Portcullis Documentation

Release 1.1.2

Daniel Mapleson, Luca Venturini and David Swarbreck

Aug 09, 2018

Contents

1 System requirements **3**

2 Installation **5**
- 2.1 From brew . . . 5
- 2.2 From bioconda . . . 5
- 2.3 From source . . . 5

3 Quickstart **7**

4 Using Portcullis **9**
- 4.1 Prepare . . . 10
- 4.2 Junction Analysis . . . 11
- 4.3 Junction Filtering . . . 12
- 4.4 Bam Filtering . . . 14

5 Metrics **17**
- 5.1 Junction metrics . . . 17
- 5.2 Extra metrics . . . 21
- 5.3 Extracted metrics . . . 21
- 5.4 Final metrics . . . 22

6 Junctools **25**
- 6.1 Installation . . . 25
- 6.2 Compare . . . 26
- 6.3 Convert . . . 27
- 6.4 GTF . . . 28
- 6.5 Markup . . . 29
- 6.6 Set . . . 30
- 6.7 Split . . . 31

7 Frequently Asked Questions **33**
- 7.1 Can I reduce portcullis' memory usage? . . . 33

8 Issues **35**

9 Availability and License **37**

10 Additional Resources **39**

11 Authors and Acknowledgements **41**

Portcullis stands for PORTable CULLing of Invalid Splice junctions from pre-aligned RNA-seq data. It is known that RNAseq mapping tools generate many invalid junction predictions, particularly in deep datasets with high coverage over splice sites. In order to address this, instead for creating a new RNAseq mapper, with a focus on SJ accuracy we created a tool that takes in a BAM file generated by an RNAseq mapper of the user's own choice (e.g. Tophat2, Gsnap, STAR2 or HISAT2) as input (i.e. it's portable). It then, analyses and quantifies all splice junctions in the file before, filtering (culling) those which are unlikely to be genuine. Portcullis output's junctions in a variety of formats making it suitable for downstream analysis (such as differential splicing analysis and gene modelling) without additional work.

Contents:

CHAPTER 1

System requirements

Portcullis supports Unix, linux or Mac systems. Windows may work but hasn't been tested. Portcullis can take advantage of multiple cores via multi-threading where possible.

A minimum of 4GB RAM, which will enable you to process small datasets, but we suggest at least 16GB to handle most medium sized datasets. Large datasets will require more RAM, the largest dataset we tested peaked at 50GB using 16 threads. Generally, except for extremely deep datasets, if you have sufficient memory to align the reads, you should have sufficient memory to run portcullis.

In terms of resource profiling the portcullis pipeline, the preparation, junction filtering and BAM filtering *subtools* have relatively modest memory requirements. The amount of memory the portcullis junction analysis stage requires will depend on the quantity of RNAseq reads, the number of potential junctions, and the contiguity of your genome and the number of threads used. We have optimised the code so that we only keep reads associated with junctions in memory as long as necessary to accurately capture all the information about that junction. As soon as we move beyond scope of a junction by encountering a read that exists beyond a junctions boundaries, we calculate the junctions metrics and discard all reads associated to that junction. Therefore a single very highly expressed junction can increase peak memory usage. In addition, because, threading processes multiple target sequences in parallel, the higher number of threads the higher the memory requirements. So try reducing the number of threads if you encounter out of memory issues or segmentation faults.

CHAPTER 2

Installation

2.1 From brew

If you have brew installed on your system you should be able to install a recent version of Portcullis by simply typing:

```
brew install brewsci/bio/portcullis
```

2.2 From bioconda

If you use bioconda you can install Portcullis using :

```
conda install portcullis --channel=bioconda
```

2.3 From source

Before installing Portcullis from source please first confirm these dependencies are installed and configured:

- **GCC** V4.8+
- **autoconf** V2.53+
- **automake** V1.11+
- **make**
- **libtool** V2.4.2+
- **zlib**
- **pthreads**
- **samtools** V1.2+
- **Python3** V3.5+ (including python3 development libraries and the *pandas*, *numpy*, *tabulate* packages)
- **Sphinx-doc** V1.3+ (Optional: only required for building the documentation.)

NOTE ON INSTALLING PYTHON: Many system python installations do not come with the C API immediately available, which prevents Portcullis from embedding python code. We typically would recommend installing anaconda3 as this would include the latest version of python, all required python packages as well as the C API. If you are running a debian system and the C libraries are not available by default and you wish to use the system python installation the you can install them using: `sudo apt-get install python-dev`. Also, if you have installed python to a custom location please verify that the *bin* directors on the *PATH* environment variable, and the lib (or lib64) directory is on the *LD_LIBRARY_PATH* or *LD_RUN_PATH* as appropriate.

Then proceed with the following steps:

- Clone the git repository (For ssh: `` `git clone git@github.com:maplesond/portcullis.git` ``; or for https: `` `git clone https://github.com/maplesond/portcullis.git` ``), into a directory on your machine.
- "cd" into root directory of the installation
- Build boost by typing `` `./build_boost.sh` ``.
- Create configuration script by typing: `` `./autogen.sh` ``.
- Generate makefiles and confirm dependencies: `` `./configure` ``
- Compile software: `` `make` ``
- Run tests (optional) `` `make check` ``
- Install: `` `sudo make install` ``

The configure script can take several options as arguments. One commonly modified option is `` `--prefix` ``, which will install portcullis to a custom directory. By default this is "/usr/local", so the portcullis executable would be found at "/usr/local/bin" by default. Type `` `./configure --help` `` for full details and available options.

NOTE: if Portcullis is failing at the `` `./autogen.sh` `` step you will likely need to install autotools. The following command should do this on MacOS: `` `brew install autoconf automake libtool` ``. On a debian system this can be done with: `` `sudo apt-get install autoconf automake libtool` ``.

2.3.1 Internal Dependencies

Portcullis contains *HTSlib* and *Ranger* (a random forest implementation) in the source tree. The user does not need to do anything special to handle *htslib* and *ranger* as these are automatically built and managed inside portcullis.

Portcullis also comes with a python package for analysing, comparing and converting junction files, called junctools. This stands alone from portcullis so is not strictly required. Should you not wish to install this you can add the `--disable-py-install` option to the `configure` script. You can manually install this by going into the `./scripts/junctools` directory and typing `python3 setup.py install`. For more information about junctools see junctools for more information. Please note however that the portcullis python package is required for the filtering stage of Portcullis to run successfully.

CHAPTER 3

Quickstart

For the impatient, just use `portcullis full -t <threads> <genome> (<bam>)+` to get up and running. This runs the following subtools in order:

- Prepare
- Junction Analysis
- Junction Filtering
- Bam Filtering (optional)

This will output both unfiltered and filtered junctions from the provided BAM file. However, we recommend you read the *Using Portcullis* section for a more detailed description of what each stage does.

CHAPTER 4

Using Portcullis

Portcullis is a C++11 program containing a number of subtools which can be used in isolation or as part of a pipeline. Typing `portcullis --help` will show a list of the available subtools. Each subtool has its own help system which you can access by typing `portcullis <subtool> --help`.

The list of subtools present in portcullis are listed below in order. A more detailed description of each subtool follows in the subsequent sections:

- *Prepare*
- *Junction Analysis*
- *Junction Filtering*
- *Bam Filtering* (optional)

However, it is possible to run all portcullis steps in one go by using the *full* subtool. The command line usage for this option is as follows:

```
Usage: portcullis full [options] <genome-file> (<bam-file>)+

System options:
  -t [ --threads ] arg (=1) The number of threads to use.  Note that increasing␣
↪the number of threads will also increase memory requirements.  Default: 1
  -v [ --verbose ]          Print extra information
  --help                    Produce help message

Output options:
  -o [ --output ] arg (="portcullis_out") Output directory. Default: portcullis_out
  -b [ --bam_filter ]                     Filter out alignments corresponding with␣
↪false junctions.  Warning: this is time consuming; make sure you really want
                                          to do this first!
  --exon_gff                              Output exon-based junctions in GFF␣
↪format.
  --intron_gff                            Output intron-based junctions in GFF␣
↪format.
  --source arg (=portcullis)              The value to enter into the "source"␣
↪field in GFF files.

Input options:
  --force               Whether or not to clean the output directory before␣
↪processing, thereby forcing full preparation of the genome and bam files.  By
```

(continues on next page)

(continued from previous page)

```
                        default portcullis will only do what it thinks it needs to.
  --copy                Whether to copy files from input data to prepared data␣
↪where possible, otherwise will use symlinks.  Will require more time and disk
                        space to prepare input but is potentially more robust.
  --use_csi             Whether to use CSI indexing rather than BAI indexing.  CSI␣
↪has the advantage that it supports very long target sequences (probably not
                        an issue unless you are working on huge genomes).  BAI has␣
↪the advantage that it is more widely supported (useful for viewing in genome
                        browsers).

Analysis options:
  --orientation arg (=UNKNOWN)  The orientation of the reads that produced the BAM␣
↪alignments: "F" (Single-end forward orientation); "R" (single-end reverse
                                orientation); "FR" (paired-end, with reads␣
↪sequenced towards center of fragment -> <-.  This is usual setting for most␣
↪Illumina
                                paired end sequencing); "RF" (paired-end, reads␣
↪sequenced away from center of fragment <- ->); "FF" (paired-end, reads both
                                sequenced in forward orientation); "RR" (paired-
↪end, reads both sequenced in reverse orientation); "UNKNOWN" (default,
                                portcullis will workaround any calculations␣
↪requiring orientation information)
  --strandedness arg (=UNKNOWN) Whether BAM alignments were generated using a type␣
↪of strand specific RNAseq library: "unstranded" (Standard Illumina);
                                "firststrand" (dUTP, NSR, NNSR); "secondstrand"␣
↪(Ligation, Standard SOLiD, flux sim reads); "UNKNOWN" (default, portcullis will
                                workaround any calculations requiring strandedness␣
↪information)
  --separate                    Separate spliced from unspliced reads.
  --extra                       Calculate additional metrics that take some time␣
↪to generate.  Automatically activates BAM splitting mode (--separate).

Filtering options:
  -r [ --reference ] arg Reference annotation of junctions in BED format.  Any␣
↪junctions found by the junction analysis tool will be preserved if found in this
                         reference file regardless of any other filtering criteria.
↪  If you need to convert a reference annotation from GTF or GFF to BED format
                         portcullis contains scripts for this.
  --max_length arg (=0)  Filter junctions longer than this value.  Default (0) is␣
↪to not filter based on length.
  --canonical arg (=OFF) Keep junctions based on their splice site status.  Valid␣
↪options: OFF,C,S,N. Where C = Canonical junctions (GT-AG), S = Semi-canonical
                         junctions (AT-AC, or  GC-AG), N = Non-canonical.  OFF␣
↪means, keep all junctions (i.e. don't filter by canonical status).  User can
                         separate options by a comma to keep two categories.
  --min_cov arg (=1)     Only keep junctions with a number of split reads greater␣
↪than or equal to this number
  --save_bad             Saves bad junctions (i.e. junctions that fail the filter),
↪ as well as good junctions (those that pass)
```

This is the typical way to run portcullis but it's still helpful to know what each step in the pipeline does in more detail. Also the subtools offer some additional controls that can be useful in certain situations so please read on.

4.1 Prepare

This prepares all the input data into a format suitable for junction analysis. Specifically, this merges the input BAMs if more than one was provided. Using samtools, it then ensures the BAM is both sorted and both the sorted BAM and genome are indexed. The prepare output directory contains all inputs in a state suitable for downstream processing by portcullis.

Normally we try to minimise the work done by avoiding re-sorting or indexing if the files are already in a suitable state. However, options are provided should the user wish to force re-sorting and re-indexing of the input.

4.1.1 Usage

```
Usage: portcullis prep [options] <genome-file> (<bam-file>)+

Options:
  -o [ --output ] arg (="portcullis_prep") Output directory for prepared files.
  --force                                  Whether or not to clean the output␣
↪directory before processing, thereby forcing full
                                           preparation of the genome and bam files.
↪  By default portcullis will only do what it thinks
                                           it needs to.
  --copy                                   Whether to copy files from input data␣
↪to prepared data where possible, otherwise will use
                                           symlinks.  Will require more time and␣
↪disk space to prepare input but is potentially more
                                           robust.
  -c [ --use_csi ]                         Whether to use CSI indexing rather than␣
↪BAI indexing.  CSI has the advantage that it
                                           supports very long target sequences␣
↪(probably not an issue unless you are working on huge
                                           genomes).  BAI has the advantage that␣
↪it is more widely supported (useful for viewing in
                                           genome browsers).
  -t [ --threads ] arg (=1)                The number of threads to used to sort␣
↪the BAM file (if required).  Default: 1
  -v [ --verbose ]                         Print extra information
  --help                                   Produce help message
```

4.2 Junction Analysis

Portcullis is designed to be as portable as possible so it does not rely on esoteric SAM tags and other artifacts that are not consistently present in all SAM/BAMs. In this stage, Portcullis analyses the BAM file to look for alignments containing gaps (REFSKIP 'N' cigar ops) and creates a detailed analysis of all distinct gaps detected, these are considered as potential junctions. A number of observations are made for each junction such as number of supporting split reads, how those reads are distributed around the junction, the actual nucleotides representing the splice sites and how repetitive the genomic region is around the splice sits (see _metrics for more details of all measurements taken). Portcullis outputs all junctions found in the BAM in a number of formats such as GFF3, BED, although the most detailed information can be found in the tab file.

4.2.1 Usage

```
Usage: portcullis junc [options] <prep_data_dir>

System options:
  -t [ --threads ] arg (=1)      The number of threads to use.  Note that␣
↪increasing the number of threads will also
                                 increase memory requirements.
  -s [ --separate ]              Separate spliced from unspliced reads.
  --extra                        Calculate additional metrics that take some time␣
↪to generate.  Automatically activates BAM
                                 splitting mode (--separate).
  --orientation arg (=UNKNOWN)   The orientation of the reads that produced the BAM␣
↪alignments: "F" (Single-end forward
```

(continues on next page)

(continued from previous page)

```
                                orientation); "R" (single-end reverse orientation);
↪ "FR" (paired-end, with reads sequenced
                                towards center of fragment -> <-.  This is usual␣
↪setting for most Illumina paired end
                                sequencing); "RF" (paired-end, reads sequenced␣
↪away from center of fragment <- ->); "FF"
                                (paired-end, reads both sequenced in forward␣
↪orientation); "RR" (paired-end, reads both
                                sequenced in reverse orientation); "UNKNOWN"␣
↪(default, portcullis will workaround any
                                calculations requiring orientation information)
  --strandedness arg (=UNKNOWN) Whether BAM alignments were generated using a type␣
↪of strand specific RNAseq library:
                                "unstranded" (Standard Illumina); "firststrand"␣
↪(dUTP, NSR, NNSR); "secondstrand"
                                (Ligation, Standard SOLiD, flux sim reads);
↪"UNKNOWN" (default, portcullis will workaround
                                any calculations requiring strandedness␣
↪information)
  -c [ --use_csi ]              Whether to use CSI indexing rather than BAI␣
↪indexing.  CSI has the advantage that it
                                supports very long target sequences (probably not␣
↪an issue unless you are working on huge
                                genomes).  BAI has the advantage that it is more␣
↪widely supported (useful for viewing in
                                genome browsers).
  -v [ --verbose ]              Print extra information
  --help                        Produce help message

Output options:
  -o [ --output ] arg (=portcullis_junc/portcullis)
                                        Output prefix for files generated by this␣
↪program.
  --exon_gff                            Output exon-based junctions in GFF format.
  --intron_gff                          Output intron-based junctions in GFF␣
↪format.
  --source arg (=portcullis)            The value to enter into the "source"␣
↪field in GFF files.
```

4.3 Junction Filtering

Portcullis provides various means for filtering junctions detected in the input BAM file. By default we use a machine learning approach, which trains on a high- confidence subset of the data, and then applies the trained model to the full set in order to score each junction. By default scores of over 0.5 are classed as genuine and those under as invalid. The user can control the threshold value via a command line option. We use the a modified version of the ranger random forest code as the learner. Portcullis outputs all junctions passing the filter in a number of formats such as GFF3, BED, and TSV format. The user can also request all junctions failing the filter are output into an additional set of GFF, BED and TSV files.

4.3.1 Reference annotations

Should the user have access to a reference annotation, they can supply that (via the *–reference* command line option) so that should portcullis filter out any junctions that are also found in the reference, then those are put back into the set of genuine junctions. This feature is useful when working with model organisms where high-quality references are available.

The portcullis filter tool requires the reference junction annotation in BED format. If this is not readily available

Portcullis comes supplied with an addition toolkit called *Junctools*, which can convert GTF annotation files to a set of junctions in BED format.

4.3.2 Validating results

Should the user know whether each junction in the input set is genuine or not, that can be provided to portcullis via the *–genuine* command line option. This file takes the format of a line separated list of either *1* indicating genuine and *0* indicating invalid in the same order as the input junctions. Portcullis, can then measure the performance of its filtering strategy.

4.3.3 Rule-based filtering

Alternatively, the user can filter junctions based on simple rules applied to the junction metrics. They do this via a JSON file describing their filter profile, which is passed to the filter tool via the *–filter_file* command line option. Examples are provided in the *data* sub-directory, which can be used directly, or as a template for deriving a custom filter profile. The rules can be combined using logic operations (and / or / not, etc) and applied to the full set of input junctions.

Here's an example set of rules that must all be satisfied to pass this filter:

```
{
        "parameters": {
                "M4-nb_rel_aln": {
                        "operator": "gte",
                        "value": 2
                },
                "M12-maxmmes": {
                        "operator": "gte",
                        "value": 10
                },
                "M11-entropy": {
                        "operator": "gte",
                        "value": 1.5
                },
                "M13-hamming5p": {
                        "operator": "gte",
                        "value": 2
                },
                "M14-hamming3p": {
                        "operator": "gte",
                        "value": 2
                }
        },
        "expression": "M4-nb_rel_aln & M11-entropy & M12-maxmmes & M13-hamming5p &␣
↪M14-hamming3p"
}
```

4.3.4 Filtering with a pre-made model

Although it is generally not recommended, the user can re-use existing random forest models to apply to new datasets. This is done via the *–model_file* option.

4.3.5 Usage

```
Usage: portcullis filter [options] <prep_data_dir> <junction_tab_file>

System options:
  -t [ --threads ] arg (=1) The number of threads to use during testing (only␣
↪applies if using forest model).
  -v [ --verbose ]          Print extra information
  --help                    Produce help message

Output options:
  -o [ --output ] arg (="portcullis_filter/portcullis")
                                        Output prefix for files generated by this␣
↪program.
  -b [ --save_bad ]                     Saves bad junctions (i.e. junctions that␣
↪fail the filter), as well as good
                                        junctions (those that pass)
  --exon_gff                            Output exon-based junctions in GFF format.
  --intron_gff                          Output intron-based junctions in GFF␣
↪format.
  --source arg (=portcullis)            The value to enter into the "source"␣
↪field in GFF files.

Filtering options:
  -f [ --filter_file ] arg If you wish to custom rule-based filter the junctions␣
↪file, use this option to provide a list
                           of the rules you wish to use.  By default we don't␣
↪filter using a rule-based method, we instead
                           filter via a self-trained random forest model.  See␣
↪manual for more details.
  -r [ --reference ] arg   Reference annotation of junctions in BED format.  Any␣
↪junctions found by the junction analysis
                           tool will be preserved if found in this reference file␣
↪regardless of any other filtering
                           criteria.  If you need to convert a reference␣
↪annotation from GTF or GFF to BED format
                           portcullis contains scripts for this.
  -n [ --no_ml ]           Disables machine learning filtering
  --max_length arg (=0)    Filter junctions longer than this value.  Default (0)␣
↪is to not filter based on length.
  --canonical arg (=OFF)   Keep junctions based on their splice site status. ␣
↪Valid options: OFF,C,S,N. Where C =
                           Canonical junctions (GT-AG), S = Semi-canonical␣
↪junctions (AT-AC, or GC-AG), N = Non-canonical.
                             OFF means, keep all junctions (i.e. don't filter by␣
↪canonical status).  User can separate
                           options by a comma to keep two categories.
  --min_cov arg (=1)       Only keep junctions with a number of split reads␣
↪greater than or equal to this number
  --threshold arg (=0.5)   The threshold score at which we determine a junction to␣
↪be genuine or not.  Increase value
                           towards 1.0 to increase precision, decrease towards 0.0␣
↪to increase sensitivity.  We generally
                           find that increasing sensitivity helps when using high␣
↪coverage data, or when the aligner has
                           already performed some form of junction filtering.
```

4.4 Bam Filtering

Portcullis can also filter the original BAM file removing alignments associated with *bad* junctions. Both the filtered junctions and BAM files are cleaner and more usable resources which can more effectively be used to assist in downstream analyses such as gene prediction and genome annotation.

4.4.1 Usage

```
Usage: portcullis bamfilt [options] <junction-file> <bam-file>

Options:
  -o [ --output ] arg (="filtered.bam")   Output BAM file generated by this␣
↪program.
  -s [ --strand_specific ] arg (=UNKNOWN) Whether BAM alignments were generated␣
↪using a strand specific RNAseq library:
                                          "unstranded" (Standard Illumina);
↪"firststrand" (dUTP, NSR, NNSR);
                                          "secondstrand" (Ligation, Standard SOLiD,
↪ flux sim reads)  Default:
                                          "unstranded".  By default we assume the␣
↪user does not know the strand specific
                                          protocol used for this BAM file.  This␣
↪has the affect that strand information is
                                          derived from splice site information␣
↪alone, assuming junctions are either
                                          canonical or semi-canonical in form.  ␣
↪Default: "unknown"
  -c [ --clip_mode ] arg (=HARD)          How to clip reads associated with bad␣
↪junctions: "HARD" (Hard clip reads at
                                          junction boundary - suitable for␣
↪cufflinks); "SOFT" (Soft clip reads at junction
                                          boundaries); "COMPLETE" (Remove reads␣
↪associated exclusively with bad junctions,
                                          MSRs covering both good and bad␣
↪junctions are kept)  Default: "HARD"
  -m [ --save_msrs ]                      Whether or not to output modified MSRs␣
↪to a separate file.  If true will output
                                          to a file with name specified by output␣
↪with ".msr.bam" extension
  -c [ --use_csi ]                        Whether to use CSI indexing rather than␣
↪BAI indexing.  CSI has the advantage
                                          that it supports very long target␣
↪sequences (probably not an issue unless you
                                          are working on huge genomes).  BAI has␣
↪the advantage that it is more widely
                                          supported (useful for viewing in genome␣
↪browsers).
  -v [ --verbose ]                        Print extra information
  --help                                  Produce help message
```

CHAPTER 5

Metrics

During the junction making stage, Portcullis generates a list of potential splice junctions by assuming any RNAseq alignment containing an 'N' cigar operation. For all potential junctions portcullis makes a number of observations about how the reads align around the region. The list of observations (or metrics) generated at this stage are described in *here*. In addition, should the user wish to spend additional computational power calculating additional metrics these are listed *here*.

We are able to create high confidence sets of genuine and invalid junctions using rules based on these metrics. However, this isn't the full set of metrics used for filtering. Using the high confidence sets we are able to extract additional features. These are described *here*.

Finally, not all metrics are used by the machine learning algorithm because we found that either some metrics do not provide any additional information and are inferior to others. We went through a process of feature selection to settle on a set of features that can be usefully applied by our machine learning algorithm. These are listed *here*.

Before we do that, here is a small glossary of terms that we will use throughout:

- Splice junction - Splice junctions are points on a DNA sequence at which 'superfluous' DNA is removed during the process of protein creation in higher organisms. For the purposes of this tool a splice junction is essentially the same as an intron.
- Splice site - A junction has both a 5' donor and 3' acceptor site, which mark the start and end of the junction. Both donor and acceptor sites are 2bp long, and usually contain a canonical motif *GT*AG*, or its reverse complement on the negative strand.
- Intron - Any nucleotide sequence within a gene that is removed by RNA splicing while the final mature RNA product of a gene is being generated. For the purposes of this tool an intron is essentially the same as a splice junction.
- Anchor - The part of a spliced read that aligns to the genome. This will represent regions both upstream and downstream of the splice junction.
- Multiple Spliced Reads (MSRs) - Reads that cover more than a single spliced junction. These types of reads will become more common as sequencers become capable of producing longer reads.

5.1 Junction metrics

These metrics are calculated in the junction analysis stage for the portcullis pipeline and are derived directly from the BAM file generated by the RNAseq mapper.

5.1.1 Splice site type (canonical_ss)

Most splice junctions contain distinctive dinucleotide pattern at the start and end of the splice junction. These patterns are called *donors* and *acceptors* respectively. If the potential junction has the typical *GT_AG* motif (or its reverse complement on the negative strand then it is considered a canonical splice site. Alternatively, if the junction contains either *AT_AC* or *GC_AG* or their reverse complements then it is considered a semi-canonical splice site. Other motifs at the donor and acceptor sites are considered novel splice sites.

5.1.2 Number of spliced alignments (nb_raw_aln)

This a count of the number of reads containing an 'N' cigar operation at exactly the genomic position represented by this junction. Because we use 'N' cigar operations to derive all potential junctions each junction will have a value of 1 or more for this metric.

5.1.3 Number of distinct alignments (nb_dist_aln)

This is the number of distinct / non-redundant spliced reads supporting the junction. Therefore duplicate reads are only counted as a single read for this metric.

5.1.4 Number of uniquely / multiply spliced alignments (nb_us_aln, nb_ms_aln)

These are counts of the number of spliced alignments that support this junction that either do or do not also support another junction.

5.1.5 Number of uniquely / multiply mapped alignments (nb_um_aln, nb_mm_aln)

These are counts of the number of spliced reads that uniquely mapped to the genome, or mapped to multiple sites in the genome. We use the MAPQ value in the alignment to determine if a read is uniquely mapped or not. We treat anything with a value over 30 as being uniquely mapped and anything less as being multi mapped. This should be a reliable threshold as most aligners will output MAPQ scores of 0, 1, 2 or 3 and then another high value, e.g. 50 or 60 to represent a unique read.

5.1.6 Number of BAM / Portcullis properly paired alignments (nb_bpp_aln, nb_ppp_aln)

The number of alignments that are properly paired, according to either the BAM file flag, or calculated by portcullis. Some aligners, such as tophat do not always produce useful figures for this property due to insert size limits being affected by introns. Other aligners such as STAR treat all paired reads as properly paired. Therefore it's not useful to rely to heavily on this value unless you fully understand what the aligner is doing. To address this we make our own calculation of whether a alignment is properly paired. To do this we require the user to pass in the orientation of the reads. Typically, FR configuration for illumina RNAseq data. This allows us to do some basic checks like whether the pairs on on the same target sequence and if the orientation of the pairs makes sense according to what the user specified.

If the user does not specify an orientation for the reads, or if the data is single end, then the nb_ppp_aln will be 0.

5.1.7 Number of Reliable Alignments (nb_rel_aln, rel2raw)

This is the number of spliced reads supporting this junction that probably do not map elsewhere in the genome and are properly paired (according to portcullis calculations). If the user does not specify an orientation for the reads then this will be the same as the number of uniquely mapped reads.

The ratio of reliable reads to raw (nb_raw_aln) reads gives a good indication of whether the junction is genuine or not. Low ratios (near 0) indicate potentially unreliable junctions and high ratios (near 1) indicate potentially reliable junctions.

5.1.8 Shannon entropy (entropy)

This describes the shannon entropy of the spliced reads associated with this junction. This score is a measure of the amount of information present in the set of spliced reads supporting this junction. This metric is used to avoid problems attributed to calling splice junctions based on read counting alone, when read counting each read is assigned equal weight, even if they all start at the same position. Typically, you would expect a uniform distribution of starting positions for reads across the upstream anchor of the splice site, therefore a situation where all reads are stacked on top of one another should be treated as suspicious. Simply counting reads also makes it difficult to assign good minimum threshold values at which to call genuine junctions. The Entropy metric circumvents these problems. The entropy score is a function of both the total number of reads that map to a given junction, the number of different offsets to which those reads map and the number that map at each offset. Thus, junctions with multiple reads mapping at each of the possible windows across the junction will be assigned a higher entropy score, than junctions where many reads map to only one or two positions.

Although very useful, one disadvantage of the entropy score is that it does not take into account the quality of the reads contained within it, for example the number of mismatches present.

Entropy for each junction j is calculated based on the starting offsets of split reads supporting the junction. The following equations:

$$p_i = r_i / T$$

$$H_{j(s,e)} = -\sum_{i=s}^{e} (p_i \log_2 p_i)$$

where:

- $j(s, e)$ defines the left anchor region of the junction, starting at s and ending at e
- r_i is the number of split reads supporting the junction that start at offset i
- T is the total number of split reads supporting the junction

Shannon Entropy scores are also used in TrueSight and SPANKI.

5.1.9 Mean mismatches (mean_mismatches)

This is the mean number of mismatches found across all spliced reads supporting the junction. This includes any mismatches at any point along the spliced read, which includes mismatches even if they are the otherside of another junction in the case of an MSR. Originally described in TrueSight paper.

5.1.10 Mean read length (mean_readlen)

The average read length for spliced alignments supporting this junction.

5.1.11 Anchor lengths (max_min_anc, maxmmes)

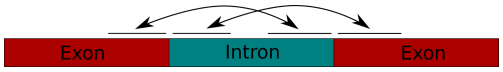
This is the maximum of the minimum (shorter) anchor sizes of all spliced reads associated with this junction. Again, all upstream and downstream junctions contained within MSRs are collapsed for the purposes of this metric. Originally used in TrueSight

MaxMMES (Maximum of the Minimal Match of Either Side of exon junction) takes into account mismatches in the anchors on either side of the junction. For each spliced read associated with the junction, we look at both anchors. The score for each anchor is the anchor length minus any mismatches to the reference. The minimal score from either the upstream or downstream anchor is taken. Then from these scores the maximum is taken

from all spliced reads. The MaxMMES for perfectly aligned reads should be the same as Max-Min-Anchor score. Therefore the difference between the two metrics is worth considering to gain an insight into how well the reads are mapping for a given junction. Originally described in Wang et al, 2010

5.1.12 Hamming distance at 5' and 3' splice sites (hamming5p, hamming3p)

Aligners can often make incorrect alignments around repeated genomic locations. In these instances it is good to know whether the region on the on the left side of the donor site and the left side of the acceptor site, in addition to the region on the right side of the donor site and the right side of the acceptor site are similar. In this is the case then it is likely that the false splice alignments have been made. We record both figures in terms of the hamming distances between the regions. Low scores indicate similarity, and therefore high change of alignment to a repeat region, high scores indicate difference and therefore low chance of alignment to a repeat region. Originally used in SPANKI.



5.1.13 Junction group metrics (uniq_junc, primary_junc)

We analyse junctions within a given loci to determine if they are isolated or have other junctions sharing splice sites or otherwise overlapping. The uniq_junc boolean metric determines whether or not there are any other junctions within this junctions region. This helps to determine if this junction might be involved in alternative splicing.

In addition, if this is not a unique junction, then this is a primary junction if it has the most spliced reads when compared to the other junctions in this grouping. If this is a unique junction, then it is also a primary junction.

5.1.14 Number of Upstream and Downstream Junctions (nb_up_juncs, nb_down_juncs)

The number of upstream and downstream junctions contained within any MSRs associated with this junction. Will be 0 for junctions without any MSRs.

5.1.15 Distance to nearest Upstream and Downstream Junctions (dist_2_up_junc, dist_2_down_junc, dist_nearest_junc)

Specifies the distance to the nearest junction detected upstream and downstream respectively (and the minimum or both).

5.1.16 Split Read Overhangs across each junction (JADxx)

Additional columns in the tab file are provided to represent the quantity of split read overhangs across each junction, up to 20bp upstream or downstream. This is similar, but more restricted, than the implementation in finesplice. The reason for the restriction is to ensure a consistent set of metrics (20) for all read lengths. The idea of this set of metrics in general is to provide a more finegrained indication of the *entropy* of the junction.

$$O^i_j = \min(L^i_j, r^i_j)$$

where:

- L^i_j defines the length of the left arm of read i across the junction j, trimmed to the first mismatching position
- R^i_j defines the length of the right arm of read i across the junction j, trimmed to the first mismatching position

We increment a vector N^j_i where i ranges from 1 to 20 for each junction representing pileups of O^i_j.

Using this vector we are able to provide some potential indications of whether the junction is genuine or not. To this end we have to columns marked: *suspicious* and *pfp* (for potential false positive).

5.2 Extra metrics

These metrics are only calculated if the user specifies that they wish to do additional processing. Note, that calculating these metrics can take a significant amount of time and has no direct influence on downstream filtering. It only makes sense to request extra processing if the user is specifically interest in these metrics.

5.2.1 Multiple Mapping Score (mm_score)

The multiple mapping score is the number of spliced reads associated with the junction divided by the number of times those same reads are found mapped anywhere in the genome. Therefore a score of 1 indicates that all spliced reads associated with the junction are only found in this junction. A low score would indicate that the those reads map to multiple locations across the genome. Originally described in TrueSight paper.

5.2.2 Unspliced coverage around junction (coverage)

When considering unspliced reads around a junction site, you would typically expect to see a tailing off of reads towards the 5' junction boundary, and a ramping up after the 3' junction boundary. However, in practice this is complicated by MSRs, alternative splicing and junctions near sequence ends.

5.2.3 Number of upstream and downstream flanking alignments (up_aln, down_aln)

This is a count of the number of unspliced reads aligning upstream of the splice junction, that overlap with the upstream anchor. Caution must be taken interpreting this metric closely packed introns could mean the presence of MSRs exclude the possibility of getting any unspliced upstream alignments. In addition, if the junction is close to the sequence start, it maybe that no unspliced upstream alignments are possible either.

5.2.4 Number of Samples

Portcullis can only be used on a single sample and will therefore always set to 1. Note that although portcullis can take multiple BAM files as input it will merge them at treat them as a single sample. However, downstream of portcullis it's useful to have a placeholder for the number of samples if applying set operations to multiple junction files. For example, if you were to create a union of 5 junction files it can be useful to know how many of those files contained the junction. That information is put into this metric.

5.3 Extracted metrics

By applying a set of rules based to junctions annotated with the metrics described in the previous section it is possible to define a subset of valid and invalid junctions with very high precision. However, there will invariably be many junctions left over that do not fit into either category. To assist with categorising the remaining junctions we use information from the high confidence sets to create additional metrics which are then calculated for all junctions. These extra metrics are described here:

5.3.1 Intron Score

Generally, long introns are not valid but mean intron lengths deviate wildly between species, hence we can't reliably filter on this criteria *a priori*. By scanning the positive set we can find the length of the intron at the 95th percentile L_{95} and then use this as a starting point for when junctions with excessively large introns look suspicious.

If $L^j < L_{95}$ then we assign a score of 0, otherwise we assign a score of $-ln(L^j - L_{95})$.

Metric originally used in TrueSight.

5.3.2 Splicing signal

By analysing the makeup of the genome around junctions in both the positive and negative sets we can try to get an idea whether certain genomic features are indicative of genuine junctions or not. A commonly used method to do this is to build markov models for k-mers upstream and downstream of the donor and acceptor splice sites in the junctions.

$$SS_{j(s,e)} = ln \sum_{i=s-3+k}^{s+19} \frac{P_{td}(X_i|X_{i-k}...X_{i-1})}{P_{fd}(X_i|X_{i-k}...X_{i-1})} + ln \sum_{i=e-20+k}^{e+19} \frac{P_{ta}(X_i|X_{i-k}...X_{i-1})}{P_{fa}(X_i|X_{i-k}...X_{i-1})}$$

where:

- P_{td} is the probability of a true donor given the following sequence
- P_{fd} is the probability of a false donor given the following sequence
- P_{ta} is the probability of a true acceptor given the following sequence
- P_{fa} is the probability of a false acceptor given the following sequence

Metric originally used in TrueSight.

5.3.3 Log deviation between expected and observed junction overhangs

We extend the observed pileup counts found by *Split Read Overhangs across each junction (JADxx)* to represent the log deviation between the observed and expected counts at each position in the junction to give us more discriminative power across datasets. To do this we use the following formula:

$$x_i^j = \log_2(\frac{N_i^j}{E_i^j})$$

where:

- E_i^j is the expected read count at this position in the junction assuming a uniform distribution of all observed split reads for this junction.

Note: Yes, theoretically this could be calculated in the junction analysis stage of portcullis!

5.4 Final metrics

Not all metrics turned out to be useful for determining whether a junction is genuine or not. We went through a process of feature selection and settled on the final set of metrics used in the machine learning part of portcullis. Those are listed here:

- *Number of distinct alignments (nb_dist_aln)*
- *Number of uniquely / multiply spliced alignments (nb_us_aln, nb_ms_aln)* (Uniquely splice reads only)
- *Number of Reliable Alignments (nb_rel_aln, rel2raw)*
- *Anchor lengths (max_min_anc, maxmmes)*
- *Mean mismatches (mean_mismatches)*
- *Intron Score*
- *Hamming distance at 5' and 3' splice sites (hamming5p, hamming3p)* (the minimum of 5' or 3')

- *Splicing signal*
- *Log deviation between expected and observed junction overhangs*

The output from the machine learning component is a probability score representing the liklihood that the junction is genuine or not. That value is represented by the `score` metric.

CHAPTER 6

Junctools

Portcullis contains a tools suite called junctools for interpretting, analysing and converting junction files. A description of all tools within junctools can be displayed by typing `junctools --help` at the command line:

```
usage: This script contains a number of tools for manipulating junction files.
       [-h] {compare,convert,gtf,markup,set,split} ...

optional arguments:
  -h, --help            show this help message and exit

Junction tools:
  {compare,convert,markup,set,split}
    compare             Compares junction files.
    convert             Converts junction files between various formats.
    gtf                 Filter or markup GTF files based on provided junctions
    markup              Marks whether each junction in the input can be found in␣
↪the reference or not.
    set                 Apply set operations to two or more junction files.
    split               Splits portcullis pass and fail juncs into 4 sets (TP, TN,␣
↪FP, FN) based on whether or not the junctions are found in the reference or not.
```

If this help message does not appear then please follow the instructions in the next section.

Note: Junctools is designed to support portcullis tab delimited files as well as many commonly used flavours of the bed format. Junctools will generally auto-detect the format based on the file extension and interpret the file as necessary.

6.1 Installation

If you have proceeded through the installation steps and have run `make install` then junctools will be installed along with portcullis as a python package to the location specified by `--prefix` when running the `configure` script. Should prefix not be specified then junctools will be installed to `/usr/local`. The junctools library reside in the `$(prefix)/lib/python3.x/site-packages` (where x is the minor version of python installed on your system) directory and the junctools executable will reside in `$(prefix)/bin`. If you wish to install junctools into a specific python environment or if you wish to install junctools without portcullis, first make sure

that you chosen environment has precedence on your system then then `cd <portcullis_dir>/scripts`, and then type `python3 setup.py install`.

Note: When you type `make uninstall` for portcullis you will also uninstall junctools from the location specified by $(prefix). Keep this in mind if you installed to a system directory you will if you prefix `sudo` to the uninstall command. Alternatively, if you installed junctools manually to a specific python environment then you can uninstall junctools with: `pip3 uninstall -y junctools`.

6.2 Compare

This tool compares one or more junction files against a reference file.

The usage information for this tool is as follows:

```
usage: Compares junction files.
       [-h] [-s] [-l LABELS] [-m] reference input [input ...]

positional arguments:
  reference             The junction file to treat as the reference
  input                 One or more junction files to compare against the
                        reference

optional arguments:
  -h, --help            show this help message and exit
  -s, --use_strand      Whether to use strand information when building keys
  -l LABELS, --labels LABELS
                        Path to a file containing labels for the reference
                        indicating whether or not each reference junction is
                        genuine (as generated using the markup tool). If
                        provided this script produces a much richer
                        performance analysis. Not compatible with '--
                        multiclass'
  -m, --multiclass      Breakdown results into multiple classes: 1) Matching
                        intron 2) Two matching splice sites but no matching
                        intron (i.e. splice sites from different introns) 3)
                        One matching splice site 4) No matching splice sites
```

Essentially this tool can be run in one of three modes depending on the options selected. The first mode does a information retrieval style comparison of the input files against a reference, as it's not possible to identify true negatives. The output contains counts of junctions, plus true positives, false positives and false negatives, along with precision, recall and F1 scores. The example below shows output from comparing 6 bed files against a reference bed using the following command `junctools compare ../ref.bed *.bed`:

```
Reference:
- # total junctions: 158156
- # distinct junctions: 158156

File     distinct    total     TP        FP      FN      REC    PRC    F1
f1.bed   146800      146800    135570    11230   22586   85.72  92.35  88.91
f2.bed   146778      146800    0         146778  158156  0.00   0.00   0.00
f3.bed   130905      130905    129103    1802    29053   81.63  98.62  89.33
f4.bed   130905      130905    129103    1802    29053   81.63  98.62  89.33
f5.bed   118865      118865    117569    1296    40587   74.34  98.91  84.88
f6.bed   117613      117613    113952    3661    44204   72.05  96.89  82.64

Mean recall:  65.89
Mean precision:  80.90
Mean f1:  72.51
```

Note: Please keep in mind that if comparing predicted junctions from a real RNAseq sample against a reference that the sample may contain genuinely novel junctions that will appear as false positives. Generally we use this method on simulated datasets where we know the definitive set of true junctions.

The second mode provides a more detailed analysis which is useful for comparing portcullis results against marked up junctions from an alignment tool. In this case we markup junctions from an alignment tool using the *markup tool* . This is essentially a list specifying whether or not each junction found by the aligner is present in a reference or not. We can then compare results from portcullis against the marked up alignment junctions. This gives us a definitive set of false negative junctions, i.e. junctions from the aligner that were genuine but incorrectly marked as negative by portcullis.

Finally, the third mode is useful for comparing junctions from real RNAseq datasets against a real reference. This breaks down results into the 4 classes:

- 1 - Matching intron
- 2 - Two matching splice sites but no matching intron (i.e. splice sites from different introns)
- 3 - One matching splice site
- 4 - No matching splice sites

This approach allows the user to better understand the set of false positives produced from real datasets, and can give some indication of whether a junction is a novel junction or a false positive.

Note: By default we do not use strand information when determining the location of a junction. To clarify, it is possible that bed file contains multiple junctions with the same sequence, start and stop sites but with a different strand. By default `junctools compare` will collapse these as a duplicate junction. Although not immediately intuitive this allows us to circumvent problems from junctions that have unknown strand. This is important as some tools do not output strand information. However, should you wish to disable this feature you can do so with the `--use_strand` option.

6.3 Convert

This can convert junction files between various commonly used formats. The conversion tool supports the following commonly used formats:

- bed = (Input only) BED format - we automatically determine if this is BED 6 or 12 format, as well as if it is intron, exon or tophat style).
- ebed = (Output only) Portcullis style exon-based BED12 format (Thick-start and end represent splice sites).
- tbed = (Output only) Tophat style exon-based BED12 format (splice sites derived from blocks).
- ibed = (Output only) Intron-based BED12 format.
- bed6 = (Output only) BED6 format (BED6 files are intron-based).
- gtf = (Input only) Transcript assembly or gene model containing transcript and exon features. NOTE: output will only contain junctions derived from this GTF.
- gff = (Input only) Transcript assembly or gene model containing introns to extract. NOTE: input must contain "intron" features, and output will only contain these introns represented as junctions.
- egff = (Output only) Exon-based junctions in GFF3 format, uses partial matches to indicate exon anchors.
- igff = (Output only) Intron-based junctions in GFF3 format

In addition we support the following application specific tab delimited formats:

- portcullis = Portcullis style tab delimited output.

- hisat = HISAT style tab delimited format.
- star = STAR style tab delimited format.
- finesplice = Finesplice style tab delimited format.
- soapslice = Soapsplice style tab delimited format.
- spanki = SPANKI style tab delimited format.
- truesight = Truesight style tab delimited format.

The usage information for the conversion tool looks like this:

```
usage: Converts junction files between various formats.
       [-h] -if INPUT_FORMAT -of OUTPUT_FORMAT [-o OUTPUT] [-is] [-d] [-s]
       [-r] [--index_start INDEX_START] [--prefix PREFIX] [--source SOURCE]
       input

positional arguments:
  input                 The input file to convert

optional arguments:
  -h, --help            show this help message and exit
  -if INPUT_FORMAT, --input_format INPUT_FORMAT
                        The format of the input file to convert.
  -of OUTPUT_FORMAT, --output_format OUTPUT_FORMAT
                        The output format.
  -o OUTPUT, --output OUTPUT
                        Output to this file.  By default we print to stdout.
  -is, --ignore_strand  Whether or not to ignore strand when creating a key for␣
↪the junction
  -d, --dedup           Whether or not to remove duplicate junctions
  -s, --sort            Whether or not to sort the junctions.  Note that sorting␣
↪requires all junctions to be loaded into memory first.  This maybe an issue for␣
↪very large input files.
  -r, --reindex         Whether or not to reindex the output.  The index is␣
↪applied after prefix.
  --index_start INDEX_START
                        The starting index to apply if the user requested␣
↪reindexing
  --prefix PREFIX       The prefix to apply to junction ids if the user requested␣
↪reindexing
  --source SOURCE       Only relevant if output is GFF format, use this option to␣
↪set the source column in the GFF
```

Note: The user can also use the conversion tool to deduplicate, sort and reindex junction files.

6.4 GTF

Provides a means of manipulating or analysing GTF files using a junctions file. Three modes are currently supported, the first two filter and markup, will process a GTF file and either remove, or mark, transcripts and their associated exons, if junctions within that transcript are not supported by the provided junction file. In compare mode, we benchmark GTF files based on whether junctions present are found in a reference junction file. This gives junction-level accuracy statistics as well as transcript-level stats.

Usage information follows:

```
usage: This script contains a number of tools for manipulating junction files. gtf
       [-h] [-is] -j JUNCTIONS [-o OUTPUT] mode input [input ...]
```

(continues on next page)

(continued from previous page)

```
GTF modes:
filter   = Filters out transcripts from GTF file that are not supported by the␣
↪provided
           junction file.
markup   = Marks transcripts from GTF file with 'portcullis' attribute, which␣
↪indicates
           if transcript has a fully supported set of junctions, or if not, which␣
↪ones are
           not supported.
compare  = For each GTF provided in the input. compare mode creates statistics␣
↪describing
           how many transcripts contain introns that are supported by a junction␣
↪file.

positional arguments:
  mode                  GTF operation to apply.  See above for details.  Available␣
↪options:
                         - filter
                         - markup
                         - compare
  input                 The input GTF file to convert

optional arguments:
  -h, --help            show this help message and exit
  -is, --ignore_strand  Whether or not to ignore strand when looking for junctions
  -j JUNCTIONS, --junctions JUNCTIONS
                        The file containing junctions that should be found in the␣
↪GTF.
  -o OUTPUT, --output OUTPUT
                        The filtered or markedup GTF output.  By default we print␣
↪to stdout.
```

6.5 Markup

This tool marks whether each junction in the input can be found in the reference or not. Output from the tool is a line seperated list of 1's (indicating junction is found in the reference) and 0's (indicating the junction is not found in the reference). Output is written to a file with the same name as the input except a '.res' extension is added. Usage information follows:

```
usage: Marks whether each junction in the input can be found in the reference or␣
↪not.
       [-h] [-o OUTPUT_DIR] [-s] reference input [input ...]

positional arguments:
  reference             The junction file to treat as the reference
  input                 One or more junction files to compare against the
                        reference

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_DIR, --output_dir OUTPUT_DIR
                        If output dir is specified this will create output
                        files for each input file with a .res extension
                        indicating whether or not the junction was found in
                        the reference. By default we write out a .res file in
                        the same directory as the input file was found in.
  -s, --use_strand      Whether to use strand information when building keys
```

6.6 Set

Apply set operations to two or more junction files. This tool supports various different ways to apply set operations between junction files. First you can merge two or more junction files using the following modes:

- intersection = Produces the intersection of junctions from multiple input files
- union = Produces the union of junctions from multiple input files
- consensus = If there are 3 or more input files, the consensus operation produces a merged set of junctions where those junctions are found across a user-defined number of input files

Output from these modes potentially involves mergeing multiple junctions from various files into a single representative. When this occurs junction anchors are extended representing the most extreme extents found across all junctions at the given site. In addition, the junction score is modified according to the setting selected by the user, by default this involves summing the scores of all junctions, although the user can alternatively choose to take the min, max or mean of the values.

The following modes only support two input files and produce an output file containing junctions which are taken directly from the input without modification:

- **subtract = Produces an output set of junctions containing all junctions present** in the first input file that also are not found in the second file
- **filter = Produces an output set of junctions containing all junctions present** in the first input file that are also found in the second file. This is similar to an intersection on two files except that duplicates and additional content assigned to junctions in the first file are retained
- **symmetric_difference =** Produces an output set containing junctions from both input files that are not present in the intersection of both

In addition, these test modes also only support 2 input files and return either True or False depending on the test requested:

- is_subset = Returns True if all junctions in the first file are present in the second
- is_superset = Returns True if all junctions in the second file are present in the first
- is_disjoint = Returns True if there is a null intersection between both files

Usage:

```
usage: Apply set operations to two or more junction files.
       [-h] [-m MIN_ENTRY] [--operator OPERATOR] [-o OUTPUT] [-p PREFIX] [-is]
       mode input [input ...]

positional arguments:
  mode                  Set operation to apply.  Available options:
                         - intersection
                         - union
                         - consensus
                         - subtract
                         - symmetric_difference
                         - is_subset
                         - is_superset
                         - is_disjoint
  input                 List of junction files to merge (must all be the same file␣
↪format)

optional arguments:
  -h, --help            show this help message and exit
  -m MIN_ENTRY, --min_entry MIN_ENTRY
                        Minimum number of files the entry is require to be in.  0␣
↪means entry must be
                        present in all files, i.e. true intersection.  1 means a␣
↪union of all input files
```

(continues on next page)

(continued from previous page)

```
  --operator OPERATOR   Operator to use for calculating the score in the merged␣
↪file.
                        This option is only applicable to 'intersection', 'union'␣
↪and 'consensus' modes.
                        Available values:
                         - min
                         - max
                         - sum
                         - mean
  -o OUTPUT, --output OUTPUT
                        Output junction file.  Required for operations that␣
↪produce an output file.
  -p PREFIX, --prefix PREFIX
                        Prefix to apply to name column in BED output file
  -is, --ignore_strand  Whether or not to ignore strand when creating a key for␣
↪the junction
```

6.7 Split

This tool splits portcullis pass and fail juncs into 4 sets (TP, TN, FP, FN) based on whether or not the junctions are found in the reference. The pass and fail files passed into this tool should be disjoint in order to get meaningful results.

Usage:

```
usage: Splits portcullis pass and fail juncs into 4 sets (TP, TN, FP, FN) based on␣
↪whether or not the junctions are found in the reference or not.
       [-h] [-o OUTPUT_PREFIX] reference passfile failfile

positional arguments:
  reference             The reference junction file
  passfile              The junction file containing junctions that pass a
                        filter
  failfile              The junction file containing junctions failing a
                        filter

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_PREFIX, --output_prefix OUTPUT_PREFIX
                        Prefix for output files
```

CHAPTER 7

Frequently Asked Questions

7.1 Can I reduce portcullis' memory usage?

Portcullis is optimised to reduce memory usage where possible, so in most cases memory usage should not be an issue if you were able to align your reads in the first place. However, if you are encountering out of memory issues, there are a few factors that can influence memory usage in portcullis, particularly during the junction analysis stage. The first is the number of threads used, more threads require more memory. You can therefore reduce memory usage at the expense of runtime. Second, we have an ability to process additional metrics in the junction analysis stage called `--extra`, this can require large amounts of memory, so unless you really need this option on (there should be no direct impact on the default filtering setup) the switch this off. Finally, the main driver for memory usage is the depth of the dataset, specifically highly covered junctions. Should have have tens of thousands of reads supporting a single junction, all these reads must be kept in memory. You can therefore reduce memory usage by pre-processing the BAM files to either cap the depth (you can use the Kmer Analysis Toolkit to identify high coverage kmers then remove reads associated with those kmers), or downsample using `samtools view -s`.

CHAPTER 8

Issues

Should you discover any issues with portcullis, or wish to request a new feature please raise a ticket at https://github.com/maplesond/portcullis/issues. Alternatively, contact Daniel Mapleson at: daniel.mapleson@earlham.ac.uk

CHAPTER 9

Availability and License

Open source code available on github: https://github.com/maplesond/portcullis.git

Spectre is available under GNU GLP V3: http://www.gnu.org/licenses/gpl.txt

CHAPTER 10

Additional Resources

Portcullis was presented at the Genome Science 2016 conference: `poster`; `slides`

CHAPTER 11

Authors and Acknowledgements

Daniel Mapleson - Project lead, software developer

David Swarbreck came up with the original plan and has helped design and guide the project from day 1.

Luca Venturini made the logic for the rule-based filtering and has been constantly testing the software helping to find bugs, improve runtime performance at every stage.

Thanks to **Gemy Kaithokattil** and **Shabonham Caim** for hunting bugs and providing valuable feedback and **Sarah Bastkowski** for helping me understanding the various machine learning algorithms and the maths behind it. Also thanks to **Chris Bennett** for making the logo.

Finally, thanks to the Earlham Insitute and the NBI computing services for supporting the work.