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 16, 2014

Declaration

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

Sam Nicholls
2014

Abstract

Over the past few years advances in genetic sequencing hardware have introduced the concept of massively parallel DNA sequencing; allowing potentially billions of chemical reactions to occur simultaneously, reducing both time and cost required to perform genetic analysis[19]. However, these "next-generation" processes are complex and open to error[11], thus quality control is an essential step to assure confidence in any downstream analyses performed.

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

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

Contents

Chapter 1

Introduction

Over the past few years advances in genetic sequencing hardware have introduced the concept of massively parallel DNA sequencing, allowing potentially billions of chemical reactions to occur simultaneously, reducing both time and cost required to perform genetic analysis[19]. However, these "next-generation" processes are complex and open to error[11], 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**[1], 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[5], **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[13].

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**[4]. 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[3] 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.

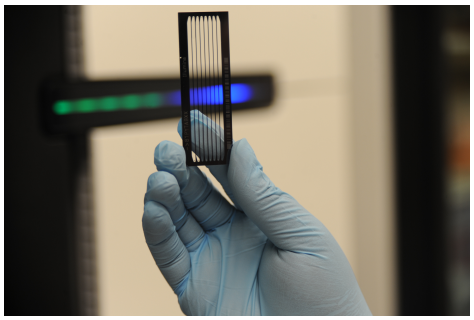

Fig. 2.1 An Illumina HiSeq Flowcell[12]

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.

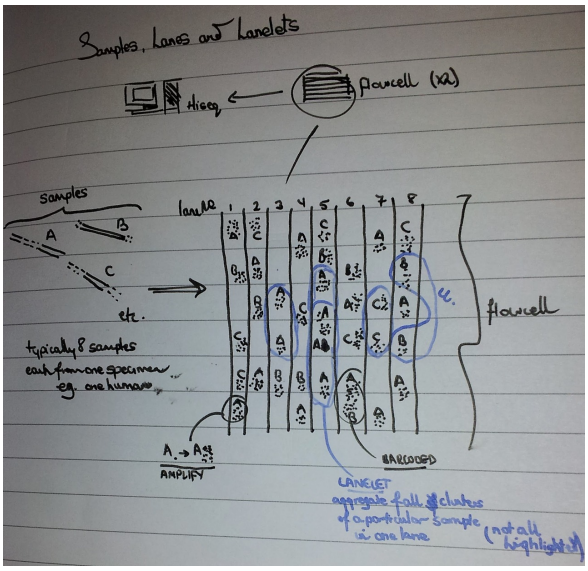

Fig. 2.2 Example of flowcell with some lanelets highlighted

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"[10] (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.

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

[1] Named such as **samtools stats** was once known as **bamcheck** and the tool is written in R

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**[8] 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**[17] and **rpart**[20] 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[16] is rather different to other languages and can significantly increase development time (and frustration[18]) 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**[6] 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[14] and scipy[9]... Fast and reliable implementations of mathematical functions... ggplot2[21] 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[7]. 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[2] 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?

Chapter 5

Implementation

5.1 Introduction

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.

5.1.1 Concepts

Classes and Labels

API Access

Cross Validation

5.2 Implementation

...provides a class to read from the "bamcheckr'd files" as seen in Appendix A.1... as well as the auto_qc output matrix briefly demonstrated in Appendix A.2...

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

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

Listing 1 Class definitions for auto_qc as passed to Frontier

Cross Validation

...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...

5.2.1 Contributions to bamcheckr

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

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

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

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

...R CMD BATCH issue

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

5.2.2 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 :(

5.3 Testing

Chapter 6

Results

6.0.1 Initial Trees

6.0.2 Parameter Sets

Part II

Identification of Qualitative Sample Properties

Chapter 7

Introduction and Motivation

7.1 Introduction

This part of the project can be outlined as follows:

- Selection of a genomic region...

7.1.1 Why Goldilocks?

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

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

For the next step of the project we 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”.

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
```

Chapter 9

Implementation

9.1 Goldilocks

9.1.1 Introduction

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.

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

Memory Leaks...

Testing

...particularly difficult.

9.2 Analysis Pipeline

9.2.1 samtools mpileup

9.2.2 samtools merge

9.2.3 bcftools call

Chapter 10

Current Status

References

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

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

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

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

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

[6] 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

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

[8] 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

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

[10] 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.

[11] 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.

[12] 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

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

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

[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] 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/

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

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

[19] 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.

[20] 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

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

Appendix A

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      ...
```