ABERYSTWYTH UNIVERSITY
COMPUTER SCIENCE AND STATISTICS (GG34)
CS396: MINOR PROJECT

Application of Machine Learning Techniques to Next Generation Sequencing Quality Control

Author:
Sam Nicholls msn

Supervisor:
Dr. Amanda Clare afc

Draft
April 13, 2014

Declaration

I certify that except where indicated, all material in this thesis is the result of my own investigation and references used in preparation of the text have been cited. The work has not previously been submitted as part of any other assessed module, or submitted for any other degree or diploma.

Sam Nicholls
2014

Abstract

Over the past few years advances in genetic sequencing hardware have introduced the concept of massively parallel DNA sequencing; allowing potentially billions of chemical reactions to occur simultaneously, reducing both time and cost required to perform genetic analysis[10]. However, these "next-generation" processes are complex and open to error[6], thus quality control is an essential step to assure confidence in any downstream analyses performed.

During sample sequencing a large number of quality control metrics are generated to determine the quality of the reads from the sequencing hardware itself. At the Wellcome Trust Sanger Institute, the automated QC system currently relies on hard thresholds to make such quality control decisions with individual hard-coded values on particular metrics determining whether a lane has reached a level that requires a warning, or has exceeded the threshold and failed entirely. Whilst this does catch most of the very poor quality lanes, a large number of lanes are flagged for manual inspection at the warning level; a time consuming task which invites inefficiency and error.

In practise most of these manual decisions are based on inspecting a range of diagnostic plots which suggests that a machine learning classifier could potentially be trained on the combinations of quality control statistics available to make these conclusions without the need for much human intervention.

Contents

Chapter 1

Introduction

Over the past few years advances in genetic sequencing hardware have introduced the concept of massively parallel DNA sequencing; allowing potentially billions of chemical reactions to occur simultaneously, reducing both time and cost required to perform genetic analysis[10]. However, these "next-generation" processes are complex and open to error[6], thus quality control is an essential step to assure confidence in any downstream analyses performed.

1.1 Project Aims

The project consists of two sub-projects;

- Analysis of a current quality control system in place
- Identification of quantifiable sample properties that affect downstream analysis

1.1.1 Analysis of Current System

With the support of the Wellcome Trust Sanger Institute in Cambridge, this project works with the Human Genetics Informatics team to investigate **auto_qc**, the institute's current automated quality control tool.

During genetic sequencing a large number of metrics are generated to determine the quality of the data read from the sequencing hardware itself. As part of the current vertebrate sequencing pipeline[1] at the institute, **auto_qc** is responsible for applying quality control to samples within the pipeline by comparing a modest subset of these metrics to simple hard-coded hard thresholds; determining whether a particular sample has reached a level that requires a warning, or has exceeded the threshold and failed entirely. Whilst this does catch most of the very poor quality outputs, a large number of samples are flagged for manual inspection at the warning level; a time consuming task which invites both inefficiency and error.

In practise most of these manual decisions are based on inspecting a range of diagnostic plots which suggests that a machine learning classifier could potentially be trained on the combinations of quality control statistics available to make these conclusions without the need for much human intervention.

The first part of the project aims to apply machine learning techniques to replicate the current **auto_qc** rule set by training a decision tree classifier on a large set of these quality metrics. The idea is to investigate whether these simple threshold based rules can be recovered from such data, or whether a new classifier would produce different rules entirely. During this analysis it is hoped the classifier may be able to identify currently unused quality metrics that improve labelling accuracy. An investigation on the possibility of aggregating or otherwise reducing the dimensions of some of the more detailed quality statistics to create new parameters will also be conducted.

The goal is to improve efficiency of quality control classification, whether by improving accuracy of pass and fail predictions over the current system or merely being able to provide additional information to a lab technician inspecting samples labelled with a warning to reduce arbitrary decisions.

1.1.2 Identification of Properties that affect Downstream Analysis

The other half of this project is motivated by the question "What *is* good and bad in terms of quality?"

To be able to classify samples as a pass or a fail with understanding, we need an idea of what actually constitutes a good or bad quality sample and must look at the effects quality has on analysis performed downstream from sequencing. An example of such is **variant calling** — the process of identifying differences between a DNA sample (such as your own) and a known reference sequence.

Given two high quality data sources where DNA sequences from individuals were identified in two different ways (one of which being next-generation sequencing) it would be possible to measure the difference between each corresponding pair. Using this, we could investigate the effect of leaving out part of the next-generation sample during the variant calling process. If we were to leave a part of a sample out of the variant calling pipeline would the variants found be more (or less) accurate than if it had been included? Would they agree more (or less) with the variants called after using the non next-generation sequencing method?

Having identified such sub-samples, can quality control metrics from the previous part be found in common? If so, such parameters would identify "good" or "bad" samples straight out of the machine! Samples that exhibit these quality variables will go on to improve or detriment analysis.

1.2 Project Methodology

Clearly some team-based practices invited by agile methodologies — pair programming immediately comes to mind — are not applicable in a solo project. It is also unreasonable to expect an "on-site" customer for this particular project. In *The Case Against Extreme Programming*, Matt Stephens describes a "self referential safety net" where the perceived traps in each practice are supported and "made safe" by other XP practices. This would seem to rule out XP as a viable methodology for a solo-project as cutting out some of the

processes that allow this form of evolutionary design to work (and flatten that cost-of-change curve) can introduce serious flaws to the management of a project and potentially result in failure. In the same breath it is important to remember that not all agile processes need be discarded just because XP seems incompatible. Indeed, some processes seem like common sense, for example; frequent refactoring, simple design, continuous integration and version control. Test driven development could also prove a useful process to consider as part of a methodology for this project, setting up a framework that allows for quick and frequent testing (before coding) and ensuring that any refactoring has a positive (or at least non-negative) effect on the system could be a worthwhile contribution to efficiency.

Could a more plan driven approach or form of agile-plan hybrid be considered appropriate here? In *Balancing Agility and Discipline*, Boehm and Turner introduce the idea of "homegrounds" for both agile and plan driven approaches; I note here that for projects that require high reliability and feature a non-collocated "CRACK customer" in fact align with some of these homegrounds for plan driven development. Combined with the thought that the project requirements will also be relatively stable it would seem that there may be no reason to switch to a more agile methodology as its primary feature is the welcoming of change that is not even needed? Perhaps this is the naivety of an optimist!

Personally I think I would approach this with a form of agile-plan hybrid; I like the idea of quick iterations and getting feedback as opposed to leaving acceptance testing until the very end of the project, but I also want a somewhat detailed feedback process. Whilst through Neil Taylor's course *Agile Methodologies* it was suggested that it is dangerous to pick and choose processes (don't anger the Ring of Snakes!) and also merely paying "lip service" to agile must be avoided (otherwise what's the point?), I feel that on this occasion it can be justified by the size of the project itself.

This project will consist of many research steps, each requiring some form of computational process to prepare the data for the next step. Whilst the implementations of the algorithms themselves pose computational complexity, there appears to be little challenge from a planning perspective and in fact a looser overall plan should probably be considered as we must account for unforseen and unexpected outcomes from each research step.

The most important part of ensuring this project stays on track will be the development of a sensible testing methodology to ensure we are not only moving in the right direction in terms of which algorithm and parameters to use but also in terms of reliably measuring performance over time in a way that allows justification of such design choices.

Despite this trail of thought, given the research grounding this project entails it might be required to look beyond traditional and even modern software development methodologies and investigate a more scientific approach. A simple "scientific method" would involve establishing a null hypothesis that can be proven false by testing (eg: "**auto_qc** classifier is more accurate than the new classifier") and executing experiments that attempt to prove this null hypothesis false in favour of an alternative hypothesis (typically the opposite, eg: "The new classifier is more accurate than the old classifier"). This form of hypothesis testing could essentially become the project's acceptance tests (providing we have an empirical definition of what "more accurate" means in terms of this system) and any modification can be classed as an experiment ("Do these parameters allow us to reject the null hypothesis?"). Although care must be taken not to let this descend into unstructured cycles of mere hack-and-test, code-and-fix style programming.

Overall it is rather difficult to select a methodology for a project such as this, the research element makes it almost impossible to draw on previous personal experience for ideas of what development processes may or may not be effective.

1.2.1 Task and Time Management

Shortly before embarking on this dissertation project, I had written my own todo list web application; **Triage**.

1.2.2 Time Considerations

It must be remembered that this is supposed to be a minor project completed alongside the study of other modules that have their own assignments...

Chapter 2

Analysis of Current System

2.1 Introduction

This part of the project can be outlined as follows:

- Collect data sets on which a machine learning classifier is to be trained
- Construct a program capable of processing and storing such data sets such that required subsets of the data can be quickly and easily returned for further analysis
- Select a suitable machine learning framework to handle the training and validation of a classifier
- Ensure a robust validation methodology exists for assuring quality of our own results
- Set up an environment capable of allowing results from such a classifier to be stored and compared
- Training a suitable classifier on the collected data sets
- Perform experiments by selecting subsets of the variables and observations and measure whether classification accuracy is improved

2.2 Materials and Methods

2.2.1 Input Data and Format

As part of the project I've been granted access to significant data sets at the Sanger Institute, unlocking quality control data for two of the largest studies currently undergoing analysis. A wide array of quality metrics are available for each and every lanelet that forms part of either of the two studies; totalling 13,455 files.

The files are created by **samtools stats** — part of a collection of widely used open-source utilities for post processing and manipulation of large alignments such as those produced by next-generation sequencers that

are released under the umbrella name of "SAMtools"[5] (Sequence Alignment and Map Tools). **samtools stats** collects statistics from sequence data files and produces key-value summary numbers as well as more complex tab delimited dataframes tabulating several metrics over time.

The output of **samtools stats** is then parsed by an in-house tool called **bamcheckr**, named so as **samtools stats** was once known as **bamcheck** and the tool is written in R. **bamcheckr** supplements the summary numbers section of the **samtools stats** output with additional metrics that are later used by **auto_qc** for classification. This process does not change the file other than adding a few additional key-value pairs in the summary numbers section. A truncated example of a "bamcheckr'd" file can be found in Appendix A.

2.2.2 Development Environment

Language

For the language of the program designed to handle this vast array of input data, Python was selected, more out of personal taste rather than a detailed analysis of required performance and features. From previous experience I was happy with the performance of Python when processing large datasets in terms of both I/O file handling operations and storing the data in memory for later use. Python's generous choice of both built-in and third-party libraries have proven useful on many occasions. Due to its concise and flexible nature it is possible to rapidly develop applications and its readability eases ongoing maintenance; useful given the short time-span allocated for this project and the possibility of others wishing to contribute to the project codebase after completion.

Whilst the choice was made primarily on preference, this is not to say other options were not considered: a highly popular Java-based collection of data mining tools, **WEKA**[3] would certainly have provided a framework for building decision tree classifiers but at the same time did not appear to offer any significant features that were unavailable elsewhere, whilst Java itself has the added constraint of requiring a virtual machine to be installed which could be undesirable from a performance or even security standpoint when the application is deployed to servers at the Sanger Institute.

Difficulty was also encountered finding example implementations for **WEKA** with most documentation and tutorials providing information for performing analysis via the graphical "Explorer" interface instead, which would not be appropriate for quickly setting up and repeating experiments automatically.

Given the quality data we'll be using to train a machine learning classifier is output from the previously mentioned R script; **bamcheckr**, it was worth briefly investigating the options available for R itself as the potential of integrating the learning and predicting functions right in to the same process that outputs the data seemed convenient.

...Whilst the **tree**[9] and **rpart**[11] packages are available for constructing decision trees in R but (and actually **RWeka** provides an R interface to **WEKA**)...they did not appear to be as robust as other more well-known frameworks. Putting it politely, the programming paradigm of R is rather different to other languages and can significantly increase development time if one is not very well versed in the patterns and grammar of the language and it seemed best to stick to one's comfort zone.

...C and C++ also a possibility, however **dlib** didn't support tree-based classifiers although **Shark** did, Python chosen in the end for ease of use...

Framework

...Having studied the Machine Learning module in final year the prospect of getting stuck in to the deep of a machine learning algorithm was exciting, however the reality is a lot of time and effort has gone in to proper optimisation of a framework which is unlikely to be surpassed successfully by a short-term one-person project. It is therefore only natural that a library seems a wise investment for the codebase...

...There are numerous machine learning frameworks available in many languages, some described above... however it seemed counter-intuitive to select a framework in another language for the introduction of an additional arbitrary output and input steps to switch between the two environments.

...A mixed bag of such frameworks exist in Python, two in particular **scikit-learn**[8] and **Orange**[2] were main contenders, partly on their recommendation from the project supervisor...

...scikit integrates the "big names" in Python: numpy, scipy and matplotlib ...put off from Orange due to difficulties in reading in data ...it did however later appear to ship with features that were not in scikit (pruning and printing) ...with more time I'd certainly like to investigate using other libraries such as **Orange** or even outside Python and take at look at WEKA or Shark...

Libraries

numpy[7] and scipy[4]... Fast and reliable implementations of mathematical functions... ggplot2[12] for beautiful graphing...

Testing

Investigated the possibility of using **Jenkins** but just didn't have the time to set up and tweak all the plugins and options of the platform to do what I wanted... Would still have been poor in terms of searching through logs to track what parameters led to improvements in cross validation... ...Quite likely I'd have needed to author a Java or Groovy plugin to keep good track of cross-validation...

Briefly considered online solutions such as **Travis** and **Wercker** which I use for some personal projects, whilst quicker to set up would probably have not been able to handle artifacts such as dot files without some convoluted solution of uploading them to dropbox or adding them to a private git repository from the build node... ...also would have needed to upload a very large quantity of training data repeatedly which would be inefficient (and more than likely against reasonable use of the platform)

Really wanted to write my own solution for this but had to settle for well formatted log files that could be searched and processed with some command line fu...

Additional Tools

Version control is critical, **git**!

2.3 Pre-Implementation

2.3.1 Classification Correlation

...An important consideration for statistical analysis is the relation between the observations. The quality data files we have are per lanelet, however a lane can house more than one lanelet... and so herein lies the trouble, if during a run, the flow cell is somehow subjected to abnormal temperatures (air conditioning failure) or the sequencing device is depleted of reagents, every lane (and thus lanelet within) will be of very poor quality. Thus there would appear to exist a relationship between the respective qualities of each lanelet in a lane as well as each lane in a sequencing run!

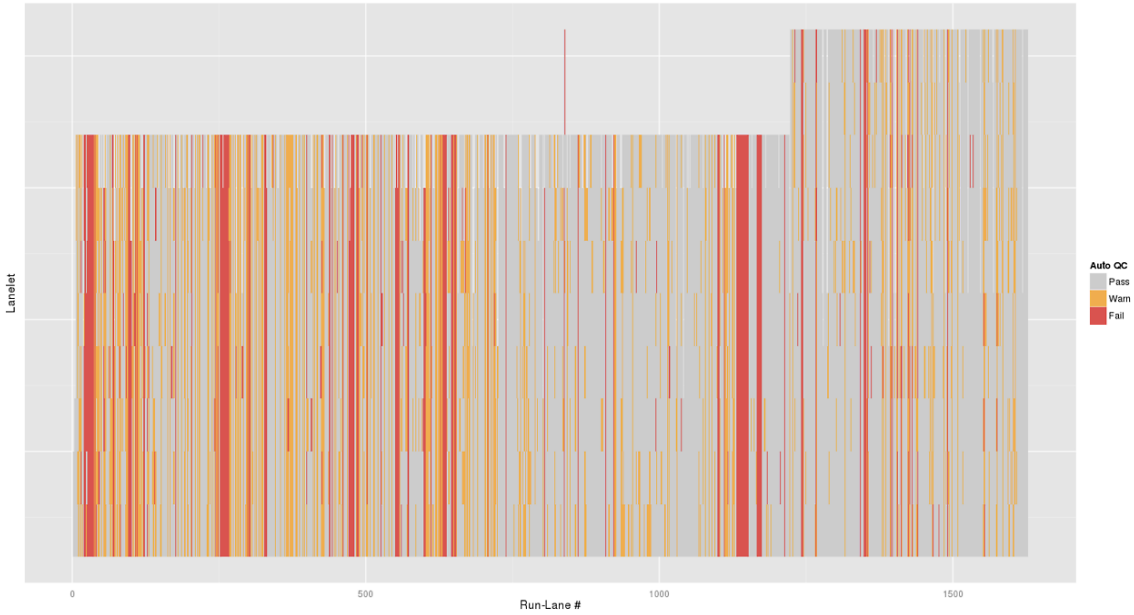

Fig. 2.1 Heatmap of Lanelet QC Status by Lane

...Figure 2.1 displays a plot of auto_qc classification for each lanelet (y) versus lane (x). ...The plot was designed to be a diagnostic test to see whether this was the case. I should mention that it seems there are few conditions under which a lanelet would fail irrespective of the auto_qc status of the rest of the lanelets in the lane; which mostly involve the preparation of the sample (which is easy to spot given it will cause poor quality across all lanelets using that library sample).

...an unbroken vertical red bar indicates that all the lanelets inside a particular lane failed. Likewise yellow represents a warning. Grey areas are passes and were desaturated to make the other outcomes immediately

obvious. It is clear that there are patches where lanelets have failed where entire lanes have not, but there does appear to be some correlation.

Having discussed this with the supervisor and contacts at the Sanger Institute we decided to continue...

Note the plot does not make a particular distinction between lanes in the same flow cell but they are sequentially identified so the red bars of thicker-width arguably display some failures across entire flow cells.

2.4 Implementation

2.4.1 Frontier

Frontier is the programmatic output for this part of the project... providing an API-like interface to the data itself... ...allowing simple commands to pull the data out from memory in to formats acceptable by the machine learning framework...

...provides a class to read from the "bamcheck files" as seen in Appendix A...

...a major refactoring to remove hard-coded classes from **Frontier** which enable it to be used as a more general purpose tool; if we were to add another class label, the definition would merely need to be included to the CLASSES (Listing 1) variable passed when the Statplexer is constructed. But use is therefore not merely limited to our problem but rather any machine learning problem where you'd like to simplify your interactions which a very large dataset.

```
CLASSES = {
        "pass": {
            "class": ["pass"],
            "names": ["pass", "passed"],
            "code": 1,
        },
        "fail": {
            "class": ["fail"],
            "names": ["fail", "failed"],
            "code": -1,
        },
        "warn": {
            "class": ["warn"],
            "names": ["warn", "warning"],
            "code": 0,
        },
}
```

Listing 1 Class definitions for auto_qc as passed to Frontier

Cross Validation

...K fold cross validation ...stratified cross validation...

Confusion Matrices

"Normal" confusion matrix and "Warnings" confusion matrix...

2.4.2 Contributions to bamcheckr

```
install.packages("devtools")
library(devtools)

# Install directly from github repository
install_github("samstudio8/seq_autoqc", subdir="bamcheckr")

# Install from local directory
install("/home/sam/Projects/seq_autoqc/bamcheckr")
```

Takes 5.5s on average, 16.1s with ratio due to inefficient implementation for overlapping_base_duplicate_percentage re-wrote in Python...

...R CMD BATCH issue

...Fixed a graph plotting failure. ...Writing additional routines...

2.4.3 SelectKBest

* incorect degrees of freedom * warnings: /usr/lib64/python2.7/site-packages/sklearn/feature_selection/univariate_selection.py:2 RuntimeWarning: invalid value encountered in divide, causing NaN * Replaced univariate_selection with version from master * needed use force np.float64 ...actually data was 0... gg :(

2.5 Results

2.5.1 Initial Trees

2.5.2 Parameter Sets

Chapter 3

Identification of Qualitative Sample Properties

3.1 Introduction

This part of the project can be outlined as follows:

- ...

3.1.1 Why Goldilocks?

Having recovered from the miscommunication that led me to attempt to find reverse strands in VCF files with already known errors, I've been making performance improvements to the script that finds candidate genomic regions.

The task poses an interesting problem in terms of complexity and memory, as the human genome is over three billion bases long in total which can easily lead to data handling impracticalities.

For the next step of the project we're looking to document what effects quality has on analysis that occurs downstream from sequencing, for example; variant calling - the process of identifying bases in a sample that differ from the reference genome (this is a little simple as you also need to discern as whether the difference is actually a polymorphism or not).

To investigate this I'll be performing leave-one-out analysis on the whole-genome data we have; consisting of leaving a lanelet out, performing variant calling and then comparing the result of the left-one-out variant call and the corresponding variant call on our "SNP chip" data.

The basic idea is to answer the question of what actually constitutes "good" or "bad" lanelet quality? Does leaving out a lanelet from the full-sequence data lead to variant calls that better match the SNP chip data, or

cause the correspondence between the two sets to decrease? In which case, having identified such lanelets, can we look back to the quality parameters we've been analysing so far and find whether they have something in common in those cases?

If we can, these parameters can be used to identify "good" or "bad" lanelets straight out of the machine! We know that lanelets that exhibit these quality variables will go on to improve or detriment analysis.

However, variant calling is both computationally and time intensive. Whilst the Sanger Institute have significant computing resources available, my dissertation has an end date and with the time I have we must focus the leave-one-out analysis on a subsection of the whole genome.

It is for this reason we're looking for what Josh at Sanger described as a "representative autosomal region". The candidate must not have too many variants, or too few; a sort of "Goldilocks genome".

3.2 Materials and Methods

3.2.1 Input Data: Variant Call Format

```
# Install
git clone htslib
git clone bcftools
make #(requires htslib to be above samtools/bcftools dir)
sudo cp bcftools usr/local/bin

# Tabix
# Download from sourceforge
# http://sourceforge.net/projects/samtools/files/tabix/
make
sudo cp tabix /usr/local/bin

# Generate an index (vcfidx) (not necessary)
vcftools --gzvcf cd.ichip.vcf.gz

# Query VCF (can be done with awk, cut etc.)
vcf-query cd.ichip.vcf.gz -f '%CHROM:%POS\t%REF\t%ALT\n'

# ! A faster alternative to using vcftools exists in bcftools ! #
bcftools query -f '%CHROM:%POS\t%REF\t%ALT\n' cd-seq.vcf.gz > cd-seq.vcf.gz.q
# Complains about lack of tabix index but still makes the file...

# Index with tabix
```

```
# "The input data file must be position sorted and compressed by bgzip
# which has a gzip(1) like interface"
tabix -p vcf file.vcf.gz
diff cd-seq.vcf.gz.tbi diff cd-seq.vcf.gz.tbi.sanger
# Shows no difference! :)
# http://samtools.sourceforge.net/tabix.shtml

# http://vcftools.sourceforge.net/perl_module.html
# http://www.biostars.org/p/51076/
# http://vcftools.sourceforge.net/htslib.html#query
```

3.3 Implementation

3.3.1 Goldilocks

As I discussed previously, on the hunt for my Goldilocks genomic region, it's important to consider both time and memory as it is simple to deliver a poor performing solution.

Searching for candidates regions over the entire genome at once is probably unwise. Luckily, since our candidate must not span chromosomes (the ends of chromosomes are not very good for reads) then we can easily yield a great improvement from processing chromosomes individually.

The process is to extract the locations of all the variants across the genome from the SNP chip VCF files (these files list the alleles for each detected variant for each sample), load them in to some sort of structure, "stride" over each chromosome (with some stride offset) and finally list the variants present between the stride start and stride start plus the desired length of the candidate. These are our regions!

Due to the use of striding, one does not simply walk the chromosome! A quick and dirty solution would be to just look up variants in a list:

```
for chromosome in autosomes:
    for start in range(1, len(chromosome), STRIDE):
        for position in range(start, start+LENGTH):
            if position in variant_position_list:
                # Do something...
```

This of course is rather dumb. Looking up a list has O(n) performance where n is the number of variants (and there are a lot of variants). Combining this with the sheer number of lookups performed; with the default parameters the list would be queried half a billion times for the first chromosome alone.

You could improve this somewhat by dividing the variant_position_list in to a list of variants for each chromosome, so at least n is smaller. It is pretty doubtful that this would make any useful impact given the number of lookups.

I'm happy to say I bypassed this option but thought it would be fun to consider it's performance.

A far more sensible solution would be to replace the list with a dictionary whose lookup is amortized to a constant time. The number of lookups is still incredibly substantial but a constant lookup is starting to make this sound viable. Using the variant positions as keys of a dictionary (in fact let's use the previous minor suggestion and have a dictionary for each chromosome) we have:

```
for chromosome in autosomes:
    for start in range(1, len(chromosome), STRIDE):
        for position in range(start, start+LENGTH):
            if position in variant_position_dict[chromosome]:
                # Do something...
```

Great! This still takes half an hour to run on my laptop though, surely there is a better way! I wondered if whether dropping the lookup would improve things. Instead, how about an area of memory is allocated to house the current chromosome and act as a variant "mask"; essentially an array where each element is a base of the current chromosome and the value of 1 represents a variant at that position and a 0 represents no variant.

```
for chromosome in autosomes:
    chro = np.zeros(len(chromosome), np.int8)
    for variant_loc in snps_by_chromosome[chromosome]:
        chro[variant_loc] = 1

    for start in range(1, len(chromosome), STRIDE):
        for position in range(start, start+LENGTH):
            if chro[position] == 1:
                # Do something...
```

We of course have to initially load the variants in to the chromosome array, but this need only be done once (per chromosome) and is nothing compared to the billions of lookups of the previous implementation.

Rather to my surprise this was slow, very slow (maybe if I have time I should check how it compared to looking up with a list). Was it the allocation of memory? Perhaps I had run out of RAM and most of the work was going in to fetching and storing data in swap?

I switched out the comparison (== 1) step for the previous dictionary based lookup and the performance improved considerably. What was going on? There must be more to looking at a given element in the numpy chromosome array, but what?

After a brief spell of playing with Python profilers and crashing my laptop by allocating considerably more memory than I had with numpy.zeroes, I read the manual (!) and discovered that numpy.zeroes returns an ndarray which uses "advanced indexing" even for single element access, effectively copying the element to a Python scalar for comparison.

That's a lot of memory action!

It then occurred to me that we're interested in just how many variants are inside each region and our chromosome is handily encapsulated in a numpy array! Why don't we just sum together the elements in each region? Remember variant positive base positions are 1 and 0 otherwise, useful! So the work boils down to some clever vector mathematics calculated by numpy!

We don't even lose any detail because the actual list of variants inside a region can be recovered in a small amount of time just given the start and end position of the region.

```
for chromosome in autosomes:
    chro = np.zeros(len(chromosome), np.int8)
    for variant_loc in snps_by_chromosome[chromosome]:
        chro[variant_loc] = 1

    for start in range(1, len(chromosome), STRIDE):
        num_variants = np.sum(chro[start:start+LENGTH])
        # Do something...
```

With conservative testing this runs at least 60 times faster than before, the entirety of the human genome can be analysed for candidate regions in less than twenty seconds (with the first few seconds taken reading in all the variant locations in the first place)!

Ignoring empty regions etc.

```
bsub -o ../../../goldilocks/joblog/samtools_mpileup.%J.o -e ../../../goldilocks/joblog/samtool
```

Memory Leaks...

Testing

...particularly difficult!

References

[1] vr-pipe, a generic pipeline system [Github]. [Online]. Available: https://github.com/wtsi-hgi/vr-pipe/

[2] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data Mining Toolbox in Python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html

[3] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[4] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[5] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and G. Subgroup, " The Sequence Alignment/Map format and SAMtools. ," *Bioinformatics*, vol. 25, p. 2078, 2009.

[6] M. Kircher, U. Stenzel and J. Kelso, "Improved base calling for the Illumina Genome Analyzer using machine learning strategies," *Genome Biology*, vol. 10, no. 8, p. R83, 2009.
Useful introduction to relevant Illumina hardware and the errors that can occur during sequencing.

[7] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
A machine learning framework for Python.

[9] B. Ripley, *tree: Classification and regression trees*, 2014, r package version 1.0-35. [Online]. Available: http://CRAN.R-project.org/package=tree

[10] T. Strachan and A. Read, *Human Molecular Genetics*, 4th ed. Garland Science, 2011, pp. 214–254.
A concise introduction to the processes involved in massively parallel DNA sequencing.

[11] T. Therneau, B. Atkinson, and B. Ripley, *rpart: Recursive Partitioning*, 2013, r package version 4.1-3. [Online]. Available: http://CRAN.R-project.org/package=rpart

[12] H. Wickham, *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. [Online]. Available: http://had.co.nz/ggplot2/book 978-0-387-98140-6.

Appendix A

samtools stats example output

```
# Summary Numbers. Use ‘grep ^SN | cut -f 2-‘ to extract this part.
SN	raw total sequences:	41400090
SN	filtered sequences:	0
SN	sequences:	41400090
SN	is paired:	1
SN	is sorted:	1
SN	1st fragments:	20700045
SN	last fragments:	20700045
SN	reads mapped:	41291484
SN	reads unmapped:	108606
SN	reads unpaired:	60000
SN	reads paired:	41231484
SN	reads duplicated:	5756822
SN	reads MQ0:	1038644
SN	reads QC failed:	0
SN	non-primary alignments:	0
SN	total length:	3105006750
SN	bases mapped:	3096861300
SN	bases mapped (cigar):	3090885143
SN	bases trimmed:	0
SN	bases duplicated:	431761650
SN	mismatches:	9107833
SN	error rate:	0.002946675
SN	average length:	75
SN	maximum length:	75
SN	average quality:	36
SN	insert size average:	178.7
SN	insert size standard deviation:	44.1
SN	inward oriented pairs:	20577242
SN	outward oriented pairs:	3140
SN	pairs with other orientation:	3711
SN	pairs on different chromosomes:	31535
SN	fwd percent insertions above baseline:	1.43135383851191
SN	fwd percent insertions below baseline:	0.686265539012562
SN	fwd percent deletions above baseline:	1.38326380878871
SN	fwd percent deletions below baseline:	0.44923551909251
SN	rev percent insertions above baseline:	1.08264446659241
SN	rev percent insertions below baseline:	0.457290262062496
SN	rev percent deletions above baseline:	1.15931214598243
SN	rev percent deletions below baseline:	0.413119424753248
SN	contiguous cycle dropoff count:	36
SN	fwd.percent.insertions.above.baseline:	1.43135383851191
```

```
SN	fwd.percent.insertions.below.baseline:	0.686265539012562
SN	fwd.percent.deletions.above.baseline:	1.38326380878871
SN	fwd.percent.deletions.below.baseline:	0.44923551909251
SN	rev.percent.insertions.above.baseline:	1.08264446659241
SN	rev.percent.insertions.below.baseline:	0.457290262062496
SN	rev.percent.deletions.above.baseline:	1.15931214598243
SN	rev.percent.deletions.below.baseline:	0.413119424753248
SN	quality.dropoff.fwd.high.iqr.start.read.cycle:	0
SN	quality.dropoff.fwd.high.iqr.end.read.cycle:	0
SN	quality.dropoff.fwd.high.iqr.max.contiguous.read.cycles:	0
SN	quality.dropoff.fwd.mean.runmed.decline.start.read.cycle:	20
SN	quality.dropoff.fwd.mean.runmed.decline.end.read.cycle:	51
SN	quality.dropoff.fwd.mean.runmed.decline.max.contiguous.read.cycles:	32
SN	quality.dropoff.fwd.mean.runmed.decline.high.value:	36.9775883578997
SN	quality.dropoff.fwd.mean.runmed.decline.low.value:	36.301749247405
SN	quality.dropoff.rev.high.iqr.start.read.cycle:	0
SN	quality.dropoff.rev.high.iqr.end.read.cycle:	0
SN	quality.dropoff.rev.high.iqr.max.contiguous.read.cycles:	0
SN	quality.dropoff.rev.mean.runmed.decline.start.read.cycle:	18
SN	quality.dropoff.rev.mean.runmed.decline.end.read.cycle:	56
SN	quality.dropoff.rev.mean.runmed.decline.max.contiguous.read.cycles:	39
SN	quality.dropoff.rev.mean.runmed.decline.high.value:	36.1517621338504
SN	quality.dropoff.rev.mean.runmed.decline.low.value:	35.3152133727245
SN	quality.dropoff.high.iqr.threshold:	10
SN	quality.dropoff.runmed.k:	25
SN	quality.dropoff.ignore.edge.cycles:	3
SN	A.percent.mean.above.baseline:	0.0991164444444441
SN	C.percent.mean.above.baseline:	0.127379555555556
SN	G.percent.mean.above.baseline:	0.0603679999999997
SN	T.percent.mean.above.baseline:	0.0868000000000005
SN	A.percent.mean.below.baseline:	0.0991164444444451
SN	C.percent.mean.below.baseline:	0.127379555555555
SN	G.percent.mean.below.baseline:	0.0603680000000002
SN	T.percent.mean.below.baseline:	0.0867999999999993
SN	A.percent.max.above.baseline:	0.601733333333332
SN	C.percent.max.above.baseline:	0.394266666666667
SN	G.percent.max.above.baseline:	0.2956
SN	T.percent.max.above.baseline:	0.768000000000001
SN	A.percent.max.below.baseline:	0.318266666666666
SN	C.percent.max.below.baseline:	0.825733333333332
SN	G.percent.max.below.baseline:	0.554400000000001
SN	T.percent.max.below.baseline:	0.251999999999999
SN	A.percent.max.baseline.deviation:	0.601733333333332
SN	C.percent.max.baseline.deviation:	0.825733333333332
SN	G.percent.max.baseline.deviation:	0.554400000000001
SN	T.percent.max.baseline.deviation:	0.768000000000001
SN	A.percent.total.mean.baseline.deviation:	0.198232888888889
SN	C.percent.total.mean.baseline.deviation:	0.254759111111111
SN	G.percent.total.mean.baseline.deviation:	0.120736
SN	T.percent.total.mean.baseline.deviation:	0.1736
# First Fragment Qualitites. Use ‘grep ^FFQ | cut -f 2-‘ to extract this part.
# Columns correspond to qualities and rows to cycles. First column is the cycle number.
FFQ	1	8968	3619	9863	747	5094	0	6642	1609	4673	4208	2
FFQ	2	21676	0	0	0	0	0	0	0	43	1885	0	0	0
FFQ	3	7	0	177	0	0	0	0	0	0	0	0	0	0
FFQ	4	0	0	0	0	65	0	0	0	272	0	0	0	14277
FFQ	5	1917	173	1249	0	1890	0	0	0	0	10874	0	0
[...]
FFQ	72	4098	0	0	4806	0	0	0	65507	0	0	0	0
FFQ	73	3894	2	0	0	0	0	4931	53483	0	0	0	0
FFQ	74	3697	39	0	919	4933	0	0	56866	1524	0	0	2
FFQ	75	4542	0	0	0	0	0	4634	77822	0	0	0	0
FFQ	76	0	0	0	0	0	0	0	0	0	0	0	0	0
```

```
# Last Fragment Qualitites. Use ‘grep ^LFQ | cut -f 2-‘ to extract this part.
# Columns correspond to qualities and rows to cycles. First column is the cycle number.
LFQ	1	8869	0	0	0	0	0	63	0	0	1156	616	173
LFQ	2	3300	0	0	0	0	0	0	0	0	0	389	0	0
LFQ	3	6816	0	0	0	573	0	83	0	7011	1171	107134	0
LFQ	4	5492	0	0	13	0	66	730	708	8134	2422	84052
LFQ	5	3512	0	0	0	1023	185	0	8653	1995	0	115559
[...]
LFQ	72	5135	166	0	0	2872	13643	0	59649	4249	11351	346
LFQ	73	6025	229	0	86	1042	13417	0	66093	3741	8151	354
LFQ	74	5980	3	91	0	0	0	1340	9696	72939	4924	304090
LFQ	75	4314	0	0	168	0	848	8591	0	70358	3827	352180
LFQ	76	0	0	0	0	0	0	0	0	0	0	0	0	0
# Mismatches per cycle and quality. Use ‘grep ^MPC | cut -f 2-‘ to extract this part.
# Columns correspond to qualities, rows to cycles. First column is the cycle number, second
# is the number of N’s and the rest is the number of mismatches
MPC	1	14078	0	2594	6777	416	1919	0	2222	352	987	537
MPC	2	21407	0	0	0	0	0	0	0	0	5	223	19	0
MPC	3	3205	0	0	37	0	43	0	12	0	691	71	6984
MPC	4	1774	0	0	0	2	29	4	65	73	863	192	6749
MPC	5	1913	0	94	885	0	969	23	0	959	213	1203	84
[...]
MPC	72	361	0	13	0	573	276	934	0	16376	426	1066
MPC	73	1005	0	11	0	4	79	777	539	15025	363	699	3
MPC	74	779	0	3	0	131	440	0	93	7485	6589	387	24
MPC	75	136	0	0	0	3	0	47	704	9302	5886	260	2650
MPC	76	0	0	0	0	0	0	0	0	0	0	0	0	0
# GC Content of first fragments. Use ‘grep ^GCF | cut -f 2-‘ to extract this part.
GCF	0.5	56
GCF	1.76	60
GCF	3.02	126
GCF	4.27	212
GCF	5.78	347
[...]
GCF	93.72	378
GCF	95.23	186
GCF	96.48	87
GCF	97.74	55
GCF	99.25	17
# GC Content of last fragments. Use ‘grep ^GCL | cut -f 2-‘ to extract this part.
GCL	0.5	118
GCL	1.76	175
GCL	3.02	230
GCL	4.27	354
GCL	5.78	525
[...]
GCL	93.72	613
GCL	95.23	430
GCL	96.48	274
GCL	97.74	185
GCL	99.25	110
# ACGT content per cycle. Use ‘grep ^GCC | cut -f 2-‘ to extract this part. The columns are: cycle, and A,C,G,T counts [%]
GCC	1	26.93	23.09	22.77	27.2
GCC	2	26.78	23.24	22.97	27.02
GCC	3	26.46	23.59	23.3	26.66
GCC	4	26.29	23.79	23.45	26.46
GCC	5	26.47	23.61	23.3	26.62
[...]
GCC	70	26.09	24.26	23.45	26.2
GCC	71	26.07	24.25	23.46	26.22
GCC	72	26.04	24.27	23.49	26.2
GCC	73	26.07	24.25	23.47	26.22
GCC	74	26.08	24.24	23.45	26.23
```

```
GCC	75	26.01	24.31	23.51	26.18
# Insert sizes. Use ‘grep ^IS | cut -f 2-‘ to extract this part. The columns are: pairs total, inward oriented pairs, outward oriented pa
IS	0	10	0	1	9
IS	1	3	0	3	0
IS	2	4	0	4	0
IS	3	5	0	5	0
IS	4	2	0	2	0
IS	5	3	0	3	0
[...]
IS	110	33952	33952	0	0
IS	111	38433	38433	0	0
IS	112	43373	43370	0	3
IS	113	48160	48159	0	1
IS	114	53175	53171	0	4
IS	115	59504	59502	0	2
IS	116	64668	64668	0	0
IS	117	71107	71105	0	2
IS	118	77157	77156	0	1
IS	119	84044	84044	0	0
IS	120	90116	90110	3	3
[...]
IS	327	6546	6546	0	0
IS	328	6483	6483	0	0
IS	329	6201	6201	0	0
IS	330	6228	6228	0	0
IS	331	5852	5852	0	0
# Read lengths. Use ‘grep ^RL | cut -f 2-‘ to extract this part. The columns are: read length, count
RL	75	41400090
# Indel distribution. Use ‘grep ^ID | cut -f 2-‘ to extract this part. The columns are: length, number of insertions, number of deletions
ID	1	128650	183418
ID	2	26409	39770
ID	3	10213	16046
ID	4	7756	11444
ID	5	1746	3455
[...]
ID	35	0	8
ID	36	0	1
ID	37	0	1
ID	38	0	1
ID	40	0	2
# Indels per cycle. Use ‘grep ^IC | cut -f 2-‘ to extract this part. The columns are: cycle, number of insertions (fwd), .. (rev) , numbe
IC	1	0	0	105	97
IC	2	24	15	150	179
IC	3	129	138	441	509
IC	4	253	310	623	829
IC	5	557	724	786	1164
[...]
IC	70	571	710	638	761
IC	71	350	428	309	434
IC	72	154	150	38	45
IC	73	60	61	15	23
IC	74	20	19	11	12
# Coverage distribution. Use ‘grep ^COV | cut -f 2-‘ to extract this part.
COV	[1-1]	1	332980694
COV	[2-2]	2	105004580
COV	[3-3]	3	29112182
COV	[4-4]	4	13415014
COV	[5-5]	5	6716815
[...]
COV	[996-996]	996	2
COV	[997-997]	997	2
COV	[998-998]	998	2
COV	[1000-1000]	1000	4
```

```
COV	[1000<]	1000	116
# GC-depth. Use ‘grep ^GCD | cut -f 2-‘ to extract this part. The columns are: GC%, unique sequence percentiles, 10th, 25th, 50th, 75th a
GCD	0	0.001	0	0	0	0	0
GCD	0.4	0.002	0.101	0.101	0.101	0.101	0.101
GCD	19	0.003	0.049	0.049	0.049	0.049	0.049
GCD	20	0.004	0.06	0.06	0.06	0.06	0.06
GCD	21	0.004	0.045	0.045	0.045	0.045	0.045
[...]
GCD	66	99.99	0.244	2.693	6.746	11.794	15.885
GCD	67	99.994	1.279	1.279	4.305	9.667	11.483
GCD	68	99.997	4.148	4.148	4.463	5.741	7.354
GCD	69	99.999	0.499	0.499	0.499	1.935	1.935
GCD	72	100	0.476	0.476	0.476	1.219	1.219
```