STA3431 - Monte Carlo Methods - Self-Directed Project

Names: Sergio E. Betancourt and Nathan Friedman
Department: Statistical Sciences, Program: MSc (2019)
E-mails: sergio.betancourt@mail.utoronto.ca and nathan.friedman@mail.utoronto.ca

Date: 2018-11-26

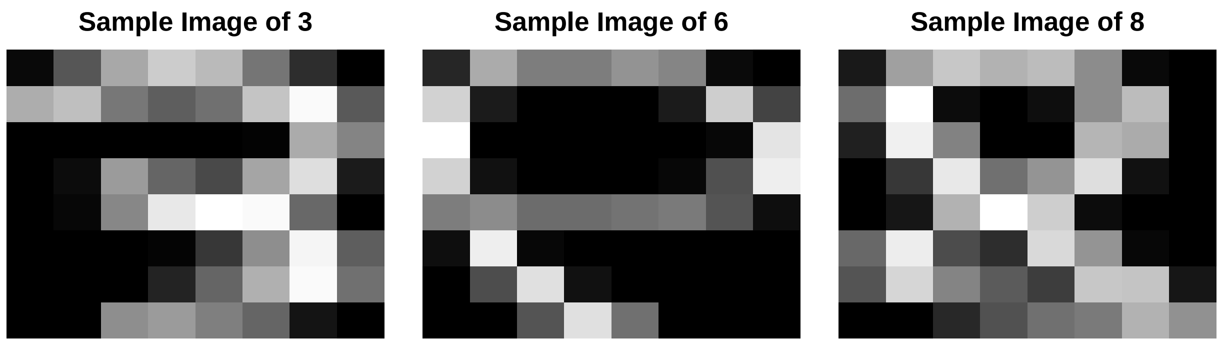
Abstract

An investigation into the performance of a variety of Monte Carlo methods in regards to the classification problem of hand-written digits (0-9) using a dimension-reduced version of the MNIST dataset. We achieve a 65% prediction accuracy—baseline of 44% using Gaussian discriminant analysis—with a training set smaller than the number of parameters estimated, thus overcoming a widespread learning condition requiring n > p.

Problem Statement

Image recognition is one of the core areas of research and application of statistical (machine) learning. Gigantic developments in the processing capabilities of modern computers has propelled scientific computation to perhaps its most advanced stage, without any signs of slowing down. In this project we seek to apply Monte Carlo methods to the classification of the hand-written digits repository MNIST (*Wiki*). This repository has been widely studied, and it contains observations for the digits 0 to 9.

Our digit data is comprised of an 8x8 pixel grid representing each image of a particular digit. We obtained it from the site for the course CSC411 (*site*). Each component in this grid describes the pixel's light intensity (in the interval 0 to 1).



In this project we consider the following techniques to achieve our classification:

1. Random-Walk Metropolis

2. Independence Sampler

3. Their Component-wise variations

In the end we built a model that carries out the classification retroactively for each new digit we examine, assigning classification probabilities for each one of the 10 possible digits. What distinguishes our implementation from the mainstream machine learning methods is the fact that **we can obtain reasonable prediction accuracy (65%) with a training data set (n=60) that is smaller than our number of parameters ($\beta \in \mathbb{R}^{65}$).**

Setting the Stage

Our classification problem is modeled below for X_i (64-vector of pixels corresponding to the ith observation) and Y_i (its corresponding label - 1 if digit of interest, 0 otherwise):

$$\text{Let } Y_i \sim bernoulli(p_i), \text{where } p_i = \sigma(X_i;\boldsymbol{\beta}) = \frac{1}{1+e^{-X_i\boldsymbol{\beta}}} \text{ ("the logit link"), and prior } \boldsymbol{\beta} \sim N(\mathbf{0},\Sigma)$$

$$[\text{Aside: } \beta \sim N(0,I) \implies \beta_i \perp \beta_j \sim N(0,1)]$$

$$\text{Then our target is: } \pi(\boldsymbol{\beta}|X,Y) = C\, g(X,Y|\boldsymbol{\beta})f(\boldsymbol{\beta}) =$$

$$C\,(\prod_{i=1}^{n}(\sigma(X_i;\boldsymbol{\beta}))^{Y_i}(1-\sigma(X_i;\boldsymbol{\beta})^{1-Y_i})f(\boldsymbol{\beta}), \text{where C unknown and } f(\boldsymbol{\beta}) \text{ is } MVN(\mathbf{0},\Sigma)$$

$$\text{and thus, } \pi(\boldsymbol{\beta}|X,Y) = C_*\,(\prod_{i=1}^{n}(\sigma(X_i;\boldsymbol{\beta}))^{Y_i}(1-\sigma(X_i;\boldsymbol{\beta}))^{1-Y_i}))\,e^{-\frac{1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta}}$$

$$\text{Ultimately, our predictive posterior is: } P(Y^{new}=1|X^{old},Y^{old}) = \int_{\boldsymbol{\beta}}\sigma(X^{new};\boldsymbol{\beta})\pi(\boldsymbol{\beta}\,|\,X,Y)d\boldsymbol{\beta}$$

Note that our target distribution $\pi(\boldsymbol{\beta}|X,Y)$ is not particularly easy to sample from, which leads the way to a perfect application of Monte Carlo methods. Moreover, in maximum-likelihood estimation and related optimization techniques, the problem of training your model of dimension “p”, where p is greater than the number of training observations “n” almost always spells trouble (overfitting or non-convergence). There are numerous penalization and regularization techniques to address this situation as the model solves for the “optimal” parameters in the model. Nonetheless, our Monte Carlo implementation is **not constrained** by this limitation. In fact, our implementation makes predictions on a monumental combination of probable coefficients and weights them based on how likely they are.

For our estimation we let:

$$g(X,\boldsymbol{\beta},Y) = C_*\prod_{i=1}^{n}(\sigma(X_i;\boldsymbol{\beta}))^{Y_i}(1-\sigma(X_i;\boldsymbol{\beta}))^{1-Y_i}f(\boldsymbol{\beta})$$

$$\text{then } \log(g) = \log(C_*) + N\,\frac{-1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta} + \sum_{i=1}^{N}Y_i\log(\sigma(X_i;\boldsymbol{\beta})) + (1-Y_i)\log(1-\sigma(X_i;\boldsymbol{\beta}))$$

Regarding our Monte Carlo implementation, we consider four algorithms in this project: **Random-Walk Metropolis**, **Independence Sampler**, and their **component-wise** flavors.

For all of these methods we are first faced with the challenge of picking initial values for the 65 β_i parameters used by our models. Understandably, choosing a smart set of these parameters in a 65-dimensional space (here $\mathbb{R}^{65}$) is not easy. First, choosing a starting point which is away from the most voluminous portion of the distribution at hand may actually lead to less efficient emulation of the target distribution. On the other hand, a small number of iterations "M," or, in the case of MCMC methods, an excessively large burn-in amount "B," may both yield overrepresentation of the distribution of the parameters in question near the starting value, and thus an underrepresentation of the target distribution's high volume area.

To this end, picking the initial values may be approached by examining their prior distribution's centrality and dispersion, where centrality means picking the vector $\boldsymbol{\beta}$ around the distribution's mode, or area of largest volume, and dispersion allows coverage of the distribution's areas of interest.

A Bayesian framework suggests three possible candidates of centrality: The mean of the parameters' prior, their maximum-likelihood estimate, and the maximum a posteriori (MAP). In our implementation we choose the starting value to be centered at a prior mean of 0 given the difficulty of calculating the MAP estimate and the non-convergence of the MLE fit.

For our final implementation, the training set consisted of 60 randomly sampled observations, such that each digit appeared at least once in the set. The training set was used in the target distribution (the posterior), specifically for the calculation of the likelihood of a given $\boldsymbol{\beta}$.

Implementation

For all approaches below we first focused on classifying observations corresponding to the digit 3, against all other possible labels. Then, once we found the component-wise algorithms to be superior to their standard counterparts, we created a classifier that estimates the probability of a given testing digit to be each of the 10 possible digits in question.

Random-Walk Metropolis

The Metropolis algorithm can be characterized by the premise of symmetric proposals of $\boldsymbol{\beta}$ parameters according to a joint proposal distribution. Recall how for data X and Y in our training set, our implementation focuses on estiamtion of the probability of our items in our testing set to be a particular digit versus them being any other digit. Simulation over different combinations of $\boldsymbol{\beta}$ parameters should yield an estimate of such probability.

Now, to invoke the Central Limit Theorem, vital for the construction of confidence intervals of our estimates, we note how for π with exponential tails, the Metropolis algorithm is geometrically ergodic.

$$\text{Recall our target } \pi(\boldsymbol{\beta}|X,Y) = C_* \left(\prod_{i=1}^{n} (\sigma(X_i;\boldsymbol{\beta}))^{Y_i}(1-\sigma(X_i;\boldsymbol{\beta}))^{1-Y_i}\right) e^{-\frac{1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta}}$$

$$\text{where } \sigma(X_i;\boldsymbol{\beta}) = \frac{1}{1+e^{-X_i\boldsymbol{\beta}}}$$

$$\text{Note, as } \boldsymbol{\beta} \to \infty,\ \sigma(X_i;\boldsymbol{\beta}) \to 1 \iff 1 - \sigma(X_i;\boldsymbol{\beta}) \to 0, \text{ and } e^{\frac{-1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta}} \to 0$$

$$\text{Thus } \pi(\boldsymbol{\beta}|X,Y) \to 0$$

Geometric ergodicity follows from the definition of exponentially-light tails

$$\text{where } \pi(\boldsymbol{\beta}) \le a e^{-b|\boldsymbol{\beta}|} \text{ with } a = C_*,\ b = \frac{-1}{2\sigma^2}, \text{ and } \mathbf{c} = 1$$

$$\text{Justification for } b\text{: Letting } \Sigma = \sigma \text{I},\ \sigma > 0, -\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta} = -\frac{1}{\sigma^2}|\boldsymbol{\beta}|^2 \le -\frac{1}{\sigma^2}|\boldsymbol{\beta}| \text{ for } \boldsymbol{\beta} \ge \mathbf{c} = \mathbf{1}$$

To claim geometric ergodicity we also require that $E_\pi(|h|^p) < \infty$ for some $p > 0$, which we assume to be true.

As it pertains to proposals, we consider the following proposal distributions, both centered at the "current" vector of parameters $\boldsymbol{\beta}$ at every iteration:

- $\text{N}(\boldsymbol{\beta}_{n-1}, \sigma I)$ where $\sigma \in R$
- $\text{N}(\boldsymbol{\beta}_{n-1}, \sigma\Sigma)$ where Σ is the covariance matrix of all 64 pixel locations across all observations in our training set ($\Sigma = \text{Cov}(X)$), plus a trailing 1 as an intercept term (β_0), and $\sigma \in R$

Independence Sampler

The Independence Sampler algorithm is a special case of the Metropolis-Hastings algorithm, where we make proposals based on a fixed density q, which is independent of the current parameter values in our iteration. In a nutshell, $\{Y_n\} \sim$ iid $q(.)$. For this implementation, we use an $MVN(\mathbf{0}, \sigma_q I)$ proposal distribution.

Given how our proposal distribution from X to Y $q(X,Y)$ simplifies to $q(Y)$ for the Independence Sampler, it follows that Π is a stationary distribution. Explicitly we consider:

$$q(\beta) = (2\pi\sigma_q^2)^{\frac{-65}{2}} \exp\left(\frac{-1}{2\sigma_q^2}\sum_{i=1}^{65}\beta_i^2\right)$$

Note how for all $\boldsymbol{\beta} \in \mathbb{R}^{65}$, $q(\boldsymbol{\beta}) > 0$, which is the same for our target distribution π. Moreover, our proposal and target distributions share the same support, which means that our implementation of the independence sampler is ergodic. We now argue for geometric ergodicity.

Recall how geometric ergodicity of the independence sampler requires that we find a δ for which $q(\boldsymbol{\beta}) \ge \delta\pi(\boldsymbol{\beta})$ for all $\boldsymbol{\beta} \in \mathbb{R}^{65}$. Namely, we are looking for a δ such that:

$$(2\pi\sigma_q^2)^{\frac{-65}{2}} \exp(\frac{-1}{2\sigma_q^2}|\boldsymbol{\beta}|^2) > \delta\, C_* \prod_{i=1}^{n} (\sigma(X_i;\boldsymbol{\beta}))^{Y_i}(1-\sigma(X_i;\boldsymbol{\beta}))^{1-Y_i}))\, e^{-\frac{1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta}}$$

where $\sigma_q^2 = 0.006/1.3\sigma^2$ and

$$C_* = \int_{\boldsymbol{\beta}} \prod_{1}^{n} \sigma(X_i\boldsymbol{\beta})^{Y_i}(1-\sigma(X_i\boldsymbol{\beta}))^{1-Y_i})\ \exp(-\frac{1}{2}\boldsymbol{\beta}'\Sigma^{-1}\boldsymbol{\beta})\, d\boldsymbol{\beta},\ \text{where } \sigma(X_i;\beta) = \frac{1}{1+\exp(-X_i\boldsymbol{\beta})}$$

We can pinpoint a bound for our δ with the below:

$$\delta \leq \frac{(2\pi\sigma_q^2)^{-32.5}\exp(-\frac{1}{2\sigma_q^2}||\boldsymbol{\beta}||_2^2}{C * L(\boldsymbol{\beta})\exp(-\frac{1}{2\sigma_p^2}||\boldsymbol{\beta}||_2^2)}, \text{ where } L(\boldsymbol{\beta}) \text{ is the likelihood}$$

Let $C_* = \frac{(2\pi\sigma_q^2)^{-\frac{65}{2}}}{C}$, and note that this quantity is greater than zero, and that $\frac{1}{L(\boldsymbol{\beta})} > 0$ given how we are using logistic regression and it is never equal to one or zero. We get that: $0 < \delta \leq c * \frac{1}{L(\boldsymbol{\beta})}\exp(-\frac{1}{2}(\frac{1}{\sigma_q^2} - \sigma_p^2)||\boldsymbol{\beta}||_2^2))$, since the product of non-zero terms is non-zero. So there exists a delta that satisfies the above equation, and thus it is geometrically ergodic. It is far too complicated to calculate an explicit delta that satisfies this equation, but we can assume it is incredibly small. In this implementation we cannot guarantee quick convergence for the independence sampler.

Component-Wise Variations

The component-wise variation of the **Independence Sampler** algorithm is almost identical to the usual flavor, except that instead of proposing an entirely new set of parameters at every iteration, only one of them is subject to proposal, while the other 64 remain unchanged. The component-wise variation of the **Random Walk Metropolis** work similarly in that proposals are done one β_i at a time. This can be done systematically or uniformly at random, but we chose a systematic approach in our implementation.

In our implementation we chose to do 10,000 iterations “M” for each element in our testing set. We chose this “M” through trial and error, seeking the minimum number to achieve good results. Additionaly, given how for each of these iterations there were 65 subiterations, corresponding to each β_i, to be performed for 10 digits, the computational intensity became very high. In regards to the burn-in “B,” we chose it to be 750–a quantity reached through trial and error, too. Here we proposed from a $N(0, (\frac{1}{5})^2)$ distribution, where the variance chosen yielded the best acceptance rate (close to 23%).

We concluded that the main benefit of a component-wise approach for our implementation of the independence sampler is the following:

$$\text{Recall } R = \frac{L(Y)}{L(X)} * \exp\{(\frac{1}{2\sigma_q^2} - \frac{1}{2\sigma_p^2})(||Y||_2^2 - ||X||_2^2)\}$$

However, since component-wise moves one component at a time, for every iteration we have:

$$R = \frac{L(Y)}{L(X)} * \exp\{(\frac{1}{2\sigma_q^2} - \frac{1}{2\sigma_p^2})(Y_i^2 - X_i^2)\}$$

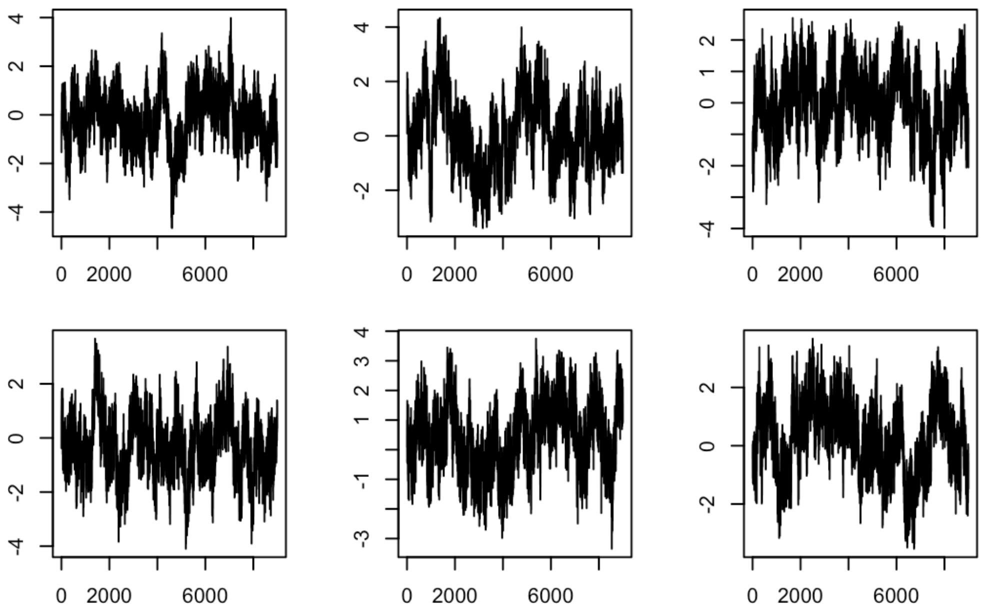
Since all but one component will differ between X and Y, the likelihood ratio will be approximately one, and thus using a component-wise approach makes is much easier to obtain a reasonable acceptance rate. A similar result applies to the Component-wise Random Walk Metropolis.

Since the $q(X)$ is still the normal distribution, for every vector $\boldsymbol{\beta}$ $q(X)$ will have positive support in $\mathbb{R}^{65}$. Accordingly, this aligns with the support of the target distribution. This follows from the fact that under these conditions it will be irreducible and aperiodic, and thus it will be ergodic. This, however, does not guarantee that it will work well.

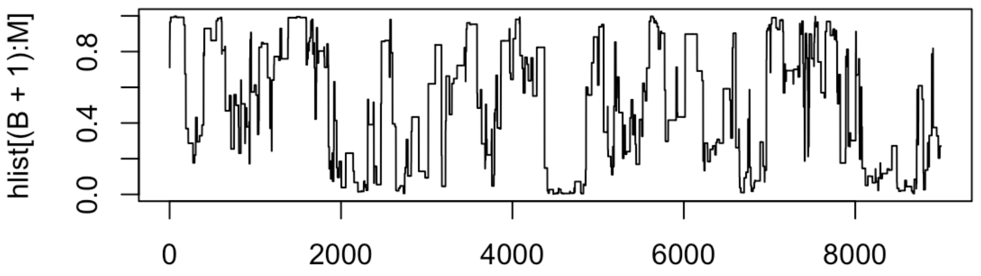
Results

Random Walk Metropolis

The Random Walk Metropolis algorithm performed extremely poorly in our implementation, yielding very unstable estimates for the classification probability, as well as the $\boldsymbol{\beta}$ parameters. Here are some plots of these parameters for a run with scale $\sigma = 1/100$ on the identity covariance matrix I :



Here is the trace plot of the values of the classification probability for this run:



We found a trade-off between the scale of the proposals and the acceptance rate, where the smaller the proposals made, the higher the acceptance rate and a very volatile trace plot of classification probabilities, with frequent swings going from 0 to

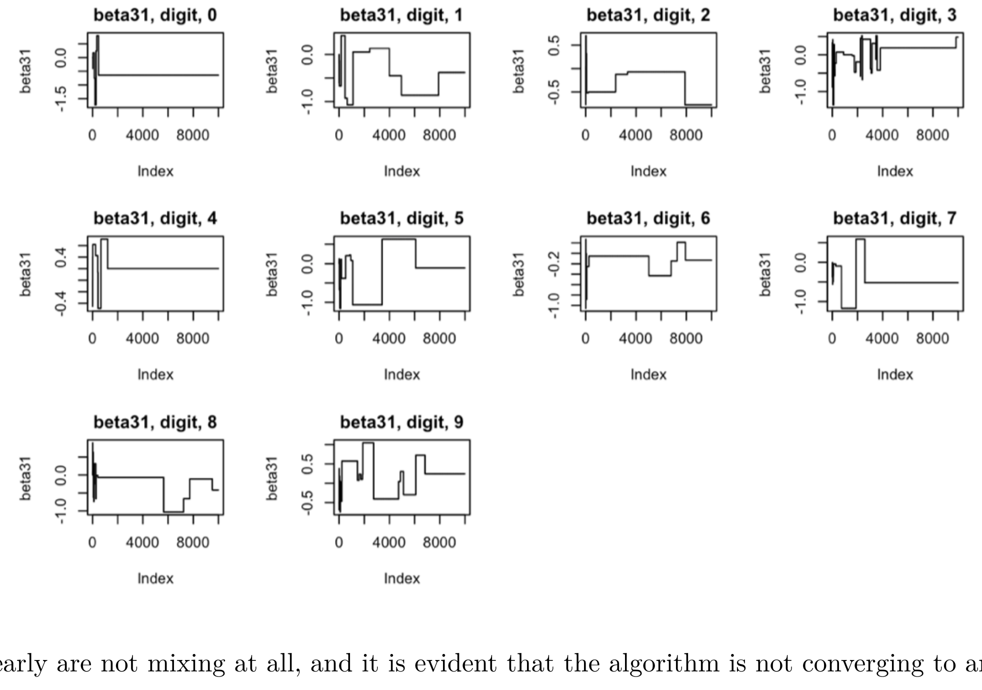
1. The other way around, with bigger proposals, also yielded unstable estimates, but an extremely low acceptance rate, which lead to the trace plots of the estimated probabilities to show a semblant of convergence, even though the estimates were not within what was expected.

In an effort to improve proposals we then considered the proposing from a multivariate normal distribution with a scaled Σ as the estimated covariance matrix from the entire training set. But this did not yield much improvement in the implementation.

The above proposal schemes yielded poor results in the estimation of the probability that dictates our classification. This may be attributed to the high dimensionality of the problem at hand, and the difficulty of getting the proposal joint probability distribution right. Although in the case of ordinary least squares regression we can look at the relationship between the different variables by examining their correlation matrix, a direct translation of this approach is not appropriate in this case.

Independence Sampler

The Independence Sampler algorithm also performed very poorly in our implementation, yielding very unstable estimates for the classification probability, as well as the $\boldsymbol{\beta}$ parameters. The trace plots are from the most "succesful" runs of this algorithm:



The trace plot clearly are not mixing at all, and it is evident that the algorithm is not converging to anything meaningful. For each β, and for each digit not a single trace plot shows good mixing.

Further investigation lead to interesting results. Recall that we accept with probability $A_n = \min(1, \frac{\pi(Y)q(X)}{\pi(X)q(Y)})$. Let $R \equiv \frac{\pi(Y)q(X)}{\pi(X)q(Y)}$, where again $\pi(\boldsymbol{\beta}; X, Y) \propto L(\boldsymbol{\beta}|Data) * \exp(-\frac{1}{2\sigma_p^2}||\boldsymbol{\beta}||_2^2)$ ($L(\boldsymbol{\beta}|Data)$ is the likelihood of beta, given the data).

Using $q(X) \propto \exp(\frac{-1}{2\sigma_q^2}||X||_2^2)$ we get that:

$$R = \frac{L(Y) * \frac{1}{(2\pi\sigma_p^2)^{-32.5}} * \exp(-\frac{1}{2\sigma_p^2}||Y||_2^2) * \frac{1}{(2\pi\sigma_q^2)^{-32.5}} * \exp(-\frac{1}{2\sigma_q^2}||X||_2^2)}{L(X) * \frac{1}{(2\pi\sigma_p^2)^{-32.5}} * \exp(-\frac{1}{2\sigma_p^2}||X||_2^2) * \frac{1}{(2\pi\sigma_q^2)^{-32.5}} * \exp(-\frac{1}{2\sigma_q^2}||Y||_2^2)} =$$

$$\frac{L(Y)}{L(X)} * \exp(\{\frac{-1}{2\sigma_p^2}(||Y||_2^2 - ||X||_2^2) - \frac{1}{2\sigma_q^2}(||X||_2^2 - ||Y||_2^2)\}) =$$

$$\frac{L(Y)}{L(X)} * \exp\{(\frac{1}{2\sigma_q^2} - \frac{1}{2\sigma_p^2})(||Y||_2^2 - ||X||_2^2)\} \text{ (likelihood ratio times} \times \text{an expression)}$$

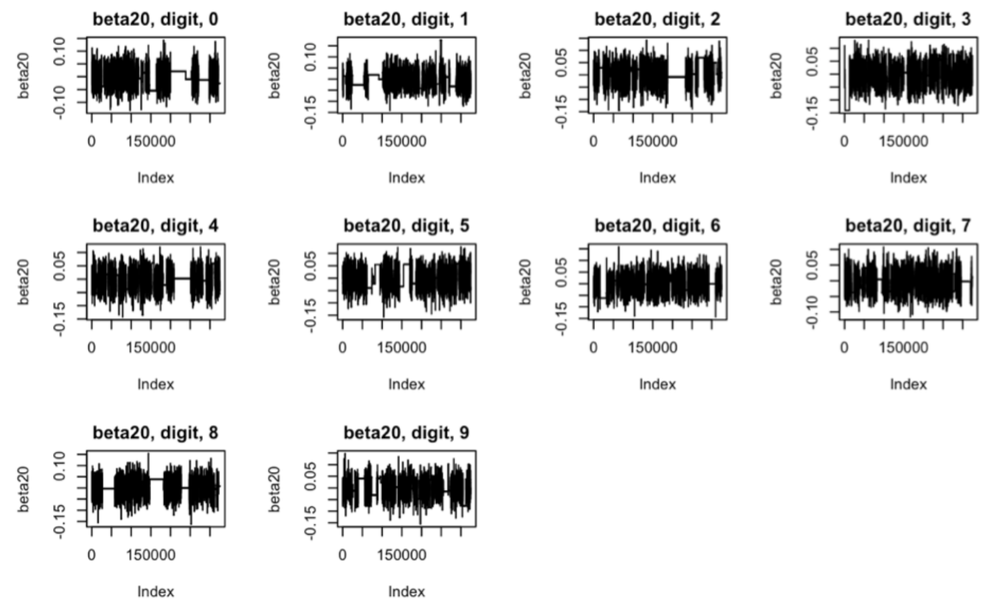
From this we can see some interesting characteristics. Specifically, one interesting result is that if $\sigma_p^2 = \sigma_q^2$ then the ratio is just equal to the likelihood ratio (which we would expect). This implies that for this problem, if a proposal distribution is the same as the prior, then you are just using the likelihood ratio to determine the probability of accepting. Another result, is that if X is far away from the origin and $\sigma_p^2 > \sigma_q^2$, then R will be small and we will get very few acceptances. Even armed with this knowledge, however, we still could not achieve sensible acceptance rates. It is incredibly easy to get stuck once a likely X is found, no matter the choices of variance. The likelihood ratio tends to be very small, which makes acceptances quite rare. But why is the likelihood so small? After some more investigation, the was determined to be the curse of dimensionality.

To illustrate this idea, consider the case for a multivariate normal distribution with mean 0, and a multivariate generating distribution with mean zero and variance 0.2I. The likelihood ratio would boil down to: $LR = \exp(\frac{-1}{2\sigma^2}(||Y||_2^2 - ||X||_2^2)$, and $Y_i \sim N(0, 1/5)$. Initially, X is taken as zero vector, and $\sigma^2 = 1$. Even with the small variance of 0.04, the expected $||Y||_2^2$ is still 13, and the expected likelihood ratio is 0.00265 (actually calculated this using a MC algorithm, notes in the appendix). This means that we'd expect to accept approximately 1 of every 378 proposals, which is not enough to produce a good sample. And thus we are compelled to move on to a **component-wise** approach.

Component-Wise Independence Sampler

The component-wise independence sampler looked as though it was mixing quite well, but it showed a propensity to get stuck. The figure below shows the trace plot for β_{20} for all digits. The plots for other betas are quite similar. What stands out is that it mixes well for the majority of iterations, but it gets stuck for prolonged periods. Since each component can only be updated once every 65 iterations, the flat parts look slightly exaggerated. It is still, however, indicative of mixing that is not optimal. Despite the algorithm getting stuck at points though, it still looks promising. The acceptances rates were around 20% (mean of 20.4%). This is miles ahead of the non-component wise approach which showed no mixing at all, and could only yield a maximum acceptance rate of 0.4%. The number of iterations here was 10,000 (times 65 sub-iterations).

A major downside of this method is that in order to obtain an adequate acceptance rate the variance had to be put very low. The effect of the this becomes evident when you look at the scale of the trace plots - the maximum range appears to be no greater than 0.25. This can very adversely effect the predictive power if the true centrality of density lies further away from 0 than what the betas reach

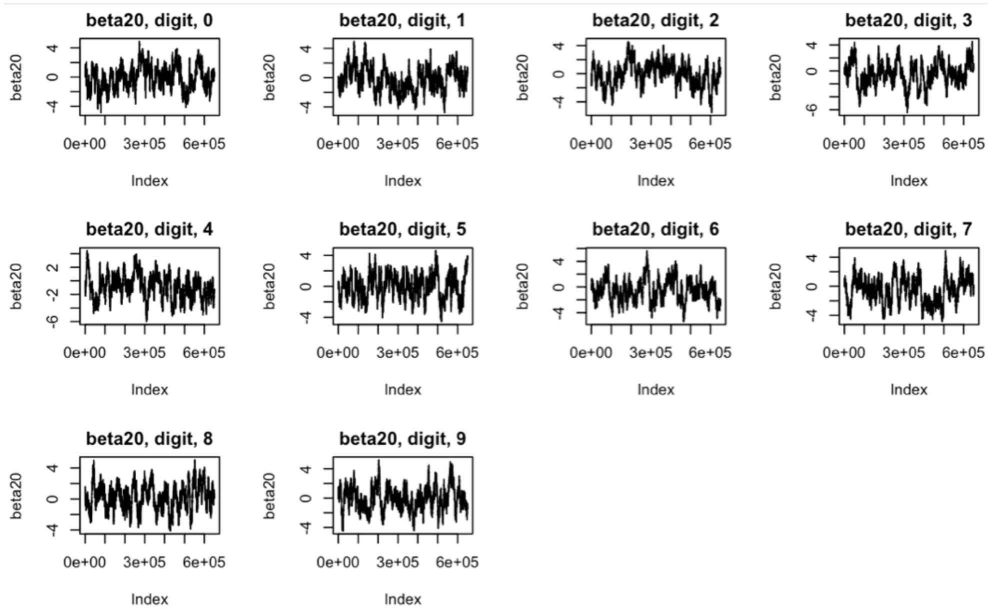


After 2 runs, an accuracy of 32% and 44% were achieved on 2 separate runs, yielding an average accuracy of 38%. The results were very interesting too. For the 44% accuracy run the highest success rate for a digit was classifying 1's. There were twelve 1's, of which 10 were correctly identified.

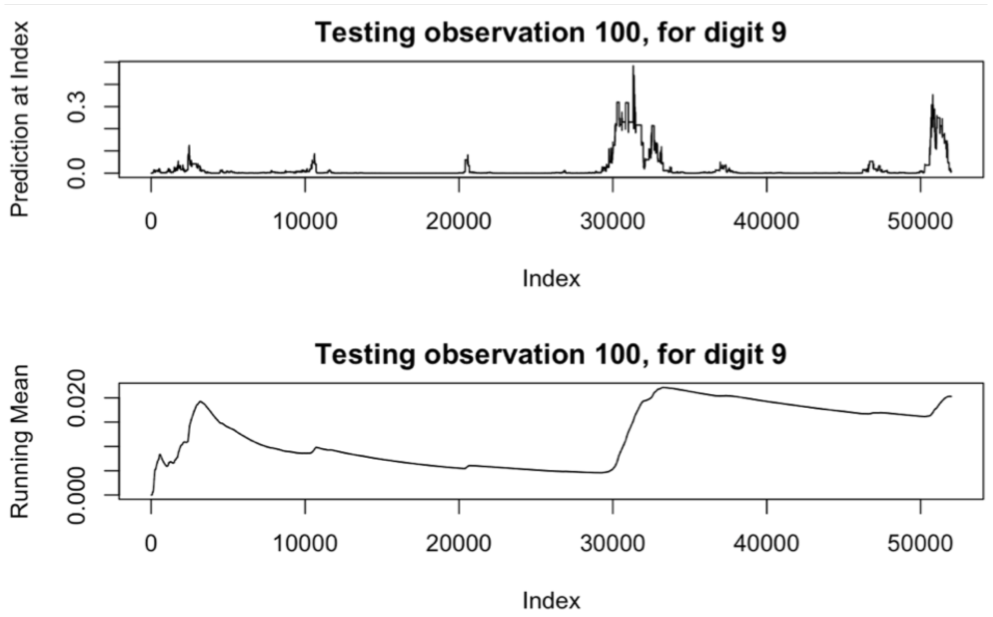
The most poorly identified digit was 5. There were nine 5's in the testing set, and every single one was estimated to be a three. Some clear patterns emerged in the mislabeling of digits, but interestingly none of them seemed to correspond to poor mixing or an unbalanced set. The algorithm estimated there were about 29 3's in total and in truth there were only 8 (7 of which were correctly identified). In the training set, however, only 7 of the 60 digits were actually 3. There were also seven 7s, and even 8 zeros, so the unbalanced training set cannot account for this. Fives were the most mislabeled digit, but the trace plots for 5 show that it is mixing about in the middle of the other digits. Given that if the same digit were guessed for each observation we would expect an accuracy of about 10%, the component-wise independence sampler performed satisfactorily. Adding the fact that a small unbalanced training set was used makes the results much more impressive.

Component-Wise Metropolis

In terms of making predictions, the component-wise Metropolis algorithm did the best. It boasted a **65%** and 44% accuracy for 2 separate runs with 10,000 and 1,000 iterations, respectively. The first beats the baseline of 55% attained via a Gaussian discriminant analysis (results from independent work for the course CSC411). Given that guessing each time would yield an expected accuracy of only 10%, these are phenomenal results. The plots below show the trace plot for each digit for β_{20} for the run with M = 10,000. The other β_i look similar.



There clearly is some mixing, but the plots also do appear quite thin at times. Although this can be indicative that our algorithm is not converging, given the high dimensionality of the problem it's likely not detrimental. The average varfact for the predictions across all the digits was around 113. The minimum was 107, and the maximum was 115. Overall, the varfact is quite high, but considering we are using a component-wise approach its expected to be high. The raw prediction that the first observation was a 6 (which it was) was 0.567. The margin of error for this observation was 0.0007, which leads to a 95% confidence interval of (0.5663, 0.5677). The margins of error for each prediction is very similar. This can be attributed to the large sample (650,000). A large indication that the algorithm is generating a sample that is emblematic of our target is that it provides good predictions. Theoretically, if our sample is good, then our predictions should be good, assuming our initial model is good, too. Therefore better predictions can be attributed to better samples. The firgures below illustrate the predictions at each iteration as to whether or not the observation was a 9 (it was a 2).



It is also reassuring that the predictions were better when the number of iterations was higher. Theoretically, the higher the number of iteration the better the sample will be, and accordingly the better the predictions. Although 44% accuracy is still very impressive, **65%** is phenomenal result that appears to be bought with more computation. The figure below summarizes the results from the run when M=1,000:

Digit	Correctly Predicted	Total	Accuracy (%)	% of training set (60 total)
0	6	14	42.9	10
1	4	9	44.4	3
2	2	11	18.2	8
3	3	6	50.0	13
4	3	9	33.3	8
5	5	13	38.5	10
6	8	14	57.1	17
7	4	6	66.7	10
8	7	11	63.6	8
9	2	7	28.6	12

Computational Intensity

Computational intensity for deploying predictions was certainly a concern throughout this process. Ideally, not resorting to a component-wise approach reduced the computational intensity greatly, albeit our examination of this option greatly favored the component-wise implementation. In addition, we understand that resorting to a component-wise approach using a random scan can, in some circumstances, also reduce the total number of iterations needed. Nonetheless, the large dimensionality of our problem did not favor random scan given how, when randomly choosing from 65 parameters, it is likely that within a great number of iteration a single parameter may not get updated for possibly hundreds of iterations. Consequently, the systematic component-wise approach was the only feasible option. The downside to the systematic component-wise, however, is that if you try and run it for say 10,000 iterations, with 65 components, this leads to 6.5 million sub-iterations between all 10 digits. For each sub iteration the posterior likelihood must be calculated for the proposed point too, which is an intensive calculation in itself, so over the entire component-wise algorithm for both Metropolis and Independence Sampler takes about 3.5 hours.

We undertook several measures to try to optimize our implementation's running time. These are the four main adjustments we made:

- Reducing the number of calculations
- Reducing the use of R's **'dmvnorm'**
- Declaring vectors that track values, preventing memory allocation within our loops
- Using R's vectorization functionality instead of relying on for-loops

In regards to the reduction of the number of calculations, we decided to calculate the probability of accepting (An) at every iteration. In the first round of code all of $g(X), g(Y), q(X)$, and $q(Y)$ were calculated each iteration.

Ultimately, two of the measures above made the most dramatic difference: Using the **'apply'** function instead of for-loops, and reducing the number of calculations.

The method of reducing the number of calculations consisted of immediately calculating $g(X)$ and $q(X)$ once the starting value of X (β_0) was chosen. Then if Y was accepted, set $g(X) = g(Y)$, and similarly $q(X) = q(Y)$, since both $g(Y)$ and $q(Y)$ were already calculated. This way they would not need to be calculated again for the next iteration. Otherwise if the proposed point was not accepted, then $g(X)$ and $q(X)$ would stay the same. Essentially, it made it so $g(X)$ and $q(X)$ we already calculated going into each iteration, and only $g(Y)$ and $q(Y)$ needed to be calculated. What made the biggest difference, however, was using the 'apply' function instead of for-loops. R is incredibly slow when it comes to for-loops, and when each sub iteration involves running a for-loop to calculate a posterior likelihood the program moves slow. By implementing R's 'apply' the run time was drastically cut down. The effects are shown below:

measure	timer
Saving g(X) and q(X), Pre-declaring hlist as vector, Not using dmvnorm function, Using apply() instead of for-loops	3.60
Saving g(X) and q(X), Pre-declaring hlist as vector, Not using dmvnorm function	54.25
Pre-declaring hlist as vector, Not using dmvnorm function	105.98
None	108.44

As shown in the table, adding the measures mentioned above cut the computation time for 1000 iterations (no sub-iterations) down from 108 seconds to just over 3. This is monumental, given that even with these measures in place the algorithm still takes upwards of 3 hours to generate predictions. Without these measure, the algorithm would literally take days to generate predictions, even with a training set of only 60.

Final Thoughts

Monte Carlo methods are widely used in the estimation of difficult quantities. In our project we sought to improve the classification of hand-written digits through a Bayesian implementation, where we simulated over the possible parameters used in the estimation of the probability of a given digit receiving a label of interest.

Our reasoning, supported by simulations, pointed to a component-wise approach as the best implementation of the Monte Carlo methods we considered. A good sample could not be generated from either algorithm using a non-component-wise approach, as the curse of dimensionality would kick in and it would become impossible to achieve an acceptance rate above even just 1% without resorting to an unrealistic proposal function (e.g. normal with variance of 0.00001). The component-wise random-walk Metropolis produced good predictions, but the sample it produced left something to be desired as it was not mixing well at all. Even when the number of iterations was increase to give it time to converge it still should no signs of steadying out.

When comparing the algorithms there are a number of factors to consider: How well the sample emulates the target distribution, the accuracy of the predictions, and the computation time. Overall, the most important metric here is the quality of the sample. If the algorithm runs quickly, produces good predictions, but produces a bad sample then it is tough to say whether or not the predictions are being generated by fluke. If the sample is good, then we should expect the predictions to be good, and we would have a systematic approach to generating predictions that could be reproduced to get similar results.

Ultimately, the component-wise algorithms produced much better samples, leading to better predictions, when compared to their non-component-wise alternatives. What these component-wise implementations suffered in regards to very high run-time, they made up in quality. The Component-wise Metropolis and Independence Sampler algorithms produced samples of similar quality, but where the Metropolis took off was in the strength of its predictions.

Computationally, both component-wise algorithms took a similar amount of time to run (about 3.5 hours for 10,000 iterations per digit in scope). This amount of time is still just a fraction of what the non-component wise algorithms took, which could usually generate predictions within 10 minutes. However, it is quality and not quantity, and as such both component-wise approaches performed better than their non-component wise counterparts, so despite the waiting component-wise is the optimal approach for this problem. What was not attempted was a batch-wise approach, where subsets of components were updated at once (e.g., 2 parameters instead of just one). This would speed up computation, but again given the high dimensionality of the problem it may prove difficult to obtain a reasonable acceptance rate.

Appendix

1. Loading the datasets and formatting training and testing sets

–Independence Sampler and Component-Wise Flavors–

Loading the data

```
data = labels = NULL

for(j in 0:9){
  path = paste("~/Desktop/hw5digits/digit_train_",j,".txt", sep = '')### Update if running from own desktop
  X = read.csv(path, header = FALSE)
  data = rbind(data,X)
  labels = c(labels, rep(j, nrow(X)))
  print(paste(j, "-", sum(is.na(X))))
}

data = cbind(1,data) #Including Intercept

print(dim(data))
sum(is.na(data))
```

Function to draw samples

```
rsample <- function(n, balanced ,num = 3) {

  if(balanced ==T){ ## For balanced & single digit, not really using since I confirmed the algo is working

    targnum = data[labels == num,]  #Taking all desired numbers
    nottargnum = data[labels != num,] #Taking all not desired numbers

    Sam = sample(c(1:nrow(nottargnum)),700) #samping 700 not target numbers
    nottargnum = nottargnum[Sam,]
    Tdata = rbind(targnum, nottargnum)
    Tlabels = c(labels[labels == num],labels[labels != num][Sam])

    TInd = sample(1:1400, 60)
    X = Tdata[TInd,]
    y = Tlabels[TInd]

    Xleft = Tdata[-c(TInd),]
    yleft = Tlabels[-c(TInd)]

    y = ifelse(y == num,1,0)
    yleft = ifelse(yleft == num,1,0)

    TestInd = sample(1:length(yleft),n)

    Xtest = Xleft[TestInd,]
    Ytest = yleft[TestInd]
    return(list(X,y,Xtest[1:n,],Ytest[1:n]))

  } else if(balanced ==FALSE){

    TInd = sample(1:nrow(data),60)## 60 here is chosen as the training set amount
    X = data[TInd,]
    y = labels[TInd]
    Xleft = data[-c(TInd),]
    yleft = labels[-c(TInd)]

    TestInd = sample(1:length(yleft),n)

    Xtest = Xleft[TestInd,]
    Ytest = yleft[TestInd]

    while(length(unique(y)) != 10 | length(unique(Ytest)) != 10){ #Making sure each
    #digit is represented well

      TInd = sample(1:nrow(data),60)
      X = data[TInd,]
      y = labels[TInd]

      Xleft = data[-c(TInd),]
      yleft = labels[-c(TInd)]

      TestInd = sample(1:length(yleft),n)

      Xtest = Xleft[TestInd,]
      Ytest = yleft[TestInd]
    }
  return(list(X,y,Xtest[1:n,],Ytest[1:n]))
  }
}
```

Functions used

```
library(MASS)
library(mvtnorm)
library(gtools)

S = diag(65) #Variance matrix, dim = components of xi + intercept
mu = rep(0,65) #0 vector to start

ldvm <- function(b,s2){ return( -32.5*log(2*pi*s2)-sum(b^2)/(2*s2)) }

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }

logit = function(x) {1/(1+exp(-x))}

h = function(x,b) { logit(b %*% xnew) }

logg <- function(X,y,b){
  #ss is covariance matrix of betas
  tur <- cbind(X,y)
  out <- sum(apply(tur, 1, function(x) {(x[66]*log(logit(x[1:65] %*% b))) +
      (1-x[66])*(log(1-logit(x[1:65] %*% b)))}))
  out <- out + ldvm(b,1/2)
  return(out);
}
```

–Random-walk Metropolis–

```
#Loading the datasets
path <- "/Users/Balthazar/Desktop/Grad_School/COURSEWORK/Fall 2018/Monte_Carlo/Project/data/"

namer <- function(n,noun){
```

```r
  out <- paste0(path,"digit_", noun, "_",n,".txt");
  return(out)
}

digit_train_0 <- read.csv(namer(0,"train"),header=F);
digit_train_1 <- read.csv(namer(1,"train"),header=F);
digit_train_2 <- read.csv(namer(2,"train"),header=F);
digit_train_3 <- read.csv(namer(3,"train"),header=F);
digit_train_4 <- read.csv(namer(4,"train"),header=F);
digit_train_5 <- read.csv(namer(5,"train"),header=F);
digit_train_6 <- read.csv(namer(6,"train"),header=F);
digit_train_7 <- read.csv(namer(7,"train"),header=F);
digit_train_8 <- read.csv(namer(8,"train"),header=F);
digit_train_9 <- read.csv(namer(9,"train"),header=F);

digit_test_0 <- read.csv(namer(0,"test"),header=F);
digit_test_1 <- read.csv(namer(1,"test"),header=F);
digit_test_2 <- read.csv(namer(2,"test"),header=F);
digit_test_3 <- read.csv(namer(3,"test"),header=F);
digit_test_4 <- read.csv(namer(4,"test"),header=F);
digit_test_5 <- read.csv(namer(5,"test"),header=F);
digit_test_6 <- read.csv(namer(6,"test"),header=F);
digit_test_7 <- read.csv(namer(7,"test"),header=F);
digit_test_8 <- read.csv(namer(8,"test"),header=F);
digit_test_9 <- read.csv(namer(9,"test"),header=F);
```

2. Code for ML Logistic Regression

```r
tepp <- sample_n(train3mat,1200) #Random sample from training set
X <- tepp[,-66]; Y <- tepp[,66]; #Create data and labels

summary(glm(as.numeric(tepp[,66]) ~ 0 +
              as.matrix(tepp[,-66],nrow=1000,ncol=66), family = binomial("logit")))
```

---

3. Code for Independence Sampler

```r
ptm <- proc.time()
n = 10000 #Total number of runs
B = 1000 #Total burn in
m = 100 #Number in testing set
numdigits = 10 #Total number of digits
d = 65 #Total number of parameters
ARtracker = NULL

hold = rsample(m, balanced = FALSE) #Pulling the sample
X = hold[[1]] #Training obersvations
ay = hold[[2]] #Training labels
X_test = hold[[3]] #Testing observations
aY_test = hold[[4]] #Testing labels

#Stuff for keeping track
AR = rep(0,(numdigits-1))
Est = matrix(0, nrow = m, ncol = (numdigits-1))
Vfact = matrix(0, nrow = m, ncol = (numdigits-1))
```

```r
ST = matrix(0, nrow = m, ncol = (numdigits-1))
betaholer = list()
#Variance parameter
s2 = 1/10

for(dig in 0:(numdigits-1)) { ###Looping through each digit

  y = ifelse(ay == dig,1,0) #Labeling everything as success of failure
  Y_test = ifelse(aY_test == dig,1,0) #Labeling everything as success of failure
  betas = matrix(0, nrow = n, ncol = 65)
  hlist = rep(0,n)
  bX = mu
  numaccept = 0

  pX = logg(X,y,as.numeric(bX))
  qX = dmvnorm(bX,mu, s2*S, log = T)
  S = diag(65)

  for(i in 1:n){ ### Generating the sample of betas


    bY = rmvnorm(1,mu, s2*S)

    pY = logg(X,y,as.numeric(bY))
    qY = dmvnorm(bY,mu, s2*S, log = T)

    An = exp(pY + qX - pX - qY)

    U = runif(1)

    if(U < An){
      bX = bY
      numaccept = numaccept +1
      pX = pY
      qX = qY
    }
    ARtracker[i] = numaccept

    betas[i,] = as.numeric(bX)
  }

  betaholer[[dig+1]] = betas[,c(2,20,31,45)] # Saving beta 2,20,31,45
  AR[dig + 1] =numaccept/n
  for(l in 1:m) { ## Fitting the model for each observation in the testing set

    hlist = rep(0, n - B)
    xnew = as.numeric(X_test[l,])

    for(k in ((B+1):n)){
      hlist[(k - B)] = h(xnew, betas[k,])
    }
    ## Recording the info
    Est[l,dig+1] = mean(hlist)
    Vfact[l,dig+1] = varfact(hlist)
    ST[l,dig+1] = sd(hlist)
  }
}
```

```r
proc.time() - ptm
```

---

4. Code for Random-Walk Metropolis

```r
M=10000; B=M/10;
bl <- 65; #Number of betas
blist = matrix(0,nrow=M,ncol=bl); #To keep track of betas

#Let X be training matrix
trainer <- sample_n(train3mat,400);
X <- trainer[,-66]; Y <- trainer[,66];

tester <- sample_n(test3mat,400);
Xnew <- trainer[,-66]; Ynew <- trainer[,66];

betas <- numeric(length(X[1,]));

acc <- 0; #Initialize Acceptance Counts
scale1 <- 1/1000; #Proposal scaling
scale2 <- 1; #Move scaling

#Initializing betas from overdidpersed starting dist
#1. ss <- identity matrix, meaning sampling from standard MVN equivalent to sampling
#from 65 indep standard normals
ss <- diag(bl);
betaold <- rnorm(bl); #basic prior

#2. ss <- sample cov matrix
#ss <- scale1*cov(X) #Covariance matrix of training set
#betaold <- mvrnorm(1,mu=rep(0,bl),ss);

for (r in 1:10){
  #Above is classification of the first 10 digits in the testing set
  hlist = NULL
  betaold <- mvrnorm(1,mu=rep(0,bl),ss);
  acc = 0

  for (i in 1:M){
    U <- runif(1);
    #1. Sigma: I
    betanew <- betaold + scale2 * mvrnorm(1,mu=rep(0,bl),ss);
    #2. Sigma: Sample cov
    #betanew <- betaold + sigma *     mvrnorm(length(betaold),rep(0,length(betaold)),ss);

    #cat("betanew: ",as.numeric(betanew),"\n");
    #cat("betaold: " ,as.numeric(betaold),"\n");
    #cat("logg(X,Y,betanew,ss): ", logg(X,Y,as.numeric(betanew),ss),"\n")
    #cat("logg(X,Y,betaold,ss): ", logg(X,Y,as.numeric(betaold),ss),"\n")

    if (log(U) < (logg(X,Y,betanew,ss) - logg(X,Y,betaold,ss)) ) {
      betaold = betanew; #Make move if accepted per above
      acc = acc + 1;
    }
    blist[i,] <- betanew; #Save parameters
    hlist[i] <- h(as.numeric(Xnew[r,]), as.numeric(betaold)) #Save classif prob
  }

  u = mean(hlist[(B+1):M])
  se1 = sd(hlist[(B+1):M])/sqrt(M-B)
  se = se1*sqrt(varfact(hlist[(B+1):M]))

  cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
  cat("acceptance rate =", acc/M, "\n");

  cat("Obs",r,"Actual Label:",Ynew[r],"\n")
  cat("Estimate:",u,"\n")
  cat("Varfact:",varfact(hlist[(B+1):M]),"\n")
  cat("95% C.I:", u + c(-1,1)*1.96*se,"\n")
  plot(hlist[(B+1):M],type="l")
  #plot(blist[(B+1):M,1], type="l")
}
proc.time() - ptm #Time-keeper
```

5. Code for Component-Wise Independence Sampler

```
n = 10000 #Total number of runs
B = 1000 #Total burn in
m = 100 #Number in testing set
numdigits = 10 #Total number of digits
d = 65 #Total number of parameters
ARtracker = NULL

hold = rsample(m, balanced = FALSE) #Pulling the sample
X = hold[[1]] #Training obersvations
ay = hold[[2]] #Training labels
X_test = hold[[3]] #Testing observations
aY_test = hold[[4]] #Testing labels

AR = rep(0,(numdigits-1))
Est = matrix(0, nrow = m, ncol = numdigits)
Vfact = matrix(0, nrow = m, ncol = numdigits)
ST = matrix(0, nrow = m, ncol = numdigits)
betaholer = list()
s2 = 1/26

for(dig in 0:(numdigits-1)) { ###Looping through each digit

  if(dig == 7){
    s2 = 1/30
  } else {
    s2 = 1/26
  }
  cat(dig,"\n")
  y = ifelse(ay == dig,1,0) #Labeling everything as success of failure
  #Y_test = ifelse(aY_test == dig,1,0) #Labeling everything as success of failure
  betas = matrix(0, nrow = d*n, ncol = 65)
  hlist = rep(0,d*n)
  bX = mu
  bY = bX
  numaccept = 0
  pY = qY = 0
  pX = logg(X,y,as.numeric(bX))
  qX = log(dnorm(0,0,s2))

  for(i in 1:n){ ### Generating the sample of betas

    for(o in 1:65){##Using Systematic

      bY[o] = rnorm(1,mu,s2)
      pY = logg(X,y,as.numeric(bY))
      qY = log(dnorm(bY[o],0,s2))
      An = exp(pY + qX - pX  - qY)

      U = runif(1)

      if(U < An){
          bX = bY
          numaccept = numaccept +1
          pX = pY
          qX = qY
      }

      betas[d*(i-1) + o,] = as.numeric(bX)
      }
    }

  betaholer[[dig+1]] = betas[,c(2,20,31,45)]# Saving beta 2,20,31,45
  AR[dig + 1] =numaccept/n
  for(l in 1:m) { ## Fitting the model for each observation in the testing set

    hlist = rep(0,(d*n - d*(B+1)))
    xnew = as.numeric(X_test[l,])

    for(k in (d*(B+1)):(d*n)){
      hlist[(k - d*B)] = h(xnew, betas[k,])
    }
    ## Recording the info
    Est[l,dig+1] = mean(hlist)
    Vfact[l,dig+1] = varfact(hlist)
    ST[l,dig+1] = sd(hlist)
  }

}
proc.time() - ptm
```

---

6. Code for Component-Wise Metropolis

```
n = 1000 #Total number of runs
B = 200 #Total burn in
m = 100 #Number in testing set
numdigits = 10 #Total number of digits
d = 65 #Total number of parameters
ARtracker = NULL

hold = rsample(m, balanced = FALSE) #Pulling the sample
```

```
X = hold[[1]] #Training obersvations
ay = hold[[2]] #Training labels
X_test = hold[[3]] #Testing observations
aY_test = hold[[4]] #Testing labels

#Objects to keep track of stuff
AR = rep(0,(numdigits-1))
Est = matrix(0, nrow = m, ncol = numdigits)
Vfact = matrix(0, nrow = m, ncol = numdigits)
ST = matrix(0, nrow = m, ncol = numdigits)
betaholer = list()
s2 = 1/4

for(dig in 0:(numdigits-1)) { ###Looping through each digit

  cat(dig,"\n")
  y = ifelse(ay == dig,1,0) #Labeling everything as success of failure
  #Y_test = ifelse(aY_test == dig,1,0) #Labeling everything as success of failure
  betas = matrix(0, nrow = d*n, ncol = 65)
  hlist = rep(0,d*n)
  bX = mu
  bY = bX
  numaccept = 0

  pY = qY = 0
  pX = logg(X,y,as.numeric(bX))
  qX = log(dnorm(0,s2))

  for(i in 1:n){ ### Generating the sample of betas

    for(o in 1:65){

      bY[o] = rnorm(1,bX[o],s2)
      pY = logg(X,y,as.numeric(bY))
      An = exp(pY  - pX)

      U = runif(1)

      if(U < An){
          bX = bY
          numaccept = numaccept +1
          pX = pY
          qX = qY
      }

      betas[d*(i-1) + o,] = as.numeric(bX)
      }
    }


  betaholer[[dig+1]] = betas[,c(2,20,31,45)]# Saving beta 2,20,31,45
  AR[dig + 1] =numaccept/n
  for(l in 1:m) { ## Fitting the model for each observation in the testing set

    hlist = rep(0,(d*n - d*(B+1)))
    xnew = as.numeric(X_test[l,])

    for(k in (d*(B+1)):(d*n)){
      hlist[(k - d*B)] = h(xnew, betas[k,])
    }
    ## Recording the info
    Est[l,dig+1] = mean(hlist)
    Vfact[l,dig+1] = varfact(hlist)
    ST[l,dig+1] = sd(hlist)
  }

}
proc.time() - ptm
```

7. Code for Evaluation of Predictions

```
Est1 = Est
Means = apply(Est1, 2,mean)
Est2 = apply(rbind(Est1, Means),2, function(x) x[1:100]/x[101])
#Est2 = Est1
Est3 = apply(Est2, 1, function(x) x[1:10]/sum(x[1:10]))
Est3 = t(Est3) #Transposed it for some reason
Estimate = apply(Est3,1,which.max)-1
D = data.frame(est = Estimate, Actual = aY_test)
sum(D$est == D$Actual)
```

```
rhmean = NULL
for(q in 1:length(hlist)){
  rhmean[q] = mean(hlist[1:q])
}



par(mfrow = c(2,1))
plot(hlist, type = 'l',main = 'Testing observation 100, for digit 9', ylab = "Prediction at Index")


plot(rhmean, type = 'l',main = 'Testing observation 100, for digit 9', ylab = "Running Mean")
```