its accuracy. On the other hand, Full Batch Gradient Descent was many magnitudes slower than Mini-Batch was. Figure 7 shows that the Full Batch model clocked in with a median computation time of over 10,000 milliseconds. Compare this to the Mini-Batch model's median computation time of about 35 milliseconds as shown in Figure 5. The miniscule (roughly 2.5 percent) increase in accuracy by going from the Mini to the Full Batch model is not worth increasing the computational cost by a factor of 285!

4 Conclusion

After careful evaluation of various optimization algorithms commonly used in machine learning, it has been observed that Newton's Method stands out as the fastest algorithm. However, its high volatility and tendency to diverge sometimes hinder its reliability and robustness. In contrast, Gradient Descent is a relatively consistent and reliable optimization technique, while slower than Newton's Method. Further analysis revealed that the accuracy of the optimization algorithms could vary significantly, with Full Batch and Mini-Batch Gradient Descent delivering the most precise results. On the other hand, Newton's Method and Stochastic Gradient Descent demonstrated comparatively lower accuracy. After weighing the pros and cons of each optimization algorithm, it is concluded that Mini-Batch Gradient Descent is the optimal choice. This method strikes an excellent balance between speed, consistency, and accuracy, making it an ideal optimization technique for most machine-learning applications. In conclusion, the Mini-Batch Gradient Descent algorithm can be considered as the preferred optimization algorithm due to its superior performance across multiple dimensions.

5 Appendix

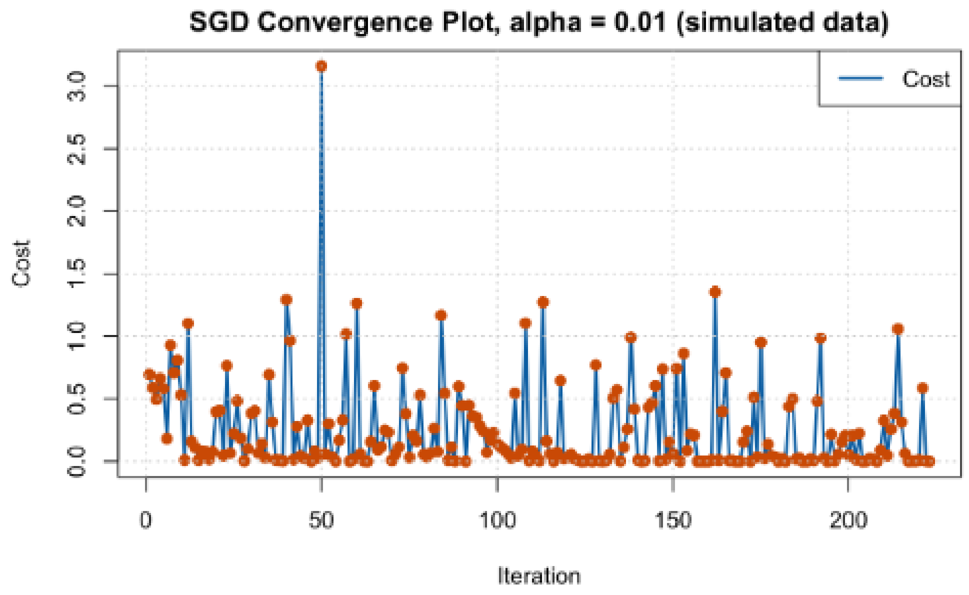

Figure 5.8

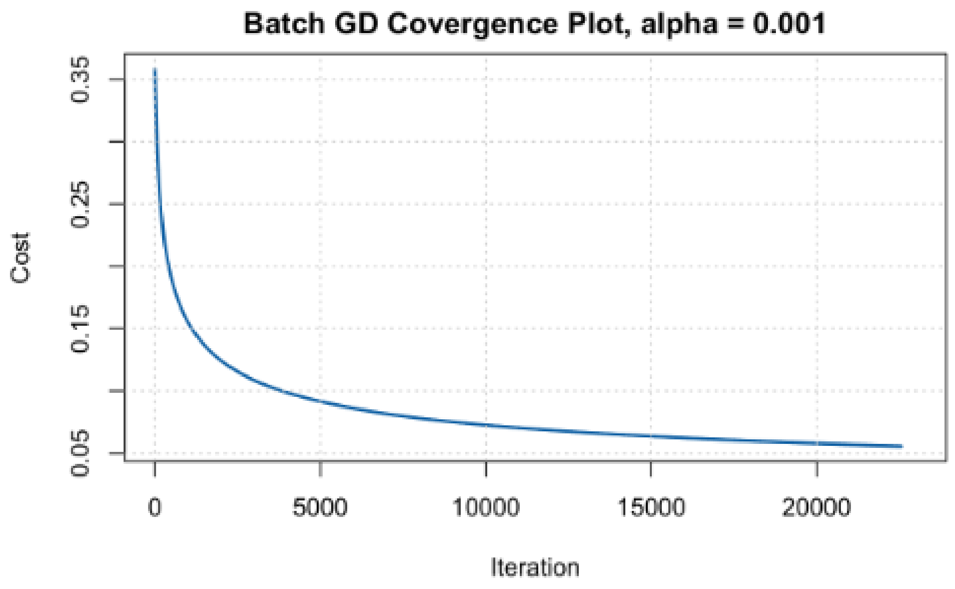

Figure 5.9

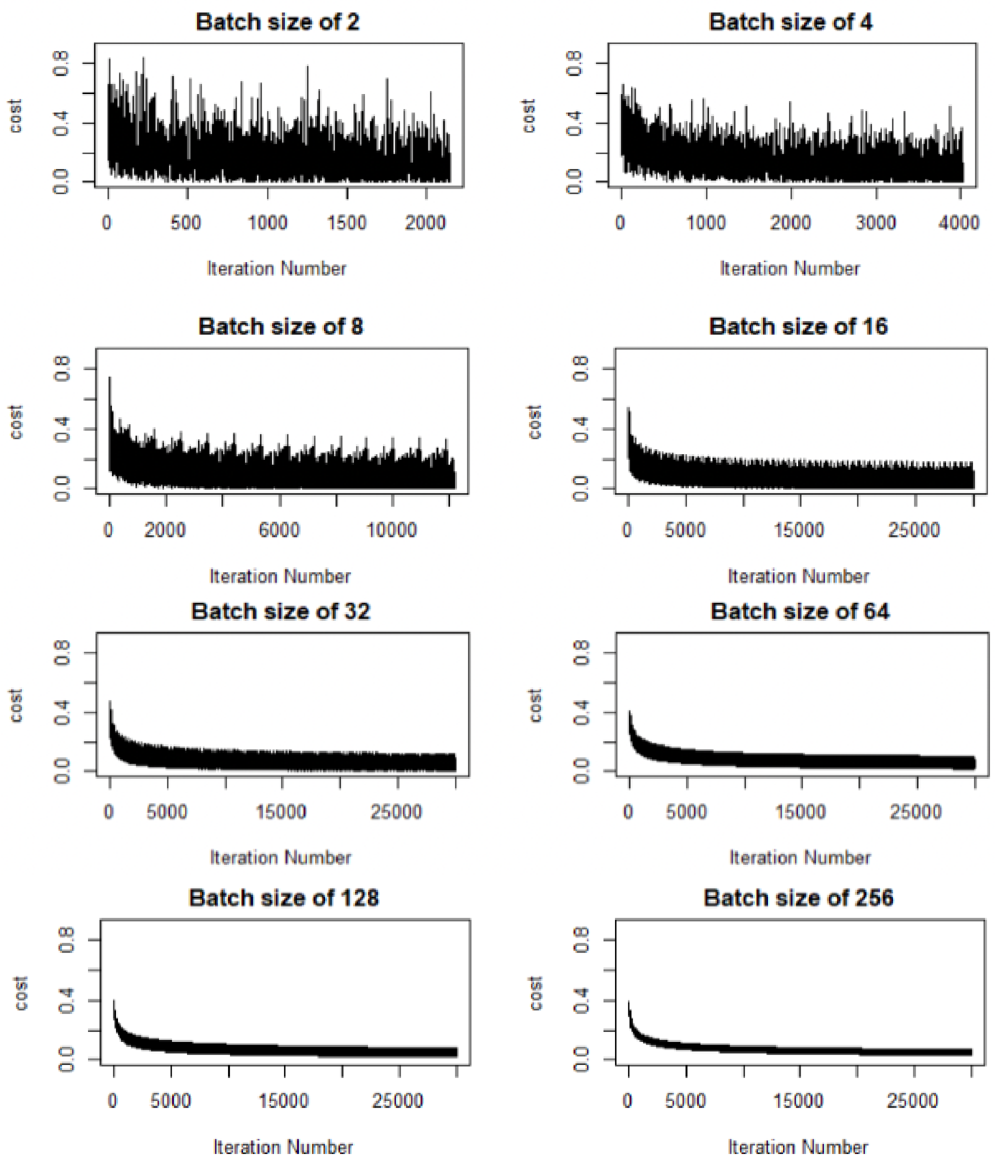
Batch size of 2
Batch size of 4
Batch size of 8
Batch size of 16
Batch size of 32
Batch size of 64
Batch size of 128
Batch size of 256
cost
Iteration Number

Figure 5.10

6 Functions

Derivation for Gradient Descent cost function[9]:

Derivative of gradient for one datapoint $(\mathbf{x}, y)$:

$$\begin{aligned}
\frac{\partial LL(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} y \log \sigma(\theta^T \mathbf{x}) + \frac{\partial}{\partial \theta_j}(1-y)\log[1-\sigma(\theta^T \mathbf{x}] \\
&= \left[\frac{y}{\sigma(\theta^T x)} - \frac{1-y}{1-\sigma(\theta^T x)}\right] \frac{\partial}{\partial \theta_j} \sigma(\theta^T x) \\
&= \left[\frac{y}{\sigma(\theta^T x)} - \frac{1-y}{1-\sigma(\theta^T x)}\right] \sigma(\theta^T x)[1-\sigma(\theta^T x)] x_j \\
&= \left[\frac{y-\sigma(\theta^T x)}{\sigma(\theta^T x)[1-\sigma(\theta^T x)]}\right] \sigma(\theta^T x)[1-\sigma(\theta^T x)] x_j \\
&= \left[y-\sigma(\theta^T x)\right] x_j
\end{aligned}$$

Derivation for Hessian of cost function[5]:

$$\frac{dLL(\theta)}{d\theta_j} = yx_j - \sigma(\theta^T x)x_j$$

$$\frac{d^2 LL(\theta)}{d\theta_j \theta_j^T} = x_i^T x_i(\sigma(\theta^T x) * (1-\sigma(\theta^T x))$$

7 Reference

[1] Shalizi, Cosma. Logistic Regression - Carnegie Mellon University. https://www.stat.cmu.edu/ cshalizi/uADA/12/lectures/ch12.pdf.

[2] "Speech and Language Processing (3rd Ed. Draft) Dan Jurafsky and James H. Martin." Speech and Language Processing, https://web.stanford.edu/ jurafsky/slp3/.

[3] Stanford University. https://web.stanford.edu/class/archive/cs/cs109/cs109.1176/lectures/23-LogisticRegression.pdf.

[4] Harrell, Frank. "Classification vs. Prediction." Statistical Thinking, 15 Jan. 2017, https://www.fharrell.com/post/classification/.

[5] Morales, Manuel, and Manuel MoralesManuel Morales 1. "Hessian of Logistic Function." Cross Validated, 1 June 1960, https://stats.stackexchange.com/questions/68391/hessian-of-logistic-function.

[6] Srihari, Sargur N. Deep Learning, https://cedar.buffalo.edu/ srihari/CSE676/.

[7] Tibshirani, Ryan J. Stochastic Gradient Descent - Carnegie Mellon University. https://www.stat.cmu.edu/ ryantibs/convexopt/lectures/stochastic-gd.pdf.

[8] 3.1 Taylor Series Approximation - Princeton University. https://www.cs.princeton.edu/courses/archive/fall18/cos597G/lecnotes/lecture3.pdf.

[9] Logisticregression - Stanford University. https://web.stanford.edu/class/archive/cs/cs109/cs109.1178/lectureHandouts/ 220-logistic-regression.pdf.

[10] Agresti, Alan. Categorical Data Analysis.

8 Code Appendix - see next page

Appendix – Code

```r
knitr::opts_chunk$set(echo = TRUE, eval = FALSE)

functions = list.files('~/Documents/UCD/STA141c/Project/functions', full.names = TRUE)



functions


sapply(functions, source) # import all necessary functions


#sigmoid function

sigmoid = function(z){

  #Logistic Function
  #applies logistic function to


  #if W,X specified:
  # z = crossprod(xW)


  sig = 1/(1 + exp(-z))


  #W,X specified


  return(sig)

}


#function to simulate the data; taken from Eungsongs Discussion Notes

sim_data = function(p = 10 , n = 1000, s = 6){
  #Function that simulates logistic data

  # n; integer -- number of observations
  # p; integer -- number of covariates

  beta = seq(1,p) #number of covariates

  sigma_sq = s #set variance
  X_sim = matrix(runif(n*p, -10,10), n, p) # sim data

  y_sim = as.vector(X_sim %*% beta + rnorm(n, mean = 0, sd = sqrt(sigma_sq))) #sim response

  y_sim = sapply(y_sim, sigmoid) # apply sigmoid to responses

  y_sim = ifelse(y_sim  > 0.5, 1, 0) # make response binary

  return(list(X_sim, y_sim))
}






#creates confusion matrix and treturns accuracy

conf_matrix = function(w,A,y){

  # w; vector of weights
  # A; a data matrix
  # y: vector of true labels

  temp = A %*% w # applies weights to data matrix

  temp_pred = 1/(1 + exp(-temp)) # apply logistic function to get probabilities P(y|X,w)

  temp_pred_labels = ifelse(temp_pred >  0.5, 1, 0) # decision rule P(y|x) > 0.5 --> 1

  accuracy = sum(temp_pred_labels == y)/length(y) # how many predicted labels match true labels?

  confusion_matrix = table(temp_pred_labels, y) # matrix comparing pred labels to true
  # labels; sum of diagonal = accuracy

  return(list(accuracy, confusion_matrix))


}






#Batch Gradient Descent Algorithm

log_regr = function(X,y, tol = 0.001, a = 0.1, plotcost = FALSE ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
```

```r
#tol, a stopping criterion


# Function performs gradient descent algorithm to obtain the
# vector of weights w* that minimize the log likelihood
# function of the data (gives MLE)

# Note: MLE for logistic regression doesnt have a closed form solution.

m = nrow(X) # number of observations

d = ncol(X) # dimension of data; number of variables

w_cur = rep(1/d,d) # initialize weight vector

error = 10 # initialize error

tol = tol # tolerance

max_iter = 30000 # max iterations

step = 0 # iteration counter

b = 0 # bias term

a = a # learning rate

X = as.matrix(X)

y = as.matrix(y)

cost_vector = c()

while ((error > tol) && (step < max_iter)){

  # prev = cur
  #
  # cur = prev - a * grad((Loss(X,w,y)))
  #
  #
  w_prev = w_cur


  yhat = 1/(1 + exp(-(X%*%w_prev + b)))



  g =  (1/m) * t(yhat - y) %*% X

  w_cur = w_prev - a * t(g)


  error = as.double(sqrt(crossprod(t(g))))
```

```r
    step = step + 1

    # cost = -sum((y[cur_i,] * log(yhat[1,1])) + ((1-y[cur_i,]) * log(1-yhat[1,1])))/m
    #
    # cost_vector = c(cost_vector, cost)

    cost = -sum((t(y) %*% log(yhat)) + (t(1-y) %*% log(1-yhat)))/m
    cost_vector = c(cost_vector, cost)




  }
  print(step)

  return(list(w_cur, error, g, yhat, cost_vector))

}








# Performs On-Line Stochastic Gradient Descent

log_regr_stoch2 = function(X,y, tol = 0.001, a = .1){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion


  # Function performs gradient descent algorithm to obtain the
  # vector of weights w* that minimize the log likelihood
  # function of the data (gives MLE)

  # Note: MLE for logistic regression doesnt have a closed form solution.

  m = nrow(X) # number of observations

  d = ncol(X) # dimension of data; number of variables

  w_cur = rep(1/d,d) # initialize weight vector

  error = 10 # initialize error

  tol = tol # tolerance

  max_iter = 30000 # max iterations
  step = 0 # iteration counter

  b = 0 # bias term

  a = a # learning rate

  X = as.matrix(X)

  y = as.matrix(y)

  #set.seed(0)

  i_rand = sample(0:m, m, replace = FALSE)

  while ((error > tol) && (step < max_iter)){


    w_prev = w_cur

    cur_i = i_rand[step + 1%%m]

    yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b)))


    g =   t(yhat - y[cur_i,]) %*% X[cur_i,]

    w_cur = w_prev - a * t(g)

    error = as.double(sqrt(crossprod(t(g))))

    step = step + 1


  }
  print(step)

  return(list(w_cur, error, g, yhat))

}




#performs mini Batch gradient descent algorithm

log_regr_minibatch = function(X,y, tol = 0.001, a = .1, batch_size ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
```

```
#y, a vector of labels (each row an observation)
#m, a batch size (1, minibatch, n)
#a, a specified learning rate
#tol, a stopping criterion


# Function performs gradient descent algorithm to obtain the
# vector of weights w* that minimize the log likelihood
# function of the data (gives MLE)

# Note: MLE for logistic regression doesnt have a closed form solution.

m = nrow(X) # number of observations

d = ncol(X) # dimension of data; number of variables

w_cur = rep(1/d,d) # initialize weight vector

error = 10 # initialize error

tol = tol # tolerance



b = 0 # bias term

a = a # learning rate

X = as.matrix(X)

y = as.matrix(y)

set.seed(0)

n_mini_batches = m %/% batch_size # number of batches

i_rand = sample(1:m, m, replace = FALSE) # randomly permute observations



batches = split(i_rand, factor(1:n_mini_batches)) # partition into batches
max_iter = 30000 # max iterations
step = 0 # iteration counter






while ((error > tol) && (step < max_iter)){
    w_prev = w_cur #previous becomes current

    cur_i = batches[[(step %% n_mini_batches) + 1]] #select next batch

    yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b))) # pred


    g =   (1/batch_size) * t(yhat - y[cur_i,]) %*% X[cur_i,] #gradient

    w_cur = w_prev - a * t(g) # update weights


    error = as.double(sqrt(crossprod(t(g)))) #check error

    step = step + 1



  }




  return(list(w_cur, error, g, yhat))

}





# performs mini-batch gradient desent and keeps track of cost


log_regr_minibatch_cost = function(X,y, tol = 0.001, a = .1, batch_size, max_iter = 30000 ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion


  # Function performs gradient descent algorithm to obtain the
  # vector of weights w* that minimize the log likelihood
  # function of the data (gives MLE)

  # Note: MLE for logistic regression doesnt have a closed form solution.

  m = nrow(X) # number of observations

  d = ncol(X) # dimension of data; number of variables

  w_cur = rep(1/d,d) # initialize weight vector
```

```r
error = 10 # initialize error

tol = tol # tolerance


b = 0 # bias term

a = a # learning rate

X = as.matrix(X) # coerce to matrix

y = as.matrix(y) # coerce to matrix

set.seed(0) # set random seed


n_mini_batches = m %/% batch_size # calculate number of batches

i_rand = sample(1:m, m, replace = FALSE) # cyclic randomization


batches = split(i_rand, factor(1:n_mini_batches)) #partition data into subsets
max_iter = max_iter # max iterations
step = 0 # iteration counter

cost_vector = vector('numeric', length = max_iter) #initialize cost vector



while ((error > tol) && (step < max_iter)){


  w_prev = w_cur #weights from last iteration become previous weights

  cur_i = batches[[(step %% n_mini_batches) + 1]] #which batch to use

  yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b))) #prediction


  g =   (1/batch_size) * t(yhat - y[cur_i,]) %*% X[cur_i,] # gradient

  w_cur = w_prev - a * t(g) # update weights

  cost_vector[step+1] = -sum((y[cur_i,] * log(yhat)) + ((1-y[cur_i,]) * log(1-yhat)))/batch_size # ap


  error = as.double(sqrt(crossprod(t(g)))) # calculate error
```

```
    step = step + 1 #increment steps


    #add a break statement
    # print(w_cur)


  }






  return(list(w_cur, error, g, yhat, cost_vector))

}







#Newtons Method

newtons_method_chol = function(X,y,W,m,iter_n =10){
  # sigmoid function

  # sigmoid function
  sigmoid  = function (X,W) {
    z <- X %*% W
    sig <- 1 / (1 + exp(-z) )
    return (sig)
  }


  cost = function (X,y,W,m) {
    # cost function

    cost <- sum(-y * log(sigmoid(X,W)) - (1 - y) * log(1 - sigmoid(X,W)))
    return( cost/m)
  }


  grad = function (X,y,W,m) {
    #gradient or first derivative of the cost function

    A <- sigmoid(X,W) - y
    return( crossprod(X,A)/m)
  }


  Hessian = function (X,W,m) {
    # Hessian
    #chol <- chol(t(X) %*% X )
    h_c <- diag(sigmoid(X,W)) *(1- diag((sigmoid(X,W))))
    hess <- (t(X)%*%X)* h_c
    #hess <- X%*%((diag(sigmoid(X,W)) - diag((sigmoid(X,W))^2))*t(X))
    return (hess/m)
  }

  cost_vector = c()
  i = 0
  while(i < iter_n) {

    chol <- chol(Hessian(X,W,m))
    inv_chol = chol2inv(chol)
    W = W - inv_chol%*% grad(X,y,W,m)
    cur_cost = cost(X,y,W,m)
    cost_vector = c(cost_vector, cur_cost)

    if(sqrt(crossprod(grad(X,y,W,m))) < 0.0001){break}
    i = i+1

  }
  return(list(W, cost_vector))
}






temp = sim_data(n = 10000) # simulate a data set with 10,000 observations

X_sim = temp[[1]] # Pull out the covariates

y_sim = temp[[2]] # Pull out the response


# temp2 = read.csv('~/Documents/Data/STA141c/heartdata.csv')
#
#
# X_heart = temp2[,1:14]
# y_heart = temp2[,15]





testtrainsep = function(A,b){ # A is a data matrix, # b is vector of labels
  # Function that splits data into 75 - 25 train/test split

  n = nrow(X_sim)

  c = n%/%(4/3)

  #Set the training data set
  X_train = A[1:c,]
  y_train = b[1:c]

  #Set the test data set
  X_test = A[(c+1):n,]
  y_test = b[(c+1):n]

  #Make list that contains the test and train data
  traintest = list(X_train = X_train, y_train = y_train,
                   X_test = X_test, y_test  = y_test)

  return(traintest)
}



data1 = testtrainsep(X_sim, y_sim) #train/test data

X_sim_train = data1$X_train # X train
y_sim_train = data1$y_train # y train

X_sim_test = data1$X_test # X test
y_sim_test = data1$y_test # y test







test1 = log_regr(X_sim_train, y_sim_train, a = 0.1)

#apply batch Gradient Descent algorithm to train

conf_matrix(test1[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions


test2 = log_regr_stoch2(X_sim_train, y_sim_train, a = 0.0001, tol = 0.000001)

#apply stochastic gradient descent algorithm

conf_matrix(test2[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions


test3 = log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3)

#apply SGD with momentum algorithm
```

```r
conf_matrix(test3[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions




test4 = log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4, a = 0.001)

#apply mini batch SGD to data

conf_matrix(test4[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions



z = newtons_method_chol(X_sim_train, y_sim_train, rep(0,10), m = 7500, iter_n = 5)


#apply newton's method to data

conf_matrix(z[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions

sgd_timings = microbenchmark::microbenchmark(
  log_regr(X_sim_train, y_sim_train, a = 0.1),
  log_regr_stoch2(X_sim_train, y_sim_train, a = 0.001, tol = 0.000001)#,
#  log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3)

)

#compute benchmarks for batch and on-line stochastic gradient descent

sgd_timings
xlabs = c( 'Batch', 'On-Line SGD')
ggplot2::autoplot(sgd_timings) + scale_x_discrete(labels = xlabs) + ggtitle('Time Anlalysis Batch vs On

#Plot the timings

sgd_timings2 = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train, y_sim_train, a = 0.001, tol = 0.0001),
  log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4, a = 0.001),
  newtons_method_chol(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4)



)

#compute timings for all methods
```

```
library(ggplot2)
sgd_timings2

xlabs = c('On-Line SGD', 'On-Line SGD with Momentum', 'Minibatch, m = 4', 'Newtons Method')

ggplot2::autoplot(sgd_timings2) + scale_x_discrete(labels = xlabs) + ggtitle('Time Anlalysis for Differe


#plot the timings


batch_timings = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train,y_sim_train),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 1),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 2),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 8),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 16),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 32),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 64)#,
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 128),
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 256),
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 512)
)



#compute timings for mini-batch of different batch sizes



library(ggplot2)


batch_timings

labels = c('Base: On-Line SGD','m = 1','m = 2','m = 4','m = 8','m = 16','m = 32', 'm = 64')
autoplot(batch_timings) + scale_x_discrete(labels = labels) +ggtitle('Time Anlalysis for Variable Batch

#plot the timings



mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 8)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l',  ylim = c(0,0.8))

# mb4_avg = ifelse(is.na(mb4_cost), 0, mb4_cost)
#
```

```r
# mb4_avg = split(mb4_avg, 1:5000)
#
#
# mb4_avg = sapply(mb4_avg, mean)

plot(1:length(mb4_cost), mb4_cost, 'l')





batch_timings = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train,y_sim_train),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 1),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 2),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 8),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 16),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 32),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 64),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 128),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 256),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 512)
)








batch_timings
#ggplot2::autoplot(batch_timings)

labels = c('Base: On-Line SGD','m = 1','m = 2','m = 4','m = 8','m = 16','m = 32', 'm = 64', 'm = 128',
autoplot(batch_timings) + scale_x_discrete(labels = labels) +ggtitle('Mini-Batch SGD Time Analysis') +





mom_cost = log_regr_stoch_mom_cost(X_sim_train, y_sim_train, nu = 0.001, alpha = 0.3)

mom_cost = mom_cost[[5]]

mom_cost = mom_cost[mom_cost != 0]

plot(1:length(mom_cost), mom_cost, 'l')



#compute cost over time of momentum
```

```
mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 7500)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l', col = '#0072B2', lwd = 2, lty = 'solid', xlim = c(0,23000), xlal

title(main = 'Batch GD Covergence Plot, alpha = 0.001')

#points(mb4_cost, col = '#D55E00', pch = 19)

grid(lty = 'dotted')


#compute minibatch cost over time



mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 1)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l')




#compute minibtach over time and plot

# Newton's Method without Cholesky


newtons_method= function(X,y,W,m,iter_n =10){
  # sigmoid function

  # sigmoid function
  sigmoid  = function (X,W) {
    z <- X %*% W
    sig <- 1 / (1 + exp(-z) )
    return (sig)
  }


  cost = function (X,y,W,m) {
    # cost function

    cost <- sum(-y * log(sigmoid(X,W)) - (1 - y) * log(1 - sigmoid(X,W)))
    return( cost/m)
  }
  grad = function (X,y,W,m) {
    #gradient or first derivative of the cost function

    A <- sigmoid(X,W) - y
    return( crossprod(X,A)/m)
  }


  Hessian = function (X,W,m) {
    # Hessian

    #chol <- chol(t(X) %*% X )
    h_c <- diag(sigmoid(X,W)) *(1- diag((sigmoid(X,W))))
    hess <- (t(X)%*%X)* h_c
    #hess <- X%*%((diag(sigmoid(X,W)) - diag((sigmoid(X,W))^2))*t(X))
    return (hess/m)
  }

  cost_vector = c()
  i = 0
  while(i < iter_n) {

    #chol <- chol(Hessian(X,W,m))
    #inv_chol = chol2inv(chol)
    W = W - solve(Hessian(X,W,m))%*% grad(X,y,W,m)
    cur_cost = cost(X,y,W,m)
    cost_vector = c(cost_vector, cur_cost)

    if(sqrt(crossprod(grad(X,y,W,m))) < 0.0001){break}
    i = i+1

  }
  return(list(W, cost_vector))
}





#benchmarking newton method algorithms (cholesky and no cholesky)

sgd_timings2 = microbenchmark::microbenchmark(
newtons_method(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4),
  newtons_method_chol(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4)



)
```

```r
data=as.list(numeric(100))
for (i in 1:100) {
  data[[i]]=sim_data()
}
newton_accuracies=numeric(100)

#100 sims for Newton's Method accuracies
for (i in 1:100) {

  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]

  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=newtons_method_chol(X_sim_train, y_sim_train, rep(0, ncol(X_sim_train)), ncol(X_sim_train), i


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  newton_accuracies[i]=z/length(y_sim_test)
}
df=data.frame(y=y, method="Newton")

#5 number summary for Newton's Method accuracy
y=c(min(newton_accuracies), quantile(newton_accuracies)[2], mean(newton_accuracies), quantile(newton_ac

mb_accuracies=as.list(numeric(7))
a=c(1, 2, 4, 8, 16, 32, 64)

#simulations of mini batch accuracy with various batch sizes
for (i in c(1, 2, 4, 8, 16, 32, 64)) {
  results=numeric(100)
  for (j in 1:100) {


    X_sim=data[[j]][[1]]
    y_sim=data[[j]][[2]]

    X_sim_train = X_sim[1:750,]
    X_sim_test = X_sim[751:1000,]

    y_sim_train = y_sim[1:750]
    y_sim_test = y_sim[751:1000]

    ret=log_regr_minibatch(X_sim_train, y_sim_train, batch_size=i)
    weight=ret[[1]]
    test_pred = X_sim_test %*% weight
    test_pred = 1/(1 + exp(-test_pred))
    test_labels = ifelse(test_pred > 0.5, 1, 0)

    z=sum(test_labels == y_sim_test)
    results[j]=z/length(y_sim_test)
  }
  mb_accuracies[[(log(i, base=2))+1]]=results
}

#list of accuracies across bacth sizes
mb_accuracies

d=data.frame(unlist(mb_accuracies))
d$batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64, 
d[701:800,]=stochcopy
d$batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64, 
d$batch_size=as.factor(d$batch_size)
names(d)=c("Accuracy", "Batch_size")
d=d[c(701:800, 1:700),]

one=d[101:200,]
two=d[201:300,]
four=d[301:400,]
eight=d[401:500,]
sixteen=d[501:600,]
tt=d[601:700,]
sf=d[701:800,]

#mini-batch 5 number summary
fivensumsmb=data.frame(quantile(one$Accuracy), quantile(two$Accuracy), quantile(four$Accuracy), quantil
fivensumsmb[6,]=means
row.names(fivensumsmb)=c("Min", "LQ", "Median", "UQ", "Max", "Mean")
names(fivensumsmb)=c("m=1", "m=2", "m=4", "m=8", "m=16", "m=32", "m=64", "Stochastic")
fivensumsmb=t(fivensumsmb)
fivensumsmb=fivensumsmb[,c(1,2,6,3,4,5)]
fivensumsmb=fivensumsmb[c(7, 6, 5, 4, 3, 2, 1, 8),]

stochcopy=stoch
names(stochcopy)=c("Accuracy", "Batch_size")
d2=d
d2[701:800,]=stochcopy
d2$Batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64,
d2$Batch_size=as.factor(d2$Batch_size)

#plot of all batch sizes' accuracies
ggplot(d, aes(x=factor(Batch_size, level=c("1 (Stochastic)", 1, 2, 4, 8, 16, 32, 64)), y=Accuracy))+geon
  coord_flip()+ggtitle("Accuracy Analysis of Various Batch Sizes")+xlab("Batch Size")


#list to store accuracies of the 4 models
models_accuracies=as.list(numeric(4))
#simulating batch logistic regression
```

```
base_accuracies=numeric(100)
for (i in 1:100) {

  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]

  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=log_regr(X_sim_train, y_sim_train)[[1]]


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  base_accuracies[i]=z/length(y_sim_test)
}
models_accuracies[[1]]=base_accuracies
models_accuracies[[2]]=newton_accuracies
#mb_accuracies[[3]] is the accuracies with batch size=4
models_accuracies[[3]]=mb_accuracies[[3]]

stoch_accuracies=numeric(100)
#simulation of stochastic logistic regression
for (i in 1:100) {

  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]

  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=log_regr_stoch2(X_sim_train, y_sim_train)[[1]]


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  stoch_accuracies[i]=z/length(y_sim_test)
}
models_accuracies[[4]]=stoch_accuracies
df=data.frame(unlist(models_accuracies))
df$model=c(rep("Batch", 100), rep("Newton's", 100), rep("Mini-Batch, m=4", 100), rep("Stochastic", 100)
df$model=as.factor(df$model)
names(df)=c("Accuracy", "Model")

#Plot of 4 models accuracies
ggplot(df, aes(x=Model, y=Accuracy))+geom_violin()+ylim(c(0.8, 1))+coord_flip()+ggtitle("Accuracy Analy

batch=df[1:100,]
newton=df[101:200,]
mb=df[201:300,]
stoch=df[301:400,]

means=c(mean(batch$Accuracy, mean(newton$Accuracy), mean(mb$Accuracy), mean(stoch$Accuracy)))

#5 number summaries for all 4 models
fivensums=data.frame(quantile(batch$Accuracy), quantile(newton$Accuracy), quantile(mb$Accuracy), quanti
fivensums[6,]=means
row.names(fivensums)=c("Min", "LQ", "Median", "UQ", "Max", "Mean")
names(fivensums)=c("Batch", "Newton's", "Mini-Batch, m=4", "Stochastic")
fivensums=t(fivensums)
fivensums=fivensums[,c(1,2,6,3,4,5)]
fivensums=fivensums[c(4, 2, 3, 1),]
```

```
functions = list.files('~/Documents/UCD/STA141c/Project/functions', full.names = TRUE)



functions
```

```
sapply(functions, source) # import all necessary functions
```

```
#sigmoid function

sigmoid = function(z){

  #Logistic Function
  #applies logistic function to


  #if W,X specified:
  # z = crossprod(xW)



  sig = 1/(1 + exp(-z))



  #W,X specified



  return(sig)

}
```

```
#function to simulate the data; taken from Eungsongs Discussion Notes

sim_data = function(p = 10 , n = 1000, s = 6){
  #Function that simulates logistic data

  # n; integer -- number of observations
  # p; integer -- number of covariates

  beta = seq(1,p) #number of covariates

  sigma_sq = s #set variance

  X_sim = matrix(runif(n*p, -10,10), n, p) # sim data

  y_sim = as.vector(X_sim %*% beta + rnorm(n, mean = 0, sd = sqrt(sigma_sq))) #sim response

  y_sim = sapply(y_sim, sigmoid) # apply sigmoid to responses

  y_sim = ifelse(y_sim  > 0.5, 1, 0) # make response binary

  return(list(X_sim, y_sim))
}
```

```
#creates confusion matrix and treturns accuracy

conf_matrix = function(w,A,y){

  # w; vector of weights
  # A; a data matrix
  # y: vector of true labels

  temp = A %*% w # applies weights to data matrix

  temp_pred = 1/(1 + exp(-temp)) # apply logistic function to get probabilities P(y|X,w)

  temp_pred_labels = ifelse(temp_pred >  0.5, 1, 0) # decision rule P(y|x) > 0.5 --> 1

  accuracy = sum(temp_pred_labels == y)/length(y) # how many predicted labels match true labels?

  confusion_matrix = table(temp_pred_labels, y) # matrix comparing pred labels to true
  # labels; sum of diagonal = accuracy

  return(list(accuracy, confusion_matrix))


}
```

```
#Batch Gradient Descent Algorithm

log_regr = function(X,y, tol = 0.001, a = 0.1, plotcost = FALSE ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion
```

```r
# Function performs gradient descent algorithm to obtain the
# vector of weights w* that minimize the log likelihood
# function of the data (gives MLE)

# Note: MLE for logistic regression doesnt have a closed form solution.

m = nrow(X) # number of observations

d = ncol(X) # dimension of data; number of variables

w_cur = rep(1/d,d) # initialize weight vector

error = 10 # initialize error

tol = tol # tolerance

max_iter = 30000 # max iterations

step = 0 # iteration counter

b = 0 # bias term

a = a # learning rate

X = as.matrix(X)

y = as.matrix(y)

cost_vector = c()

while ((error > tol) && (step < max_iter)){

  # prev = cur
  #
  # cur = prev - a * grad((Loss(X,w,y)))
  #
  #
  w_prev = w_cur


  yhat = 1/(1 + exp(-(X%*%w_prev + b)))



  g =  (1/m) * t(yhat - y) %*% X

  w_cur = w_prev - a * t(g)


  error = as.double(sqrt(crossprod(t(g))))

  step = step + 1
```

```
    # cost = -sum((y[cur_i,] * log(yhat[1,1])) + ((1-y[cur_i,]) * log(1-yhat[1,1])))/m
    #
    # cost_vector = c(cost_vector, cost)

    cost = -sum((t(y) %*% log(yhat)) + (t(1-y) %*% log(1-yhat)))/m
    cost_vector = c(cost_vector, cost)




  }
  print(step)

  return(list(w_cur, error, g, yhat, cost_vector))

}
```

```
# Performs On-Line Stochastic Gradient Descent

log_regr_stoch2 = function(X,y, tol = 0.001, a = .1){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion


  # Function performs gradient descent algorithm to obtain the
  # vector of weights w* that minimize the log likelihood
  # function of the data (gives MLE)

  # Note: MLE for logistic regression doesnt have a closed form solution.

  m = nrow(X) # number of observations

  d = ncol(X) # dimension of data; number of variables

  w_cur = rep(1/d,d) # initialize weight vector

  error = 10 # initialize error

  tol = tol # tolerance

  max_iter = 30000 # max iterations

  step = 0 # iteration counter

  b = 0 # bias term

  a = a # learning rate

  X = as.matrix(X)
  y = as.matrix(y)

  #set.seed(0)

  i_rand = sample(0:m, m, replace = FALSE)

  while ((error > tol) && (step < max_iter)){



    w_prev = w_cur

    cur_i = i_rand[step + 1%%m]

    yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b)))



    g =   t(yhat - y[cur_i,]) %*% X[cur_i,]

    w_cur = w_prev - a * t(g)


    error = as.double(sqrt(crossprod(t(g))))

    step = step + 1



  }
  print(step)

  return(list(w_cur, error, g, yhat))

}
```

```
#performs mini Batch gradient descent algorithm

log_regr_minibatch = function(X,y, tol = 0.001, a = .1, batch_size ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion


  # Function performs gradient descent algorithm to obtain the
  # vector of weights w* that minimize the log likelihood
  # function of the data (gives MLE)

  # Note: MLE for logistic regression doesnt have a closed form solution.

  m = nrow(X) # number of observations
```

```
d = ncol(X) # dimension of data; number of variables

w_cur = rep(1/d,d) # initialize weight vector

error = 10 # initialize error

tol = tol # tolerance



b = 0 # bias term

a = a # learning rate

X = as.matrix(X)

y = as.matrix(y)

set.seed(0)

n_mini_batches = m %/% batch_size # number of batches

i_rand = sample(1:m, m, replace = FALSE) # randomly permute observations



batches = split(i_rand, factor(1:n_mini_batches)) # partition into batches
max_iter = 30000 # max iterations
step = 0 # iteration counter






while ((error > tol) && (step < max_iter)){



  w_prev = w_cur #previous becomes current

  cur_i = batches[[(step %% n_mini_batches) + 1]] #select next batch

  yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b))) # pred


  g =   (1/batch_size) * t(yhat - y[cur_i,]) %*% X[cur_i,] #gradient

  w_cur = w_prev - a * t(g) # update weights


  error = as.double(sqrt(crossprod(t(g)))) #check error
```

```
    step = step + 1



  }




  return(list(w_cur, error, g, yhat))

}
```

```
# performs mini-batch gradient desent and keeps track of cost


log_regr_minibatch_cost = function(X,y, tol = 0.001, a = .1, batch_size, max_iter = 30000 ){
  # Function that takes in:
  #X, a data matrix (each row an observation)
  #y, a vector of labels (each row an observation)
  #m, a batch size (1, minibatch, n)
  #a, a specified learning rate
  #tol, a stopping criterion


  # Function performs gradient descent algorithm to obtain the
  # vector of weights w* that minimize the log likelihood
  # function of the data (gives MLE)

  # Note: MLE for logistic regression doesnt have a closed form solution.

  m = nrow(X) # number of observations

  d = ncol(X) # dimension of data; number of variables

  w_cur = rep(1/d,d) # initialize weight vector

  error = 10 # initialize error

  tol = tol # tolerance



  b = 0 # bias term

  a = a # learning rate

  X = as.matrix(X) # coerce to matrix

  y = as.matrix(y) # coerce to matrix

  set.seed(0) # set random seed
  n_mini_batches = m %/% batch_size # calculate number of batches

  i_rand = sample(1:m, m, replace = FALSE) # cyclic randomization



  batches = split(i_rand, factor(1:n_mini_batches)) #partition data into subsets
  max_iter = max_iter # max iterations
  step = 0 # iteration counter

  cost_vector = vector('numeric', length = max_iter) #initialize cost vector




  while ((error > tol) && (step < max_iter)){



    w_prev = w_cur #weights from last iteration become previous weights

    cur_i = batches[[(step %% n_mini_batches) + 1]] #which batch to use

    yhat = 1/(1 + exp(-(X[cur_i,]%*%w_prev + b))) #prediction


    g =   (1/batch_size) * t(yhat - y[cur_i,]) %*% X[cur_i,] # gradient

    w_cur = w_prev - a * t(g) # update weights

    cost_vector[step+1] = -sum((y[cur_i,] * log(yhat)) + ((1-y[cur_i,]) * log(1-yhat)))/batch_size # ap



    error = as.double(sqrt(crossprod(t(g)))) # calculate error

    step = step + 1 #increment steps


    #add a break statement
    # print(w_cur)


  }





  return(list(w_cur, error, g, yhat, cost_vector))

}
```

```r
#Newtons Method

newtons_method_chol = function(X,y,W,m,iter_n =10){
  # sigmoid function

  # sigmoid function
  sigmoid  = function (X,W) {
    z <- X %*% W
    sig <- 1 / (1 + exp(-z) )
    return (sig)
  }


  cost = function (X,y,W,m) {
    # cost function

    cost <- sum(-y * log(sigmoid(X,W)) - (1 - y) * log(1 - sigmoid(X,W)))
    return( cost/m)
  }


  grad = function (X,y,W,m) {
    #gradient or first derivative of the cost function

    A <- sigmoid(X,W) - y
    return( crossprod(X,A)/m)
  }


  Hessian = function (X,W,m) {
    # Hessian

    #chol <- chol(t(X) %*% X )
    h_c <- diag(sigmoid(X,W)) *(1- diag((sigmoid(X,W))))
    hess <- (t(X)%*%X)* h_c
    #hess <- X%*%((diag(sigmoid(X,W)) - diag((sigmoid(X,W))^2))*t(X))
    return (hess/m)
  }

  cost_vector = c()
  i = 0
  while(i < iter_n) {

    chol <- chol(Hessian(X,W,m))
    inv_chol = chol2inv(chol)
    W = W - inv_chol%*% grad(X,y,W,m)
    cur_cost = cost(X,y,W,m)
    cost_vector = c(cost_vector, cur_cost)

    if(sqrt(crossprod(grad(X,y,W,m))) < 0.0001){break}
    i = i+1

  }
  return(list(W, cost_vector))
}
```

```r
temp = sim_data(n = 10000) # simulate a data set with 10,000 observations

X_sim = temp[[1]] # Pull out the covariates

y_sim = temp[[2]] # Pull out the response


# temp2 = read.csv('~/Documents/Data/STA141c/heartdata.csv')
#
#
# X_heart = temp2[,1:14]
# y_heart = temp2[,15]
```

```r
testtrainsep = function(A,b){ # A is a data matrix, # b is vector of labels
  # Function that splits data into 75 - 25 train/test split

  n = nrow(X_sim)

  c = n%/%(4/3)

  #Set the training data set
  X_train = A[1:c,]
  y_train = b[1:c]

  #Set the test data set
  X_test = A[(c+1):n,]
  y_test = b[(c+1):n]

  #Make list that contains the test and train data
  traintest = list(X_train = X_train, y_train = y_train,
                   X_test = X_test, y_test  = y_test)

  return(traintest)
}
```

```r
data1 = testtrainsep(X_sim, y_sim) #train/test data

X_sim_train = data1$X_train # X train
y_sim_train = data1$y_train # y train

X_sim_test = data1$X_test # X test
y_sim_test = data1$y_test # y test
```

Batch Gradient Descent

```r
test1 = log_regr(X_sim_train, y_sim_train, a = 0.1)

#apply batch Gradient Descent algorithm to train
```

```r
conf_matrix(test1[[1]], X_sim_test, y_sim_test)
```

```
#return accuracy and confusion matrix of predictions
```

On-Line Stochastic Gradient Descent

```
test2 = log_regr_stoch2(X_sim_train, y_sim_train, a = 0.0001, tol = 0.000001)

#apply stochastic gradient descent algorithm
```

```
conf_matrix(test2[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions
```

On-Line Stochastic Gradient Descent with Momentum

```
test3 = log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3)

#apply SGD with momentum algorithm
```

```
conf_matrix(test3[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions
```

Mini-Batch Stochastic Gradient Descent, m = 4

```
test4 = log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4, a = 0.001)

#apply mini batch SGD to data
```

```
conf_matrix(test4[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions
```

Newtons Method

```
z = newtons_method_chol(X_sim_train, y_sim_train, rep(0,10), m = 7500, iter_n = 5)


#apply newton's method to data
```

```
conf_matrix(z[[1]], X_sim_test, y_sim_test)

#return accuracy and confusion matrix of predictions
```

Timings: Batch, On-Line, On-Line with Momentum

Run at own risk (batch takes forever)

```
sgd_timings = microbenchmark::microbenchmark(
  log_regr(X_sim_train, y_sim_train, a = 0.1),
  log_regr_stoch2(X_sim_train, y_sim_train, a = 0.001, tol = 0.000001)#,
#  log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3)

)
```

```r
#compute benchmarks for batch and on-line stochastic gradient descent
```

Table and Violin Plots for Timings above

```r
sgd_timings
xlabs = c( 'Batch', 'On-Line SGD')
ggplot2::autoplot(sgd_timings) + scale_x_discrete(labels = xlabs) + ggtitle('Time Anlalysis Batch vs On

#Plot the timings
```

Timings

```r
sgd_timings2 = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train, y_sim_train, a = 0.001, tol = 0.0001),
  log_regr_stoch_mom(X_sim_train, y_sim_train, nu = 0.0001, alpha = 0.3),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4, a = 0.001),
  newtons_method_chol(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4)




)

#compute timings for all methods
```

```r
library(ggplot2)
sgd_timings2

xlabs = c('On-Line SGD', 'On-Line SGD with Momentum', 'Minibatch, m = 4', 'Newtons Method')

ggplot2::autoplot(sgd_timings2) + scale_x_discrete(labels = xlabs) + ggtitle('Time Anlalysis for Differ


#plot the timings
```

```r
batch_timings = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train,y_sim_train),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 1),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 2),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 8),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 16),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 32),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 64)#,
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 128),
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 256),
  # log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 512)
)




#compute timings for mini-batch of different batch sizes
```

```r
library(ggplot2)
```

```
batch_timings

labels = c('Base: On-Line SGD','m = 1','m = 2','m = 4','m = 8','m = 16','m = 32', 'm = 64')
autoplot(batch_timings) + scale_x_discrete(labels = labels) +ggtitle('Time Anlalysis for Variable Batch

#plot the timings
```

```
mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 8)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l',  ylim = c(0,0.8))

# mb4_avg = ifelse(is.na(mb4_cost), 0, mb4_cost)
#
# mb4_avg = split(mb4_avg, 1:5000)
#
#
# mb4_avg = sapply(mb4_avg, mean)
```

```
plot(1:length(mb4_cost), mb4_cost, 'l')
```

```
batch_timings = microbenchmark::microbenchmark(
  log_regr_stoch2(X_sim_train,y_sim_train),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 1),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 2),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 4),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 8),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 16),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 32),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 64),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 128),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 256),
  log_regr_minibatch(X_sim_train, y_sim_train, batch_size = 512)
)
```

```
batch_timings
#ggplot2::autoplot(batch_timings)

labels = c('Base: On-Line SGD','m = 1','m = 2','m = 4','m = 8','m = 16','m = 32', 'm = 64', 'm = 128',
autoplot(batch_timings) + scale_x_discrete(labels = labels) +ggtitle('Mini-Batch SGD Time Analysis') +
```

```
mom_cost = log_regr_stoch_mom_cost(X_sim_train, y_sim_train, nu = 0.001, alpha = 0.3)

mom_cost = mom_cost[[5]]

mom_cost = mom_cost[mom_cost != 0]

plot(1:length(mom_cost), mom_cost, 'l')
```

```
#compute cost over time of momentum
```

```
mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 7500)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l', col = '#0072B2', lwd = 2, lty = 'solid', xlim = c(0,23000), xla

title(main = 'Batch GD Covergence Plot, alpha = 0.001')

#points(mb4_cost, col = '#D55E00', pch = 19)

grid(lty = 'dotted')


#compute minibatch cost over time
```

```
mb4_cost = log_regr_minibatch_cost(X_sim_train, y_sim_train, a = 0.001, batch_size = 1)

mb4_cost = mb4_cost[[5]]

mb4_cost = mb4_cost[mb4_cost != 0]

plot(1:length(mb4_cost), mb4_cost, 'l')




#compute minibtach over time and plot
```

```
# Newton's Method without Cholesky


newtons_method= function(X,y,W,m,iter_n =10){
  # sigmoid function

  # sigmoid function
  sigmoid  = function (X,W) {
    z <- X %*% W
    sig <- 1 / (1 + exp(-z) )
    return (sig)
  }


  cost = function (X,y,W,m) {
    # cost function

    cost <- sum(-y * log(sigmoid(X,W)) - (1 - y) * log(1 - sigmoid(X,W)))
    return( cost/m)
  }
  grad = function (X,y,W,m) {
    #gradient or first derivative of the cost function

    A <- sigmoid(X,W) - y
    return( crossprod(X,A)/m)
  }


  Hessian = function (X,W,m) {
    # Hessian

    #chol <- chol(t(X) %*% X )
    h_c <- diag(sigmoid(X,W)) *(1- diag((sigmoid(X,W))))
    hess <- (t(X)%*%X)* h_c
    #hess <- X%*%((diag(sigmoid(X,W)) - diag((sigmoid(X,W))^2))*t(X))
    return (hess/m)
  }

  cost_vector = c()
  i = 0
  while(i < iter_n) {

    #chol <- chol(Hessian(X,W,m))
    #inv_chol = chol2inv(chol)
    W = W - solve(Hessian(X,W,m))%*% grad(X,y,W,m)
    cur_cost = cost(X,y,W,m)
    cost_vector = c(cost_vector, cur_cost)

    if(sqrt(crossprod(grad(X,y,W,m))) < 0.0001){break}
    i = i+1

  }
  return(list(W, cost_vector))
}
```

```
#benchmarking newton method algorithms (cholesky and no cholesky)

sgd_timings2 = microbenchmark::microbenchmark(
newtons_method(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4),
  newtons_method_chol(X_sim_train, y_sim_train, W = rep(0,10), m = 7500, iter_n = 4)




)
```

```
data=as.list(numeric(100))
for (i in 1:100) {
  data[[i]]=sim_data()
}
newton_accuracies=numeric(100)

#100 sims for Newton's Method accuracies
for (i in 1:100) {
  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]

  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=newtons_method_chol(X_sim_train, y_sim_train, rep(0, ncol(X_sim_train)), ncol(X_sim_train), i


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  newton_accuracies[i]=z/length(y_sim_test)
}
df=data.frame(y=y, method="Newton")

#5 number summary for Newton's Method accuracy
y=c(min(newton_accuracies), quantile(newton_accuracies)[2], mean(newton_accuracies), quantile(newton_ac

mb_accuracies=as.list(numeric(7))
a=c(1, 2, 4, 8, 16, 32, 64)

#simulations of mini batch accuracy with various batch sizes
for (i in c(1, 2, 4, 8, 16, 32, 64)) {
  results=numeric(100)
  for (j in 1:100) {


    X_sim=data[[j]][[1]]
    y_sim=data[[j]][[2]]

    X_sim_train = X_sim[1:750,]
    X_sim_test = X_sim[751:1000,]

    y_sim_train = y_sim[1:750]
    y_sim_test = y_sim[751:1000]

    ret=log_regr_minibatch(X_sim_train, y_sim_train, batch_size=i)
    weight=ret[[1]]

    test_pred = X_sim_test %*% weight
    test_pred = 1/(1 + exp(-test_pred))
    test_labels = ifelse(test_pred > 0.5, 1, 0)

    z=sum(test_labels == y_sim_test)
    results[j]=z/length(y_sim_test)
  }
  mb_accuracies[[(log(i, base=2))+1]]=results
```

```
}

#list of accuracies across bacth sizes
mb_accuracies

d=data.frame(unlist(mb_accuracies))
d$batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64, 
d[701:800,]=stochcopy
d$batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64, 
d$batch_size=as.factor(d$batch_size)
names(d)=c("Accuracy", "Batch_size")
d=d[c(701:800, 1:700),]

one=d[101:200,]
two=d[201:300,]
four=d[301:400,]
eight=d[401:500,]
sixteen=d[501:600,]
tt=d[601:700,]
sf=d[701:800,]

#mini-batch 5 number summary
fivensumsmb=data.frame(quantile(one$Accuracy), quantile(two$Accuracy), quantile(four$Accuracy), quantil
fivensumsmb[6,]=means
row.names(fivensumsmb)=c("Min", "LQ", "Median", "UQ", "Max", "Mean")
names(fivensumsmb)=c("m=1", "m=2", "m=4", "m=8", "m=16", "m=32", "m=64", "Stochastic")
fivensumsmb=t(fivensumsmb)
fivensumsmb=fivensumsmb[,c(1,2,6,3,4,5)]
fivensumsmb=fivensumsmb[c(7, 6, 5, 4, 3, 2, 1, 8),]

stochcopy=stoch
names(stochcopy)=c("Accuracy", "Batch_size")
d2=d
d2[701:800,]=stochcopy
d2$Batch_size=c(rep(1, 100), rep(2, 100), rep(4, 100), rep(8, 100), rep(16, 100), rep(32, 100), rep(64,
d2$Batch_size=as.factor(d2$Batch_size)

#plot of all batch sizes' accuracies
ggplot(d, aes(x=factor(Batch_size, level=c("1 (Stochastic)", 1, 2, 4, 8, 16, 32, 64)), y=Accuracy))+geo
  coord_flip()+ggtitle("Accuracy Analysis of Various Batch Sizes")+xlab("Batch Size")


#list to store accuracies of the 4 models
models_accuracies=as.list(numeric(4))
#simulating batch logistic regression
base_accuracies=numeric(100)
for (i in 1:100) {

  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]
  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=log_regr(X_sim_train, y_sim_train)[[1]]


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  base_accuracies[i]=z/length(y_sim_test)
}
models_accuracies[[1]]=base_accuracies
models_accuracies[[2]]=newton_accuracies
#mb_accuracies[[3]] is the accuracies with batch size=4
models_accuracies[[3]]=mb_accuracies[[3]]

stoch_accuracies=numeric(100)
#simulation of stochastic logistic regression
for (i in 1:100) {

  X_sim=data[[i]][[1]]
  y_sim=data[[i]][[2]]

  X_sim_train = X_sim[1:750,]
  X_sim_test = X_sim[751:1000,]

  y_sim_train = y_sim[1:750]
  y_sim_test = y_sim[751:1000]

  weights=log_regr_stoch2(X_sim_train, y_sim_train)[[1]]


  test_pred = X_sim_test %*% weights
  test_pred = 1/(1 + exp(-test_pred))
  test_labels = ifelse(test_pred > 0.5, 1, 0)

  z=sum(test_labels == y_sim_test)
  stoch_accuracies[i]=z/length(y_sim_test)
}
models_accuracies[[4]]=stoch_accuracies
df=data.frame(unlist(models_accuracies))
df$model=c(rep("Batch", 100), rep("Newton's", 100), rep("Mini-Batch, m=4", 100), rep("Stochastic", 100)
df$model=as.factor(df$model)
names(df)=c("Accuracy", "Model")

#Plot of 4 models accuracies
ggplot(df, aes(x=Model, y=Accuracy))+geom_violin()+ylim(c(0.8, 1))+coord_flip()+ggtitle("Accuracy Analy

batch=df[1:100,]
newton=df[101:200,]
mb=df[201:300,]
```

```r
stoch=df[301:400,]

means=c(mean(batch$Accuracy, mean(newton$Accuracy), mean(mb$Accuracy), mean(stoch$Accuracy)))

#5 number summaries for all 4 models
fivensums=data.frame(quantile(batch$Accuracy), quantile(newton$Accuracy), quantile(mb$Accuracy), quantil
fivensums[6,]=means
row.names(fivensums)=c("Min", "LQ", "Median", "UQ", "Max", "Mean")
names(fivensums)=c("Batch", "Newton's", "Mini-Batch, m=4", "Stochastic")
fivensums=t(fivensums)
fivensums=fivensums[,c(1,2,6,3,4,5)]
fivensums=fivensums[c(4, 2, 3, 1),]
```