Applying Machine Learning to the Genome

Daniel Kang

Introduction

Chromatin has been recognized as a vital factor in genomic regulation [1]. Thus, understanding the grammatical structure underlying chromatin state has been one of the primary goals in computational biology. My project aims to construct a simple grammatical model of chromatin structure as a type of regression problem. This approach unifies the ideas behind sequence bias correction, motif-finding, event detection and genome segmentation. In contrast to prior work, my rigorous approach to problem setup allows me to fit a simple model with no tuning parameters.

Both sequence bias correction and motif-finding algorithms fall under the broad category of sequence to phenotype prediction problems, but to my knowledge all algorithms have focused upon the enrichment of particular k-mers or motifs within selected regions [2]. Instead, I focus upon the effect that each k-mer induces upon nearby bases, which will allow me to produce experimentally testable predictions as well as automatically tune the complexity of the model using held-out data.

The cis-regulatory model will allow us to answer questions such as: what regulates the binding of transcription factors and what is the relative importance of epigenetic information such as chromatin marks; what determines changes to chromatin structure, and what effects do variation in DNA-sequence have on observed phenotype.

Computationally, I will construct a state of the art optimizer for sparse L_1 regularized Poisson regression using optimization techniques such as accelerated gradient descent combined with map-reduce type cloud computing techniques to allow for model fitting on a genome-wide scale.

Model

The genome can be viewed as a vector of *bases* (characters). Every position i in the genome has an associated *k-mer* (a string of *k* bases starting at position *i*), for every natural number *k* less than the size of the genome. We view the strings as integers and we consider $k = 8$ in this report.

The current model I am analyzing can be abstracted as follows. Denote the counts observed in a high-throughput sequencing experiment as $c \in [0, \infty)^N$, where $c(i)$ denotes the counts at position i and N is the size of the genome. Denote the genome viewed as k-mers as g, where $g(i,\ k) \in \{1...4^k\}$ denotes the k-mer of length k starting at position i. Denote the set of parameter matrices as υ. For fixed k, we define a matrix $\upsilon^k \in R^{4^k \times 2W+1}$, where W is the fixed window size we are considering. There are 4^k k-mers of length k, which is why $\upsilon^k \in R^{4^k \times 2W+1}$. Denote for a given k-mer, ${\upsilon^k}_{g(i,k)} \in R^{2W+1}$, which is the local effect that k-mer has.

Formally, the we write the log-Poisson rate at which we observe high-throughput sequencing reads as

$$\lambda_i = \sum_{j=i-W}^{i+W} \upsilon^k{}_{g(i,k)}[j] - x_0 .$$

The log-likelihood per base is given as

$$LH_i = c_i \lambda_i - exp(\lambda_i) .$$

The objective function which we minimize is a combination of a goodness of fit term with a L_1 penalty term:

$$-\sum_{i=1}^{N} LH_i + \eta(\sum_{k=1}^{8} \sum_{i=1}^{4^k} \sum_{j=1}^{2W+1} |\upsilon^k{}_i[j]|) .$$

Inference method

The objective function we minimize is globally and strictly convex, but not smooth. However, we can apply proximal methods to smooth the problem, which will allow the use of gradient descent to minimize the transformed objective function. Globally convex functions have a global minimum, with no local minima. Gradient descent works well because 1) it is guaranteed to converge to the global minima, 2) it is simple to implement and converges quickly.

The general gradient descent method is as follows:

1. Set the parameter vector v to 0.
2. Repeat until convergence:
    a. Evaluate the gradient dv_i at v_i.
    b. Update the parameter vector using the linear approximation: $v_{i+1} = v_i + \varepsilon\, dv_i$

We can compute the gradient as follows. First, compute the derivative of the log-likelihood as

$$dLH_i = c_i - exp(\lambda_i - x0) .$$

The gradient of the parameter matrix is

$$d\upsilon^k{}_l[j] = \sum_{i=1}^{N} I_l^k(i + j - 200)\, dLH_i$$

where $I_l^k(i)$ is an indicator for k-mer l, i.e. $I_l^k(i)$ is 1 if $g(i, k) = l$ and 0 otherwise.

Implementation

We use chromosomes 1 to 11, which is approximately half the genome or one billion bases (i.e. $N \approx 10^9$); half the data was held out to evaluate the algorithm.

Initial experiments show that the iterative algorithm works well on synthetic and real data, with convergence taking approximately 200-500 iterations. However, computing the gradient serially takes over 30 minutes when the code is written in Python for synthetic data sets 1/10 the size of actual data sets (one chromosome, roughly 200 thousand bases). The rest of the computation is negligible compared to the gradient computation. Thus, the computation must be optimized for realistic applications; the naive implementation cannot leverage the information in the entire genome.

Nesterov's method, proximal gradient methods, and trust region methods can be used to decrease the number of iterations for convergence, but the gradient step itself must be optimized to achieve realistic runtimes.

Parallelization

Because speed was of paramount importance, I implemented the algorithm in C++ and MPI, using Amazon EC2 for scale. MPI operates by spawning *M* processes, each of which have an independent memory layout and communicate by passing messages. MPI's scheme has the side effect of child processes on the same physical machine being unable to share memory access.

The gradient computation can be decomposed into two separate computations. Since we are optimizing the log-likelihood, we first compute the exponential of the sum of parameters per base. Then, we compute the sum of the exponentials. Specifically, we compute λ_i first, then compute dLH_i. This transformation turns the problem into two easily parallelizable subtasks and also increases cache efficiency.

The parallel algorithm is as follows (with MPI operations in parentheses):

1. Send the necessary data from master to slaves (bcast).
2. Set the parameter vector v to 0.
3. Repeat until convergence:
    a. Send the current parameter vector to the slaves (bcast).
    b. Each slave computes the gradient on a subset of the genome.
    c. The slaves send the gradient back to master, which then computes the full gradient dv_i (reduce).
    d. Master updates the parameter vector using the linear approximation: $v_{i+1} = v_i + \varepsilon\, dv_i$

As seen in figure 1, the naive parallelization scales well. Compared to the approximately 300 minutes per iteration of the NumPy implementation, the parallelized gradient takes about 28 seconds per iteration with 16 machines. Thus, by parallelization, we achieve almost a 650 fold speedup.

Figure two plots the inverse gradient time against the number of processes. As seen, the R^2 is approximately 0.85. The genome can be split evenly over the processes, which explains the near linear growth. However, the amount of communication increases with the number of physical machines, so the gains are lost to communication overhead as the number of physical machines increase.

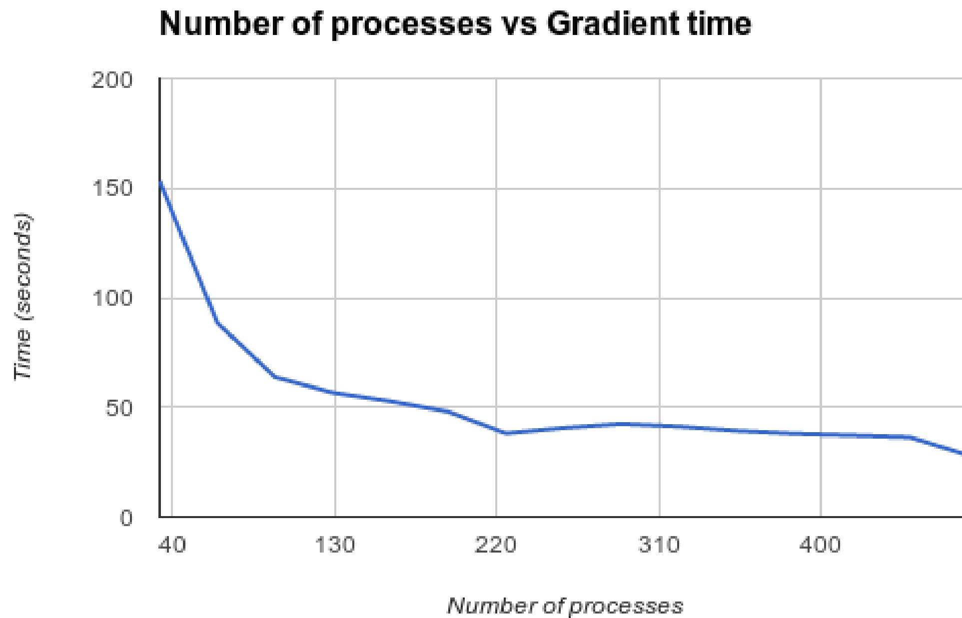

Figure 1. This figure plots the number of processes against the time it takes for a single gradient step. We used the c3.8xlarge Amazon EC2 instance which had 32 processes per physical machine, so the plot ranges from two to 16 machines.

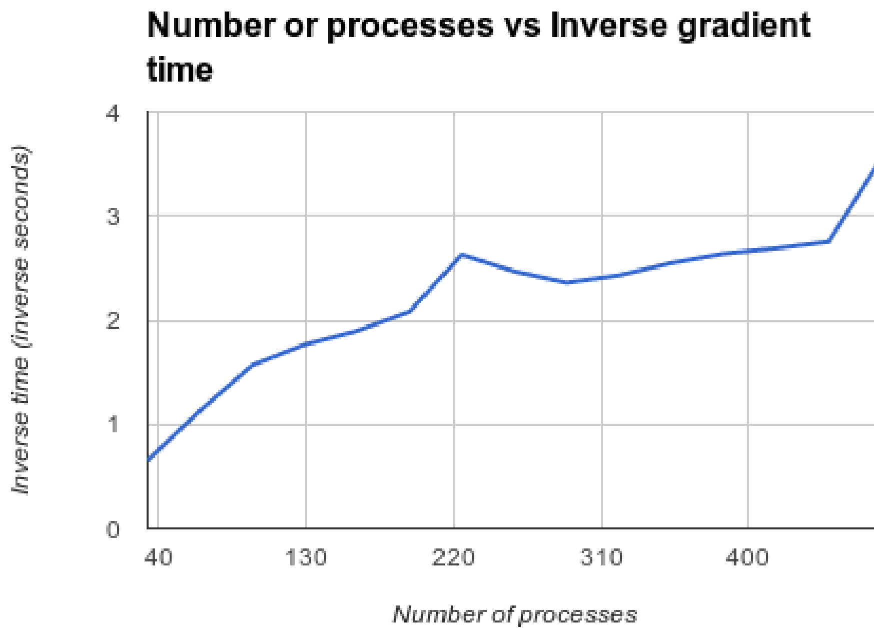

Figure 2. This figure plots the number of processes against the inverse gradient time, scaled for easier visual fit. The R^2 is approximately 0.85, indicating the gradient time decreases inversely with the number of processes.

Conclusion

My initial version of the algorithm was implemented in pthreads on a single machine. pthreads not only has a shared memory model, but since it runs on a single machine, there is little to no communication overhead. However, to my knowledge, pthreads does not scale across multiple physical machines so I had to switch to the MPI framework. My initial MPI versions of the algorithm also did not have communication overhead issues, since it was compute-bound. After optimizing the MPI version, the communication became the bottleneck. Using multiple physical machines requires network communication, which is extremely slow. One of the most important things I learned from this project were the tradeoffs between communication and compute time, and the need to switch between regimes to fully reap the benefits of parallelization.

One of the most important steps in the algorithm is to have the master update the parameter vector of the slaves. I initially updated the parameter matrix on the master node and broadcasted the result to the slaves using MPI's bcast method. However, bcast does not take into account that multiple MPI processes may be running on the same machine. Broadcasting once from the master to each physical machine, then to the local processes nearly gave a two-fold speedup in some cases. I was especially surprised since MPI's reduce command appears to use the same scheme that I implemented.

I believe the sheer scale of the problem was one of the reasons genome-wide cis-regulatory models have not been analyzed before. As mentioned above, the naive Python implementation took 300 minutes per iteration. I believe as problems in computational biology scale beyond a single genome (e.g. to studying populations instead of single individuals), parallel computation will become a vital tool.

Future Work

The framework I constructed should be general enough to solve a large class of globally and strictly convex problems. I am considering turning my code into a gradient descent module, as I may study problems requiring similar techniques later.

For reasonable values of *k*, the parameter matrix can 10 million elements or larger. They are represented by floating point numbers, so the cost of broadcasting the parameter matrix eventually overwhelms the gains of parallelization. However, the communication overhead can be lowered via "shotgun" and block descent methods, which descend on only a subset of the parameter matrix at a time.

Shotgun methods leverage global convexity, which guarantees descending on subsets of the parameters will converge, given that all parameters are eventually descended on [3]. Descending on a subset of parameters only requires that subset to be sent to the slaves, which dramatically decreases the amount of communication necessary for a gradient step.

Block coordinate descent also leverages global convexity, but it decomposes the problem into separable components. Thus, it may be possible to analytically update a subset of the parameters at a time.

I will investigate shotgun and block coordinate descent methods to further optimize my framework.

References

[1] Mendenhall, E. M., & Bernstein, B. E. (2008). Chromatin state maps: new technologies, new insights. *Current opinion in genetics & development*, *18*(2), 109-115.
[2] van Heeringen, S. J., & Veenstra, G. J. C. (2011). GimmeMotifs: a de novo motif prediction pipeline for ChIP-sequencing experiments. *Bioinformatics*,*27*(2), 270-271.
[3] Schmidt, M., Roux, N. L., & Bach, F. (2011). Convergence rates of inexact proximal-gradient methods for convex optimization. *arXiv preprint arXiv:1109.2415*.