Altuna Akalin

Computational Genomics With R

To our families

Contents

List of Tables

List of Figures

Preface

The aim of this book is to provide the fundamentals for data analysis for genomics. We developed this book based on the computational genomics courses we are giving every year. We have had invariably an interdisicplineary audience with backgrounds from physics, biology medicine, math, computer science or other quantitative fields. We want this book to be a starting point for computational genomics students and a guide for further data analysis in more specific topics in genomics. This is why we tried to cover a large variety of topics from programming to basic genome biology. As the field is interdisciplineary, it requires different starting points for people with different backgrounds. A biologist might skip sections on basic genome biology and start with R programming whereas a computer scientist might want to start with genome biology. In the same manner, a more experienced person might want to refer to this book when s/he needs a certain type of analysis where s/he does not have prior experience.

Who is this book for?

The book contains practical and theoretical aspects for computational genomics. Biology and medicine generate more data than ever before

[1] http://creativecommons.org/licenses/by-nc-sa/4.0/
[2] https://www.crcpress.com/product/isbn/

and we need to educate more people with data analysis skills and understanding of computational genomics. Since computational genomics is interdisciplinary; this book aims to be accessible for biologists, medical scientists, computer scientists and people from other quantitative backgrounds. We wrote this book for the following audiences:

- Biologists and medical scientists who generate the data and are keen on analyzing it themselves.
- Students and researchers who are formally starting to do research on or using computational genomics but do not have extensive domain specific knowledge but has at least a beginner level in a quantitative field: math, stats
- Experienced researchers looking for recipes or quick how-tos to get started in specific data analysis tasks relating to computational genomics.

What will you get out of this?

This resource describes the skills and provides how-tos that will help readers analyze their own genomics data.

After reading:

- If you are not familiar with R, you will get the basics of R and dive right in to specialized uses of R for computational genomics.
- you will understand genomic intervals and operations on them, such as overlap
- You will be able to use R and its vast package library to do sequence analysis: Such as calculating GC content for given segments of a genome or find transcription factor binding sites
- You will be familiar with visualization techniques used in genomics, such as heatmaps,meta-gene plots and genomic track visualization
- You will be familiar with supervised and unsupervised learning techniques which are important in data modelling and exploratory analysis of high-dimensional data
- You will be familiar with analysis of different high-throughput sequencing data sets mostly using R based tools.

Structure of the book

The book is designed with the idea that practical and conceptual understanding of data analysis methods is as important, if not more important, than the theoretical understanding, such as detailed derivation of equations in statistics or machine-learning. That is why we first try to give a conceptual explanation of the concepts then we try to give essential parts of the mathematical formulas for more detailed understanding. In this sprit, we always show the code and explain the code for a particular data analysis task. In addition, we give additional references such as books, websites and scientific papers for readers who desire to gain deeper theoretical understanding of data analysis related methods or concepts.

Introduction chapter introduce data analysis paradigms and what computational genomics looks like in practice. "Essentials for genomics" chapter introduces the basic concepts in genome biology and genomics. Understanding these concepts are important for computational genomics.

Chapters "Introduction to R" and "Statistics and Exploratory Data Analysis for Genomics" introduce the necessary programming and practical data analysis skillls for computational genomics. Related to these, "Operations on genomic intervals and genome arithmetic" introduces fundemental tools for dealing with genomic intervals and their relationship to eachother over the genome. These three chapters composes the necessary technical toolkit for analysis of specific high-throughput sequencing techniques.

Chapters "RNA-seq analysis", "ChIP-seq analysis" and "BS-seq analysis" deals with the analysis techniques for popular high-throughput sequencing techniques. The last chapter "multi-omics data integration" deals with methods for integrating multiple omics data sets.

To sum it up, this book is a comprehensive guide for computational genomics. Some sections are there for the sake of the wide interdisciplineary audience and completeness, and not all sections will equally useful to all readers from this broad audience.

Software information and conventions

This book is primarily about the using R packages to analyze genomics data, therefore if you want to reproduce the analysis in this book you need to install the relevant packages in each chapter using `install.packages` or `BiocManager::install` functions. We rely on data from different R and Bioconductor packages through out the book. For the datasets that do not ship with those packages, we created our own package compGenomRData.

Package names are in bold text (e.g., methylKit), and inline code and filenames are formatted in a typewriter font. Function names are followed by parentheses (e.g., `genomation::ScoreMatrix()`). The double-colon operator `::` means accessing an object from a package.

Acknowledgements

We wish to thank R and Bioconductor community for developing and maintaining libraries for genomic data analysis. Without their constant work and dedication, writing such a book will not be possible.

About the Authors

The authors have decades of combined experience in data analysis for genomics. They are developers of Bioconductor packages such as methylKit, genomation, RCAS and netSmooth. In addition, they have played key roles in developing end-to-end genomics data analysis pipelines for RNA-seq, ChIP-seq, Bisulfite-seq, and single cell RNA-seq called PiGx.

1

Introduction to Genomics

The aim of this chapter is to provide the reader with some of the fundamentals required for understanding genome biology. By no means, this is a complete overview of the subject but just a summary that will help the non-biologist reader understand the recurring biological concepts in computational genomics. Readers that are well-versed in genome biology and modern genome-wide quantitative assays should feel free to skip this chapter or skim it through.

1.1 Genes, DNA and central dogma

A central concept that will come up again and again is "the gene". Before we can explain that we need to introduce a few other concepts that are important to understand the gene concept. Human body is made up of billions of cells. These cells specialize in different tasks. For example, in the liver there are cells that help produce enzymes to break toxins. In the heart, there are specialized muscle cells that make the heart beat. Yet, all these different kinds of cells come from a single celled embryo. All the instructions to make different kinds of cells are contained within that single cell and with every division of that cell, those instructions are transmitted to new cells. These instructions can be coded into a string - a molecule of DNA, a polymer made of recurring units called nucleotides. The four nucleotides in DNA molecules, Adenine, Guanine, Cytosine and Thymine (coded as four letters: A, C, G, and T) in a specific sequence, store the information for life. DNA is organized in a double-helix form where two complementary polymers interlace with each other and twist into the familiar helical shape.

1.1.1 What is a genome?

The full DNA sequence of an organism, which contains all the hereditary information, is called a genome. The genome contains all the information to build and maintain an organism. Genomes come in different sizes and structures. Our genome is not only a naked strech of DNA. In eukaryotic cells, DNA is wrapped around proteins (histones) forming higher-order structures like nucleosomes which make up chromatins and chromosomes (see Figure 1.1).

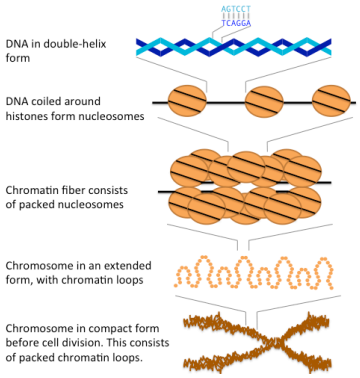


FIGURE 1.1: Chromosome structure in animals

There might be several chromosomes depending on the organism. However, in some species (such as most prokaryotes) DNA is stored in a circular form. The size of genome between species differs too. Human genome has 46 chromosomes and over 3 billion base-pairs, whereas wheat genome has 42 chromosomes and 17 billion base-pairs, both genome size and chromosome numbers are variable between different organisms. Genome sequences of organisms are obtained using sequencing technology. With this technology, fragments of the DNA sequence from the genome, called reads, are obtained. Larger chunks of the genome sequence is later obtained by stitching the initial fragments to larger ones by using the overlapping reads. Latest, sequencing technologies made genome sequencing cheaper and faster. These technologies output more reads, longer reads and more accurate reads.

Estimated cost of the first human genome is $300 million in 1999-2000, today a high-quality human genome can be obtained for $1500. Since the

costs are going down, researchers and clinicians can generate more data. This drives up to costs for data storage and also drives up the demand for qualified people to analyze genomic data. This was one of the motivations behind writing this book.

1.1.2 What is a gene?

In the genome, there are specific regions containing the precise information that encodes for physical products of genetic information. A region in the genome with this information is traditionally called a "gene". However, the precise definition of the gene is still developing. According to the classical textbooks in molecular biology, a gene is a segment of a DNA sequence corresponding to a single protein or to a single catalytic and structural RNA molecule [1]. A modern definition is: "A region (or regions) that includes all of the sequence elements necessary to encode a functional transcript" [2]. No matter how variable the definitions are, all agree on the fact that genes are basic units of heredity in all living organisms.

All cells use their hereditary information in the same way most of the time; the DNA is replicated to transfer the information to new cells. If activated, the genes are transcribed into messenger RNAs (mRNAs) in nucleus (in eukaryotes), followed by mRNAs (if the gene is protein coding) getting translated into proteins in the cytoplasm. This is essentially a process of information transfer between information carrying polymers; DNA, RNA and proteins, known as the "central dogma" of molecular biology (see Figure 1.2 for a summary). Proteins are essential elements for life. The growth and repair, functioning and structure of all living cells depends on them. This is why the gene is a central concept in genome biology, because a gene can encode information for proteins and other functional molecules. How genes are controled and activated dictates everything about an organism. From the identity of a cell to response to an infection, how cells develop and behave against certain stimuli is governed by activity of the genes and functional molecules they encode. The liver cell becomes a liver cell because certain genes are activated and their functional products are produced to help liver cell achieve its tasks.

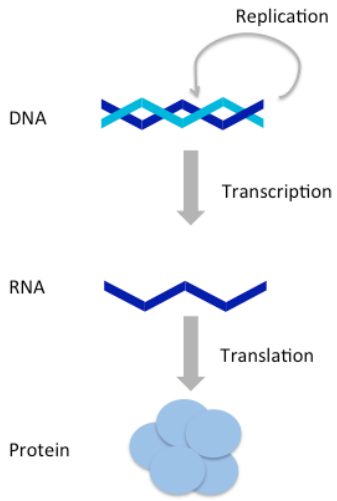


FIGURE 1.2: Central Dogma: replication, transcription, translation

1.1.3 How genes are controlled ? The transcriptional and the post-transcriptional regulation

In order to answer this question, we have to dig a little deeper on the transcription concept we introduced via the central dogma. The first step in a process of information transfer - a production of an RNA copy of a part of the DNA sequence - is called transcription. This task is carried out by the RNA polymerase enzyme. RNA polymerase-dependent initiation of transcription is enabled by the existence of a specific region in the sequence of DNA - a core promoter. Core promoters are regions of DNA that promote transcription and are found upstream from the start site of transcription. In eukaryotes, several proteins, called general transcription factors recognize and bind to core promoters and form a pre-initiation complex. RNA polymerases recognize these complexes and initiate synthesis of RNAs, the polymerase travels along the template DNA and making an RNA copy[3]. After mRNA is produced it is often spliced by splicesosome. The sections called 'introns' are removed and sections called 'exons' left in. Then, the remaining mRNA translated into proteins. Which exons will be

part of the final mature transcript can also be regulated and creates diversity in protein structure and function (See Figure 1.3').

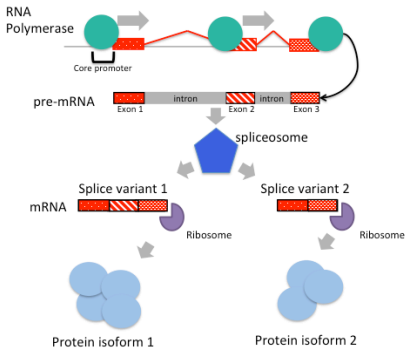


FIGURE 1.3: Transcription could be followed by splicing, which creates different transcript isoforms. This will in return create different protein isoforms since the information required to produce the protein is encoded in the transcripts. Differences in transcript of the same gene can give rise to different protein isoforms

On the contrary to protein coding genes, non-coding RNA (ncRNAs) genes are processed and assume their functional structures after transcription and without going into translation, hence the name: non-coding RNAs. certain ncRNAs can also be spliced but still not translated. ncRNAs and other RNAs in general can form complementary base-pairs within the RNA molecule which gives them additional complexity. This self-complementarity based structure, termed RNA secondary structure, is often necessary for functions of many ncRNA species.

In summary, the set of processes, from transcription initiation to production of the functional product, is referred to as gene expression. Gene expression quantification and regulation is a fundamental topic in genome biology.

1.1.4 What does a gene look like?

Before we move forward, it will be good to discuss how we can visualize genes. As someone interested in computational genomics, you will frequently encounter a gene on a computer screen, and how it is represented on the computer will be equivalent to what you imagine when you hear the word "gene". In the online databases, the genes will appear as a sequence of letters or as a series of connected boxes showing exon-intron structure which may include the direction of transcription as well (see Figure 1.4). direction. You will encounter more with the latter so this is likely what will pop into your mind when you think of genes.

As we have mentioned DNA has two strands, and a gene can be located on either of them, and direction of transcription will depend on that. In the Figure you can see arrows on introns (lines connecting boxes) indicating the direction of the gene.

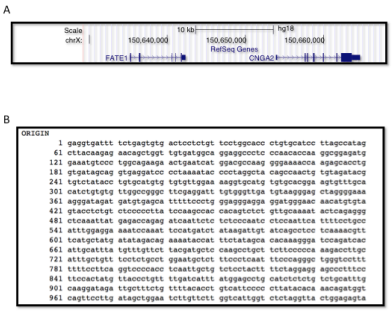


FIGURE 1.4: A) Representation of a gene at UCSC browser. Boxes indicate exons, and lines indicate introns. B) Partial sequence of FATE1 gene as shown in NCBI GenBank database.

1.2 Elements of gene regulation

The mechanisms regulating gene expression are essential for all living organisms as they dictate where and how much of a gene product (may it be protein or ncRNA) should be manufactured. This regulation could occur at the pre- and co-transcriptional level by controlling how many transcripts should be produced and/or which version of the transcript should be produced by regulating splicing. Different versions of the same gene could encode for proteins by regulating splicing the process can decide which parts will go into the final mRNA that will code for the protein. In addition, gene products can be regulated post-transcriptionally where certain molecules bind to RNA and mark them for degradation even before they can be used in protein production.

Gene regulation drives cellular differentiation; a process during which different tissues and cell types are produced. It also helps cells maintain differentiated states of cells/tissues. As a product of this process, at the final stage of differentiation, different kinds of cells maintain different expression profiles although they contain the same genetic material. As mentioned above there are two main types of regulation and next we will provide information on those.

1.2.1 Transcriptional regulation

The rate of transcription initiation is the primary regulatory element in gene expression regulation. The rate is controlled by core promoter elements as well as distant-acting regulatory elements such as enhancers. On top of that, processes like histone modifications and/or DNA methylation have a crucial regulatory impact on transcription. If a region is not accessible for the transcriptional machinery, e.g. in the case when chromatin structure is compacted due to the presence of specific histone modifications, or if the promoter DNA is methylated, transcription may not start at all. Last but the not least, gene activity is also controlled post-transcriptionally by ncRNAs such as microRNAs (miRNAs), as well as by

cell signaling resulting in protein modification or altered protein-protein interactions.

1.2.1.1 Regulation by transcription factors through regulatory regions

Transcripton factors are proteins that recognize a specific DNA motif to bind on a regulatory region and regulate the transcription rate of the gene associated with that regulatory region (See Figure 1.5)‘ for an illustration). These factors bind to a variety of regulatory regions summarized in Figure 1.5, and their concerted action controls the transcription rate. Apart from their binding preference, their concentration, the availability of synergistic or competing transcription factors will also affect the transcription rate.

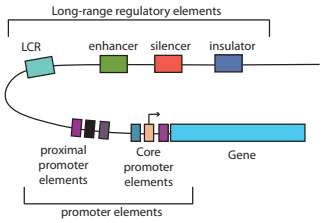


FIGURE 1.5: Representation of regulatory regions in animal genomes

1.2.1.1.1 Core and proximal promoters

Core promoters are the immediate neighboring regions around the transcription start site (TSS) that serves as a docking site for the transcriptional machinery and pre-initiation complex (PIC) assembly. The textbook model for transcription initiation is as follows: The core promoter has a TATA motif (referred as TATA-box) 30 bp upstream of an initiator sequence (Inr), which also contains TSS. Firstly, transcription factor TFIID binds to the TATA-box. Next, general transcription factors are recruited and transcription is initiated on the initiator sequence. Apart from the TATA-box and Inr, there are a number of sequence elements on the animal core promoters that are associated with transcription initiation and

PIC assembly, such as downstream promoter elements (DPEs), the BRE elements and CpG islands. DPEs are found 28-32 bp downstream of the TSS in TATA-less promoters of Drosophila melanogaster, it generally co-occurs with the Inr element, and is thought to have a similar function to the TATA-box. The BRE element is recognized by TFIIB protein and lies upstream of the TATA-box. CpG islands are CG dinucleotide-enriched segments of vertebrate genomes, despite the general depletion of CG dinucleotides in those genomes. 50-70% of promoters in human genome are associated with CpG islands.

Proximal promoter elements are typically right upstream of the core promoters and usually contain binding sites for activator transcription factors and they provide additional control over gene expression.

1.2.1.1.2 Enhancers:

Proximal regulation is not the only, nor the most important mode of gene regulation. Most of the transcription factor binding sites in the human genome are found in intergenic regions or in introns . This indicates the widespread usage of distal regulatory elements in animal genomes. On a molecular function level, enhancers are similar to proximal promoters; they contain binding sites for the same transcriptional activators and they basically enhance the gene expression. However, they are often highly modular and several of them can affect the same promoter at the same time or in different time-points or tissues. In addition, their activity is independent of their orientation and their distance to the promoter they interact with. A number of studies showed that enhancers can act upon their target genes over several kilobases away. According to a popular model, enhancers achieve this by looping the DNA and coming to contact with their target genes.

1.2.1.1.3 Silencers:

Silencers are similar to enhancers; however their effect is opposite of en-

hancers on the transcription of the target gene, and results in decreasing their level of transcription. They contain binding sites for repressive transcription factors. Repressor transcription factors can either block the binding of an activator , directly compete for the same binding site, or induce a repressive chromatin state in which no activator binding is possible. Silencer effects, similar to those of enhancers, are independent of orientation and distance to target genes. In contradiction to this general view, in Drosophila there are two types of silencers, long-range and short-range. Short-range silencers are close to promoters and long-range silencers can silence multiple promoters or enhancers over kilobases away. Like enhancers, silencers bound by repressors may also induce changes in DNA structure by looping and creating higher order structures. One class of such repressor proteins, which is thought to initiate higher-order structures by looping, is Polycomb group proteins (PcGs).

1.2.1.1.4 Insulators:

Insulator regions limit the effect of other regulatory elements to certain chromosomal boundaries; in other words, they create regulatory domains untainted by the regulatory elements in regions outside that domain. Insulators can block enhancer-promoter communication and/or prevent spreading of repressive chromatin domains. In vertebrates and insects, some of the well-studied insulators are bound by CTCF (CCCTC-binding factor). Genome-wide studies from different mammalian tissues confirm that CTCF binding is largely invariant of cell type, and CTCF motif locations are conserved in vertebrates. At present, there are two models of explaining the insulator function; the most prevalent model claims insulators create physically separate domains by modifying chromosome structure. This is thought to be achieved by CTCF-driven chromatin looping and recent evidence shows that CTCF can induce a higher-order chromosome structure through creating loops of chromatins. According to the second model, an insulator-bound activator cannot bind an enhancer; thus enhancer-blocking activity is achieved and insulators can also recruit active histone domain, creating an active domain for enhancers to function.

1.2.1.1.5 Locus control regions:

Locus control regions (LCRs) are clusters of different regulatory elements that control entire set of genes on a locus. LCRs help genes achieve their temporal and/or tissue-specific expression programs. LCRs may be composed of multiple cis-regulatory elements, such as insulators, enhancers and they act upon their targets even from long distances. However LCRs function with an orientation dependent manner, for example the activity of beta-globin LCR is lost if inverted. The mechanism of LCR function otherwise seems similar to other long-range regulators described above. The evidence is mounting in the direction of a model where DNA-looping creates a chromosomal structure in which target genes are clustered together, which seems to be essential for maintaining open chromatin domain.

1.2.1.2 Epigenetic regulation

Epigenetics in biology usually refers to constructions (chromatin structure, DNA methylation etc.) other than DNA sequence that influence gene regulation. In essence, epigenetic regulation is the regulation of DNA packing and structure, the consequence of which is gene expression regulation. A typical example is that DNA packing inside the nucleus can directly influence gene expression by creating accessible regions for transcription factors to bind. There are two main mechanisms in epigenetic regulation: i) DNA modifications ii) histone modifications. Below, we will introduce these two mechanisms.

1.2.1.2.1 DNA modifications such as methylation:

DNA methylation is usually associated with gene silencing. DNA methyltransferase enzyme catalyzes the addition of a methyl group to cytosine of CpG dinucleotides (while in mammals the addition of methyl group is largely restricted to CpG dinucleotides, methylation can occur in other bases as well) . This covalent modification either interferes with transcrip-

tion factor binding on the region, or methyl-CpG binding proteins induce the spread of repressive chromatin domains, thus the gene is silenced if its promoter has methylated CG dinucleotides. DNA methylation usually occurs in repeat sequences to repress transposable elements, these elements when active can jump around and insert them to random parts of the genome, potentially distrupting the genomic functions.

.DNA methylation is also related to a key core and proximal promoter element: CpG islands. CpG islands are usually unmethylated, however for some genes CpG island methylation accompanies their silenced expression. For example, during X-chromosome inactivation many CpG islands are heavily methylated and the associated genes are silenced. In addition, in embryonic stem cell differentiation pluripotency-associated genes are silenced due to DNA methylation. Apart from methylation, there are other kinds of DNA modifications present in mamalian genomes, such as hydroxy-methylation and formylcytosine. These are other modifications under current research that are either intermediate or stable modifications with distinct functional associations. There are as many as 12 distinct DNA modifications observed when we look accross all studied species.

1.2.1.2.2 Histone Modifications:

Histones are proteins that constitute nucleosome. In eukaryotes, eight histones nucleosomes are wrapped around by DNA and build nucleosome. They help super-coiling of DNA and inducing high-order structure called chromatin. In chromatin, DNA is either densely packed (called heterochromatin or closed chromatin), or it is loosely packed (called euchromatin or open chromatin) [60, 61]. Heterochromatin is thought to harbor inactive genes since DNA is densely packed and transcriptional machinery cannot access it. On the other hand, euchromatin is more accessible for transcriptional machinery and might therefore harbor active genes. Histones have long and unstructured N-terminal tails which can be covalently modified. The most studied modifications include acetylation, methylation and phosphorylation [60]. Using their tails, histones interact with neighboring nucleosomes and the modifications on the tail af-

fect the nucleosomes affinity to bind DNA and therefore influence DNA packaging around nucleosomes. Different modifications on histones are used in different combinations to program the activity of the genes during differentiation. Histone modifications have a distinct nomenclature, for example: H3K4me3 means the lysine (K) on the 4th position of histone H3 is tri-methylated.

TABLE 1.1: Table 1 Histone modifications and their effects. If more than one histone modification has the same effect, they are separated by commas.

Modifications	Effect
H3K9ac	Active promoters and enhancers
H3K14ac	Active transcription
H3K4me3/me2/me1 Active p	romoters and enhancers, H3K4me1 and H3K27ac is enhancer-specific
H3K27ac	H3K27ac is enhancer-specific
H3K36me3	Active transcribed regions
H3K27me3/me2/me1	Silent promoters
H3K9me3/me2/me1	Silent promoters

Histone modifications are associated with a number of different transcription-related conditions; some of them are summarized in Table 1. Histone modifications can indicate where the regulatory regions are and they can also indicate activity of the genes. From a gene regulatory perspective, maybe the most important modifications are the ones associated with enhancers and promoters. Certain genes in mouse embryonic stem cells have both active H3K4me3 and inactive H3K27me3 modifications in their promoters. Surprisingly, most of these genes have high CpG content, are important for development and are shown to have paused RNA polymerase II [66, 67]. In addition, Heintzman et al. showed that H3K4me1 could predict tissue-specific active enhancers in human cells [68, 69]. The examples above demonstrate the capability of histone modifications in predicting regulatory potential.

Furthermore, certain proteins can influence chromatin structure by interacting with histones. Some of these proteins, like those of the Polycomb Group (PcG) and CTCF, are discussed above in the insulators and silencer

sections. In vertebrates and insects, PcGs are responsible for maintaining the silent state of developmental genes, and trithorax group proteins (trxG) for maintaining their active state [70, 71]. PcGs and trxGs induce repressed or active states by catalyzing histone modifications or DNA methylation. Both the proteins bind PREs that can be on promoters or several kilobases away [30, 31, 71]. Another protein that induces histone modifications is CTCF. In b-globin locus, CTCF binding is shown to be associated with repressive H3K9/K27me2 modifications [72].

Want to know more ?

- Transcriptional regulatory elements in the human genome: http://www.ncbi.nlm.nih.gov/pubmed/16719718
- On metazoan promoters: types and transcriptional properties: http://www.ncbi.nlm.nih.gov/pubmed/22392219
- General principles of regulatory sequence function http://www.nature.com/nrg/journal/v15/n7/abs/nrg3684.html
- DNA methylation: roles in mammalian development http://www.nature.com/doifinder/10.1038/nrg3354
- Histone modifications and organization of the genome http://www.nature.com/nrg/journal/v12/n1/full/nrg2905.html
- DNA methylation and histone modifications are linked http://www.nature.com/nrg/journal/v10/n5/abs/nrg2540.html

1.2.2 Post-transcriptional regulation

1.2.2.1 Regulation by non-coding RNAs

Recent years have witnessed an explosion in noncoding RNA (ncRNA)-related research. Many publications implicated ncRNAs as important regulatory elements. Plants and animals produce many different types of ncRNAs such as long non-coding RNAs (lncRNAs), small-interferring RNAs (siRNAs), microRNAs (miRNAs), promoter-associated RNAs (PARs) and small nucleolar RNAs (snoRNAs). lncRNAs are typically >200 bp long,

they are involved in epigenetic regulation by interacting with chromatin remodeling factors and they function in gene regulation. siRNAs are short double-stranded RNAs which are involved in gene-regulation and transposon control, they silence their target genes by cooperating with Argonaute proteins . miRNAs are short single-stranded RNA molecules that interact with their target genes by using their complementary sequence and mark them for quicker degradation. PARs may regulate gene expression as well: they are ~18-200bp long ncRNAs originating from promoters of coding genes. snoRNAs also shown to play roles in gene regulation, although they are mostly believed to guide ribosomal RNA modifications.

1.2.2.2 Splicing regulation

Splicing is regulated by regulatory elements on the pre-mRNA and proteins binding to those elements . Regulatory elements are categorized as splicing enhancers and repressors. They can be located either in exons or introns. Depending of their activity and their locations there are four types of regulatory elements: - exonic splicing enhancers (ESEs) - exonic splicing silencers (ESSs) - intronic splicing enhancers (ISEs) - intronic splicing silencers (ISSs).

The majority of splicing repressors are heterogeneous nuclear ribonucleoproteins (hnRNPs). If splicing repressor protein bind silencer elements they reduce the change of nearby site to be used as splice junction. On the contrary, splicing enhancers are sites to which splicing activator proteins bind and binding on that region increases the probability that a nearby site will be used as a splice junction. Most of the activator proteins that bind to splicing enhancers are members of the SR protein family. Such proteins can recognize specific RNA recognition motifs. By regulating splicing exons can be skipped or included which creates protein diversity.

Want to know more ?

- On miRNAs: Their genesis and modes of regulation http://www.sciencedirect.com/science/article/pii/S0092867404000455
- Functions of small RNAs http://www.nature.com/nrg/journal/v15/n9/abs/nrg3765.html

- Functions of non coding RNAs http://www.nature.com/nrg/journal/v15/n6/abs/nrg3722.html
- on splicing: Wang, Zefeng; Christopher B. Burge (May 2008). "Splicing regulation: From a parts list of regulatory elements to an integrated splicing code". RNA 14 (5): 802–813. doi:10.1261/rna.876308. ISSN 1355-8382. PMC 2327353. PMID 18369186. Retrieved 2013-08-15.

1.3 Shaping the genome: DNA mutation

Human and chimpanzee genomes are 98.8% similar. The 1.2% difference is what separetes us from chimpanzees. The further you move away from human in terms of evolutionary distance the higher the difference gets. However, even between the members of the same species differences in genome sequences exists. These differences are due to a process called mutation which drives differences between individuals but also the provides the fuel for evolution as the source of the genetic variation. Individuals with beneficial mutations can adapt to their surroundings better than others and in time these mutations which are beneficial for survival spreads in the population due to a process called "natural selection". Selection acts upon individuals with beneficial features which gives them an edge for survival in a given environment. Genetic variation created by the mutations in individuals provide the material on which selection can act upon. If the selection process goes for a long time in a relatively isolated environment that requires adaptation, this population can evolve into a different species given enough time.This is the basic idea behind evolution in a nutshell, and without mutations providing the genetic variation there will be no evolution.

Mutations in the genome occur due to multiple reasons. First, DNA replication is not an error-free process. Before a cell division, the DNA is replicated with 1 mistake per 10^8 to 10^10 base-pairs. Second, mutagens such as UV light can induce mutations on the genome. Third factor that con-

tributes to mutation is imperfect DNA repair. Every day any human cell suffers multiple instances DNA damage. DNA repair enzymes are there to cope with this damage but they are also not error-free, depending on which DNA repair mechanism is used (there are multiple) mistakes will be made at varying rates.

Mutations are classified by how many bases they effect, their effect on DNA structure and gene function. By their effect on DNA structure the mutations are classified as follows:

- Base substitution: A base is changed with another.
- Deletion: One or more bases is deleted.
- Insertion: New base or bases inserted into the genome.
- Microsatellite mutation: Small insertions or deletions of small tandemly repeating DNA segments.
- Inversion: A DNA fragment changes its orientation 180 degrees.
- Translocation: A DNA fragment moves to another location in the genome.

Mutations can also be classified by their size as follows:

- Point mutations: mutations that involve one base. Substitutions, deletions and insertions are point mutations. They are also termed as single nucleotide polymorphisms (SNPs).
- small-scale mutations: mutations that involve several bases.
- Large-scale mutations: mutations which involve larger chromosomal regions. Transposable element insertions (where a segment of the genome jumps to another region in the genome) and segmental duplications (a large region is copied multiple times in tandem) are typical large scale mutations.
- Aneuploidies: Insertions or deletions of whole chromosomes.
- Whole-genome polyploidies: duplications involving whole genome.

Mutations by their effect on gene function can be classified as follows:

- gain-of-function mutations: A type of mutation in which the altered gene product possesses a new molecular function or a new pattern of gene expression.
- loss-of-function mutations: A mutation that results in reduced or

abolished protein function. This is the more common type of mutation.

1.4 High-throughput experimental methods in genomics

Most of the biological phenomena described above relating to transcription , gene regulation or DNA mutation can be measured over the entire genome using high-throughput experimental techniques, which are quickly becoming the standard for studying genome biology. In addition, their applications in the clinic are also gaining momemntum: there are already diagnostic tests that are based on these techniques.

Some of the things that can be measured by high-throughput assays are as follows:

- Which genes are expressed and how much ?
- Where does a transcription factor bind ?
- Which bases are methylated in the genome ?
- Which transcripts are translated ?
- Where does RNA-binding proteins bind ?
- Which microRNAs are expressed ?
- Which parts of the genome are in contact with each other ?
- Where are the mutations in the genome located ?
- Which parts of the genome are nucleosome-free ?

There are many more questions one can answer using modern genome-wide techniques and every other day a new variant of the existing techniques comes along to answer a new question. However, One has to keep in mind that these methods are at varying degrees of maturity and they all come with technical limitations and are not noise-free. Despite this, they are extremely useful for research and clinical purposes. And, thanks to these methods we are able to sequence and annotate genomes at a massive scale.

1.4.1 The general idea behind high-throughput techniques

High-throughput methods aim to quantify or locate all or most of the genome that harbours the biological feature (expressed genes, binding sites, etc.) of interest. Most of the methods rely on some sort of enrichment of the the targeted biological feature. For example, if you want to measure expression of protein coding genes you need to be able to extract mRNA molecules with special post-transcriptional alterations that protein-coding genes acquire. If you are looking for transcription factor binding, you need to enrich for the DNA fragments that are bound by the protein if interest.This parts depends on available molecular biology and chemistry technques, and the final product of this part is RNA or DNA fragments.

Next, you need to be able to tell where these fragments are coming from in the genome and how many of them are there. Microarrays were the standard tool for the quantification step until spread of sequencing techniques. In microarrays, one had to design complementary bases ,called "oligos" or "probes", to the genetic material enriched via the experimental protocol. If the enriched material is complementary to the genetic material, a light signal will be produced and the intensity of the signal will be proportional to the amount of the genetic material pairing with that oligo. There will be more probes available for hybrdization (process of complementary bases forming bonds), so the more fragments available stronger the signal. For this to be able to work, you need to know at least part of your genome sequence, and design probes. If you want to measure gene expression, your probes should overlap with genes and should be unique enough to not to bind sequences from other genes. This technology is now being replaced with sequencing technology, where you directly sequence your genetic material. If you have the sequence of your fragments, you can align them back to genome, see where they are coming from and count them. This is a better technology where the quantification is based on the real identiy of fragments rather than based on hybridization to designed probes.

In summary HT techniques has the following steps, and this also summarized in Figure 1.6:

- Extraction: This is the step where you extract the genetic material of interest, RNA or DNA. and nrichment
- Enrichment: In this step, you enrich for the event you are interested in. For example, protein binding sites. In some cases such as whole-genome DNA sequencing there is no need for enrichment step. You just get fragments of genomic DNA and sequence them.
- Quantification: This is where you quantify your enriched material. Depending on the protocol you may need to quantify a control set as well, where you should see no enrichment or only background enrichment.

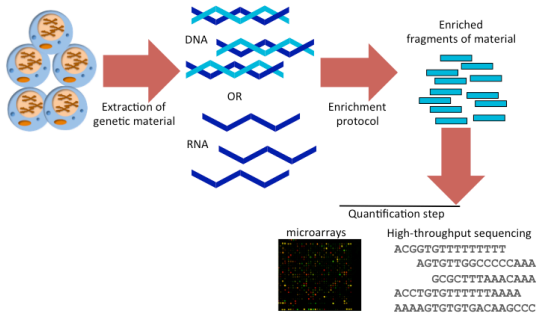


FIGURE 1.6: Common steps of High-throughput assays in genome biology

1.4.2 High-throughput sequencing

High-throughput sequencing, or massively parallel sequencing, is a collection of methods and technologies that can sequence DNA thousands/millons of fragments at a time. This is in contrast to older technologies that can produce a limited number of fragments at a time. Here, throughput refers to number of sequenced bases per hour. The older low-throughput sequencing methods have ~ 100 times less throughput compared to modern high-throughput methods. The increased throughput gives the ability to measure biological features on a genome-wide scale in a shorter time frame.

Similar to other high-throughput methods, sequencing based methods also require an enrichment step. This step enriches for the features we

are interested in. The main difference of the sequencing based methods is the quantification step. In high-throughput sequencing, enriched fragments are put through the sequencer which outputs the sequences for the fragments. Due to limitations in current leading technologies, only limited number of bases can be sequenced using from the input fragments. However, the length is usually enough to uniquely map the reads to the genome and quantify the input fragments.

1.4.2.1 High-throughput sequencing data

If there is a genome available, the reads are aligned to the genome and based on the sequencing protocol different strategies are applied for analysis. Some of the potential analysis strategies and processed output of read alignments are depicted in Figure 1.7. For example, we maybe interested to quantify the gene expression. The experimental protocol, called RNA sequencing- RNA-seq, enriches for fragments of RNA that are coming from protein coding genes. Upon alignment, we can calculate the coverage profile which gives us a read count per base along the genome. This information can be stored in a text file or specialized file formats to be used in subsequent analysis or visualization. We can also just count how many reads overlap with exons of each gene and record read counts per gene for further analysis. This essentially produces a table with gene names and read counts for different samples. As we will see in later chapters, this is an essential information for statistical models that model RNA-seq data. Furthermore, we can stack up the reads and count how many times we see a base position in a read mismatches the base in the genome. Read aligners allow for mismatches, and for this reason we can see reads with mismatches. This information can be used to identify SNPs, and can be stored again in a tabular format with the information of position and mismatch type and number of reads supporting the mismatch. The original algorithms are a bit more complicated than just counting mismatches but the general idea is the same, what they are doing differently is trying to minimize false positive rates by using filters, so that not every mismatch is recorded as SNP.

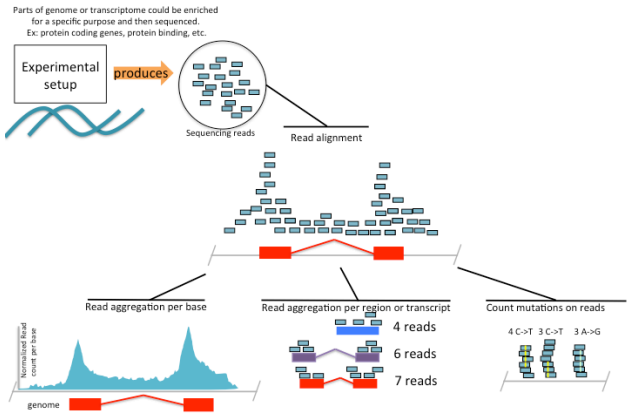


FIGURE 1.7: High-throughput sequencing summary

1.4.2.2 Future of high-throughput sequencing

The sequencing tech is still evolving. Next frontier is the obtaining longer single-molecule reads and preferably being able to call base modifications on the fly. With longer reads, the genome asssembly will be easier for the regions that have high repeat content. With single-molecule sequencing, we will be able tell how many transcripts are present in a given cell population without relying on fragment amplification methods which can introduce biases.

Another recent development is single-cell sequencing. Current technologies usually work on genetic material from thousands to millions of cells. This means that the results you receive represents the population of cells that were used in the experiment. However, there is a lot variation between the same type of cells, this variation is not observed at all. Newer sequecing techniques can work on single cells and give quantititave information on each cell.

Want to know more ?

- Current and the future high-throughput sequencing technologies http://www.sciencedirect.com/science/article/pii/S1097276515003408

- Illumina repository for different library preperation protocols for sequencing http://www.illumina.com/techniques/sequencing/ngs-library-prep/library-prep-methods.html

1.5 Visualization and data repositories for genomics

There are ~100 animal genomes sequenced as of 2016. On top these, there are many research projects from either individual labs or consortiums that produce petabytes of auxiliary genomics data, such as ChIP-seq, RNA-seq, etc.

There are two requirements for to be able to visualize genomes and its associated data, 1) you need to be able to work with a species that has a sequenced genome and 2) you want to have annotation on that genome, meaning at the very least you want to know where the genes are. Most genomes after sequencing quickly annotated with gene-predictions or know gene sequences are mapped on to it, you can also have conservation to other species to filter functional elements. If you are working with a model organism or human you will also have a lot of auxiliary information to help demarcate the functional regions such as regulatory regions, ncRNA, SNPs that are common in the population. Or you might have disease or tissue specific data available. . The more the organism is worked on the more auxiliary data you will have.

1.5.0.1 Accessing genome sequences and annotations via genome browsers

As someone intends to work with genomics, you will need to visualize a large amount of data to make biological inferences or simply check regions of interest in the genome visually. Looking at the genome case by case with all the additional datasets is a necessary step to develop hypothesis and understand the data.

Many genomes and associated data is available through genome browsers.

A genome browser is a website or an app that helps you visualize the genome and all the available data associated with it. Via genome browsers, you will be able to see where genes are in relation to each other and other functional elements. You will be able to see gene structure. You will be able to see auxiliary data such as conservation, repeat content and SNPs. Here we review some of the popular browsers.

UCSC genome browser: This is an online browser hosted by University of California, Santa Cruz at http://genome.ucsc.edu/. This is an interactive website that contains genomes and annotations for many species. You can search for genes or genome coordinates for the species of your interest. It is usually very responsive and allows you to visualize large amounts of data. In addition, it has multiple other tools that can be used in connection with the browser. One of the most useful tool is UCSC Table Browser, which lets you download the all the data you see on the browser, including sequence data, in multiple formats . Users can upload data or provide links to the data to visualize user specific data.

Ensembl: This is another online browser maintained by
European Bioinformatics Institute and the Wellcome Trust Sanger Institute in the UK, http://www.ensembl.org. Similar to UCSC browser, users can visualize genes or genomic coordinates from multiple species and it also comes with auxiliary data. Ensemlb is associated with Biomart tool which is similar to UCSC Table browser, can download genome data including all the auxiliary data set in multiple formats.

IGV: Integrated genomics viewer (IGV) is a desktop application developed by Broad institute (https://www.broadinstitute.org/igv/). It is developed to deal with large amounts of high-throughput sequencing data which is harder to view in online browsers. IGV can integrate your local sequencing results with online annotation on your desktop machine. This is useful when viewing sequencing data, especially alignments. Other browsers mentioned above have similar functionalities however you will need to make your large sequencing data available online somewhere before it can be viewed by browsers.

1.5.0.2 Data repositories for high-throughput assays

Genome browser contain lots of auxiliary high-throughput data. However, there are many more public high-throughput data sets available and they are certainly not available through genome browsers. Normally, every high-throughput dataset associated with a publication should be deposited to public archieves. There are two major public archives we use to deposit data. One of them is Gene expression Omnibus(GEO) hosted at http://www.ncbi.nlm.nih.gov/geo/, and the other one is European nucleotide archive(ENA) hosted at http://www.ebi.ac.uk/ena. These repositories accept high-throughput datasets and users can freely download and use these public data sets for their own research. Many data sets in these repositories are in their raw format, for example the format the sequencer provides mostly. Some data sets will also have processed data but that is not a norm.

Apart from these repositories, there are multiple multi-national consortia that is dedicated to certain genome biology or disease related problems and they maintain their own databases and provide access to processed and raw data. Some of these consortia is mentioned below.

Consortium	what is it for?
ENCODE[1]	Transcription factor binding sites, gene expression and epigenomics data for cell lines
Epigenomics Roadmap[2]	Epigenomics data for multiple cell types
The cancer genome atlas (TCGA)[3]	Expression, mutation and epigenomics data for multiple cancer types
1000 genomes project[4]	Human genetic variation data obtained by sequencing 1000s of individuals

[1] https://www.encodeproject.org/
[2] http://www.roadmapepigenomics.org/
[3] http://cancergenome.nih.gov/
[4] http://www.1000genomes.org/

2

Introduction to R for genomic data analysis

The aim of computational genomics is to provide biological interpretation and insights from high dimensional genomics data. Generally speaking, it is similar to any other kind of data analysis endeavor but often times doing computational genomics will require domain specific knowledge and tools.

As new high-throughout experimental techniques on the rise, data analysis capabilities are sought-after features for researchers. The aim of this chapter is to first familiarize the readers with data analysis steps and then provide basics of R programming within the context of genomic data analysis.R is a free statistical programming language that is popular among researchers and data miners to build software and analyze data. Although basic R programming tutorials are easily accessible, we are aiming to introduce the subject with the genomic context in the background. The examples and narrative will always be from real-life situations when you try to analyze genomic data with R. We believe tailoring material to the context of genomics makes a difference when learning this programming language for sake of analyzing genomic data.

2.1 Steps of (genomic) data analysis

Regardless of the analysis type, the data analysis has a common pattern. We will discuss this general pattern and how it applies to genomics problems. The data analysis steps typically include data collection, quality check and cleaning, processing, modeling, visualization and reporting. Although, one expects to go through these steps in a linear fashion, it is normal to go back and repeat the steps with different parameters or tools.

In practice, data analysis requires going through the same steps over and over again in order to be able to do a combination of the following: a) answering other related questions, b) dealing with data quality issues later realized, c) including new data sets to the analysis.

We will now go through brief explanation of the steps within the context of genomic data analysis.

2.1.1 Data collection

Data collection refers to any source, experiment or survey that provides data for the data analysis question you have. In genomics, data collection is done by high-throughput assays introduced in chapter 1. One can also use publicly available data sets and specialized data bases also mentioned in chapter 1. How much data and what type of data you should collect depends on the question you are trying to answer ant the technical and biological variability of the system you are studying.

2.1.2 Data quality check and cleaning

In general, the data analysis almost always deals with imperfect data. It is common to have missing values or measurements that are noisy. Data quality check and cleaning aims to identify any data quality issue and clean it from the dataset.

High-throughout genomics data is produced by technologies that could embed technical biases into the data. If we were to give an example from sequencing, the sequenced reads do not have the same quality of bases called. Towards the ends of the reads, you might have bases that might be called incorrectly. Identifying those low quality bases and removing them will improve read mapping step.

2.1.3 Data processing

This step refers to processing the data to a format that is suitable for exploratory analysis and modeling. Often times, the data will not come in

ready to analyze format. You may need to convert it to other formats by transforming data points (such as log transforming, normalizing etc), or subset the data set with some arbitrary or pre-defined condition. In terms of genomics, processing includes multiple steps. Following the sequencing analysis example above, processing will include aligning reads to the genome and quantification over genes or regions of interest. This simply counting how many reads are covering your regions of interest. This quantity can give you ideas about how much a gene expressed if your experimental protocol was RNA sequencing. This can be followed by some normalization to aid the next step.

2.1.4 Exploratory data analysis and modeling

This phase usually takes in the processed or semi-processed data and applies machine-learning or statistical methods to explore the data. Typically, one needs to see relationship between variables measured, relationship between samples based on the variables measured. At this point, we might be looking to see if the samples group as expected by the experimental design, are there outliers or any other anomalies ? After this step you might want to do additional clean up or re-processing to deal with anomalies.

Another related step is modeling. This generally refers to modelling your variable of interest based on other variables you measured. In the context of genomics, it could be that you are trying to predict disease status of the patients from expression of genes you measured from their tissue samples. Then your variable of interest is the disease status and . This is generally called predictive modeling and could be solved with regression based or any other machine-learning methods. This kind of approach is generally called “predictive modeling”.

Statistical modeling would also be a part of this modeling step, this can cover predictive modeling as well where we use statistical methods such as linear regression. Other analyses such as hypothesis testing, where we have an expectation and we are trying to confirm that expectation is also related to statistical modeling. A good example of this in genomics is the differential gene expression analysis. This can be formulated as comparing two data sets, in this case expression values from condition A and con-

dition B, with the expectation that condition A and condition B has similar expression values. You will see more on this on chapter 3.

2.1.5 Visualization and reporting

Visualization is necessary for all the previous steps more or less. But in the final phase, we need final figures, tables and text that describes the outcome of your analysis. This will be your report. In genomics, we use common data visualization methods as well as specific visualization methods developed or popularized by genomic data analysis. You will see many popular visualization methods in chapters 3 and @{genomicIntervals}

2.1.6 Why use R for genomics ?

R, with its statistical analysis heritage, plotting features and rich user-contributed packages is one of the best languages for the task of analyzing genomic data. High-dimensional genomics datasets are usually suitable to be analyzed with core R packages and functions. On top of that, Bioconductor and CRAN have an array of specialized tools for doing genomics specific analysis. Here is a list of computational genomics tasks that can be completed using R.

2.1.6.1 Data cleanup and processing

Most of general data clean up, such as removing incomplete columns and values, reorganizing and transforming data, these tasks can be achieved using R. In addition, with the help of packages R can connect to databases in various formats such as mySQL, mongoDB, etc., and query and get the data to R environment using database specific tools.

On top of these, genomic data specific processing and quality check can be achieved via R/Bioconductor packages. For example, sequencing read quality checks and even HT-read alignments can be achieved via R packages.

2.1.6.2 General data anaylsis and exploration

Most genomics data sets are suitable for application of general data analysis tools. In some cases, you may need to preprocess the data to get it to a state that is suitable for application such tools. Here is a non-exhaustive list of what kind of things can be done via R.

- unsupervised data analysis: clustering (k-means, hierarchical), matrix factorization (PCA, ICA etc)
- supervised data analysis: generalized linear models, support vector machines, randomForests

2.1.6.3 Genomics specific data analysis methods

R/Bioconductor gives you access to multitude of other bioinformatics specific algorithms. Here are some of the things you can do.

- Sequence analysis: TF binding motifs, GC content and CpG counts of a given DNA sequence
- Differential expression (or arrays and sequencing based measurements)
- Gene set/Pathway analysis: What kind of genes are enriched in my gene set
- Genomic Interval opertaions such as Overlapping CpG islands with transcription start sites, and filtering based on overlaps
- Overlapping aligned reads with exons and counting aligned reads per gene

2.1.6.4 Visualization

Visualization is an important part of all data analysis techniques including computational genomics. Again, you can use core visualization techniques in R and also genomics specific ones with the help of specific packages. Here are some of the things you can do with R.

- Basic plots: Histograms, scatter plots, bar plots, box plots, heatmaps
- ideograms and circus plots for genomics provides visualization of different features over the whole genome.

- meta-profiles of genomic features, such as read enrichment over all promoters
- Visualization of quantitative assays for given locus in the genome

2.2 Getting started with R

Download and install R http://cran.r-project.org/ and RStudio http://www.rstudio.com/ if you do not have them already. Rstudio is optional but it is a great tool if you are just starting to learn R. You will need specific data sets to run the codes in this document. Download the data.zip[URL to come] and extract it to your directory of choice. The folder name should be "data" and your R working directory should be level above the data folder. That means in your R console, when you type "dir("data")" you should be able to see the contents of the data folder. You can change your working directory by setwd() command and get your current working directory with getwd() command in R. In RStudio, you can click on the top menu and change the location of your working directory via user interface.

2.2.1 Installing packages

R packages are add-ons to base R that help you achieve additional tasks that are not directly supported by base R. It is by the action of these extra functionality that R excels as a tool for computational genomics. Bioconductor project (http://bioconductor.org/) is a dedicated package repository for computational biology related packages. However main package repository of R, called CRAN, has also computational biology related packages. In addition, R-Forge(http://r-forge.r-project.org/), GitHub(https://github. com/), and googlecode(http://code.google.com) are other locations where R packages might be hosted. You can install CRAN packages using install.packages(). (# is the comment character in R)

```
# install package named "randomForests" from CRAN
install.packages("randomForests")
```

You can install bioconductor packages with a specific installer script

```
# get the installer package
source("http://bioconductor.org/biocLite.R")
# install bioconductor package "rtracklayer"
biocLite("rtracklayer")
```

You can install packages from github using install_github() function from devtools

```
library(devtools)
install_github("hadley/stringr")
```

Another way to install packages are from the source.

```
# download the source file
download.file("http://goo.gl/3pvHYI",
              destfile="methylKit_0.5.7.tar.gz")
# install the package from the source file
install.packages("methylKit_0.5.7.tar.gz",
                 repos=NULL,type="source")
# delete the source file
unlink("methylKit_0.5.7.tar.gz")
```

You can also update CRAN and Bioconductor packages.

```
# updating CRAN packages
update.packages()

# updating bioconductor packages
source("http://bioconductor.org/biocLite.R")
biocLite("BiocUpgrade")
```

2.2.2 Installing packages in custom locations

If you will be using R on servers or computing clusters rather than your personal computer it is unlikely that you will have administrator access to install packages. In that case, you can install packages in custom locations by telling R where to look for additional packages. This is done by setting up an .Renviron file in your home directory and add the following line:

```
R_LIBS=~/Rlibs
```

This tells R that "Rlibs" directory at your home directory will be the first choice of locations to look for packages and install packages (The directory name and location is up to you above is just an example). You should go and create that directory now. After that, start a fresh R session and start installing packages. From now on, packages will be installed to your local directory where you have read-write access.

2.2.3 Getting help on functions and packages

You can get help on functions by help() and help.search() functions. You can list the functions in a package with ls() function

```
library(MASS)
ls("package:MASS") # functions in the package
ls() # objects in your R enviroment
# get help on hist() function
?hist
help("hist")
# search the word "hist" in help pages
help.search("hist")
??hist
```

2.2.3.1 More help needed?

In addition, check package vignettes for help and practical understanding of the functions. All Bionconductor packages have vignettes that walk you through example analysis. Google search will always be helpful as well,

there are many blogs and web pages that have posts about R. R-help, Stackoverflow and R-bloggers are usually source of good and reliable information.

2.3 Computations in R

R can be used as an ordinary calculator, some say it is an over-grown calculator. Here are some examples. Remember # is the comment character. The comments give details about the operations in case they are not clear.

```
2 + 3 * 5       # Note the order of operations.
log(10)         # Natural logarithm with base e
5^2             # 5 raised to the second power
3/2             # Division
sqrt(16)       # Square root
abs(3-7)       # Absolute value of 3-7
pi              # The number
exp(2)         # exponential function
# This is a comment line
```

2.4 Data structures

R has multiple data structures. If you are familiar with excel you can think of data structures as building blocks of a table and the table itself, and a table is similar to a sheet in excel. Most of the time you will deal with tabular data sets, you will manipulate them, take sub-sections of them. It is essential to know what are the common data structures in R and how they can be used. R deals with named data structures, this means you can give names to data structures and manipulate or operate on them using those names.

2.4.1 Vectors

Vectors are one the core R data structures. It is basically a list of elements of the same type (numeric,character or logical). Later you will see that every column of a table will be represented as a vector. R handles vectors easily and intuitively. You can create vectors with c() function, however that is not the only way. The operations on vectors will propagate to all the elements of the vectors.

```
x<-c(1,3,2,10,5)    #create a vector named x with 5 components
x = c(1,3,2,10,5)
x
```

```
## [1]  1  3  2 10  5
```

```
y<-1:5              #create a vector of consecutive integers y
y+2                 #scalar addition
```

```
## [1] 3 4 5 6 7
```

```
2*y                 #scalar multiplication
```

```
## [1]  2  4  6  8 10
```

```
y^2                 #raise each component to the second power
```

```
## [1]  1  4  9 16 25
```

```
2^y                 #raise 2 to the first through fifth power
```

```
## [1]  2  4  8 16 32
```

```
y                   #y itself has not been unchanged
```

```
## [1] 1 2 3 4 5
```

```
y<-y*2
y                    #it is now changed
```

```
## [1]  2  4  6  8 10
```

```
r1<-rep(1,3)         # create a vector of 1s, length 3
length(r1)            #length of the vector
```

```
## [1] 3
```

```
class(r1)             # class of the vector
```

```
## [1] "numeric"
```

```
a<-1                 # this is actually a vector length one
```

2.4.2 Matrices

A matrix refers to a numeric array of rows and columns. You can think of it as a stacked version of vectors where each row or column is a vector. One of the easiest ways to create a matrix is to combine vectors of equal length using cbind(), meaning 'column bind'.

```
x<-c(1,2,3,4)
y<-c(4,5,6,7)
m1<-cbind(x,y);m1
```

```
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
## [4,] 4 7
```

```
t(m1)                # transpose of m1
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y    4    5    6    7
```

```
dim(m1)              # 2 by 5 matrix
```

```
## [1] 4 2
```

You can also directly list the elements and specify the matrix:

```
m2<-matrix(c(1,3,2,5,-1,2,2,3,9),nrow=3)
m2
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    2
## [2,]    3   -1    3
## [3,]    2    2    9
```

Matrices and the next data structure data frames are tabular data structures. You can subset them using and providing desired rows and columns to subset. Figure 2.1 shows how that works conceptually.

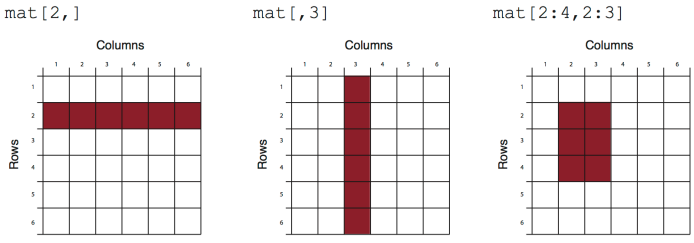


FIGURE 2.1: slicing/subsetting of a matrix and a data frame

2.4.3 Data Frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.). A data frame can be constructed by data.frame() function. For example, we illustrate how to construct a data frame from genomic intervals or coordinates.

```
chr <- c("chr1", "chr1", "chr2", "chr2")
strand <- c("-","-","+","+")
start<- c(200,4000,100,400)
end<-c(250,410,200,450)
mydata <- data.frame(chr,start,end,strand)
#change column names
names(mydata) <- c("chr","start","end","strand")
mydata # OR this will work too
```

```
##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +
```

```
mydata <- data.frame(chr=chr,start=start,end=end,strand=strand)
mydata
```

```
##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +
```

There are a variety of ways to extract the elements of a data frame. You can extract certain columns using column numbers or names, or you can extract certain rows by using row numbers. You can also extract data using logical arguments, such as extracting all rows that has a value in a column larger than your threshold.

```
mydata[,2:4] # columns 2,3,4 of data frame
```

```
##   start end strand
## 1   200 250      -
## 2  4000 410      -
## 3   100 200      +
## 4   400 450      +
```

```
mydata[,c("chr","start")] # columns chr and start from data frame
```

```
##    chr start
## 1 chr1   200
## 2 chr1  4000
## 3 chr2   100
## 4 chr2   400
```

```
mydata$start # variable start in the data frame
```

```
## [1]  200 4000  100  400
```

```
mydata[c(1,3),] # get 1st and 3rd rows
```

```
##    chr start end strand
## 1 chr1   200 250      -
## 3 chr2   100 200      +
```

```
mydata[mydata$start>400,] # get all rows where start>400
```

```
##    chr start end strand
## 2 chr1  4000 410      -
```

2.4.4 Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components
# a string, a numeric vector, a matrix, and a scalar
w <- list(name="Fred",
       mynumbers=c(1,2,3),
       mymatrix=matrix(1:4,ncol=2),
       age=5.3)
w
```

```
## $name
## [1] "Fred"
##
## $mynumbers
## [1] 1 2 3
##
## $mymatrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $age
## [1] 5.3
```

You can extract elements of a list using the [] convention using either its position in the list or its name.

```
w[[3]] # 3rd component of the list
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
w[["mynumbers"]] # component named mynumbers in list
```

```
## [1] 1 2 3
```

```
w$age
```

```
## [1] 5.3
```

2.4.5 Factors

Factors are used to store categorical data. They are important for statistical modeling since categorical variables are treated differently in statistical models than continuous variables. This ensures categorical data treated accordingly in statistical models.

```
features=c("promoter","exon","intron")
f.feat=factor(features)
```

Important thing to note is that when you are reading a data.frame with read.table() or creating a data frame with data.frame() character columns are stored as factors by default, to change this behavior you need to set stringsAsFactors=FALSE in read.table() and/or data.frame() function arguments.

2.5 Data types

There are four common data types in R, they are numeric, logical, character and integer. All these data types can be used to create vectors natively.

```
#create a numeric vector x with 5 components
x<-c(1,3,2,10,5)
x
```

```
## [1]  1  3  2 10  5
```

```
#create a logical vector x
x<-c(TRUE,FALSE,TRUE)
x
```

```
## [1]  TRUE FALSE  TRUE
```

```
# create a character vector
x<-c("sds","sd","as")
x
```

```
## [1] "sds" "sd"  "as"
```

```
class(x)
```

```
## [1] "character"
```

```
# create an integer vector
x<-c(1L,2L,3L)
x
```

```
## [1] 1 2 3
```

```
class(x)
```

```
## [1] "integer"
```

2.6 Reading and writing data

Most of the genomics data sets are in the form of genomic intervals associated with a score. That means mostly the data will be in table format with columns denoting chromosome, start positions, end positions, strand and score. One of the popular formats is BED format used primarily by UCSC genome browser but most other genome browsers and tools

will support BED format. We have all the annotation data in BED format. In R, you can easily read tabular format data with read.table() function.

```
enhancerFilePath=system.file("extdata",
                "subset.enhancers.hg18.bed",
                package="compGenomRData")
cpgiFilePath=system.file("extdata",
                "subset.cpgi.hg18.bed",
                package="compGenomRData")
# read enhancer marker BED file
enh.df <- read.table(enhancerFilePath, header = FALSE)

# read CpG island BED file
cpgi.df <- read.table(cpgiFilePath, header = FALSE)

# check first lines to see how the data looks like
head(enh.df)
```

```
##      V1     V2     V3 V4   V5 V6    V7    V8 V9
## 1 chr20 266275 267925  . 1000  .  9.11 13.17 -1
## 2 chr20 287400 294500  . 1000  . 10.53 13.02 -1
## 3 chr20 300500 302500  . 1000  .  9.10 13.39 -1
## 4 chr20 330400 331800  . 1000  .  6.39 13.51 -1
## 5 chr20 341425 343400  . 1000  .  6.20 12.99 -1
## 6 chr20 437975 439900  . 1000  .  6.31 13.52 -1
```

```
head(cpgi.df)
```

```
##      V1     V2     V3       V4
## 1 chr20 195575 195851  CpG:_28
## 2 chr20 207789 208148  CpG:_32
## 3 chr20 219055 219437  CpG:_33
## 4 chr20 225831 227155 CpG:_135
## 5 chr20 252826 256323 CpG:_286
## 6 chr20 275376 276977 CpG:_116
```

You can save your data by writing it to disk as a text file. A data frame or

matrix can be written out by using write.table() function. Now let us write out cpgi.df, we will write it out as a tab-separated file, pay attention to the arguments.

```
write.table(cpgi.df,file="cpgi.txt",quote=FALSE,
            row.names=FALSE,col.names=FALSE,sep="\t")
```

You can save your R objects directly into a file using save() and saveRDS() and load them back in with load() and readRDS(). By using these functions you can save any R object whether or not they are in data frame or matrix classes.

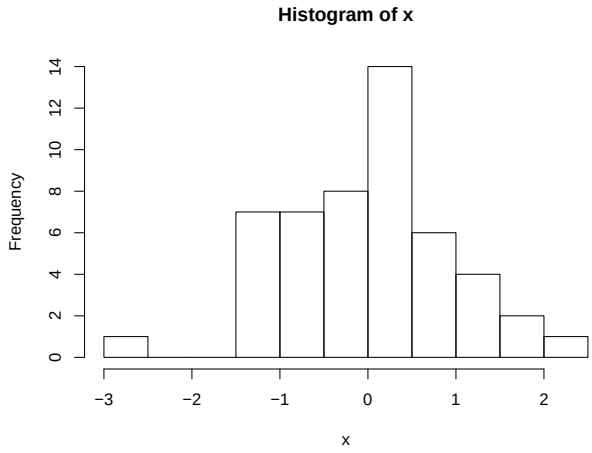
```
save(cpgi.df,enh.df,file="mydata.RData")
load("mydata.RData")
# saveRDS() can save one object at a type
saveRDS(cpgi.df,file="cpgi.rds")
x=readRDS("cpgi.rds")
head(x)
```

One important thing is that with `save()` you can save many objects at a time and when they are loaded into memory with `load(` they retain their variable names. For example, in the above code when you use `load("mydata.RData")` in a fresh R session, an object names "cpg.df" will be created. That means you have to figure out what name you gave it to the objects before saving them. On the contrary to that, when you save an object by `saveRDS()` and read by `readRDS()` the name of the object is not retained, you need to assign the output of `readRDS()` to a new variable ("x" in the above code chunk).

2.7 Plotting in R

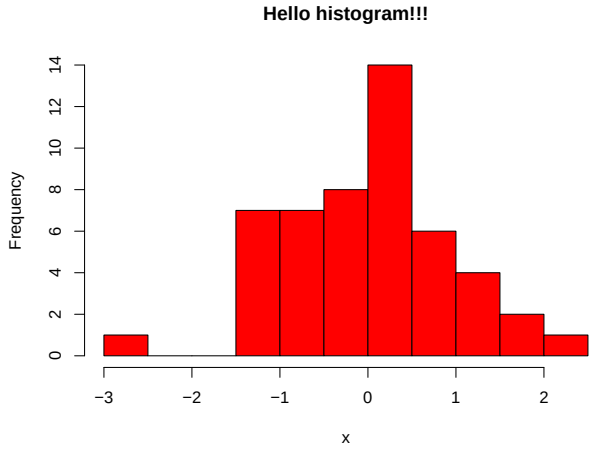
R has great support for plotting and customizing plots. We will show only a few below. Let us sample 50 values from normal distribution and plot them as a histogram.

```
# sample 50 values from normal distribution
# and store them in vector x
x<-rnorm(50)
hist(x) # plot the histogram of those values
```



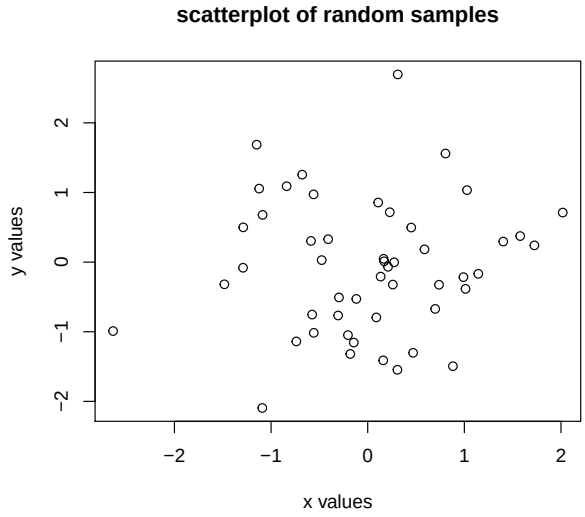
We can modify all the plots by providing certain arguments to the plotting function. Now let's give a title to the plot using 'main' argument. We can also change the color of the bars using 'col' argument. You can simply provide the name of the color. Below, we are using 'red' for the color. See Figure below for the result this chunk.

```
hist(x,main="Hello histogram!!!",col="red")
```



Next, we will make a scatter plot. Scatter plots are one the most common plots you will encounter in data analysis. We will sample another set of 50 values and plotted those against the ones we sampled earlier. Scatterplot shows values of two variables for a set of data points. It is useful to visualize relationships between two variables. It is frequently used in connection with correlation and linear regression. There are other variants of scatter plots which show density of the points with different colors. We will show examples of those that in following chapters. The scatter plot from our sampling experiment is shown in the figure. Notice that, in addition to main we used "xlab" and "ylab" arguments to give labels to the plot. You can customize the plots even more than this. See ?plot and ?par for more arguments that can help you customize the plots.

```
# randomly sample 50 points from normal distribution
y<-rnorm(50)
#plot a scatter plot
# control x-axis and y-axis labels
plot(x,y,main="scatterplot of random samples",
        ylab="y values",xlab="x values")
```



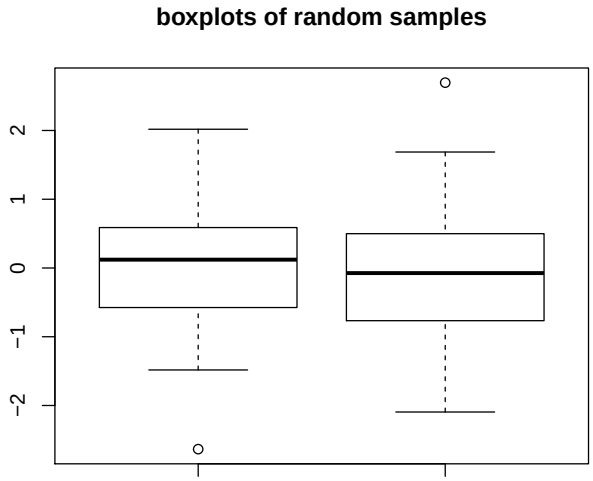
we can also plot boxplots for vectors x and y. Boxplots depict groups of numerical data through their quartiles. The edges of the box denote 1st

and 3rd quartile, and the line that crosses the box is the median. Whiskers usually are defined using interquantile range:

lowerWhisker=Q1-1.5[IQR] and upperWhisker=Q1+1.5[IQR]

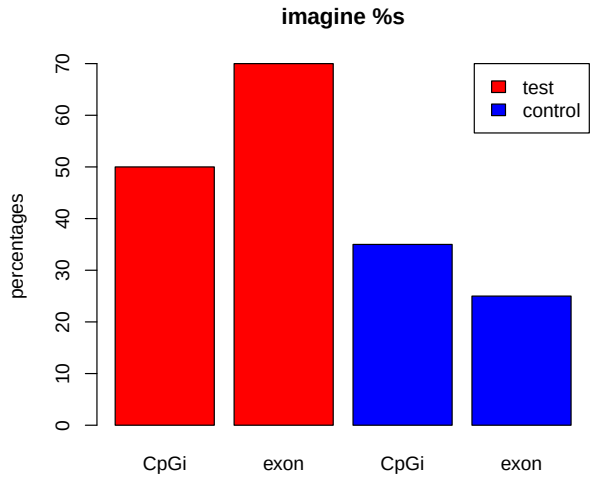
In addition, outliers can be depicted as dots. In this case, outliers are the values that remain outside the whiskers.

```
boxplot(x,y,main="boxplots of random samples")
```



Next up is bar plot which you can plot by barplot() function. We are going to plot four imaginary percentage values and color them with two colors, and this time we will also show how to draw a legend on the plot using legend() function.

```
perc=c(50,70,35,25)
barplot(height=perc,
        names.arg=c("CpGi","exon","CpGi","exon"),
        ylab="percentages",main="imagine %s",
        col=c("red","red","blue","blue"))
legend("topright",legend=c("test","control"),
       fill=c("red","blue"))
```



2.8 Saving plots

If you want to save your plots to an image file there are couple of ways of doing that. Normally, you will have to do the following: 1. Open a graphics device 2. Create the plot 3. Close the graphics device

```
pdf("mygraphs/myplot.pdf",width=5,height=5)
plot(x,y)
dev.off()
```

Alternatively, you can first create the plot then copy the plot to a graphic device.

```
plot(x,y)
dev.copy(pdf,"mygraphs/myplot.pdf",width=7,height=5)
dev.off()
```

2.9 Functions and control structures (for, if/else etc.)

2.9.1 User defined functions

Functions are useful for transforming larger chunks of code to re-usable pieces of code. Generally, if you need to execute certain tasks with variable parameters then it is time you write a function. A function in R takes different arguments and returns a definite output, much like mathematical functions. Here is a simple function takes two arguments, x and y, and returns the sum of their squares.

```
sqSum<-function(x,y){
result=x^2+y^2
return(result)
}
# now try the function out
sqSum(2,3)
```

```
## [1] 13
```

Functions can also output plots and/or messages to the terminal. Here is a function that prints a message to the terminal:

```
sqSumPrint<-function(x,y){
result=x^2+y^2
cat("here is the result:",result,"\n")
}
# now try the function out
sqSumPrint(2,3)
```

```
## here is the result: 13
```

Sometimes we would want to execute a certain part of the code only if certain condition is satisfied. This condition can be anything from the type of an object (Ex: if object is a matrix execute certain code), or it can be more complicated such as if object value is between certain thresholds. Let us

see how they can be used3. They can be used anywhere in your code, now we will use it in a function.

```
cpgi.df <- read.table("intro2R_data/data/subset.cpgi.hg18.bed", header = FALSE)
# function takes input one row
# of CpGi data frame
largeCpGi<-function(bedRow){
 cpglen=bedRow[3]-bedRow[2]+1
 if(cpglen>1500){
    cat("this is large\n")
 }
 else if(cpglen<=1500 & cpglen>700){
    cat("this is normal\n")
 }
 else{
    cat("this is short\n")
 }
}
largeCpGi(cpgi.df[10,])
largeCpGi(cpgi.df[100,])
largeCpGi(cpgi.df[1000,])
```

2.9.2 Loops and looping structures in R

When you need to repeat a certain task or a execute a function multiple times, you can do that with the help of loops. A loop will execute the task until a certain condition is reached. The loop below is called a "for-loop" and it executes the task sequentially 10 times.

```
for(i in 1:10){ # number of repetitions
cat("This is iteration") # the task to be repeated
print(i)
}
```

```
## This is iteration[1] 1
## This is iteration[1] 2
```

```
## This is iteration[1] 3
## This is iteration[1] 4
## This is iteration[1] 5
## This is iteration[1] 6
## This is iteration[1] 7
## This is iteration[1] 8
## This is iteration[1] 9
## This is iteration[1] 10
```

The task above is a bit pointless, normally in a loop, you would want to do something meaningful. Let us calculate the length of the CpG islands we read in earlier. Although this is not the most efficient way of doing that particular task, it serves as a good example for looping. The code below will be execute hundred times, and it will calculate the length of the CpG islands for the first 100 islands in the data frame (by subtracting the end coordinate from the start coordinate).

Note:If you are going to run a loop that has a lot of repetitions, it is smart to try the loop with few repetitions first and check the results. This will help you make sure the code in the loop works before executing it for thousands of times.

```
# this is where we will keep the lenghts
# for now it is an empty vector
result=c()
# start the loop
for(i in 1:100){
    #calculate the length
    len=cpgi.df[i,3]-cpgi.df[i,2]+1
    #append the length to the result
    result=c(result,len)
}
# check the results
head(result)
```

```
## [1]  277  360  383 1325 3498 1602
```

2.9.2.1 apply family functions instead of loops

R has other ways of repeating tasks that tend to be more efficient than using loops. They are known as the "apply" family of functions, which include apply, lapply, mapply and tapply (and some other variants). All of these functions apply a given function to a set of instances and returns the result of those functions for each instance. The differences between them is that they take different type of inputs. For example apply works on data frames or matrices and applies the function on each row or column of the data structure. lapply works on lists or vectors and applies a function which takes the list element as an argument. Next we will demonstrate how to use apply() on a matrix. The example applies the sum function on the rows of a matrix, it basically sums up the values on each row of the matrix, which is conceptualized in Figure 2.2.

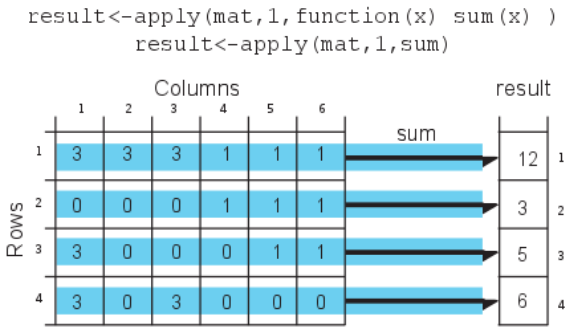


FIGURE 2.2: apply concept in R

```
mat=cbind(c(3,0,3,3),c(3,0,0,0),c(3,0,0,3),c(1,1,0,0),c(1,1,1,0),c(1,1,1,0))
result<-apply(mat,1,sum)
result
```

```
## [1] 12  3  5  6
```

```
# OR you can define the function as an argument to apply()
result<-apply(mat,1,function(x) sum(x))
result
```

```
## [1] 12  3  5  6
```

Notice that we used a second argument which equals to 1, that indicates that rows of the matrix/ data frame will be the input for the function. If we change the second argument to 2, this will indicate that columns should be the input for the function that will be applied. See Figure 2.3 for the visualization of apply() on columns.

```
result<-apply(mat,2,function(x) sum(x))
result<-apply(mat,2,sum)
```

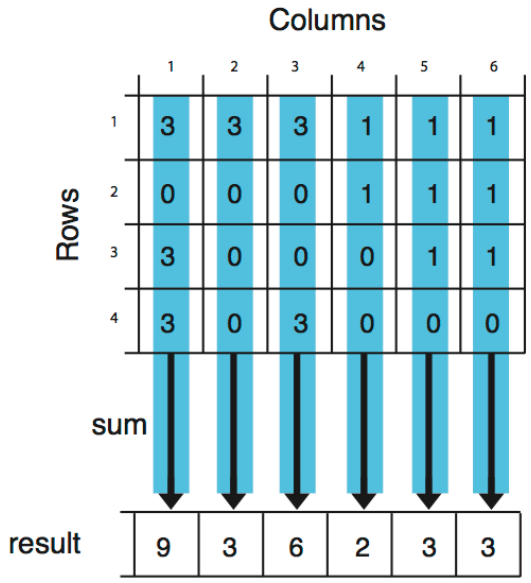


FIGURE 2.3: apply function on columns

```
result<-apply(mat,2,sum)
result
```

```
## [1] 9 3 6 2 3 3
```

Next, we will use lapply(), which applies a function on a list or a vector. The

function that will be applied is a simple function that takes the square of a given number.

```
input=c(1,2,3)
lapply(input,function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

mapply() is another member of apply family, it can apply a function on an unlimited set of vectors/lists, it is like a version of lapply that can handle multiple vectors as arguments. In this case, the argument to the mapply() is the function to be applied and the sets of parameters to be supplied as arguments of the function. This conceptualized Figure 2.4, the function to be applied is a function that takes to arguments and sums them up. The arguments to be summed up are in the format of vectors, Xs and Ys. `mapply()` applies the summation function to each pair in Xs and Ys vector. Notice that the order of the input function and extra arguments are different for mapply.

```
Xs=0:5
Ys=c(2,2,2,3,3,3)
result<-mapply(function(x,y) sum(x,y),Xs,Ys)
result
```

```
## [1] 2 3 4 6 7 8
```

2.9.2.2 apply family functions on multiple cores

If you have large data sets apply family functions can be slow (although probably still better than for loops). If that is the case, you can easily use

```
result<-mapply(function(x,y) sum(x,y),Xs,Ys)
result<-mapply(sum,Xs,Ys)
```

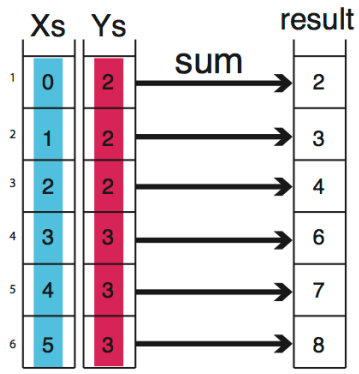


FIGURE 2.4: mapply concept

the parallel versions of those functions from parallel package. These functions essentially divide your tasks to smaller chunks run them on separate CPUs and merge the results from those parallel operations. This concept is visualized at Figure below , `mcapply` runs the summation function on three different processors. Each processor executes the summation function on a part of the data set, and the results are merged and returned as a single vector that has the same order as the input parameters Xs and Ys.

2.9.2.3 Vectorized Functions in R

The above examples have been put forward to illustrate functions and loops in R because functions using sum() are not complicated and easy to understand. You will probably need to use loops and looping structures with more complicated functions. In reality, most of the operations we used do not need the use of loops or looping structures because there are already vectorized functions that can achieve the same outcomes, meaning if the input arguments are R vectors the output will be a vector as well, so no need for loops or vectorization.

For example, instead of using mapply() and sum() functions we can just use + operator and sum up Xs and Ys.

```
result<-mcmapply(function(x,y) sum(x,y),
                 Xs,Ys,mc.cores=3)
result<-mcmapply(sum,Xs,Ys,mc.cores=3)
```

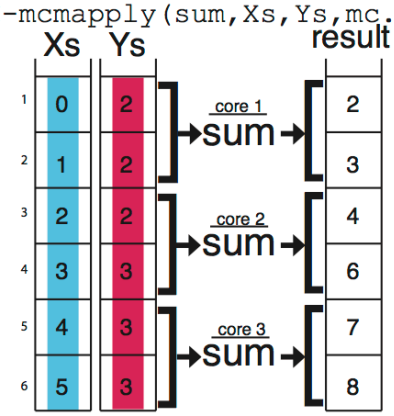


FIGURE 2.5: mcapplyconcept

```
result=Xs+Ys
result
```

```
## [1] 2 3 4 6 7 8
```

In order to get the column or row sums, we can use the vectorized functions colSums() and rowSums().

```
colSums(mat)
```

```
## [1] 9 3 6 2 3 3
```

```
rowSums(mat)
```

```
## [1] 12  3  5  6
```

However, remember that not every function is vectorized in R, use the ones that are. But sooner or later, apply family functions will come in handy.

2.10 Exercises

2.10.1 Computations in R

1. Sum 2 and 3, use +
2. Take the square root of 36, use sqrt()
3. Take the log10 of 1000, use function `log10()`
4. Take the log2 of 32, use function `log2()`
5. Assign the sum of 2,3 and 4 to variable x
6. Find the absolute value of `5 - 145` using `abs()` function
7. Calculate the square root of 625, divide it by 5 and assign it to variable `x`. Ex: `y= log10(1000)/5`, the previous statement takes log10 of 1000, divides it by 5 and assigns the value to variable y
8. Multiply the value you get from previous exercise with 10000, assign it variable x Ex: `y=y*5`, multiplies y with 5 and assigns the value to y.

KEY CONCEPT: results of computations or arbitrary values can be stored in variables we can re-use those variables later on and over-write them with new values

2.10.2 Data structures in R

10. Make a vector of 1,2,3,5 and 10 using `c()`, assign it to `vec` variable. Ex: `vec1=c(1,3,4)` makes a vector out of 1,3,4.
11. Check the length of your vector with length(). Ex: `length(vec1)` should return 3
12. Make a vector of all numbers between 2 and 15. Ex: `vec=1:6` makes a vector of numbers between 1 and 6, assigns to vec variable

13. Make a vector of 4s repeated 10 times using `rep()` function. Ex: `rep(x=2,times=5)` makes a vector of 2s repeated 5 times
14. Make a logical vector with TRUE, FALSE values of length 4, use `c()`. Ex: `c(TRUE,FALSE)`
15. Make a character vector of gene names PAX6,ZIC2,OCT4 and SOX2. Ex: `avec=c("a","b","c")` a makes a character vector of a,b and c
16. Subset the vector using `[]` notation, get 5th and 6th elements. Ex: `vec1[1]` gets the first element. `vec1[c(1,3)]` gets 1st and 3rd elements
17. You can also subset any vector using a logical vector in `[]`. Run the following:

```
myvec=1:5
myvec[c(TRUE,TRUE,FALSE,FALSE,FALSE)] # the length of the logical vector should be equal to le
myvec[c(TRUE,FALSE,FALSE,FALSE,TRUE)]
```

18. `==,>,<, >=, <=` operators create logical vectors. See the results of the following operations:

```
myvec > 3
myvec == 4
myvec <= 2
myvec != 4
```

19. Use `>` operator in `myvec[ ]` to get elements larger than 2 in `myvec` whic is described above
20. make a 5x3 matrix (5 rows, 3 columns) using `matrix()`. Ex: matrix(1:6,nrow=3,ncol=2) makes a 3x2 matrix using numbers between 1 and 6
21. What happens when you use `byrow = TRUE` in your matrix() as an

additional argument? Ex: `mat=matrix(1:6,nrow=3,ncol=2,byrow = TRUE)`

22. Extract first 3 columns and first 3 rows of your matrix using `[]` notation.
23. Extract last two rows of the matrix you created earlier.

Ex: `mat[2:3,]` or `mat[c(2,3),]` extracts 2nd and 3rd rows.

24. Extract the first two columns and run `class()` on the result.
25. Extract first column and run `class()` on the result, compare with the above exercise.
26. Make a data frame with 3 columns and 5 rows, make sure first column is sequence of numbers 1:5, and second column is a character vector.

Ex: `df=data.frame(col1=1:3,col2=c("a","b","c"),col3=3:1) # 3x3 data frame.`

Remember you need to make 3x5 data frame

27. Extract first two columns and first two rows.

HINT: Same notation as matrices

28. Extract last two rows of the data frame you made.

HINT: Same notation as matrices

29. Extract last two columns using column names of the data frame you made.
30. Extract second column using column names. You can use `[]` or `$` as in lists, use both in two different answers.
31. Extract rows where 1st column is larger than 3. HINT: you can get a logical vector using > operator ,logical vectors can be used in `[]` when subsetting.

32. Extract rows where 1st column is larger than or equal to 3.
33. Convert data frame to the matrix. HINT: use `as.matrix()`.

Observe what happens to numeric values in the data frame.

34. Make a list using `list()` function, your list should have 4 elements the one below has 2.

Ex: `mylist= list(a=c(1,2,3),b=c("apple,"orange"))`

35. Select the 1st element of the list you made using `$` notation.

Ex: `mylist$a` selects first element named "a"

36. Select the 4th element of the list you made earlier using `$` notation.
37. Select the 1st element of your list using `[ ]` notation.

Ex: `mylist[1]` selects first element named "a", you get a list with one element.

Ex: `mylist["a"]` selects first element named "a", you get a list with one element.

38. select the 4th element of your list using '' notation.
39. Make a factor using factor(), with 5 elements. Ex: `fa=factor(c("a","a","b"))`
40. Convert a character vector to factor using `as.factor()`. First, make a character vector using `c()` then use `as.factor()`.
41. Convert the factor you made above to character using `as.character()`.

2.10.3 Reading in and writing data out in R

42. Read CpG island (CpGi) data from the compGenomRData package `CpGi.table.hg18.txt`, this is a tab-separated file, store it in a variable called "cpgi". Use `cpgtFilePath=system.file("extdata", "CpGi.table.hg18.txt", package="compGenomRData")` to get the file path within the installed `compGenomRData` package.
43. Use `head()` on CpGi to see first few rows.
44. Why doesn't the following work? see 'sep' argument at `help(read.table)`.

```
cpgtFilePath=system.file("extdata",
                "CpGi.table.hg18.txt",
                package="compGenomRData")
cpgtFilePath
```

```
## [1] "/Users/aakalin/Rlibs/compGenomRData/extdata/CpGi.table.hg18.txt"
```

```
cpgiSepComma=read.table(cpgtFilePath,header=TRUE,sep=",")
head(cpgiSepComma)
```

```
##   chrom.chromStart.chromEnd.name.length.cpgNum.gcNum.perCpg.perGc.obsExp
## 1         chr1\t18598\t19673\tCpG: 116\t1075\t116\t787\t21.6\t73.2\t0.83
## 2          chr1\t124987\t125426\tCpG: 30\t439\t30\t295\t13.7\t67.2\t0.64
## 3          chr1\t317653\t318092\tCpG: 29\t439\t29\t295\t13.2\t67.2\t0.62
## 4         chr1\t427014\t428027\tCpG: 84\t1013\t84\t734\t16.6\t72.5\t0.64
## 5         chr1\t439136\t440407\tCpG: 99\t1271\t99\t777\t15.6\t61.1\t0.84
## 6            chr1\t523082\t523977\tCpG: 94\t895\t94\t570\t21\t63.7\t1.04
```

43. What happens when `stringsAsFactors=FALSE` ? `cpgiHF=read.table("intro2R_data/data/CpGi.table.hg18.txt",header=FALSE,sep="\t", stringsAsFactors=FALSE)`
44. Read only first 10 rows of the CpGi table.

45. Use `cpgtFilePath=system.file("extdata","CpGi.table.hg18.txt",package="compGenomRData")` to get the file path, then use `read.table()` with argument `header=FALSE`. Use `head()` to see the results.

46. Write CpG islands to a text file called "my.cpgi.file.txt". Write the file to your home folder, you can use `file="~/my.cpgi.file.txt"` in linux. `~/` denotes home folder.

47. Same as above but this time make sure use `quote=FALSE,sep="\t"` and `row.names=FALSE` arguments. Save the file to "my.cpgi.file2.txt" and compare it with "my.cpgi.file.txt"

48. Write out the first 10 rows of 'cpgi' data frame. HINT: use subsetting for data frames we learned before.

49. Write the first 3 columns of 'cpgi' data frame.

50. Write CpG islands only on chr1. HINT: use subsetting with `[]`, feed a logical vector using `==` operator.

51. Read two other data sets "rn4.refseq.bed" and "rn4.refseq2name.txt" with `header=FALSE`, assign them to df1 and df2 respectively. They are again included in the compGenomRData package, and you can use `system.file()` function to get the file paths.

52. Use `head()` to see what is inside of the the data frames above.

53. Merge data sets using `merge()` and assign the results to variable named 'new.df', and use `head()` to see the results.

2.10.4 Plotting in R

Please run the following for the rest of the exercises.

```
set.seed(1001)
x1=1:100+rnorm(100,mean=0,sd=15)
y1=1:100
```

44. Make a scatter plot using `x1` and `y1` vectors generated above.

45. Use `main` argument to give a title to `plot()` as in `plot(x,y,main="title")`
46. Use `xlab` argument to set a label to x-axis.Use `ylab` argument to set a label to y-axis.
47. See what `mtext(side=3,text="hi there")` does. HINT: mtext stands for margin text.
48. See what `mtext(side=2,text="hi there")` does.check your plot after execution.
49. You can use `paste()` as 'text' argument in mtext() try that, you need to re-plot. your plot first. HINT: `mtext(side=3,text=paste(...))` See what `paste()` is used for.

```
paste("Text","here")
```

```
## [1] "Text here"
```

```
myText=paste("Text","here")
myText
```

```
## [1] "Text here"
```

Use mtext() and paste() to put a margin text on the plot.

44. `cor()` calculates correlation between two vectors. Pearson correlation is a measure of the linear correlation (dependence) between two variables X and Y.

```
corxy=cor(x1,y1) # calculates pearson correlation
```

44. Try use `mtext()`,`cor()` and `paste()` to display correlation coefficient on your scatterplot ?

45. Change the colors of your plot using `col` argument. Ex: `plot(x,y,col="red")`
46. Use `pch=19` as an argument in your `plot()` command.
47. Use `pch=18` as an argument to your `plot()` command.
48. Make histogram of x1 with `hist()` function.Histogram is a graphical representation of the data distribution.
49. You can change colors with 'col', add labels with 'xlab', 'ylab', and add a 'title' with 'main' arguments. Try all these in a histogram.
50. Make boxplot of y1 with `boxplot()`.
51. Make boxplots of `x1` and `y1` vectors in the same plot.
52. In boxplot use horizontal = TRUE argument
53. make multiple plots with par(mfrow=c(2,1))
    - run par(mfrow=c(2,1))
    - make a boxplot
    - make a histogram
54. Do the same as above but this time with par(mfrow=c(1,2))
55. Save your plot using "Export" button in Rstudio
56. Save your plot by running :

```
dev.copy(pdf,file="plot.file.pdf");dev.off()
```

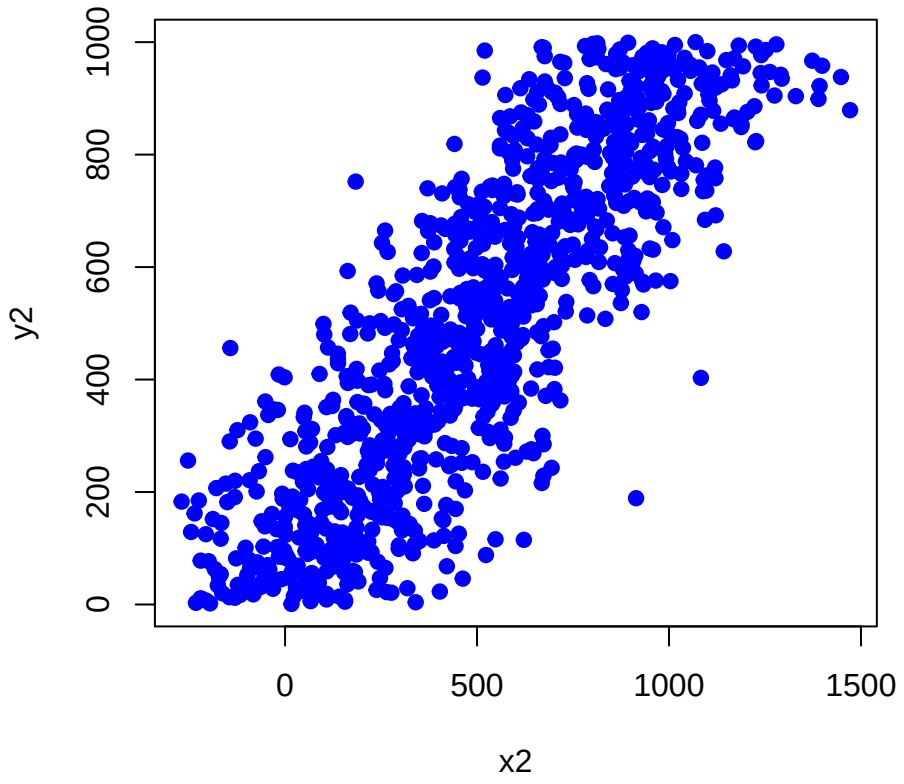
44. Save your plot running :

```
dev.copy(png,filename="plot.file.png");dev.off()
```

44. Another way to save the plot is the following
45. Open a graphics device
46. Create the plot

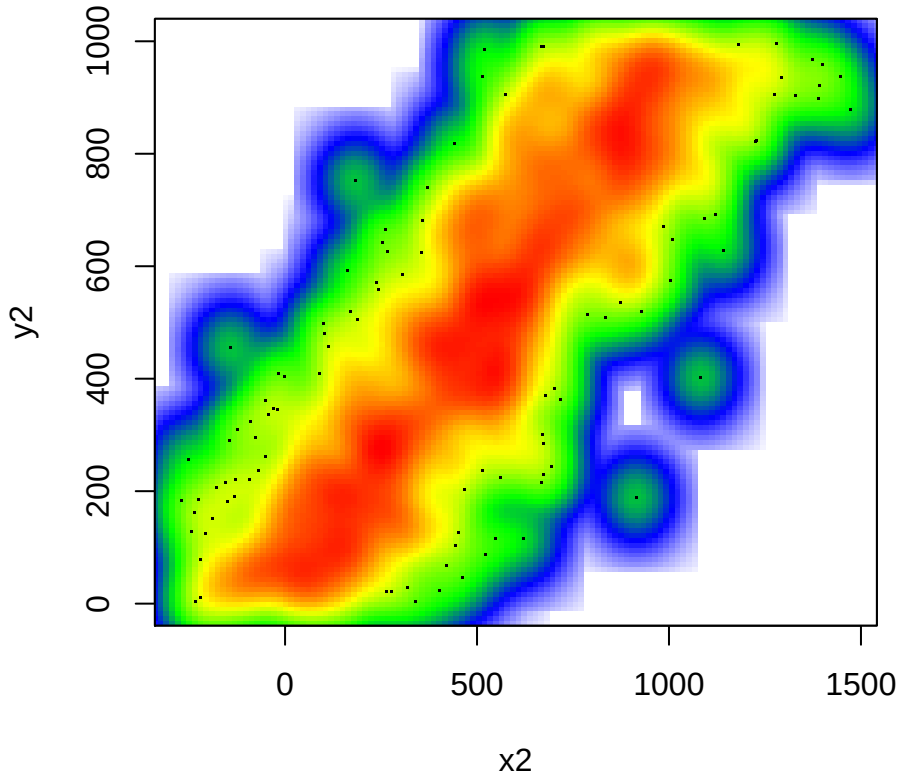
47. Close the graphics device
48. EXTRA: Making color density scatterplot. You can make a scatter plot showing density of points rather than points themselves. If you use points it looks like this:

```
x2=1:1000+rnorm(1000,mean=0,sd=200)
y2=1:1000
plot(x2,y2,pch=19,col="blue")
```



If you use smoothScatter() function, you get the densities.

```
smoothScatter(x2,y2,colramp=colorRampPalette(c("white","blue", "green","yellow","red")))
```



Now, plot with `colramp=heat.colors` argument and then use a custom color scale using the following argument.

```
colramp = colorRampPalette(c("white","blue", "green","yellow","red")))
```

2.10.5 Functions and control structures (for, if/else etc.)

Read CpG island data as shown below for the rest of the exercises.

```
cpgtFilePath=system.file("extdata",
                "CpGi.table.hg18.txt",
                package="compGenomRData")
cpgi=read.table(cpgtFilePath,header=TRUE,sep="\t")
head(cpgi)
```

```
##   chrom chromStart chromEnd     name length cpgNum
```

```
## 1  chr1      18598    19673 CpG: 116   1075    116
## 2  chr1     124987   125426  CpG: 30    439     30
## 3  chr1     317653   318092  CpG: 29    439     29
## 4  chr1     427014   428027  CpG: 84   1013     84
## 5  chr1     439136   440407  CpG: 99   1271     99
## 6  chr1     523082   523977  CpG: 94    895     94
##    gcNum perCpg perGc obsExp
## 1    787   21.6  73.2   0.83
## 2    295   13.7  67.2   0.64
## 3    295   13.2  67.2   0.62
## 4    734   16.6  72.5   0.64
## 5    777   15.6  61.1   0.84
## 6    570   21.0  63.7   1.04
```

44. Check values at perGc column using a histogram. 'perGc' stands for GC percent => percentage of C+G nucleotides
45. Make a boxplot for 'perGc' column
46. Use if/else structure to decide if given GC percent high, low or medium. If it is low, high, or medium. low < 60, high>75, medium is between 60 and 75 use greater or less than operators < or > . Fill in the values in the in code below, where it is written 'YOU_FILL_IN'

```
GCper=65

  # check if GC value is lower than 60,
  # assign "low" to result
  if('YOU_FILL_IN'){
    result="low"
    cat("low")
  }
  else if('YOU_FILL_IN'){  # check if GC value is higher than 75, assign "high" to result
    result="high"
    cat("high")
  }else{ # if those two conditions fail then it must be "medium"
```

```
    result="medium"
  }

result
```

44. Write a function that takes a value of GC percent and decides if it is low, high, or medium. low < 60, high>75, medium is between 60 and 75. Fill in the values in the in code below, where it is written 'YOU_FILL_IN'

```
GCclass<-function(my.gc){

  YOU_FILL_IN

  return(result)
}
GCclass(10) # should return "low"
GCclass(90) # should return "high"
GCclass(65) # should return "medium"
```

44. Use a for loop to get GC percentage classes for gcValues below. Use the function you wrote above.

```
gcValues=c(10,50,70,65,90)
for( i in YOU_FILL_IN){
  YOU_FILL_IN
}
```

44. Use `lapply` to get to get GC percentage classes for gcValues. Example:

```
vec=c(1,2,4,5)
power2=function(x){ return(x^2)  }
    lapply(vec,power2)
```

44. Use sapply to get values to get GC percentage classes for gcValues

45. Is there a way to decide on the GC percentage class of given vector of GCpercentages without using if/else structure and loops ? if so, how can you do it? HINT: subsetting using < and > operators

3

Statistics and Exploratory Data Analysis for Genomics

This chapter will summarize statistics and exploratory data analysis methods frequently used in computational genomics. As these fields are continuously evolving, the techniques introduced here do not form an exhaustive list but mostly corner stone methods that are often and still being used. In addition, we focused on giving intuitive and practical understanding of the methods with relevant examples from the field.
If you want to dig deeper into statistics and math, beyond what is described here, we included appropriate references with annotation after each major section.

3.1 How to summarize collection of data points: The idea behind statistical distributions

In biology and many other fields data is collected via experimentation. The nature of the experiments and natural variation in biology makes it impossible to get the same exact measurements every time you measure something. For example, if you are measuring gene expression values for a certain gene, say PAX6, and let's assume you are measuring expression per sample and cell with any method(microarrays, rt-qPCR, etc.). You will not get the same expression value even if your samples are homogeneous. Due to technical bias in experiments or natural variation in the samples. Instead, we would like to describe this collection of data some other way that represents the general properties of the data. The figure shows a sample of 20 expression values from PAX6 gene.

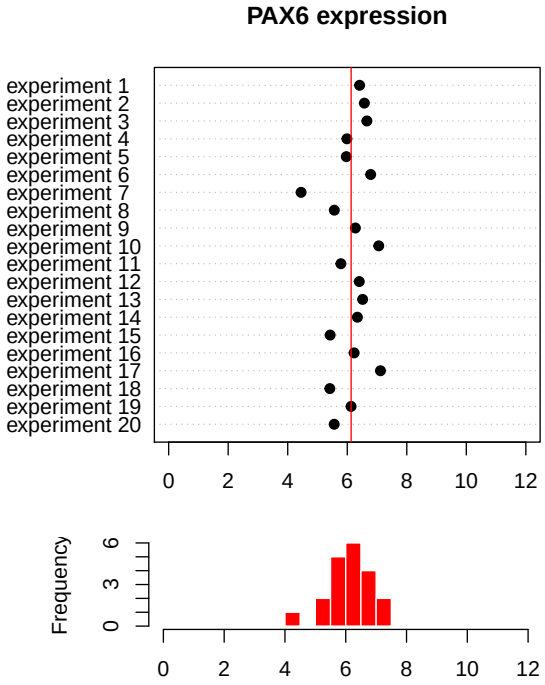


FIGURE 3.1: Expression of PAX6 gene in 20 replicate experiments

3.1.1 Describing the central tendency: mean and median

As seen in the figure above, the points from this sample are distributed around a central value and the histogram below the dot plot shows number of points in each bin. Another observation is that there are some bins that have more points than others. If we want to summarize what we observe, we can try to represent the collection of data points with an expression value that is typical to get, something that represents the general tendency we observe on the dot plot and the histogram. This value is sometimes called central value or central tendency, and there are different ways to calculate such a value. In the figure above, we see that all the values are spread around 6.13 (red line), and that is indeed what we call mean value of this sample of expression values. It can be calculated with the following formula $\overline{X} = \sum_{i=1}^{n} x_i/n$, where x_i is the expression value of an experiment and n is the number of expression value obtained from the experiments. In R, `mean()` function will calculate the mean of a provided vector of numbers. This is called a "sample mean". In reality the possible values of

PAX6 expression for all cells (provided each cell is of the identical cell type and is in identical conditions) are much much more than 20. If we had the time and the funding to sample all cells and measure PAX6 expression we would get a collection values that would be called, in statistics, a "population". In our case the population will look like the left hand side of the figure below. What we have done with our 20 data points is that we took a sample of PAX6 expression values from this population, and calculated the sample mean.

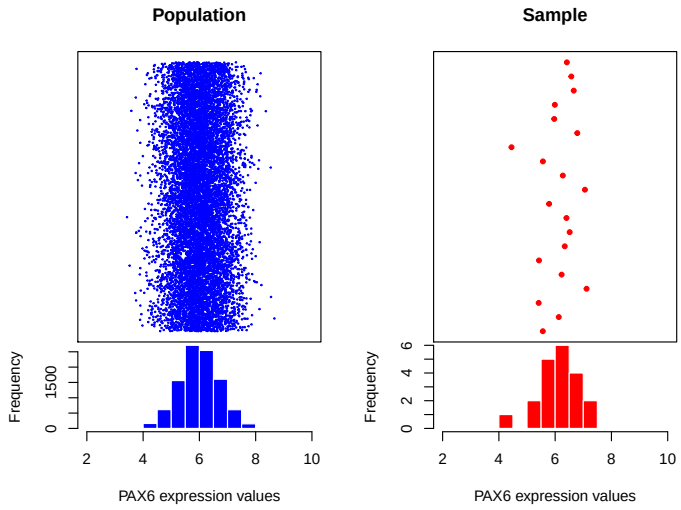


FIGURE 3.2: Expression of all possible PAX6 gene expressions measures on all available biological samples (left). Expression of PAX6 gene from statistical sample, a random subset, from the population of biological samples (Right).

The mean of the population is calculated the same way but traditionally Greek letter μ is used to denote the population mean. Normally, we would not have access to the population and we will use sample mean and other quantities derived from the sample to estimate the population properties. This is the basic idea behind statistical inference which we will see this in action in later sections as well. We estimate the population parameters from the sample parameters and there is some uncertainty associated with those estimates. We will be trying to assess those uncertainties and make decisions in the presence of those uncertainties.

We are not yet done with measuring central tendency. There are other

ways to describe it, such as the median value. Mean can be affected by outliers easily. If certain values are very high or low from the bulk of the sample this will shift mean towards those outliers. However, median is not affected by outliers. It is simply the value in a distribution where half of the values are above and the other half is below. In R, `median()` function will calculate the mean of a provided vector of numbers.

Let's create a set of random numbers and calculate their mean and median using R.

```
#create 10 random numbers from uniform distribution
x=runif(10)
# calculate mean
mean(x)
```

```
## [1] 0.3739
```

```
# calculate median
median(x)
```

```
## [1] 0.3278
```

3.1.2 Describing the spread: measurements of variation

Another useful way to summarize a collection of data points is to measure how variable the values are. You can simply describe the range of the values , such as minimum and maximum values. You can easily do that in R with `range()` function. A more common way to calculate variation is by calculating something called "standard deviation" or the related quantity called "variance". This is a quantity that shows how variable the values are, a value around zero indicates there is not much variation in the values of the data points, and a high value indicates high variation in the values. The variance is the squared distance of data points from the mean. Population variance is again a quantity we usually do not have access to and is simply calculate as follows $\sigma^2 = \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{n}$, where μ is the population mean, x_i is the ith data point in the population and n is the population size. However, when the we have only access to a sample this formulation

is biased. It means that it underestimates the population variance, so we make a small adjustment when we calculate the sample variance, denoted as s^2:

$$s^2 = \sum_{i=1}^{n} \frac{(x_i - \overline{X})^2}{n-1}$$ where x_i is the ith data point and $\overline{X}$ is the sample mean.

The sample standard deviation is simply the square-root of the sample variance. The good thing about standard deviation is that it has the same unit as the mean so it is more intuitive.

$$s = \sqrt{\sum_{i=1}^{n} \frac{(x_i - \overline{X})^2}{n-1}}$$

We can calculate sample standard deviation and variation with `sd()` and `var()` functions in R. These functions take vector of numeric values as input and calculate the desired quantities. Below we use those functions on a randomly generated vector of numbers.

```
x=rnorm(20,mean=6,sd=0.7)
var(x)
```

```
## [1] 0.2531
```

```
sd(x)
```

```
## [1] 0.5031
```

One potential problem with the variance is that it could be affected by outliers. The points that are too far away from the mean will have a large affect on the variance even though there might be few of them. A way to measure variance that could be less affected by outliers is looking at where bulk of the distribution is. How do we define where the bulk is? One common way is to look at the the difference between 75th percentile and 25th percentile, this effectively removes a lot of potential outliers which will be towards the edges of the range of values. This is called interquartile range , and can be easily calculated using R via `IQR()` function and the quantiles of a vector is calculated with `quantile()` function.

Let us plot the boxplot for a random vector and also calculate IQR using R. In the boxplot below, 25th and 75th percentiles are the edges of the box, and the median is marked with a thick line going through roughly middle the box.

```
x=rnorm(20,mean=6,sd=0.7)
IQR(x)
```

```
## [1] 0.5011
```

```
quantile(x)
```

```
##    0%   25%   50%   75%  100%
## 5.437 5.743 5.860 6.244 6.558
```

```
boxplot(x,horizontal = T)
```

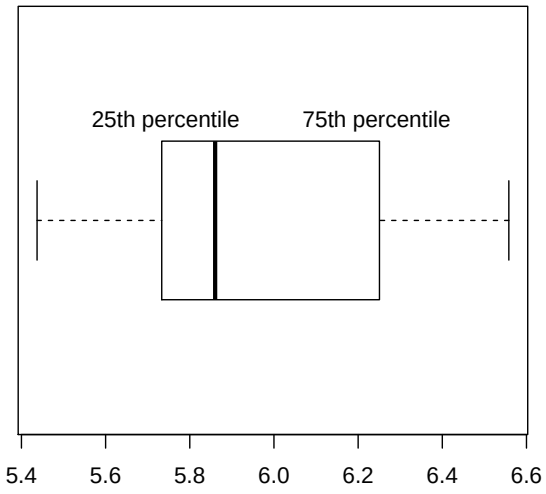


FIGURE 3.3: Boxplot showing 25th percentile and 75th percentile and median for a set of points sample from a normal distribution with mean=6 and standard deviation=0.7

3.1.2.1 Frequently used statistical distributions

The distributions have parameters (such as mean and variance) that summarizes them but also they are functions that assigns each outcome of a statistical experiment to its probability of occurrence. One distribution that you will frequently encounter is the normal distribution or Gaussian distribution. The normal distribution has a typical "bell-curve" shape and, characterized by mean and standard deviation. A set of data points that follow normal distribution mostly will be close to the mean but spread around it controlled by the standard deviation parameter. That means if we sample data points from a normal distribution we are more likely to sample nearby the mean and sometimes away from the mean. Probability of an event occurring is higher if it is nearby the mean. The effect of the parameters for normal distribution can be observed in the following plot.

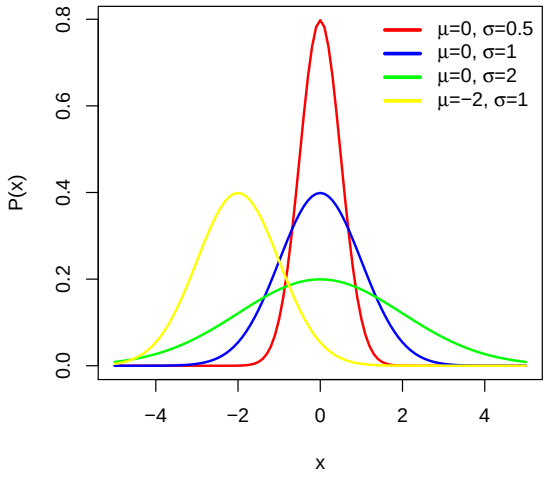


FIGURE 3.4: Different parameters for normal distribution and effect of those on the shape of the distribution

The normal distribution is often denoted by $\mathcal{N}(\mu,\ \sigma^2)$ When a random variable X is distributed normally with mean μ and variance σ^2, we write:

$$X\ \sim\ \mathcal{N}(\mu,\ \sigma^2).$$

The probability density function of Normal distribution with mean μ and standard deviation σ is as follows

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The probability density function gives the probability of observing a value on a normal distribution defined by μ and σ parameters.

Often times, we do not need the exact probability of a value but we need the probability of observing a value larger or smaller than a critical value or reference point. For example, we might want to know the probability of X being smaller than or equal to -2 for a normal distribution with mean 0 and standard deviation 2. ,$P(X <= -2 \mid \mu = 0, \sigma = 2)$. In this case, what we want is the are under the curve shaded in blue. To be able to that we need to integrate the probability density function but we will usually let software do that. Traditionally, one calculates a Z-score which is simply $(X - \mu)/\sigma = (-2 - 0)/2 = -1$, and corresponds to how many standard deviations you are away from the mean. This is also called "standardization", the corresponding value is distributed in "standard normal distribution" where $\mathcal{N}(0, 1)$.

After calculating the Z-score, we can go look up in a table, that contains the area under the curve for the left and right side of the Z-score, but again we use software for that tables are outdated.

Below we are showing the Z-score and the associated probabilities derived from the calculation above for $P(X <= -2 \mid \mu = 0, \sigma = 2)$.

In R, family of `*norm` functions (`rnorm`,`dnorm`,`qnorm` and `pnorm`) can be used to operate with normal distribution, such as calculating probabilities and generating random numbers drawn from normal distribution.

```
# get the value of probability density function when X= -2,
# where mean=0 and sd=2
dnorm(-2, mean=0, sd=2)
```

```
## [1] 0.121
```

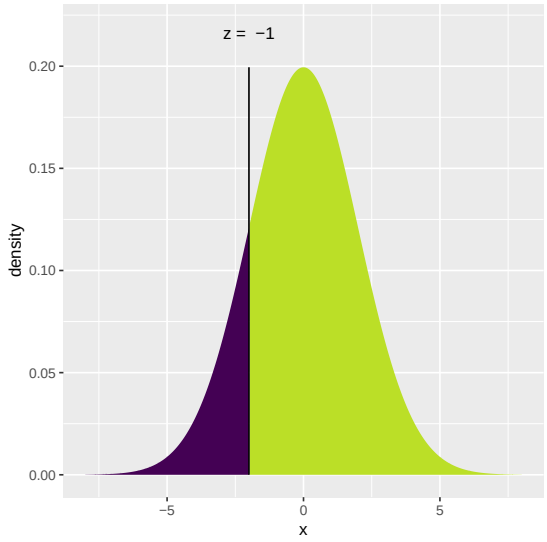


FIGURE 3.5: Z-score and associated probabilities for Z= -1

```
# get the probability of P(X =< -2) where mean=0 and sd=2
pnorm(-2, mean=0, sd=2)
```

```
## [1] 0.1587
```

```
# get the probability of P(X > -2) where mean=0 and sd=2
pnorm(-2, mean=0, sd=2,lower.tail = FALSE)
```

```
## [1] 0.8413
```

```
# get 5 random numbers from normal dist with  mean=0 and sd=2
rnorm(5, mean=0 , sd=2)
```

```
## [1] -1.8109 -1.9221 -0.5147  0.8217 -0.7901
```

```
# get y value corresponding to P(X > y) = 0.15 with  mean=0 and sd=2
qnorm( 0.15, mean=0 , sd=2)
```

```
## [1] -2.073
```

There are many other distribution functions in R that can be used the same way. You have to enter the distribution specific parameters along with your critical value, quantiles or number of random numbers depending on which function you are using in the family.We will list some of those functions below.

- `dbinom` is for binomial distribution. This distribution is usually used to model fractional data and binary data. Examples from genomics includes methylation data.
- `dpois` is used for Poisson distribution and `dnbinom` is used for negative binomial distribution. These distributions are used to model count data such as sequencing read counts.
- `df` (F distribution) and `dchisq` (Chi-Squared distribution) are used in relation to distribution of variation. F distribution is used to model ratios of variation and Chi-Squared distribution is used to model distribution of variations. You will frequently encounter these in linear models and generalized linear models.

3.1.3 Precision of estimates: Confidence intervals

When we take a random sample from a population and compute a statistic, such as the mean, we are trying to approximate the mean of the population. How well this sample statistic estimates the population value will always be a concern. A confidence interval addresses this concern because it provides a range of values which is plausible to contain the population parameter of interest. Normally, we would not have access to a population. If we did, we would not have to estimate the population parameters and its precision.

When we do not have access to the population, one way to estimate intervals is to repeatedly take samples from the original sample with replacement, that is we take a data point from the sample we replace, and we take another data point until we have sample size of the original sample. Then, we calculate the parameter of interest, in this case mean, and repeat this step a large number of times, such as 1000. At this point, we would have a distribution of re-sampled means, we can then calculate the 2.5th and 97.5th percentiles and these will be our so-called 95% confidence interval.

This procedure, resampling with replacement to estimate the precision of population parameter estimates, is known as the bootstrap.

Let's see how we can do this in practice. We simulate a sample coming from a normal distribution (but we pretend we don't know the population parameters). We will try to estimate the precision of the mean of the sample using bootstrap to build confidence intervals.

```
set.seed(21)
sample1= rnorm(50,20,5) # simulate a sample

# do bootstrap resampling, sampling with replacement
boot.means=do(1000) * mean(resample(sample1))

# get percentiles from the bootstrap means
q=quantile(boot.means[,1],p=c(0.025,0.975))

# plot the histogram
hist(boot.means[,1],col="cornflowerblue",border="white",
                    xlab="sample means")
abline(v=c(q[1], q[2] ),col="red")
text(x=q[1],y=200,round(q[1],3),adj=c(1,0))
text(x=q[2],y=200,round(q[2],3),adj=c(0,0))
```

If we had a convenient mathematical method to calculate confidence interval we could also do without resampling methods. It turns out that if we take repeated samples from a population of with sample size n, the distribution of means ($\overline{X}$) of those samples will be approximately normal with mean μ and standard deviation $\sigma/\sqrt{n}$. This is also known as Central Limit Theorem(CLT) and is one of the most important theorems in statistics. This also means that $\frac{\overline{X}-\mu}{\sigma\sqrt{n}}$ has a standard normal distribution and we can calculate the Z-score and then we can get the percentiles associated with the Z-score. Below, we are showing the Z-score calculation for the distribution of $\overline{X}$, and then we are deriving the confidence intervals starting with the fact that probability of Z being between -1.96 and 1.96 is 0.95. We then use algebra to show that the probability that unknown μ is

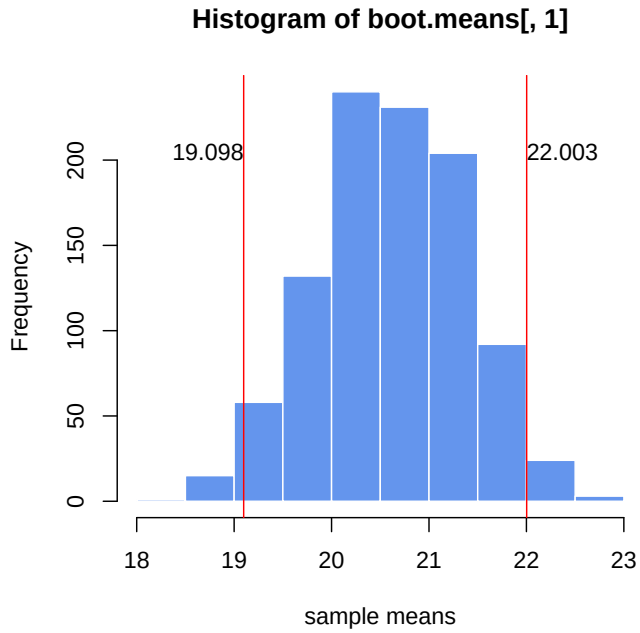


FIGURE 3.6: Precision estimate of the sample mean using 1000 bootstrap samples. Confidence intervals derived from the bootstrap samples are shown with red lines.

captured between $\overline{X} - 1.96\sigma/\sqrt{n}$ and $\overline{X} + 1.96\sigma/\sqrt{n}$ is 0.95, which is commonly known as 95% confidence interval.

$$
\begin{aligned}
Z &= \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \\
P(-1.96 < Z < 1.96) &= 0.95 \\
P(-1.96 < \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} < 1.96) &= 0.95 \\
P(\mu - 1.96\sigma/\sqrt{n} < \overline{X} < \mu + 1.96\sigma/\sqrt{n}) &= 0.95 \\
P(\overline{X} - 1.96\sigma/\sqrt{n} < \mu < \overline{X} + 1.96\sigma/\sqrt{n}) &= 0.95 \\
confint &= [\overline{X} - 1.96\sigma/\sqrt{n}, \overline{X} + 1.96\sigma/\sqrt{n}]
\end{aligned}
$$

A 95% confidence interval for population mean is the most common common interval to use, and would mean that we would expect 95% of the interval estimates to include the population parameter, in this case the mean. However, we can pick any value such as 99% or 90%. We can generalize the confidence interval for $100(1-\alpha)$ as follows:

$$\overline{X} \pm Z_{\alpha/2}\sigma/\sqrt{n}$$

In R, we can do this using `qnorm()` function to get Z-scores associated with $\alpha/2$ and $1-\alpha/2$. As you can see, the confidence intervals we calculated using CLT are very similar to the ones we got from bootstrap for the same sample. For bootstrap we got $[19.21, 21.989]$ and for the CLT based estimate we got $[19.23638, 22.00819]$.

```
alpha=0.05
sd=5
n=50
mean(sample1)+qnorm(c(alpha/2,1-alpha/2))*sd/sqrt(n)
```

```
## [1] 19.24 22.01
```

The good thing about CLT as long as the sample size is large regardless of the population distribution, the distribution of sample means drawn from that population will always be normal. Here we are repeatedly drawing samples 1000 times with sample size n=10,30, and 100 from a bimodal, exponential and a uniform distribution and we are getting sample mean distributions following normal distribution.

However, we should note that how we constructed the confidence interval using standard normal distribution, $N(0,1)$, only works when the when we know the population standard deviation. In reality, we usually have only access to a sample and have no idea about the population standard deviation. If this is the case we should use estimate the standard deviation using sample standard deviation and use something called t distribution instead of standard normal distribution in our interval calculation. Our confidence interval becomes $\overline{X} \pm t_{\alpha/2}s/\sqrt{n}$, with t distribution parameter $d.f = n-1$, since now the following quantity is t distributed $\frac{\overline{X}-\mu}{s/\sqrt{n}}$ instead of standard normal distribution.

The t distribution is similar to standard normal distribution has mean 0 but its spread is larger than the normal distribution especially when sample size is small, and has one parameter v for the degrees of freedom, which is $n-1$ in this case. Degrees of freedom is simply number of data points minus number of parameters estimated. Here we are estimating

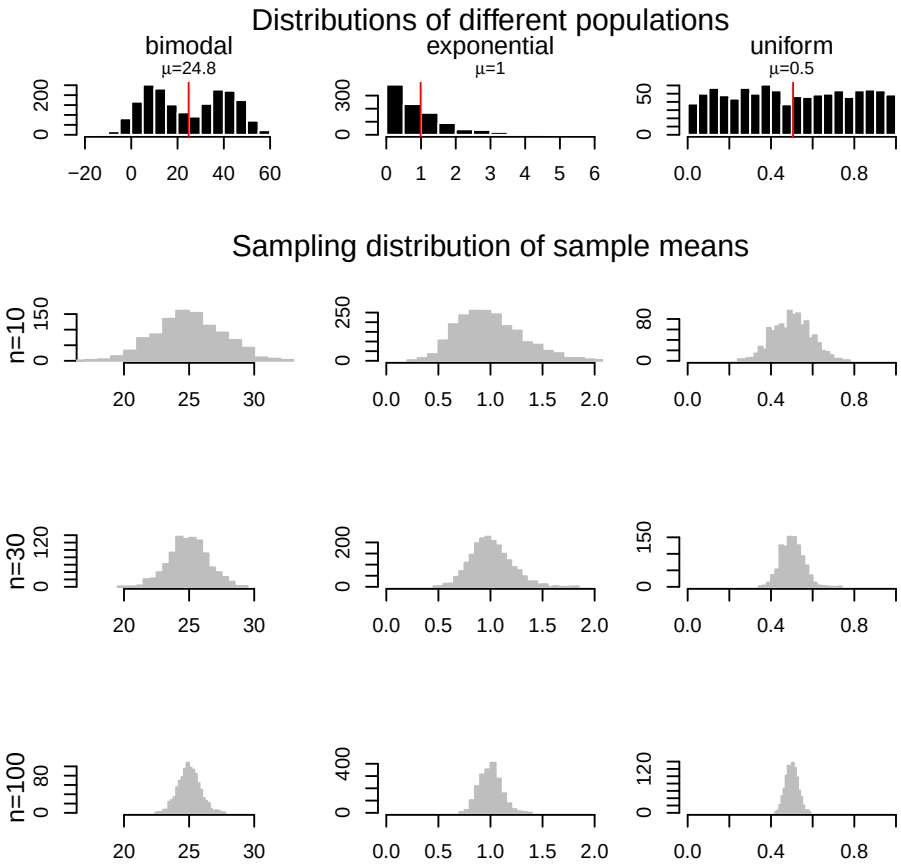


FIGURE 3.7: Sample means are normally distributed regardless of the population distribution they are drawn from.

the mean from the data and the distribution is for the means, therefore degrees of freedom is $n - 1$.

3.2 How to test for differences between samples

Often times we would want to compare sets of samples. Such comparisons include if wild-type samples have different expression compared to mutants or if healthy samples are different from disease samples in some

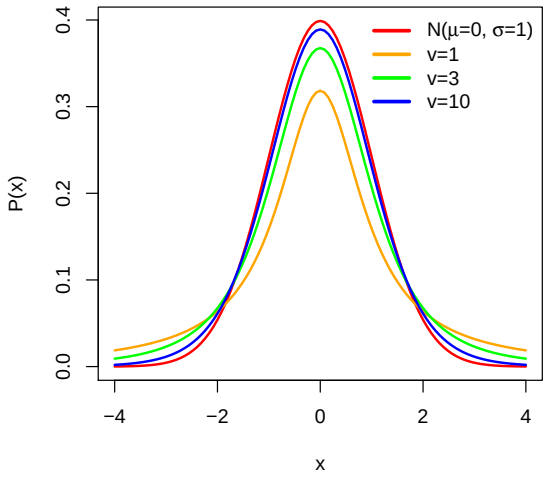


FIGURE 3.8: Normal distribution and t distribution with different degrees of freedom. With increasing degrees of freedom, t distribution approximates the normal distribution better.

measurable feature (blood count, gene expression, methylation of certain loci). Since there is variability in our measurements, we need to take that into account when comparing the sets of samples. We can simply subtract the means of two samples, but given the variability of sampling, at the very least we need to decide a cutoff value for differences of means, small differences of means can be explained by random chance due to sampling. That means we need to compare the difference we get to a value that is typical to get if the difference between two group means were only due to sampling. If you followed the logic above, here we actually introduced two core ideas of something called "hypothesis testing", this is simply using statistics to determine the probability that a given hypothesis (if two sample sets are from the same population or not) is true. Formally, those two core ideas are as follows:

1. Decide on a hypothesis to test, often called "null hypothesis" (H_0). In our case, the hypothesis is there is no difference between sets of samples. An the "Alternative hypothesis" (H_1) is there is a difference between the samples.
2. Decide on a statistic to test the truth of the null hypothesis.
3. Calculate the statistic

4. Compare it to a reference value to establish significance, the P-value. Based on that either reject or not reject the null hypothesis, H_0

3.2.1 randomization based testing for difference of the means

There is one intuitive way to go about this. If we believe there are no differences between samples that means the sample labels (test-control or healthy-disease) has no meaning. So, if we randomly assign labels to the samples that and calculate the difference of the mean, this creates a null distribution for the H_0 where we can compare the real difference and measure how unlikely it is to get such a value under the expectation of the null hypothesis. We can calculate all possible permutations to calculate the null distribution. However, sometimes that is not very feasible and equivalent approach would be generating the null distribution by taking a smaller number of random samples with shuffled group membership.

Below, we are doing this process in R. We are first simulating two samples from two different distributions. These would be equivalent to gene expression measurements obtained under different conditions. Then, we calculate the differences in the means and do the randomization procedure to get a null distribution when we assume there is no difference between samples, H_0. We then calculate how often we would get the original difference we calculated under the assumption that H_0 is true.

```
set.seed(100)
gene1=rnorm(30,mean=4,sd=2)
gene2=rnorm(30,mean=2,sd=2)
org.diff=mean(gene1)-mean(gene2)
gene.df=data.frame(exp=c(gene1,gene2),
                   group=c( rep("test",30),rep("control",30) ) )


exp.null <- do(1000) * diff(mosaic::mean(exp ~ shuffle(group), data=gene.df))
hist(exp.null[,1],xlab="null distribution | no difference in samples",
     main=expression(paste(H[0]," :no difference in means") ),
```

```
     xlim=c(-2,2),col="cornflowerblue",border="white")
abline(v=quantile(exp.null[,1],0.95),col="red" )
abline(v=org.diff,col="blue" )
text(x=quantile(exp.null[,1],0.95),y=200,"0.05",adj=c(1,0),col="red")
text(x=org.diff,y=200,"org. diff.",adj=c(1,0),col="blue")
```

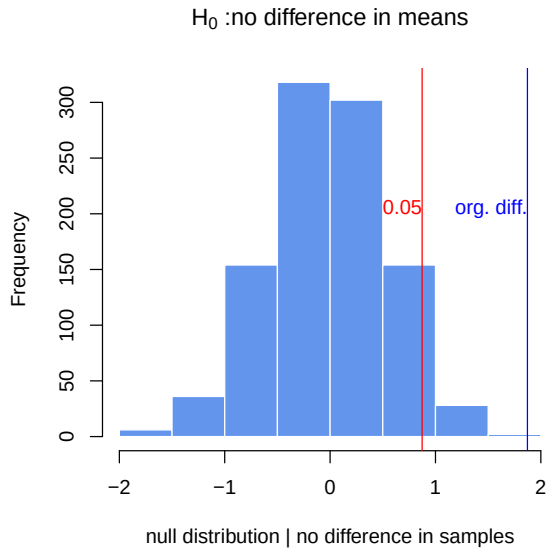


FIGURE 3.9: The null distribution for differences of means obtained via randomization. The original difference is marked via blue line. The red line marks the value that corresponds to P-value of 0.05

```
p.val=sum(exp.null[,1]>org.diff)/length(exp.null[,1])
p.val
```

```
## [1] 0
```

After doing random permutations and getting a null distribution, it is possible to get a confidence interval for the distribution of difference in means. This is simply the 2.5th and 97.5th percentiles of the null distribution, and directly related to the P-value calculation above.

3.2.2 Using t-test for difference of the means between two samples

We can also calculate the difference between means using a t-test. Sometimes we will have too few data points in a sample to do meaningful randomization test, also randomization takes more time than doing a t-test. This is a test that depends on the t distribution. The line of thought follows from the CLT and we can show differences in means are t distributed. There are couple of variants of the t-test for this purpose. If we assume the variances are equal we can use the following version

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_{X_1 X_2} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

where

$$s_{X_1 X_2} = \sqrt{\frac{(n_1 - 1)s_{X_1}^2 + (n_2 - 1)s_{X_2}^2}{n_1 + n_2 - 2}}$$

In the first equation above the quantity is t distributed with $n_1 + n_2 - 2$ degrees of freedom. We can calculate the quantity then use software to look for the percentile of that value in that t distribution, which is our P-value. When we can not assume equal variances we use "Welch's t-test" which is the default t-test in R and also works well when variances and the sample sizes are the same. For this test we calculate the following quantity:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{s_{\overline{X}_1 - \overline{X}_2}}$$

where

$$s_{\overline{X}_1 - \overline{X}_2} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

and the degrees of freedom equals to:

$$\text{d.f.} = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{(s_1^2/n_1)^2/(n_1 - 1) + (s_2^2/n_2)^2/(n_2 - 1)}$$

Luckily, R does all those calculations for us. Below we will show the use of `t.test()` function in R. We will use it on the samples we simulated above.

```
# Welch's t-test
stats::t.test(gene1,gene2)
```

```
##
##  Welch Two Sample t-test
##
## data:  gene1 and gene2
## t = 3.8, df = 48, p-value = 5e-04
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.8724 2.8728
## sample estimates:
## mean of x mean of y
##     4.058     2.185
```

```
# t-test with equal varience assumption
stats::t.test(gene1,gene2,var.equal=TRUE)
```

```
##
##  Two Sample t-test
##
## data:  gene1 and gene2
## t = 3.8, df = 58, p-value = 4e-04
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.8771 2.8681
## sample estimates:
## mean of x mean of y
##     4.058     2.185
```

A final word on t-tests: they generally assume population where samples coming from have normal distribution, however it is been shown t-test can tolerate deviations from normality. Especially, when two distributions are moderately skewed in the same direction. This is due to central limit theorem which says means of samples will be distributed normally no matter the population distribution if sample sizes are large.

3.2.3 multiple testing correction

We should think of hypothesis testing as a non-error-free method of making decisions. There will be times when we declare something significant and accept H_1 but we will be wrong. These decisions are also called "false positives" or "false discoveries", this is also known as "type I error". Similarly, we can fail to reject a hypothesis when we actually should. These cases are known as "false negatives", also known as "type II error".

The ratio of true negatives to the sum of true negatives and false positives ($\frac{TN}{FP+TN}$) is known as specificity. And we usually want to decrease the FP and get higher specificity. The ratio of true positives to the sum of true positives and false negatives ($\frac{TP}{TP+FN}$) is known as sensitivity. And, again we usually want to decrease the FN and get higher sensitivity. Sensitivity is also known as "power of a test" in the context of hypothesis testing. More powerful tests will be highly sensitive and will do less type II errors. For the t-test the power is positively associated with sample size and the effect size. Higher the sample size, smaller the standard error and looking for the larger effect sizes will similarly increase the power.

The general summary of these the different combination of the decisions are included in the table below.

	H_0 is TRUE, [Gene is NOT differentially expressed]	H_1 is TRUE, [Gene is differentially expressed]	
Accept H_0 (claim that the gene is not differentially expressed)	True Negatives (TN)	False Negatives (FN) ,type II error	m_0: number of truly null hypotheses

	H_0 is TRUE, [Gene is NOT differentially expressed]	H_1 is TRUE, [Gene is differentially expressed]	
reject H_0 (claim that the gene is differentially expressed)	False Positives (FP) ,type I error	True Positives (TP)	$m - m_0$: number of truly alternative hypotheses

We expect to make more type I errors as the number of tests increase, that means we will reject the null hypothesis by mistake. For example, if we perform a test the 5% significance level, there is a 5% chance of incorrectly rejecting the null hypothesis if the null hypothesis is true. However, if we make 1000 tests where all null hypotheses are true for each of them, the average number of incorrect rejections is 50. And if we apply the rules of probability, there are is almost a 100% chance that we will have at least one incorrect rejection. There are multiple statistical techniques to prevent this from happening. These techniques generally shrink the P-values obtained from multiple tests to higher values, if the individual P-value is low enough it survives this process. The most simple method is just to multiply the individual, P-value (p_i) with the number of tests (m): $m \cdot p_i$, this is called "Bonferroni correction". However, this is too harsh if you have thousands of tests. Other methods are developed to remedy this. Those methods rely on ranking the P-values and dividing $m \cdot p_i$ by the rank,i, :$\frac{m \cdot p_i}{i}$, this is derived from Benjamini–Hochberg procedure. This procedure is developed to control for "False Discovery Rate (FDR)" , which is proportion of false positives among all significant tests. And in practical terms, we get the "FDR adjusted P-value" from the procedure described above. This gives us an estimate of proportion of false discoveries for a given test. To elaborate, p-value of 0.05 implies that 5% of all tests will be false positives. An FDR adjusted p-value of 0.05 implies that 5% of significant tests will be false positives. The FDR adjusted P-values will result in a lower number of false positives.

One final method that is also popular is called the "q-value" method and re-

lated to the method above. This procedure relies on estimating the proportion of true null hypotheses from the distribution of raw p-values and using that quantity to come up with what is called a "q-value", which is also an FDR adjusted P-value . That can be practically defined as "the proportion of significant features that turn out to be false leads." A q-value 0.01 would mean 1% of the tests called significant at this level will be truly null on average. Within the genomics community q-value and FDR adjusted P-value are synonymous although they can be calculated differently.

In R, the base function `p.adjust()` implements most of the p-value correction methods described above. For the q-value, we can use the `qvalue` package from Bioconductor. Below we are demonstrating how to use them on a set of simulated p-values.The plot shows that Bonferroni correction does a terrible job. FDR(BH) and q-value approach are better but q-value approach is more permissive than FDR(BH).

```
library(qvalue)
data(hedenfalk)

qvalues <- qvalue(hedenfalk$p)$q
bonf.pval=p.adjust(hedenfalk$p,method ="bonferroni")
fdr.adj.pval=p.adjust(hedenfalk$p,method ="fdr")

plot(hedenfalk$p,qvalues,pch=19,ylim=c(0,1),
     xlab="raw P-values",ylab="adjusted P-values")
points(hedenfalk$p,bonf.pval,pch=19,col="red")
points(hedenfalk$p,fdr.adj.pval,pch=19,col="blue")
legend("bottomright",legend=c("q-value","FDR (BH)","Bonferroni"),
       fill=c("black","blue","red"))
```

3.2.4 moderated t-tests: using information from multiple comparisons

In genomics, we usually do not do one test but many, as described above. That means we may be able to use the information from the parameters obtained from all comparisons to influence the individual parameters. For example, if you have many variances calculated for thousands

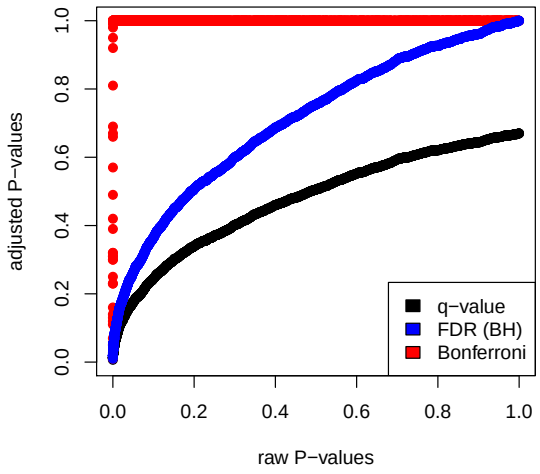


FIGURE 3.10: Adjusted P-values via different methods and their relationship to raw P-values

of genes across samples, you can force individual variance estimates to shrunk towards the mean or the median of the distribution of variances. This usually creates better performance in individual variance estimates and therefore better performance in significance testing which depends on variance estimates. How much the values be shrunk towards a common value comes in many flavors. These tests in general are called moderated t-tests or shrinkage t-tests. One approach popularized by Limma software is to use so-called "Empirical Bayesian methods". The main formulation in these methods is $\hat{V_g} = aV_0 + bV_g$, where V_0 is the background variability
and V_g is the individual variability. Then, these methods estimate a and b in various ways to come up with shrunk version of variability, $\hat{V_g}$. In a Bayesian viewpoint, the prior knowledge is used to calculate the variability of an individual gene. In this case, V_0 would be the prior knowledge we have on variability of the genes and we use that knowledge to influence our estimate for the individual genes.

Below we are simulating a gene expression matrix with 1000 genes, and 3 test and 3 control groups. Each row is a gene and in normal circumstances we would like to find out differentially expressed genes. In this case, we are simulating them from the same distribution so in reality we do not ex-

pect any differences. We then use the adjusted standard error estimates in empirical Bayesian spirit but in a very crude way. We just shrink the gene-wise standard error estimates towards the median with equal a and b weights. That is to say, we add individual estimate to the median of standard error distribution from all genes and divide that quantity by 2. So if we plug that in the to the above formula what we do is:

$$\hat{V_g} = (V_0 + V_g)/2$$

In the code below, we are avoiding for loops or apply family functions by using vectorized operations.

```
set.seed(100)

#sample data matrix from normal distribution

gset=rnorm(3000,mean=200,sd=70)
data=matrix(gset,ncol=6)

# set groups
group1=1:3
group2=4:6
n1=3
n2=3
dx=rowMeans(data[,group1])-rowMeans(data[,group2])

require(matrixStats)

# get the esimate of pooled variance
stderr <- sqrt( (rowVars(data[,group1])*(n1-1) + rowVars(data[,group2])*(n2-1)) / (n1+n2-2) *

# do the shrinking towards median
mod.stderr <- (stderr + median(stderr)) / 2 # moderation in variation

# esimate t statistic with moderated variance
t.mod = dx / mod.stderr
```

```
# calculate P-value of rejecting null
p.mod = 2*pt( -abs(t.mod), n1+n2-2 )

# esimate t statistic without moderated variance
t = dx / stderr

# calculate P-value of rejecting null
p = 2*pt( -abs(t), n1+n2-2 )

par(mfrow=c(1,2))
hist(p,col="cornflowerblue",border="white",main="",xlab="P-values t-test")
mtext(paste("signifcant tests:",sum(p<0.05))  )
hist(p.mod,col="cornflowerblue",border="white",main="",xlab="P-values mod. t-test")
mtext(paste("signifcant tests:",sum(p.mod<0.05))  )
```

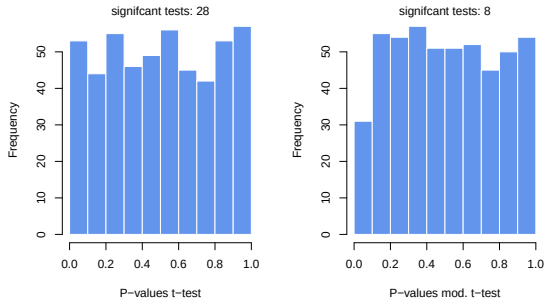


FIGURE 3.11: The distributions of P-values obtained by t-tests and moderated t-tests

Want to know more ?

- basic statistical concepts
  - "Cartoon guide to statistics" by Gonick & Smith
  - "Introduction to statistics" by Mine Rundel, et al. (Free e-book)
- Hands-on statistics recipes with R
  - "The R book" by Crawley
- moderated tests

* comparison of moderated tests for differential expression http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-11-17
* limma method: Smyth, G. K. (2004). Linear models and empirical Bayes methods for assessing differential expression in microarray experiments. Statistical Applications in Genetics and Molecular Biology, 3, No. 1, Article 3. http://www.statsci.org/smyth/pubs/ebayes.pdf

3.3 Relationship between variables: linear models and correlation

In genomics, we would often need to measure or model the relationship between variables. We might want to know about expression of a particular gene in liver in relation to the dosage of a drug that patient receives. Or, we may want to know DNA methylation of certain locus in the genome in relation to age of the sample donor's. Or, we might be interested in the relationship between histone modifications and gene expression. Is there a linear relationship, the more histone modification the more the gene is expressed ?

In these situations and many more, linear regression or linear models can be used to model the relationship with a "dependent" or "response" variable (expression or methylation in the above examples) and one or more "independent"" or "explanatory" variables (age, drug dosage or histone modification in the above examples). Our simple linear model has the following components.

$$Y = \beta_0 + \beta_1 X + \epsilon$$

In the equation above, Y is the response variable and X is the explanatory variable. ϵ is the mean-zero error term. Since, the line fit will not be able to precisely predict the Y values, there will be some error associated with each prediction when we compare it to the original Y values. This er-

ror is captured in ϵ term. We can alternatively write the model as follows to emphasize that the model approximates Y, in this case notice that we removed the ϵ term: $Y \sim \beta_0 + \beta_1 X$

The graph below shows the relationship between histone modification (trimethylated forms of histone H3 at lysine 4, aka H3K4me3) and gene expression for 100 genes. The blue line is our model with estimated coefficients ($\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 X$, where $\hat{\beta}_0$ and $\hat{\beta}_1$ the estimated values of β_0 and β_1, and $\hat{y}$ indicates the prediction). The red lines indicate the individual errors per data point, indicated as ϵ in the formula above.

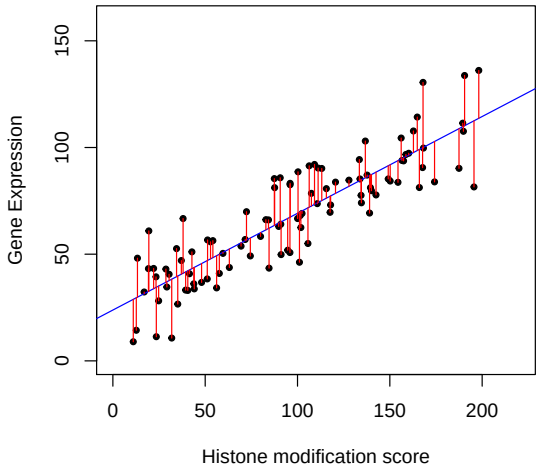


FIGURE 3.12: Relationship between histone modification score and gene expression. Increasing histone modification, H3K4me3, seems to be associated with increasing gene expression. Each dot is a gene

There could be more than one explanatory variable, we then simply add more X and β to our model. If there are two explanatory variables our model will look like this:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

In this case, we will be fitting a plane rather than a line. However, the fitting process which we will describe in the later sections will not change. For our gene expression problem. We can introduce one more histone modification, H3K27me3. We will then have a linear model with 2 explana-

tory variables and the fitted plane will look like the one below. The gene expression values are shown as dots below and above the fitted plane.

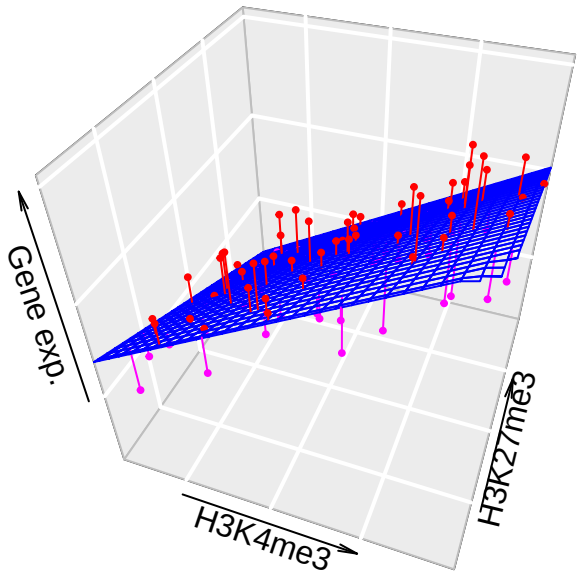


FIGURE 3.13: Association of Gene expression with H3K4me3 and H27Kme3 histone modifications.

3.3.0.1 Matrix notation for linear models

We can naturally have more explanatory variables than just two.The formula below has n explanatory variables.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + .. + \beta_n X_n + \epsilon$$

If there are many variables, it would be easier to write the model in matrix notation. The matrix form of linear model with two explanatory variables will look like the one below. First matrix would be our data matrix. This contains our explanatory variables and a column of 1s. The second term is a column vector of β values. We add a vector of error terms,ϵs to the matrix multiplication.

$$\mathbf{Y} = \begin{bmatrix} 1 & X_{1,1} & X_{1,2} \\ 1 & X_{2,1} & X_{2,2} \\ 1 & X_{3,1} & X_{3,2} \\ 1 & X_{4,1} & X_{4,2} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_0 \end{bmatrix}$$

The multiplication of data matrix and β vector and addition of the error terms simply results in the the following set of equations per data point:

$$\begin{aligned} Y_1 &= \beta_0 + \beta_1 X_{1,1} + \beta_2 X_{1,2} + \epsilon_1 \\ Y_2 &= \beta_0 + \beta_1 X_{2,1} + \beta_2 X_{2,2} + \epsilon_2 \\ Y_3 &= \beta_0 + \beta_1 X_{3,1} + \beta_2 X_{3,2} + \epsilon_3 \\ Y_4 &= \beta_0 + \beta_1 X_{4,1} + \beta_2 X_{4,2} + \epsilon_4 \end{aligned}$$

This expression involving the multiplication of the data matrix, the β vector and vector of error terms (ϵ) could be simply written as follows.

$$Y = X\beta + \epsilon$$

In the equation above Y is the vector of response variables and X is the data matrix and β is the vector of coefficients. This notation is more concise and often used in scientific papers. However, this also means you need some understanding of linear algebra to follow the math laid out in such resources.

3.3.1 How to fit a line

At this point a major questions is left unanswered: How did we fit this line? We basically need to define β values in a structured way. There are multiple ways or understanding how to do this, all of which converges to the same end point. We will describe them one by one.

3.3.1.1 The cost or loss function approach

This is the first approach and in my opinion is easiest to understand. We try to optimize a function, often called "cost function" or "loss function". The cost function is the sum of squared differences between the predicted

$\hat{Y}$ values from our model and the original Y values. The optimization procedure tries to find β values that minimizes this difference between reality and the predicted values.

$$min \sum (y_i - (\beta_0 + \beta_1 x_i))^2$$

Note that this is related to the the error term, ϵ, we already mentioned above, we are trying to minimize the squared sum of ϵ_i for each data point. We can do this minimization by a bit of calculus. The rough algorithm is as follows:

1. Pick a random starting point, random β values
2. Take the partial derivatives of the cost function to see which direction is the way to go in the cost function.
3. Take a step toward the direction that minimizes the cost function.
    - step size is parameter to choose, there are many variants.
4. repeat step 2,3 until convergence.

This is the basis of "gradient descent" algorithm. With the help of partial derivatives we define a "gradient" on the cost function and follow that through multiple iterations and until convergence, meaning until the results do not improve defined by a margin. The algorithm usually converges to optimum β values. Below, we show the cost function over various β_0 and β_1 values for the histone modification and gene expression data set. The algorithm will pick a point on this graph and traverse it incrementally based on the derivatives and converge on the bottom of the cost function "well".

3.3.1.2 Not cost function but maximum likelihood function

We can also think of this problem from more a statistical point of view. In essence, we are looking for best statistical parameters, in this case β values, for our model that are most likely to produce such a scatter of data points given the explanatory variables.This is called "Maximum likelihood" approach. Probability of observing a Y value, given that the distribution of it on a given X value follows a normal distribution with mean

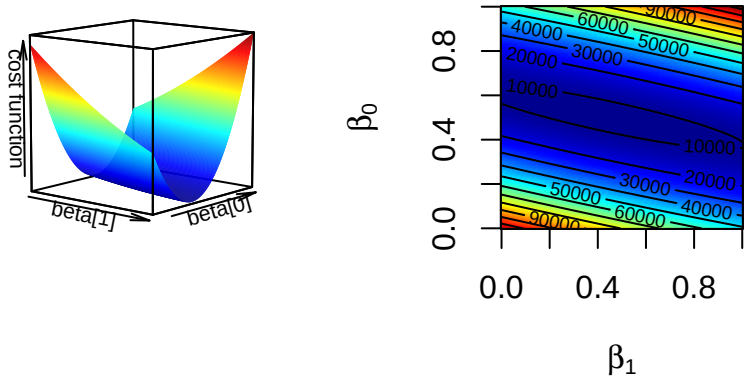


FIGURE 3.14: Cost function landscape for linear regression with changing beta values. The optimization process tries to find the lowest point in this landscape by implementing a strategy for updating beta values towards the lowest point in the landscape.

$\beta_0 + \beta_1 x_i$ and variance s^2 , and is shown below. Note that this assumes variance is constant and $s^2 = \frac{\sum \epsilon_i}{n-2}$ is an unbiased estimation for population variance, σ^2.

$$P(y_i) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - (\beta_0 + \beta_1 x_i)}{s}\right)^2}$$

Following from this, then the likelihood function ,shown as L below, for linear regression is multiplication of $P(y_i)$ for all data points.

$$L = P(y_1)P(y_2)P(y_3)..P(y_n) = \prod_{i=1}^{n} P_i$$

This can be simplified to this by some algebra and taking logs (since it is easier to add than multiply)

$$ln(L) = -nln(s\sqrt{2\pi}) - \frac{1}{2s^2}\sum_{i=1}^{n}(y_i - (\beta_0 - \beta_1 x_i))^2$$

As you can see, the right part of the function is the negative of the cost function defined above. If we wanted to optimize this function we would

need to take derivative of the function with respect to β parameters. That means we can ignore the first part since there is no β terms there. This simply reduces to the negative of the cost function. Hence, this approach produces exactly the same result as the cost function approach. The difference is that we defined our problem within the domain of statistics. This particular function has still to be optimized. This can be done with some calculus without the need for an iterative approach.

3.3.1.3 Linear algebra and closed-form solution to linear regression

The last approach we will describe is the minimization process using linear algebra. If you find this concept challenging, feel free to skip it but scientific publications and other books frequently use matrix notation and linear algebra to define and solve regression problems. In this case, we do not use an iterative approach. Instead, we will minimize cost function by explicitly taking its derivatives with respect to β's and setting them to zero. This is doable by employing linear algebra and matrix calculus. This approach is also called "ordinary least squares". We will not show the whole derivation here but the following expression is what we are trying to minimize in matrix notation, this is basically a different notation of the same minimization problem defined above. Remember $\epsilon_i = Y_i - (\beta_0 + \beta_1 x_i)$

$$\begin{aligned}\sum \epsilon_i^2 = \epsilon^T \epsilon &= (Y - \beta X)^T (Y - \beta X) \\ &= Y^T Y - 2\beta^T Y + \beta^T X^T X \beta\end{aligned}$$

After rearranging the terms, we take the derivative of $\epsilon^T \epsilon$ with respect to β, and equalize that to zero. We then arrive at the following for estimated β values, $\hat{\beta}$:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

This requires for you to calculate the inverse of the $X^T X$ term, which could be slow for large matrices. Iterative approach over the cost function derivatives will be faster for larger problems. The linear algebra notation is something you will see in the papers or other resources often. If you input the data matrix X and solve the $(X^T X)^{-1}$, you get the following

values for β_0 and β_1 for simple regression . However, we should note that this simple linear regression case can easily be solved algebraically without the need for matrix operations. This can be done by taking the derivative of $\sum (y_i - (\beta_0 + \beta_1 x_i))^2$ with respect to β_1, rearranging the terms and equalizing the derivative to zero.

$$\hat{\beta_1} = \frac{\sum (x_i - \overline{X})(y_i - \overline{Y})}{\sum (x_i - \overline{X})^2}$$
$$\hat{\beta_0} = \overline{Y} - \hat{\beta_1}\overline{X}$$

3.3.1.4 Fitting lines in R

After all this theory, you will be surprised how easy it is to fit lines in R. This is achieved just by `lm()` command, stands for linear models. Let's do this for a simulated data set and plot the fit. First step is to simulate the data, we will decide on β_0 and β_1 values. The we will decide on the variance parameter,σ to be used in simulation of error terms, ϵ. We will first find Y values, just using the linear equation $Y = \beta 0 + \beta_1 X$, for a set of X values. Then, we will add the error terms get our simulated values.

```
# set random number seed, so that the random numbers from the text
# is the same when you run the code.
set.seed(32)

# get 50 X values between 1 and 100
x = runif(50,1,100)

# set b0,b1 and varience (sigma)
b0 = 10
b1 = 2
sigma = 20
# simulate error terms from normal distribution
eps = rnorm(50,0,sigma)
# get y values from the linear equation and addition of error terms
y = b0 + b1*x+ eps
```

Now let us fit a line using lm() function. The function requires a formula,

and optionally a data frame. We need the pass the following expression within the lm function, `y~x`, where `y` is the simulated Y values and `x` is the explanatory variables X. We will then use `abline()` function to draw the fit.

```
mod1=lm(y~x)

# plot the data points
plot(x,y,pch=20,
     ylab="Gene Expression",xlab="Histone modification score")
# plot the linear fit
abline(mod1,col="blue")
```

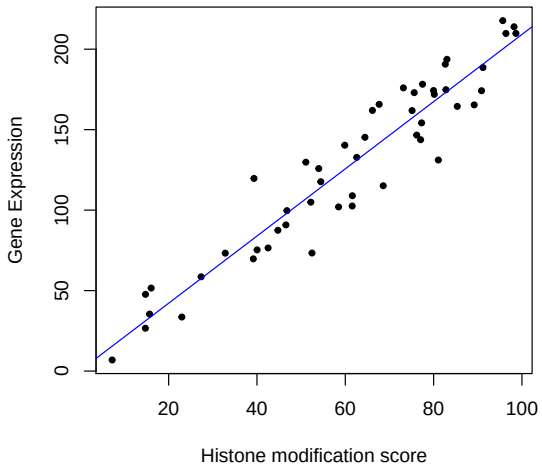


FIGURE 3.15: Gene expression and histone modification score modelled by linear regression

3.3.2 How to estimate the error of the coefficients

Since we are using a sample to estimate the coefficients they are not exact, with every random sample they will vary. Below, we are taking multiple samples from the population and fitting lines to each sample, with each sample the lines slightly change.We are overlaying the points and the lines for each sample on top of the other samples .When we take 200 sam-

ples and fit lines for each of them,the lines fit are variable. And, we get a normal-like distribution of β values with a defined mean and standard deviation a, which is called standard error of the coefficients.

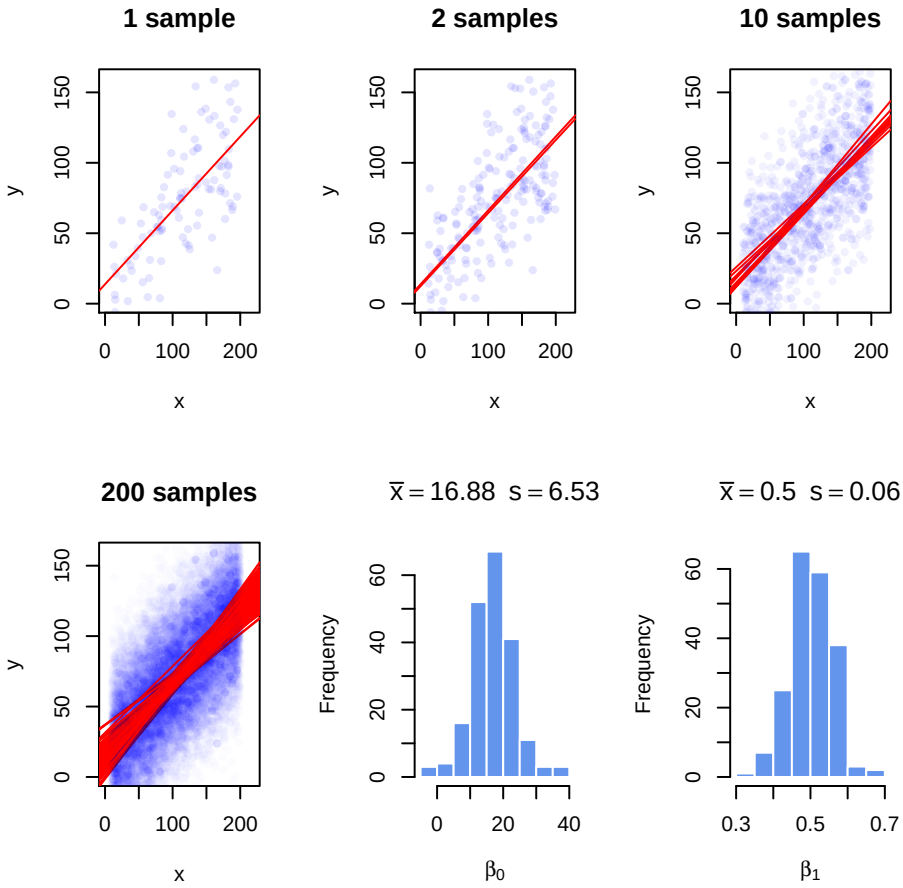


FIGURE 3.16: Regression coefficients vary with every random sample. The figure illustrates the variability of regression coefficients when regression is done using a sample of data points. Histograms depict this variability for b_0 and b_1 coefficients.

As usually we will not have access to the population to do repeated sampling, model fitting and estimation of the standard error for the coefficients. But there is statistical theory that helps us infer the population properties from the sample. When we assume that error terms have constant variance and mean zero , we can model the uncertainty in the regres-

sion coefficients, βs. The estimates for standard errors of βs for simple regression are as follows and shown without derivation.

$$s = RSE = \sqrt{\frac{\sum (y_i - (\beta_0 + \beta_1 x_i))^2}{n-2}} = \sqrt{\frac{\sum \epsilon^2}{n-2}}$$

$$SE(\hat{\beta}_1) = \frac{s}{\sqrt{\sum (x_i - \overline{X})^2}}$$

$$SE(\hat{\beta}_0) = s\sqrt{\frac{1}{n} + \frac{\overline{X}^2}{\sum (x_i - \overline{X})^2}}$$

Notice that that $SE(\beta_1)$ depends on the estimate of variance of residuals shown as s or Residual Standard Error (RSE). Notice alsos standard error depends on the spread of X. If X values have more variation, the standard error will be lower. This intuitively makes sense since if the spread of the X is low, the regression line will be able to wiggle more compared to a regression line that is fit to the same number of points but covers a greater range on the X-axis.

The standard error estimates can also be used to calculate confidence intervals and test hypotheses, since the following quantity called t-score approximately follows a t-distribution with $n-p$ degrees of freedom, where n is the number of data points and p is the number of coefficients estimated.

$$\frac{\hat{\beta}_i - \beta_t est}{SE(\hat{\beta}_i)}$$

Often, we would like to test the null hypothesis if a coefficient is equal to zero or not. For simple regression this could mean if there is a relationship between explanatory variable and response variable. We would calculate the t-score as follows $\frac{\hat{\beta}_i - 0}{SE(\hat{\beta}_i)}$, and compare it t-distribution with $d.f. = n - p$ to get the p-value.

We can also calculate the uncertainty of the regression coefficients using confidence intervals, the range of values that are likely to contain β_i. The

95% confidence interval for $\hat{\beta}_i$ is $\hat{\beta}_i \pm t_{0.975} SE(\hat{\beta}_i)$. $t_{0.975}$ is the 97.5% percentile of the t-distribution with $d.f. = n–p$.

In R, `summary()` function will test all the coefficients for the null hypothesis $\beta_i = 0$. The function takes the model output obtained from the `lm()` function. To demonstrate this, let us first get some data. The procedure below simulates data to be used in a regression setting and it is useful to examine what the linear model expect to model the data.

Since we have the data, we can build our model and call the `summary` function. We will then use `confint()` function to get the confidence intervals on the coefficients and `coef()` function to pull out the estimated coefficients from the model.

```
mod1=lm(y~x)
summary(mod1)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -77.11 -18.44   0.33  16.06  57.23
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  13.2454     6.2887    2.11    0.038 *
## x             0.4995     0.0513    9.74  4.5e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28.8 on 98 degrees of freedom
## Multiple R-squared:  0.492,  Adjusted R-squared:  0.486
## F-statistic: 94.8 on 1 and 98 DF,  p-value: 4.54e-16
```

```
# get confidence intervals
confint(mod1)
```

```
##               2.5 %  97.5 %
## (Intercept) 0.7657 25.7251
## x           0.3977  0.6014
```

```
# pull out coefficients from the model
coef(mod1)
```

```
## (Intercept)           x
##     13.2454      0.4995
```

The `summary()` function prints out an extensive list of values. The "Coefficients" section has the estimates, their standard error, t score and the p-value from the hypothesis test $H_0 : \beta_i = 0$. As you can see, the estimate we get for the coefficients and their standard errors are close to the ones we get from the repeatedly sampling and getting a distribution of coefficients. This is statistical inference at work, we can estimate the population properties within a certain error using just a sample.

3.3.3 Accuracy of the model

If you have observed the table output by `summary()` function, you must have noticed there are some other outputs, such as "Residual standard error", "Multiple R-squared" and "F-statistic". These are metrics that are useful for assessing the accuracy of the model. We will explain them one by one.

_ (RSE)_ simply is the square-root of the the sum of squared error terms, divided by degrees of freedom, $n - p$, for simple linear regression case, $n - 2$. Sum of of the squares of the error terms is also called "Residual sum of squares", RSS. So RSE is calculated as follows:

$$s = RSE = \sqrt{\frac{\sum (y_i - \hat{Y}_i)^2}{n-p}} = \sqrt{\frac{RSS}{n-p}}$$

RSE is a way of assessing the model fit. The larger the RSE the worse the model is. However, this is an absolute measure in the units of Y and we have nothing to compare against. One idea is that we divide it by RSS of a simpler model for comparative purposes. That simpler model is in this case is the model with the intercept,β_0. A very bad model will have close zero coefficients for explanatory variables, and the RSS of that model will be close to the RSS of the model with only the intercept. In such a model intercept will be equal to $\overline{Y}$. As it turns out, RSS of the the model with just the intercept is called "Total Sum of Squares" or TSS. A good model will have a low RSS/TSS. The metric R^2 uses these quantities to calculate a score between 0 and 1, and closer to 1 the better the model. Here is how it is calculated:

$$R^2 = 1 - \frac{RSS}{TSS} = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}$$

$TSS - RSS$ part of the formula often referred to as "explained variability" in the model. The bottom part is for "total variability". With this interpretation, higher the "explained variability" better the model. For simple linear regression with one explanatory variable, the square root of R^2 is a quantity known as absolute value of the correlation coefficient, which can be calculated for any pair of variables, not only the response and the explanatory variables. Correlation is a general measure of linear relationship between two variables. One of the most popular flavors of correlation is the Pearson correlation coefficient. Formally, It is the covariance of X and Y divided by multiplication of standard deviations of X and Y. In R, it can be calculated with `cor()` function.

$$r_{xy} = \frac{cov(X,Y)}{\sigma_x \sigma_y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

In the equation above, cov is the covariance, this is again a measure of how much two variables change together, like correlation. If two variables show similar behavior they will usually have positive covariance value, if they have opposite behavior, the covariance will have negative value. However, these values are boundless. A normalized way of looking at covari-

ance is to divide covariance by the multiplication of standard errors of X and Y. This bounds the values to -1 and 1, and as mentioned above called Pearson correlation coefficient. The values that change in a similar manner will have a positive coefficient, the values that change in opposite manner will have negative coefficient, and pairs do not have a linear relationship will have 0 or near 0 correlation. In the figure below, we are showing R^2, correlation coefficient and covariance for different scatter plots.

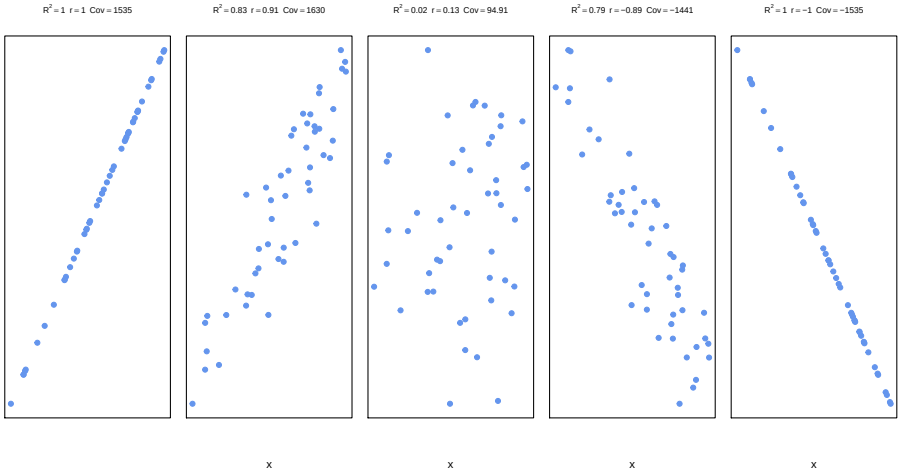


FIGURE 3.17: Correlation and covariance for different scatter plots

For simple linear regression, correlation can be used to asses the model. However, this becomes useless as a measure of general accuracy if the there are more than one explanatory variable as in multiple linear regression. In that case, R^2 is a measure of accuracy for the model. Interestingly, square of the correlation of predicted values and original response variables ($(cor(Y, \hat{Y}))^2$) equals to R^2 for multiple linear regression.

The last accuracy measure or the model fit in general we are going to explain is F-statistic. This is a quantity that depends on RSS and TSS again. It can also answer one important question that other metrics can not easily answer. That question is whether or not any of the explanatory variables have predictive value or in other words if all the explanatory variables are zero. We can write the null hypothesis as follows:

$$H_0 : \beta_1 = \beta_2 = \beta_3 = ... = \beta_p = 0$$

where the alternative is:

$$H_1 : \text{at least one } \beta_i \neq 0$$

Remember $TSS - RSS$ is analogous to "explained variability" and the RSS is analogous to "unexplained variability". For the F-statistic, we divide explained variance to unexplained variance. Explained variance is just the $TSS - RSS$ divided by degrees of freedom, and unexplained variance is the RSE. The ratio will follow the F-distribution with two parameters, the degrees of freedom for the explained variance and the degrees of freedom for the the unexplained variance.F-statistic for a linear model is calculated as follows.

$$F = \frac{(TSS - RSS)/(p-1)}{RSS/(n-p)} = \frac{(TSS - RSS)/(p-1)}{RSE} \sim F(p{-}1, n{-}p)$$

If the variances are the same, the ratio will be 1, and when H_0 is true, then it can be shown that expected value of $(TSS - RSS)/(p-1)$ will be σ^2 which is estimated by RSE. So, if the variances are significantly different, the ratio will need to be significantly bigger than 1. If the ratio is large enough we can reject the null hypothesis. To asses that we need to use software or look up the tables for F statistics with calculated parameters. In R, function `qf()` can be used to calculate critical value of the ratio. Benefit of the F-test over looking at significance of coefficients one by one is that we circumvent multiple testing problem. If there are lots of explanatory variables at least 5% of the time (assuming we use 0.05 as P-value significance cutoff), p-values from coefficient t-tests will be wrong. In summary, F-test is a better choice for testing if there is any association between the explanatory variables and the response variable.

3.3.4 Regression with categorical variables

An important feature of linear regression is that categorical variables can be used as explanatory variables, this feature is very useful in genomics where explanatory variables often could be categorical. To put it in context, in our histone modification example we can also include if promoters

have CpG islands or not as a variable. In addition, in differential gene expression, we usually test the difference between different condition which can be encoded as categorical variables in a linear regression. We can sure use t-test for that as well if there are only 2 conditions, but if there are more conditions and other variables to control for such as Age or sex of the samples, we need to take those into account for our statistics, and t-test alone can not handle such complexity. In addition, when we have categorical variables we can also have numeric variables in the model and we certainly do not have to include only one type of variable in a model.

The simplest model with categorical variables include two levels that can be encoded in 0 and 1.

```
set.seed(100)
gene1=rnorm(30,mean=4,sd=2)
gene2=rnorm(30,mean=2,sd=2)
gene.df=data.frame(exp=c(gene1,gene2),
                   group=c( rep(1,30),rep(0,30) ) )

mod2=lm(exp~group,data=gene.df)
summary(mod2)
```

```
##
## Call:
## lm(formula = exp ~ group, data = gene.df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.729 -1.066  0.012  1.384  4.563
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    2.185      0.352    6.21    6e-08 ***
## group          1.873      0.497    3.77  0.00039 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 1.93 on 58 degrees of freedom
## Multiple R-squared:  0.196,  Adjusted R-squared:  0.183
## F-statistic: 14.2 on 1 and 58 DF,  p-value: 0.000391
```

```
require(mosaic)
plotModel(mod2)
```

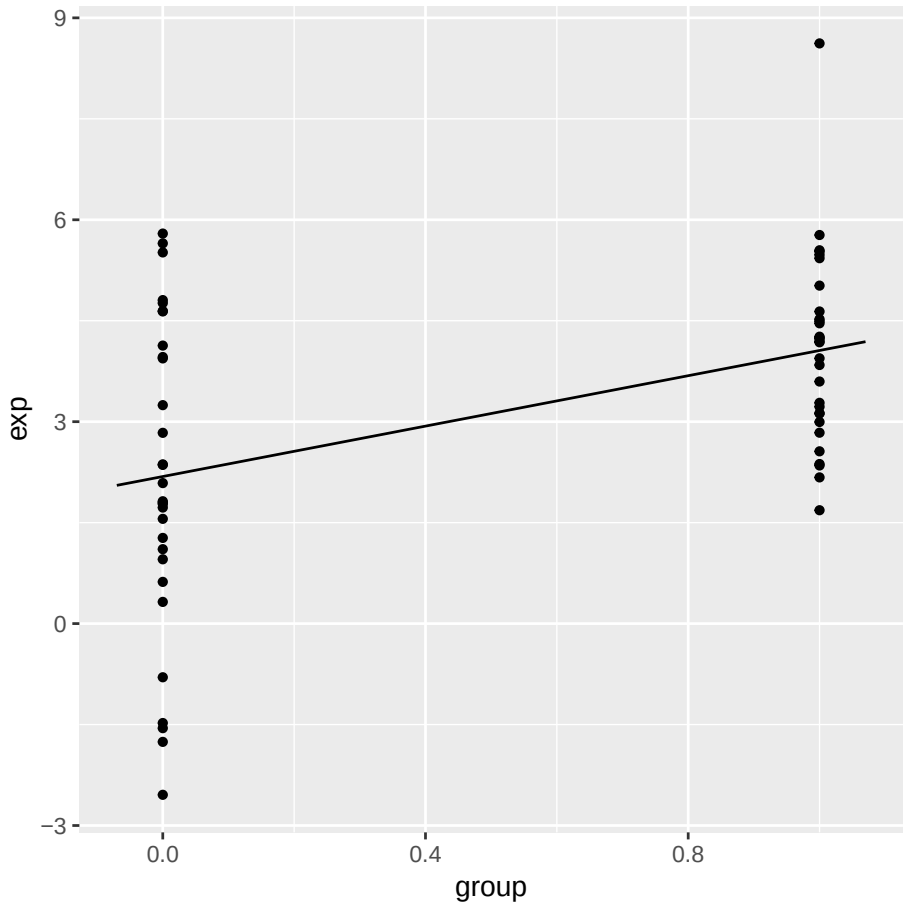


FIGURE 3.18: Linear model with a categorical variable coded as 0 and 1

we can even compare more levels, we do not even have to encode them ourselves. We can pass categorical variables to `lm()` function.

```
gene.df=data.frame(exp=c(gene1,gene2,gene2),
                   group=c( rep("A",30),rep("B",30),rep("C",30) )
                   )

mod3=lm(exp~group,data=gene.df)
summary(mod3)
```

```
##
## Call:
## lm(formula = exp ~ group, data = gene.df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.729 -1.079 -0.098  1.484  4.563
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    4.058      0.378    10.7  < 2e-16 ***
## groupB        -1.873      0.535    -3.5  0.00073 ***
## groupC        -1.873      0.535    -3.5  0.00073 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.07 on 87 degrees of freedom
## Multiple R-squared:  0.158,  Adjusted R-squared:  0.139
## F-statistic: 8.17 on 2 and 87 DF,  p-value: 0.000558
```

3.3.5 Regression pitfalls

In most cases one should look at the error terms (residuals) vs fitted values plot. Any structure in this plot indicates problems such as non-linearity, correlation of error terms, non-constant variance or unusual values driving the fit. Below we briefly explain the potential issues with the linear regression.

3.3.5.0.1 non-linearity

If the true relationship is far from linearity, prediction accuracy is reduced and all the other conclusions are questionable. In some cases, transforming the data with $logX$, $\sqrt{X}$ and X^2 could resolve the issue.

3.3.5.0.2 correlation of explanatory variables

If the explanatory variables are correlated that could lead to something known as multicolinearity. When this happens SE estimates of the coefficients will be too large. This is usually observed in time-course data.

3.3.5.0.3 correlation of error terms

This assumes that the errors of the response variables are uncorrelated with each other. If they are confidence intervals in the coefficients might too narrow.

3.3.5.0.4 Non-constant variance of error terms

This means that different response variables have the same variance in their errors, regardless of the values of the predictor variables. If the errors are not constant, if for the errors grow as X grows this will result in unreliable estimates in standard errors as the model assumes constant variance. Transformation of data, such as $logX$ and $\sqrt{X}$ could help in some cases.

3.3.5.0.5 outliers and high leverage points

Outliers are extreme values for Y and high leverage points are unusual X values. Both of these extremes have power to affect the fitted line and the standard errors. In some cases (measurement error), they can be removed from the data for a better fit.

Want to know more ?

- linear models and derivations of equations including matrix notation
  * Applied Linear Statistical Models by Kutner, Nachtsheim, et al.
  * Elements of statistical learning by Hastie & Tibshirani
  * An Introduction to statistical learning by James, Witten, et al.

3.4 Clustering: grouping samples based on their similarity

In genomics, we would very frequently want to assess how our samples relate to each other. Are our replicates similar to each other? Do the samples from the same treatment group have the similar genome-wide signals ? Do the patients with similar diseases have similar gene expression profiles ? Take the last question for example. We need to define a distance or similarity metric between patients' expression profiles and use that metric to find groups of patients that are more similar to each other than the rest of the patients. This, in essence, is the general idea behind clustering. We need a distance metric and a method to utilize that distance metric to find self-similar groups. Clustering is a ubiquitous procedure in bioinformatics as well as any field that deals with high-dimensional data. It is very likely every genomics paper containing multiple samples have some sort of clustering. Due to this ubiquity and general usefulness, it is an essential technique to learn.

TABLE 3.2: Gene expressions from patients

	IRX4	OCT4	PAX6
patient1	11	10	1
patient2	13	13	3
patient3	2	4	10
patient4	1	3	9

3.4.1 Distance metrics

The first required step for clustering is the distance metric. This is simply a measurement of how similar gene expressions to each other are. There are many options for distance metrics and the choice of the metric is quite important for clustering. Consider a simple example where we have four patients and expression of three genes measured. Which patients look similar to each other based on their gene expression profiles ?

It may not be obvious from the table at first sight but if we plot the gene expression profile for each patient, we will see that expression profiles of patient 1 and patient 2 is more similar to each other than patient 3 or patient 4.

But how can we quantify what see by eye ? A simple metric for distance between gene expression vectors between a given patient pair is the sum of absolute difference between gene expression values This can be formulated as follows: $d_{AB} = \sum_{i=1}^{n} |e_{Ai} - e_{Bi}|$, where d_{AB} is the distance between patient A and B, and e_{Ai} and e_{Bi} expression value of the ith gene for patient A and B. This distance metric is called "Manhattan distance" or "L1 norm".

Another distance metric using sum of squared distances and taking a square root of resulting value, that can be formulaized as: $d_{AB} = \sqrt{\sum_{i=1}^{n}(e_{Ai} - e_{Bi})^2}$. This distance is called "Euclidean Distance" or "L2 norm". This is usually the default distance metric for many clustering algorithms. due to squaring operation values that are very different get higher contribution to the distance. Due to this, compared to Manhattan distance it can be more affected by outliers but generally if the outliers are rare this distance metric works well.

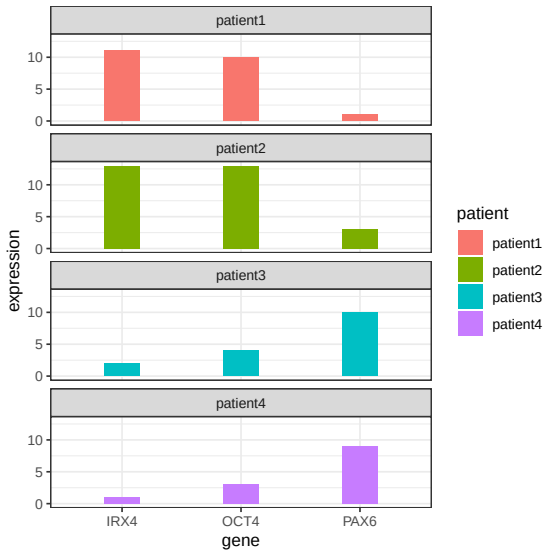


FIGURE 3.19: Gene expression values for different patients. Certain patients have similar gene expression values to each other.

The last metric we will introduce is the "correlation distance". This is simply $d_{AB} = 1 - \rho$, where ρ is the pearson correlation coefficient between two vectors, in our case those vectors are gene expression profiles of patients. Using this distance the gene expression vectors that have a similar pattern will have a small distance whereas when the vectors have different patterns they will have a large distance. In this case, the linear correlation between vectors matters, the the scale of the vectors might be different.

Now let's see how we can calculate these distance in R. First, we have our gene expression per patient table.

```
df
```

```
##          IRX4 OCT4 PAX6
## patient1   11   10    1
## patient2   13   13    3
## patient3    2    4   10
## patient4    1    3    9
```

Next, we calculate the distance metrics using `dist` function and `1-cor()`.

```
dist(df,method="manhattan")
```

```
##          patient1 patient2 patient3
## patient2        7
## patient3       24       27
## patient4       25       28        3
```

```
dist(df,method="euclidean")
```

```
##          patient1 patient2 patient3
## patient2    4.123
## patient3   14.071   15.843
## patient4   14.595   16.733    1.732
```

```
as.dist(1-cor(t(df)))
```

```
##          patient1 patient2 patient3
## patient2 0.004129
## patient3 1.988522 1.970725
## patient4 1.988522 1.970725 0.000000
```

3.4.1.1 Scaling before calculating the distance

Before we proceed to the clustering, one more thing we need to take care. Should we normalize our data ? Scale of the vectors in our expression matrix can affect the distance calculation. Gene expression tables are usually have some sort of normalization, so the values are in comparable scales. But somehow if a gene's expression values were on much higher scale than the other genes, that gene will effect the distance more than other when using Euclidean or Manhattan distance. If that is the case we can scale the variables.The traditional way of scaling variables is to subtract their mean, and divide by their standard deviation, this operation is also called "standardization". If this is done on all genes, each gene will have the same affect on distance measures. The decision to apply scaling ultimately depends on our data and what you want to achieve. If the gene expression values are previously normalized between patients, hav-

ing genes that dominate the distance metric could have a biological meaning and therefore it may not be desireable to further scale variables. In R, the standardization is done via `scale()` function. Here we scale the gene expression values.

```
df
```

```
##          IRX4 OCT4 PAX6
## patient1   11   10    1
## patient2   13   13    3
## patient3    2    4   10
## patient4    1    3    9
```

```
scale(df)
```

```
##             IRX4    OCT4    PAX6
## patient1  0.6933  0.5213 -1.0734
## patient2  1.0195  1.1468 -0.6214
## patient3 -0.7748 -0.7298  0.9604
## patient4 -0.9379 -0.9383  0.7344
## attr(,"scaled:center")
## IRX4 OCT4 PAX6
## 6.75 7.50 5.75
## attr(,"scaled:scale")
##  IRX4  OCT4  PAX6
## 6.131 4.796 4.425
```

3.4.2 Hiearchical clustering

This is one of the most ubiqutous clustering algorithms. Using this algorithm you can see the relationship of individual data points and relationships of clusters. This is achieved successively joining small clusters to each other based on the intercluster distance. Eventually, you get a tree structure or a dendrogram that shows the relationship between the individual data points and clusters. The height of the dendrogram is the distance between clusters. Here we can show how to use this on our toy data

set from four patients. The base function in R to do hierarchical clustering in `hclust()`. Below, we apply that function on Euclidean distances between patients.

```
d=dist(df)
hc=hclust(d,method="complete")
plot(hc)
```

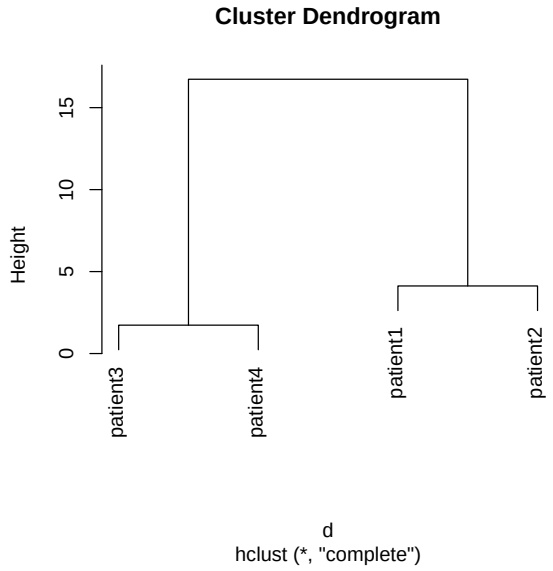


FIGURE 3.20: Dendrogram of distance matrix

In the above code snippet, we have used `method="complete"` argument without explaining it. The `method` argument defines the criteria that directs how the sub-clusters are merged. During clustering starting with single-member clusters, the clusters are merged based on the distance between them. There are many different ways to define distance between clusters and based on which definition you use the hierarchical clustering results change. So the `method` argument controls that. There are a couple of values this argument can take, we list them and their description below:

- "complete" stands for "Complete Linkage" and the distance between two clusters is defined as largest distance between any members of the two clusters.

- "single" stands for "Single Linkage" and the distance between two clusters is defined as smallest distance between any members of the two clusters.
- "average" stands for "Average Linkage" or more precisely UPGMA (Unweighted Pair Group Method with Arithmetic Mean) method. In this case, the distance between two clusters is defined as average distance between any members of the two clusters.
- "ward.D2" and "ward.D" stands for different implementations of Ward's minimum variance method. This method aims to find compact, spherical clusters by selecting clusters to merge based on the change in the cluster variances. The clusters are merged if the increase in the combined variance over the sum of the cluster specific variances is minimum compared to alternative merging operations.

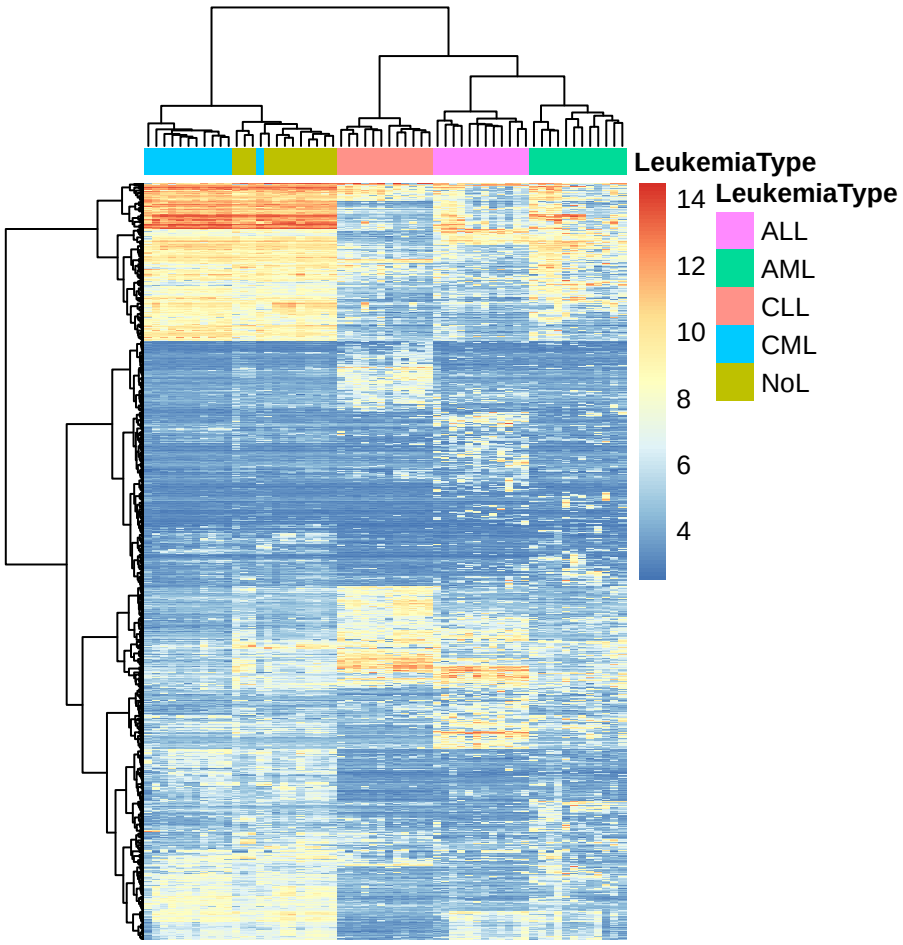
In real life, we would get expression profiles from thousands of genes and we will typically have many more patients than our toy example. One such data set is gene expression values from 60 bone marrow samples of patients with one of the four main types of leukemia (ALL, AML, CLL, CML) or no-leukemia controls. We trimmed that data set down to top 1000 most variable genes to be able to work with it easier and in addition genes that are not very variable do not contribute much to the distances between patients. We will now use this data set to cluster the patients and display the values as a heatmap and a dendrogram. The heatmap shows the expression values of genes across patients in a color coded manner. The heatmap function, `pheatmap()`, we will use performs the clustering as well. The matrix that contains gene expressions has the genes in the rows and the patients in the columns. Therefore, we will also use a column-side color code to mark the patients based on their leukemia type. For the hierarchical clustering, we will use Ward's method designated by `clustering_method` argument to `pheatmap()` function.

```
library(pheatmap)
expFile=system.file("extdata","leukemiaExpressionSubset.rds",package="compGenomRData")
mat=readRDS(expFile)

# set the leukemia type annotation for each sample
annotation_col = data.frame(
```

```
                    LeukemiaType =substr(colnames(mat),1,3))
rownames(annotation_col)=colnames(mat)


pheatmap(mat,show_rownames=FALSE,show_colnames=FALSE,annotation_col=annotation_col,scale = "no
```



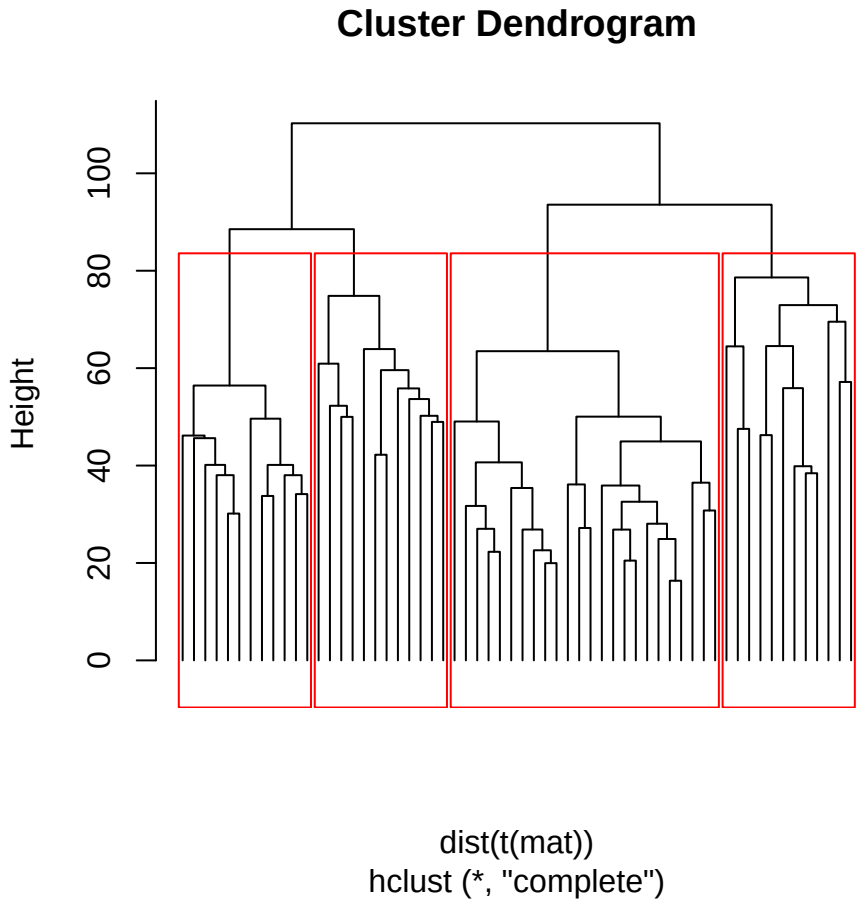
As we can observe in the heatmap each cluster has a distinct set of expression values. The main clusters almost perfectly distinguish the leukemia types. Only one CML patient is clustered as a non-leukemia sample. This could mean that gene expression profiles are enough to classify leukemia

type. More detailed analysis and experiments are needed to verify that but by looking at this exploratory analysis we can decide where to focus our efforts next.

3.4.2.1 where to cut the tree ?

The example above seems like a clear cut example where we can pick by eye clusters from the dendrogram. This is mostly due to the Ward's method where compact clusters are preffered. However, as it is usually the case we do not have patient labels and it would be difficult to tell which leaves (patients) in the dendrogram we should consider as part of the same cluster. In other words, how deep we should cut the dendrogram so that every patient sample still connected via the remaining sub-dendrograms constitute clusters. The `cutree()` function provides the functionality to output either desired number of clusters or clusters obtained from cutting the dendrogram at a certain height. Below, we will cluster the patients with hierarchical clustering using the default method "complete linkage" and cut the dendrogram at a certain height. In this case, you will also observe that, changing from Ward's distance to complete linkage had an effect on clustering. Now two clusters that are defined by Ward's distance are closer to each other and harder to separate from each other.

```
hcl=hclust(dist(t(mat)))
plot(hcl,labels = FALSE, hang= -1)
rect.hclust(hcl, h = 80, border = "red")
```




```
clu.k5=cutree(hcl,k=5) # cut tree so that there are 4 clusters

clu.h80=cutree(hcl,h=80) # cut tree/dendrogram from height 80
table(clu.k5) # number of samples for each cluster
```

```
## clu.k5
##  1  2  3  4  5
## 12  3  9 12 24
```

Apart from the arbitrary values for the height or the number of the clusters, how can we define clusters more systematically? As this is a general question, we will show later how to decide the optimal number of clusters later in this chapter.

3.4.3 K-means clustering

Another, very common clustering algorithm is k-means.This method divides or partitions the data points, our working example patients, into a pre-determined, "k" number of clusters. Hence, this type of methods are generally called "partioning" methods. The algorithm is initialized with randomly choosen k centers or centroids. In a sense, a centroid is a data point with multiple values. In our working example, it is a hypothetical patient with gene expression values. But in the initialization phase, those gene expression values are choosen randomly within the boundaries of the gene expression distributions from real patients. As the next step in the algorithm, each patient is assigned to the closest centroid and in the next iteration centroids are set to the mean of values of the genes in the cluster. This process of setting centroids and assigning patients to the clusters repeats itself until sum of squared distances to cluster centroids is minimized.

As you might see, the cluster algorithm starts with random initial centroids. This feature might yield different results for each run of the algorithm. We will know show how to use k-means method on the gene expression data set. We will use `set.seed()` for reproducbility. In the wild, you might want to run this algorithm multiple times to see if your clustering results are stable.

```
set.seed(101)

# we have to transpore the matrix t()
# so that we calculate distances between patients
kclu=kmeans(t(mat),centers=5)

# number of data points in each cluster
table(kclu$cluster)
```

```
##
##  1  2  3  4  5
## 12 12 14 11 11
```

Now let us check the percentage of each leukemia type in each cluster. We

can visualize this as a table. Looking at the table below, we see that each of the 5 clusters are predominantly representing one of the 4 leukemia types or the control patients without leukemia.

```
type2kclu = data.frame(
                    LeukemiaType =substr(colnames(mat),1,3),
                    cluster=kclu$cluster)

table(type2kclu)
```

```
##             cluster
## LeukemiaType  1  2  3  4  5
##          ALL 12  0  0  0  0
##          AML  0  0  1  0 11
##          CLL  0 12  0  0  0
##          CML  0  0  1 11  0
##          NoL  0  0 12  0  0
```

Another related and maybe more robust algorithm is called "k-medoids" clustering. The procedure is almost identical to k-means clustering with a couple of differences. In this case, centroids choosen are real data points in our case patients, and the metric we are trying to optimize in each iteration is based on manhattan distance to the centroid. In k-means this was based on sum of squared distances so euclidean distance. Below we are showing how to use k-medoids clustering function `pam()` from the `cluster` package.

```
kmclu=cluster::pam(t(mat),k=5) #  cluster using k-medoids

# make a data frame with Leukemia type and cluster id
type2kmclu = data.frame(
                    LeukemiaType =substr(colnames(mat),1,3),
                    cluster=kmclu$cluster)

table(type2kmclu)
```

```
##             cluster
```

```
## LeukemiaType  1  2  3  4  5
##          ALL 12  0  0  0  0
##          AML  0 10  1  1  0
##          CLL  0  0  0  0 12
##          CML  0  0  0 12  0
##          NoL  0  0 12  0  0
```

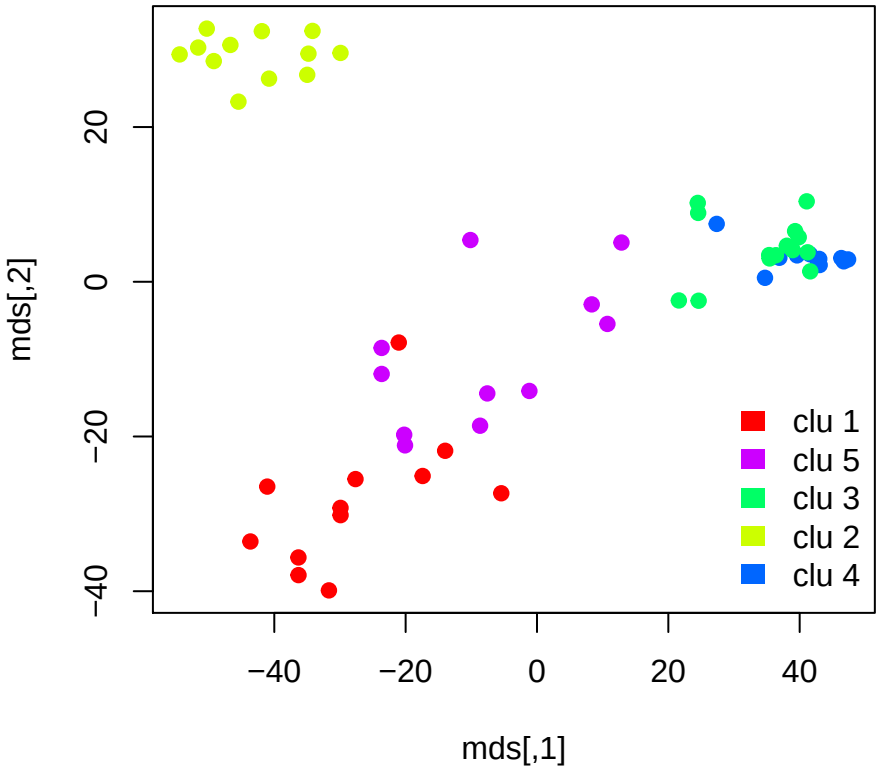
We can not visualize the clustering from partioning methods with a tree like we did for hierarchical clustering. Even if we can get the distances between patients the algorithm does not return the distances between clusters out of the box. However, if we had a way to visualize the distances between patients in 2 dimensions we could see the how patients and clusters relate each other. It turns out, that there is a way to compress between patient distances to a 2-dimensional plot. There are many ways to do this and we introduce these dimension reduction methods including the one we will use now later in this chapter. For now, we are going to use a method called "multi-dimensional scaling" and plot the patients in a 2D plot color coded by their cluster assignments.

```
# Calculate distances
dists=dist(t(mat))

# calculate MDS
mds=cmdscale(dists)

# plot the patients in the 2D space
plot(mds,pch=19,col=rainbow(5)[kclu$cluster])

# set the legend for cluster colors
legend("bottomright",
       legend=paste("clu",unique(kclu$cluster)),
       fill=rainbow(5)[unique(kclu$cluster)],
       border=NA,box.col=NA)
```



The plot we obtained shows the separetion between clusters. However, it does not do a great job showing the separation between cluster 3 and 4, which represent CML and "no leukemia" patients. We might need another dimension to properly visualize that separation. In addition, those two clusters were closely related in the hierarhical clustering as well.

3.4.4 how to choose "k", the number of clusters

Up to this point, we have avoided the question of selecting optimal number clusters. How do we know where to cut our dendrogram or which k to choose ? First of all, this is a difficult question. Usually, clusters have different granuality. Some clusters are tight and compact and some are wide,and both these types of clusters can be in the same data set. When visualized, some large clusters may look like they may have sub-clusters. So should we consider the large cluster as one cluster or should we consider the sub-clusters as individual clusters ? There are some metrics to

help but there is no definite answer. We will show a couple of them below.

3.4.4.1 Silhouhette

One way to determine how well the clustering is to measure the expected self-similar nature of the points in a set of clusters. The silhouette value does just that and it is a measure of how similar a data point is to its own cluster compared to other clusters. The silhouette value ranges from -1 to +1, where values that are positive indicates that the data point is well matched to its own cluster, if the value is zero it is a borderline case and if the value is minus it means that the data point might be mis-clustered because it is more simialr to a neighboring cluster. If most data points have a high value, then the clustering is appropriate. Ideally, one can create many different clusterings with different parameters such as k,number of clusters and assess their appropriateness using the average silhouette values. In R, silhouette values are refered to as silhouette widths in the documentation.

A silhouette value is calculated for each data point. In our working example, each patient will get silhouette values showing how well they are matched to their assigned clusters. Formally this calculated as follows. For each data point i, we calculate $a(i)$, which denotes the average distance between i and all other data points within the same cluster. This shows how well the point fits into that cluster. For the same data point, we also calculate $b(i)$ b(i) denotes the lowest average distance of i to all points in any other cluster, of which i is not a member. The cluster with this lowest average $b(i)$ is the "neighbouring cluster" of data point i since it is the next best fit cluster for that data point. Then, the silhouette value for a given data point is:

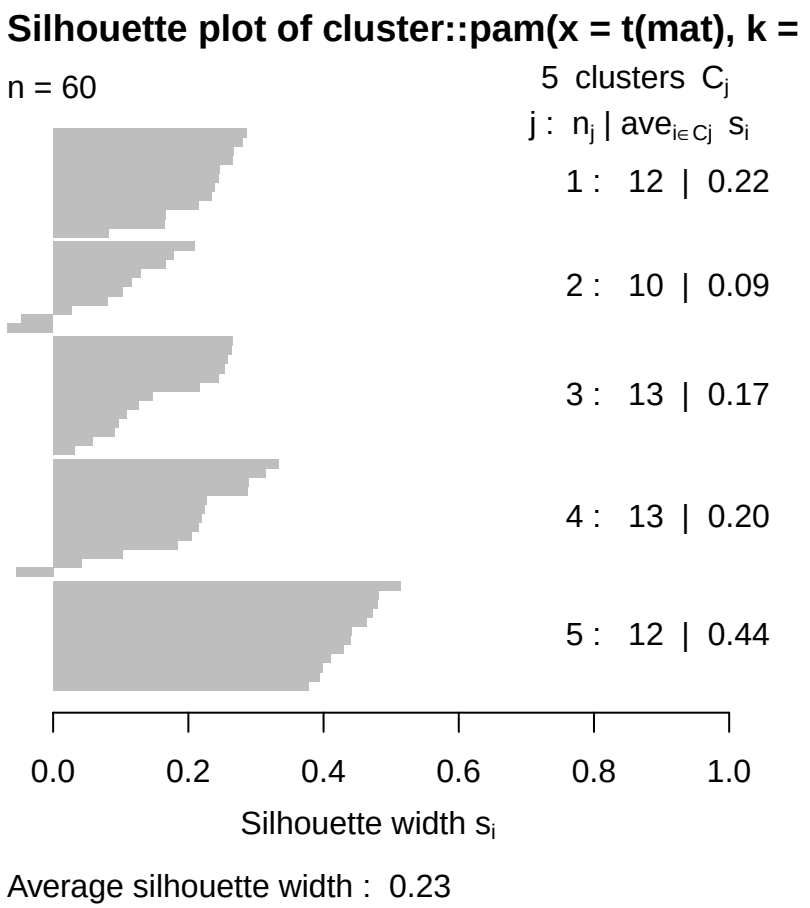
$$s(i) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$$

As described, this quantity is positive when $b(i)$ is high and $a(i)$ is low, meaning that the data point i is self-similar to its cluster. And the silhouette value, $s(i)$, is negative if it is more similar to its neighbours than its assigned cluster.

In R, we can calculate silhouette values using `cluster::silhouette()` func-

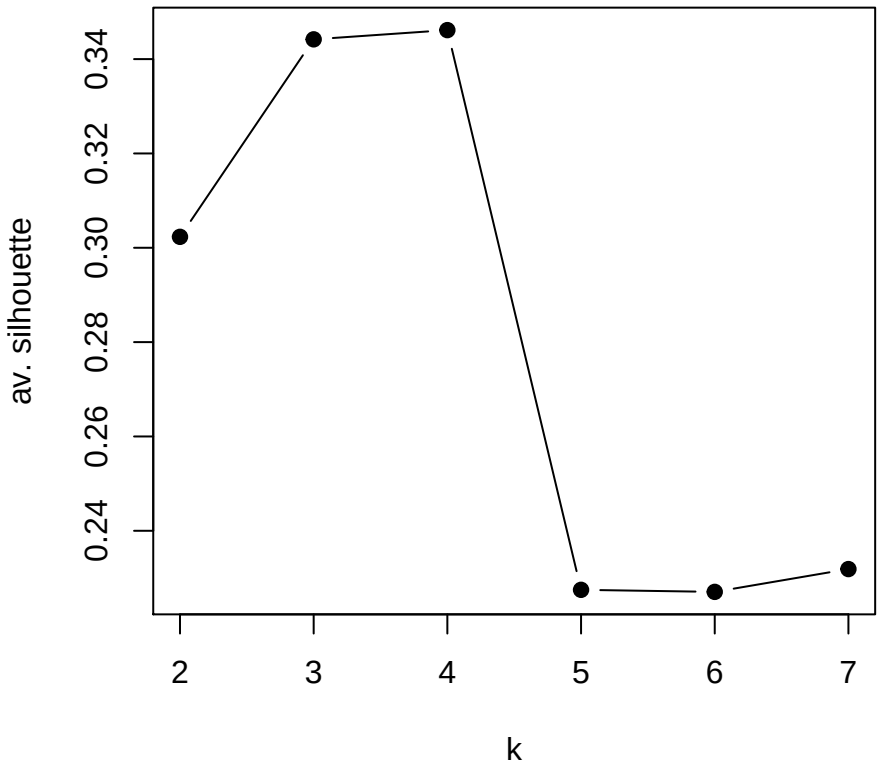
tion. Below, we calculate the silhouette values for k-medoids clustering with `pam()` function with `k=5`.

```
library(cluster)
set.seed(101)
pamclu=cluster::pam(t(mat),k=5)
plot(silhouette(pamclu),main=NULL)
```



Now, let us calculate average silhouette value different k values and compare. We will use `sapply()` function to get average silhouette values across k values between 2 and 7. Within `sapply()` there is an anonymous function that that does the clustering and calculates average silhouette values for each k.

```
Ks=sapply(2:7,
    function(i)
      summary(silhouette(pam(t(mat),k=i)))$avg.width)
plot(2:7,Ks,xlab="k",ylab="av. silhouette",type="b",
     pch=19)
```



In this case, it seems the best value for k is 4. The k-medoids function `pam()` will usually cluster CML and noLeukemia cases together when `k=4`, which are also related clusters according to hierarchical clustering we did earlier.

3.4.4.2 Gap statistic

As clustering aims to find self-similar data points, it would be reasonable to expect with the correct number of clusters the total within-cluster variation is minimized. Within-cluster variation for a single cluster can sim-

ply be defined as sum of squares from the cluster mean, which in this case is the centroid we defined in k-means algorithm. The total within-cluster variation is then sum of within-cluster variations for each cluster. This can be formally defined as follows:

$$W_k = \sum_{k=1}^{K} \sum_{\mathrm{x}_i \in C_k} (\mathrm{x}_i - \mu_k)^2$$

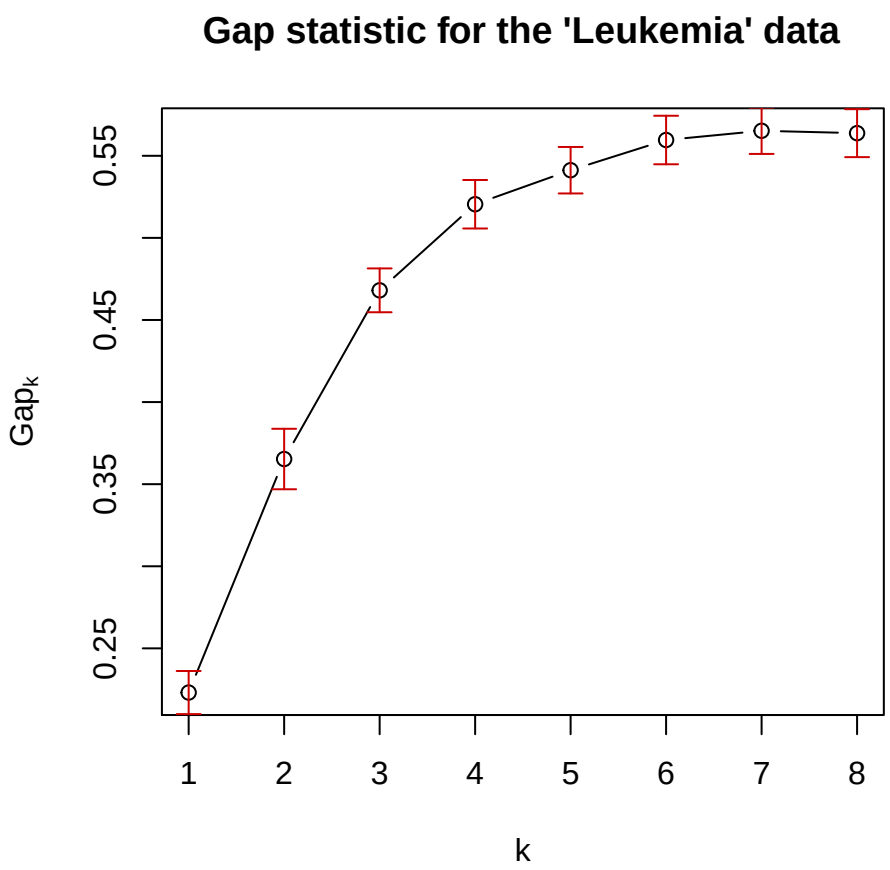
Where x_i is data point in cluster k, and μ_k is the cluster mean, and W_k is the total within-cluster variation quantity we described. However, the problem is that the variation quantity decreases with number of clusters. The more centroids we have, the smaller the distances to the centroids get. A more reliable approach would be somehow calculating the expected variation from a reference null distribution and compare that to the observed variation for each k. In gap statistic approach, the expected distribution is calculated via sampling points from the boundaries of the original data and calculating within-cluster variation quantity for multiple rounds of sampling. This way we have an expectation how about the variability when there is no expected clustering, and then compare that expected variation to the observed within-cluster variation. The expected variation should also go down with increasing number of clusters, but for the optimal number of clusters the expected variation will be furthest away from observed variation. This distance is called the "gap statistic" and defined as follows: $\mathrm{Gap}_n(k) = E_n^*\{\log W_k\} - \log W_k$, where $E_n^*\{\log W_k\}$ is the expected variation in log-scale under a sample size n from the reference distribution and $\log W_k$ is the observed variation. Our aim is choose the k, number of clusters, that maximizes $\mathrm{Gap}_n(k)$.

We can easily calculate the gap statistic with `cluster::clusGap()` function. We will now use that function to calculate the gap statistic for our patient gene expression data.

```
library(cluster)
set.seed(101)
# define the clustering function
pam1 <- function(x,k)
  list(cluster = pam(x,k, cluster.only=TRUE))
```

```
# calculate the gap statistic
pam.gap= clusGap(t(mat), FUN = pam1, K.max = 8,B=50)

# plot the gap statistic accross k values
plot(pam.gap, main = "Gap statistic for the 'Leukemia' data")
```



In this case, gap statistic shows $k = 7$ is the best. However, after $K = 6$ the statistic has more or less a stable curve. In this case, we know that there are 5 main patient categories but this does not mean there is no sub-categories or sub-types for the cancers we are looking at.

https://statweb.stanford.edu/~gwalther/gap

3.4.4.3 Other methods

There are several other methods that provide insight into how many clusters. In fact, the package `NbClust` provides 30 different ways to determine the number of optimal clusters and can offer a voting mechanism to pick the best number. Below, we are showing how to use this function for some of the optimal number of cluster detection methods.

```
library(NbClust)
nb = NbClust(data=t(mat),
             distance = "euclidean", min.nc = 2,
        max.nc = 7, method = "kmeans",
        index=c("kl","ch","cindex","db","silhouette",
                "duda","pseudot2","beale","ratkowsky",
                "gap","gamma","mcclain","gplus",
                "tau","sdindex","sdbw"))

table(nb$Best.nc[1,]) # consensus seems to be 3 clusters
```

However, the readers should keep in mind that clustering is an exploratory technique. If you have solid labels for your data points maybe clustering is just a sanity check, and you should just do predictive modeling instead. However, in biology there are rarely solid labels and things have different granularity. Take the leukemia patients case we have been using for example, it is know that leukemia types have subtypes and those sub-types that have different mutation profiles and consequently have different molecular signatures. Because of this, it is not surprising that some optimal cluster number techniques will find more clusters to be appropriate. On the other hand, CML (Chronic myeloid leukemia) is a slow progressing disease and maybe as molecular signatures goes could be the closest to no leukemia patients, clustering algorithms may confuse the two depending on what granuality they are operating with. It is always good to look at the heatmaps after clustering, if you have meaningful self-similar data points even if the labels you have do not agree that there can be different clusters you can perform downstream analysis to understand the sub-clusters better. As we have seen, we can estimate optimal number of clusters but we can not take that estimation as the absolute truth, given

more data points or different set of expression signatures you may have different optimal clusterings, or the supposed optimal clustering might overlook previously known sub-groups of your data.

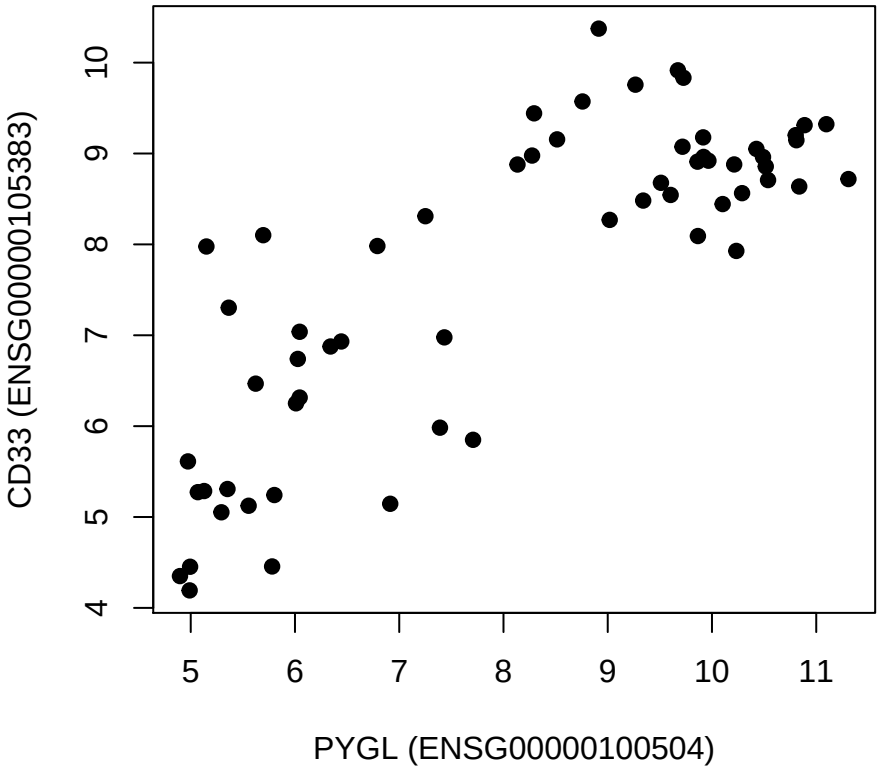
3.5 Dimensionality reduction techniques: visualizing complex data sets in 2D

In statistics, dimension reduction techniques are a set of processes for reducing the number of random variables by obtaining a set of principal variables. For example, in the context of a gene expression matrix accross different patient samples, this might mean getting a set of new variables that cover the variation in sets of genes. This way samples can be represented by a couple of principal variables instead of thousands of genes. This is useful for visualization, clustering and predictive modeling.

3.5.1 Principal component analysis

Principal component analysis (PCA) is maybe the most popular technique to examine high-dimensional data. There are multiple interpretations of how PCA reduces dimensionality. We will first focus on geometrical interpretation, where this operation can be interpreted as rotating the orignal dimensions of the data. For this, we go back to our example gene expression data set. In this example, we will represent our patients with expression profiles of just two genes, CD33 (ENSG00000105383) and PYGL (ENSG00000100504) genes. This way we can visualize them in a scatterplot.

```
plot(mat[rownames(mat)=="ENSG00000100504",],
     mat[rownames(mat)=="ENSG00000105383",],pch=19,
     ylab="CD33 (ENSG00000105383)",
     xlab="PYGL (ENSG00000100504)")
```



PCA rotates the original data space such that the axes of the new coordinate system point into the directions of highest variance of the data. The axes or new variables are termed principal components (PCs) and are ordered by variance: The first component, PC 1, represents the direction of the highest variance of the data. The direction of the second component, PC 2, represents the highest of the remaining variance orthogonal to the first component. This can be naturally extended to obtain the required number of components which together span a component space covering the desired amount of variance. In our toy example with only two genes, the principal componets are drawn over the original scatter plot and in the next plot we show the new coordinate system based on the pricinpal components. We will calculate the PCA with the `princomp()` function, this function returns the new coordinates as well. These new coordinates are simply a projection of data over the new coordinates. We will decorate the scatter plots with eigenvectors showing the direction of greatest variation. Then, we will plot the new coordinates. These are automatically calculated by `princomp()` function. Notice that we are using

the `scale()` function when plotting coordinates and also before calculating PCA. This function centers the data, meaning substracts the mean of the each column vector from the elements in the vector. This essentially gives the columns a zero mean. It also divides the data by the standard deviation of the centered columns. These two operations helps bring the data to a common scale which is important for PCA not to be affected by different scales in the data.

```
par(mfrow=c(1,2))

# create the subset of the data with two genes only
# notice that we transpose the matrix so samples are
# on the columns
sub.mat=t(mat[rownames(mat) %in% c("ENSG00000100504","ENSG00000105383"),])

# ploting our genes of interest as scatter plots
plot(scale(mat[rownames(mat)=="ENSG00000100504",]),
     scale(mat[rownames(mat)=="ENSG00000105383",]),
     pch=19,
     ylab="CD33 (ENSG00000105383)",
     xlab="PYGL (ENSG00000100504)",
     col=annotation_col$LeukemiaType,
     xlim=c(-2,2),ylim=c(-2,2))

# create the legend for the Leukemia types
legend("bottomright",
       legend=unique(annotation_col$LeukemiaType),
       fill =palette("default"),
       border=NA,box.col=NA)

# calculate the PCA only for our genes and all the samples
pr=princomp(scale(sub.mat))


# plot the direction of eigenvectors
# pr$loadings returned by princomp has the eigenvectors
arrows(x0=0, y0=0, x1 = pr$loadings[1,1],
```

```
         y1 = pr$loadings[2,1],col="pink",lwd=3)
arrows(x0=0, y0=0, x1 = pr$loadings[1,2],
         y1 = pr$loadings[2,2],col="gray",lwd=3)


# plot the samples in the new coordinate system
plot(-pr$scores,pch=19,
     col=annotation_col$LeukemiaType,
     ylim=c(-2,2),xlim=c(-4,4))

# plot the new coordinate basis vectors
arrows(x0=0, y0=0, x1 =-2,
         y1 = 0,col="pink",lwd=3)
arrows(x0=0, y0=0, x1 = 0,
         y1 = -1,col="gray",lwd=3)
```

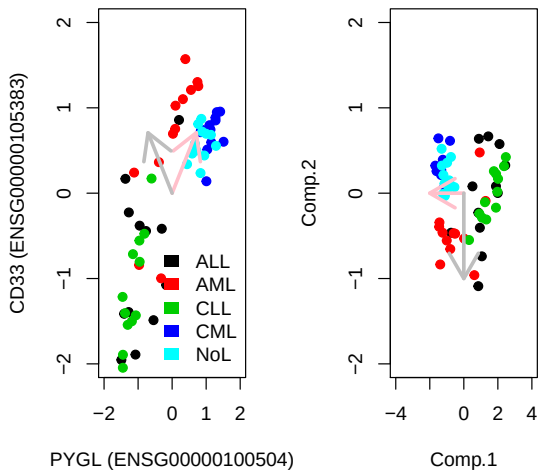


FIGURE 3.21: Geometric interpretation of PCA finding eigenvectors that point to direction of highest variance. Eigenvectors can be used as a new coordinate system.

As you can see, the new coordinate system is useful by itself.The X-axis which represents the first component separates the data along the lymphoblastic and myeloid leukemias.

PCA in this case is obtained by calculating eigenvectors of the covariance matrix via an operation called eigen decomposition. Covariance matrix is obtained by covariance of pairwise variables of our expression matrix, which is simply $\text{cov}(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_X)(y_i - \mu_Y)$, where X and Y expression values of genes in a sample in our example. This is a measure of how things vary together, if high expressed genes in sample A are also highly expressed in sample B and lowly expressed in sample A are also lowly expressed in sample B, then sample A and B will have positive covariance. If the opposite is true then they will have negative covariance. This quantity is related to correlation and in fact correlation is standardized covariance. Covariance of variables can be obtained with `cov()` function, and eigen decomposition of such a matrix will produce a set of ortahogonal vectors that span the directions of highest variation. In 2D, you can think of this operation as rotating two perpendicular lines together until they point to the directions where most of the variation in the data lies on, similar to the figure 3.21. An important intuition is that, after the rotation prescribed by eigenvectors is complete the covariance between variables in this rotated dataset will be zero. There is a proper mathematical relationship between covariances of the rotated dataset and the original dataset. That's why operating on covariance matrix is related to the rotation of the original dataset.

```
cov.mat=cov(sub.mat) # calculate covariance matrix
cov.mat
eigen(cov.mat) # obtain eigen decomposition for eigen values and vectors
```

Eigenvectors and eigenvalues of the covariance matrix indicates the direction and the magnitute of variation of the data. In our visual example the eigenvectors are so-called principal components. The eigenvector indicates the direction and the eigen values indicate the variation in that direction. Eigenvectors and values exist in pairs: every eigenvector has a corresponding eigenvalue and the eigenvectors are linearly independent from each other, this means they are orthogonal or uncorrelated in the our working example above. The eigenvectors are ranked by their corresponding eigen value, the higher the eigen value the more important the eigenvector is, because it explains more of the variation compared to the other eigenvectors. This feature of PCA makes the dimension reduction

possible. We can sometimes display data sets that have many variables only in 2D or 3D because the these top eigenvectors are sometimes enough to capture most of variation in the data.

3.5.1.1 Singular value decomposition and principal component analysis

A more common way to calculate PCA is through something called singular value decomposition (SVD). This results in another interpretation of PCA, which is called "latent factor" or "latent component" interpretation. In a moment, it will be more clear what we mean by "latent factors". SVD is a matrix factorization or decomposition algorithm that decomposes an input matrix,X, to three matrices as follows: $\mathrm{X} = USV^T$. In essence many matrices can be decomposed as a product of multiple matrices and we will come to other techniques later in this chapter. Singular Value Decomposition is shown in figure 3.22. U is the matrix with eigenarrays on the columns and this has the same dimensions as the input matrix, you might see elsewhere the columns are named as eigenassays. S is the matrix that contain the singular values on the diagonal. The singular values are also known as eigenvalues and their square is proportional to explained variation by each eigenvector. Finally, the matrix V^T contains the eigenvectors on its rows. It is interpretation is still the same. Geometrically, eigenvectors point to the direction of highest variance in the data. They are uncorrolated or geometrically orthogonal to each other. These interpretations are identical to the ones we made before. The slight difference is that the decomposition seem to output V^T which is just the transpose of the matrix V. However, the SVD algorithms in R usually return the matrix V. If you want the eigenvectors, you either simply use the columns of matrix V or rows of V^T.

One thing that is new in the figure 3.22 is the concept of eigenarrays. The eigenarrays or sometimes called eigenassays reprensent the sample space and can be used to plot the relationship between samples rather than genes. In this way, SVD offers additional information than the PCA using the covariance matrix. It offers us a way to summarize both genes and samples. As we can project the gene expression profiles over the top two eigengenes and get a 2D representation of genes, but with SVD we can also project the samples over the the top two eigenarrays and get a representation of samples in 2D scatterplot. Eigenvector could represent inde-

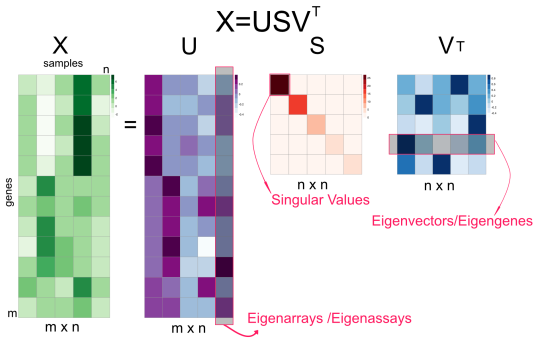


FIGURE 3.22: Singular Value Decomposition (SVD) explained in a diagram.

pendent expression programs across samples, such as cell-cycle if we had time-based expression profiles. However, there is no guarantee that each eigenvector will be biologically meaningful. Similarly each eigenarray represent samples with specific expression characteristics. For example, the samples that have a particular pathway activated might be corrolated to an eigenarray returned by SVD.

Previously, in order to map samples to the reduced 2D space we had to transpose the genes-by-samples matrix when using `princomp()` function. We will now first use SVD on genes-by-samples matrix to get eigenarrays and use that to plot samples on the reduced dimensions. We will project the columns in our original expression data on eigenarrays and use the first two dimensions in the scatter plot. If you look at the code you will see that for the projection we use $U^T X$ operation, which is just V^T if you follow the linear algebra. We will also perform the PCA this time with `prcomp()` function on the transposed genes-by-samples matrix to get a similar information, and plot the samples on the reduced coordinates.

```
par(mfrow=c(1,3))
d=svd(scale(mat)) # apply SVD
assays=t(d$u) %*% scale(mat) # projection on eigenassays
plot(assays[1,],assays[2,],pch=19,
     col=annotation_col$LeukemiaType)
#plot(d$v[,1],d$v[,2],pch=19,
```

```
#     col=annotation_col$LeukemiaType)
pr=prcomp(t(mat),center=TRUE,scale=TRUE) # apply PCA on transposed matrix

# plot new coordinates from PCA, projections on eigenvectors
# since the matrix is transposed eigenvectors represent
plot(pr$x[,1],pr$x[,2],col=annotation_col$LeukemiaType)
```

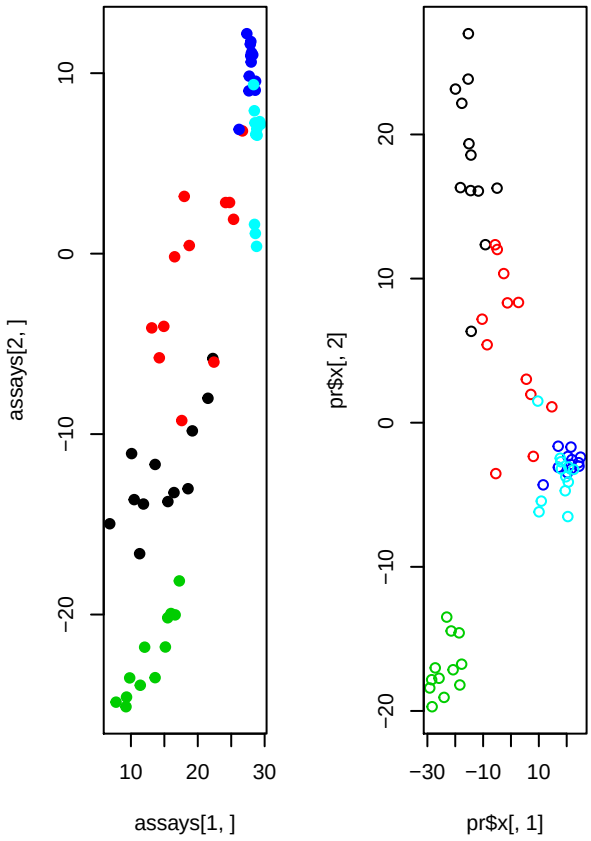


FIGURE 3.23: SVD on matrix and its transpose

As you can see in the figure 3.23, the two approaches yield separation of samples, although they are slightly different. The difference comes from the centering and scaling. In the first case, we scale and center columns and the second case we scale and center rows since the matrix is trans-

posed. If we do not do any scaling or centering we would get identical plots.

3.5.1.1.1 Eigenvectors as latent factors/variables

Finally, we can introduce the latent factor interpretation of PCA via SVD. As we have already mentioned eigenvectors can also be interpreted as expression programs that are shared by several genes such as cell cycle expression program when measuring gene expression accross samples taken in different time points. In this intrepretation, linear combination of expression programs makes up the expression profile of the genes. Linear combination simply means multiplying the expression program with a weight and adding them up. Our USV^T matrix multiplication can be rearranged to yield such an understanding, we can multiply eigenarrays U with the diagonal eigenvalues S, to produce a m-by-n weights matrix called W, so $W = US$ and we can re-write the equation as just weights by eigenvectors matrix, $X = WV^T$ as shown in figure 3.24.

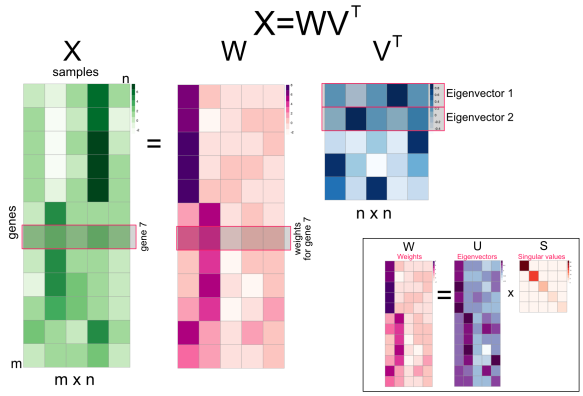


FIGURE 3.24: Singular Value Decomposition (SVD) reorgonized as multiplication of m-by-n weights matrix and eigenvectors

This simple transformation now makes it clear that indeed if eigenvectors are representing expression programs, their linear combination is making up individual gene expression profiles. As an example, we can show the liner combination of the first two eigenvectors can approximate the

expression profile of an hypothetical gene in the gene expression matrix. The figure 3.25 shows eigenvector 1 and eigenvector 2 combined with certain weights in W matrix can approximate gene expression pattern our example gene.

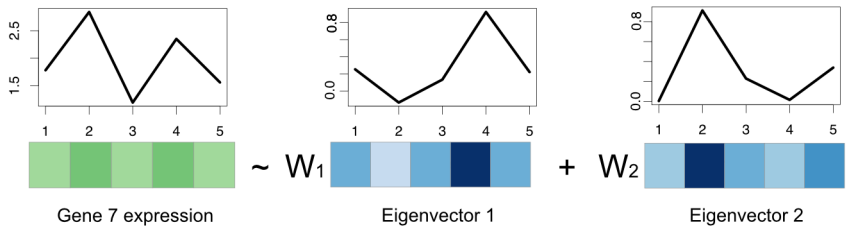


FIGURE 3.25: Gene expression of a gene can be thought as linear combination of eigenvectors.

However, SVD does not care about biology. The eigenvectors are just obtained from the data with constraints of ortagonality and the direction of variation. There are examples of eigenvectors representing real expression programs but that does not mean eigenvectors will always be biologically meaningful. Sometimes combination of them might make more sense in biology than single eigenvectors. This is also the same for the other matrix factorization techniques we describe below.

3.5.2 Other dimension reduction techniques using other matrix factorization methods

We must mention a few other techniques that are similar to SVD in spirit. Remember we mentioned that every matrix can be decomposed to other matrices where matrix multiplication operations reconstruct the original matrix. In the case of SVD/PCA, the constraint is that eigenvectors/arrays are ortogonal, however there are other decomposition algorithms with other constraints.

3.5.2.1 Independent component analysis (ICA)

We will first start with independent component analysis (ICA) which is an extension of PCA. ICA algorithm decomposes a given matrix X as follows: $X = SA$. The rows of A could be interpreted similar to the eigengenes and columns of S could be interpreted as eigenarrays, these components are sometimes called metagenes and metasamples in the literature. Traditionally, S is called source matrix and A is called mixing matrix. ICA is developed for a problem called "blind-source separation". In this problem, multiple microphones record sound from multiple instruments, and the task is disentagle sounds from original instruments since each microphone is recording a combination of sounds. In this respect, the matrix S contains the original signals (sounds from different instruments) and their linear combinations identified by the weights in A, and the product of A and S makes up the matrix X, which is the observed signal from different microphones. With this interpretation in mind, if the interest is strictly expression patterns similar that represent the hidden expression programs we see that genes-by-samples matrix is transposed to a samples-by-genes matrix, so that the columns of S represent these expression patterns , here refered to as "metagenes", hopefully representing distinct expression programs (Figure 3.26).

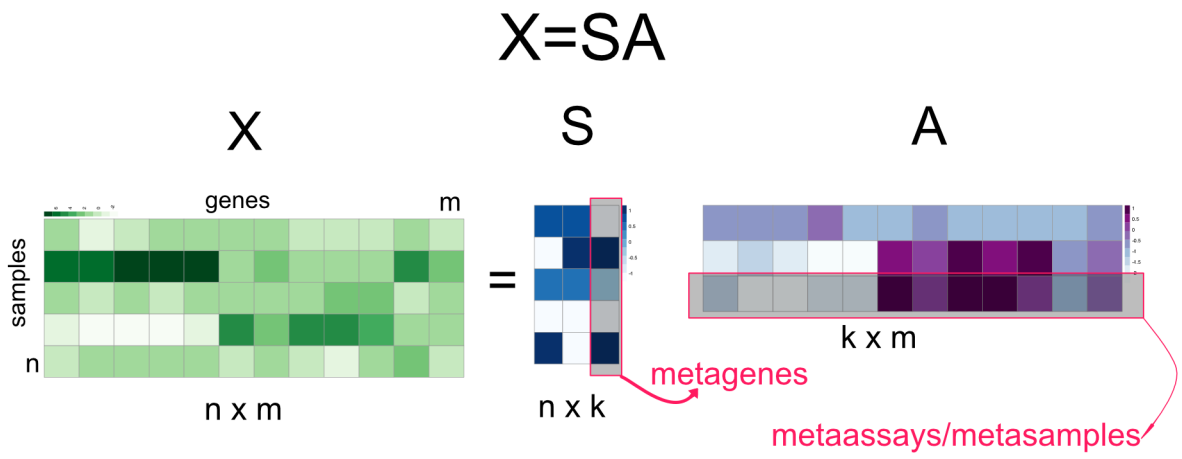


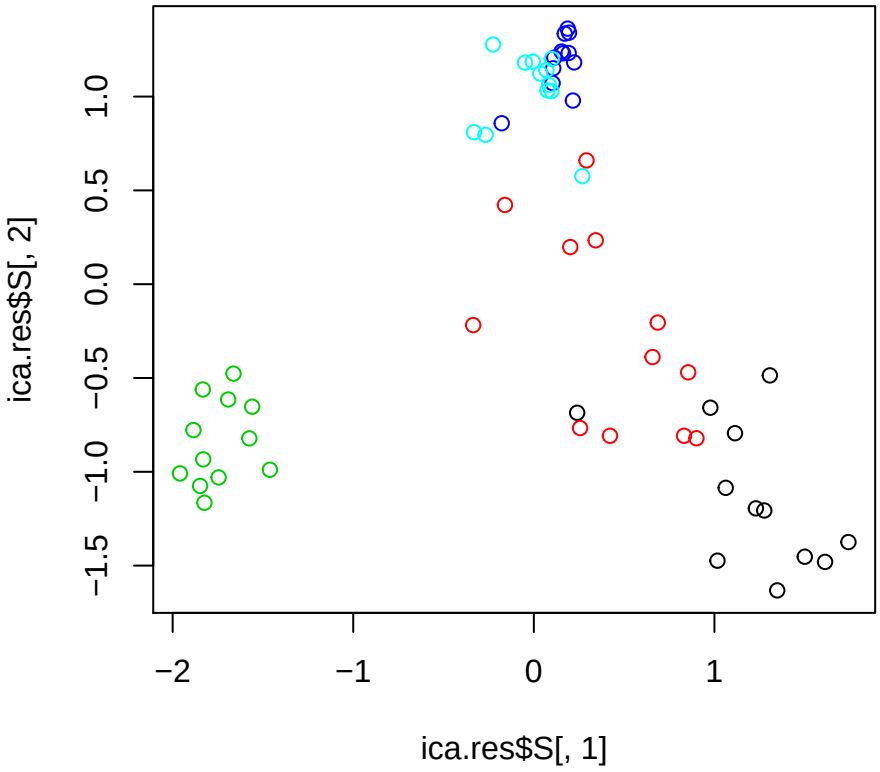
FIGURE 3.26: Independent Component Analysis (ICA)

ICA requires that the columns of S matrix, the "metagenes" in our ex-

ample above to be statistical independent. This is a stronger constraint than uncorrelatedness. In this case, there should be no relationship between non-linear transformation of the data either. There are different ways of ensuring this statistical indepedence and this is the main constraint when finding the optimal A and S matrices. The various ICA algorithms use different proxies for statistical independence, and the definition of that proxy is the main difference between many ICA algorithms. The algorithm we are going to use requires that metagenes or sources in the S matrix are non-gaussian as possible. Non-gaussianity is shown to be related to statistical independence [REF]. Below, we are using `fastICA::fastICA()` function to extract 2 components and plot the rows of matrix A which represents metagenes. This way, we can visualize samples in a 2D plot. If we wanted to plot the relationship between genes we would use the the columns of matrix S.

```
library(fastICA)
ica.res=fastICA(t(mat),n.comp=2) # apply ICA

# plot reduced dimensions
plot(ica.res$S[,1],ica.res$S[,2],col=annotation_col$LeukemiaType)
```



3.5.2.2 Non-negative matrix factorization (NMF)

Non-negative matrix factorization algorithms are series of algorithms that aim to decompose the matrix X into the product or matrices W and H, $X = WH$ (Figure 3.27). The constraint is that W and H must contain non-negative values, so must X. This is well suited for data sets that can not contain negative values such as gene expression. This also implies addivity of components, in our example expression of a gene across samples are addition of multiple metagenes. Unlike ICA and SVD/PCA, the metagenes can never be combined in subtractive way. In this sense, expression programs potentially captured by metagenes are combined additively.

The algorithms that compute NMF tries to minimize the cost function $D(X, WH)$, which is the distance between X and WH. The early algorithms just use the euclidean distance which translates to $\sum(X-WH)^2$, this is also known as Frobenious norm and you will see in the literature it is written as :$\|\|V - WH\|\|_F$ However this is not the only distance met-

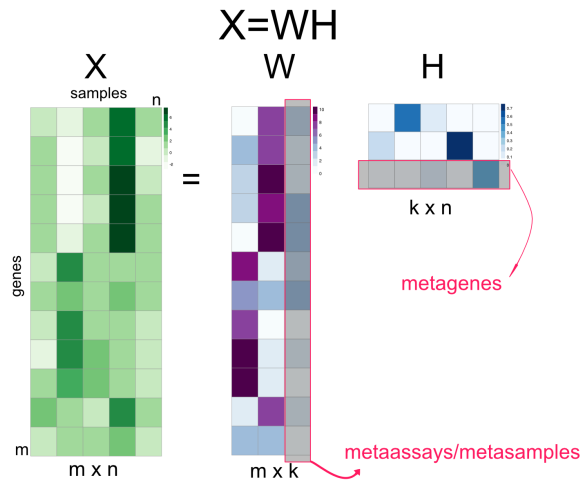


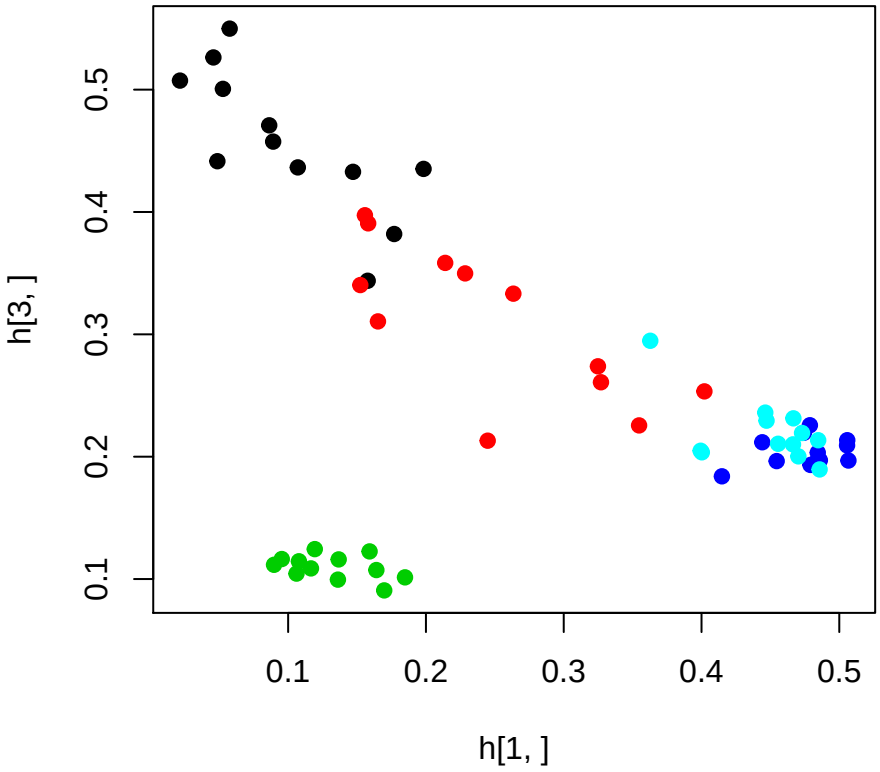
FIGURE 3.27: Non-negative matrix factorization

ric, other distance metrics are also used in NMF algorithms. In addition, there could be other parameters to optimize that relates to sparseness of the W and H matrices. With sparse W and H, each entry in the X matrix is expressed as the sum of a small number of components. This makes the interpretation easier, if the weights are 0 than there is not contribution from the corresponding factors.

Below, we are plotting the values of metagenes (rows of H) for component 1 and 3. In this context, these values can also be interpreted as relationship between samples. If we wanted to plot the relationship between genes we would plot the columns of W matrix.

```
library(NMF)
res=nmf(mat,rank=3,seed=123) # nmf with 3 components/factors
w <- basis(res) # get W
h <- coef(res)  # get H

# plot 1st factor against 3rd factor
plot(h[1,],h[3,],col=annotation_col$LeukemiaType,pch=19)
```



We should add the note that due to random starting points of the optimization algorithm, NMF is usually run multiple times and a consensus clustering approach is used when clustering samples. This simply means that samples are clustered together if they cluster together in multiple runs of the NMF. The NMF package we used above has built-in ways to achieve this. In addition, NMF is a family of algorithms the choice of cost function to optimize the difference between X and WH and the methods used for optimization creates multiple variants of NMF. The "method" parameter in the above `nmf()` function controls the which algorithm for NMF.

3.5.2.3 chosing the number of components and ranking components in importance

In both ICA and NMF, there is no well-defined way to rank components or to select the number of components. There are couple of approaches that might suit to both ICA and NMF for ranking components. One can use the norms of columns/rows in mixing matrices. This could simply mean

take the sum of absolute values in mixing matrices. In our examples above, For our ICA example above, ICA we would take the sum of the absolute values of the rows of A since we transposed the input matrix X before ICA. And for the NMF, we would use the columns of W. These ideas assume that the larger coefficients in the weight or mixing matrices indicate more important components.

For selecting the optimal number of components, NMF package provides different strategies. One way is to calculate RSS for each k, number of components, and take the k where the RSS curve starts to stabilize.However, these strategies require that you run the algorithm with multiple possible component numbers. `nmf` function will run these automatically when the `rank` argument is a vector of numbers. For ICA there is no straightforward way to choose the right number of components, a common strategy is to start with as many components as variables and try to rank them by their usefullness.

Want to know more ?

NMF package vignette has extensive information on how to run NMF to get stable resuts and getting an estimate of components `https://cran.r-project.org/web/packages/NMF/vignettes/NMF-vignette.pdf`

3.5.3 Multi-dimensional scaling

MDS is a set of data analysis techniques that display the structure of distance data in a high dimensional space into a lower dimensional space without much loss of information. The overall goal of MDS is to faithfully represent these distances with the lowest possible dimensions. So called "classical multi-dimensional scaling" algorithm, tries to minimize the following function:

$$Stress_D(z_1, z_2, ..., z_N) = \left(\frac{\sum_{i,j} \left(d_{ij} - \|z_i - z_j\|\right)^2}{\sum_{i,j} d_{ij}^2} \right)^{1/2}$$

Here the function compares the new data points on lower dimension $(z_1, z_2, ..., z_N)$ to the input distances between data points or distance be-

tween samples in our gene expression example. It turns out, this problem can be efficiently solved with SVD/PCA on the scaled distance matrix, the projection on eigenvectors will be the most optimal solution for the equation above. Therefore, classical MDS is sometimes called Principal Coordinates Analysis in the litereuature. However, later variants improve on classical MDS this by using this as a starting point and optimize a slightly different cost function that again measures how well the low-dimensional distances correspond to high-dimensional distances. This variant is called non-metric MDS and due to the nature of the cost function, it assumes a less stringent relationship between the low-dimensional distances $|z_{i}-z_{j}| and input distances d_{ij}. Formally, this procedure tries to optimize the following function.

$$Stress_D(z_1, z_2, ..., z_N) = \left(\frac{\sum_{i,j} \left(\|z_i - z_j\| - \theta(d_{ij}) \right)^2}{\sum_{i,j} \|z_i - z_j\|^2} \right)^{1/2}$$

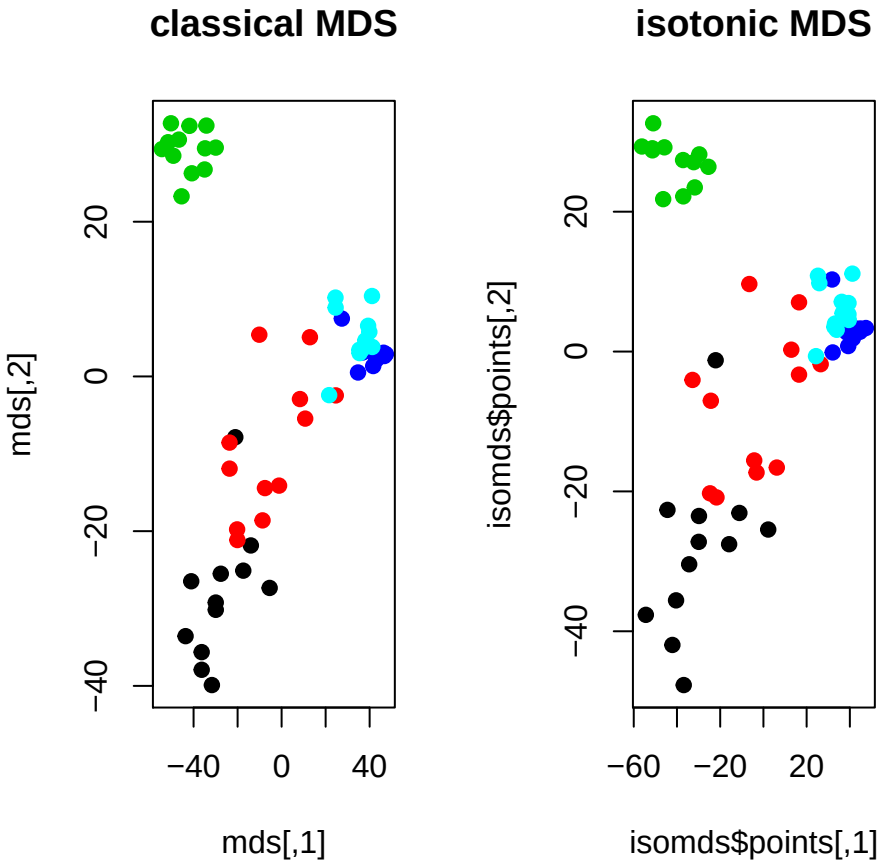
The core of a non-metric MDS algorithm is a twofold optimization process. First the optimal monotonic transformation of the distances has to be found, this is shown in the above formula as $\theta(d_{ij})$. Secondly, the points on a low dimension configuration have to be optimally arranged, so that their distances match the scaled distances as closely as possible. This two steps are repeated until some convergence criteria is reached. This usually means that the cost function does not improve much after certain number of iterations. The basic steps in a non-metric MDS algorithm are: 1. Find a random low dimensional configuration of points, or in the variant we will be using below we start with the configuration returned by classical MDS 2. Calculate the distances between the points in the low dimension $|z_{i}-z_{j}|, z_i and z_j are vector of positions for sample i and j. 3. Find the optimal monotonic transformation of the input distance, $\theta(d_{ij})$, to approximate input distances to low-dimensional distances. This is achieved by isotonic regression, where a monotonically increasing free-form function is fit. This step practically ensures that ranking of low-dimensional distances are similar to rankings of input distances. 4. Minimize the stress function by re-configuring low-dimensional space and keeping θ function constant. 5. repeat from step 2 until convergence.

We will now demonstrate both classical MDS and Kruskal's isometric MDS.

```
mds=cmdscale(dist(t(mat)))
isomds=MASS::isoMDS(dist(t(mat)))
```

```
## initial  value 15.907414
## final  value 13.462986
## converged
```

```
# plot the patients in the 2D space
par(mfrow=c(1,2))
plot(mds,pch=19,col=annotation_col$LeukemiaType,
     main="classical MDS")
plot(isomds$points,pch=19,col=annotation_col$LeukemiaType,
     main="isotonic MDS")
```



In this example, there is not much difference between isotonic MDS and classical MDS. However, there might be cases where different MDS methods provides visiable changes in the scatter plots.

3.5.4 t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE maps the distances in high-dimensional space to lower dimensions and it is similar to MDS method in this respect. But the benefit of this particular method is that it tries to preserve the local structure of the data so the distances and grouping of the points we observe in a lower dimensions such as a 2D scatter plot is as close as possible to the distances we observe in the high-dimensional space. As with other dimension reduction methods, you can choose how many lower dimensions you need. The main difference of t-SNE is that it tries to preserve the local structure of the data.

This kind of local structure embedding is missing in the MDS algorithm which also has a similar goal. MDS tries to optimize the distances as a whole, whereas t-SNE optimizes the distances with the local structure in mind. This is defined by the "perplexity" parameter in the arguments. This parameter controls how much the local structure influences the distance calculation. The lower the value the more the local structure is take into account. Similar to MDS, the process is an optimization algorithm. Here, we also try to minimize the divergence between observed distances and lower dimension distances. However, in the case of t-SNE, the observed distances and lower dimensional distances are transformed using a probabilistic framework with their local variance in mind.

From here on, we will provide a bit more detail on how the algorithm works in case conceptual description above is too shallow. In t-SNE the euclidiean distances between data points are transformed into a conditional similarity between points. This is done by assuming a normal distribution on each data point with a variance calculated ultimately by the use of "perplexity" parameter. The perplexity paramater is, in a sense, a guess about the number of the closest neighbors each point has. Setting it to higher values gives more weight to global structure. Given d_{ij} is the euclidean distance between point i and j, the similarity score p_{ij} is calculated as shown below.

$$p_{j|i} = \frac{\exp(-\|d_{ij}\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|d_{ik}\|^2/2\sigma_i^2)}$$

This distance is symmetrized by incorparating $p_{i | j} as shown below.

$$p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$$

For the distances in the reduced dimension, we use t-distribution with one degree of freedom. In the formula below, $|y_i - y_j\|^2$ is euclidean distance between points i and j in the reduced dimensions.

$$q_{ij} = \frac{(1+\|y_i - y_j\|^2)^{-1}}{(\sum_{k\neq l} 1+\|y_k - y_l\|^2)^{-1}}$$

As most of the algorithms we have seen in this section, t-SNE is an optimization process in essence. In every iteration the points along lower dimensions are re-arranged to minimize the formulated difference be-

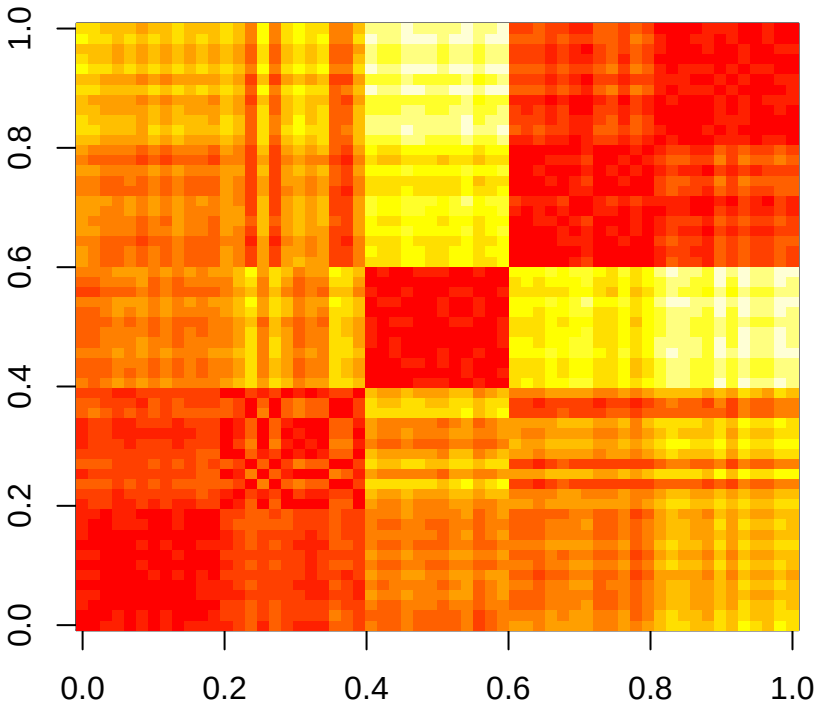
tween the the observed joint probabilities (p_{ij}) and low-dimensional joint probabilities (q_{ij}). Here we are trying to compare probability distributions. In this case, this is done using a method called Kullback-Leibler divergence, or KL-divergence. In the formula below, since the p_{ij} is predefined using original distances, only way to optimize is to play with q_{ij}) because it depends on the configuration of points in the lower dimensional space. This configuration is optimized to minimize the KL-divergence between p_{ij} and q_{ij}.

$$KL(P||Q) = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

Strictly speaking, KL-divergence measures how well the distribution P which is observed using the original data points can be aproximated by distribution Q, which is modeled using points on the lower dimension. If the distributions are identical KL-divergence would be 0. Naturally, the more divergent the distributions are the higher the KL-divergence will be.

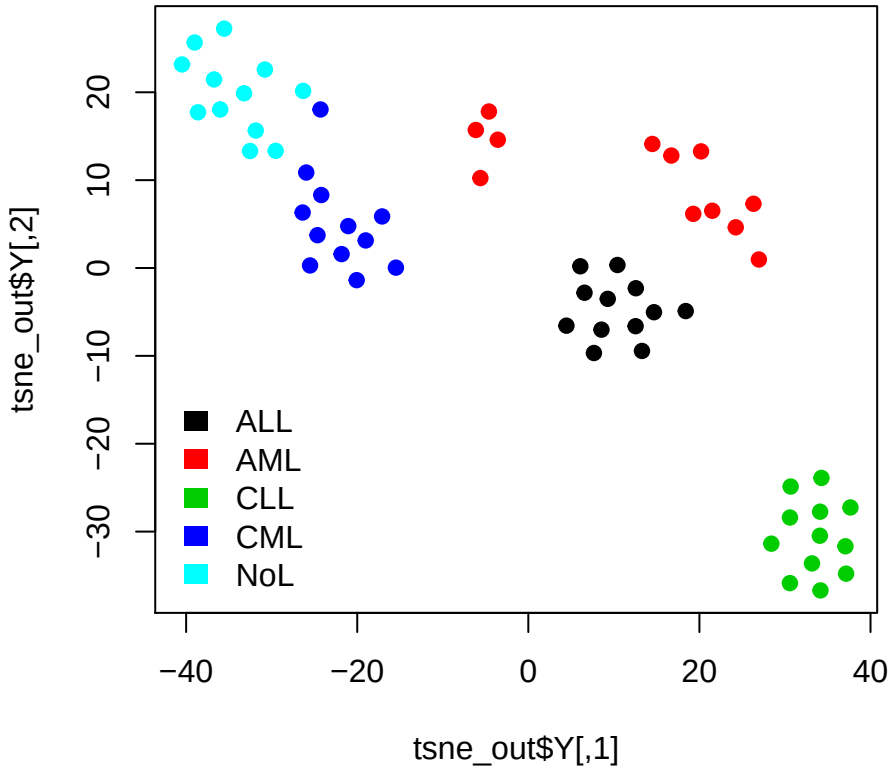
We will now show how to use t-SNE on our gene expression data set. We are setting the random seed because again t-SNE optimization algorithm have random starting points and this might create non-identical results in every run. After calculating the t-SNE lower dimension embeddings we will plot the points in a 2D scatter plot.

```
library("Rtsne")
set.seed(42) # Set a seed if you want reproducible results
tsne_out <- Rtsne(t(mat),perplexity = 10) # Run TSNE
 image(t(as.matrix(dist(tsne_out$Y))))
```




```
# Show the objects in the 2D tsne representation
plot(tsne_out$Y,col=annotation_col$LeukemiaType,
     pch=19)

# create the legend for the Leukemia types
legend("bottomleft",
       legend=unique(annotation_col$LeukemiaType),
       fill =palette("default"),
       border=NA,box.col=NA)
```



As you might have noticed, we set again a random seed with `set.seed()` function. The optimization algorithm starts with random configuration of points in the lower dimension space, and each iteration it tries to improve on the previous lower dimension confugration, that is why starting points can result in different final outcomes.

Want to know more ?

- How perplexity effects t-sne, interactive examples https://distill.pub/2016/misread-tsne/
- more on perplexity: https://blog.paperspace.com/dimension-reduction-with-t-sne/
- Intro to t-SNE https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm

##Exercises

3.5.5 How to summarize collection of data points: The idea behind statistical distributions

3.5.5.1

Calculate the means and variances of the rows of the following simulated data set, plot the distributions of means and variances using `hist()` and `boxplot()` functions.

```
set.seed(100)

#sample data matrix from normal distribution
gset=rnorm(600,mean=200,sd=70)
data=matrix(gset,ncol=6)
```

3.5.5.2

Using the data generated above, calculate the standard deviation of the distribution of the means using `sd()` function. Compare that to the expected standard error obtained from central limit theorem keeping in mind the population parameters were $\sigma = 70$ and $n = 6$. How does the estimate from the random samples change if we simulate more data with `data=matrix(rnorm(6000,mean=200,sd=70),ncol=6)`

3.5.5.3

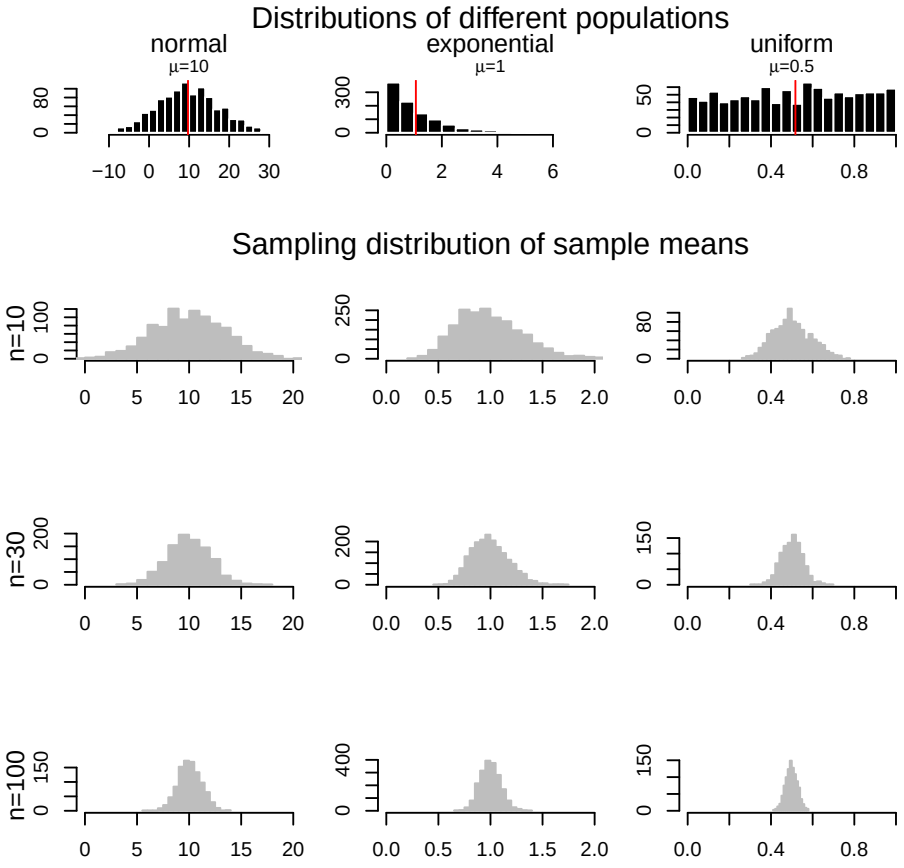
0. simulate 30 random variables using `rpois()` function, do this 1000 times and calculate means of sample. Plot the sampling distributions of the means using a histogram. Get the 2.5th and 97.5th percentiles of the distribution.
1. Use `t.test` function to calculate confidence intervals of the first random sample `pois1` simulated from`rpois()` function below.
2. Use bootstrap confidence interval for the mean on `pois1`
3. compare all the estimates

```
set.seed(100)

#sample 30 values from poisson dist with lamda paramater =30
pois1=rpois(30,lambda=5)
```

3.5.5.4

Optional exercise: Try to recreate the following figure, which demonstrates the CLT concept.



3.5.6 How to test for differences in samples

3.5.6.1

Test the difference of means of the following simulated genes using the randomization, t-test and `wilcox.test()` functions. Plot the distributions using histograms and boxplots.

```
set.seed(101)
gene1=rnorm(30,mean=4,sd=3)
gene2=rnorm(30,mean=3,sd=3)
```

3.5.6.2

Test the difference of means of the following simulated genes using the randomization, t-test and `wilcox.test()` functions. Plot the distributions using histograms and boxplots.

```
set.seed(100)
gene1=rnorm(30,mean=4,sd=2)
gene2=rnorm(30,mean=2,sd=2)
```

3.5.6.3

read the gene expression data set with `data=readRDS("StatisticsForGenomics/geneExpMat.rds")`. The data has 100 differentially expressed genes.First 3 columns are the test samples, and the last 3 are the control samples. Do a t-test for each gene (each row is a gene), record the p-values. Then, do a moderated t-test, as shown in the lecture notes and record the p-values. Do a p-value histogram and compare two approaches in terms of the number of significant tests with 0.05 threshold. On the p-values use FDR (BH), bonferroni and q-value adjustment methods. Calculate how many adjusted p-values are below 0.05 for each approach.

3.5.7 Relationship between variables: linear models and correlation

3.5.7.1

Below we are going to simulate X and Y values.

1. Run the code then fit a line to predict Y based on X.
2. Plot the scatter plot and the fitted line.
3. Calculate correlation and R^2.
4. Run the `summary()` function and try to extract P-values for the model from the object returned by `summary`. see `?summary.lm`
5. Plot the residuals vs fitted values plot, by calling `plot` function with `which=1` as the second argument. First argument is the model returned by `lm`.

```
# set random number seed, so that the random numbers from the text
# is the same when you run the code.
set.seed(32)

# get 50 X values between 1 and 100
x = runif(50,1,100)

# set b0,b1 and varience (sigma)
b0 = 10
b1 = 2
sigma = 20
# simulate error terms from normal distribution
eps = rnorm(50,0,sigma)
# get y values from the linear equation and addition of error terms
y = b0 + b1*x+ eps
```

3.5.7.2

Read the data set histone modification data set with using a variation of: `df=readRDS("StatisticsForGenomics_data/HistoneModeVSgeneExp.rds")`.

There are 3 columns in the data set these are measured levels of H3K4me3, H3K27me3 and gene expression per gene.

1. plot the scatter plot for H3K4me3 vs expression
2. plot the scatter plot for H3K27me3 vs expression
3. fit the model model for prediction of expression data using:
    - only H3K4me3 as explanatory variable
    - only H3K27me3 as explanatory variable
    - using both H3K4me3 and H3K27me3 as explanatory variables
4. inspect summary() function output in each case, which terms are significant
5. Is using H3K4me3 and H3K27me3 better than the model with only H3K4me3.
6. Plot H3k4me3 vs H3k27me3. Inspect the points that does not follow a linear trend. Are they clustered at certain segments of the plot. Bonus: Is there any biological or technical interpretation for those points ?

4

Operations on Genomic Intervals and Genome Arithmetic

A considerable time in computational genomics is spent on overlapping different features of the genome. Each feature can be represented with a genomic interval