MONASH University

Fluent statistical computing interfaces for biological data analysis

Stuart Andrew Lee

B.MatComSci., University of Adelaide; MSc.Bioinformatics.

University of Melbourne

A thesis submitted for the degree of Doctor of Philosophy at

Monash University in 2020

Department of Econometrics and Business Statistics

Contents

Copyright notice

Abstract

Declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any university or equivalent institution, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 3 publications, two of which have been published and one which has not yet been submitted (Lee, Cook, and Lawrence, 2019; Lee, Lawrence, and Love, 2020) . As the core theme of my thesis is the development of software interfaces for biological data analysis, and given the collaborative nature of statistical computing and bioinformatics research; all of the included papers in this thesis reflect and acknowledge the contributions of my co-authors, including my primary supervisors Professor Dianne Cook (DC) in the Department of Econometrics and Business Statistics at Monash University and Associate Professor Matthew Ritchie in the Epigenetics and Development Division at the Walter and Eliza Hall Institute. The following table details the publications and my and my fellow co-authors contributions:

Thesis Chapter	Publication Title	Status (published, in press, accepted or returned for revision)	Nature and % of student contribution	Co-author name(s) Nature and % of Co-author's contribution	Co-author(s), Monash student Y/N
2	plyranges: a grammar of genomic data transformation	Published	70%. Concept, software development, data analysis, and manuscript writing	(1) Dianne Cook, Concept and manuscript revision 10% (2) Michael Lawrence, Concept and software development 20%	N
3	Fluent genomics with *plyranges* and *tximeta*	Published (awaiting peer review)	60%. Concept, software development, data analysis and manuscript writing	(1) Michael Lawrence, manuscript feedback and editing 5% (2) Michael I Love, Concept, data analysis and manuscript writing 35%	N
5	Casting multiple shadows: high-dimensional interactive data visualisation with tours and embeddings	To be submitted	80%. Concept, software development, and manuscript writing	(1) Dianne Cook, Concept and manuscript writing 15%	N

Note that chapter 4, reflects my contribution (in the form of a software package and data analysis) to the submitted manuscript entitled *Covering all your bases: incorporating intron signal from RNA-seq data* (Lee et al., 2019) and is joint work under the supervision of Dr Charity Law and Associate Professor Matthew Ritchie.

Student name: Stuart Andrew Lee

Student signature:

Acknowledgements

Preface

Chapter 1

Introduction

Exploratory data analysis (EDA) is a vital element of the modern statistical workflow - it is an analyst's first pass at understanding their data; revealing all it's messes and (possibly) uncovering hidden insights (Tukey, 1977; Grolemund and Wickham, 2017). It is an iterative process involving both computation and visualization that can generate new hypotheses that can be tested and formalised using statistical modelling. Moreover, as datasets grow in complexity and become increasingly heterogeneous and multidimensional, the use of EDA becomes vital to ensure the integrity and quality of analysis outputs. This is certainly true in high-throughput biological data analysis, where there are constraints on computation time and memory, leading to the situation that as data becomes larger, EDA becomes increasingly difficult.

The approach taken by the suite of software packages collectively known as the *tidyverse* is an attempt to formalise aspects of the EDA process in the R programming language under a single semantic known as *tidy data* (R Core Team, 2019a; Wickham et al., 2019; Wickham, 2014). Simply put, a *tidy data* set is a rectangular table where each row of the table corresponds to an observation and each column corresponds to a variable. There is a surprisingly large amount of utility that can be achieved with this definition. By having each column representing a variable, variables can be mapped to graphical aesthetics of visualizations. This enables the grammar of graphics as implemented by *ggplot2* (Wickham, 2016; Wilkinson, 2005). User interfaces as implemented by *tidyverse* and in

particular the *dplyr* package are *fluent*; they form a domain specific language (DSL) that gives users a mental model for performing and composing common data transformation tasks (Wickham et al., 2017; Fowler, n.d.).

It is unclear whether the *fluent* interfaces as implemented using the *tidy data* framework can be more generally applied and useful in fields such as high-throughput biology where domain specific semantics as implemented through more complex data structures are required. This is particularly true in the Bioconductor ecosystem, where much thought has gone into the design of data structures that enable interoperability between different tools (Huber et al., 2015a). One contribution of this thesis is the implementation of a *fluent* interface for genomics data that is expressive enough to enable EDA and be used as a building block for more software.

1.1 A grammar for genomic data analysis

1.2 Integration and representation of genomic data structures

1.3 Visual analytics for dimension reduction and clustering workflows

1.4 Summary

This thesis presents a set of tools for performing exploratory data analysis in the context of genomics and transcriptomics. The first tool, "plyranges" provides a domain specific language for reasoning about range based genomics data that simplifies common operations such as overlaps and aggregations. Next I extend "plyranges" to develop "superintronic" which presents an approach for discovering interesting genomic regions within an experimental design. We apply this tool to explore intron signal in transcriptomic data. Finally, I turn my attention to focus on embedding methods currently used for visualisation of large data sets…

Chapter 2

plyranges: a grammar of data transformation for genomics

There is a cognitive load placed on users in learning a data abstraction from the Bioconductor project and understanding its appropriate use. Users must navigate these abstractions to perform a genomic analysis task, when a single data abstraction, a GRanges object will suffice. By recognizing that the GRanges class follows 'tidy' data principles, we create a grammar of genomic data transformation, defining verbs for performing actions on and between genomic interval data and providing a way of performing common data analysis tasks through a coherent interface to existing Bioconductor infrastructure. We implement this grammar as a Bioconductor/R package called `plyranges`.

2.1 Background

High-throughput genomics promises to unlock new disease therapies, and strengthen our knowledge of basic biology. To deliver on those promises, scientists must derive a stream of knowledge from a deluge of data. Genomic data is challenging in both scale and complexity. Innovations in sequencing technology often outstrip our capacity to process the output. Beyond their common association with genomic coordinates, genomic data are heterogeneous, consisting of raw sequence read alignments, genomic feature annotations like genes and exons, and summaries like coverage vectors, ChIP-seq peak calls, variant

calls, and per-feature read counts. Genomic scientists need software tools to wrangle the different types of data, process the data at scale, test hypotheses, and generate new ones, all while focusing on the biology, not the computation. For the tool developer, the challenge is to define ways to model and operate on the data that align with the mental model of scientists, and to provide an implementation that scales with their ambition.

Several domain specific languages (DSLs) enable scientists to process and reason about heterogeneous genomics data by expressing common operations, such as range manipulation and overlap-based joins, using the vocabulary of genomics. Their implementations either delegate computations to a database, or operate over collections of files in standard formats like BED. An example of the former is the Genome Query Language (GQL) and its distributed implementation GenAp which use a SQL-like syntax for fast retrieval of information of unprocessed sequencing data (Kozanitis, Christos et al., 2014; Kozanitis and Patterson, 2016). Similarly, the Genometric Query Language (GMQL) implements a DSL for combining genomic datasets (Kaitoua, A et al., 2017). The command line application BEDtools develops an extensive algebra for performing arithmetic between two or more sets of genomic regions (Quinlan and Hall, 2010). All of the aforementioned DSLs are designed to be evaluated either at the command line or embedded in scripts for batch processing. They exist in a sparse ecosystem, mostly consisting of UNIX and database tools that lack biological semantics and operate at the level of files and database tables.

The Bioconductor/R packages `IRanges` and `GenomicRanges` (R Core Team, 2018; Lawrence et al., 2013a; Huber et al., 2015a) define a DSL for analyzing genomics data with R, an interactive data analysis environment that encourages reproducibility and provides high-level abstractions for manipulating, modelling and plotting data, through state of the art methods in statistical computing. The packages define object-oriented (OO) abstractions for representing genomic data and enable interoperability by allowing users and developers to use these abstractions in their own code and packages. Other genomic DSLs that are embedded in programming languages include pybedtools and valr (Dale, Pedersen, and Quinlan, 2011; Riemondy et al., 2017), however these packages lack the interoperability provided by the aforementioned Bioconductor packages and are not easily extended.

The Bioconductor infrastructure models the genomic data and operations from the perspective of the power user, one who understands and wants to take advantage of the subtle differences in data types. This design has enabled the development of sophisticated tools, as evidenced by the hundreds of packages depending on the framework. Unfortunately, the myriad of data structures have overlapping purposes and important but obscure differences in behavior that often confuse the typical end user.

Recently, there has been a concerted, community effort to standardize R data structures and workflows around the notion of tidy data (Wickham, 2014). A tidy dataset is defined as a tabular data structure that has observations as rows and columns as variables, and all measurements pertain to a single observational unit. The tidy data pattern is useful because it allows us to see how the data relate to the design of an experiment and the variables measured. The `dplyr` package (Wickham et al., 2017) defines an application programming interface (API) that maps notions from the general relational algebra to verbs that act on tidy data. These verbs can be composed together on one or more tidy datasets with the pipe operator from the `magrittr` package (Bache and Wickham, 2014). Taken together these features enable a user to write human readable analysis workflows.

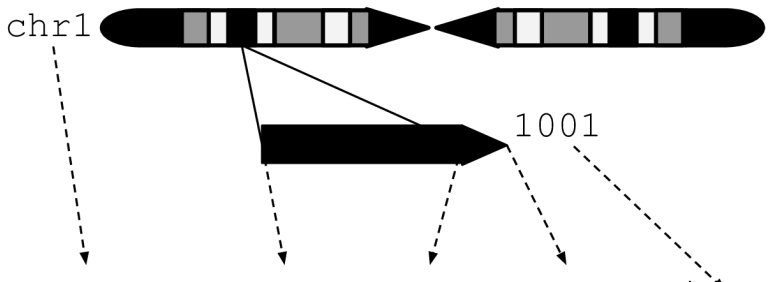
We have created a genomic DSL called `plyranges` that reformulates notions from existing genomic algebras and embeds them in R as a genomic extension of `dplyr`. By analogy, `plyranges` is to the genomic algebra, as `dplyr` is to the relational algebra. The `plyranges` Bioconductor package implements the language on top of a key subset of Bioconductor data structures and thus fully integrates with the Bioconductor framework, gaining access to its scalable data representations and sophisticated statistical methods.

2.2 Results

2.2.1 Genomic Relational Algebra

Data Model

The `plyranges` DSL is built on the core Bioconductor data structure GRanges, which is a constrained table, with fixed columns for the chromosome, start and end coordinates, and the strand, along with an arbitrary set of additional columns, consisting of measurements



seqname	start	end	strand	gene_id	score
chr1	102012	120303	+	1001	10
chr1	520211	526211	-	2151	25
...	...	...	...	...	...

Figure 2.1: *An illustration of the GRanges data model for a sample from an RNA-seq experiment. The core components of the data model include a seqname column (representing the chromosome), a ranges column which consists of start and end coordinates for a genomic region, and a strand identifier (either positive, negative, or unstranded). Metadata are included as columns to the right of the dotted line as annotations (gene_id) or range level covariates (score).*

or metadata specific to the data type or experiment (figure 2.1). GRanges balances flexibility with formal constraints, so that it is applicable to virtually any genomic workflow, while also being semantically rich enough to support high-level operations on genomic ranges. As a core data structure, GRanges enables interoperability between `plyranges` and the rest of Bioconductor. Adhering to a single data structure simplifies the API and makes it easier to learn and understand, in part because operations become endomorphic, i.e., they return the same type as their input.

GRanges follow the intuitive tidy data pattern: it is a rectangular table corresponding to a single biological context. Each row contains a single observation and each column is a variable describing the observations. GRanges specializes the tidy pattern in that the observations always pertain to some genomic feature, but it largely remains compatible with the general relational operations defined by `dplyr`. Thus, we define our algebra as an extension of the `dplyr` algebra, and borrow its syntax conventions and design principles.

	Verb	Description
Aggregate	***summarize()***	aggregate over column(s)
	disjoin_ranges()	aggregate column(s) over the union of end coordinates
	reduce_ranges()	aggregate column(s) by merging overlapping and neighboring ranges
Modify (Unary)	***mutate()***	modifies any column
	select()	select columns
	arrange()	sort by columns
	stretch()	extend range by fixed amount
	shift_(direction)	shift coordinates
	flank_(direction)	generate flanking regions
Modify (Binary)	*%intersection%*	row-wise intersection
	%union%	row-wise union
	compute_coverage	coverage over all ranges
	%setdiff%	row-wise set difference
	between()	row-wise gap range
	span()	row-wise spanning range
Merge	*join_overlap_*()*	merge by overlapping ranges
	join_nearest	merge by nearest neighbor ranges
	join_follow	merge by following ranges
	join_precedes	merge by preceding ranges
	union_ranges	range-wise union
	intersect_ranges	range-wise intersect
	setdiff_ranges	range-wise set difference
	complement_ranges	range-wise set complement
Operate	*anchor_direction()*	fix coordinates at direction
	group_by()	partition by column(s)
	group_by_overlaps()	partition by overlaps
Restrict	***filter()***	subset rows
	filter_by_overlaps()	subset by overlap
	filter_by_non_overlaps()	subset by no overlap

Table 2.1: *Overview of the* `plyranges` *grammar. The core verbs are briefly described and categorized into one of the following higher level categories: aggregate, modify, merge, operate, or restrict. A verb is given bold text if its origin is from the* `dplyr` *grammar.*

Algebraic operations

The `plyranges` DSL defines an expressive algebra for performing genomic operations with and between GRanges objects (see table 2.1). The grammar includes several classes of operation that cover most use cases in genomics data analysis. There are range arithmetic operators, such as for resizing ranges or finding their intersection, and operators for merging, filtering and aggregating by range-specific notions like overlap and proximity.

Arithmetic operations transform range coordinates, as defined by their *start*, *end* and *width*. The three dimensions are mutually dependent and partially redundant, so direct manipulation of them is problematic. For example, changing the *width* column needs to change either the *start*, *end* or both to preserve integrity of the object. We introduce the *anchor* modifier to disambiguate these adjustments. Supported anchor points include the start, end and midpoint, as well as the 3′ and 5′ ends for strand-directed ranges. For example, if we anchor the start, then setting the width will adjust the end while leaving the start stationary.

The algebra also defines conveniences for relative coordinate adjustments: *shift* (unanchored adjustment to both start and end) and *stretch* (anchored adjustment of width). We can perform any relative adjustment by some combination of those two operations. The *stretch* operation requires an anchor and assumes the midpoint by default. Since *shift* is unanchored, the user specifies a suffix for indicating the direction: left/right or, for stranded features, upstream/downstream. For example, *shift_right* shifts a range to the right.

The *flank* operation generates new ranges that are adjacent to existing ones. This is useful, for example, when generating upstream promoter regions for genes. Analogous to *shift*, a suffix indicates the side of the input range to flank.

As with other genomic grammars, we define set operations that treat ranges as sets of integers, including *intersect*, *union*, *difference*, and *complement*. There are two sets of these: parallel and merging. For example, the parallel intersection (x *%intersect%* y) finds the intersecting range between xi and yi for i in $1 \ldots n$, where n is the length of both x and y. In contrast, the merging intersection (*intersect_ranges*(x, y)) returns a new set of disjoint

ranges representing wherever there was overlap between a range in x and a range in y. Finding the parallel union will fail when two ranges have a gap, so we introduce a *span* operator that takes the union while filling any gap. The *complement* operation is unique in that it is unary. It finds the regions not covered by any of the ranges in a single set. Closely related is the *between* parallel operation, which finds the gap separating xi and yi. The binary operations are callable from within arithmetic, restriction and aggregation expressions.

To support merging, our algebra recasts finding overlaps or nearest neighbors between two genomic regions as variants of the relational join operator. A join acts on two GRanges objects: x and y. The join operator is relational in the sense that metadata from the x and y ranges are retained in the joined range. All join operators in the `plyranges` DSL generate a set of hits based on overlap or proximity of ranges and use those hits to merge the two datasets in different ways. There are four supported matching algorithms: *overlap*, *nearest*, *precede*, and *follow* (figure 2.2). We can further restrict the matching by whether the query is completely *within* the subject, and adding the *directed* suffix ensures that matching ranges have the same direction (strand).

For merging based on the hits, we have three modes: *inner*, *intersect* and *left*. The *inner* overlap join is similar to the conventional inner join in that there is a row in the result for every match. A major difference is that the matching is not by identity, so we have to choose one of the ranges from each pair. We always choose the left range. The *intersect* join uses the intersection instead of the left range. Finally, the overlap *left* join is akin to left outer join in Codd's relational algebra: it performs an overlap inner join but also returns all x ranges that are not hit by the y ranges.

Since the GRanges object is a tabular data structure, our grammar includes operators to filter, sort and aggregate by columns in a GRanges. These operations can be performed over partitions formed using the *group_by* modifier. Together with our algebra for arithmetic and merging, these operations conform to the semantics and syntax of the `dplyr` grammar. Consequently, `plyranges` code is generally more compact than the equivalent `GenomicRanges` code (figure 2.3).

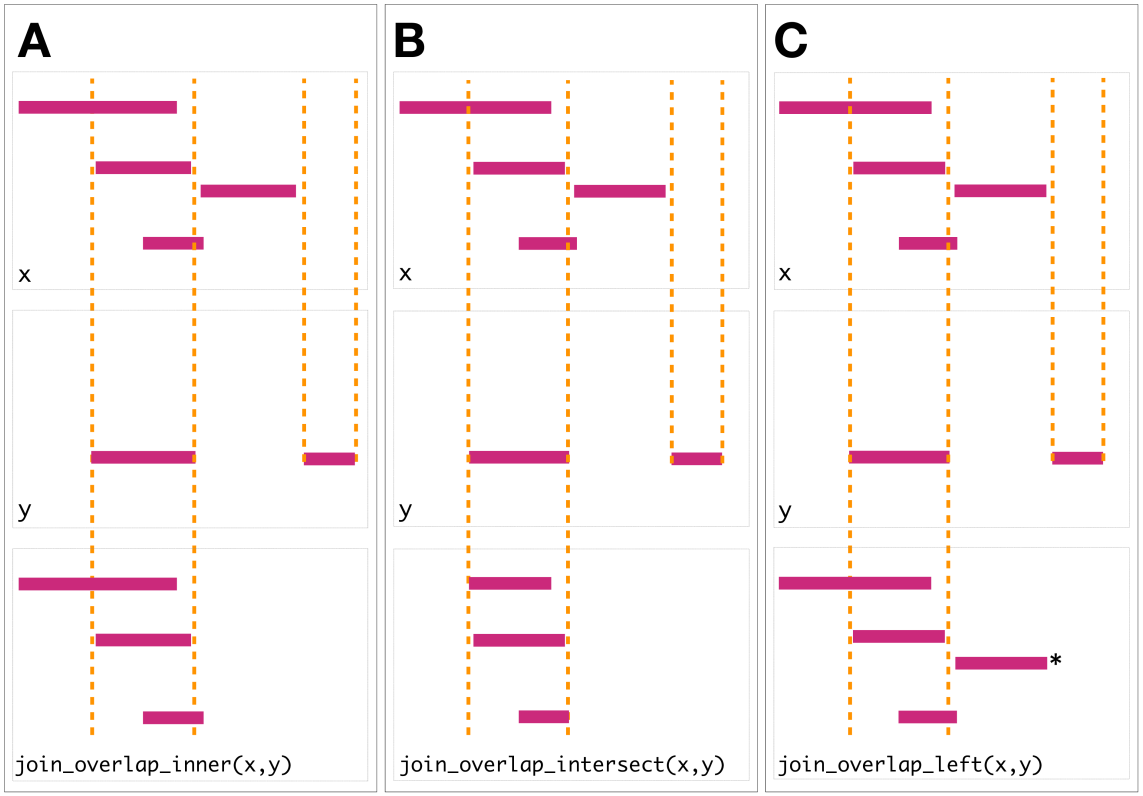


Figure 2.2: *Illustration of the three overlap join operators. Each join takes two GRanges objects, x and y as input. A 'Hits' object for the join is computed which consists of two components. The first component contains the indices of the ranges in x that have been overlapped (the rectangles of x that cross the orange lines). The second component consists of the indices of the ranges in y that overlap the ranges in x. In this case a range in y overlaps the ranges in x three times, so the index is repeated three times. The resulting 'Hits' object is used to modify x by where it was 'hit' by y and merge all metadata columns from x and y based on the indices contained in the 'Hits' object. This procedure is applied generally in the* `plyranges` *DSL for both overlap and nearest neighbor operations. The join semantics alter what is returned:* ***A****: for an* ***inner*** *join the x ranges that are overlapped by y are returned. The returned ranges also include the metadata from the y range that overlapped the three x ranges.* ***B*** *An* ***intersect*** *join is identical to an inner join except that the intersection is taken between the overlapped x ranges and the y ranges.* ***C*** *For the* ***left*** *join all x ranges are returned regardless of whether they are overlapped by y. In this case the third range (rectangle with the asterisk next to it) of the join would have missing values on metadata columns that came from y.*

A
```
library(plyranges)
gwas <- read_bed('snps.bed')
exons <- read_bed('exons.bed')
res <- exons %>%
          join_overlap_inner(snps) %>%
          group_by(rsID) %>%
          summarise(n = n_distinct(exonID))
```

B
```
library(GenomicRanges)
library(rtracklayer)
gwas  <- import('snps.bed')
exons <- import('exons.bed')
hits <- findOverlaps(exons, gwas,
                     ignore.strand = FALSE)
olap <- splitAsList(exons$name[queryHits(hits)],
                    gwas$name[subjectHits(hits)])
n <- lengths(unique(olap))
res <- DataFrame(rsID = names(n),
                 n = as.integer(n))
```

Figure 2.3: *Idiomatic code examples for* `plyranges` *(**A**) and* `GenomicRanges` *(**B**) illustrating an overlap and aggregate operation that returns the same result. In each example, we have two BED files consisting of SNPs that are genome-wide association study (GWAS) hits and reference exons. Each code block counts for each SNP the number of distinct exons it overlaps. The* `plyranges` *code achieves this with an overlap join followed by partitioning and aggregation. Strand is ignored by default here. The* `GenomicRanges` *code achieves this using the 'Hits' and 'List' classes and their methods.*

2.2.2 Developing workflows with `plyranges`

Here we provide illustrative examples of using the `plyranges` DSL to show how our grammar could be integrated into genomic data workflows. As we construct the workflows we show the data output intermittently to assist the reader in understanding the pipeline steps. The workflows highlight how interoperability with existing Bioconductor infrastructure, enables easy access to public datasets and methods for analysis and visualization.

Peak Finding

In the workflow of ChIP-seq data analysis, we are interested in finding peaks from islands of coverage over chromosome. Here we will use `plyranges` to call peaks from islands of coverage above 8 then plot the region surrounding the tallest peak.

Using `plyranges` and the the Bioconductor package `AnnotationHub` (Morgan, 2017) we can download and read BigWig files from ChIP-Seq experiments from the Human Epigenome Roadmap project (Roadmap Epigenomics Consortium et al., 2015). Here we analyse a BigWig file corresponding to H3 lysine 27 trimethylation (H3K27Me3) of primary T CD8+ memory cells from peripheral blood, focussing on coverage islands over chromosome 10.

First, we extract the genome information from the BigWig file and filter to get the range for chromosome 10. This range will be used as a filter when reading the file.

```
library(plyranges)
chr10_ranges <- bw_file %>%
  get_genome_info() %>%
  filter(seqnames == "chr10")
```

Then we read the BigWig file only extracting scores if they overlap chromosome 10. We also add the genome build information to the resulting ranges. This book-keeping is good practice as it ensures the integrity of any downstream operations such as finding overlaps.

```
chr10_scores <- bw_file %>%
  read_bigwig(overlap_ranges = chr10_ranges) %>%
  set_genome_info(genome = "hg19")
chr10_scores
```

```
#> GRanges object with 5789841 ranges and 1 metadata column:
#>             seqnames              ranges strand |              score
#>                <Rle>           <IRanges>  <Rle> |          <numeric>
#>         [1]    chr10             1-60602      * | 0.0422799997031689
```

```
#>         [2]    chr10         60603-60781      * |  0.163240000605583
#>         [3]    chr10         60782-60816      * |  0.372139990329742
#>         [4]    chr10         60817-60995      * |  0.163240000605583
#>         [5]    chr10         60996-61625      * | 0.0422799997031689
#>         ...      ...                 ...    ... .                ...
#>   [5789837]    chr10 135524723-135524734      * |  0.144319996237755
#>   [5789838]    chr10 135524735-135524775      * |  0.250230014324188
#>   [5789839]    chr10 135524776-135524784      * |  0.427789986133575
#>   [5789840]    chr10 135524785-135524806      * |  0.730019986629486
#>   [5789841]    chr10 135524807-135524837      * |   1.03103005886078
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

We then filter for regions with a coverage score greater than 8, and following this reduce individual runs to ranges representing the islands of coverage. This is achieved with the `reduce_ranges()` function, which allows a summary to be computed over each island: in this case we take the maximum of the scores to find the coverage peaks over chromosome 10.

```
all_peaks <- chr10_scores %>%
  filter(score > 8) %>%
  reduce_ranges(score = max(score))
all_peaks
```

```
#> GRanges object with 1085 ranges and 1 metadata column:
#>          seqnames              ranges strand |            score
#>             <Rle>           <IRanges>  <Rle> |        <numeric>
#>      [1]    chr10     1299144-1299370      * | 13.2264003753662
#>      [2]    chr10     1778600-1778616      * | 8.20512008666992
#>      [3]    chr10     4613068-4613078      * | 8.76027011871338
#>      [4]    chr10     4613081-4613084      * | 8.43659973144531
#>      [5]    chr10             4613086      * | 8.11507987976074
```

```
#>      ...      ...                 ...    ... .             ...
#>   [1081]    chr10 135344482-135344488      * | 9.23237991333008
#>   [1082]    chr10 135344558-135344661      * |  11.843409538269
#>   [1083]    chr10 135344663-135344665      * | 8.26965999603271
#>   [1084]    chr10 135344670-135344674      * | 8.26965999603271
#>   [1085]    chr10 135345440-135345441      * | 8.26965999603271
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

Returning to the GRanges object containing normalized coverage scores, we filter to find the coordinates of the peak containing the maximum coverage score. We can then find a 5000 nt region centered around the maximum position by anchoring and modifying the width.

Finally, the overlap inner join is used to restrict the chromosome 10 coverage islands, to the islands that are contained in the 5000nt region that surrounds the max peak (figure 2.4).

```
#> GRanges object with 890 ranges and 2 metadata columns:
#>         seqnames            ranges strand |            score.x          score.y
#>            <Rle>         <IRanges>  <Rle> |          <numeric>        <numeric>
#>     [1]    chr10 21805891-21805988      * | 0.0206599999219179 29.9573001861572
#>     [2]    chr10 21805989-21806000      * | 0.0211200006306171 29.9573001861572
#>     [3]    chr10 21806001-21806044      * |  0.022069999948144 29.9573001861572
#>     [4]    chr10 21806045-21806049      * | 0.0215900000184774 29.9573001861572
#>     [5]    chr10 21806050-21806081      * | 0.0211200006306171 29.9573001861572
#>     ...      ...               ...    ... .                ...              ...
#>   [886]    chr10          21810878      * |   5.24951982498169 29.9573001861572
#>   [887]    chr10          21810879      * |   5.83534002304077 29.9573001861572
#>   [888]    chr10 21810880-21810884      * |   6.44267988204956 29.9573001861572
#>   [889]    chr10 21810885-21810895      * |   7.07054996490479 29.9573001861572
#>   [890]    chr10 21810896-21810911      * |   6.44267988204956 29.9573001861572
```

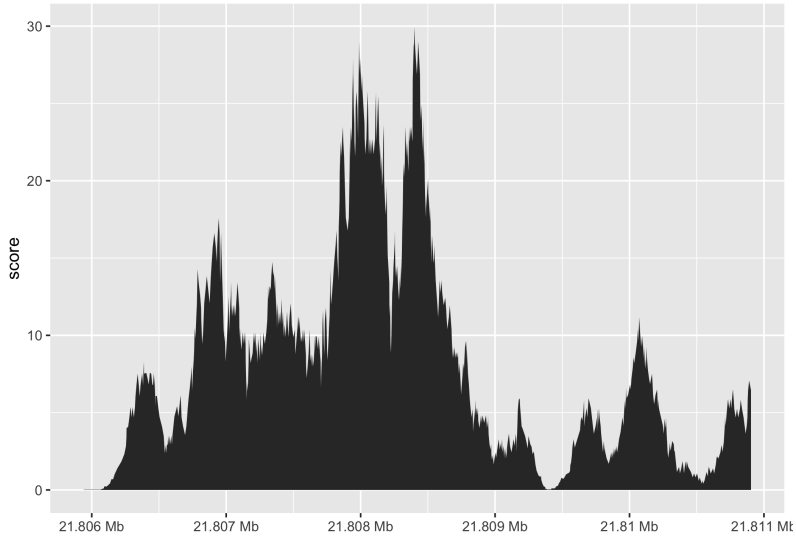


Figure 2.4: *The final result of the* `plyranges` *operations to find a 5000nt region surrounding the peak of normalised coverage scores over chromosome 10, displayed as a density plot.*

```
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

Computing Windowed Statistics

Another common operation in genomics data analysis is to compute data summaries over genomic windows. In `plyranges` this can be achieved via the `group_by_overlaps()` operator. We bin and count and find the average GC content of reads from a H3K27Me3 ChIP-seq experiment by the Human Epigenome Roadmap Consortium.

We can directly obtain the genome information from the header of the BAM file: in this case the reads were aligned to the hg19 genome build and there are no reads overlapping the mitochondrial genome.

```
bam <- read_bam(h1_bam_sorted, index = h1_bam_sorted_index)
locations <- bam %>%
  get_genome_info()
```

Next we only read in alignments that overlap the genomic locations we are interested in and select the query sequence. Note that the reading of the BAM file is deferred: only

alignments that pass the filter are loaded into memory. We can add another column representing the GC proportion for each alignment using the `letterFrequency()` function from the `Biostrings` package (Pagès et al., 2018). After computing the GC proportion as the score column, we drop all other columns in the GRanges object.

```
alignments <- bam %>%
  filter_by_overlaps(locations) %>%
  select(seq) %>%
  mutate(
    score = as.numeric(letterFrequency(seq, "GC", as.prob = TRUE))
  ) %>%
  select(score)
alignments
```

```
#> GRanges object with 8275595 ranges and 1 metadata column:
#>             seqnames            ranges strand |             score
#>                <Rle>         <IRanges>  <Rle> |         <numeric>
#>         [1]    chr10       50044-50119      - | 0.276315789473684
#>         [2]    chr10       50050-50119      + |              0.25
#>         [3]    chr10       50141-50213      - | 0.447368421052632
#>         [4]    chr10       50203-50278      + | 0.263157894736842
#>         [5]    chr10       50616-50690      + | 0.276315789473684
#>         ...      ...               ...    ... .               ...
#>   [8275591]     chrY 57772745-57772805      - | 0.513157894736842
#>   [8275592]     chrY 57772751-57772800      + | 0.526315789473684
#>   [8275593]     chrY 57772767-57772820      + | 0.565789473684211
#>   [8275594]     chrY 57772812-57772845      + |              0.25
#>   [8275595]     chrY 57772858-57772912      + | 0.592105263157895
#>   -------
#>   seqinfo: 24 sequences from an unspecified genome
```

Finally, we create 10000nt tiles over the genome and compute the number of reads and average GC content over all reads that fall within each tile using an overlap join and merging endpoints.

```
bins <- locations %>%
  tile_ranges(width = 10000L)


alignments_summary <- bins %>%
  join_overlap_inner(alignments) %>%
  disjoin_ranges(n = n(), avg_gc = mean(score))
alignments_summary
```

```
#> GRanges object with 286030 ranges and 2 metadata columns:
#>            seqnames            ranges strand |         n            avg_gc
#>               <Rle>         <IRanges>  <Rle> | <integer>         <numeric>
#>        [1]    chr10       49999-59997      * |        88 0.369019138755981
#>        [2]    chr10       59998-69997      * |        65 0.434210526315789
#>        [3]    chr10       69998-79996      * |        56 0.386513157894737
#>        [4]    chr10       79997-89996      * |        71  0.51297257227576
#>        [5]    chr10       89997-99996      * |        64 0.387746710526316
#>        ...      ...               ...    ... .       ...               ...
#>   [286026]     chrY 57722961-57732958      * |        36 0.468201754385965
#>   [286027]     chrY 57732959-57742957      * |        38 0.469529085872576
#>   [286028]     chrY 57742958-57752956      * |        38 0.542936288088643
#>   [286029]     chrY 57752957-57762955      * |        42 0.510651629072682
#>   [286030]     chrY 57762956-57772954      * |       504 0.526942355889723
#>   -------
#>   seqinfo: 24 sequences from an unspecified genome; no seqlengths
```

Quality Control Metrics

We have created a GRanges object from genotyping performed on the H1 cell line, consisting of approximately two million single nucleotide polymorphisms (SNPs) and short

insertion/deletions (indels). The GRanges object consists of 7 columns, relating to the alleles of a SNP or indel, the B-allele frequency, log relative intensity of the probes, GC content score over a probe, and the name of the probe. We can use this information to compute the transition-transversion ratio, a quality control metric, within each chromosome in GRanges object.

First we filter out the indels and mitochondrial variants. Then we create a logical vector corresponding to whether there is a transition event.

```
h1_snp_array <- h1_snp_array %>%
  filter(!(ref %in% c("I", "D")), seqnames != "M") %>%
  mutate(transition = (ref %in% c("A", "G") & alt %in% c("G","A"))|
                      (ref %in% c("C","T") & alt %in% c("T", "C")))
```

We then compute the transition-transversion ratio over each chromosome using `group_by()` in combination with `summarize()` (figure 2.5).

```
ti_tv_results <- h1_snp_array %>%
  group_by(seqnames) %>%
  summarize(n_snps = n(),
            ti_tv = sum(transition) / sum(!transition))
ti_tv_results
```

```
#> DataFrame with 24 rows and 3 columns
#>     seqnames    n_snps            ti_tv
#>     <factor> <integer>        <numeric>
#> 1          Y      2226  1.4381161007667
#> 2          6    154246 3.32013219807305
#> 3         13     83736 3.40669403220714
#> 4         10    120035 3.49400973418195
#> 5          4    153243 3.29528828096533
#> ...      ...       ...              ...
#> 20        16     77538 3.19827819589583
```

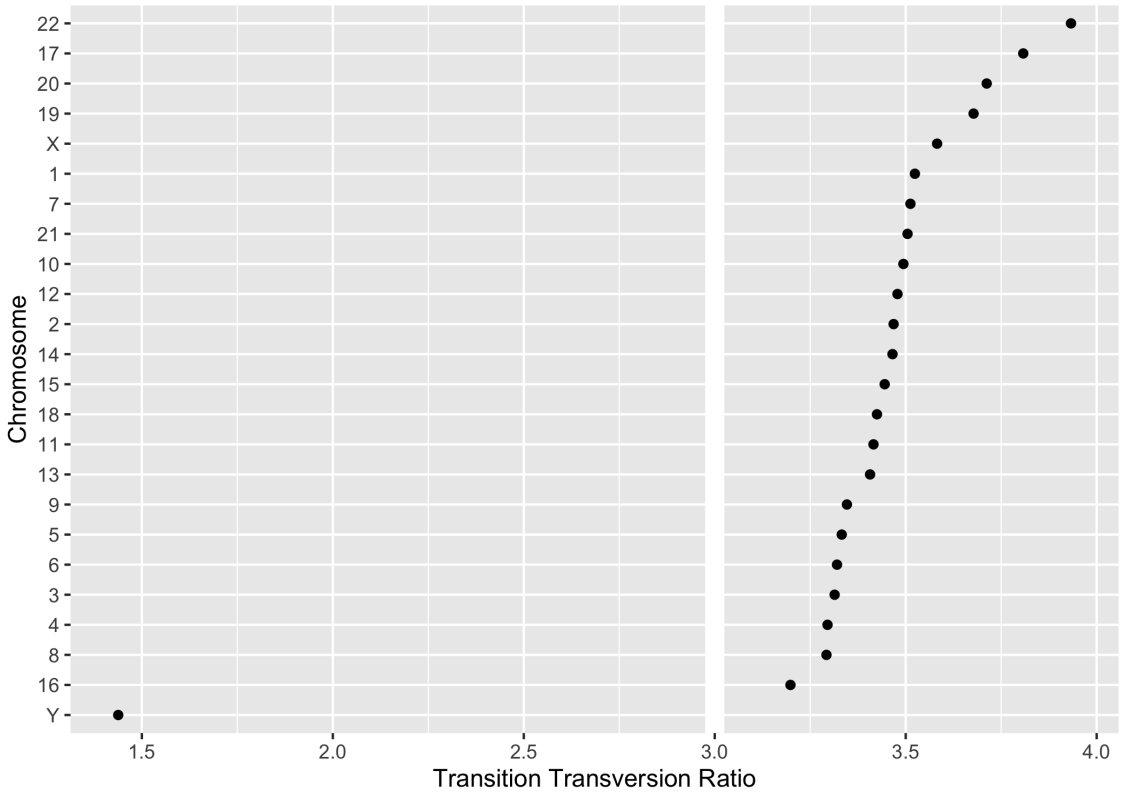


Figure 2.5: *The final result of computing quality control metrics over the SNP array data with* `plyranges`*, displayed as a dot plot. Chromosomes are ordered by their estimated transition-transversion ratio. A white reference line is drawn at the expected ratio for a human exome."*

```
#> 21        12    113208 3.47851887016378
#> 22        20     57073  3.7121036988111
#> 23        21     32349 3.50480434479877
#> 24         X     55495 3.58219800181653
```

2.3 Discussion

The design of `plyranges` adheres to well understood principles of language and API design: cognitive consistency, cohesion, endomorphism and expressiveness (Green and Petre, 1996). To varying degrees, these principles also underlie the design of `dplyr` and the Bioconductor infrastructure.

We have aimed for `plyranges` to have a simple and direct mapping to the user's cognitive model, i.e., how the user thinks about the data. This requires careful selection of the

level of abstraction so that the user can express workflows in the language of genomics. This motivates the adoption of the tidy GRanges object as our central data structure. The basic data.frame and `dplyr` tibble lack any notion of genomic ranges and so could not easily support our genomic grammar, with its specific verbs for range-oriented data manipulation. Another example of cognitive consistency is how `plyranges` is insensitive to direction/strand by default when, e.g., detecting overlaps. `GenomicRanges` has the opposite behavior. We believe that defaulting to purely spatial overlap is most intuitive to most users.

To further enable cognitive consistency, `plyranges` functions are cohesive. A function is defined to be cohesive if it performs a singular task without producing any side-effects. Singular tasks can always be broken down further at lower levels of abstraction. For example, to resize a range, the user needs to specify which position (start, end, midpoint) should be invariant over the transformation. The `resize()` function from the `GenomicRanges` package has a `fix` argument that sets the anchor, so calling `resize()` coalesces anchoring and width modification. The coupling at the function call level is justified since the effect of setting the width depends on the anchor. However, `plyranges` increases cohesion and decouples the anchoring into its own function call.

Increasing cohesion simplifies the interface to each operation, makes the meaning of arguments more intuitive, and relies on function names as the primary means of expression, instead of a more complex mixture of function and argument names. This results in the user being able to conceptualize the `plyranges` DSL as a flat catalog of functions, without having to descend further into documentation to understand a function's arguments. A flat function catalog also enhances API discoverability, particularly through auto-completion in integrated developer environments (IDEs). One downside of pushing cohesion to this extreme is that function calls become coupled, and care is necessary to treat them as a group when modifying code.

Like `dplyr`, `plyranges` verbs are functional: they are free of side effects and are generally endomorphic, meaning that when the input is a GRanges object they return a GRanges object. This enables chaining of verbs through syntax like the forward pipe operator from the `magrittr` package. This syntax has a direct cognitive mapping to natural language and

the intuitive notion of pipelines. The low-level object-oriented APIs of Bioconductor tend to manipulate data via sub-replacement functions, like `start(gr) <- x`. These ultimately produce the side effect of replacing a symbol mapping in the current environment and thus are not amenable to so-called fluent syntax.

Expressiveness relates to the information content in code: the programmer should be able to clarify intent without unnecessary verbosity. For example, our overlap-based join operations are more concise than the multiple steps necessary to achieve the same effect in the original `GenomicRanges` API. In other cases, the `plyranges` API increases verbosity for the sake of clarity and cohesion. Explicitly calling *anchor()* can require more typing, but the code is easier to comprehend. Another example is the set of routines for importing genomic annotations, including *read_gff()*, *read_bed()*, and *read_bam()*. Compared to the generic *import()* in `rtracklayer`, the explicit format-based naming in `plyranges` clarifies intent and the type of data being returned. Similarly, every `plyranges` function that computes with strand information indicates its intentions by including suffixes such as *directed*, *upstream* or *downstream* in its name, otherwise strand is ignored. The `GenomicRanges` API does not make this distinction explicit in its function naming, instead relying on a parameter that defaults to strand sensitivity, an arguably confusing behavior.

The implementation of `plyranges` is built on top of Bioconductor infrastructure, meaning most functions are constructed by composing generic functions from core Bioconductor packages. As a result, any Bioconductor packages that uses data structures that inherit from GRanges will be able to use `plyranges` for free. Another consequence of building on top of Bioconductor generics is that the speed and memory usage of `plyranges` functions are similar to the highly optimized methods implemented in Bioconductor for GRanges objects.

A caveat to constructing a compatible interface with `dplyr` is that `plyranges` makes extensive use of non-standard evaluation in R via the `rlang` package (Henry and Wickham, 2017). Simply, this means that computations are evaluated in the context of the GRanges objects. Both `dplyr` and `plyranges` are based on the `rlang` language, because it allows for more expressive code that is free of repeated references to the container. Implicitly referencing the container is particularly convenient when programming interactively.

Consequently, when programming with `plyranges`, a user needs to generally understand the `rlang` language and how to adapt their code accordingly. Users familiar with the tidyverse should already have such knowledge.

2.4 Conclusion

We have shown how to create expressive and reproducible genomic workflows using the `plyranges` DSL. By realising that the GRanges data model is tidy we have highlighted how to implement a grammar for performing genomic arithmetic, aggregation, restriction and merging. Our examples show that `plyranges` code is succinct, human readable and can take advantage of the interoperability provided by the Bioconductor ecosystem and the R language.

We also note that the grammar elements and design principles we have described are programming language agnostic and could be easily be implemented in another language where genomic information could be represented as a tabular data structure. We chose R because it is what we are familiar with and because the aforementioned Bioconductor packages have implemented the GRanges data structure.

We aim to continue developing the `plyranges` package and to extend it for use with more complex data structures, such as the SummarizedExperiment class, the core Bioconductor data structure for representing experimental results (e.g., counts) from multiple sample experiments in conjunction with feature and sample metadata. Although, the SummarizedExperiment is not strictly tidy, it does consist of three tidy data structures that are related by feature and sample identifiers. Therefore, the grammar and design of the `plryanges` DSL is naturally extensible to the SummarizedExperiment.

As the `plyranges` interface encourages tidy data practices, it integrates well with the grammar of graphics (Wickham, 2016). To achieve responsive performance, interactive graphics rely on lazy data access and computing patterns, so the deferred mechanisms within `plyranges` should help support interactive genomics applications.

2.5 Availability of Data and Materials

The BigWig file for the H3K27Me3 primary T CD8+ memory cells from peripheral blood ChIP-seq data from the Human Roadmap Epigenomics project
was downloaded from the `AnnotationHub` package (2.13.1) under accession AH33458 (Morgan, 2017; Roadmap Epigenomics Consortium et al., 2015). The BAM file corresponding to the H1 cell line ChIP-seq data is available at NCBI GEO under accession GSM433167 (Barrett et al., 2013; Roadmap Epigenomics Consortium et al., 2015). The SNP array data for the H1 cell line data is available at NCBI GEO under accession GPL18952 (Roadmap Epigenomics Consortium et al., 2015).

The `plyranges` package is open source under an Artistic 2.0 license (Lee, Lawrence, and Cook, 2018). The software can be obtained via the Bioconductor project website `https://bioconductor.org` or accessed via Github `https://github.com/sa-lee/plyranges`.

Acknowledgements

We would like to thank Dr Matthew Ritchie at the Walter and Eliza Hall Institute and Dr Paul Harrison for their feedback on earlier drafts of this work. We would also like to thank Lori Shepherd and Hèrve Pages for the code review they performed and users who have submitted feedback and pull requests.

Chapter 3

Fluent genomics with *plyranges* and *tximeta*

We construct a simple workflow for fluent genomics data analysis using the R/Bioconductor ecosystem. This involves three core steps: import the data into an appropriate abstraction, model the data with respect to the biological questions of interest, and integrate the results with respect to their underlying genomic coordinates. Here we show how to implement these steps to integrate published RNA-seq and ATAC-seq experiments on macrophage cell lines. Using tximeta, we import RNA-seq transcript quantifications into an analysis-ready data structure, called the SummarizedExperiment, that contains the ranges of the reference transcripts and metadata on their provenance. Using SummarizedExperiments to represent the ATAC-seq and RNA-seq data, we model differentially accessible (DA) chromatin peaks and differentially expressed (DE) genes with existing Bioconductor packages. Using plyranges we then integrate the results to see if there is an enrichment of DA peaks near DE genes by finding overlaps and aggregating over log-fold change thresholds. The combination of these packages and their integration with the Bioconductor ecosystem provide a coherent framework for analysts to iteratively and reproducibly explore their biological data.

3.1 Introduction

In this workflow, we examine a subset of the RNA-seq and ATAC-seq data from Alasoo et al. (2018), a study that involved treatment of macrophage cell lines from a number of human donors with interferon gamma (IFNg), *Salmonella* infection, or both treatments combined. Alasoo et al. (2018) examined gene expression and chromatin accessibility in a subset of 86 successfully differentiated induced pluripotent stem cells (iPSC) lines, and compared baseline and response with respect to chromatin accessibility and gene expression at specific quantitative trait loci (QTL). The authors found that many of the stimulus-specific expression QTL were already detectable as chromatin QTL in naive cells, and further hypothesize about the nature and role of transcription factors implicated in the response to stimulus.

We will perform a much simpler analysis than the one found in Alasoo et al. (2018), using their publicly available RNA-seq and ATAC-seq data (ignoring the genotypes). We will examine the effect of IFNg stimulation on gene expression and chromatin accessibility, and look to see if there is an enrichment of differentially accessible (DA) ATAC-seq peaks in the vicinity of differentially expressed (DE) genes. This is plausible, as the transcriptomic response to IFNg stimulation may be mediated through binding of regulatory proteins to accessible regions, and this binding may increase the accessibility of those regions such that it can be detected by ATAC-seq.

Throughout the workflow (Figure 3.1), we will use existing Bioconductor infrastructure to understand these datasets. In particular, we will emphasize the use of the Bioconductor packages `plyranges` and `tximeta`. The `plyranges` package fluently transforms data tied to genomic ranges using operations like shifting, window construction, overlap detection, etc. It is described by Lee, Cook, and Lawrence (2019) and leverages underlying core Bioconductor infrastructure (Lawrence et al., 2013b; Huber et al., 2015b) and the `tidyverse` design principles Wickham et al. (2019).

The `tximeta` package described by Love et al. (2019) is used to read RNA-seq quantification data into R/Bioconductor, such that the transcript ranges and their provenance

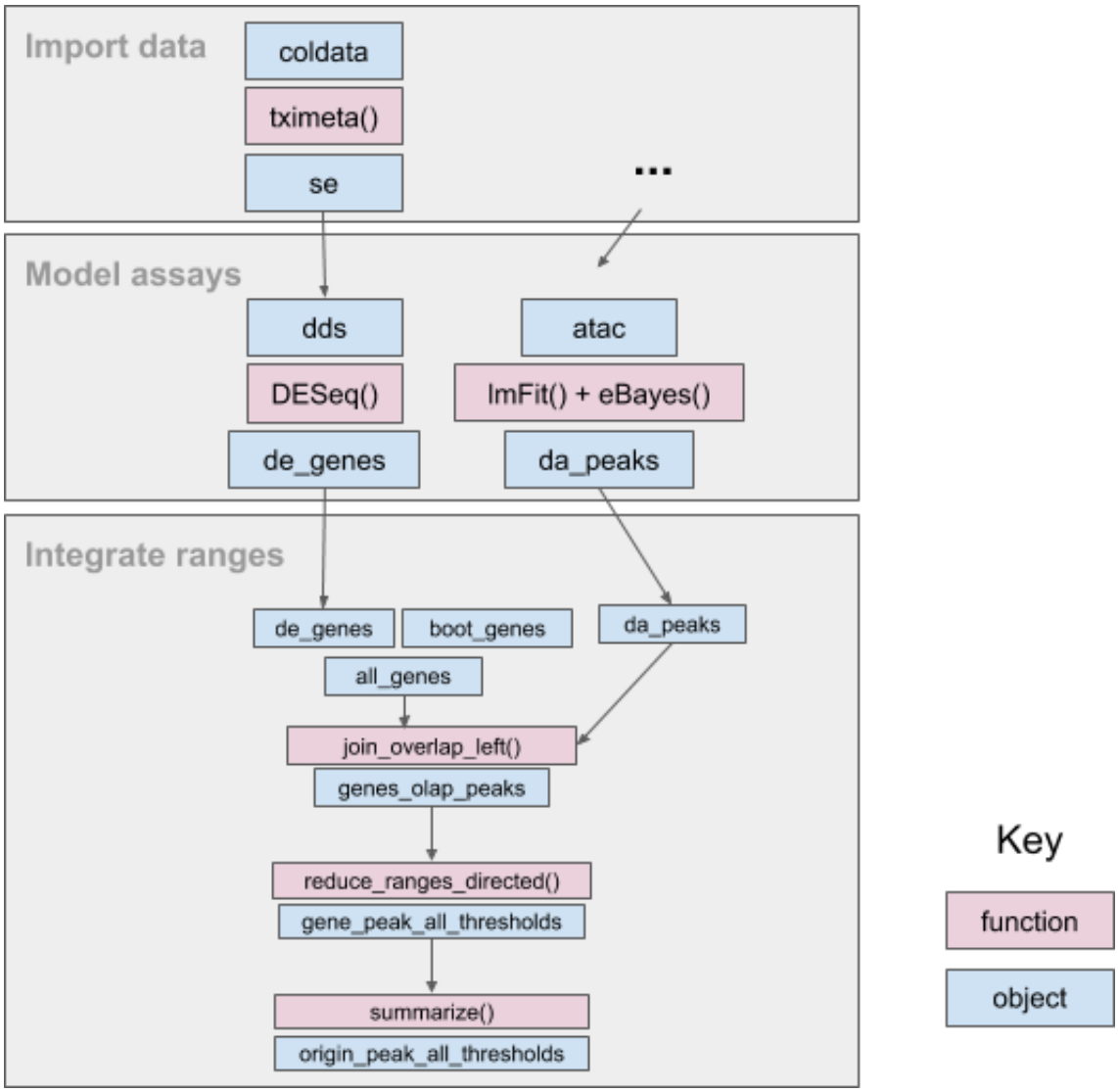


Figure 3.1: *An overview of the fluent genomics workflow. First, we* import *data as a* SummarizedExperiment *object, which enables interoperability with downstream analysis packages. Then we* model *our assay data, using the existing Bioconductor packages* `DESeq2` *and* `limma`*. We take the results of our models for each assay with respect to their genomic coordinates, and* integrate *them. First, we compute the overlap between the results of each assay, then aggregate over the combined genomic regions, and finally summarize to compare enrichment for differentially expressed genes to non differentially expressed genes. The final output can be used for downstream visualization or further transformation.*

are automatically attached to the object containing expression values and differential expression results.

3.1.1 Experimental Data

The data used in this workflow is available from two packages: the `macrophage` Bioconductor ExperimentData package and from the workflow package `fluentGenomics`.

The `macrophage` package contains RNA-seq quantification from 24 RNA-seq samples, a subset of the RNA-seq samples generated and analyzed by Alasoo et al. (2018). The paired-end reads were quantified using *Salmon* (Patro et al., 2017), using the Gencode 29 human reference transcripts (Frankish, GENCODE-consoritum, and Flicek, 2018). For more details on quantification, and the exact code used, consult the vignette of the macrophage package. The package also contains the `Snakemake` file that was used to distribute the *Salmon* quantification jobs on a cluster (Köster and Rahmann, 2012).

The `fluentGenomics` package contains functionality to download and generate a cached *SummarizedExperiment* object from the normalized ATAC-seq data provided by Alasoo and Gaffney (2017). This object contains all 145 ATAC-seq samples across all experimental conditions as analyzed by Alasoo et al. (2018). The data can be also be downloaded directly from the Zenodo deposition.

The following code loads the path to the cached data file, or if it is not present, will create the cache and generate a *SummarizedExperiment* using the the `BiocFileCache` package (Shepherd and Morgan, 2019).

```
library(fluentGenomics)
path_to_se <- cache_atac_se()
```

We can then read the cached file and assign it to an object called `atac`.

```
atac <- readRDS(path_to_se)
```

A precise description of how we obtained this *SummarizedExperiment* object can be found in section 3.2.2.

3.2 Import Data as a *SummarizedExperiment*

3.2.1 Using `tximeta` to import RNA-seq quantification data

First, we specify a directory `dir`, where the quantification files are stored. You could simply specify this directory with:

```
dir <- "/path/to/quant/files"
```

where the path is relative to your current R session. However, in this case we have distributed the files in the `macrophage` package. The relevant directory and associated files can be located using `system.file`.

```
dir <- system.file("extdata", package="macrophage")
```

Information about the experiment is contained in the `coldata.csv` file. We leverage the `dplyr` and `readr` packages (as part of the `tidyverse`) to read this file into R (Wickham et al., 2019). We will see later that `plyranges` extends these packages to accommodate genomic ranges.

```
library(readr)
library(dplyr)
colfile <- file.path(dir, "coldata.csv")
coldata <- read_csv(colfile) %>%
  dplyr::select(
    names,
    id = sample_id,
    line = line_id,
    condition = condition_name
  ) %>%
  dplyr::mutate(
    files = file.path(dir, "quants", names, "quant.sf.gz"),
    line = factor(line),
```

```
    condition = relevel(factor(condition), "naive")
  )
coldata
```

```
#> # A tibble: 24 x 5
#>    names      id     line  condition  files
#>    <chr>      <chr>  <fct> <fct>      <chr>
#>  1 SAMEA1038~ diku_A diku~ naive      /Library/Frameworks/R.framework/Versions/~
#>  2 SAMEA1038~ diku_B diku~ IFNg       /Library/Frameworks/R.framework/Versions/~
#>  3 SAMEA1038~ diku_C diku~ SL1344     /Library/Frameworks/R.framework/Versions/~
#>  4 SAMEA1038~ diku_D diku~ IFNg_SL13~ /Library/Frameworks/R.framework/Versions/~
#>  5 SAMEA1038~ eiwy_A eiwy~ naive      /Library/Frameworks/R.framework/Versions/~
#>  6 SAMEA1038~ eiwy_B eiwy~ IFNg       /Library/Frameworks/R.framework/Versions/~
#>  7 SAMEA1038~ eiwy_C eiwy~ SL1344     /Library/Frameworks/R.framework/Versions/~
#>  8 SAMEA1038~ eiwy_D eiwy~ IFNg_SL13~ /Library/Frameworks/R.framework/Versions/~
#>  9 SAMEA1038~ fikt_A fikt~ naive      /Library/Frameworks/R.framework/Versions/~
#> 10 SAMEA1038~ fikt_B fikt~ IFNg       /Library/Frameworks/R.framework/Versions/~
#> # ... with 14 more rows
```

After we have read the `coldata.csv` file, we select relevant columns from this table, create a new column called `files`, and transform the existing `line` and `condition` columns into factors. In the case of `condition`, we specify the "naive" cell line as the reference level. The `files` column points to the quantifications for each observation – these files have been gzipped, but would typically not have the 'gz' ending if used from *Salmon* directly. One other thing to note is the use of the pipe operator,`%>%`, which can be read as "then", i.e. first read the data, *then* select columns, *then* mutate them.

Now we have a table summarizing the experimental design and the locations of the quantifications. The following lines of code do a lot of work for the analyst: importing the RNA-seq quantification (dropping *inferential replicates* in this case), locating the relevant reference transcriptome, attaching the transcript ranges to the data, and fetching genome information. Inferential replicates are especially useful for performing transcript-level

analysis, but here we will use a point estimate for the per-gene counts and perform gene-level analysis.

The result is a *SummarizedExperiment* object.

```
suppressPackageStartupMessages(library(SummarizedExperiment))
library(tximeta)
se <- tximeta(coldata, dropInfReps=TRUE)
se
```

```
#> class: RangedSummarizedExperiment
#> dim: 205870 24
#> metadata(6): tximetaInfo quantInfo ... txomeInfo txdbInfo
#> assays(3): counts abundance length
#> rownames(205870): ENST00000456328.2 ENST00000450305.2 ...
#>   ENST00000387460.2 ENST00000387461.2
#> rowData names(3): tx_id gene_id tx_name
#> colnames(24): SAMEA103885102 SAMEA103885347 ... SAMEA103885308
#>   SAMEA103884949
#> colData names(4): names id line condition
```

On a machine with a working internet connection, the above command works without any extra steps, as the `tximeta` function obtains any necessary metadata via FTP, unless it is already cached locally. The `tximeta` package can also be used without an internet connection, in this case the linked transcriptome can be created directly from a *Salmon* index and gtf.

```
makeLinkedTxome(
  indexDir=file.path(dir, "gencode.v29_salmon_0.12.0"),
  source="Gencode",
  organism="Homo sapiens",
  release="29",
  genome="GRCh38",
```

```
  fasta="gencode.v29.transcripts.fa.gz", # ftp link to fasta file
  gtf=file.path(dir, "gencode.v29.annotation.gtf.gz"), # local version
  write=FALSE
)
```

Because `tximeta` knows the correct reference transcriptome, we can ask `tximeta` to summarize the transcript-level data to the gene level using the methods of Soneson, Love, and Robinson (2015).

```
gse <- summarizeToGene(se)
```

One final note is that the `start` of positive strand genes and the `end` of negative strand genes is now dictated by the genomic extent of the isoforms of the gene (so the `start` and `end` of the reduced *GRanges*). Another alternative would be to either operate on transcript abundance, and perform differential analysis on transcript (and so avoid defining the TSS of a set of isoforms), or to use gene-level summarized expression but to pick the most representative TSS based on isoform expression.

3.2.2 Importing ATAC-seq data as a *SummarizedExperiment* object

The *SummarizedExperiment* object containing ATAC-seq peaks can be created from the following tab-delimited files from Alasoo and Gaffney (2017):

- The sample metadata: `ATAC_sample_metadata.txt.gz` (<1M)
- The matrix of normalized read counts: `ATAC_cqn_matrix.txt.gz` (109M)
- The annotated peaks: `ATAC_peak_metadata.txt.gz` (5.6M)

To begin, we read in the sample metadata, following similar steps to those we used to generate the `coldata` table for the RNA-seq experiment:

```
atac_coldata <- read_tsv("ATAC_sample_metadata.txt.gz") %>%
  select(
    sample_id,
```

```
    donor,
    condition = condition_name
  ) %>%
  mutate(condition = relevel(factor(condition), "naive"))
```

The ATAC-seq counts have already been normalized with *cqn* (Hansen, Irizarry, and Wu, 2012) and log2 transformed. Loading the *cqn*-normalized matrix of log2 transformed read counts takes ~30 seconds and loads an object of ~370 Mb. We set the column names so that the first column contains the rownames of the matrix, and the remaining columns are the sample identities from the `atac_coldata` object.

```
atac_mat <- read_tsv(
  "ATAC_cqn_matrix.txt.gz",
  skip = 1,
  col_names = c("rownames", atac_coldata[["sample_id"]])
 )
rownames <- atac_mat[["rownames"]]
atac_mat <- as.matrix(atac_mat[,-1])
rownames(atac_mat) <- rownames
```

We read in the peak metadata (locations in the genome), and convert it to a *GRanges* object. The `as_granges()` function automatically converts the *data.frame* into a *GRanges* object. From that result, we extract the peak_id column and set the genome information to the build "GRCh38". We know this from the Zenodo entry.

```
library(plyranges)
peaks_df <- read_tsv(
  "ATAC_peak_metadata.txt.gz",
  col_types = c("cidciicdc")
  )

peaks_gr <- peaks_df %>%
```

```
  as_granges(seqnames = chr) %>%
  select(peak_id=gene_id) %>%
  set_genome_info(genome = "GRCh38")
```

Finally, we construct a *SummarizedExperiment* object. We place the matrix into the assays slot as a named list, the annotated peaks into the row-wise ranges slot, and the sample metadata into the column-wise data slot:

```
atac <- SummarizedExperiment(
  assays = list(cqndata=atac_mat),
  rowRanges = peaks_gr,
  colData = atac_coldata
)
```

3.3 Model assays

3.3.1 RNA-seq differential gene expression analysis

We can easily run a differential expression analysis with `DESeq2` using the following code chunks (Love, Huber, and Anders, 2014). The design formula indicates that we want to control for the donor baselines (`line`) and test for differences in gene expression on the condition. For a more comprehensive discussion of DE workflows in Bioconductor see Love et al. (2016) and Law et al. (2018).

```
library(DESeq2)
dds <- DESeqDataSet(gse, ~line + condition)
# filter out lowly expressed genes
# at least 10 counts in at least 6 samples
keep <- rowSums(counts(dds) >= 10) >= 6
dds <- dds[keep,]
```

The model is fit with the following line of code:

```
dds <- DESeq(dds)
```

Below we set the contrast on the condition variable, indicating we are estimating the $\log_2$ fold change (LFC) of IFNg stimulated cell lines against naive cell lines. We are interested in LFC greater than 1 at a nominal false discovery rate (FDR) of 1%.

```
res <- results(dds,
               contrast=c("condition","IFNg","naive"),
               lfcThreshold=1, alpha=0.01)
```

To see the results of the expression analysis, we can generate a summary table and an MA plot:

```
summary(res)
```

```
#>
#> out of 17806 with nonzero total read count
#> adjusted p-value < 0.01
#> LFC > 1.00 (up)    : 502, 2.8%
#> LFC < -1.00 (down) : 247, 1.4%
#> outliers [1]       : 0, 0%
#> low counts [2]     : 0, 0%
#> (mean count < 3)
#> [1] see 'cooksCutoff' argument of ?results
#> [2] see 'independentFiltering' argument of ?results
```

```
DESeq2::plotMA(res, ylim=c(-10,10))
```

We now output the results as a *GRanges* object, and due to the conventions of `plyranges`, we construct a new column called `gene_id` from the row names of the results. Each row now contains the genomic region (`seqnames`, `start`, `end`, `strand`) along with corresponding metadata columns (the `gene_id` and the results of the test). Note that `tximeta` has

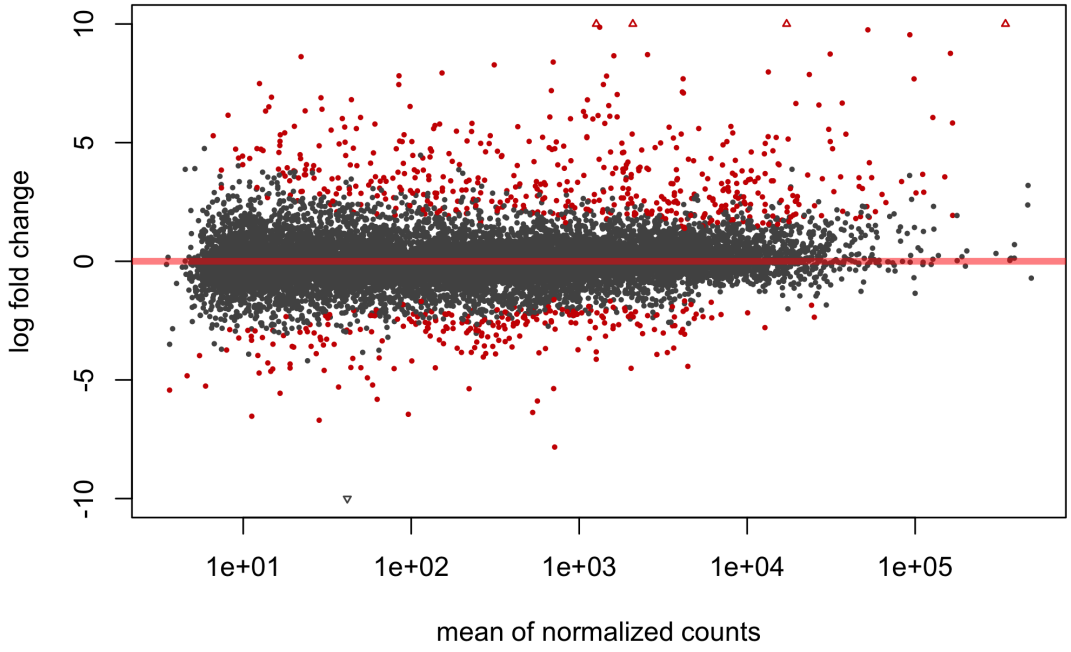


Figure 3.2: *Visualization of `DESeq2` results as an "MA plot". Genes that have an adjusted* p-value *below 0.01 are colored red.*

correctly identified the reference genome as "hg38", and this has also been added to the *GRanges* along the results columns. This kind of book-keeping is vital once overlap operations are performed to ensure that `plyranges` is not comparing across incompatible genomes.

```
suppressPackageStartupMessages(library(plyranges))
de_genes <- results(dds,
                    contrast=c("condition","IFNg","naive"),
                    lfcThreshold=1,
                    format="GRanges") %>%
  names_to_column("gene_id")
de_genes
```

```
#> GRanges object with 17806 ranges and 7 metadata columns:
#>           seqnames              ranges strand |           gene_id
```

```
#>              <Rle>          <IRanges>  <Rle> |       <character>
#>       [1]     chrX 100627109-100639991      - | ENSG00000000003.14
#>       [2]    chr20   50934867-50958555      - | ENSG00000000419.12
#>       [3]     chr1 169849631-169894267      - | ENSG00000000457.13
#>       [4]     chr1 169662007-169854080      + | ENSG00000000460.16
#>       [5]     chr1   27612064-27635277      - | ENSG00000000938.12
#>       ...      ...                ...    ... .               ...
#>   [17802]    chr10   84167228-84172093      - |  ENSG00000285972.1
#>   [17803]     chr6   63572012-63583587      + |  ENSG00000285976.1
#>   [17804]    chr16   57177349-57181390      + |  ENSG00000285979.1
#>   [17805]     chr8 103398658-103501895      - |  ENSG00000285982.1
#>   [17806]    chr10   12563151-12567351      + |  ENSG00000285994.1
#>                  baseMean     log2FoldChange             lfcSE
#>                 <numeric>          <numeric>         <numeric>
#>       [1] 171.570646163445 -0.282245015065582  0.300571026277417
#>       [2] 967.751278980391 0.0391222756936352 0.0859707605047955
#>       [3] 682.432885098654    1.2846178585311  0.196906721741941
#>       [4] 262.963397841117  -1.47187616421189  0.218691645887265
#>       [5] 2660.10225731917  0.675478091290521  0.236053041372838
#>       ...              ...                ...               ...
#>   [17802] 10.0474624496157  0.548451844773876  0.444318686394084
#>   [17803] 4586.34616821518 -0.033929582570062  0.188004977365846
#>   [17804] 14.2965310090402  0.312347650582085  0.522699844356108
#>   [17805] 27.7629588245413  0.994518742790125   1.58237312176743
#>   [17806] 6.60408582708505   0.25399752352481    0.5957511892896
#>                       stat             pvalue              padj
#>                  <numeric>          <numeric>         <numeric>
#>       [1]                0                  1                 1
#>       [2]                0                  1                 1
#>       [3]  1.44544511235177  0.148332899695748                 1
#>       [4] -2.15772377722715 0.0309493141635637 0.409727500369082
```

```
#>       [5]                 0                 1                 1
#>       ...               ...               ...               ...
#>   [17802]                 0                 1                 1
#>   [17803]                 0                 1                 1
#>   [17804]                 0                 1                 1
#>   [17805]                 0                 1                 1
#>   [17806]                 0                 1                 1
#>   -------
#>   seqinfo: 25 sequences (1 circular) from hg38 genome
```

From this, we can restrict the results to those that meet our FDR threshold and select (and rename) the metadata columns we're interested in:

```
de_genes <- de_genes %>%
  filter(padj < 0.01) %>%
  dplyr::select(
    gene_id,
    de_log2FC = log2FoldChange,
    de_padj = padj
  )
```

We now wish to extract genes for which there is evidence that the LFC is *not* large. We perform this test by specifying an LFC threshold and an alternative hypothesis (`altHypothesis`) that the LFC is less than the threshold in absolute value. To visualize the result of this test, you can run `results` without `format="GRanges"`, and pass this object to `plotMA` as before. We label these genes as `other_genes` and later as "non-DE genes", for comparison with our `de_genes` set.

```
other_genes <- results(dds,
                       contrast=c("condition","IFNg","naive"),
                       lfcThreshold=1,
                       altHypothesis="lessAbs",
```

```
                        format="GRanges") %>%
  filter(padj < 0.01) %>%
  names_to_column("gene_id") %>%
  dplyr::select(
    gene_id,
    de_log2FC = log2FoldChange,
    de_padj = padj
  )
```

3.3.2 ATAC-seq peak differential abundance analysis

The following section describes the process we have used for generating a *GRanges* object of differential peaks from the ATAC-seq data in Alasoo et al. (2018).

The code chunks for the remainder of this section are not run.

For assessing differential accessibility, we run `limma` (Smyth, 2004), and generate the a summary of LFCs and adjusted p-values for the peaks:

```
library(limma)
design <- model.matrix(~donor + condition, colData(atac))
fit <- lmFit(assay(atac), design)
fit <- eBayes(fit)
idx <- which(colnames(fit$coefficients) == "conditionIFNg")
tt <- topTable(fit, coef=idx, sort.by="none", n=nrow(atac))
```

We now take the `rowRanges` of the *SummarizedExperiment* and attach the LFCs and adjusted p-values from `limma`, so that we can consider the overlap with differential expression. Note that we set the genome build to "hg38" and restyle the chromosome information to use the "UCSC" style (e.g. "chr1", "chr2", etc.). Again, we know the genome build from the Zenodo entry for the ATAC-seq data.

```
atac_peaks <- rowRanges(atac) %>%
  remove_names() %>%
  mutate(
    da_log2FC = tt$logFC,
    da_padj = tt$adj.P.Val
  ) %>%
  set_genome_info(genome = "hg38")

seqlevelsStyle(atac_peaks) <- "UCSC"
```

The final *GRanges* object containing the DA peaks is included in the workflow package and can be loaded as follows:

```
library(fluentGenomics)
peaks
```

```
#> GRanges object with 296220 ranges and 3 metadata columns:
#>             seqnames              ranges strand |          peak_id
#>                <Rle>           <IRanges>  <Rle> |      <character>
#>        [1]     chr1          9979-10668      * |      ATAC_peak_1
#>        [2]     chr1         10939-11473      * |      ATAC_peak_2
#>        [3]     chr1         15505-15729      * |      ATAC_peak_3
#>        [4]     chr1         21148-21481      * |      ATAC_peak_4
#>        [5]     chr1         21864-22067      * |      ATAC_peak_5
#>        ...      ...                 ...    ... .              ...
#>   [296216]     chrX 155896572-155896835      * | ATAC_peak_296216
#>   [296217]     chrX 155958507-155958646      * | ATAC_peak_296217
#>   [296218]     chrX 156016760-156016975      * | ATAC_peak_296218
#>   [296219]     chrX 156028551-156029422      * | ATAC_peak_296219
#>   [296220]     chrX 156030135-156030785      * | ATAC_peak_296220
#>                     da_log2FC              da_padj
```

```
#>                     <numeric>           <numeric>
#>        [1]  0.266185396736073 9.10672732956434e-05
#>        [2]   0.32217712436691 2.03434717570469e-05
#>        [3] -0.574159538548115 3.41707743345703e-08
#>        [4]  -1.14706617895329 8.22298606986521e-26
#>        [5] -0.896143162633654 4.79452571676397e-11
#>        ...                ...                 ...
#>   [296216] -0.834628897017445  1.3354605397165e-11
#>   [296217] -0.147537281935847    0.313014754316915
#>   [296218] -0.609732301631964 3.62338775135558e-09
#>   [296219] -0.347678474957794 6.94823191242968e-06
#>   [296220]  0.492442459200901 7.07663984067763e-13
#>   -------
#>   seqinfo: 23 sequences from hg38 genome; no seqlengths
```

3.4 Integrate ranges

3.4.1 Finding overlaps with `plyranges`

We have already used `plyranges` a number of times above, to `filter`, `mutate`, and `select` on *GRanges* objects, as well as ensuring the correct genome annotation and style has been used. The `plyranges` package provides a grammar for performing transformations of genomic data (Lee, Cook, and Lawrence, 2019). Computations resulting from compositions of `plyranges` "verbs" are performed using underlying, highly optimized range operations in the `GenomicRanges` package (Lawrence et al., 2013b).

For the overlap analysis, we filter the annotated peaks to have a nominal FDR bound of 1%.

```
da_peaks <- peaks %>%
  filter(da_padj < 0.01)
```

We now have *GRanges* objects that contain DE genes, genes without strong signal of DE, and DA peaks. We are ready to answer the question: is there an enrichment of DA ATAC-seq peaks in the vicinity of DE genes compared to genes without sufficient DE signal?

3.4.2 Down sampling non-differentially expressed genes

As `plyranges` is built on top of *dplyr*, it implements methods for many of its verbs for *GRanges* objects. Here we can use `slice` to randomly sample the rows of the `other_genes`. The `sample.int` function will generate random samples of size equal to the number of DE-genes from the number of rows in `other_genes`:

```
size <- length(de_genes)
slice(other_genes, sample.int(plyranges::n(), size))
```

```
#> GRanges object with 749 ranges and 3 metadata columns:
#>         seqnames              ranges strand |            gene_id
#>            <Rle>           <IRanges>  <Rle> |        <character>
#>     [1]     chr5 168486445-168519299      + |  ENSG00000113643.8
#>     [2]     chr2   27356246-27367622      + |  ENSG00000234072.1
#>     [3]     chr7 144194858-144280547      + |  ENSG00000244198.6
#>     [4]     chr6   11183298-11382348      - | ENSG00000111859.16
#>     [5]    chr19   51685363-51693456      - |  ENSG00000269959.1
#>     ...      ...                 ...    ... .                ...
#>   [745]    chr14   69726900-69772005      + | ENSG00000100650.15
#>   [746]     chr9         14521-73865      - | ENSG00000181404.16
#>   [747]    chr19     1259384-1274880      + | ENSG00000099622.13
#>   [748]    chr14   36298558-36320676      - | ENSG00000151332.18
#>   [749]    chr11   14443440-14500027      - | ENSG00000129083.12
#>                  de_log2FC              de_padj
#>                  <numeric>            <numeric>
#>     [1]   0.186516196419689 4.71325191585823e-18
```

```
#>     [2] -0.0274372318428813 2.95938040043472e-13
#>     [3] -0.0483757376755817  0.00341552605245743
#>     [4] -0.0596189088466196  0.00285906305775345
#>     [5]  -0.110077465658236  0.00262070004529918
#>     ...                 ...                  ...
#>   [745]   0.283740153067817 1.02083548365204e-07
#>   [746] -0.0144915263455031 7.00523592219126e-16
#>   [747] -0.0717794786944121 9.54050291182279e-36
#>   [748] -0.0246183972472181 4.87702908650434e-10
#>   [749]   0.316257709291474 5.56084198449402e-08
#>   -------
#>   seqinfo: 25 sequences (1 circular) from hg38 genome
```

We can repeat this many times to create many samples via `replicate`. By replicating the sub-sampling multiple times, we minimize the variance on the enrichment statistics induced by the sampling process.

```
# set a seed for the results
set.seed(2019-08-02)
boot_genes <- replicate(10,
                        slice(other_genes, sample.int(plyranges::n(), size)),
                        simplify = FALSE)
```

This creates a list of *GRanges* objects as a list, and we can bind these together using the `bind_ranges` function. This function creates a new column called "resample" on the result that identifies each of the input *GRanges* objects:

```
boot_genes <- bind_ranges(boot_genes, .id = "resample")
```

Similarly, we can then combine the `boot_genes` *GRanges*, with the DE *GRanges* object. As the resample column was not present on the DE *GRanges* object, this is given a missing value which we recode to a 0 using `mutate()`

```
all_genes <- bind_ranges(
  de=de_genes,
  not_de = boot_genes,
  .id="origin"
) %>%
  mutate(
    origin = factor(origin, c("not_de", "de")),
    resample = ifelse(is.na(resample), 0L, as.integer(resample))
  )
all_genes
```

```
#> GRanges object with 8239 ranges and 5 metadata columns:
#>          seqnames              ranges strand |            gene_id
#>             <Rle>           <IRanges>  <Rle> |        <character>
#>      [1]     chr1 196651878-196747504      + | ENSG00000000971.15
#>      [2]     chr6   46129993-46146699      + |  ENSG00000001561.6
#>      [3]     chr4   17577192-17607972      + | ENSG00000002549.12
#>      [4]     chr7 150800403-150805120      + |  ENSG00000002933.8
#>      [5]     chr4   15778275-15853230      + | ENSG00000004468.12
#>      ...      ...                 ...    ... .                ...
#>   [8235]    chr17   43527844-43579620      - | ENSG00000175832.12
#>   [8236]    chr17   18260534-18266552      + | ENSG00000177427.12
#>   [8237]    chr20   63895182-63936031      + | ENSG00000101152.10
#>   [8238]     chr1   39081316-39487177      + | ENSG00000127603.25
#>   [8239]     chr8   41577187-41625001      + | ENSG00000158669.11
#>                  de_log2FC              de_padj  resample   origin
#>                  <numeric>            <numeric> <integer> <factor>
#>      [1]   4.98711071930695 1.37057050625117e-13         0       de
#>      [2]   1.92721595378787  3.1747750217733e-05         0       de
#>      [3]   2.93372501059128  2.0131038573066e-11         0       de
#>      [4]   3.16721751137972 1.07359906028984e-08         0       de
```

```
#>      [5]   5.40894352968188 4.82904694023763e-18         0       de
#>      ...                ...                  ...       ...      ...
#>   [8235] -0.240918426099239  0.00991611085813261        10   not_de
#>   [8236] -0.166059030395757  9.1205141062356e-05        10   not_de
#>   [8237]  0.250538999517482 1.74084544559733e-09        10   not_de
#>   [8238] -0.385053503003028  0.00265539384929076        10   not_de
#>   [8239]  0.155922038318879  2.9637514745875e-17        10   not_de
#>   -------
#>   seqinfo: 25 sequences (1 circular) from hg38 genome
```

3.4.3 Expanding genomic coordinates around the transcription start site

Now we would like to modify our gene ranges so they contain the 10 kilobases on either side of their transcription start site (TSS). There are many ways one could do this, but we prefer an approach via the anchoring methods in `plyranges`. Because there is a mutual dependence between the start, end, width, and strand of a *GRanges* object, we define anchors to fix one of `start` and `end`, while modifying the `width`. As an example, to extract just the TSS, we can anchor by the 5′ end of the range and modify the width of the range to equal 1.

```
all_genes <- all_genes %>%
  anchor_5p() %>%
  mutate(width = 1)
```

Anchoring by the 5′ end of a range will fix the `end` of negatively stranded ranges, and fix the `start` of positively stranded ranges.

We can then repeat the same pattern but this time using `anchor_center()` to tell `plyranges` that we are making the TSS the midpoint of a range that has total width of 20kb, or 10kb both upstream and downstream of the TSS.

```
all_genes <- all_genes %>%
  anchor_center() %>%
  mutate(width=2*1e4)
```

3.4.4 Use overlap joins to find relative enrichment

We are now ready to compute overlaps between RNA-seq genes (our DE set and bootstrap sets) and the ATAC-seq peaks. In `plyranges`, overlaps are defined as joins between two *GRanges* objects: a *left* and a *right GRanges* object. In an overlap join, a match is any range on the *left GRanges* that is overlapped by the *right GRanges*. One powerful aspect of the overlap joins is that the result maintains all (metadata) columns from each of the *left* and *right* ranges which makes downstream summaries easy to compute.

To combine the DE genes with the DA peaks, we perform a left overlap join. This returns to us the `all_genes` ranges (potentially with duplication), but with the metadata columns from those overlapping DA peaks. For any gene that has no overlaps, the DA peak columns will have `NA`'s.

```
#> GRanges object with 27766 ranges and 8 metadata columns:
#>           seqnames              ranges strand |            gene_id
#>              <Rle>           <IRanges>  <Rle> |        <character>
#>       [1]     chr1 196641878-196661877      + | ENSG00000000971.15
#>       [2]     chr6   46119993-46139992      + |  ENSG00000001561.6
#>       [3]     chr4   17567192-17587191      + | ENSG00000002549.12
#>       [4]     chr4   17567192-17587191      + | ENSG00000002549.12
#>       [5]     chr4   17567192-17587191      + | ENSG00000002549.12
#>       ...      ...                 ...    ... .                ...
#>   [27762]     chr1   39071316-39091315      + | ENSG00000127603.25
#>   [27763]     chr1   39071316-39091315      + | ENSG00000127603.25
#>   [27764]     chr8   41567187-41587186      + | ENSG00000158669.11
#>   [27765]     chr8   41567187-41587186      + | ENSG00000158669.11
#>   [27766]     chr8   41567187-41587186      + | ENSG00000158669.11
```

```
#>                   de_log2FC              de_padj  resample   origin
#>                   <numeric>            <numeric> <integer> <factor>
#>       [1]   4.98711071930695 1.37057050625117e-13         0       de
#>       [2]   1.92721595378787  3.1747750217733e-05         0       de
#>       [3]   2.93372501059128  2.0131038573066e-11         0       de
#>       [4]   2.93372501059128  2.0131038573066e-11         0       de
#>       [5]   2.93372501059128  2.0131038573066e-11         0       de
#>       ...                ...                  ...       ...      ...
#>   [27762] -0.385053503003028  0.00265539384929076        10   not_de
#>   [27763] -0.385053503003028  0.00265539384929076        10   not_de
#>   [27764]  0.155922038318879  2.9637514745875e-17        10   not_de
#>   [27765]  0.155922038318879  2.9637514745875e-17        10   not_de
#>   [27766]  0.155922038318879  2.9637514745875e-17        10   not_de
#>                   peak_id          da_log2FC              da_padj
#>               <character>          <numeric>            <numeric>
#>       [1]  ATAC_peak_21236 -0.546582189082724 0.000115273676444232
#>       [2] ATAC_peak_231183   1.45329684862127  9.7322474682763e-17
#>       [3] ATAC_peak_193578  0.222371496904895 3.00939005719989e-11
#>       [4] ATAC_peak_193579 -0.281615137872819 7.99888515457195e-05
#>       [5] ATAC_peak_193580  0.673705317951604 7.60042918890061e-15
#>       ...              ...                ...                  ...
#>   [27762]   ATAC_peak_5357  -1.05823584693303 3.69051674661467e-16
#>   [27763]   ATAC_peak_5358  -1.31411238041643 6.44280493172654e-26
#>   [27764] ATAC_peak_263396 -0.904080135059089 8.19576651692093e-13
#>   [27765] ATAC_peak_263397  0.364737985368599 2.08834835864614e-08
#>   [27766] ATAC_peak_263399  0.317386691052334 1.20088116314111e-08
#>   -------
#>   seqinfo: 25 sequences (1 circular) from hg38 genome
```

Now we can ask, how many DA peaks are near DE genes relative to "other" non-DE genes? A gene may appear more than once in `genes_olap_peaks`, because multiple peaks

may overlap a single gene, or because we have re-sampled the same gene more than once, or a combination of these two cases.

For each gene (that is the combination of chromosome, the start, end, and strand), and the "origin" (DE vs not-DE) we can compute the distinct number of peaks for each gene and the maximum peak based on LFC. This is achieved via `reduce_ranges_directed`, which allows an aggregation to result in a *GRanges* object via merging neighboring genomic regions. The use of the directed suffix indicates we're maintaining strand information. In this case, we are simply merging ranges (genes) via the groups we mentioned above. We also have to account for the number of resamples we have performed when counting if there are any peaks, to ensure we do not double count the same peak:

We can then filter genes if they have any peaks and compare the peak fold changes between non-DE and DE genes using a boxplot:

```
library(ggplot2)
gene_peak_max_lfc %>%
  filter(peak_count > 0) %>%
  as.data.frame() %>%
  ggplot(aes(origin, peak_max_lfc)) +
  geom_boxplot()
```

In general, the DE genes have larger maximum DA fold changes relative to the non-DE genes.

Next we examine how thresholds on the DA LFC modify the enrichment we observe of DA peaks near DE or non-DE genes. First, we want to know how the number of peaks within DE genes and non-DE genes change as we change threshold values on the peak LFC. As an example, we could compute this by arbitrarily chosen LFC thresholds of 1 or 2 as follows:

```
origin_peak_lfc <- genes_olap_peaks %>%
  group_by(origin) %>%
  summarize(
```

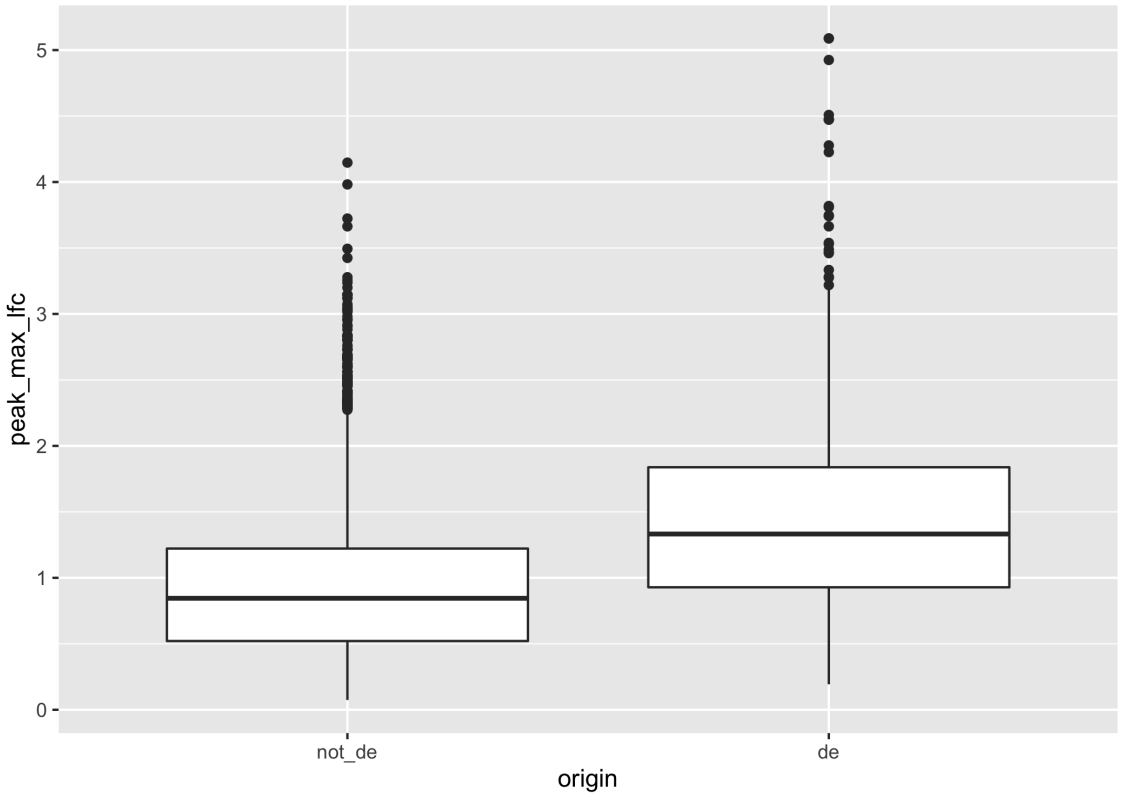


Figure 3.3: *A boxplot of maximum LFCs for DA peaks for DE genes compared to non-DE genes where genes have at least one DA peak.*

```
    peak_count = sum(!is.na(da_padj)) / plyranges::n_distinct(resample),
    lfc1_peak_count =sum(abs(da_log2FC) > 1, na.rm=TRUE)/ plyranges::n_distinct(resa
    lfc2_peak_count = sum(abs(da_log2FC) > 2, na.rm=TRUE)/ plyranges::n_distinct(res
  )
origin_peak_lfc
```

```
#> DataFrame with 2 rows and 4 columns
#>     origin peak_count lfc1_peak_count lfc2_peak_count
#>   <factor>  <numeric>       <numeric>       <numeric>
#> 1   not_de     2391.8           369.5            32.5
#> 2       de       3416            1097             234
```

Here we see that DE genes tend to have more DA peaks near them, and that the number of DA peaks decreases as we increase the DA LFC threshold (as expected). We now show

how to compute the ratio of peak counts from DE compared to non-DE genes, so we can see how this ratio changes for various DA LFC thresholds.

For all variables except for the `origin` column we divide the first row's values by the second row, which will be the enrichment of peaks in DE genes compared to other genes. This requires us to reshape the summary table from long form back to wide form using the *tidyr* package. First we pivot the results of the `peak_count` columns into name-value pairs, then pivot again to place values into the `origin` column. Then we create a new column with the relative enrichment:

```
origin_peak_lfc %>%
  as.data.frame() %>%
  tidyr::pivot_longer(cols = -origin) %>%
  tidyr::pivot_wider(names_from = origin, values_from = value) %>%
  mutate(enrichment = de / not_de)
```

```
#> # A tibble: 3 x 4
#>   name            not_de    de enrichment
#>   <chr>            <dbl> <dbl>      <dbl>
#> 1 peak_count      2392.   3416       1.43
#> 2 lfc1_peak_count  370.   1097       2.97
#> 3 lfc2_peak_count   32.5   234       7.2
```

The above table shows that relative enrichment increases for a larger LFC threshold.

Due to the one-to-many mappings of genes to peaks, it is unknown if we have the same number of DE genes participating or less, as we increase the threshold on the DA LFC. We can examine the number of genes with overlapping DA peaks at various thresholds by grouping and aggregating twice. First, the number of peaks that meet the thresholds are computed within each gene, origin, and resample group. Second, within the origin column, we compute the total number of peaks that meet the DA LFC threshold and the number of genes that have more than zero peaks (again averaging over the number of resamples).

```
genes_olap_peaks %>%
  group_by(gene_id, origin, resample) %>%
  reduce_ranges_directed(
    lfc1 = sum(abs(da_log2FC) > 1, na.rm=TRUE),
    lfc2 = sum(abs(da_log2FC) > 2, na.rm=TRUE)
  ) %>%
  group_by(origin) %>%
  summarize(
    lfc1_gene_count = sum(lfc1 > 0) / plyranges::n_distinct(resample),
    lfc1_peak_count = sum(lfc1) / plyranges::n_distinct(resample),
    lfc2_gene_count = sum(lfc2 > 0) / plyranges::n_distinct(resample),
    lfc2_peak_count = sum(lfc2) / plyranges::n_distinct(resample)
  )
```

```
#> DataFrame with 2 rows and 5 columns
#>     origin lfc1_gene_count lfc1_peak_count lfc2_gene_count lfc2_peak_count
#>   <factor>       <numeric>       <numeric>       <numeric>       <numeric>
#> 1   not_de           271.2           369.5            30.3            32.5
#> 2       de             515            1097             151             234
```

To do this for many thresholds is cumbersome and would create a lot of duplicate code. Instead we create a single function called `count_above_threshold` that accepts a variable and a vector of thresholds, and computes the sum of the absolute value of the variable for each element in the `thresholds` vector.

```
count_if_above_threshold <- function(var, thresholds) {
  lapply(thresholds, function(.) sum(abs(var) > ., na.rm = TRUE))
}
```

The above function will compute the counts for any arbitrary threshold, so we can apply it over possible LFC thresholds of interest. We choose a grid of one hundred thresholds based on the range of absolute LFC values in the `da_peaks` *GRanges* object:

```
thresholds <- da_peaks %>%
  mutate(abs_lfc = abs(da_log2FC)) %>%
  with(
    seq(min(abs_lfc), max(abs_lfc), length.out = 100)
  )
```

The peak counts for each threshold are computed as a new list-column called `value`. First, the *GRanges* object has been grouped by the gene, origin, and the number of resamples columns. Then we aggregate over those columns, so each row will contain the peak counts for all of the thresholds for a gene, origin, and resample. We also maintain another list-column that contains the threshold values.

```
genes_peak_all_thresholds <- genes_olap_peaks %>%
  group_by(gene_id, origin, resample) %>%
  reduce_ranges_directed(
    value = count_if_above_threshold(da_log2FC, thresholds),
    threshold = list(thresholds)
  )
genes_peak_all_thresholds
```

```
#> GRanges object with 8239 ranges and 5 metadata columns:
#>          seqnames              ranges strand |            gene_id   origin
#>             <Rle>           <IRanges>  <Rle> |        <character> <factor>
#>      [1]     chr1 196641878-196661877      + | ENSG00000000971.15       de
#>      [2]     chr6   46119993-46139992      + |  ENSG00000001561.6       de
#>      [3]     chr4   17567192-17587191      + | ENSG00000002549.12       de
#>      [4]     chr7 150790403-150810402      + |  ENSG00000002933.8       de
#>      [5]     chr4   15768275-15788274      + | ENSG00000004468.12       de
#>      ...      ...                 ...    ... .                ...      ...
#>   [8235]    chr17   43569620-43589619      - | ENSG00000175832.12   not_de
#>   [8236]    chr17   18250534-18270533      + | ENSG00000177427.12   not_de
```

```
#>   [8237]    chr20   63885182-63905181      + | ENSG00000101152.10   not_de
#>   [8238]     chr1   39071316-39091315      + | ENSG00000127603.25   not_de
#>   [8239]     chr8   41567187-41587186      + | ENSG00000158669.11   not_de
#>            resample         value
#>           <integer> <IntegerList>
#>      [1]         0     1,1,1,...
#>      [2]         0     1,1,1,...
#>      [3]         0     6,6,6,...
#>      [4]         0     4,4,4,...
#>      [5]         0  11,11,11,...
#>      ...       ...           ...
#>   [8235]        10     1,1,1,...
#>   [8236]        10     3,3,2,...
#>   [8237]        10     5,5,5,...
#>   [8238]        10     3,3,3,...
#>   [8239]        10     3,3,3,...
#>                                                          threshold
#>                                                      <NumericList>
#>      [1] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>      [2] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>      [3] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>      [4] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>      [5] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>      ...                                                       ...
#>   [8235] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>   [8236] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>   [8237] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>   [8238] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>   [8239] 0.0658243106359027,0.118483961449043,0.171143612262182,...
#>   -------
#>   seqinfo: 25 sequences (1 circular) from hg38 genome
```

Now we can expand these list-columns into a long *GRanges* object using the `expand_ranges()` function. This function will unlist the `value` and `threshold` columns and lengthen the resulting *GRanges* object. To compute the peak and gene counts for each threshold, we apply the same summarization as before:

```
origin_peak_all_thresholds <- genes_peak_all_thresholds %>%
  expand_ranges() %>%
  group_by(origin, threshold) %>%
  summarize(
    gene_count = sum(value > 0) / plyranges::n_distinct(resample),
    peak_count = sum(value) / plyranges::n_distinct(resample)
  )
origin_peak_all_thresholds
```

```
#> DataFrame with 200 rows and 4 columns
#>       origin          threshold gene_count peak_count
#>     <factor>          <numeric>  <numeric>  <numeric>
#> 1     not_de 0.0658243106359027        708     2391.4
#> 2     not_de  0.118483961449043      698.8     2320.6
#> 3     not_de  0.171143612262182      686.2     2178.6
#> 4     not_de  0.223803263075322      672.4     1989.4
#> 5     not_de  0.276462913888462      650.4     1785.8
#> ...      ...                ...        ...        ...
#> 196       de   5.06849113788419          2          2
#> 197       de   5.12115078869733          0          0
#> 198       de   5.17381043951047          0          0
#> 199       de   5.22647009032361          0          0
#> 200       de   5.27912974113675          0          0
```

Again we can compute the relative enrichment in LFCs in the same manner as before, by pivoting the results to long form then back to wide form to compute the enrichment. We visualize the peak enrichment changes of DE genes relative to other genes as a line chart:

```
origin_threshold_counts <- origin_peak_all_thresholds %>%
  as.data.frame() %>%
  tidyr::pivot_longer(cols = -c(origin, threshold),
                      names_to = c("type", "var"),
                      names_sep = "_",
                      values_to = "count") %>%
  dplyr::select(-var)


origin_threshold_counts %>%
  filter(type == "peak") %>%
  tidyr::pivot_wider(names_from = origin, values_from = count) %>%
  mutate(enrichment =  de / not_de) %>%
  ggplot(aes(x = threshold, y = enrichment)) +
  geom_line() +
  labs(x = "logFC threshold", y = "Relative Enrichment")
```

We computed the sum of DA peaks near the DE genes, for increasing LFC thresholds on the accessibility change. As we increased the threshold, the number of total peaks went down (likewise the mean number of DA peaks per gene). It is also likely the number of DE genes with a DA peak nearby with such a large change went down. We can investigate this with a plot that summarizes many of the aspects underlying the enrichment plot above.

```
origin_threshold_counts %>%
  ggplot(aes(x = threshold,
             y = count + 1,
             color = origin,
             linetype = type)) +
  geom_line() +
  scale_y_log10()
```

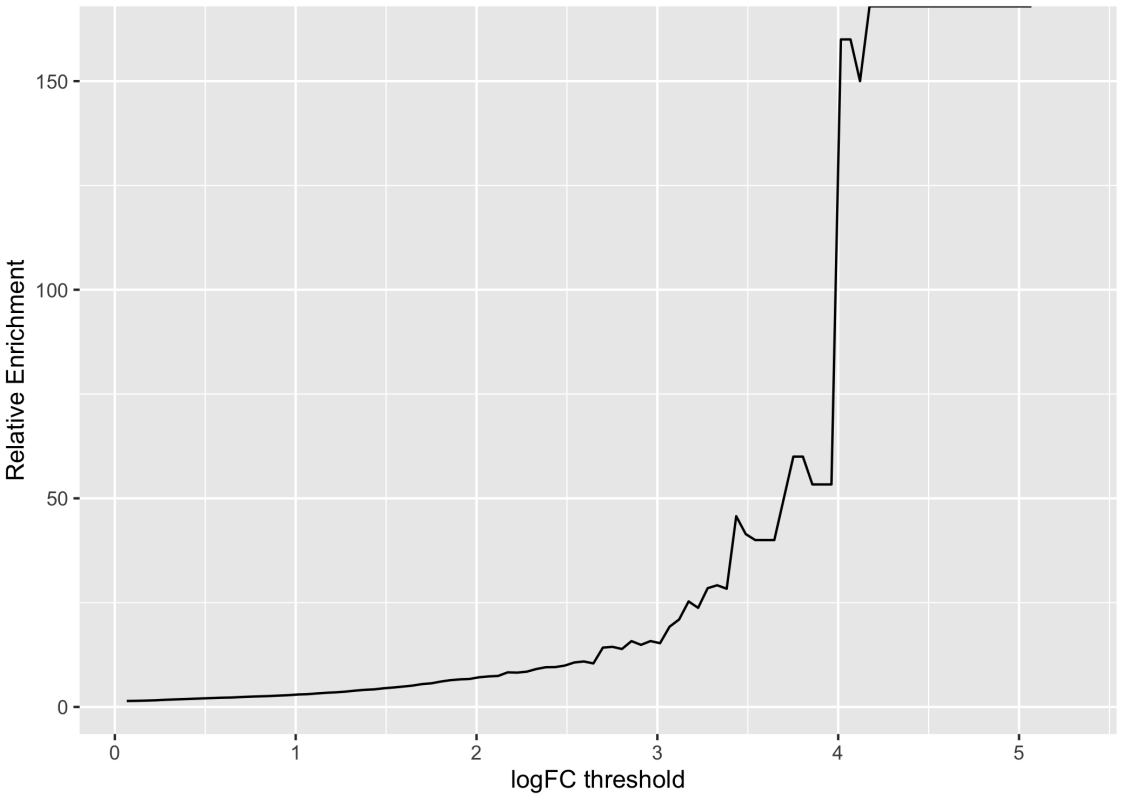


Figure 3.4: *A line chart displaying how relative enrichment of DA peaks change between DE genes compared to non-DE genes as the absolute DA LFC threshold increases.*

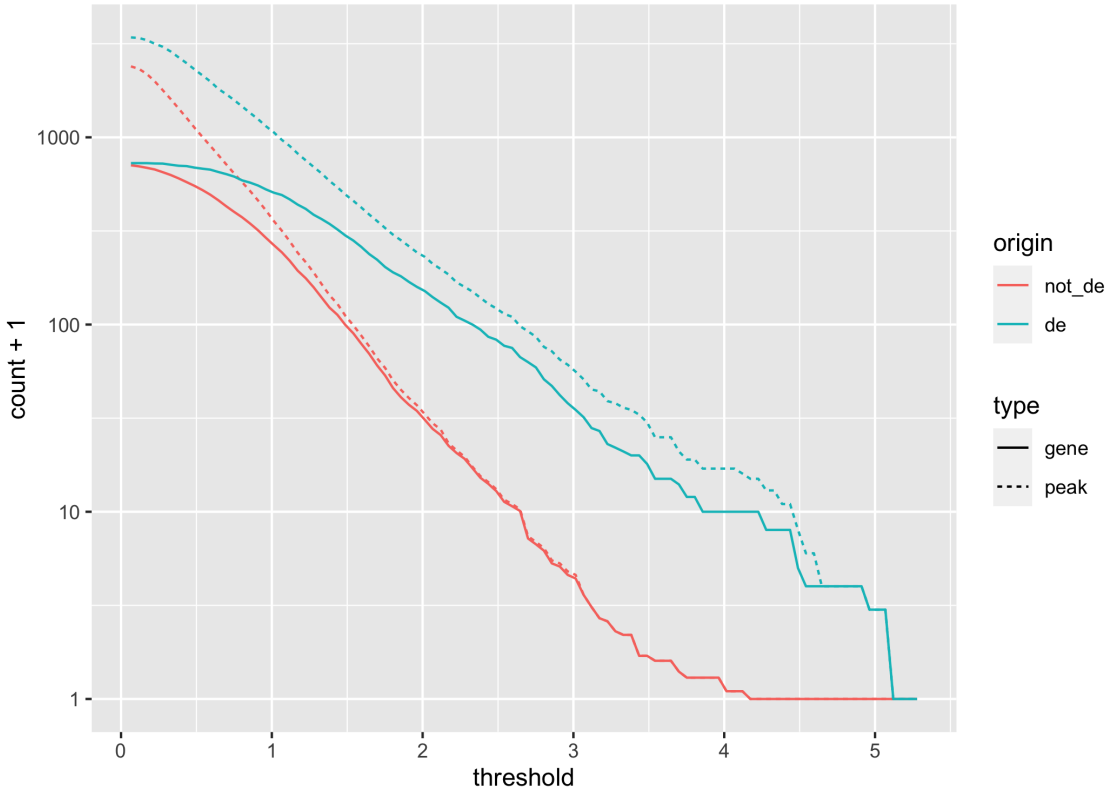


Figure 3.5: *A line chart displaying how gene and peak counts change as the absolute DA LFC threshold increases. Lines are colored according to whether they represent a gene that is DE or not. Note the x-axis is on a log_{10} scale.*

3.5 Discussion

We have shown that by using `plyranges` and `tximeta` (with support of Bioconductor and `tidyverse` ecosystems) we can fluently iterate through the biological data science workflow: from import, through to modeling, and data integration.

There are several further steps that would be interesting to perform in this analysis; for example, we could modify window size around the TSS to see how it affects enrichment, and vary the FDR cut-offs for both the DE gene and DA peak sets. We could also have computed variance in addition to the mean of the bootstrap set, and so drawn an interval around the enrichment line.

Finally, our workflow illustrates the benefits of using appropriate data abstractions provided by Bioconductor such as the *SummarizedExperiment* and *GRanges*. These abstractions provide users with a mental model of their experimental data and are the building blocks for constructing the modular and iterative analyses we have shown here. Consequently, we have been able to interoperate many decoupled R packages (from both Bioconductor and the tidyverse) to construct a seamless end-to-end workflow that is far too specialized for a single monolithic tool.

3.6 Software Availability

The workflow materials can be fully reproduced following the instructions found at the Github repository sa-lee/fluentGenomics. Moreover, the development version of the workflow and all downstream dependencies can be installed using the `BiocManager` package by running:

```
# development version from Github
BiocManager::install("sa-lee/fluentGenomics")
# version available from Bioconductor
BiocManager::install("fluentGenomics")
```

Acknowledgements

We would like to thank all participants of the Bioconductor 2019 and BiocAsia 2019 conferences who attended and provided feedback on early versions of this workflow paper.

Chapter 4

Tidy coverage analysis and visualisation with superintronic

4.1 Introduction

4.2 Methods

4.2.1 Motivation: intron signal in RNA-seq data

4.3 Results

4.3.1 Representation of coverage estimation

4.3.2 Integration of external gene annotations

4.3.3 Discovery of regions of interest via ‘ranglers’

4.3.4 A superintronic workflow for uncovering intron retention

4.4 Discussion

4.5 Conclusion

Chapter 5

Casting multiple shadows: high-dimensional interactive data visualisation with tours and embeddings

There has been a rapid uptake in the use of non-linear dimensionality reduction (NLDR) methods such as t-distributed stochastic neighbour embedding (t-SNE) in the natural sciences as part of cluster orientation and dimension reduction workflows. The appropriate use of these methods is made difficult by their complex parameterisations and the multitude of decisions required to balance the preservation of local and global structure in the resulting visualisation. We present a visual analytics framework for the pragmatic usage of NLDR methods by combining them with a technique called the tour. A tour is a sequence of interpolated linear projections of multivariate data onto a lower dimensional space. The sequence is displayed as a dynamic visualisation, allowing a user to see the shadows the high-dimensional data casts in a lower dimensional view. By linking the tour to a view obtained from an NLDR method, we can preserve global structure and through user interactions like spatial linked brushing observe where the NLDR view may be misleading. We display show several case studies from single cell genomics, that our approach

is useful for cluster orientation tasks. The implementation of our framework is available as an R package called `liminal` available at https://github.com/sa-lee/liminal.

5.1 Introduction

High dimensional data is increasingly prevalent in the natural sciences and beyond but presents a challenge to the analyst in terms of both data cleaning / pre-processing and visualisation. Methods to embed data from a high-dimensional space into a low-dimensional one now form a core step of the data analysis workflow where they are used to ascertain hidden structure and de-noise data for downstream analysis (thereby nullifying the 'curse of dimensionality').

Choosing an appropriate embedding presents a challenge to the analyst. How does an analyst know whether the embedding has captured the underlying topology and geometry of the high dimensional space? The answer depends on the analyst's workflow. Brehmer et al. (2014) characterised two main workflow steps that an analyst performs when using embedding techniques: dimension reduction and cluster orientation. The first relates to dimension reduction achieved by using an embedding method, here an analyst wants to characterise and map meaning onto the embedded form, for example identifying batch effects from a high throughput sequencing experiment, or identifying a gradient or trajectory along the embedded form Nguyen and Holmes (2019). The second relates to using embeddings as part of a clustering workflow. Here analysts are interested in identifying and naming clusters and verifying them by either applying known labels or colouring by variables that are a-priori known to distinguish clusters. Both of these workflow steps rely on the embedding being 'faithful' or the original high dimensional dataset, and become much more difficult when there is no underlying ground truth.

Embedding methods can be classified into two broad groups: linear and non-linear methods. Linear methods perform a linear transformation of the data; one example is principal components analysis (PCA) which performs an eigendecomposition of the estimated sample covariance matrix. The eigenvalues are sorted in decreasing order and represent the variance explained by each component (eigenvector). A common approach

to deciding on the number of principal components to retain is to plot the proportion of variance explained by each component and choose a cut-off.

Non-linear methods generally perform pre-processing on the high-dimensional data such as generating a neighborhood graph and perform transformations on the pre-processed form. We restrict our attention to three methods that are commonly used in high-throughput biology: t-distributed stochastic neighbor embedding (t-SNE), uniform manifold alignment and projection (UMAP), and potential of heat-diffusion for affinity-based transition embedding (PHATE). The t-SNE algorithm estimates the similarity of (Euclidean) distances of points in a high dimensional space using a Gaussian distribution and then estimates a configuration in the low dimensional embedding space by modelling similarities using a t-distribution with 1 degree of freedom. The resulting configuration is the one that minimizes the Kullback-Leibler divergence between the two distributions. A recent theoretical contribution by Linderman and Steinerberger (2019) proved that t-SNE can recover spherical and well separated cluster shapes, and proposed new approaches for tuning the optimisation parameters. It is a known problem that t-SNE can have trouble recovering global structure and that configurations can be highly dependent on how the algorithm is initialised and parameterized (Wattenberg, Viégas, and Johnson, 2016; Kobak and Berens, 2019). UMAP is a method that is related to LargeVis (Tang et al., 2016), and like t-SNE acts on the k-nearest neighbor graph. Its main differences are that it uses a different cost function (cross entropy) which is optimized using stochastic gradient descent and defines a different kernel for similarities in the low dimensional space. Due to it's computational speed it's possible to generate UMAP embeddings in more than three dimensions. Finally, PHATE, inspired by diffusion maps, is based on estimating an affinity matrix via a distance matrix and k-nearest neighbors graph. The algorithm de-noises estimated distances in high dimensional space via transforming the affinity matrix into a Markov transition probability matrix and diffusing this matrix over a fixed number of time steps. Then the diffused probabilities are transformed once more to construct a distance matrix, and multidimensional scaling is performed to construct a 2d embedding for visualization.

As part of a visualization workflow, it's important to consider the perception and interpretation of embedding methods as well. Sedlmair, Munzner, and Tory (2013) showed that 2D scatter plots were mostly sufficient for detecting class separation, however they noted that often multiple embeddings were required. For the task of cluster identification, Lewis, Van der Maaten, and Sa (2012) showed experimentally that novice users of non-linear embedding techniques were more likely to consider clusters of points on a 2d scatter plot to be the result of a spurious embedding compared to advanced users who were aware of the inner workings of the embedding algorithm.

A complimentary approach for visualizing structure in high dimensional data is the tour. A tour is a sequence of projections of a high dimensional dataset onto a low-dimensional orthonormal basis matrix, that is represented as a dynamic visualization. The sequence of generated bases are interpolated to form the tour path, allowing a user to explore the subspace of projections. A grand tour corresponds to choosing new bases at random, and can give an overview of the structure in the data. Instead of picking projections at random, a guided tour can be used to generate a sequence 'interesting' projections as quantified by an index function. Given the dynamic nature of the tour, user interaction is important for controlling and exploring the visualisation: the tour has been used previously by Wickham, Cook, and Hofmann (2015) as tool for exploring statistical model fits and by Buja, Cook, and Swayne (1996) for exploring factorial experimental designs.

While there has been much work on the algorithmic details of the aforementioned embedding methods, there has been relatively few tools designed to assist users to interact with these techniques and assist them in making comparisons between embeddings and performing the aforementioned cluster orientation tasks. Several interactive interfaces have been proposed for evaluating or using embedding techniques: the Sleepwalk interface provides a click and highlight visualisation for colouring points in an embedding according to their distance in the original high-dimensional space (Ovchinnikova and Anders, 2019). The work by Pezzotti et al. (2017) provides a user guided and modified form of the t-SNE algorithm, that allows users to modified optimisation parameters in real-time. Similarly, the embedding projector is a web interface to running UMAP, t-SNE

or PCA live in the browser and provides interactions to color points, and highlights nearest neighbors (Smilkov et al., 2016).

There is no one-size fits all: finding an appropriate embedding for a given dataset is a difficult and somewhat poorly defined problem. For non-linear methods, there are a lot of parameters to explore that can have an effect on the resulting visualisation and interpretation. Interfaces for evaluating embeddings require interaction but should also be able to be incorporated into an analysts workflow. We propose a more pragmatic workflow inspired by incorporating interactive graphics and tours with embeddings that allows users to see a global overview of their high dimensional data and assists them with cluster orientation tasks. This workflow is incorporated into an R package called `liminal` (Available: `https://github.com/sa-lee/liminal`).

5.2 Design

We propose using tours as part of an analyst's workflow in in performing dimensionality reduction tasks. We have made extensive use of ensemble graphics, that is aligning related plots alongside each other to provide context. As we will see in the case studies, this allows analysts to quickly compare views from embedding methods and allows them to see how the embedding method alters the global structure of their data. Using ensembles allows the use of interaction techniques, that allow analysts to perform cluster orientation tasks via linking multiple views. This approach allows our interface, to achieve the three principles for interactive high-dimensional data visualisation outlined by Buja, Cook, and Swayne (1996): finding gestalt, posing queries, and making comparisons.

5.2.1 Finding Gestalt: focus and context

To investigate latent structure and the shape of a high dimensional dataset, a tour can be run without the use of an external embedding. It is often useful to first run principal components on the input as an initial dimension reduction step, and then tour a subset of those components instead, i.e. by selecting them from a scree plot. The default tour layout is a scatter plot with an axis layout displaying the magnitude and direction of each basis vector. Since the tour is dynamic, it is often useful to be able to pause and highlight

a particular view. In our interface, brushing will pause the tour path, allowing users to identify 'interesting' projections. The domain of the axis scales from running a tour is called the half range, and is computed by rescaling the input data onto hyper-dimensional unit cube. We bind the half range to a mouse wheel event, allowing a user to pan and zoom on the tour view dynamically. This is useful for peeling back dense clumps of points to reveal structure.

5.2.2 Posing Queries: multiple views, many contexts

We combine the tour view in a side by side layout with a scatter plot view from an embedding algorithm. The views are linked; analysts can brush a group of points in the embedding view, and see its shape in the tour view. This can be used for either manually identifying clusters, or verifying the location of clusters relative to other groupings. Consequently, simple linked brushing can alleviate a downfall of non-linear methods such as UMAP or t-SNE that break hierarchical structure of clusters. Aside from simple linked brushing, we can also pose more complex queries via interrogating the neighbourhood graph. Using a linked neighbourhood brush, we can visually investigate the nearest neighbour relationships in the high-dimensional space via brushing in the embedding view. The user can select the number of nearest neighbours directly, and modify the distance metric used for determining the neighbours. This allows users to interrogate the stability of clusters generated in the embedding view. Multiple brushes can be used to pose queries in either the tour view or the embedding view; interface controls allow these brushes to combine using logical operators such as 'and', 'or', or 'not'. The use of linked brushing goes beyond simple color highlights, allowing analysts to get a more holistic view of the effect of an embedding algorithm.

5.2.3 Making comparisons

TODO

5.3 Implementation

The `liminal` software is implemented as an open source R package R Core Team (2019b). All visualisations produced by the package were constructed using the `vegawidget` package, which is an R interface to the Vega-Lite grammar of interactive graphics. Ensemble graphics for arranging tour and embedding views are generated using the `shiny` packages. We have made extensive use of the `tourr` package for constructing tour paths.

5.3.1 Tours as a streaming data problem

5.3.2 Linking views via spatial brushes

5.4 Case Studies

5.4.1 Case Study 1: Fake trees

5.4.2 Case Study 2: Clustering PBMC 10x single cell RNA-seq data

5.4.3 Case Study 3: Single cell mouse retina data

5.5 Discussion

Chapter 6

Conclusion

6.1 Findings and limitations

6.2 Software development and impact

6.3 Further research

Bibliography

Alasoo, K, J Rodrigues, S Mukhopadhyay, A Knights, A Mann, K Kundu, HIPSCI-Consortium, C Hale, D G, and D Gaffney (2018). Shared genetic effects on chromatin and gene expression indicate a role for enhancer priming in immune response. *Nature Genetics* **50**, 424–431.

Alasoo, K and D Gaffney (2017). *Processed read counts from macrophage RNA-seq and ATAC-seq experiments*. https://doi.org/10.5281/zenodo.1188300.

Bache, SM and H Wickham (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. https://CRAN.R-project.org/package=magrittr.

Barrett, T, SE Wilhite, P Ledoux, C Evangelista, IF Kim, M Tomashevsky, KA Marshall, KH Phillippy, PM Sherman, M Holko, A Yefanov, H Lee, N Zhang, CL Robertson, N Serova, S Davis, and A Soboleva (2013). NCBI GEO: archive for functional genomics data sets—update. *Nucleic Acids Research* **41**(D1), D991–D995. eprint: /oup/backfile/content_public/journal/nar/41/d1/10.1093/nar/gks1193/2/gks1193.pdf.

Brehmer, M, M Sedlmair, S Ingram, and T Munzner (2014). Visualizing dimensionally-reduced data: Interviews with analysts and a characterization of task sequences. In: *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*. dl.acm.org, pp.1–8. https://dl.acm.org/doi/abs/10.1145/2669557.2669559.

Buja, A, D Cook, and DF Swayne (1996). Interactive High-Dimensional Data Visualization. *J. Comput. Graph. Stat.* **5**(1), 78–99.

Dale, RK, BS Pedersen, and AR Quinlan (2011). Pybedtools: a flexible Python library for manipulating genomic datasets and annotations. en. *Bioinformatics* **27**(24), 3423–3424.

Fowler, M (n.d.). *FluentInterface*. https://martinfowler.com/bliki/FluentInterface.html. Accessed: 2020-2-23. https://martinfowler.com/bliki/FluentInterface.html.

Frankish, A, GENCODE-consoritum, and P Flicek (2018). GENCODE reference annotation for the human and mouse genomes. *Nucleic Acids Research*.

Green, TRG and M Petre (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* **7**(2), 131–174.

Grolemund, G and H Wickham (2017). *R for Data Science*. en, p. 89. http://r4ds.had.co.nz/.

Hansen, KD, RA Irizarry, and Z Wu (2012). Removing technical variability in RNA-seq data using condition quantile normalization. *Biostatistics* **13** (2), 204–16.

Henry, L and H Wickham (2017). *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. http://rlang.tidyverse.org, https://github.com/r-lib/rlang. http://rlang.tidyverse.org.

Huber, W, VJ Carey, R Gentleman, S Anders, M Carlson, BS Carvalho, HC Bravo, S Davis, L Gatto, T Girke, R Gottardo, F Hahne, KD Hansen, RA Irizarry, M Lawrence, MI Love, J MacDonald, V Obenchain, AK Oleś, H Pagès, A Reyes, P Shannon, GK Smyth, D Tenenbaum, L Waldron, and M Morgan (2015a). Orchestrating high-throughput genomic analysis with Bioconductor. en. *Nat. Methods* **12**(2), 115–121.

Huber, W, VJ Carey, R Gentleman, S Anders, M Carlson, BS Carvalho, HC Bravo, S Davis, L Gatto, T Girke, R Gottardo, F Hahne, KD Hansen, RA Irizarry, M Lawrence, MI Love, J MacDonald, V Obenchain, AK Oleś, H Pagès, A Reyes, P Shannon, GK Smyth, D Tenenbaum, L Waldron, and M Morgan (2015b). Orchestrating high-throughput genomic analysis with Bioconductor. en. *Nature Methods* **12**(2), 115–121.

Kaitoua, A, Pinoli, P, Bertoni, M, and Ceri, S (2017). Framework for Supporting Genomic Operations. *IEEE Trans. Comput.* **66**(3), 443–457.

Kobak, D and P Berens (2019). The art of using t-SNE for single-cell transcriptomics. en. *Nat. Commun.* **10**(1), 5416.

Köster, J and S Rahmann (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* **28**(19), 2520–2522.

Kozanitis, Christos, Heiberg, Andrew, Varghese, George, and Bafna, Vineet (2014). Using Genome Query Language to uncover genetic variation. en. *Bioinformatics* **30**(1), 1–8.

Kozanitis, C and DA Patterson (2016). GenAp: a distributed SQL interface for genomic data. en. *BMC Bioinformatics* **17**, 63.

Law, CW, M Alhamdoosh, S Su, X Dong, L Tian, GK Smyth, and ME Ritchie (2018). RNA-seq analysis is easy as 1-2-3 with limma, Glimma and edgeR. en. *F1000 Research* **5**(1408), 1408.

Lawrence, M, W Huber, H Pagès, P Aboyoun, M Carlson, R Gentleman, M Morgan, and V Carey (2013a). Software for Computing and Annotating Genomic Ranges. *PLoS Comput. Biol.* **9**.

Lawrence, M, W Huber, H Pagès, P Aboyoun, M Carlson, R Gentleman, M Morgan, and V Carey (2013b). Software for Computing and Annotating Genomic Ranges. *PLoS Comput. Biol.* **9**.

Lee, S, D Cook, and M Lawrence (2019). plyranges: a grammar of genomic data transformation. *Genome Biology* **20**(1), 4.

Lee, S, M Lawrence, and D Cook (2018). *plyranges: a grammar of genomic data transformation*. https://doi.org/10.5281/zenodo.1469841.

Lee, S, M Lawrence, and MI Love (2020). Fluent genomics with *plyranges* and *tximeta*. en. *F1000Res.* **9**(109), 109.

Lee, S, AY Zhang, S Su, AP Ng, AZ Holik, ML Asselin-Labat, ME Ritchie, and CW Law (2019). "Covering all your bases: incorporating intron signal from RNA-seq data". en. https://www.biorxiv.org/content/10.1101/352823v2.

Lewis, J, L Van der Maaten, and V de Sa (2012). A behavioral investigation of dimensionality reduction. In: *Proceedings of the Annual Meeting of the Cognitive Science Society*. Vol. 34. escholarship.org. https://escholarship.org/content/qt0ss3m55c/qt0ss3m55c.pdf.

Linderman, GC and S Steinerberger (2019). Clustering with t-SNE, Provably. *SIAM Journal on Mathematics of Data Science* **1**(2), 313–332.

Love, MI, S Anders, V Kim, and W Huber (2016). RNA-Seq workflow: gene-level exploratory analysis and differential expression. en. *F1000 Research* **4**(1070), 1070.

Love, MI, W Huber, and S Anders (2014). Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2. *Genome Biology* **15** (12), 550.

Love, MI, C Soneson, PF Hickey, LK Johnson, N Tessa Pierce, L Shepherd, M Morgan, and R Patro (2019). Tximeta: reference sequence checksums for provenance identification in RNA-seq. en. *bioRxiv*, 777888.

Morgan, M (2017). *AnnotationHub: Client to access AnnotationHub resources*. R package version 2.13.1.

Nguyen, LH and S Holmes (2019). Ten quick tips for effective dimensionality reduction. en. *PLoS Comput. Biol.* **15**(6), e1006907.

Ovchinnikova, S and S Anders (2019). "Exploring dimension-reduced embeddings with Sleepwalk". en. https://www.biorxiv.org/content/10.1101/603589v1.

Pagès, H, P Aboyoun, R Gentleman, and S DebRoy (2018). *Biostrings: Efficient manipulation of biological strings*. R package version 2.49.0.

Patro, R, G Duggal, M Love, R Irizarry, and C Kingsford (2017). Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods* **14**, 417–419.

Pezzotti, N, BPF Lelieveldt, L Van Der Maaten, T Hollt, E Eisemann, and A Vilanova (2017). Approximated and User Steerable tSNE for Progressive Visual Analytics. en. *IEEE Trans. Vis. Comput. Graph.* **23**(7), 1739–1752.

Quinlan, AR and IM Hall (2010). BEDTools: a flexible suite of utilities for comparing genomic features. en. *Bioinformatics* **26**(6), 841–842.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. https://www.R-project.org/.

R Core Team (2019a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. https://www.R-project.org/.

R Core Team (2019b). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. https://www.R-project.org/.

Riemondy, KA, RM Sheridan, A Gillen, Y Yu, CG Bennett, and JR Hesselberth (2017). valr: Reproducible Genome Interval Arithmetic in R. *F1000Research*.

Roadmap Epigenomics Consortium, A Kundaje, W Meuleman, J Ernst, M Bilenky, A Yen, A Heravi-Moussavi, P Kheradpour, Z Zhang, J Wang, MJ Ziller, V Amin, JW Whitaker, MD Schultz, LD Ward, A Sarkar, G Quon, RS Sandstrom, ML Eaton, YC

Wu, AR Pfenning, X Wang, M Claussnitzer, Y Liu, C Coarfa, RA Harris, N Shoresh, CB Epstein, E Gjoneska, D Leung, W Xie, RD Hawkins, R Lister, C Hong, P Gascard, AJ Mungall, R Moore, E Chuah, A Tam, TK Canfield, RS Hansen, R Kaul, PJ Sabo, MS Bansal, A Carles, JR Dixon, KH Farh, S Feizi, R Karlic, AR Kim, A Kulkarni, D Li, R Lowdon, G Elliott, TR Mercer, SJ Neph, V Onuchic, P Polak, N Rajagopal, P Ray, RC Sallari, KT Siebenthall, NA Sinnott-Armstrong, M Stevens, RE Thurman, J Wu, B Zhang, X Zhou, AE Beaudet, LA Boyer, PL De Jager, PJ Farnham, SJ Fisher, D Haussler, SJM Jones, W Li, MA Marra, MT McManus, S Sunyaev, JA Thomson, TD Tlsty, LH Tsai, W Wang, RA Waterland, MQ Zhang, LH Chadwick, BE Bernstein, JF Costello, JR Ecker, M Hirst, A Meissner, A Milosavljevic, B Ren, JA Stamatoyannopoulos, T Wang, and M Kellis (2015). Integrative analysis of 111 reference human epigenomes. *Nature* **518**(7539). NCBI GEO accession numbers GSM433167 https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSM433167 and GPL18952 https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GPL18952.

Sedlmair, M, T Munzner, and M Tory (2013). Empirical guidance on scatterplot and dimension reduction technique choices. en. *IEEE Trans. Vis. Comput. Graph.* **19**(12), 2634–2643.

Shepherd, L and M Morgan (2019). *BiocFileCache: Manage Files Across Sessions*. R package version 1.10.2.

Smilkov, D, N Thorat, C Nicholson, E Reif, FB Viégas, and M Wattenberg (2016). Embedding Projector: Interactive Visualization and Interpretation of Embeddings. arXiv: 1611.05469 [stat.ML].

Smyth, GK (2004). Linear Models and Empirical Bayes Methods for Assessing Differential Expression in Microarray Experiments. *Statistical Applications in Genetics and Molecular Biology* **3**(1).

Soneson, C, MI Love, and M Robinson (2015). Differential analyses for RNA-seq: transcript-level estimates improve gene-level inferences. *F1000Research* **4** (1521).

Tang, J, J Liu, M Zhang, and Q Mei (2016). Visualizing Large-scale and High-dimensional Data. arXiv: 1602.00370 [cs.LG].

Tukey, JW (1977). *Exploratory data analysis*. Vol. 2. Reading, Mass.: Addison-Wesley Pub. Co.

Wattenberg, M, F Viégas, and I Johnson (2016). How to Use t-SNE Effectively. *Distill* **1**(10).

Wickham, H (2014). Tidy Data. *Journal of Statistical Software, Articles* **59**(10), 1–23.

Wickham, H (2016). *ggplot2: Elegant Graphics for Data Analysis*. en. Use R! Springer International Publishing. https://play.google.com/store/books/details?id=RTMFswEACAAJ.

Wickham, H, M Averick, J Bryan, W Chang, LD McGowan, R François, G Grolemund, A Hayes, L Henry, J Hester, M Kuhn, TL Pedersen, E Miller, SM Bache, K Müller, J Ooms, D Robinson, DP Seidel, V Spinu, K Takahashi, D Vaughan, C Wilke, K Woo, and H Yutani (2019). Welcome to the tidyverse. *Journal of Open Source Software* **4**(43), 1686.

Wickham, H, D Cook, and H Hofmann (2015). Visualizing statistical models: Removing the blindfold. *Statistical Analysis and Data Mining: The ASA Data Science Journal* **8**(4), 203–225.

Wickham, H, R Francois, L Henry, and K Müller (2017). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4. https://CRAN.R-project.org/package=dplyr.

Wilkinson, L (2005). *The Grammar of Graphics*. en. Statistics and Computing. Springer Science & Business Media. https://market.android.com/details?id=book-_kRX4LoFfGQC.