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 26, 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

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[1]. However, these "next-generation" processes are complex and open to error[2], 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**[3], 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 institute's vertebrate sequencing pipeline[4], **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 thresholds; determining whether a particular sample has reached a level that requires a warning, or has exceeded the threshold and failed entirely. Whilst this catches 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 practice most of these manual decisions are based on inspecting a range of diagnostic plots, which suggest 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[5].

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 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 Method

1.2.1 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 extreme programming (XP) practices. This would 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 are 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 as 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; noting 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 end of the project, but I also want a somewhat detailed feedback process. In Neil Taylor's *Agile Methodologies* course 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 be considered as we must account for unforeseen 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 (*e.g.* "**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, *e.g.* "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 as the research element makes it almost impossible to draw on previous personal experience for ideas of what development processes would be effective.

1.2.2 Task Management

It is useful to be able to keep track of current tasks preferably via a medium that would allow some method of sorting and filtering. For various personal projects and the second year group project I have used **Redmine**, a Ruby-based web application designed for bug tracking. However over time I have come to find keeping the task information stored in Redmine a task in itself. Attempts to extend the platform to implement additional functionality have been fruitless.

Out of frustration with thus and other alternatives – including **TaskWarrior** whose simple but effective command line interface was overshadowed by its occasional storage corruptions, I wrote my own open source web based task management application; **Triage**[6]. This will be useful in keeping track of current objectives and allow prioritisation in a quick and simple manner.

The list used to organise my project is also publicly available[7] for transparent progress tracking by my supervisor and those interested at the Sanger Institute.

1.2.3 Time Considerations

It must be remembered that this project needs to meet the requirements for a *minor* project and is to be completed alongside the study of several other modules which each have their own assignments and obligations. It would be easy to become overly ambitious and thus aims and goals will need to be revised as both obstacles and breakthroughs are encountered over the lifetime of the project.

Part I

Analysis of Current System

Chapter 2

Introduction and Background

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 Concepts and Terminology

2.2.1 Whole Genome Sequencing

...the process of recovering the sequence of bases called nucleotides...

2.2.2 Samples, Lanes and Lanelets

A **sample** is a distinct DNA specimen extracted from a particular person. For the purpose of sequencing, samples are pipetted in to a *flowcell* such as the one in Figure 2.1 – a glass slide containing a series of very thin tubules known as **lanes**. It is throughout these lanes that the chemical reactions involved in sequencing take place.

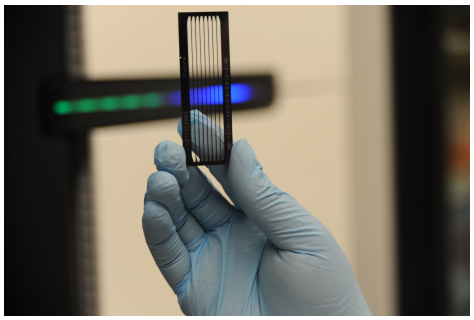

Fig. 2.1 An Illumina HiSeq Flowcell[8]

Once inserted, samples are amplified *in situ*, in the flowcell itself. A process in which the genetic material of each sample is caused to multiply in magnitude to form a dense cluster of the sample around the original. Millions of clusters will be created throughout each lane of the flowcell.

Note that a lane can contain more than one sample and a sample can appear in more than one lane; this is known as *sample multiplexing* and helps to ensure that the failure of a particular lane does not hinder analysis of a sample (as it will still be sequenced as part of another lane).

The more abstract of the definitions, a **lanelet** is the aggregate read of all clusters of a particular sample in a single lane. Figure 2.2 attempts to highlight examples of a this (circled in blue – not all lanelets are highlighted). For example Lane 5 shows the four clusters (in reality there would be millions) of Sample A combine to represent a lanelet. A lane will have as many lanelets as it does samples.

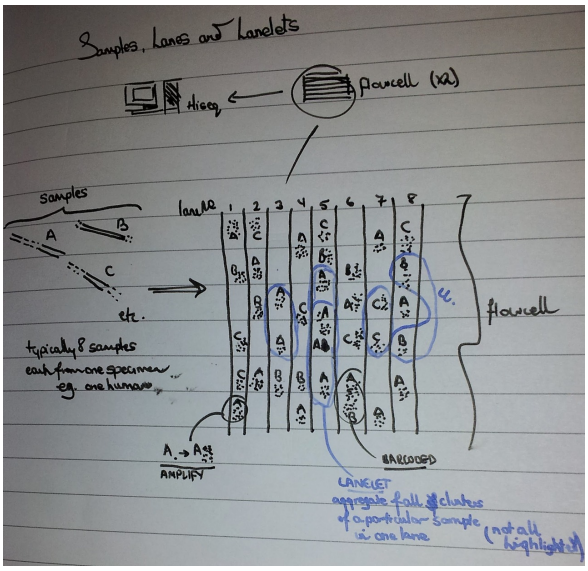

Fig. 2.2 Example of flowcell with some lanelets highlighted

Chapter 3

Materials and Methods

3.1 Input Data and Format

3.1.1 "BAMcheckR'd" Data

As part of the project I have 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"[9] (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**[1] which supplements the summary numbers section of the **samtools stats** output with additional metrics that are later used by **auto_qc** for classification. This process appends additional key-value pairs in the summary numbers section. A truncated example of a "bamcheckr'd" file can be found in Appendix A.1.

It is these summary numbers that will be the main focus of our learning task.

3.1.2 auto_qc Decision Data

To use these "bamcheckr'd" files for training and testing a machine learning classifier, it is necessary to map each file to a classification result from **auto_qc**. The one-to-one mapping between each input file and its label

[1] Named such as **samtools stats** now incorporates **bamcheck** and the tool is written in R

are provided by the Sanger Institute in a separate file hereafter referred to as the *AQC Decision Matrix* or *AQC (Decision) File*.

A truncated example of such a file can be found in Appendix A.2. Only the first few columns are included – indeed we are only interested in the *lanelet* and *aqc* which provide an identifier that maps the row to a given input file and its classification by **auto_qc** respectively. Latter columns pertain to a breakdown of decisions made by **auto_qc** which are not included in the example for confidentiality (and brevity).

3.2 Development Environment

3.2.1 Language

Python was selected for the language of the program designed to handle this vast array of input data, 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 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. 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**[10] would certainly have provided a framework for building decision tree classifiers but 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 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**[11] and **rpart**[12] packages are available for constructing decision trees in R (and actually **RWeka** provides an R interface to **WEKA**) neither appeared to be as robust as other more well-known frameworks. Also putting it politely, the programming paradigm of R[13] is rather different to other languages and can significantly increase development time (and frustration[14]) if one is not well versed in the patterns and grammar of the language and it seemed best to stick to one's comfort zone given the brief timescale for the project.

Had performance been a critical decision factor, lower level languages such as C, C++ or even Fortran could be

used. Briefly looking at two popular frameworks available for C in particular; **dlib** did not support tree-based classifiers although an alternative, **Shark** did.

3.2.2 Framework

Having studied the *Machine and Intelligent 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 cumulative time and effort has gone in to creation and optimisation of a framework which is unlikely to be surpassed successfully by a short-term one-person project. Thus utilisation of a third party machine learning library would seem a wise investment for the project's codebase.

There are numerous machine learning frameworks available in many languages, some of which were described above and formed part of the development environment decisions. Whilst it is obviously unnecessary to select a framework which uses the same language as the project, it seemed counter-intuitive to select otherwise, for the establishing of additional arbitrary output and input steps to move data between the two environments could impede quick experiment repeatability and introduce error.

...A mixed bag of machine learning frameworks exist in Python, two in particular **scikit-learn**[15] and **Orange**[16] 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...

3.2.3 Additional External Libraries

numpy[17] and scipy[18]... Fast and reliable implementations of mathematical functions... ggplot2[19] for beautiful graphing...

3.2.4 Testing

As discussed in Chapter 1.2.1, testing forms a critical part of the project given the need to monitor the impact of changes to classification accuracy as well as to ensure the program is working correctly. Ideally, execution of a test suite should be simple and easily repeatable. Results that pertain to accuracy should also be stored for future reference to monitor ongoing performance of the classifier.

Such requirements could be fulfilled by a continuous integration platform – a server dedicated to the building and testing of the code contained in a centralised repository typically to which an entire team will have write access[20]. Whilst in this scenario there will be much less "risk" from integration issues due to the single person team size, the themes of automated building and self-testing code can be taken on board.

Jenkins is a highly popular[21] example of such a platform with which I am familiar. Although an out-of-the-box Jenkins instance is suitable for variety of software engineering projects, it would be necessary to invest some time to install and tweak plugins to perform actions on test results (such as failing a build that causes accuracy to decrease). However, previous experience found that highly specific tasks will often require a plugin to be authored to overcome limitations in the feature set of a more generic plugin, which given the intricacies of the Jenkins package layout could easily turn in to a project of its own. Unfortunately, other features that would be useful to the project including the indexing and searching of build logs are somewhat lacking in Jenkins.

Online solutions such as **Travis** and **Wercker** could potentially offer a quicker set up as both merely requires a small configuration file in the root of the repository and a hook to be registered... ...however such services would not have 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 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...

3.2.5 Tools

Version control is critical, **git**

Chapter 4

Pre-Implementation

4.1 Classification Correlation

An important consideration for statistical analysis is the relation between observations. The "bamcheckr'd" input data described in Chapter 3.1.1 is available per lanelet, however as shown in Chapter 2.2.2 a lane may contain more than one lanelet. Herein lies the trouble: if during a sequencing run the flowcell is somehow subjected to abnormal conditions (*e.g.* a temperature increase due to an air conditioning failure) or the device is depleted of reagents then every lane (and thus all lanelets within) will be of considerably poor quality.

Fig. 4.1 **Heatmap of lanelet QC status by lane**: Lanes are vertical bars with each lanelet cell coloured red to represent a failure, yellow for a warning and grey for a pass.

In such a case 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. To examine this further, an R script utilising **ggplot2** was authored to visually inspect whether correlation existed and if so to what degree that data is affected.

Figure 4.1 displays a plot of **auto_qc** classification for each lanelet in a lane. The plot itself is a dense heatmap where each lane stands as a vertical bar, broken in to horizontal cells, each of which represents a lanelet that was sequenced in that particular lane. These lanelets are colour coded using; red for failures, yellow for warnings and grey for passes (to allow the other two classes to be more easily seen).

Therefore an unbroken vertical red line indicates that all lanelets that comprise of that line failed to pass some aspect of the current **auto_qc** thresholds. In reality there are few conditions under which a lanelet would fail irrespective of the status of the rest of the lanelets in the same lane which typically involve an error during the preparation of the sample (an easy to spot result as it will cause poor quality across all lanelets using that sample).

Overall there are a series of instances appearing to support correlation for whole-lane failures and warnings but despite this there do appear to be occasions where a lanelet has failed where the remainder of the lane has not. Having discussed this with the project supervisor and contacts at the Sanger Institute we decided to continue to the implementation stage, agreeing that whilst some evidence of correlation between lanelets in the same lane has emerged, we will still be able to recover parameters that will be useful to quality control and statistical testing may be required following this analysis to describe how powerful such parameters are taking this possible correlation in to account.

It should be noted that the proportion of failures and warnings is considerably smaller than passes and so care will need to be taken to find a balance; for example it would not be feasible to merely discard lanelets from lanes that have failed entirely as there'd hardly be any data on which to train a classifier. Indeed other solutions may be possible, perhaps weighting observations which exhibit similar behaviour to other lanelets in their lane so as to give their parameter values less priority during the construction of the classifier itself.

It is worth noting that although the plot does not make a particular distinction between lanes in the same flow cell, lanes are sequentially identified so the red bars of thicker-width arguably display some failures across entire flow cells.

As a final note it should be stressed that this plot should be regarded as a diagnostic rather than an experiment with a direct conclusion. Given more time it would be useful to investigate the nature of these possible correlations, given a lane that has failed across all lanelets: do those lanelets actually express similar quality metrics?

4.2 Recovering Ratios

An initial scan of the summary numbers available in the "bamcheckr'd" files introduced in Chapter 3.1.1 revealed 82 different parameters. However, when comparing this parameter set to the threshold rules of **auto_qc**, it appeared that some parameters were "missing".

In the pursuit of replicating the decisions of the existing system, it is ideal to have all the parameters used

in the making of those decisions at hand. In particular, the missing parameters were of a normalised nature, typically in the form of a ratio or percentage which should make them more valuable than a parameter that just represents a raw count of some property.

It was found that these missing parameters are calculated in a step preceeding **auto_qc** as part of the **vr-pipe** pipeline[1].

Whilst I could have attempted to set-up my own instance of **vr-pipe** to recover this data, speaking with contacts at the Sanger Institute, it was decided that this would prove troublesome work; requiring an involved deployment to a cloud based facility such as Amazon's Elastic Compute Cloud and installation of various Perl dependencies as well as requiring access to many controlled databases within the institute.

Most of the functions that calculated these parameters turned out to be straightforward and could easily be ported to another application. At first it was intended to add these functions to the program authored for this part of the project, but the Sanger Institute suggested it would be more useful to implement such functionality in **bamcheckr** directly, removing some of **auto_qc**'s dependence on its position in **vr-pipe**.

```
# Install and include the 'devtools' package
install.packages("devtools")
library(devtools)

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

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

Listing 1 : Installing an in-development R package with **devtools**

With the help of **devtools**[22] (see Listing 1) it was simple to develop and test contributions to **bamcheckr** locally without needing to re-publish the package after changes. An additional script was added to **bamcheckr**'s NAMESPACE... to recover the missing ratio and percentage based parameters...

...unfortunately, the performance of the script was poor, taking 5.5s on average and increasing to 16.1s when enabling a complex function containing many vector operations (potentially inefficient due to my lack of R experience)... for overlapping_base_duplicate_percentage

...utilising the program designed and implemented in the following chapter... loaded required information and used the API to calculate the missing parameters...

...with such an intriguing difference in performance with significantly more time it would be appealing to explore the idiosyncrasies of each implementation ...although such an investigation would quite likely sufficiently generate an entire project of its own.

[1] https://github.com/wtsi-hgi/vr-pipe/blob/hgi-release/modules/VRPipe/Steps/vrtrack_auto_qc_hgi_3.pm

4.3 Contributions to bamcheckr

...R CMD BATCH issue ...Fixed a graph plotting failure.

Chapter 5

Frontier

This chapter introduces **Frontier**: the main programming effort for this part of the project. Frontier is a Python package that serves as a data manager, providing both interfaces to read inputs into structures in memory and to retrieve them in formats acceptable to a machine learning framework.

5.1 Design

5.1.1 Purpose

Frontier's purpose is to supplement analysis with **scikit-learn** by allowing a user to read in and parameterise data in a format that can then be used for analysis by the **scikit-learn** library. Frontier was designed to simplify the process of setting up machine learning tasks and enable experiment repeatability by removing the need for users to spend time constructing classes and functions to parse their input data and to just get on with analysis using **scikit-learn**.

Initially Frontier was to act as a wrapper around **scikit-learn**, essentially removing the end user's interaction with the library and merely providing an interface for data to be passed in and some sort of classifier to be returned. However this quite clearly limited Frontier's audience by tying it to a particular framework and would quickly become unmanageable in the task of providing wrappers for all aspects of an external library.

Instead it was decided that Frontier would be used alongside a user's chosen machine learning framework, providing a useful API to parse and extract observations and variables from input data and arrange them in structures suitable for processing with **scikit-learn**.

If possible, Frontier could also handle any objects returned, displaying or logging any textual or graphical information pertaining to a returned classifier's accuracy to assist a user in the ongoing performance monitoring of changes to used data subsets or parameters.

5.1.2 Format

Frontier is designed as a Python package, allowing a user to import its functionality in to other programs. The result of this project's technical output could almost be considered as two separate entities: Frontier itself, the package designed to ease user interaction with scikit-learn and **Front**, a Python script which implements Frontier's functionality in order to interact with scikit-learn to conduct analysis on the **auto_qc** data.

5.1.3 Method

.. the result of abstracting code and tools created during the use of **scikit-learn** for analysis of the current **auto_qc** system .. initially hard-coded to suit the specific needs of the project but followed an evolutionary design process...

designed specifically for the given learning problem, with hard coded classes and encodings...

5.2 Concepts

In this section we introduce some terminology and ideas that will assist understanding of the application's purpose and terminology used in the following implementation section.

5.2.1 Observations and Parameters

An **observation** refers to a distinct object or "thing" from which we will attempt to understand the relationship between its known properties and classification to be able to label future observations with unknown class based on those properties. In the context of this project, this would be a lanelet. Other documentation in the field may use the term *sample* but to avoid confusion with the definition of sample introduced in Chapter 2.2.2 we will use the term observation.

The aforementioned *properties* or *attributes* of an observation will be referred to as the **parameters** of that observation. Traditionally these may be described as *features* but it was felt that this wording may have connotations with discrete data. Early versions of Frontier referred to parameters as *regressors*, using terminology from statistical modelling. This wording was dropped to remove any confusion between regression and classification machine learning problems.

5.2.2 Data and Targets

Data will be used somewhat generically to refer to a matrix in which observations act as rows and their parameters act as columns. It is expected that all observations will have the same set of parameters.

For problems of a classification nature such as ours, an observation's **target** is the known classification of that particular observation. Depending on the context of the learning problem targets may be discovered manually

(*e.g.* counting leaves from images of plants, one will need to manually count all the leaves in the image before being able to use it for learning) or as the output from another system such as **auto_qc**.

In the context of this project where the objective is to begin learning the rules of **auto_qc** the targets refer to whether a particular lanelet observation was labelled as a pass, fail or warn by the current system.

5.2.3 Class Labels and Codes

An observation's target may also be known as its **label**. A labelled observation is said to be a member of that label's **class**. For example a lanelet that has failed **auto_qc** is said to be labelled as a fail and is a member of the class of failures.

String based class labels can pose difficulties when handling data later on, not only taking more memory to store as opposed to a type such as an integer but also introducing accidental subsetting or discarding of data whose labels differ unexpectedly. For example are "fail", ,"fial", "FAIL" and "Failed" supposed to be members of the same class?

Often such labels are **encoded** in to simpler types such as integers.

5.2.4 Training and Testing

...

5.3 Implementation

This section investigates the implementation of Frontier's major components:

- Informing the package of the problem domain...
- Interfaces provided for reading in data
- Storage of read data in memory
- Retrieval of stored data
- Interaction with scikit-learn

5.3.1 Class Definitions

Frontier was designed to support classification machine learning problems, to adequately support this task, the package must be aware of each of the possible classes in the problem space. Early versions of Frontier were specifically designed for training and testing data from **auto_qc** and could only support encoding and decoding of the pass, fail and warn classes.

However this implementation was clearly esoteric and held little to no further use outside the domain of the project's learning task. What would happen if a class label were to be added or removed in future? Most likely many lines of code would need to be re-written to handle such cases; the package was inflexible.

To counter this, Frontier was refactored to remove hard coded label definitions enabling its use as a more general purpose tool where users can specify the domain's classes and their associated labels and encodings. Listing 2 shows the definitions used by Front to work with the **auto_qc** data.

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

Listing 2 : Class definitions for **auto_qc** as passed to Frontier

As per Listing 2, to define classes a user must provide a Python dict containing the following for each label:

- **class** *String*
  The dictionary key is used as the canonical name of the class label
- **names** *[String]*
  A list of labels that denote membership of this class
- **code** *Integer*
  The encoded representation of this class (See Chapter 5.2.3)
- **_recode** *Boolean, Private*
  A flag to indicate whether an API action has changed the code of this class, typically used when treating all members of one class as a member of another – A user should never set this manually
- **_count** *Integer, Private*
  The number of observations with this label, typically used when calculating weightings based on the proportions of class sizes and outputting logging information – A user should not set this manually

This simple structure allows Frontier to be compatible with almost any classification learning task with minimum input from the end user. Additional utilities provided by the Frontier utils subpackage use this

structure to automatically classify and encode labels without user intervention with the following functions:

- **classify_label**
  Attempt to classify a label by comparing a given string to each set of *names*, locating an exact match will return the relevant canonical class label
- **encode_class**
  Given a canonical class label, return its *code*
- **decode_class**
  Given a *code*, return the canonical class label unless *_recode* is True
- **count_class**
  Increment the *_count* for a particular class given its canonical label

These functions are put to use throughout Frontier and are essential for reader classes (detailed in the next chapter) to parse, classify and then encode observation targets automatically from relevant files.

5.3.2 Input Handling

Frontier's modular nature allows users to write their own Python classes to read data from any form of input file or stream. Two examples of which are the classes used to read from the "bamcheckr'd files" documented in Appendix A.1 and the **auto_qc** decisions matrix briefly demonstrated in Appendix A.2.

These classes are described as **Readers** and implement a common base class, **AbstractReader** which takes care of setting up the file handler, including functions to both close and iterate over the file's contents. It will also automatically call its own **process_file** function that skips over any header lines before passing each line in the file stripped of any newline characters to **process_line**.

It is expected that derived classes will at least provide their own implementations for **process_line** and **get_data**. Failing to do so will cause Frontier to throw a **NotImplementedError** when attempting to use the class to read data.

process_line defines the line handling operations that extract and store desired data found in a given line. This responsibility includes returning None for lines that contain comments and irrelevant data.

get_data must return any read in data in a suitable structure for storage by Frontier. Typically this will be a Python dictionary using some unique identifier for each observation as a key, mapping to an arbitrary value or object containing that observation's parameters. Further discussion on Frontier's storage of data and targets is to follow.

The **AbstractReader** class is designed to simplify the process of reading in observations and their targets for end users, however it is still up to the author of the dervied class to set up any structures to store data (which cannot be done automatically without likely enforcing potentially unhelpful constraints) before initialisation of the inherited base class (via the call to **super**) as shown in Listing 3.

```
class BamcheckReader(AbstractReader):
    [...]

    def __init__(self, filepath, CLASSES, auto_close=True):
        self.summary = SummaryNumbers()
        self.indel = IndelDistribution()
        super(BamcheckReader, self).__init__(filepath, CLASSES, auto_close, 0)

    [...]
```

Listing 3 : Extract from **BamcheckReader** class documenting initialisation of necessary data structures and calling for initialisation of its inherited base class

As shown in Listing 3, the initialisation of the **AbstractReader** allows four arguments:

- **filepath** *String*
  A relative or absolute path to a file from which to extract data or targets
- **CLASSES** *Dictionary*
  A dictionary of user specified class labels defined as described in the previous chapter
- **auto_close** *Boolean, Optional*
  Whether or not to close the file handle immediately after executing **process_file**, this is True by default to prevent either memory leaking when users are reading in a large number of files and are perhaps unaware that they require closing or the throwing of an IOError caused by having too many file handles open at once
- **header** *Integer, Optional*
  The number of lines to ignore before the reader should begin passing stripped lines to **process_line**, defaults to 0

Currently it is also the responsibility of the author of a derived class to perform relevant sanity checking of any extracted data. For example the **BamcheckReader** class checks for the presence of multiple entries of a particular metric which will print a notice if found, unless the entries have differing values, upon which an exception is thrown and the process is halted.

Once a reader has been defined for a particular file format, a user need only provide a directory of files to be parsed and the name of the class designed to complete the parsing. Frontier will then take care of executing the parsing process on all files in the directory. After a derived reader has completed file handling, Frontier will call its **get_data** function to "move"[1] the extracted data to its own storage.

At this point Frontier will also check the integrity of the data, primarily that all parameters have a non-zero variance. Users will be warned when this requirement is violated; parameters with no variance cannot provide much information for successful classification as their values are equal for all class labels!

[1] Rather, a pointer to the address of the extracted data's dictionary hashmap will be copied to memory inside a Frontier class

In future it would be useful to investigate whether it would be feasible to perform such sanity checking in a generic manner to ensure it could be applied to a wide enough range of scenarios to make it worth including functionality in the **AbstractReader** directly.

With more time, future improvements could overhaul the reader interfaces to allow users to simply specify the format of a file in a string that can be parsed by Frontier's IO subpackage, rather than having to write their own derived class. Classes could also list file extensions they are capable of processing which could potentially be used to automatically determine which readers to use without requiring the user to explicitly specify.

5.3.3 Storage

Frontier specifies a class called the **Statplexer**[2] which provides users with a single point of access to all read in data. The reader interfaces described in the previous chapter implement their own loading functions to populate the **_data** and **_targets** class members of the **Statplexer** object.

_data and **_targets** are Python dictionaries, a structure in which hashed keys are mapped to an arbitrary value or object. During the parsing of observation data with the relevant reader class, the reader is expected to locate an appropriate unique ID for each observation. In the case of processing of **auto_qc** data, this would be the lanelet's barcode which is collected from the filename of that particular lanelet's "bamcheckr'd" file.

This identifier is then used as a key in both the **_data** and **_targets** dictionaries to map to a structure (typically another dictionary or an arbitrary class) that stores that observation's parameters and its known target (encoded class label), respectively.

Although these attributes can be manipulated directly (and indeed they are for testing purposes) the leading underscore follows a popular convention defined in Python's style guideline, PEP8[23], where class members with leading underscores should be treated as non-public. Python doesn't have private variables such as those that may be found in other languages like Java, indeed the Python style guide points out that "no attribute is really private in Python"[23]. In an interesting StackOverflow answer on the subject, a user describes that this is "cultural"[24] and that Python programmers are trusted not to circumvent convention and "mess around with those private members". Users are therefore expected to use the functionality Frontier provides for controlled getting and setting of data stored in these pseudo-private **_data** and **_targets** members.

For example, **load_data** should be used exclusively when populating the two dictionaries and is automatically called on construction of the **Statplexer** if the arguments are valid. **load_data** will automatically call other (pseudo-private) functions of the class including **_test_variance** which checks that parameter variances are non-zero whilst also warning users if new observations are overwriting old ones or if an observation does not appear to have a corresponding target.

...Python dictionaries offer (constant) lookup access and more importantly querying the structure for whether it contains a particular key can also be performed in constant $O(1)$ as opposed to searching a list for membership[25]...

...The following section describes the retrieval of data and targets in the form of lists or **numpy** arrays... so why not just store read in data in one of these structures to begin with instead?

[2] A somewhat contrived contraction of 'Statistics Multiplexer'

This is primarily due to the underlying layout of a list structure, which must be represented in memory as a contiguous block. Great expense is therefore incurred if the list grows beyond its bounds and must be relocated to resize. Given the nature of input data, the number of observations and their parameters are unknown before the input data is read and so memory cannot be reserved to prevent these operations.

It is arguable that despite this, the data could be unloaded from the **_data** and **_targets** dictionaries in to some form of array at the end of the call to **load_data**. However the **Statplexer** allows loading of data at any time, which would not only risk increasingly expensive resizing operations as the list continues to expand but the sanity checking that takes place in **load_data** involves checking for membership of an ID in **_data** and **_targets** where performance is constant for dictionaries and O(n) for lists.

...although the latter point is somewhat easily fixed for an array implementation by using a dictionary to provide a mapping between a sample name and where it is stored in an array...

...an issue with use of a dictionary however is by default they are unsorted, not only this but the order in which the keys are iterated over will be undefined...

...currently this causes the list of keys to require sorting each time a user queries the Statplexer API for data... as obviously the parameters and targets must map one-to-one in a predictable and repeatable manner.

It should be noted that the **Statplexer** stores a copy of the user defined CLASSES dictionary (as the class member **_classes**), which is an argument to its construction, allowing the **Statplexer** to share this information with any readers or utility functions who require it.

5.3.4 Retrieval

The Statplexer is designed to provide functions to an end user for extraction of desired data in a format suitable for an external framework or library, in our case; scikit-learn...

...leverages **numpy**...

...Frontier can be used to inspect the parameter set extracted from the observations... some of the methods provided for this purpose include:

- **list_parameters**
  Return a sorted list of all parameters
- **find_parameters**
  Given a list of input strings, return a list of parameters which contain any of those strings as a substring
- **exclude_parameters**
  Given a list of input strings, return a list of parameters which do not contain any of the input strings as a substring, or if needed an exact match

...data is returned sorted by key, ...with parameters sorted alphanumerically... ...targets map 1:1 with the data array...

- **get_data_by_parameters**
  Return data for each observation, but only include columns for each parameter in the given list
- **get_data_by_target**
  Return data for each observation that have been classified in one of the targets specified and additionally only return columns for the parameters in the given list

5.3.5 Interaction with scikit-learn

Cross Validation

...method in which to measure classification accuracy... ...potentially use a weighting to penalise mistakes in smaller classes...

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

Confusion Matrices

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

5.4 Usage Example

```
from Frontier import frontier
from Frontier.IO import DataReader, TargetReader

data_dir = "/home/sam/Projects/owl_classifier/data/"
target_path = "/home/sam/Projects/owl_classifier/targets.txt"

CLASSES = {
        "hoot": {
            "names": ["owl", "owls"],
            "code": 1,
        },
        "unhoot": {
            "names": ["cat", "dog", "pancake"],
            "code": 0,
        },
}

statplexer = frontier.Statplexer(data_dir,
                                 target_path,
                                 CLASSES,
                                 DataReader,
                                 TargetReader)
```

Listing 4 : Example usage of Frontier

Constructing the Statplexer requires the following arguments:

- **data_dir** *String*
  Root data directory under which all files will be parsed for observation data
- **target_path** *String*
  Path to file to be parsed for observation targets
- **CLASSES** *Dictionary*
  A dictionary of user specified class labels defined as described in Chapter 5.3.1
- **DataReader** *Module*
  Module containing the class (of the same name) to be used to parse each file in the *data_dir* tree for observation data
- **TargetReader** *Module*
  Module containing the class (of the same name) to be used to parse the *target_path* file for target data

5.5 Testing

...

Chapter 6

Results

6.1 Introduction

6.1.1 Why Decision Trees?

6.1.2 CART

6.2 Parameter Selection

...important to find the "best" parameters ...what is best? scikit-learn uses total gini information

...frontier uses two methods:

- Backward elimination; pruning parameters with the lowest total gini
- Call scikit-learn's SelectKBest

* incorrect degrees of freedom * warnings: /usr/lib64/python2.7/site-packages/sklearn/feature_selection/univariate_selection.py:
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

6.3 Trees

6.3.1 Initial Trees

6.3.2 Parameter Sets

6.3.3 Ignoring Warnings

Part II

Identification of Qualitative Sample Properties

Chapter 7

Introduction and Motivation

7.1 Introduction

The second part of the project can be outlined as follows:

- Collation of variant locations...
- Construction of script to select a representative genomic region...
- Extraction of such regions...
- Merging of regions to one file...
- Establishment of pipeline... :
  - Select each sample in turn...
  - ...
  - Call variants...
  - Compare concordance...

7.2 Concepts and Terminology

7.2.1 Variants and SNPs

...for the purpose of this study we are only interested in single nucleotide polymorphisms (SNPs) "snips" ...

...their *position* refers to the index of the base of the chromosome in which the polymorphism exists... ...commonly these are 1-indexed (although this is not always the case) and thus care must be taken to ensure this is accounted for at any step where positions are considered...

7.2.2 The 'Goldilocks' Region

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 are 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 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 "Goldilocks genome".

7.2.3 Groups

ichip vs gwas... ...will become clear later

7.2.4 Length and Stride

...*length* will refer to the number of bases included in the region ...*stride* will refer to the number of bases to be used as an offset between the starting base of one region and the beginning of the next...

...a length of 1000 and a stride of 500 for example ...the first region for example will consist of 1:1000, then 501:1500 inclusive

...ensure sensible choices of these parameters!

7.2.5 Candidates

...a **candidate** is a region of *length*... ...a candidate will contain a number of variants, counted for each *group*...

...simple object consisting of start and end base...

Chapter 8

Materials and Methods

8.1 Input Data and Format

8.1.1 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
```

8.1.2 Variant Query File

...named such for the command that creates it, the Variant Query File is the proposed input format for **Goldilocks**...

8.1.3 Paths File

...esoteric file format to handle grouping of Variant Query files...

8.2 Development Environment

8.2.1 Language

Once again, Python was the language of choice for this script... ...for many of the reasons previously discussed in Chapter 3.2.1 ...we needed this quickly, Python is useful for fast prototyping... ...expected reasonable performance without the need for a lower level language like C...

8.2.2 Testing

...a little more difficult ...discussed more at length in the following chapter...

8.2.3 Tools

As before... (Chapter 3.2.5)

Chapter 9

Goldilocks

This chapter introduces **Goldilocks**; the first programming hurdle for this part of the project. **Goldilocks** is a Python script designed specifically to read in and store locations of user-selected variants on the human genome and to then find regions of a given size on the genome where variants are of a desired density.

9.1 Design

9.1.1 Purpose

Expanding on the introduction, **Goldilocks** is responsible for parsing input files that contain line-delimited, colon-seperated chromosome and base position entries (*e.g.* 1:5000 is the base at position 5000 on the first chromosome). Input files can be grouped, pooling the set of unique listed variants between the files and treating them as one search space.

For each file group, a list of positions for each chromosome is stored in memory. Once all variants have been loaded **Goldilocks** is designed to iterate over each chromosome[1], load all of the variants from the lists in to an array and essentially count the number of variants present (in each group) between a given start and end position, determined by the arguments specified by the user as described in Section 7.2.4.

This combination of a start and end position and the counts of variants present inbetween forms the structure of a **candidate**. Having performed this search over all the autosomes, candidates can then be ranked by user-specified criteria regarding the density of the counted variants.

9.1.2 Format

Goldilocks is a modestly sized but elegantly complex Python script which can also serve as a stand-alone module to be imported in to other programs. Whilst the use case is certainly more esoteric than **Frontier**, I

[1] Strictly speaking we are only handling the 22 *autosomes* and ignoring the remaining 2 sex chromsomes (*allosomes*)

made the choice to modularise the functionality, in the hope that if an end user were to find some function of the script particularly useful they may call it as part of their own program.

With more time it would be interesting to research other potential uses for such a script...

9.1.3 Method

Development was akin to a more traditional methodology; requirements were simple and available up front, testing could wait until all the components had been completed to properly measure whether outcomes were as expected.

Goldilocks had to be constructed quickly but carefully. Although avoiding a quick and dirty solution, the initial codebase was arguably cluttered with blocks of repeated code to perform actions for different hard coded groups. Testing was not automated and utilised eyeballing of resulting data and occasional checking of input files to ensure results were as expected.

The first completed version consisted of a somewhat monolithic script, prime for breaking up in to a more organised set of functions...

9.2 Implementation

...we investigate the following implementation stories:

- Handling user input of grouped Variant Query files
- Loading variants from identified Variant Query files
- Storage of read data in memory (and troubles thereof)...
- Striding and window size and data structures thereof
- Final output and selection, counters and buckets (avoid chr6)
- Testing, generate output files, call functions akin to API

9.2.1 Loading Variant Query File List

Goldilocks requires the user provide their own valid "pathfile" as meeting the specification demonstrated in Section 8.1.3 (an example of such a file can be found in Appendix **??**).

merely lists locations of input files and groups them together as desired ...all files in a particular group will be considered as a whole (all variants from all files in the group will be added to the set of chromosomes for that group) ...each group will have a three data structures created...

9.2.2 Loading Variants from Input Files

...no duplicate checking to avoid needing to check for list membership ...this is not an issue later (setting a flag to 1 multiple times will cause no effect)... seemed faster to repeat those actions than to check for membership...

...however this process of appending variants to a list would suffer from expensive memory moving operations and is ripe for alteration... ...perhaps a set would be more appropriate... ...investigating the difference in performance would be of interest...

...although the input for this part of the study handles <x> variants in <y> seconds and thus time was better spent elsewhere...

9.2.3 Storage

As I discussed previously, on the hunt for my Goldilocks genomic region, it is 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 is of course rather dumb. Looking up a list has $\mathcal{O}$(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 its performance.

A far more sensible solution is 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 us 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...
```

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 (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.

...supervisor suggested bloom filters... ...would be a useful experiment to measure what further performance gains could be acheived with such a method...

9.2.4 Searching for Regions

..

...for the purpose of this project we're looking at: ...the median of the GWAS (representative) ...the maximum of the iCHIP (enriching for maximum comparisons between data sets)

9.2.5 Final Region Selection

.. ...with more time some of the functions would be more abstracted ...currently criteria are still hard coded to meet the needs of the project and propose little re-use value unless altered...

...in this state does however still allow for repeatable results in future with different data sets or additional groups...

Fig. 9.1 **Megabase Plot**: ...

9.2.6 Testing

...particularly difficult. ...required generation of input files that would contain known regions to test were found by the program... ...needed to re-write large aspects of the scripts to allow unit-like testing to take place... ...wanted to avoid a monolithic test class...

...tests are still rather modest and with more time it would be appropriate to improve their function but I have faith in their results...

Chapter 10

Analysis Pipeline

Following the extraction of the Goldilocks region for each of the samples... we need to begin putting together the pipeline that would process the work for the leave-one-out analysis...

...involving generation of indexes for each sample ...execute the pipeline and withold one of the Goldilocks samples... ...using **samtools** to pileup the variants across all the samples (excluding one of course) and then performing variant calling... ...must then compare the resulting VCF against the SNP VCF and calculate the concordance to discover whether leaving out that particular sample has caused a change in the accuracy of the variants called...

10.1 Implementation

10.1.1 Region Extraction

...regions extracted from IRODS ...difficulty gaining user access for both IRODS and "the farm"

10.1.2 samtools index

...index extracted regions

10.1.3 samtools mpileup

Following extraction of the Goldilocks region for each full-genome sample, the next stage is to use **samtools mpileup** to ... ...each of the various full genome samples we have for processing. These files are then summarised (and likelihoods are calculated) with samtools mpileup...

samtools mpileup is a rather intensive process especially when in a scenario such as ours where there are thousands of input files, one for each of the extracted regions. This not only places considerable strain on the cluster's file system but is incredibly inefficient given the overhead of file handling. As this pipeline will need to be executed once for each leave-one-out experiment, it would be required to reduce the strain on the file server before the job could be submitted. Usefully another member of the **samtools** collection provides functionality to merge a list of given input files such as those extracted Goldilocks regions.

```
bsub -o ~/goldilocks/joblog/samtools_mpileup.%J.o -e
~/goldilocks/joblog/samtools_mpileup.%J.e -G hgi -J "samtools_mpileup"
-M1000 -R "select[mem>1000] rusage[mem=1000]" bash -c 'samtools mpileup -b
../goldilocks-3:46000001-47000000.fofn -g -I -f
/lustre/scratch113/resources/ref/Homo_sapiens/1000Genomes_hs37d5/hs37d5.fa >
~/goldilocks/all.withref.bcf'
```

10.1.4 samtools merge

Usage

...undocumented feature to use a file of filenames... allowing more than two files to be specified at a time...

Memory Leak

By default the LSF scheduler at the Sanger Institute will issue 100MB to any submitted job. Given both the number of input files and their total size it was anticipated that this merge job would require more memory. At the very least an estimated 35kB for a 64-bit pointer to each input file and approximately 150MB to house a 32kB buffer to read data from each file also. Yet executing the job with 500MB caused the LSF scheduler to forceably terminate the process for exceeding the maximum allocated memory limit.

Assuming that the intermediate structures for storing and sorting the input data must have been greater than expected, the job's memory limit was generously increased with the syntax demonstrated by Listing 5 only to meet the same fate, even when reserving 16GB of memory...

This is an absurd amount of memory even considering the vast input... It appeared a memory leak had been discovered... that drastically increases in severity as more files are provided as input,

```
bsub -R"select[mem>4000] rusage[mem=4000]" -M4000 ...
#                  |                 |         | Raise maximum job memory to 4000mb
#                  |                 | Pre-reserve 4000mb for job before execution
#                  | Only run on a node with more than 4000mb memory
```

Listing 5 : Flags required to raise job memory allocation

...initial memory leak fixed by the author, several large variables not being freed from memory... ...following this, further **samtools merge** jobs were submitted only to also be repeatedly terminated by the LSF scheduler , this time for exceeding the maximum execution time limit for the queue...

Memory Leak in Test Harnesses

...getline ...regcomp...

...other memory leaks still remained in the tests and wanting to brush up on finding memory leaks with **valgrind**, I volunteered to locate and patch these...

```
==30464== 416 bytes in 1 blocks are indirectly lost in loss record 85 of 103
==30464==    at 0x4A082F7: realloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==30464==    by 0x3FBCCCA725: duplicate_node (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCD3ADA: duplicate_node_closure (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCD415A: calc_eclosure_iter (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCD79F6: re_compile_internal (in /usr/lib64/libc-2.17.so)
==30464==    at 0x4A08121: calloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==30464==    by 0x3FBCCCCD88: create_cd_newstate (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCCD506: re_acquire_state_context.constprop.41 (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCC2132E: build_trtable (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCD3792: re_search_internal (in /usr/lib64/libc-2.17.so)
==30464==    by 0x3FBCCD8E94: regexec@@GLIBC_2.3.4 (in /usr/lib64/libc-2.17.so)
==30464==    by 0x40C6FA: check_test_2 (test_trans_tbl_init.c:124)
==30464==    by 0x402CC4: main (test_trans_tbl_init.c:348)
```

Listing 6 : Example of **valgrind** locating a memory leak in one of the **samtools** test harnesses following the failure to release memory allocated to a compiled regular expression

Poor Time Performance

...submitting the same job to the "long" (48hr) and "basement" (essentially unlimited) queues, it is clear that the job is taking an extraordinary length of time to complete...

...during this time I took the opportunity to patch various memory leaks in the test harnesses of both merge and split...

valgrind, the tool I used to track down memory leaks in the test harnesses of both samtools merge and samtools split actually consists of more than just memcheck.

callgrind is a profiling tool that keeps track of a program's call stack history, a handy feature built in to some development environments such as QtCreator.

Having constructed a modest test set of files to merge, I called samtools merge from the command line, attaching callgrind and later imported the resulting text file to QtCreator's profiling tool to interpret the results (I usually use KCacheGrind for this but I've been investigating QtCreator's feature set for reasons unrelated

to this project), the result is immediately obvious - millions of calls to functions in zlib; a free compression library.

Further investigations using the QtCreator Analyze interface revealed that these calls all boiled down to one line called not during the process of deflating the input files (as I had expected) but actually during the compression of the output!

Function	Location	Called	Self Cost: Ir	Incl. Cost: Ir
0x0000003fbdc032d0	/usr/lib64/libz.so.1.2.7	1097	2,904,064,736	10,226,990,234
0x0000003fbdc0b4e0	/usr/lib64/libz.so.1.2.7	1112	944,594,568	944,594,568
crc32	/usr/lib64/libz.so.1.2.7	2202	220,011,172	220,011,172
re_search_internal	/usr/lib64/libc-2.17.so	690	118,935,704	283,936,905
bam_aux_get	/home/sam/git/htslib/sam.c in /usr/local/bin/samtools	232074	112,513,788	112,513,788
0x0000003fbdc0a770	/usr/lib64/libz.so.1.2.7	756420	95,768,664	95,768,664
__memcpy_ssse3_back	/usr/lib64/libc-2.17.so	938957	79,871,763	79,871,763
0x0000003fbdc0a870	/usr/lib64/libz.so.1.2.7	3336	60,220,888	155,989,552
bam_translate	/home/sam/git/samtools/bam_sort.c:(387) in /usr/local/bin/samtools	116037	36,996,141	210,497,579
check_halt_state_context.isra.20	/usr/lib64/libc-2.17.so	276477	33,089,554	40,556,431

Fig. 10.1 callgrind output following merge with default output compression

Function	Location	Called	Self Cost: Ir	Incl. Cost: Ir
sam_format1	/home/sam/git/htslib/sam.c:(127) in /usr/local/bin/samtools	116037	1,465,277,216	1,470,263,575
0x0000003fbdc07370	/usr/lib64/libz.so.1.2.7	1188	1,075,724,775	1,075,724,775
re_search_internal	/usr/lib64/libc-2.17.so	690	118,935,056	283,738,997
bam_aux_get	/home/sam/git/htslib/sam.c in /usr/local/bin/samtools	232074	112,513,788	112,513,788
__memcpy_ssse3_back	/usr/lib64/libc-2.17.so	953054	50,203,177	50,203,177
bam_translate	/home/sam/git/samtools/bam_sort.c:(387) in /usr/local/bin/samtools	116037	36,996,141	209,790,146
check_halt_state_context.isra.20	/usr/lib64/libc-2.17.so	276470	33,088,917	40,555,605
0x0000003fbdc0a230	/usr/lib64/libz.so.1.2.7	3516	29,249,135	29,249,135
build_rtable	/usr/lib64/libc-2.17.so	3381	24,214,798	38,420,667
bam_aux_del	/home/sam/git/htslib/sam.c in /usr/local/bin/samtools	232074	18,673,215	36,202,474

Fig. 10.2 callgrind output following merge with uncompressed output

Looking at a brief explanation of the deflate algorithm, it seems reasonable to conclude the computational cost is rather asymmetric between compressing and uncompressing - in that the effort is locating blocks to compress and in comparison uncompressing is a reversible function on the known blocks.

Indeed, samtools merge specifies a -u option for uncompressed output and the callgrind output (second image) indicates significantly less calls to zlib functionality.

It remains to be seen whether this option will cut down the time needed for the large merge job, perhaps this is merely a red herring and we're yet to discover the true speed trouble. In the meantime let's see if sending this job to the farm will work.

The Red Herring

... might be interesting to use gprof which is more geared towards finding functions that spend all your execution time as opposed to callgrind which I believe counts CPU instructions.

After re-compiling htslib and samtools with the -pg flag to enable such profiling and executing the same previous merge command on the modest test set, the output as parsed by gprof seems to indicate that the trouble lies with bam_aux_get in htslib, with almost 50% of the execution time being spent in this particular function.

```
uint8_t *bam_aux_get(const bam1_t *b, const char tag[2])
{
    uint8_t *s;
    int y = tag[0]<<8 | tag[1];
```

```
        s = bam_get_aux(b);
        while (s < b->data + b->l_data) {
            int x = (int)s[0]<<8 | s[1];
            s += 2;
            if (x == y) return s;
            s = skip_aux(s);
        }
        return 0;
    }
```

At a glance it seems that bam_aux_get receives a pointer to a BAM record and a “tag”, an array of two characters representing an optional field as defined in Section 1.5 of the SAM file spec.

The function then appears to fetch all these auxiliary tags and iterates over each, comparing a transformation of that tag (x) to a pre-computed transformation on the input tag (y).

This would of course be inherently slow for files with many such tags; especially given that the function is called twice for potentially each line in a BAM file.

The Plot Thickens

...as we increase the number of input files, the time taken to read them in becomes non-linearly slower. Currently my money is on the seemingly inefficient trans_tbl_init that appears to be called for each file, with the current table of all previous files as an input...

10.1.5 bcftools call

...Unfortunately during the initial testing run of this step with all the Goldilocks regions it was discovered that the output only included the standard header information and not a single line for the variants themselves.

...is because the piled up file was not generated with a corresponding reference DNA sequence and so the REF (reference) column is set to N (an ambiguity code which translates to ‘any base’). bcftools call does have an -M flag to prevent ignoring rows where the REF base is N (apparently called a "masked reference") however this is currently causing a segmentation fault. Having recompiled htslib, samtools and bcftools I am now able to run bcftools call on my local machine on some test data. I guess I’ll need to have someone recompile the source for me on the cluster I’m using....

Part III

Discussions and Conclusions

Chapter 11

Current Status

Chapter 12

Critical Evaluation

Chapter 13

Conclusions

References

[1] 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.

[2] 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.

[3] auto_qc: Additional high-throughput sequencing autoQC steps [Github]. [Online]. Available: https://github.com/wtsi-hgi/seq_autoqc

[4] vr-pipe: A generic pipeline system [Github]. [Online]. Available: https://github.com/wtsi-hgi/vr-pipe/

[5] M. Niemi, "Internal QC Summary Report," 09 2012, unpublished.

[6] Triage: A todo manager for the disaster that is your day [Github]. [Online]. Available: https://github.com/SamStudio8/triage

[7] sam/Dissertation | Triage. [Online]. Available: https://triage.ironowl.io/sam/dissertation/

[8] Maggie Bartlett. (2011) Illumina HiSeq Flow Cell. National Human Genome Research Institute. Image. [Online]. Available: http://www.genome.gov/dmd/img.cfm?node=Photos/Technology/Genome%20analysis%20technology&id=80102

[9] 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.

[10] 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

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

[12] 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

[13] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[14] T. Smith. (2012–2013) aRrgh: a newcomer's (angry) guide to R. [Online]. Available: http://tim-smith.us/arrgh/

[15] 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.

[16] 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

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

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

[19] 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.

[20] M. Fowler, "Continuous Integration," 2006. [Online]. Available: http://www.martinfowler.com/articles/continuousIntegration.html

[21] Jenkins Usage Statistics. [Online]. Available: http://stats.jenkins-ci.org/jenkins-stats/svg/svgs.html

[22] Hadley Wickham and Winston Chang, *devtools: Tools to make developing R code easier*, 2013, r package version 1.4.1. [Online]. Available: http://CRAN.R-project.org/package=devtools

[23] W. B. Van Rossum, Guido and N. Coghlan, "PEP 8 – Style Guide for Python Code," *Python Enhancement Proposals*, 2001. [Online]. Available: http://legacy.python.org/dev/peps/pep-0008/

[24] Kirk Strauser (http://stackoverflow.com/users/32538/kirk-strauser), "Does python have 'private' variables in classes?" [Online]. Available: http://stackoverflow.com/questions/1641219/does-python-have-private-variables-in-classes

[25] Python Wiki. (2012) Time Complexity.

Appendix A

Input Examples

A.1 BAMcheckR'd 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	...
FFQ	2	21676	0	0	0	0	0	0	0	43	1885	...
FFQ	3	7	0	177	0	0	0	0	0	0	0	...
[...]
FFQ	74	3697	39	0	919	4933	0	0	56866	1524	...
FFQ	75	4542	0	0	0	0	0	4634	77822	0	...
FFQ	76	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	...
LFQ	2	3300	0	0	0	0	0	0	0	0	0	...
LFQ	3	6816	0	0	0	573	0	83	0	7011	...
[...]
LFQ	74	5980	3	91	0	0	0	1340	9696	72939	...
LFQ	75	4314	0	0	168	0	848	8591	0	70358	...
LFQ	76	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	...
MPC	2	21407	0	0	0	0	0	0	0	5	...
MPC	3	3205	0	0	37	0	43	0	12	0	...
[...]
MPC	74	779	0	3	0	131	440	0	93	7485	...
MPC	75	136	0	0	0	3	0	47	704	9302	...
MPC	76	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 pairs, other pairs
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) , number of deletions (fwd), .. (rev)
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 and 90th depth percentile
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
```

A.2 auto_qc Decision Matrix

```
lanelet     sample      study     npg       aqc       ...
9999_9#1    AQCTest000000     AQCTest     pass      passed      ...
9999_9#3    AQCTest000002     AQCTest     fail      failed      ...
9999_9#2    AQCTest000001     AQCTest     pass      pass      ...
9999_9#5    AQCTest000004     AQCTest     warn      warn      ...
9999_9#4    AQCTest000003     AQCTest     warn      warning      ...
9999_9#7    AQCTest000006     AQCTest     fail      fail      ...
9999_9#6    AQCTest000005     AQCTest     pass      passed      ...
9999_9#9    AQCTest000008     AQCTest     warn      warning      ...
9999_9#8    AQCTest000007     AQCTest     pass      passed      ...
```

A.3 Goldilocks Pathfile

```
# gwas
cd-gwas     /pools/encrypted/sanger/vcf/cd-seq.vcf.gz.q
uc-gwas     /pools/encrypted/sanger/vcf/uc-seq.vcf.gz.q
# ichip
cd-ichip     /pools/encrypted/sanger/vcf/cd.ichip.vcf.gz.q
uc-ichip     /pools/encrypted/sanger/vcf/uc.ichip.vcf.gz.q
```