A set of notes by Charlie **Spring 2017**

Notes on an 'Introduction to Genomic Technologies'

First Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One: Overview

Why study genomics? Everybody has a genome which governs their biology. Despite differences in peoples appearances, we are 99.9% identical. But what drives the differences? We start as a single cell, dividing into an embryo, and a whole person: all encoded within our genome. This code determines all the different cell types. Another big area of research in genomics is cancer: cells in your body which replicate without control. Mutations are changes in genome because of damage or errors in replication.

The central dogma of molecular biology: information flows in a single direction from your genome (DNA) to RNA to proteins. DNA contains the code for making proteins - mRNA is a copy located on the DNA molecule which leaves the nucleus of the cell, and the ribosome reads the coding sequence to put amino acids together.

Proteins are comprised of 20 letters (amino acids) – long molecules (3-400 amino acids long). Made from triplets on RNA, where each triplet encodes an amino acid. There are four possible RNA nucleotides, so there are 3^4 (64) triplets which get translated either into an amino acid or one of 3 stop codons which end a sequence. Over time, we've learnt that information can flow both ways: some proteins bind and modify the DNA - self regulating - and the dogma has been largely refuted. Other modifiers can affect DNA itself. Sequencing technology is at the heart of the genomics revolution: creating enormous datasets which now take longer to analyze than create. The cost of sequencing a genome has dropped from $100m to about $10k since about 2002. The biggest international repository is NCBI.

Genomics is the branch of molecular biology concerned with the structure, function, evolution and mapping of genomes – where a genome is all of the molecular information inside your cells which defines how your body works. The structure is a sequence of four nucleotides (A,C,G,T), and in the human genome there are 3bn of these letters strung together, divided into 23 chromozone pairs – 22 identical and then {x,y} for men, and {x,x} for women. Within each chromosone there's a centromere, and at the end, a telomore. A chromozone is a thread-like structure of nucleic acids and protein found in the nucleus of most living cells, carrying genetic information in the form of genes. The function of a genome is all the things it does: how to develope from an embryo, respiration, metabolism, etc. Chimpanzies diverged from us about 6m years ago – but the sequence similiarities are close, even for unrelated organisms like bacteria. For example: reproduction. A gene is a heritable unit - a small section of the genome which encoded a protein which has some specific function.

Note: Important difference between genetics and genomics – genetics typically studied one or a few genes, whereas genomics studies all genes at once. Data challenges drive genomics with global, high-throughput experiments.

Genomic data science is at the intersection of biology, statistics and computer science, also known as computational genomics, computational biology, bioinformatics or statistical genomics. Steps to undertaking genomic data science:

1. Start with 'subjects' - humans, mice, etc. Collect samples from the subjects (e.g. skin cells) – experimental design is important, even though data collection is now cheaper.
2. Prepare the samples in the lab and send them for sequencing, which generates enormous amounts of data.
3. Take sequences ('reads') which are short fragments of the genome - align them to the reference human genome - which represents 'average northern european male' - that tells us how they differ from the reference genome.

4. Preprocess and normalize data in order to correct various types of biases - this is because the sequencing machine can make mistakes (random and non-random), there exist data collection biases, etc.

5. Apply statistics and machine learning techniques to go from preprocessed/normalized data to scientific conclusions.

6. The whole process typically involves a lot of software development.

Moving from individual genomics (i.e. how a cell mutates) to population genomics can show how genomic differences cause recogniseable changes between people. Celluar organisms can be split into 3 domains: 1.) bacteria, 2.) archaea, and 3.) eukaryota - which have cell nuclei, evolved as a way to sequester our DNA from the rest of the cell. Yeast is a single cell eukaryote - which still has a nuclei. Prokaryote (bacteria and archaea) has loosely organized DNA floating around. In Eukarayote cells (surrounded by a wall), there is a nucleus (which has its own wall), inside of which is chromozones (long molecoles of DNA). Mitocondria has its own (small amount of) DNA - the 'powerhouse' of the cell - these genes are responsible for energy metabolism. Over the course of their life, cells have a well defined cycle (e.g. mitosis - DNA replicates within a cell, then the cell splits into two 'diploid' cells). Diploid means that we have two copies of every chromozone - male and female. Not all Eukaryotes are diploid (but humans are). During the course of development - cells need to develop into different type of cells – all begining from stem cells. They go down developmental paths which differentiate them. During sexual reproduction, something different happens: different recombinations - 'crossing over' - part of chromozone 1 from female crosses over into chromosone 2 with male - typically about one crossover per chromozone. This is the main reason why family members aren't necesssarily alike. DNA is the molecule which comprises all of our genetic material, composed of nucleotides of nucleic acids - A,G,C,T ([Adenine, Guannie] - two ring 'Purines', [Cytosine, Thymine (and Uracil)] - Pyrimidines which are a little smaller). A's **always** bind to T's, and G's to C's. This property is very important for how DNA copies itself from one generation to another: even with one strand, you know what the other strand is. Every one of your cells has all of your DNA in it, stored in the famous double helix structure - coiled up and packed into the nucleus. The DNA sequence looks like this - note the direction (3' and 5')!

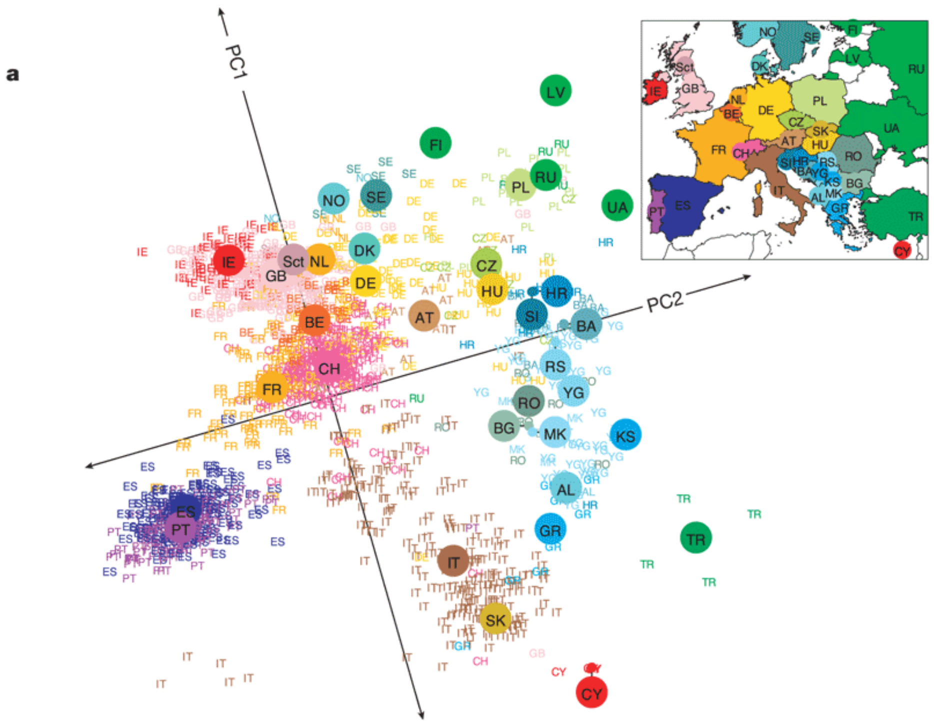
3' - ACACCGGTT - 5'
3' - TGTGGCCAA - 5'

where the second is the reverse compliament (or 'negative' strand). RNA is *almost* like DNA, the most notable difference is that Thymine is replaced with Uracil - RNA molecules are single stranded in general - and from this RNA template we create proteins. DNAs are replicated between cells, and the RNA uses the DNA template to make proteins (although an otherwise identical template). DNA is basically a program - RNA makes proteins - translate from RNA into proteins. Every combination of 3 letters of RNA encodes an amino acid. Translation machinery reads across RNA molecule 3 at a time - when it hits a codon, thats the end of the protein.

The Human Genome Project: first proposed by the Department of Energy in the 1980s, was originally opposed on the basis of cost by geneticists who thought only individual cells should be targetted. In the early 1990s, the plan was to create 'maps' - small peices of DNA and place them on a genome. In 1995, TIGR sequenced the first complete bacterial genome - haemophilus influenza - 1.8 million bases, 1742 genes. In 1998, Applied Biosystems developd a new sequencing machine - faster. Celera Genomics - a for-profit company enters the race, and the public sector increases its efforts in 1998. Each genome is different - the Human Genome Project sequenced one 'mosasic' of about 12 people of northern Eropean descent. Two papers published simultaneously on a draft of the genome - estimates between 30,000-40,000 genes. Now we believe the number is between 20,000 - 22,000. This number relates to the number of protein coding genes - a piece of DNA which gets transcribed into RNA and translated into a protein. However, there are also some genes where DNA gets transcribed into RNA, which itself has a function (without getting translated into a protein).

DNA is inside every cell, 23 chromozone pairs - if you stretch it out - it's about 2m long *per cell*. DNA is coiled around histones - highly alkaline proteins found in eukaryotic cell nuclei that package and order the DNA into structural units called nucleosomes. When translated, it needs to unwrap. The structure also repeats itself, classified either as 'tandem repeats' or 'interspersed repeats'. Typical human messenger RNA has several regions: 5' UTR - 'untranslated region at the start and the 3' UTR - the end untranslated regions. After transcription occurs, a long series of A's is added -'poly-a-tail'. In the middle is the 'coding sequence' - CDS.

Decomposition by PCA (Novembre et al., 2008)



Proteins themselves have structure: the amino acids can also form sheets or helixes. There are thousands and thousands of structures of proteins which confer upon them thier structure. Cells behave differently (i.e. skin and blood) - this is because different genes are active in different cells - controlled by proteins which go back and bind to the DNA itself (accelerate or decelarate) - a control mechanism to decide how much of a protein to produce.

Epigenetic means 'beyond genetics' - factors outside the DNA itself which control something about how the cell functions and how genes are expressed. Affected by: development, environmental chemicals, such as drugs or pharmaceuticals or aging or diet. This can lead to: cancer, autoimmune disease, mental disorders and diabetes. For example - methyl groups can tag DNA and activate or repress genes. The binding of epigenetic factors to histone 'tails' alters the extent to which DNA is wrapped around histones and the availability of genes in the DNA to be activated.

Genotype: collection of all sequences of all genes in all cells - determines how cells functions and whether you have certain traits or diseases - *inherited information*. Phenotypes are traits such as height, hair colour, weight or even personality - *something we observe*. We have two copies of every gene. Traits can either be recessive or dominent: if we have two copies of a gene then we have the recessive trait, and if it's a dominent trait: we only need one copy of that gene. Note: from two pairs (mates), there are four possible combinations of traits - this is how genotype affects/confers phenotype. We have now started to map out genetic variations across the world - specific mutations are common to specific parts of the world - either single nucleotide polymorphisms (SNPS) or larger chunks of DNA. This allows us to categorize populations by their genomes:

The figure ('Decomposition by PCA') shows the axes upon which the genes vary the most with all the information reduced to two dimensions (two PCAs with the most variance). It is possible to measure the connection between genotype (.e.g. AA, AG, GG) and phenotype (brown/green/blue eyes), where, for example, HERC2 is the gene for eye colour. Genome Wide Association Studies (GWAS) study large groups of people and the association between single nucleotide polymorphisms or other associations and diseases or traits of interest.

Reading for Week One (hyperlinks)

- A nice discussion of (the demise of) central dogma.

- A nice discussion of epigenetics and cancer.
- An overview of what DNA is.
- The real cost of sequencing (i.e. downstream analysis, etc.
- The need to combine bioinformatics and medical informatics.
- An overview of what NGS makes possible.

Week Two - Measurement Technology

Polymerase Chain Reaction (PCR) is a way to copy DNA. DNA is always double stranded: e.g. primers such as 5' ACACCGGTTCGTAGAGCAT 3' and 3' TGGRGGCCAAGCATCTCGTA 5'. But how do we do PCR?

1. Melt gently (94 degrees) - so strands fall apart and so the primers wont stick to the DNA and the two strands of the DNA wont stick to each other.
2. When cooling (anneal - 54 degrees), the primers will stick to the DNA before the DNA strands stick together.
3. We need a second mixture (provided by nature) - a copier molecule - DNA polymerase which we can synthesize - enzymes that synthesize DNA molecules from deoxyribonucleotides, the building blocks of DNA.
4. It will find the sites and fill in missing sequences at primers - 'extension' (72 degrees)
5. Repeat this: with each round, we double the amount of DNA we had before. Typically done 30 times to get about 2 billion copies.

Summary of PCR: we need DNA to copy, primers, DNA polymerase and lots of As,Cs,Gs and Ts. Melt at 94 degrees, cool to 54, warm to 72, repeat. *Next Generation Sequencing* – NGS. DNA sequencing began with Sanger DNA sequencing, then DNA microarrays, then 2nd generation DNA sequencing (most common in use now), then 3rd generation & single molecule sequencing (2010 - present). Understanding sequencing techniques helps to understand the data to be analyzed.

First, DNA is copied using DNA polymerase, which takes free nucleotides floating around in a cell/synthesized, to copy DNA trying to be synthesized using rules that Gs bind to Cs, As to Ts, etc. NGS takes template DNA, chops it up into small pieces (maybe 1000 bases long), attaches them to a slide and uses PCR to replicate them. We then hit them with a light which shows them in four different colours (they 'fluoresce'): add them to the slide and have them basepair with nucleotides. We then cycle through the bases. However, errors increase later in cycles: clusters can be 'on schedule', 'behind' or 'ahead' - where bases arent added in the correct sequence. The base calling software also estimates how likely there is an error at a speciic point based on how pure the colour signal was.

The basic idea of NGS applications: convert a molecule to DNA and apply 2nd generation sequencing to measure it (i.e. exon sequencing). Exons are part of RNA - concatenated together to get translated into proteins. For this reason, we only look at the protein coding exons using 'beads' - which hybridize and attaches to exomes. Another technology is RNA sequencing - capture all genes being turned on/expressed in a cell/collection of cells. After transcription, the cell attaches a long string of As to it, (the 'polyA' tail) and this is the basis of RNA sequencing. Reverse transcribe complimentary DNA (cDNA) to it. A third application is 'chip-seq' - trying to understand where on the DNA certain proteins might bind - link the proteins onto the DNA and crosslink the protein to the DNA in the cells. Chip-seq involves anti-bodies which pull out fragments of DNA which proteins are bound to. Methyl-seq determines where on the genome the DNA has been methylated (which proteins have been expressed in the cell). The methylation marks can be passed on from one cell cycle to another as cells divide - split DNA into two identical samples, and treat one differently (in a way which isolates Cs which bind to methyl groups).

Reading for Week Two (hyperlinks)

- Explaining the ‘sequencing-by-synthesis’ methodology.
- Design and analysis of ChIP-seq experiments.
- An overview of NGS from a vendor.
- Information on the exact role of RNA in protein synthesis.
- Further information on the transcription process (DNA into RNA).

Week 3: Computing Technology

We can broadly split the computing technologies into theory, systems and applications. Thinking computationally is essential. Don’t forget: computers do exactly what you tell them to do! There are dozens, if not hundreds of programming languages: how we talk to computers. Specific defined terms with specific meanings - but writing good code is hard! Engineering means testing your code - robust, debugged. An algorithm describes what a computer can do: a step by step series of instructions on how to do something. Not necessarily used by computers: e.g. finding the maximum of a hill, or sorting a bunch of numbers. If your dataset is large, efficiency becomes key.

How do we get the sequencing data into memory? Most importantly, *we want to be able to find stuff* - e.g. 100 nucleoids. To understand how memory works, note that not only do we have the data itself, but also an address, termed ’pointers’. Numbers and characters are stored as 8 bits in a row - a byte - 0000 0000 - native computer code. Everything on a keyboard is represented as a single byte. Considering genomics specifically: DNA = {A,C,G,T} and A=00, C =01, G=10, T=11. When it comes to efficiency, think: how do they plan to deliver mail - when should the truck go back to the warehouse? It’s important to understand that computer programs arent just black boxes - know what’s going on under the hood. Of particular importance is the software behind ‘alignment’ - examples of RNA editing/differences across ‘big’ datasets. Typically 1 misalignment in 1 million: which yields 100s of errors in large genomics datasets. **Make sure your software handles all possible cases** - verify all your software if you think you have important results!

Computational biology software transforms raw data into information to guide discoveries: DNA sequencing data – A,C,G,Ts etc. There are a multitude of programs and pipelines. What’s the difference from my genome than the reference genome? To find out: run it through a pipeline. Example – RNA-seq: used to measure difference between cell types, which genes are turned on, turn RNA into raw sequences, etc. Go from raw-reads to a list of genes and expression levels. ‘Tuxedo tools’: bowtie - alignment to human genome. tophat2 - spliced alignment. cufflinkks - transcript assembly and quantitation. Cuffdiff2 – differential expression between one set of data and another set of expressions. These programs are being superceded - bowtie2 - faster alignment, hisat - spliced alignment, ballgrown - differential expression - stringtie - transcript assembly and quantitation. What is surprising is that even though these are well defined, discrete tasks, different software make different alignments of the same reads - perhaps only 98-98.5% similarity? Even when aligning a read, the alignment may be to different places. Software is changing extremely quickly in this field (as is the data).

In comparison to statistical/econometric estimation, where software will almost unilaterally produce identical results, this is not necessarily the case with computational biology toolboxes.

Reading for Week Three (hyperlinks)

- Further information the specifics of alignment.
- Details on hash based compression for storing DNA
- A guide to big data management with R
- An overview of RNASeq Analysis.

Week 4: Why Care About Statistics?

Genomic datascience is based on three core disciplins of biology, computer science and statistics. Statistics is often overlooked in this triage: however, several existing studies are limited through their poor use of statistics which are often implemented poorly. This can even result in some lawsuits when results lead to clinical trials. There are two key reasons as to why statistical analysis can go wrong: a.) a lack of transparency and reproducibility of work done by others. This can also be further hampered by a lack of cooperation by leading authors. b.) The second key issue is the lack of statistical expertise and study design problems (e.g. batch problems - see below). Statistical expertise can help 'upstream' - before the study even begins (see power of tests - below). The central dogma of statistics:

The central dogma of statistics involves inference on a population of interest from a carefully stratified sample. The key idea involves quantifying the variability of the sample.

Data sharing plans are essential. All genomic datasets need four things:

1. Raw data: no processing, no computing, no deleting.
2. Tidy data: one variable per column, one observation per row, one table per 'kind' of variable, linking indicators per table.
3. Code Book: variable names, variable descriptions, study deisgn information.
4. Recipe to go from raw to tidy data with the code book: r/python code, inputs raw data and outputs tidy data with no parametres (avoid the temptation to not use a script).

An excellent example of a datasharing plan can be found here. This also introduces us to Github - a hugely valueable resource for code sharing and wikis. There are a multitude of resources available for learning statistics on the internet. One way to get a bit of help is via Cross Validated. A 'lonely bioinformatician' is a single statistician working in a lab without other statisticians (as opposed to a computational biology lab).

'Make big data as small as possible as quick as possible' to enable sharing and visualising. Sometimes statistical summary measures can be deceptive (hiding hidden patterns). One common, useful task is to 'plot replicates' - a very common plot where you compare the same sample being run through the technology twice. Log and other types of transforms are extremely useful. A common genomic example is an MA plot, which is an application of a Bland–Altman plot for visual representation of genomic data. The plot visualises the differences between measurements taken in two samples by transforming the data onto M (log ratio) and A (mean average) scales, then plotting these values. A common problem in genomics: a meaningless albeit visually impressive image of a network. Statistical graphs must not only look pretty, but convey scientific information to the reader.

Sample size and variability: with our best guest from a sample, we also get a guess about the variability. Typically, $N = \frac{\$youhave}{\$costpermeasurement}$. Even if means are different from each other, how confident can we be about that? This is measured with 'power': e.g., n=10, delta=5, sd=10:

```
power.t.test(n=10,delta=5,sd=10)
```

We can also 'back this out': how many observations will we need to have given a specific power, delta, sd, to detect a difference of specific magnitude? As you vary different paremtres (e.g. n, variance, etc), you get different powers: these calculations are therefore hypothetical based on what you think the effect size might be, and what power you might have. There are three types of variability in genomic measurement: phenotypic variability, measurement error and natural biological variation. Measurement error may decrease over time with technological advancements, but biological variation does not get elimiated by technology.

Statistical significance: are observed differences replicable? Are they real? A common metric that people use is the t-statistic:

$$t = \frac{\bar{Y} - \bar{X}}{\sqrt{\frac{S_y^2}{N} + \frac{S_X^2}{M}}} \tag{1.1}$$

This is the average of the Y minus the average of the X divided by some measure of variability: if they're very far apart in terms of variability units, we might think that they are statistically different. The most commonly used statistic is the p-value, calculated by ‘scrambling’ or permutating the relationship between the label and data of Y and X. Then, plot a histogram and see where the original value lands: calculate number of times the scrambled statistic is bigger than your observed value:

$$p = \frac{\#|S^{permutations}| \geq |S^{Obs}|}{\#permutations} \tag{1.2}$$

The p-value is the probability of observing a statistic that extreme if the null hypothesis is true. It is **not** the probability that the null or alternative is true, or a measure of statistical evidence.

However, this classical approach to p-values is not designed for multiple hypothesis tests at once. Note: if there is no differnce in what’s going on, p-values are uniformly distributed - this means that 5% of the p-values will be less than 0.05. How do we correct for this? With different error rates:

- Standard p-value: $0.05\times$# tests = # false positives.
- False discovery rate $\leq 0.05\times 550 = 27.5$
- Family wise error rate controlled at 0.05: The probability of at least 1 false positive ≤ 0.05

However, it’s important to report negative results and avoid p-value hacking - one way to do this is to state an analysis pla in advance of looking at your data, and stick to it. Confounding: a variable related to other variables which may look like there is a relationship when there isn’t. One important confounder in genomics is the time at which the sample is taken (i.e. due to technological advancements). To overcome this, better consider your study and experimental design - through stratified sampling or randomziation. Other important things: balanced design (# treatments and controls), study should be replicated - technical replicates (measure how well technology works) and biology replicates (sample across people).

Reading for Week Four (hyperlinks)

- An excellent discussion on p-values and power.
- A guide on ‘gene set bagging’ - to calculate probability of results will replicate in the future.
- A nice paper on the significance thresholds/MHT in GWAS studies.
- A related paper on power and significance testing in ‘large-scale’ genetic studies.
- Methods to overcome confounding.
- An overview of hypothesis testing (philosophical).

A set of notes by Charlie **Spring 2017**

Notes on an 'Genomic Data Science with Galaxy'

Second Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One: Introduction

The Galaxy Project (http://galaxyproject.org/) provides (free) user friendly software for producing reproducible genomic pipelines for data analysis. Galaxy is a web-based platform for performing data intensive science. It is particularly suited for genomics, and tools for many different types of genomic data analysis have been integrated in Galaxy. Galaxy is motivated by the need for reproducibility - fueled by the data intensive nature of modern day biology which are heavily dependent on complex statistical methods. Reproducibility of genomic studies is a 'crisis'. Vasilevsky et al. (2014) show that a tiny fraction of 'high impact' papers make clear the exact tools and software they used to conduct research. Nekrutenko and Taylor (2012) show how not only the tools, but also the versions of the tools can create different outcomes. Note: reproducibility is not correctness! It means that an analysis is described in sufficient detail that it can be reproduced by another person in another environment. However, most published analyses are not reproducible - missing software, versions, parameters (etc). Recommendations for research:

1. Accept that computation is critical.
2. Always provide access to raw primary data.
3. Record all auxiliary version of all datasets.
4. Store the exact versions of all software used, and archieve it.
5. Record all parametres, even default ones.

Galaxy is one solution: an analysis environment where all the details and provenance are automatically documented. It's also opensource and can be extended as required - it's also extensible for sharing tools, datatypes, workflows, etc. There are four main features to it:

1. Analysis tool - the primary unit.
2. Analysis environment - takes the tools and gives them a uniform UI.
3. Workflow system - for complex multi-step analysis which can then be re-run in one shot.
4. Importantly, it has pervasive sharing and documentation with integrated analysis for collaboration.
5. It also provides a platform for visualisation.

Galaxy offers free accounts, with limitations on data-storage and the number of jobs which can be ran. The two main alternatives are: i.) locally, ii.) on the cloud (most easily on AWS).

Week Two: Galaxy 101

We're now going to look at human chromozone 22: which coding exons have the largest number of repetitive elements overlapping them? First, we need to get data into Galaxy, then identify which exons have repeats, count repeats, then save and download exons with the most repeat. **Refresher:** The term exon refers to both the DNA sequence within a gene and to the corresponding sequence in RNA transcripts. We're going to undertake our analysis from https://usegalaxy.org. The 'Get Data' tab connects directly to a large range of genomic databases for analysis directly. In our example, we're going to use the UCSC Main table browser - a database associated with their genome browser - hg19, chr22, output format: BED (browser extensible data, send output data to Galaxy). Galaxy fetches data in the background: while we're waiting we can continue to use Galaxy. A job is queued (grey), in progress (yellow), failed (red) or completed (green). Clicking gives the preview, and the eye format shows us the data. The bed format gives us the chromozone, position, name, score, strand, etc. To count overlaps, use the 'join' tool in the 'operate on genomic intervals'. When pairs are found, it turns green to show the job is completed. The 'pencil' icon allows us to edit attributes (e.g. field names). After our join is finished, we'll have 6 fields from the exon field, and 6 from the repeat files, joined side by side where there's overlap. To count the number of repeats: 'grouping tool' - join, subtract and group - Group. We then want to group by column 4 - our exon variable. We then want to 'add new operation': 'count' on column 4, and execute. We can then insert operation, and count. Note: for every task and dataset, Galaxy is keeping the provenance of each operation and storing it for reproducibility. The 're-run' button allows us to slightly re-parameterize the problem and run it in a slightly different way. To get data for output: we can join on names on the group, as before. Summary:

- Interactive analysis in Galaxy is performed using tools to operate on datasets.
- Datasets are **immutable**: running tools always creates one or more new datasets.
- Datasets are available through 'history' - which gives complete provenance.

Galaxy work-flows try and take abstract analysis and allow it to be run on other datasets. We can create work-flows from scratch using the work-flow editor, or perform them by example - within the analysis directly, and then extract the work-flow. For example, we can make saved histories our current histories using the history tab on the right of the interface. All of the steps which are undertaken previously can be brought forward and ran all at once sequentially for different datasets. You also have datasets which can be marked specifically for output. 'Post-step' actions can rename datasets, change datatype, assign columns, email notification, etc.

One of the main strengths of Galaxy is the ability to share and collaborate with other users. Nearly all Galaxy entities can be tagged (short structured metadata) or annotated (free form description). Above the datasets there are the 'tag' and 'speech bubble' icon. The tags can be used for searching and be used to find specific histories. Galaxy allows us to share with a specific user (via email or link), or to publish to everyone publicly. There's also a tab of all published histories of all users, etc. The community can actually rate the quality of the shared objects. You can also publish and disable links very easily. In addition to sharing and publishing, there are also 'Galaxy Page's: text documents which permit embedded galaxy objects (such as workflows). Pages also have formatting for headers , etc. The page can then be viewed with objects/histories/etc embedded - similar to an iPy notebook?

Week 3: Working with Sequence Data

Quality control for sequence data: Trial with Illumina iDEA datasets: BT20 paired-end RNA-seq subsamples - search for it through the Data Library. Click on the eye to preview the first 1mb. We can see that this data format is based on records of 4 lines each, where each record begins with an @identifier, followed by sequence identifier and then an encoded quality score. Keys tell us how to map characters e.g. Sanger encoding. FastQC can provide us with information and quality metrics about the data. To access it: NGS: QC and manipulation: FastQC Read Quality reports which gives us two datasets - one is a report which is a `.html` page with summary information and plots (e.g. quality scores). We can also use these tools to trim specific sequences, and only look for regions of certain lengths which have quality scores above a certain level. What tools do you need to use? It depends on your downstream analysis. Summarize: many factors which affect the quality of DNA sequencing data. FASTQC is one tool which allows us to evaluate these metrics. Galaxy provides FASTQC manipulation options to 'recover' even low quality data to salvage for analysis.

We now move to **ChIP-Sequence Analysis with MACS** - map locations of protein/DNA binding and histone modifications. MACS - Model-based Analysis of ChIP-Seq: tag distributions which represent the end of a sequenced fragment in order to resolve the centre of the bound region. MACS empirically models the amount of this shift to better determine the boundaries of the region. It works by sampling a number of high quality windows which are enriched for binding, and using these regions to estimate the shift. As an example: use Mouse ChIP: G1E CTCF binding. These are data from two different cell conditions. G1E is a mouse cell line where a specific transcription factor: we have null and restored. The transcription factor is CTCF involved in genome structure and gene regulation. Use Bowtie2 to map our data onto the mouse genome. Map this to mouse build mm9 with default parameters. Aligned in BAM format: binary alignment format. NGS Peak calling -> MACS - and then we can see how the reads align, and where they peak. We can visualize the `.BED` file of peaks. In doing ChIP-Seq analysis, there are a number of issues and biases: chromatin accessibility affects fragmentation, amplification, repetitive regions, etc. We can add a control to MACS to determine the background expectation for the number of peaks you should see, and compute a false discovery group. This is extremely important. NGS Peak calling MACS explicitly allows us to add a 'control': again - a BED file and a html report. Other tools may be more appropriate for broad histone modifications. Summary: MACs is one tool for ChIP-seq data. Controls are extremely important for accurately calling ChIP-seq peaks, but there are other tools which may be more appropriate depending on the type of data, such as broad histone modifications.

Week Four: RNA-seq and Running your own Galaxy

RNA-seq Analysis: mapping of RNA-seq data using Galaxy. Use a demonstration dataset called Human RNA-seq: CHB Encode Exercise. Grab all 5 datasets: 2 sequenced datasets (replicates) for two different cell types, and then gene annotations. Each of this is a FastQ dataset (ideally, run quality control on these datasets using FastQC). We want to look at levels of mRNA - which are isolated and fragmented using random priming to create short DNA sequences which are complimentary to the original mRNA sequences. Now we need to analyze this RNA sequencing data: which can be done in many different ways. For transcriptones - two paths we can take - we take the align then assemble: take all the reads then align them back to the reference genome. This is potentially more sensitive than the de-novo approach - which is likely to capture only highly expressed transcripts. We need to be able to align reads back to the genome in a way that is aware of splicing (with ChIP-Seq - we expected every read to align exactly back to our genome with some small differences from sequencing error). However, transcripts have been spliced: our reads may align to two regions with a large gap in-between them. To do the read-aligner, we can use Tophat: to get 4 new datasets: align summary, insertions, deletions, splice junctions and accepted hits: a mix of `.bed` and matched reads. We can run this over multiple datasets: select multiple datasets simultaneously using shift click using HG19 as the reference genome (this is a single ended dataset). Importantly, when we are using RNA-seq analysis using a reference genome, we require an aligner that is splicing aware - which means that it can handle what appear to be long deletions in the reads. Tophat is one such aligner which is available in Galaxy.

Spliced alignment provides estimates of 1.) locations of exons and 2.) splice junctions. Is this enough to know what transcripts are present? We're going to use our reference annotations as a guide: use the information from both RNA-seq and reference annotation for developing gene models. We're going to use reference annotation as a guide and run this over all 4 of our accepted hits datasets (use multiple datatsets feature in Galaxy - the BAM format ones) using Cufflinks. This will give us sets of transcripts assembled from each of our spliced map reads. Each Cufflinks job generates a couple of different datasets. We want to quantify the expression levels of each of the gene models discovered using a number called FPKN. We have a tracking name - either cuff.x or genes coming in from our reference annotation. GTF files are the actual gene models which Cufflinks has discovered. We'd like to do a statistical test of whether genes are differentially expressed between the two conditions. However, we have 4 different sets of gene annotations coming from whether the genes are differentially expressed between the two conditions. However, we have 4 different sets of gene annotations coming from the 4 different RNA-seq samples. First run a tool called cuff-merge which joins the different assemblies together. Don't forget to use the correct (chr-19) reference annotation to get a set of merged transcripts. We can now use this as our consensus gene model set for analysis - are things differentially expressed? To answer this - use Cuffdiff, which allows us to bring in all 4 of our datasets to ask whether things are differentially expressed. This gives us IDs for each of the genes and a status of whether a test was done: in some cases there is no expression data, in which case the test cannot be performed. In the cases where we have p and q value: a p-value that has been corrected for multiple testing (and then a column for whether they are expressed differently at all). Using Galaxy's filtering tools you can then extract genes which are expressed differently at different thresholds. **Summary**: Using the spliced alignments from Tophat, Cufflinks allows us to assemble transcript in Galaxy. Cufflinks will also quantify the relative abundance of each transcript in each sample. Cuffdiff performs a statistical test for differential expression based on quantitative data from multiple conditions.

Running your own Galaxy

Galaxy comes with limits on power and storage (250gb and 6 concurrent jobs) - 'Galaxy Main': the web based interface. The easiest way to get the local instance is from getgalaxy.org. Note: at the minute, it seems that Windows is not supported. You do need a version of Python 2.x (not 3.x). Ideally: clone from git:

```
git clone https://github.com/galaxyproject/galaxy
```

To get Galaxy running: `galaxy demo$ virtualenv. venv` into a virtual environment. Then: `sh run.sh` so Galaxy will automatically use this virtual environment. The server will run on localhost: 127.0.0.1:8080 - a brand new Galaxy running locally (with a limited number of tools). Use administrative access to configure access to the Galaxy instance (.ini). Create an account locally corresponding to the email address entered in the .ini, and you can then see the admin tab which allows you to administer your instance. To learn more: `galaxyproject.org/admin`. **Summary:** Galaxy is distributed through Git: `github.com/galaxyproject`. This can then be configured to run on a cluster, be more robust, etc.

Cloud computing is a model in which computing resources are available over the internet: acquired and configured quickly, and then released. There are many providers. Examples are: Amazon Web Services, Rackspace, etc. It's also possible to build local cloud computing environments (e.g. OpenStack). The recommended platform for Galaxy is `aws.amazon.com`. In particular: EC2 (elastic compute cloud - block storage and compute servers) and S3 (storage service). Galaxy itself can deal with setting it up automatically! All we really need to do is allow the cloud launch component of Galaxy to access our account. Create a new group (e.g. EC2) with certain permissions - i.e. access to EC2 and S3 (full access). We want to create a user called 'cloud-launch', and then add them to the EC2 group. This will create a new compute instance with a pre-configured version of Galaxy available by the web, through which we can then access the 'cloudman' console to configure our instance further. When authenticating: you can leave U/N blank, and use the same password as per cloudlaunch. When the cluster is ready for use, we can add additional compute nodes and nodes of different types. The grid shows the nodes of which are currently on. The instance should look the same as local or Galaxy Main, albeit now with more power and storage. To add the tools which we need: use the Galaxy Tool Shed (toolshed). This is done through the CloudMan console (Admin): there are a number of options, such as restarting services. If we add our own user here, it will modify our Galaxy configuration to allow modifications to Galaxy. If we then register a new user with the same user-name that we added an administrator, we can then access (Galaxy Main) toolshed. **Summary:** automated support provided for AWS instances which allows you to acquire as much resources as you need, and then release those resources when done.

A set of notes by Charlie **Spring 2017**

Notes on ‘Python for Genomic Data Science’

Third Module of the Coursera Genomic Data Science Specialization from John Hopkins University

No prior knowledge expected here. Steps to programming:

1. Identify the required inputs.
2. Make an overall design for the program, including the inputs list.
3. What will the output be? a file? a printout? a figure?
4. Refine the specification by adding more detail, then write the code.

Another way to organize your thoughts is to write ‘pseudocode’. One of the most productive environments for genomic data-science is **Python** - easy to learn and extremely powerful - a simple but effective approach to object orientated programming. It’s ‘interpreted’ – this means that you dont have to compile it: the computer will instantly interpret one line of code at a time, if required. For compiled programs, you have to run it through a compiler to create a binary object which gets run: Python converts to binary ‘on the fly’. Python is extremely ‘interactive’, portable (cross-platform), and extensible (’wrapper’ functions for other languages’) with a large range of inbuilt functions and is very scale-able (small programs to large functions). Python can be used as a simple calculator. For example: `5+5`. `**` can be used for powers e.g. `10**2=100`. The order of precedence of operators is the same as in standard maths (i.e. multiplication takes precedence over addition). Numbers can be different types: i.e. `type(5)` is an int, but `type(3.5)` is a float. Complex numbers: `type(3+2j)`. However, DNA and are really just strings of letters. e.g. `atg` or `this is a codon, isnt it`. You can also use a backslash as an escape character i.e. `''This is a string isnt it''`. Strings can span multiple lines with tripple quotes. The newline character is \n. Other escape characters: tab: \t, backslash:\\, double quote: \". Here are the basic string operators:

- `+` : concatenate strings.
- `*` : copy string (replicate).
- `in` : membership.
- `not in` : non-membership.

Without variables, you cannot do anything in Python - theyre storage conatiners for numbers, strings , etc. The equals sign assigns a value to a variable. for example: `codon= 'atg'` and `dnaseq='gtcgcctraaccgtatatat'`. The name associated with a value is called a ‘variable’ because its value can change. In general, give your variables a meaningful name, rather than ‘a’ or ‘b’ which have no meaning. Note: case sensitive. The underscore character helps add readability of variable names. Variable names cannot use ‘illegal’ characters (e.g. pound sign) or begin with numbers. One important thing that we do is search for substrings in a bigger string. *This is especially important in genomics*. We can index strings using things like `[x]` and slice them with an operation like `[x:y]`. Note: we begin indexing things with 0. You can also use negative numbers while indexing to start from backwards. When we give two numbers in a slice, e.g. `[0:3]` will pick out a range of the first three characters of our string. If we leave off the second number, we index to the end of the whole string. `len` tells us the length of a string and `type` tells us the type of a string. We can find help about any built in function using `help()` which tells you everything that function () can do. Object orientated programming tells you about the objects, rather than the actions. The `string.count(str)` function tells us how many times `str` features in `string`. We can also case variables using `string.upper()` and `string.lower()`. The `string.find(str)` tells us what position `str` is in within `string` (but only the first occurance). We can reverse find using `string.rfind(str)` which will look from the end of the string. We can get booleans (True/False) with `string.isupper()` and `string.islower()`. We can also use other functions, such as `string.replace.()`

For our first example, we want to compute the GC content of a DNA sequence: the percentage of CGs in a sequence which is all Cs,Gs,As and Ts. Use the `string.count()` method and `str.len()`. To execute all commands at once: put them all into a .py file, and execute the file (might need `#/usr/bin/python` ?). Make the file executable with `chmod a+x`. An important principle of all programming is to 'comment' your programs: what was it that you had in mind when you wrote the code? Everything inbetween tripple quotes is ignored by Python, as is everything after a hashtag at the end of the line.

We can capture raw input from the user using the `'input'` function, e.g: `dna=input('enter a dna sequence')`, where whatever the user types is then captured to the variable `dna`. This will always return a string (although you can convert it later, of course). Some string conversion functions:

- `int(x,[base])`: convert to an integer
- `float(x)`: convert to a floating-point
- `complex(real,[,imaginary])`: convert to a floating-point.
- `str(x)`: converts to a string.
- `char(x)`: converts to a character.

Don't forget! You can use formatting command when using enhanced `print.()` commands. For example: `print('The DNA sequences GC content is %5.3f %%' % x)`. There are a range of different formatting commands (e.g. % d, % 3d, % o, %e etc). One type of structure is a `list`: an ordered set of values: `geneexpression=['gene','5.16','0.0001512']`. We can alter values of an element of a list, print them out, slice them, etc. There are a whole host of operations on lists, such as concatenation and list. The `del` operation can be used destructively: e.g. `del list[1]` will delete the second element of the list and we can also count the number of times an element appears in a list and reverse it with list.reverse(). We can also `append` and `pop`, which allow us to treat list as 'stacks'. `append()` adds an element to the end of the list. The difference between append and extend is that append takes one element, but extend adds a second list to the end. The `list.pop()` function removes the last element added to the list. There are two ways to sort a list: `sorted()` function and another method is `mylist.sort()`. However, note that they do slightly different things. **Don't forget! help(list) and help(string) for all appropriate functions!** A `tuple` is like a list, but it is *immutable*. For example: `t=1,2,3`. We can surround them with or without parentheses. Another data structure is a `set`: an unordered collection with no duplicate elements (because no order, they have no index). Sets support mathematical operations like unions (|), intersections (&) and differences (-). Another important data type is the `dictionary`: which stores a *pair*: a key and value pair. This is especially useful for genomic data science: such as keys for sequences. We create a dictionary like this, wrapping the dictionary in {:

exampledict={'key1':'value1','key2':'value2','key3':'value3'}

Then, to gets its value out, index on the key, i.e. `exampledict[key1]`. You can do boolean tests of whether specific things are in your dictionary. We can also delete a key from the dictionary using `del exampledict[`key1]`. `len(dict)` tells us the length of our dictionary (the number of key-value pairs), and `list` returns all of the keys in the dictionary, and `list.values()` gives you all values (as opposed to keys). We can also sort on keys and values. Here is a comparison summary of the sequence data types:

Control statements allow non-sequential execution of code. `if` and `else` are extremely common examples which occur in the majority of programming languages. The keyword `if` is followed by a condition. The condition is a *boolean*: a true or a false. If `True`, the code is executed. Boolean are formed with the help of *comparison* (e.g. ==), *identity* (e.g. `is` and `is not`) and *membership* (e.g. `in` and `not in`) operations. If the results of these statements are `False`, then we need an `else:` statement to determine what to do next. The `else` statement must be at the same level of indentation. We can have multiple sequential conditions with `elif:`. We can nest multiple conditions into one boolean with 'logical operations': 'and', 'or', and 'not'. For example: `if (X and Y) and not Z:`.

Loops are one of the most fundamental tools in any programming language. Two types: `while` (do something while a condition is true) and `for` (do something iteratively across a list or over a range). Again, note, that you have to be careful with *identation*! Examples: `for s in list:` or `for i in range(4):`. A genomic example: `for each character in proteinseq:`, `if protein[i] not in protein`, do something. We can break out of a loop prematurely using the `break` command: this leaves our loop entirely. The `continue` statement jumps out of the loop and doesnt finish iterating. The pass statement is a *placeholder* - it does nothing. (for example: `try:` and `except`). Python is also slightly different to other programming languages in its `for` loops also have an `else:` clause. Functions are used extensively in programming languages: just as in maths - the function takes an input as an argument, and returns a single result. The functions that we create are called 'user-defined functions'. They a.) allow us to re-use bits of code without having to write multiple times and b.) allow 'abstraction' - easier to understand what a block of code does. Examples for DNA sequencing:

- A function which computes the GC percentage of a sequence
- Checking that a DNa sequence has an in-frame stop codon
- A function to reverse complement a DNA sequence.

The general structure of the function is:

```
def functionname(input args):
    #document the function here
    function code here
    return output
```

`help(functionname)` will return 'string that documents the function'. An important concept: the scope of a variable declaration - the contents within which the program remembers it. A varible can either have a local or a global scope: a global variable is defined outside all functions, and local within functions. Similarly, functions can only see variables which are local (passed) to them. A boolean function tells us if something is true or false and can be used to evaluate conditionals. An example of a function to check for stop codons (note `frame=0` is the default value if no frame value passed to the function):

```
def hasstopcodon(dna,frame=0):
    #this function looks for stop coons
    stopcodonfound=False
    stopcodons=['tga','tag','taa']
    for i in range(frame,len(dna),3):
        codon=dna[i:i+3].lower()
        if codon in stopcodons:
            stopcodonfound=True
            break
        break
```

A reverse complement function is very useful for making the reverse complement of a sequence. For example: `revseq=reversestring(seq), compseq=complement(revseq)`. An example of this, including joins and splits and, importantly, *list comprehension*. One final important thing for this section: a variable number of function arguments. For example: `def myfunction(first,second,third,*therest)`.

Modules are a way to put functions together in a file: python files with a `.py` extension. For example: `import dnautil`, when dnautil is in the PYTHONPATH or your current working directory. To check the path list: `import sys` and `sys.path`. If it doesnt contain the path to the location where your function file is, you can add it: `sys.path.append(path)` where (path) is the path to your functionfile/module. You can import all functions and their defintions using something like `from module import *`. Packages are a way to group modules together into a larger collection. For example: module name A.B designated a submodule named B in a package named A. Each package is a directory, which has to have a special file: __init__.py. This indiciates to python that the directory contains a python package to be imported. We can group modules into one package by putting the package .py files into the same subdirectory as the init file. We can import in two ways: e.g `import package.module`, or `from package import module`.

To read from a file: `open(filename, mode)`, where mode can be 'r' for read (default mode), 'w' for write to the file (overwrite), 'a' for append mode. The result of an `open` function call is a file object. We can use try and excepts to check if a file exists before we try to open it. A simple way to read the file is to iterate over it line by line: `for line in f: print(line)`. Another way is to use `f.read()`. We can go to specific lines of the file using `f.seek()`. We can use `f.write()` to write to a file using the 'w' or 'a' option with a file open. It's good practice to use `f.close()` in order to free up system resources.

To build a dictionary containing all sequences from a FASTA file: open the file, read the line: is the line a header? If yes, get the sequence name and create a new dictionary entry. If not: update sequence in dictionary. Then check whether there are more lines in the file: if not - close the file. We can then retrieve the key-value pairs. The sys module allows us to process command line arguments. When we run a script/program in the unix environment, there are standard streams recognized by the program:

- `stdin` - standard in - a stream of data (text) which a program can read. Unless redirected, standard input is expected from the keyboard which started the program.
- `stdout` - standard out is the stream which writes the output data.
- `stderr` - standard error is another output stream used to output error messages (in addition to `stdout` if required)

We can give two different outputs: stdout and stderr from the command line (redirecting two different streams):

```
myprogram | myscript.sh 1>programoutput.txt 2>errormessages.txt
```

Within the unix environment using `stdin`, you need to enter Ctrl+D to end the input. You can also call external programs from inside of Python. We can do this with the `call()` function in the subprocess module. For example: `import subprocess, subprocess.call()`. An genomic example of a subprocess call:

```
subprocess.call([''tophat'',''genomemouseidx'',''PEreads1.fq.gz'',''PEreads2.fq.gz''])
```

BioPython: founded in 1999 - a large collection of modules and scripts for bioinformatics and research. Includes parsers for various bioformatics file formats (such as FASTA, Genbank, etc) with tools for accessing NCBI, etc. Download from http://biopython.org/wiki/Download and `import Bio`. To align a sequence against any other reference sequence, we can use **BLAST** (BioPython has full functionality with the Blast servers): `from Bio.Blast import NCBIWWW` - the web methods from NCBI:

```
from Bio.Blastr import NCBIWWW
fastastring=open('myseq.fa'}.read()
results=NCBIWWW.qblast('blastn','nt',fastastring)
```

There are many parameters to set,although we choose the default ones. Blast may take a few minutes to come back, outputing an .xml file as a BLAST record (e.g. which species it matches what against). For alignment, you get HSP record : high scoring pairs for each separate alignment. Blast limits to 50 alignments. We can write simple scripts to write a threshold for E-values, etc. (i.e. unlikely to be chance matches).

A set of notes by Charlie **Spring 2017**

Notes on 'Algorithms for DNA Sequencing'

Fourth Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One: DNA sequencing, strings and matching

Most importantly, we'll learn how genomes can be represented as strings and substrings. There are two main technical problems: a.) read alignment, and b.) assembly. This is such a timely area of study because sequencing has become so cheap. DNA encodes your genome: the sum total of all your genes. Your genome is a recipe book - for the machines which do the work of building and maintaining you. The recipes are written in As, Cs, Gs and Ts. A-T and C-G (these are 'bases'). The fact that we can write DNA molecules as a string has huge implications. DNA sequencers are good at reading lots of short stretches of DNA to produce 'reads' which are many magnitudes of order shorter than DNA.

A string `S` is a finite sequence of characters - a set of Σ={A,C,G,T} in this case. $|S|$= the number of characters in `S` - `len(S)`. ε is the empty string: `len('')`. The left-most offset is 0. The concatenation of two strings glues them together: e.g. s + t. A substring of S is a string occurring inside S. e.g. s[0:6] is the same as s[:6] - if the zero is ommitted, it is an implied zero. A suffix occurs at the end of S. In Python: double and single quotes represent the same thing. Index with square brackets. `len('')` tells the length, and `print()` prints them. `.join()` joins a list together into one string. `random.choice('ACGT')` will randomly pick bases. Looping through with an underscore can be used when we dont need to reference the looped variable in the loop e.g. `for _ in range(0,10)`. Example function to find longest common prefix between two strings:

```
def longestCommonPrefix(s1,s2):
    i=0
    while i < len(s1) and i < len(s2) and s1[i]==s2[i]:
        return s1[:i]
```

A double equal sign tests equivalence. How about an example to get a reverse compliment? Create a dictionary called compliment to get a reverse: `compdict=['A':'T', 'C':'G','T':'A','G':'C']`. It is 'FASTA' files which contain these base sequences. **Cool trick - ! allows us to use command line within ipy notebooks!** For example: `!wget .....` How to read genomes into strings, and then print it out the first hundred bases with `genome[0:100]`.

```
def readGenome(filename):
    genome =''
    with open(filename,'r') as f:
        for line in f:
            if not line[0] == '>':
                genome+=line.rstrip()
```

How about we count the bases? `import collections`, `collections.Counter(genome)`. How do second generation sequencers work? Double stranded DNA gets split in half to act as a template. DNA polymerase takes bases to build the complimentary strand (of the template strand) piece by piece. Make DNA single stranded: deposit them into flat suraces (like a slide). Snap photo of terminated bases, iterating the process, to get a series of photos: one per sequencing cycle. One dot per template strand - billions of single stranded templates on a slide - photographing all at once, terminators keep the strands in sync and give us time to snap the photograph. However, there are sequencing errors potentially because a base is not terminated or is ahead of schedule. For each base call, the base caller reports a score on the probability that the base type is accurate. Reads are encoded within a FASTQ format. The second line is the sequence of bases, and the fourth line is the line of base qualities. Each base quality is an adjusted version of the probability that the base call is incorrect: $Q = -10log_{10}p$. Each value is ASCII encoded: each value maps to an integer on a corresponding lookup table. The most common way of encoding is 'Phred+33': take a value from the ASCII table and add 33 to it. Each FASTQ file comes in sets of 4.

```
def readFastq(filename):
    sequences = []
    qualities = []
    with open(filename) as fh:
        while True:
            fh.readline()
            seq=fh.readline().rstrip()
            fh.readline()
            qual=fh.readline().rstrip()
            if len(seq) == 0 ;
                break
            sequences.append(seq)
            qualities.append(qual)
    return sequences qualities
```

Convert Phred33 to their corresponding quality scores:

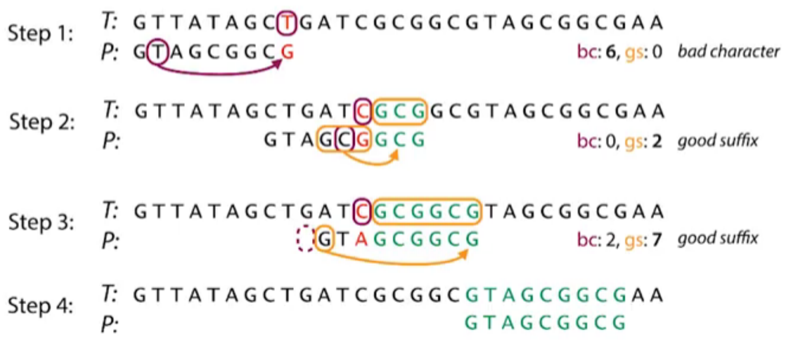
```
def phred33toQuality(qual):
    return ord(qual)-33
```

Why are we interested in the GC content? It's different from species to species - we are trying to figure out if the mix of different bases is moving along the read - can be useful for quality control to see if anything strange is happening. Note that the average human genome GC content is higher than 0.5. The base caller can sometimes report an N - when it has no confidence in the read.

```
def findGCbypos(reads):
    gc=[0]*100
    totals=[0]*100
    for read in reads:
        for i in range(len(read)):
            if read[i]== 'C' or read[i] == 'G':
               gc[i]+=1
               totals[i]+=1
    for i in range(len(gc)):
        gc[i]/=float(totals[i])
```

However, to infer meaningful information from these snippets, we must take these reads and stitch them back together to infer the sequence of the DNA. We do this with a reference genome - this is typically called the 'read-alignment' problem (unless we have no reference - then we use 'de novo' assembly). We're looking for the place where the sequence looks most closely to the reference genome (which is about 3 billion bases long). In Python, `string.find('word')` returns the index of the first occurrence.

Steps to Suffixing


Step 1:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
bc: 6, gs: 0 bad character
Step 2:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
bc: 0, gs: 2 good suffix
Step 3:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
bc: 2, gs: 7 good suffix
Step 4:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG

```
def naive(p,t):
    occurrences=[]
    for i in range(len(t) - len(p)+1):
        match = True
        for j in range(len(p)):
            if not t[i+j] == p[j]:
                match = False
                break
        if match:
            occurrences.append(i)
```

Sequencing errors and the fact that we are not using the correct reference genome mean that we might not match a high percentage. We should also try and align the reverse complement of the DNA also (i.e. is 'strand aware'). However, these techniques are restrictive in that they are focusing on exact matches: we want to allow for approximate matches through sequencing errors.

Week Two: Preprocessing, indexing and approximate matching

The naive algorithm is slow and only considers exact. Another choice: Boyer-Moore - fast, practical and simple. Indexing is the reason why you type a query string into a search engine, you get a result back instantly. **Boyer-Moore**: Similar to naive exact matching - it skips many alignments that it doesn't need to examine - the benchmark exact matching algorithm. (i) It learns from character comparisons to skip pointless alignments - try alignments in left to right order, but try character comparisons in right-to-left order. (ii) The bad character rule of Boyer-Moore: upon a mismatch, skip alignments until a mismatch becomes a match, or the substring (P) moves past the mismatched character. (iii) The good suffix rule: skip until there are no mismatches or P moves past t ('tries to keep the good matches a match'). Each of these rules will tell us some amount that we can shift P - take the maximum of the two.

Repetitive elements of the genome creates ambiguity for the read-alignment problem - where does our read come from? For example: about 10% of the human genome is covered by Alu repeats.

```
def boyer_moore(p,p_bm,t):
    i=0
    occurrences=[]
    while i< len(t)-len(p)+1:
        shift=1
        mismatched=False
        for j in range(len(p)-1,-1,-1):
            if not p[j]==t[i+j]:
                skip_bc = p_bm.bad_character_rule(j,t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched=True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs=p_bm.match_skip()
            shift=max(shift,skip_gs)
        i+=shift
    return occurrences
```

The BM algorithm pre-processes to build lookup tables to make it faster for bad character and good suffix rules - this 'amortizes' the cost over time. An algorithm which does pre-process the text (T) is called 'offline' (one that does is 'online'). For example, the naive algorithm is 'online', BM is 'online' because it `preprocesses only the pattern P`. Two concepts which are useful: 'ordering' as per the index of a book, and 'grouping' as per the areas of a grocery store. When p matches within T, we call it an occurrence - but not all index hits lead to matches. A 'k-mer' index - an index built by taking all k-mers of text T and adding them to a data-structure (‘multi-map’) which relates to them where they occurred in the text - it associates keys with offset values in the genome. We then order the indexed datastructure, and query the index with this 3-mer. The number of queries that we need to make is approximately equal to $\log_2(n)$ bisections. Python makes a bunch of tools for binary search such as the `import bisect` module. For example: `bisect.bisect_left(a,x)`: gives the leftmost offset where `x` can be inserted into `a` in order to maintain order. Hash tables can be used to implement multi-maps: used to represent sets and maps in practice. The hash-function h maps 3-mers (for example) to ‘buckets’ in the hash table - we then append the corresponding key-value pair to that bucket. The python dictionary type is a type of implementation of a hash table. For example:

```
t='GTGCGTGTGGGGG'
table={'GTG':[0,4,6],'TGC':[1],'GCG':[2], 'CGT':[3],'TGT':[5],'TGG':[7],'GGG':[8,9,10]}
```

Before, we built our map table of every k-mer. What if we didn’t take every k-mer? What if we took every other k-mer (such as those which started only at even-offsets?). This makes the index smaller, and is a little faster to query - but wont this cause us to miss some of the matches? What about if we only chose every n-th k-mer? We can also build this over subsequences. Substrings are always subsequences, but subsequences are not always substrings. This kind of technique can increase the *specificity* of the filter provided by the index - when we get an index hit it is going to lead to a correct verification more of the time than if we had taken the characters to be consecutive. Indexes used in genomic research: another idea - suffix index. Instead of extracting every substring and putting it into an index, we could take every suffix - all suffixes in alphabetical order - query it using binary search. This can be represented by just one integer - the *offset* at which it occurs - this leads to a more manage-ably sized data structure (growing linearly, rather than quadratically). Another example is the suffix tree (like the suffix array), but organizes them using the principles of grouping (whereby the suffix array puts everything in order). Another type is the FM index: based on the BW transform: this is much more compressed and can easily be fit into memory on a local computer (commonly used in BowTie and BowTie2). All of these methods are *exact*. But we might want *approximate* methods due to sequencing errors or natural variation. There might be deletions, insertions or substitutions. There are multiple of these measures of ‘distance’: e.g. for strings X and Y where ($|X| = |Y|$), the **Hamming distance** is the minimum number of substitutions needed to turn one into another. The **Levenshtein** distance is the minimum number of edits (substitutions, insertions and deletions) needed to turn one string into another. We can very easily modify the naive exact algorithm above to turn it into a naive Hamming algorithm as follows:

```
def naiveHamming(p,t,maxDistance):
    occurrences=[]
    for i in xrange(len(t)-len(p)+1):
        nmm=0
        match=True
        for j in xrange(len(p)):
        if t[i+j]!=p[j]:
            nmm+=1
            if nmm>maxDistance:
                break
            if nmm <= maxDistance:
                occurrences.append(i)
    return occurrences
```

The pigeonhole algorithm is a method which allows us to extend our exact algorithms to approximate. For example: in the one edit case - if we split P into u and v - then either u or v will be an exact match. This can be extended to k-edits - **at least** one of these k+1 partitions must appear with no edits. The pigeonhole principle - 10 pigeons, but 9 holes. In our problem, we have 9 holes, but 8 pigeons. It is this bridge which allows us to combine our approx algorithms with the aforementioned techniques (i.e. BM, with respect to hits and occurrences, etc). For example: if we’re searching for approximate matches of P within T allowing up to k mismatches, then we should first divide P into k+1 partitions, and at least 1 partition will have no mismatches.

Week Three: Edit Distance, Assembly, Overlaps

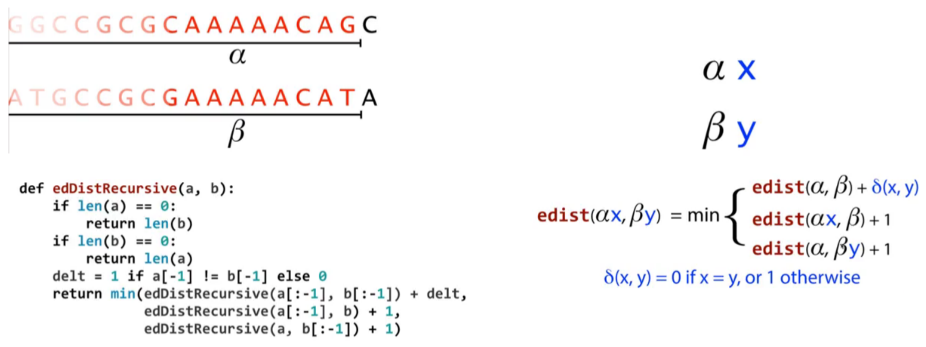
Dynamic programming algorithms allow us to look for approximate occurrences of a pattern in text, including occurrences with insertions and deletions and allow us to calculate flexible string similarities with appropriate penalties. The Hamming distance is actually quite easy to to calculate, as shown in the function above. How about an algorithm to calculate the edit distance between two strings? More complicated. What's the relationship between the Hamming and edit distance? Are they equal? Is one greater or equal? The edit distance will always be less than or equal to the Hamming distance. Can we put a lower bound on the edit distance between X and Y? In order to edit X and Y, we at least have to introduce as many edits to make them the same length - and then more maybe to make them the same sequence. Tactical insertions potentially allow us to reduce the distance to zero much more quickly than substituting and deleting.

This function (delta) is like a recursive function, as shown by the Python code. However, these functions are typically extremely computational and take a very long time - e.g. `edDistRecursive('ABC','BBC')` this initial call will make 3 recursive calls, which will each make 3 more recursive calls, etc. However, a lot of these calls are the same - and it would be useful to remember what calls are made. In order to prevent this, we can rewrite the function in terms of a matrix where the characters label the rows and columns: each column corresponds to a particular prefix (where first row and column is the empty string). This type of dynamic programming is very useful for calculating these types of sequencing applications.

```
def editDistance(x,y):
    D=[]
    for i in range (len(x)*+1):
        D.append([0]*len(y)+1)
    for i in range(len(x)+1):
        D[i][0]=i
    for i in range(len(y)+1):
        D[0][i]=i
    for i in range(1,len(x)+1):
        for j in range(1,len(y)+1):
            distHor=D[i][j-1]+1
            distVer=D[i-1][j]+1
            if x[i-1]==y[j-1]:
                distDiag=D[i-1][j-1]
            else:
                distDiag=D[i-1][j-1]+1
            D[i,j]=min(distHor,distVer,distDiag)
    return D[-1][-1]
```

Edit distance allows us to find the distance between two strings, but what about if now apply it to approximate matches? Revert to looking for patterns (P) in a string of text (T). Now, initialize the first row with all 0s and the first column as ascending integers as before. We can then use the 'traceback' path to find the substring which has the minimum number of potential edits. This traceback path also tells us the shape of the vertical alignments (having looked in th final row to find the smallest number of edits). This is a lot of work: proportional to the size of the matrix (len(P) times len(T)) - with no skipping. This algorithm is not practical on its own for the read alignment problem. It tends to be used in addition to other techniques, such as indexing, the pigeonhole principle, etc. We can use these dynamic programming tools for both global and local alignment problems. Global alignment: edit distance penalizes all types of edits the same amount - no difference between substitution and insertion, etc. What if certain base to base substitutions were more likely, and we wanted to penalize them less? This is the case with genomic sequencing: we can divide all DNA substitutions into two categories: transmissions and transversions. For substitutions that convert a purine to a purine or pyrimidines: this is a transition - all other are transversions. In reality, transitions are twice as frequent as transversions - we might want to penalize transversions more then. With respect to the human reference genome, the substitution rate is something like 1 in 1000, the indel (insertion/deletion) rate is 1 in 3000. To incorporate this, we can use something like a penalty matrix. Incorporating this into our edit distance function is very simple: instead of our delta function or adding 1, for example, just add the relevant lookup from our penalty matrix. Global alignment gives the user the ability to set penalties applicable to the

Edit Distance Suffixes



```
def edDistRecursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

$$\text{edist}(\alpha x, \beta y) = \min \begin{cases} \text{edist}(\alpha, \beta) + \delta(x, y) \\ \text{edist}(\alpha x, \beta) + 1 \\ \text{edist}(\alpha, \beta y) + 1 \end{cases}$$

$\delta(x, y) = 0$ if $x = y$, or 1 otherwise

biological problem at hand. Local alignment - we're trying to identify the substring of x and y which are most similar to each other. Instead of using a penalty matrix - use a scoring matrix, where we give a positive bonus for a match and a negative penalty for all other types of differences. We can then use the same traceback procedure as above where we stop when we reach an element whose value is zero. (Summary: Local alignment is compared with similiarities between substrings of x and y). To modify the `editDistance` function above to a global alignment one, all we need to do is define a score matrix and add the appropriate elements of this matrix to each distance when a character is skipped, substituted, etc.

What do the read alignment tools do in practice? They make use of indexing and dynamic programming in conjunction. Indexing allows us to rapidly hone in on candidate locations and acts like a filter. However, in practice, our matrices for each read are huge, because the reference genome is huge also and this would take years without an index. Indexes are really good at finding exact matches, and then use dynamic programming to figure out whether the pattern as a whole has an approximate match to that index hit. On the one hand, the index is very fast and good at narrowing down the space of places to look, but it doesn't handle mismatches and gaps naturally at all. And on the other hand, dynamic programming does very naturally handle mismatches and gaps (but is slow).

Until now, we've been focusing on the problem of 'alignment' - where we're given a reference genome. The assembly problem relates to when we have no reference - this is also called 'de novo'/shotgun assembly. The figure shows us the task behind the goal of assembly. One important concept: coverage - which relates to the number of reads we have over one base in the genome - i.e. the number of 'votes' for what a specific base should be. We can average this into the 'overall coverage'. How can we piece together the genome sequence? **The first law of assembly: If a suffix of read A is similar to a prefix of read B, then A and B might overlap on the genome**. Why are there possibly differences in coverage over one specific base? i.) sequencing errors, ii.) polyploidy - humans have 2 copies of each chromosome - and copies can differ because they have different bases at that position. **The second law of assembly: more coverage leads to more and longer overlaps.**. The way to approach this problem is through one big structure - because the overlaps involve the same read - we need to build a directed graph where the nodes and edges have meaning. But what do we mean by an overlap? Some overlaps are less convincing than others: a length 1 overlap could be just a coincidence which we don't want to consider an overlap - we need to set a threshold and say as long as the overlap is more convincing, we can count it as an overlap and draw the corresponding directed edge (e.g. have an overlap of at least length 4). We can walk through the nodes to infer the sequence of the original genome. An example function is something like:

```
def overlap(a,b,minlength=3)
    start = 0
    while True:
        start=a.find(b[:minlength], start)
        if start == 1:
            return 0
        if b.startswith(a[start:]):
            return len(a)-start
        start+=1
```

```
def naive_overlap_map(reads,k):
    olaps{}
    for a,b in permutations(reads,2):
        olen=overlap(ab,minlength=k)
        if olen>0:
            olaps[(a,b,)]=olen
    return olaps
```

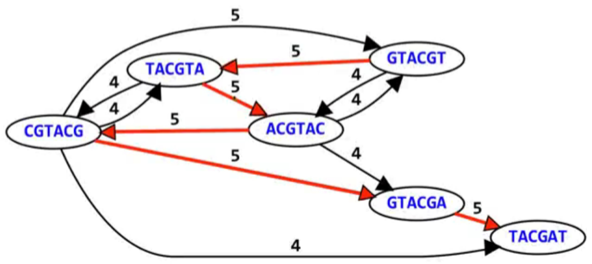
Week Four: Algorithms for Assembly

The problem that is critical in assembly is the 'shortest common superstring' (SCS) problem - the shortest possible string that contains all of our input strings as substring. For example, given a set of strings, S, find the shortest common superstring: where the set is `baa`, `aab`, `bba`, `aba`, `abb`, `bbb`, `aaa`, `bab`. This concatenates to: `baaaabbbaabaabbbbbaaabab`, but the shortest common superstring is `aaabbbabaa` - there is no shorter string which contains all these substrings. This is at the heart of the assembly problem. However, there are some downsides to this problem - it's not tractable with no efficient algorithms for it (NP-complete) - as input strings grow, it'll slow considerably. One solution is to just pick the order for strings and then construct the superstring. However, this is extremely slow - if S contains n strings, there are n! possible orderings. We can use the overlap function from above to implement a brute force method which will be extremely slow, but correct. A faster algorithm: a 'greedy' common superstring - which chooses largest possible overlap at each point and then moves forward. This is faster than the naive approach - but at a cost - doesn't always find the correct answer - reporting a superstring which is not necessarily the shortest one.

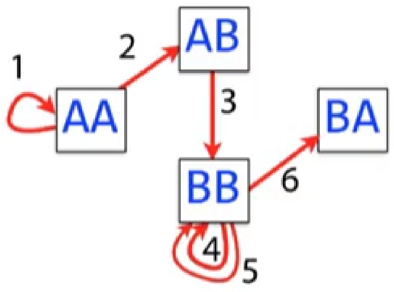
```
def pick_maximal_overlap(reads,k):
    reada, readb=None, None
    best_olen=0
    for a,b in itertools.permutations(reads,2):
        olen=overlap(a,b,min_length=k)
        if olen>best_olen:
            reada, readb = a,b
            best_olen=olen
    return reada, readb, best_olen
```

So far: we've seen greedy and brute force, but both have downsides. When the genome is repetitive, the SCS of the reads is repetitive - we can't tell exactly how many copies we have in the original sequence - the reads just aren't long enough to tell us how long it is, causing ambiguity and its not easy or possible to reassemble the genome. **Third law of assembly: repeats make assembly difficult**. You don't want to collapse them down into too few a number of repeats e.g. - 'this is a serious serious serious problem' into 'this is a serious serious problem'. About half the genome is covered by repetitive elements. A De Bruijn graph: assume that sequencing reads consist of each of the k-mers once - for each k-mer make an addition to the corresponding graph. From a De Bruijn graph, you can re-construct the original graph for walking through each graph following each edge as we go, respecting the edges: it uses each k-mer exactly once. **This is called an Eulerian walk** through the node to node to give us the reconstructed genome sequence back again.

Directed Graph Example



A Eulerian walk



However, this doesn't allow us to escape from the third law - that repeats make things difficult. However, there might be multiple Eulerian walks which creates ambiguity for sequence reads. Decreasing the length of the k-mer increases the chance of there being multiple different walks - the curse of the repetative genome. The De Brujin graph is a very common way to represent the assembly problem - used in a lot of softwares (internally) - despite the fact that the SCS and Eulerian walk are flawed formulations for the assembly problem, the overlap (overlap-layout consensus assembly - OLC) /DB (De Brujin graph assembly - DBG) graph are still going to be very useful to us in practice. How do real software tools work? In practice, the graphs are extremely messy for lots of reasons like sequencing errors or other 'dead ends'. Another reason is polyploidy. We can deal with these problems by chopping our assembly into pieces which have no ambiguity: we can still put portions of the puzzle together - these are called 'contigs': partial reconstructions which we can put together unambiguously. Therefore, an assembler reports a set of 'contigs' - even the Human Reference Genome still has holes in it, which is something we have to live with due to repetitive DNA. One solution is to make the reads longer - to glue it to some surrounding non-repetitive sequence. This is like making the puzzle pieces bigger. There doesn't exist technology which can collect reads longer than tens of thousands of bases. Paired-end 'Template' - can give us a bit more information per sequence. It gives about twice as many bases and is extremely common in practice. It doesn't sacrifice much in terms of accuracy or speed. Other technologies are getting better, but are much slower - sequencing one molecule at a time (rather than a lot in parallel) with lots of mistakes.

```
def greedy_scs(reads,k):
    read_a,read_b, olen=pick_maximal_overlap(reads,k)
    while olen>0:
         reads.remove(read_a)
         reads.remove(read_b)
         reads.append(read_a+read_b[olen:])
    return ''.join(reads)
```

A set of notes by Charlie **Spring 2017**

Notes on 'Command Line Tools for Genomic Data Science'

Fifth Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One: Basic Unix Commands

The most important first step is to think about how files and directories are organized: at the top is the 'root', and then directories like branches - subdirectories - and other files. The operating system is how you communicate with the computer - via an interpretter. Files with various files have extensions - and the OS tells the computer what program to utilize to access these files. However, the typical range of operations is quite limited - under some circumstances, you might want to do your own set of operations - and this is why we need Unix. The way that we communicate with the Unix operating system is via an interpretter - or 'shell' - we give it commands and it responds (if it understands it!). For example: `date` - gives us the date, and `echo hello` responds with 'hello' - it understands. If we try something deliberately obtuse, it will not work - such as `foo` - command not found. One of the most useful commands is `pwd` - print working directory - the current position in the filesystem. Any particular location in the filesystem can be represented by its direct path to the root of the system (as shown by `pwd`), where every level of the heirarchy is represented by a '/'. We can move from one location to any other one using `cd`. For example: `cd /Users/username/Desktop/` and then `cd /Coursera` to get to /Users/username/Desktop/Coursera. There are two special directories. The current directory is represented as `.` and the parent directory is represented as ... `ls` gives us a 'list' of all files currently in a specific directory. If we wish to go the parent directory: `cd ..` - it takes us 'up a level'. `ls` lists files in alphabetical order, with upper case first. Several command line parametres can be used - `ls -l` lists additional information about the files (such as groups, whether it's a file or subdirectory, modification times, etc). To get reverse chronological order: `ls -lt`. We can find manual pages on any specific command by typing `man ls` - for example - to get the manual pages for `ls`.

Effectively naming files is critical - especially when we have a large number of files of a specific type. This is where `wildcards` come in - for example - ***** - which represents any combination of letters or numbers following the specified input characters. To replace just one character, we can use a question mark - **?** - which stands for just one part of a filename. A range of characters can be represented in square brackets using regular expression type commands. For example - any range of alphabetic characters by **[a-z]**. We can also do combinations of the above. Curly brackets allow us to specify a number of options: for example - `ls {pear,peach}.genome` will return us pear.genome and peach.genome.

How do we create and remove content? `mkdir` makes new directories. We can copy multiple files using the `cp` command - this duplicates between directories. `cp` can also be used to copy multiple files e.g. `cp fileone.genome filetwo.genome newsubdirectory`. These can be combined with the wildcards above. We can move, as opposed to copy files using `mv` (and again, several files at one time using wildcards as before). If we try and move twice - it wont let us, because at the second iteration, they've already been moved. We can remove files using `rm`. **However, this operation is final!**. Using the command `rm -i` will give us a second Y/N chance. This doesn't allow us to remove directories - first remove all the files in the directory, and then remove it (`rmdir`). Alternatively, we can 'recursively' remove the files with the `rm -r` command.

An easy way to look at the contents of each file is using `more`. For example: `more apple.genome` allows us to look at the contents of apple.genome one page at a time. `more` and `less` commands are closely related:

- The `more` command is used to view text output of commands or files one page at a time. When the user is ready for the next block of text, they press the space bar.
- `less` extends the functionality of more by adding forward and backwards movement, the ability to search multiple files, view gzipped files without uncompressing them first, set marks for navigation and can invoke an editor when viewing files.

If the file is very long though, perhaps we only want to see the portion at the top: this is why we need the command `head`. For example - to get the first 50 lines: `head -50` will give us the first fifty lines. In a similar way, we can look at the last few lines of a file with e.g. `tail -15` will extract the last 15 lines. We can concatenate multiple files with `cat`. To concatenate all genome files: `cat */*.genomes`.

The next logical step is *redirecting* content. `wc` - word count - gives us four pieces of information: the number of lines in the file, the number of words in the file, the number of characters, and then, finally, the name of the file. `wc -l` gives us just the number of lines in the file. However, what if we don't want content directed to the terminal? Redirect using >.For example: `wc -l filename.extension > nlines` **redirects** the output to the file called 'nlines'. We can use < to redirect the standard input from the terminal to a particular file.

Less extends the functionality of more by adding forward and backwards movement, the ability to search multiple files, view gzipped files without uncompressing them first, set marks for navigation and can invoke an editor when viewing files. The > sign is used for redirecting the output of a program to something other than stdout (standard output, which is the terminal by default). >> appends to a file or creates the file if it doesn't exist. > overwrites the file if it exists or creates it if it doesn't exist. Examples:

- `$ ls > allmyfiles.txt` creates the file 'allmyfiles.txt' and fills it with the directory listing from the ls command.
- `$ echo "End of directory listing" >> allmyfiles.txt` adds 'End of directory listing' to the end of the file 'allmyfiles.txt'.

Piping '`|`' takes one output from a command and uses it as an input to another command. For example: `ls | wc -l` takes the output from `ls` and uses it as input to `wc -l`.

But what about querying content? `sort` allows us to sort the files alphabetically. We can also sort in reverse alphabetical order (`sort -r`). What about if we want to sort on a numeric column in the file? `sort -k 2nr file` sorts the file by columns (`-k` option) and the `2nr` tells us to sort by the second column, which is numeric, and to go in reverse. We can stack the columns to sort by. `sort -u` eliminates duplicate entries in the list (unique). To sort case insensitively, use `-f`. To add reverse, append an `r` to any of these commands (e.g. `-fr`). As with the above unix commands, we can putput this to a new file e.g. `sort filename > newfilename` to output to a new file called newfilename, and we can pipe it as input e.g. `sort filename | more`. We can use the `cut` command to look at individual fields in specific files. It takes one file, and extracts a particular range. For example: `cut -f1 filename` gives us the first column from filename. We can cut columns 1-2 with `cut -f1-3 filename`. We can also cut on different delimiters. We can combine all of the commands we've seen, to sort by unqiue, pipe it, and stdout it. Another very important command is `grep`, which allows you to search one file or multiple files for lines that contain a pattern. What about comparing content? `diff` allows us to compare files line by line: it lists all of the differences between the files line by line at the command line and tells us the position at which the changes have been made. `compare` allows us to explicitly compare the two files: if a row occurs in just the first file - it goes into column 1, if just the second, column 2, and if it's listed in both files, it goes into the third column (however, the files must be sorted in the same order first for this to work correctly - otherwise the same items will appear in different columns on different rows).

One critical idea is 'archieving content'. What if we want to compress the content of *filename*? We can create an archieve creating `gzip filename.extension`. This will add a .gz file extension, and compresses the file. To uncompress it: `gunzip filename.extension.gz` - and there is no loss of information. Need even more compression? `bzip2 filename.extension`. Uncompress with `bunzip2`. What if we want to compress multiple files together? First, we will need to create an archieve which links together all the files - then compress that. In order to create the archive (here called Archieve.tar), `tar -cvf Archieve.tar filename1.extension filename2.extension filename3.extension`. The options: `c` pulls together the archieve, `v` is verbose to say we'll provide a list of files to include, and `f` means there will be a target file which follows (Archieve.tar). However, there is at present no compression (until we `gzip` it). This allows us to sytematically compress and archieve files. To 'un-tar' the archieve, we need a different set of commands: `tar -xvf Archieve.tar` - extract the following file to the target. `ZCad` allows us to inside any file which has already been compressed. Some genomic examples:

- How many chromosomes are there in the genome? Use the command `more` on a FASTA file - where headerlines identify and follow as the sequence. Lets count how many header lines we have by counting '>' with `grep -c ''>'' filename.genomefile` where the `c` is for count.
- How many genes and variants are in the species? We can do this by `cut`ting a gene-name column, and counting the number of unique gene names. For example: `cut -f1 filename.genes | unique | wc -l`, and to get the list printed out: `cut -f1 filename.genes | sort -u`. A similar operation finds the

number of variants (just a different column). These can then be extended and combined as appropriate across species.

- What plant systems contain a smell gene? `grep` for the word 'Smell' across all directories in files which have gene extensions: `grep Smell */*.genes`. This returns all lines somewhere which contain the word 'Smell'.
- What genes are in common? To do this - first cut by specific genes across multiple files, use the `comm` command to compare, and ignore all the lines that are in common across both files (pipe to `uniq`)

Week Two

A genome is a totality of genes in a set, organized in chromosomes - 22 pairs of autosomes and 2 sex chromonsones. Along the genome are genes: beads along the string, organized on both strands of the DNA molecule. Genes are made up of different types of blocks: exons split by spacers (introns) - genes have to take a form of their own by the process of gene expression: the production of a protein. The first step is transcription, where the molecular mechanism creates a copy of every gene as a single stranded RNA module: pre-mRNA. This is modified - capping/chopping/splicing (where the informative blocks get connected together) to get RNA - these are transcripts. The main question we need to address: What are the genes? What are the proteins? What are their sequences?

The mRNA gene sequences are strings over an alphabet of 4 letters - A,C,G,T(U). How do we figure out where these strings are? Sequencing instrument: start at the beginning and tell us every letter - current technology can only split it into a number of pieces (fragments) which are then sequenced. The fragments are roughly the same in length (normally distributed) - each can be amplified and sequenced from both (paired-end reads) or one end (single-end reads). We need to then reconstruct the original molecule. This gives us an 'assembly graph' - the task of the bio-informatition is to traverse this graph to include all the reads. Sanger sequences are longer reads but more expensive (and slow). Illumina sequencing: shorter, faster, cheaper - but more challenging for bio-informatics of assembly. For Sanger sequences, the prevailing format is Fasta/-Pearson format - which has one header line, starting with > and an identifier and some information about the reads. This is followed by the sequence (with no blank lines). Example:

```
>gi|28178860|ref|NM_000586.2| Homo sapiens interleukin 2 (IL2), mRNA.
```

Fastq is the format for next generation sequences where the header is preceded by '@' followed by an identifier (geometry, cluster, etc.). Every record contains 4 lines. The last two letters include a / to tell us what read it is in the pair (for paired end reads). Following the header is the sequence. The third line either repeats the header or is a '+'. The fourth line is a letter - one letter for each read - which gives us the quality of the read. Header:

```
@UNC14-SN744:186:D078MACXX:1:1101:1203:1868/1
```

The base quality score is calculated (where p_b is the probability that the call at base b is correct), where the maximum quality in practice is about 40 and anything below 20 is unacceptable as:

$$Q_{sanger} = -log_{10}p_b \tag{5.1}$$

The next step is to annotate important features across the genome - genes, exons, promoters, binding sites, translation start/stops, etc. Gene annotation - exon/intron structure, strand, start and end sites. Browser Extensible Data format (BED) - 3 columns signifying information in the genome: #, start, end. The extensible format can add more fields with more information, such as scores, strands, thickstarts and thickends and an rgb color representation. The score is typically a value indicating how intense the colour should be. Thickstarts and thickends tell where the feature can become thick. 0-based counts spaces, and 1-based counts bases. Here, each entity is a separate entity. A clustering or interval can be put at the end of existing BED records. The GTF format (genomic transfer format) provides more columns and is more structured (.e.g source, score and beginning and end of specific features, etc) and is 'one-base'. A related format: GFF3 -

genomic feature format - preceded by a line which says '## gff-version 3' - columns for exons, source (gene finder), type of feature (mrna/exon), start and end coordinates, score, coding frame, strength, etc.

Alignment: map letters of reads to the genomic sequence, matching each base using some spacers to fill in gaps, taking into account differences such as polymorphisms and sequencing errors and introns etc. If a letter is in a (reference) genome but missing from a read, we call it a 'deletion'. If there's a letter in the read but not in genome: insertion. If genes trade places: substitution. Each read can map contiguously along the genome. The mRNA splices together information at different positions - if a read belongs entirely within an exon - a contiguous alignment. However, if the read spans the boundary between exons, theyre divided along the genome - a 'spliced' alignment. We can have 'properly paired' and 'non-concordant' mapping. The standard format for alignments of NGS is 'SAM' (compressed to 'BAM'). The first few lines represent a header (each line has an): is it sorted? what's the identifier (orderd by chr)? whats the length (in mega-bases)? how was the file generated (e.g. TopHat - spliced alignment program) and with what options? Following headers, we have alignments: one line per alignment. Does the pair have a chance of being concordent, etc? SAM formats have a large number of flags: multiple pieces of information represented in binary (base 10) e.g. `0x1`: multiple segments, `0x2`: each segment properly aligned, `0x4`: segment unmapped, etc. For example: 0011 = paired, proper pair, Mate mapped. There are specific ways to interpret each sequence of codes. CIGAR entries: whether matched, inserted, deleted, skipped, clipping, padding, sequence match or mismatch. The main source for all data of this type is https://www.ncbi.nlm.nih.gov/. It contains a huge wealth of datasets for various species across all of the aforementioned fileformats (with accompanying meta-data about the samples). One extremely useful way to get data from internet addresses at the command line is 'wget'. For example, for a specific NCBI transfer:

```
wget ftp://ftp-trace.ncbi.nih.gov/sra/sra-instant/reads/ByRun/sra/SRR304976/SRR304976.sra
```

Alternatives with slightly different functionality are `axel`, `scp` and `rsync`. Genomic data can also be downloaded from the USCS Table Browser: https://genome.ucsc.edu/.

SAMTOOLS: command line modules for manipulating SAM and BAM files. Usage: `samtools <commands> [options]`. Commands for indexing, editing, file operations, statistics, viewing and converting (e.g. to Fastq). Lets assume that we have `example.bam`: how many alignments? `samtools1 flagstat example.bam` gives us this and other summary statistics about the alignments. `nohup samtools1 sort example.bam example.sorted` will sort and give us a new, sorted output. The `nohup` command prevents the command from being interrupted, even after the shell session is closed. Remember: `zcat` allows us to view the contents of a file without unzipping it. `samtools` is also extremely proficient at merging files: for example - `nohup samtools1 merge file1.bam file2.bam file3.bam`. `samtools view` allows one to view allignment, allows conversion between BAM and SAM, and allows extraction only within a range of the genome: `samtools1 view -h example.bam | more` gives us the SAM format with header (`-h`) information. We can save it out to a SAM by changing it to:`samtools1 view -bT reference.fa example.sam > example.sam.bam`. We can also get specific parts ('region') of the alignment on indexed (`samtools1 index example.bam`: `samtools1 view example.bam 'chr22:24000000-25000000' | head`.

`BEDtools` is a versatile suite to perform arithmetics and manipulation of intervals or groups of intervals along the genome - 'the Swiss-army tool for genome feature analysis'. It also easily converts, and extracts sequences in a range or interval. It allows one to intersect, merge, count, complement, and shuffle genomic intervals from multiple files. `bedtools intersect` allows us to show overlaps, and to view and convert GTF and BED formats. Format conversion options: `bamtobed`, `bedtobam`,`bamtofastq`,`bed12tobed6`. To convert to bam to bed: `bedtools bamtobed [options] -i bamfile`.

Week Three

Each individual genome is unique: differences are responsible for observable traits and hidden ones also. 1% differences between humans and chips, and 0.1% between humans. Big projects such as HapMap and the 1,000 genomes project have aimed to map this entire variation. A variant which occurs in at least 1% of the population is called a polymorphism. How do we identify this variation? Extract DNA, sequence it (Illumina), map to genome using alignment algorithm, and analyze. At every position, we are looking at all aligned reads and base at that position. Shotgunning - sequencing from one end to the next, is expensive. We expect that the variants which are disease causing will be in protein encoding parts of genes - sequence these portions which correspond to genes - this is called whole exon sequencing. As above, the standard alignment for NGS is SAM/BAM format: header (information on file), followed by a number of sequences, followed by program/command line options to create this file, then one line for each alignment. A number of optional fields give us information on the number of hits, etc. The SAMtools `mpileup` format is used for representing variants with a number of fields seperated by tabs - chromosome number, position (1-base co-ords), where the base is, the number of aligned reads (depth), a string of characters to inform us about the bases of the reads at that position (coded), and the specific position of that particular letter in the read.

Sequencing variation can also be stored in `VCF` format - a fairly dense file with a preamble (compressed to BCF). The first column is the chromosome, the second is the position, the third is an identifier (e.g. in the 1000 genomes), the fourth is the letter of the reference of the variance and the fifth is the alternate, followed by the quality, the filter and information about the reads. Finally - format (template) and genotype information. It *only* contains information on positions where there is variation (info, filter, format). The format is simply a template for genotype information. The simplest type of variation is a SNP - single nucleotide polymorphism - a substitution when the letter in the reference has a different variation to the sequencing reads. There can also be a ‘Mix’ - a more complex example where variants have substitutions and deletions.

`Bowtie` - A fast tool to map a large number of sequences using a compressed representation of the genome as an index. Looks for ‘contiguous matches’ with only a small number of insertions and deletions. `bowtie2-build HPV_all.fasta hpv/hpv` which will gives a number of files which collectively represent the genome indexes in directory ‘hpv’. `bowtie2` has an extremely large number of options - query input file format - presets (alignments to go end-toend?) - portion of input reads (how fast or sensitive?) - parameters to control the scoring scheme? Example usage: `bowtie2 -p 4 -x /data1/igm3/genomes/hg38/hg38c exome.fastq -S exome.bt2.sam`. The output tells us how many were unpaired, how many aligned 0 times, how many aligned 1 time and more than 1 time. We can also try this with the ‘local’ option: `bowtie2 -p 4 --local -x /data1/igm3/genomes/hg38/hg38c exome.fastq -S exome.local.bt2.sam`. This will give us more sequences which align more than one time, and fewer which align exactly one time. This is because you can find more (partial) matches for every read.

`BWA` (written by Heng Li) can create alignments - you first need to index the genome using one of the three possible alignment algorithms e.g. `bwa index_index HPV_all.fasta`. Lets try to map these reads - with 3 options for alignment (i.e. `bwa mem`). This has a large number of options related to scoring, threads, chain length, input/output options, etc. To map the exon reads to the human genome: `bwa mem -t 4 hg38c.fa exome.fastq > exome.bwa.sam` (4 threads option - faster). We can then convert this into a bam file for further analysis and view with `samtools`: `samtools view -bT exome.bwa.bam | more`. `SAMtools mpileup` is used for variant calling - this creates at every base a tally of the information present in the reads at that position (optionally calculating a genotype likelihood). The output format is either mpileup or DCF/VCF. The typical execution: `samtools mphileup [options] in1.bam [in2.bam [...]]`. A set of important options: -g/-v will output a -BCF/VCF for optional genotype likelihoods. The file needs to be sorted and indexed (e.g. `samtools index filename.bam`. `BCFtools` usually operates on the output from the mpilup/samtools program, and uses this information to make a call as to whether the position contains a variation. Just like `SAMtools` - a suite of packages for manipulating VCF files - `bcftools <command> <argument>`. One example: the bcf is all in binary - lets transform it into a viewable file - `bcftools view filename.bcf`. The second package that we are interested in from the `BCFtools` suite is the genotype calling package: `bcftools call [options] <in.vcf.gz>`. Use the `mpileup` output to make variant calls using `BCFtools`. Example: `bcftools call -v -m -O z -o sample.vcf.gz sample.bcf` (-v only write lines which correspond to variants, lets use the more recent version -0 z for vcf compressed, -o to specify where output goes). Then: we have the preamble (info on reference genome), template for alternate alleles, information on the algorithms used for variant calling, the format lines, scores for variant calling, and lastly - one line for every variant which has been called. From this we can see deletions and substitutions, etc. Lets now put together an application for variation and variant calling:

- `samtools index sample.bam` to create an index file.
- `samtools mpileup -g -f Homosapiens_assembly19.fasta sample.bam > sample.bcf`: use `SAMtools` to create a bcf file which has one line for every position in the genome which contains align reads, and has genotypes which are going to be used by variant calling.
- `bcftools call -v -m -O z -o sample.vcf.gz sample.bcf`: uses `BCFtools` to call variants.
- Optional: a number of filters.

- `samtools tview -p 17:7579600 sample.bam Homo_sapiens_assembly19.fasta | more`: visually inspects the quality of alignments at one particular position (7579643).

Week Four

Tools for transcriptomics: genes are encoded within the genome - occupying a specific location - started with an initiator, with a very specific organization. Formed of informative blocks (exons) interspersed with uninformative blocks (introns). Gene expression is complex: starting with transcription - starting with pre-RNA. This molecule is highly unstable, and is modified to stabilize it - such as capping, clipping, addition and most importantly splicing - to get mRNA which is later translated into a protein. This is gene expression. mRNAs are also called 'transcripts'. The process by which genes can express different combinations of exons is called 'alternative splicing', where different exons are skipped. Therefore, there are a large variety of transcripts - 90% of human genes are alternatively spliced. Both in the nucleos and the cytoplasm are many genes being expressed. What genes are expressed in the sample? What transcripts are expressed? (Assembly) How many copies of genes do we have - what are their expression levels? (Quantification) How do these expression levels and splicing patterns differ? (Differential analysis)

A transcriptomic (RNA-seq) analysis surveys the state of the transcriptome within a particular cell: look at map of short reads (alignment), assemble and then conduct differential analysis at the expression or slicing level. i.) RNA molecules, ii.) read alignments, iii.) read coverage levels, iv.) splice junctions, v) graph representation (overlap, connectivity, exon graph - and then traverse it to obtain all possible spliced variants). If a read falls entirely inside an exon, it'll be aligned as a contiguous fragment - if it spans the boundary, it has to be spliced. This gives us clues, such as roughly where the exons are located. How do we identify the most likely spliced variants in the graph? The entire workflow and basic command line tools at every stage:

- Map reads to genome: `Tophat*`, `STAR`, `Hisat`.
- Assemble reads into transcripts: `Cufflink*`, `CLASS`, `iReckon`.
- Reconcile transcripts across multiple samples: `CuffMerge`
- Quantify isoform expression and compare across samples: `e.g. Cuffdiff*`, `Ballgown`.

`Tophat`: used to align reads against a reference genome allowing for spliced alignments (such as with fastq files). Example: `tophat >& tophat.log` stores it into a log file for us to view: `tophat [options] <bowtieindex> <reads>` where options include things like version, anchors, threads, etc. Most options have already been calibrated for the best performance. One thing we can do to simplify the execution of this is to make a Bash shell script. Give execute permission to your script:

```
chmod +x /path/to/yourscript.sh
```

And to run your script:

```
/path/to/yourscript.sh
```

Since . refers to the current directory: if yourscript.sh is in the current directory, you can simplify this to: `./yourscript.sh`. In the tophat context: `nohup sh com.tophat >& com.tophat.log &`. We can then `grep` for the information on the summary outputs.

Once reads are aligned to the genome, the next step is to assemble them into transcripts: `Cufflinks` - which uses an overlap graph - finding a minimum path through the graph to explain all the splicing patterns. How can we use it? In general, it's good to create a new subdirectory for every individual file, and then use a batch script on the `SAM` files. Example: `nohup sh com.cufflinks >& com.cufflinks.log &`. The main outputs here are assembled transcripts in `gtf` format. How many genes and transcripts in the transcripts.gtf? `cut -f9 transcripts.gtf transcripts.gtf | cut -d ' ' -f2 | uniq | sort -u | wc -l`. `cuffmerge` is designed to reconcile the gene structure across the samples and to combine and compare with the reference annotation (`cuffmerge [Options] <assembly_GTF_list.txt>`). You can combine `cuffdiff` and `cuffmerge`. From cuffdiff we can tell how many genes are (significantly) differentially expressed and then perform the same analysis on the isoform levels and so-on.

We then need to use the integrated genomics viewer (IGV) to curate these results. We need the tophat files from alignment and the transcripts assembled with cufflinks - sort and index for easy access, then load into IGV. We can, for example, extract results from only chromosome 9, for example: `cat transcripts.gft | awk '{ if (`$1 =='' chr9'')print\_;$`}' > Ctrl1.gtf`. In order to index, the files must be sorted. `igvtools` can take care of both these issues: `igvtools sort file1.gtf file1.sorted.gtf` and `igvtools index file1.sorted.gtf`. However, alignment files are already sorted (then use `samtools`). When we have a set of .gtf files which represent our annotations, and our alignment files (.bam) files, we are ready to import them into the IGV. You can either run the IGV from the Broad institutes website, or download a copy (after registration) - all you then have to do is click the IGV icon and load HG19 and then the sorted .gtf files for each chromosome. You can then squish the alignments, change colours, etc. We can also see a detailed view of the alignments in the .bam file.

A set of notes by Charlie **Spring 2017**

Notes on ‘Bioconductor for Genomic Data Science’

Sixth Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week One

Bioconductor is written in `R` - a widely used language for data science - chosen because it’s flexible, but specific to data analysis, and is easy to embed in other languages. It is a collection of packages within a repository - with some rules and guiding principles (V3 has 936 software packages). Bioconductor has huge impact and positive mindset within the bioinformatics community - with a focus on reproducibility. The two main reasons to focus on Bioconductor are: i) productivity and ii) flexibility. To install: source script from the internet:

```
source("http://www.bioconductor.org.biocLite.R")
```

To check that all your packages are synchronized and the right version for your version of R: `biocValid()` - do not mix and match packages and versions! To install a new package: `bioLite(s) `packagename'`. Run the development version to live on the edge! The website is organized into categories; install, learn, use, develop. ‘Software’ packages are the main workhorse - they can be split into different keywords. Importantly, each package has vignettes - sometimes hundreds of packages - long descriptions about each package and how to use it. RStudio also hosts information on the packages within the ‘Home’ tab (and link to package news, etc). Bioconductor also has details on ‘courses’ and notebooks for self-learning. The support site is the main resource for help. Don’t forget to include ‘session info’ and reproducible code/output. Naturally, Stack Overflow is always there for you! Other useful websites: `rdocumentation.org` and `rseek.org`

`Basic Types`: Atomic vectors - vectors where every element is of the same type - i.e. a sequence of numbers from 1-10. All vectors have names and classes (i.e. ‘integer’). We can subset vectors with an index based on numbers or names - names do not have to be unique - e.g. `c(`a','a','b')` - but then we only get the first element for ‘a’ back. How to represent a single integer? `x=1L`. The integer limit is a machine constant: `.Machine$integer.max`. You can’t have more than 2.1bn elements in a vector, unless it’s a ‘long vector’. Convert into numerics: `as.numeric()`. Matrices in R: two dimensions: `x=matrix(1:9,ncol=3,nrow=3)` - `dim(x)`. Subset matrices using two dimensional subsetting: `x[1:2,1:2]`. Note - we get rows back as a vector, rather than a matrix . Again, indices can be numeric or not. Lists don’t need to even be composed of the same type. e.g: `x=list(a=rnorm(3), b=letters[1:5], matrix)`. We can then index lists like a vector ( and also referred to with names). `lapply`: a function to each element. Dataframes are fundamental for data analysis - e.g: `x<-data.frame(sex=c(`M','M','F'), age=c(32,34,29)`. Row names have to be unique! We can also convert types - e.g. to convert the dataframe into a matrix: as.matrix(x) turns the dataframe into a matrix or, more generally: `a(x,'matrix')` for a range of different types.

`GRanges`: a data structure/class (from package = `library(GenomicRangeS)` for storing genomic intervals in R - fast and efficient - essential. Many entities in genomics can be thought of as intervals or sets of intervals - promoters, genes, SNPs, CpG islands, reads, etc. GRanges can help us relate intervals to each other - for example - ‘which promoters contain SNPs’. The whole concept is related to ‘computing on intervals’. GRanges (as opposed to IRanges - Granges have additional bookingkeeping/strands) have a direction and also has ‘promoters’. We can simply sort them with `sort(gr)`, `seqlevels(gr)`, assign genomes, etc. `gaps()` gives us all ranges which are not covered by the GRanges. There exists a safety feature which prevents people from doing find overlaps between incomparable genomes. `DataFrames` (as opposed to dataframes) allow us to have multiple classes within one dataframe. You can use the dollar operator to access specific columns. A very useful convenience function is `subsetByOverlaps()` - which only selects ranges on the GRange which overlaps some other elements. We can also make GRranges from dataframes (from classic R dataframes).

`IRanges`: a vector which contains integer intervals - using the IRanges constructor function i.e. `ir1 <- IRanges(start=c(1,3,5), end=c(3,5,7)`, which can then be resized on the fly. Range: `dim(ir1)`. Subset them using a single bracket and index i.e. `ir1[1]`. We can also `reduce(ir1)` and `disjoin(ir)` and `resize`. We can take the `union(ir1,ir2)` of them also. `FindOverlaps` allows us to relate two sets of IRanges to each other - very powerful and memory efficient.

`seqinfo`: part of GRanges objects - contains information about length and names of chromosomes (i.e. try and get rid of some information on some of the chromosomes). `gr = GRanges (seqnames=c(`chr1',chr2'), ranges = IRanges (start=1:2, end=4:5))`. We can also only just keep standard/standalone chromosomes - `keepStandardChromosomes(gr)`. Different chromosome names across different datasets is *extremely frustrating* - depends on different sources, and even between organisims. `NewStyles=mapSeqlevels(seqlevels(gr),''NCBI'')`, then `gr=renameSeqlevels(gr, Newtyles)`.

AnnotationHub is an interface to a lot of online resources: a relatively new, scaleable way to access a lot of different data. For example: `ah=AnnotationHub()`, with a snapshot of the different data resources - then you go online and retrieve the data you want. Retrieve objects with double brackets. Currently contains institutes like NCBI, UCSC, etc, and we can list species (a lot are bacteria). One way you can filter is to subset based on homosapiens (or genomes), i.e.: `ah=subset(ah,species= ''Homo sapiens'')`. We can also filter on specific histone modifications with `query`.

Week Two

`Biostrings` package: functionality for representing and manipulating biological strings such as RNA strings, DNA strings or amino acids. Construct a single string: e.g. `dna1=DNAString(''ACGT=G'')`. Convenience functions: `width`, `reverse`, etc. `alphabetFrequency(dna1)` computes frequencies, `dinucleotideFrequency()`, etc. `consensusMatrix()` tells us how many strings have specific nucleotide at certain positions. The `BSgenome` package is used for representing full genomes in Bioconductor. The `available.genome` command tells us what genomes are available from Bioconductor. There is a wealth of opportunities here. We can apply on the entire genome using `BSapply` - such as `bsapply(param, ''GC'')`. `Biostrings` for matching substrings in substrings (however, use a dedicated short-read aligner if you have millions of short reads, such as BowTie). However, it can be very useful for matching a small set of sequences to the genome. It has a range of functions which are useful, such as `countPattern`, `vmatchPattern`, and `countPattern`, etc. Take a set if sequences/reads of a short read and max them against the full genome (`maxPredict`) - a fast and efficient way of searching the genome for small sequences. Precision Weight Matrix: a probabilistic representation of a short sequence (`matchPWM`). Pairwise alignment of millions of reads against the short sequence of a gene (`pairwiseAlignment()`). `trimLRPatterns()` is for trimming off specific patterns on the let and right of a DNA string set (i.e. for trimming off sequence adapters).

`Views` object comes out of `matchPattern` - represented as an IRange - get the ranges out by writing `ranges()`. We can also take things like `alphabetFrequency` on these `Views`. A `View` consists of a set of co-ordinates and is a subsequence or sub-part of a bigger object. It allows us to easily represent promoters or exons, and things such as GC content, and is generally very efficient. `RLE` - 'run length encoding' - ways of representing very long vectors where some sequences are the same - a form of compression. Useful for representing signal over the genome - i.e. RNA sequencing or Chp-sequencing and many other applications - how many short reads cover a particular base? The standard genomic example is a coverage vector. We can get the original vector out using `as.numeric`. We can also combine functionality with things such as `IRanges` to make everything more powerful. We can also slice these `RLEs`. We can get an `RLE list` for each chromosome and use `Views` on it. We can have RLEs on logicals, characters, etc, and we can slice, compute on them etc also. `GenomicRanges - Lists` - similar to a normal list, but each element is itself a `GRange` - behind the scenes there are some optimizations going on to make everything efficient. A single GRange in the list could be for describing the exons of a transcript. We can also think about it as one really long `GRange`. `shift` a `GRanges` list, for example shift 10 positions with `shift(gr, 10)`. We can use `findOverlaps`, but here the result is slightly different - useful to overlap a transcript with a chp-seq peak/read - or to know whether some part of the exons overlap at a given read. `GenomicFeatures`: support for transcript database objects. A useful function here: `transcriptLength` - which gives us a length of a different transcript not in terms of their pre-MRA, but in terms of the spliced RNA. Underneath it all, all of this is to order something called a SQLight database. And there's a set of functions for querying this database directly using SQL commands. `rtracklayer` is used for importing data - interfacing with a genomebrowser - take data in R and put it on the genomebrowser, and vice versa. Examples include, something like GFF format, this is all of the general feature folders, BIT files, WPL files And big and big weight files We can also just pass in specific parts of the data with specific filters (i.e. `chr22`).

Week Three

Basic data types: data consists of 3 different types - experimental (i.e. sequence reads/align sequences), metadata (e.g. phenotype), annotation (e.g. information on specific genes to give context to the experiment with external information - usually with various databases or repositories). We typically go through a pre-processing stage to transform the raw data to make different samples comparable. One of the most important concepts is that of a 'common data container' to facilitate analysis. The annotation stage can be hard, and often ambiguous. One specific type of annotation - specific genes - where is it on the genome? what function does it have? Another type is annotating a genomic interval - are there genes or regulatory elements nearby? Take the interval and go to UCSC and figure it out. How do you annotate 10,000 items programatically? Two main approaches: such as R based annotation packages or query like online resources such as UCSC or ENSEMBL. A data container called an `ExpressionSet` measures the expression of thousands of genes on hundreds of samples. An `ExpressionSet` provides a tight linking between rows of the expression matrix and the relevant columns of the feature and phenotype data - this generalizes into an `ESet` when we have multiple matrices. The core point of `ExpressionSets` is to keep relationships between these types of data together while satisfying two dimensional subsetting - you can then annotate on the array.

A more modern version of the `ExpressionSet` is the `SummarizedExperiment` - it returns a capital DataFrame - which has specific details of the experiment such as the authors of which paper the data comes from, PubMed ID, etc. We can access it with row and col and we have the GRanges we can access, and the assay function for getting the expression measures or whatever is measured in this summarized experiment. `GEOquery` package - for interfacing with NCBIGEO (gene expression omnibus). In this resource, different datasets are stored with a session number. The starting point is the GEO identifier. The `biomaRt` package is an interface to a front-end database. Users can then access these `biomaRt` databases which are internet facing, and be accessed by resources. There are a couple of extremely useful databases available through `biomaRt`. A database is called a maRt. It's usually possible to operate the same databases from their website. `biomaRt` has an especially good vignette. `S4 Classes`: usual R packages dont use this S4 system. S3 is the 'classic' system of object orientated programming. `S4`: classes and methods are separate things - useful for representing complicated data structures. In base R, you can make any object into any class. In S3, a class is just a list with an attribute. A class should have a capital letter, and should have a constructor. Note: some of the slots in S4 are not necessarily meant to be accessed by end user - therefore, use accessor functions (documented in the help page for the class). `validObject` checks whether something is a valid version of the class - typically required when you start assigning things to slots directly. A method is a function that allows you to run different sets of code based on different values of the argument.

Week Four

Getting data into Bioconductor: it depends on the file format. 'Bioinformatics: the science of creating new file formats'. Use format specific functions. E.g. for microarrays - high level functions such as Affyio to read a directory full of files and put it into a container. How you deal with things such as `bam` or `vcf` or `bcf` files depends on what you data you have. The `ShortReads` package is for reading in raw sequencing reads typically in the format of a FASTQ file. It also has some old functionality for examining base level calls. We can also read in smaller chunks of the file. Phread scale qualities are between 0-40 - the chance of a sequencing error with quality re-mapping. For handling aligned reads, turn to the R SAMtools package - a set of collections for dealing with files in SAM and BAM (faster and more convenient due to the index) format. Read them in called `scanBam` as the outer container. The BAM format supports a rich set of alignment: each read can have multiple alignments to the genome, and it can also contain unaligned reads, which can also be spliced and chunks of the read can be mapped into different parts of the genome. You can also read in pre-specified parts of the genome (with a `ScanBam Param!` - what argument sets flags of what to read in - i.e. which pieces of the genome). `QuickBamFlagSummary` gives you an overview of what the file contains. `BAMViews` takes in multiple `BAM` files at once. `oligo` is a package for pre-processing and handling Affymetrix and nimble chimp micro ray - a continuation of the `Affy` package. Most often, you want to start off by normalizing, and a very popular method for gene expression microarrays from Affymetrix is the RMA method.

The `limma` package stands for linear models for micro arrays - fits a class of models for a lot of types of genomic data - not just useful for microarrays. Useful for continuous data (as opposed to count data). The allow us to find genes which are differentially expressed. Empirical Bayes techniques allow us to gain a lot of power. `lmfit` fits a linear model to all of the genes separately. `minfi` handles data from DNA methylation microarrays (a chemical modification which only occurs in humans in a CBG context). When reading in, we need to decompress the files first and read them in using a convenience files (to an `RGChannelSet`). We can use `getIslandStatus` to tell whether or not it's a CBG inside an island or inside an area that's close to an island. These areas are called CBG shores, a little bit further away, there's CBG shelves. And if you're really far away from a CBG island, you are an open sea CBG.

A set of notes by Charlie **Spring 2017**

Notes on 'Statistics for Genomic Data Science'

Seventh Module of the Coursera Genomic Data Science Specialization from John Hopkins University

Week Zero: Introduction

Statistics is: 'the science of learning generalizable knowledge from data'. Study design, visualization, exploration, preprocessing/normalizing data, inference and then communicating all of these things. When finding statistical software, be sure it's trustworthy! Packages on CRAN have only passed very minimal checks! Bioconductor adds some exclusive checks on the software. Bioconductor forums and StackOverflow are useful community resources for asking questions. Data are values of qualitative of quantitative variables belonging to a set of items e.g. - created from a sequencing machine - one slice of a sequence at a time. The sequencing machine then chemically attaches A,C,G and T of a different colour one slice of a sequence at a time. Then associate a likelihood with each colour being present at each point of the image, and summarize them further into counts or reads. Central dogma of statistics: **Large population of people, and measuring all of them is expensive, so use probability to sample from this big population and make measurements of them - use inference to say something about them**. Don't forget: hats over characters represent estimates! Data points are represented by letters, subscripts for different data points e.g. H_1 represents the height of person 1.

Reproducibility is defined informally as the ability to recompute data analytic results conditional on an observed data set and knowledge of the statistical pipeline used to calculate them. Replicability of a study is the chance that a new experiment targeting the same scientific question will produce a consistent result.

Week One

Reproducibility should be front and centre of all research! It's so critical that all the data and code are available. How do you achieve reproducibility? Data sharing plan: raw data (no preprocessing), tidy data (one variable per column, one observation per column, shareable), codebook (information on variables in tidy data) and recipe used to go between these three things. In general, `R markdown - (.Rmd)` and `IPython Notebooks` are indispensible, especially for embedding 'code chunks'. To compile `.rmd` files, use 'knit' in RStudio, with a frontmatter including title, author, and output format, etc. # primary header, ## secondary, etc. * bulleted lists, 1.,2. numbered lists, just like regular `.md` files:

```
'''{ r chunk1 options}
insert code chunk here
'''
```

where some options are to set the cache, echo, etc. You can insert inline code with something like `r Sys.Date()` which will compile the code into the package directly in-line (as opposed to chunks). The processed data from genomic experiments can be stored in three tidy datasets:

1. The phenotype data: Who are the cases and controls? Information on batches, or other technical or biological information used for modeling.
2. The genomics dataset: tall and skinny - few samples, but a lot of features/genes (e.g. SNPs).
3. The features dataset - a bit about the features which have been measured - e.g. what genome did they come from? What biological pathways do they belong to?

As a check, you want to make sure there are the same number of rows in the phenotype data as there are columns in the genomics data. There should be the same number of rows in the features data as there are in the genomics data. There should always be 'indicators' for linking. In general, there are three major sources of variation in genomic measurement: a.) phenotypic variability (between cancers and normals) b.) measurement error (random or biased - i.e. batch effects) c.) natural biological variation (any two individuals will have variation due to the fact that theyre different people).

Technical Replicates: Process the sample two separate times on the same biological sample.
Biological Replicates: Different samples, prepared the same way - to measure natural biological variation.

For GWAS - measure lots and lots of people (N between 10,000 to 1,000,000). Sequencing technology does not eliminate biological variability. A low number of observations is bad, because it means studies have low power (higher better!). **What is power? Probability of discovering a real signal if it is there**. Based on: signal size, variability of measurements and N. Typically set to 80%. However, all power calculations require a guess about variability, signal size, etc. We can calculate it with R code: `power.t.test(n=,delta=,sd=)` or `power.t.test(n=,delta=,power=)`. Or, if we know it will go in one direction: `power.t.test(n=,delta=,sd=, alternative='one.sided')`.

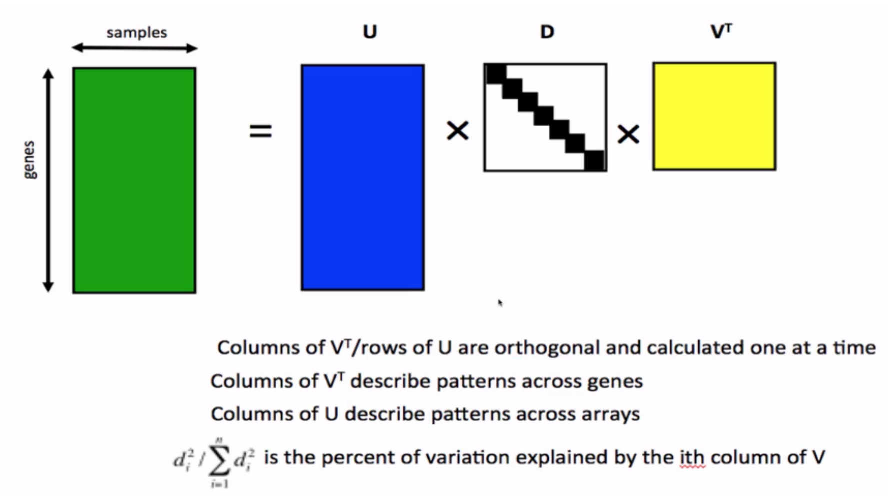
A confounder is another variable which mediates the relationship between two variables. An example is batch effects. With randomization, confounding does not differ among treatments - 'block' them to eliminate bias. Balance the experiment: equal number of treatment and controls (negative and positive), and have different methods of replication. The first thing to do in a study: exploratory analysis - which can also suggest modeling strategies (don't use pie charts!). Plots also help to uncover non-linear patterns. Rediculograms: meaningless albeit visually impressive image of a network. Log-scales can also uncover a large amount of hidden information by making the data more visible. Consider using t-tests of equal or unequal variance. In R we can use commands like `table()` and `summary()` to undertake some exploratory data analysis (don't forget to check for missing values 'useNA'). `dim()` tells us the dimensions of the data - check that they match and then begin to plot. The second thing to explore is done visually: if most of the missing values are near to zero, take log transformations. Boxplots for single values, or boxplots per matrix (gives multiple plots). Density plots allow you to write multiple histograms on top of each other. The third thing to do is: check for consistency (potentially using external data), potentially using ensemble IDs and chromozone information (chromozone labels). Check that the chromozone data matches with the expression data. Filter to just chromozone Y samples: convert dataframe and apply filtering command - to get a smaller dataset which consists of just genes on y-chromozone. Check, for example, that the males have more y-chromozone genes than for females: this is an independent check of consistency! `jitter` adds random noise so that the points dont land all on top of each of other. Heatmaps are a useful multivariate plot, and the `heatmap.2` function adds color key. If you have highly skewed count data, taking logs is essential. Add a value of one to the count to make the log transform defined: this wont affect the big counts, but will make the log transformation defined for the rest of them. A `log2` transform is a different base - allows comparison of two values - difference is equivalent to the log of the full change: an increase of 1 is equal to a doubling of the size. After we filter on the basis of the mean, dimensions should be lower after we remove genes with lower values, and transform plots should be easier to see.

Clustering: can we identify points close to each other and then cluster them together into groups somehow? Two common measures of distance between observations: (i.) Euclidean distance: $\sqrt{(X_1+X_2)^2+(Y_1-Y_2)^2}$ and (ii.) Manhatten distance: $|X_1-X_2|+|Y_1-Y_2|$. How can we use this to cluster points? Start with nearest points: merge them together, as per hierarchical clustering. K-means clustering says: what are the centers of the clusters? Should we start off by guessing centres, then assign all points to the closest centers. If you guess repeatedly, you can iteratively identify cluster centers and then re-identify centers repeatedly. This is a little different to hierarchical clustering, as you have to define the number of clusters that you want. Be careful! The scaling matters a lot, as do outliers and starting values and the number of clusters (non-trivial): better to visualize the data. These clustering methods are widely over-utilized and over-interpreted. `matplot(t(kmeans1$centers),col=1:3,type='l',lwd=3)`.

Week Two

We now move on to preprocessing. With genomic data, you often make a large number of measurements. One important tool is to be able to reduce the dimension. One way to do this is to take the average of the rows or the columns. There are two related problems in general: You have multivariate data (X): find a new set of

Dimension Reduction


samples
genes
U
D
V^T
=
×
×
Columns of V^T/rows of U are orthogonal and calculated one at a time
Columns of V^T describe patterns across genes
Columns of U describe patterns across arrays
$d_i^2 / \sum_{i=1}^{n} d_i^2$ is the percent of variation explained by the ith column of V

multivariate variables that are uncorrelated and explain as much of the variability across rows as possible, and find the best matrix created with fewer variables that explain the original data. This is essentially a singular value decomposition (a factorization of a real or complex matrix).

An example of this is Novembre et al (2013) from the first module. There are also many other decompositions which people use: multidimensional scaling, independent components analysis and non-negative matrix factorization. Practically, we should 'preprocess' by subtracting out all the row-means which are less than a hundred (`edata=edata[rowMeans(edata) > 100,]`), and take logs so the scales are easier to work with: `edata=log2(edata+1)`. We need to 'centre' the data: to remove the (row or column) means otherwise the first singular value or vector will always be the mean level. The SVD matrix has three parts to it: **u**, **v** and **d**: `svd1=svd(edatacentered)`. To plot the percent of variance explained, you need to calculate each singular value squared divided by the sum of singular values squared. Principle components are not quite the same as SVD: subtract column means rather than row means. Outliers can really drive these decompositions - be careful to pick centering and scaling so that all of the different measurements for all the different features are on a common scale.

A preprocessing step might involve adding the reads up, or looking for patterns which arise due to GC content (which might need normalizing). Quantile normalization: order the values (within columns or rows), then average across rows and substitute value with average. Then: re-order averaged values in original order - this forces the distribution to be the same as each other - only useful when you see big bulk differences between these distributions. Be careful! Distributions shouldn't necessarily be the same. Preprocessing: the step where you take the raw data and make a set of data you can do modeling on. Normalizing: make samples have common distribution across samples. You would likely want to do this when the distributions are driven by some technical considerations (i.e. biases?): One example is `normalize.quantiles()` in R - does gene to gene variability still remain after normalization - are they still seperated by study? Both preprocessing and normalization are highly platform and problem dependent. Note!:

'First, researchers starting out in genomics must keep in mind that interesting outliers - that is, results that deviate significantly from the sample - will inevitably contain a plethora of experimental or analytical artefact.'

The linear model: a best line related to two variables - take some genomic measurement (e.g. technical artifact or phenotype and relate it to some outcome). You're trying to minimize the distance between Y-b_0-b_1X_1. Example: average height of children and average height of parents - explain more variability than all genomic methods . What's e in:

$$C = b_0 + b_1 P + e \tag{7.1}$$

random noise - everything that we didnt measure - sampling variability or bias or variation which we didn't measure. Fit by minimizing distance between Children (C) and line that we care about: $\sum (C - b_0 - b1P)^2$.

Table 7.1: Interpreting Coefficients

Quantity	Log Odds	Odds
Definition	log(p/(1-p))	p/(1-p)
In Logistic Regression:	b	exp(b)
No Effect'	0	1

We can always plot a line, but the line may not always make sense. In genomics, it's common to have a both continuous and categorical covariates e.g. X={0,1,2} - the interpretation is that the difference between 0 and 1 is the same between 1 and 2. Another way is to fit different means: 1's and 0's, where the 0's are subsumed into the constant. Another way is to expand to have 2 dummies equal to 1s: to fit mean levels to different values. The most important thing is to be aware of the interpretation - how many levels do you want to fit? When we have binary or categorical variables, be aware that for observations when the value is equal to zero, this gets subsumed into the constant (e.g. b_0+b_2). Interaction models allow multiple means for each of the different values - these can get tricky when dealing with multiple covariates - what exactly does each β mean? In R: `lm2 = lm(edata[1,]-pdata$gender)` - where under the hood it takes the gender variable into a 1s and 0s dummy variable and edata[1,] is our dependent - this expands naturally to categorical variables with more categories. An interactive term: `lm3=lm(edata[1,]-pdata$age*pdata$gender)`. You can easily overlay these points on a graph - and this can really help detect outliers. Another way we can diagnose is through histograms of the residuals. A unique thing in genomics is that you typically want to run multiple regressions all at once, and this will give us hundreds, thousands or millions of model fits, for which there can be structure in the estimates, structure in the noise, residuals, etc. The key is that we need to think of linear models as one tool which can be applied many times across many different samples. In R, two packages that we will need the most are `lima` and `edge` (which creates an *edge* study object which you can just fit models to directly from passing in a dataset. We can fit on a matrix: `fit=lm.fit(mod,t(edata))`. `lmfit` is much faster (vectorized?): but you'll get basically the same thing when you're running multiple regressions. We can then look at histograms of coefficients, etc.

Batch effects and con-founders: the biggest confounder in genomics is batch effects: how do you adjust for them? Batch: the time at which samples were processed: Control like this: $Y = b_0 + b_1 P + b_2 b + e$ - which is straight ahead to estimate. There are other ways (i.e. empirical Bayes - Johnson and Li), surrogate analysis, etc. To do this in `R` is as simple as: `fit=model.matrix(-as.factor(cancer)+as.factor(batch),data=pheno)`. Some R functions will return you a dataset *cleaned* for batch effects (SVA?) which you can then regress normally.

Week Three

Statistical significance and p-values: the most commonly used statistic in the world. In genomics, corrections for multiple comparisons are essential. A common occurrence is that the outcome is a binary rather than a continuous variable:

$$C_i = b_0 + b_1 + G_i + e_i \tag{7.2}$$

where C and G can take 0 or 1. We can instead formulate it as a probability $pr(C_i = 1) = b + 0 + b_1 + G_i$ - however, this probability is between zero and one. One way is to model the dependent as log(p) - a little better - values between negative infinity and zero - more like a continuous variable. Or: go even further - model the *log odds*: $log(\frac{p}{(1-p)} = b_0 + b_1 + G_i$ but we need to interpret b_i as a change in the log-odds.

A common type of data is count data: a number of reads which overlap a region or varient. There are a number of choices about how you can count each gene with this type of data. The most common type of distribution for modeling this type of data is the Poisson distribution, where it's important to remember that here the mean and the variance are the same: low mean-low variance, etc. : $f(k;\lambda) = \frac{e^{-lambda}\lambda^k}{k!}$. The regression model becomes more complicated here (a type of generalized linear model):

$$g(E[f(c_{i,j})|y_j]) = b_{i,0} + \gamma_i log(q_j) + b_{i,1} y_j \tag{7.3}$$

where the left hand side is the link function conditional (typically logged) on the group indicator as some function of the data (where we are testing b). We can go one step further and use a type of local/smoothed regression to estimate the relationship between mean and variance, and then plug this into the negative binomial model using a pair of parameters for mean and variance:

$$K_{i,j}\tilde{N}B(\mu_{i,j}, \alpha_i) \tag{7.4}$$

$$\mu_{i,j} = s_j q_{i,j} \tag{7.5}$$

$$log_2(q_{i,j}) = x_j \beta_i \tag{7.6}$$

where $K_{i,j}$ are counts of reads for gene i, sample j, $\mu_{i,j}$ is the fitted mean, α_i is gene-specific dispersion, s_j is sample-specific size factor, $q_{i,j}$ is the parametre proportional to the expected true concentration of fragments, x_j is the j-th row of the design matrix X and β_i is the log fold changes for gene i for each column of X. A lot of the work on these topics in R comes in the form of the `glm` command. Don't forget: remember to calculate the PCs first to use for adjustment (a typical confounder is the population structure). The variability is a quality that matters a lot. One way to measure it is to take the standard deviation or variance:

$$S_X^2 = \frac{1}{M-1}\sum_{i=1}^{M}(X_i - \bar{X})^2 \tag{7.7}$$

$$CI = (\bar{X} - c\frac{S_x}{\sqrt{n}}, \bar{X}, c\frac{S_x}{\sqrt{n}} \tag{7.8}$$

Note - as sample size gets bigger, variability gets smaller. The CI is defined by the probability that the real parameter covers the interval: $Pr(\theta \in CI) = f(c)$: you're looking for confidence intervals to give you some idea of some expected range of values that youre somewhat confident will cover the range of values if you repeat the study over and over again. It acts to quantify the uncertainty. The null hypothesis is that the relationship is zero. The alternative hypothesis is that there is a relationship. For example: $y = b_0 + b_1X + e$, and the null is $H_0 : b_1 = 0$ and the alternative of $H_1 : b_1 \neq 0$. In general, it's not possible to accept the null: you're going to reject it 'in favor' of the alternative. Ideally your statistic should be set up to be 'monotone': bigger/smaller is 'less null'. The null depends on the specification, and it must make intuitive sense. Example: is the average height of a child equal to 70? $\bar{X} - 70$ quantifies how far we are. We want to put it on an interpretable scale: $\frac{\bar{X}-70}{\frac{s_x}{\sqrt{n}}}$. This tells us something about how many 'variance units' we see the difference to be. This allows us to make comparisons to standard distributions and calculate probabilities of observing statistics this extreme. Imagine we have two samples:

$$S_X^2 = \frac{1}{M-1}\sum_{i}^{M}(X_i - \bar{X})^2 \tag{7.9}$$

$$S_Y^2 = \frac{1}{M-1}\sum_{i}^{M}(Y_i - \bar{Y})^2 \tag{7.10}$$

To compare whether these two groups are the same, use the t-statistic:

$$t = \frac{\bar{Y} - \bar{X}}{\sqrt{(}\frac{S_y^2}{N} + \frac{S_X^2}{M}} \tag{7.11}$$

When you divide by the estimate of the variability, the results may be tiny, and one way to get around this is to use an empirical Bayes procedure - the simplest one of this is to add a constant to add to the denominator: $\frac{\hat{b}}{s.e.(\hat{b})+c}$. We might want to compare the fit of two or more models: does one fit better than one which plots an overall average? One way we can consider this is through the residuals - the data point subtracting the model fit: $y - b_0 - b_1B$ vs $y - b_0 - b_2B$, for example. One way we can test this is through the commonly used F-statistic:

$$F = \frac{n - p_1}{p_1 - p_0}\frac{RSS_0 - RSS_1}{RSS_1} \tag{7.12}$$

$$RSS_k = \sum_i (y_i - \hat{y}_i^k)^2 \tag{7.13}$$

In R, `limma` is indispensable for calculating statistics such as this (and other empirical Bayes calculations), but again, `edge` can be used more simply if the user has no familiarity with matrix notation, and with `edge` you can use an adjust variable. **Permutation:** for example you might calculate a statistic to compare the difference in expression level between responders and non-responders standardized by some measure of their variability. One thing we can do is randomly scramble the labels. The idea is that you do this permutation and then re-calculate this statistic for the original gene: how extreme is the original (unscrambled) statistic? This concept is *extremely* common in genomics - not just for simple comparisons, but also network comparison, etc. By switching the labels we can still assume that the data come from the exact same distribution: by permuting them, we are making the assumption that the labels don't matter. When doing this computationally, don't forget to set the `seed` vale.

The p-value is so popular that if it were cited every time it were used, it would have 3m citations - and it's consistently mis-interpreted. **It is the probability of observing a statistic that extreme if the null hypothesis is true**. It is not: (i) the probability that the null or (ii) alternative are true, or (iii) a measure of statistical evidence. The p-value can be thought of as the number of permutation statistics that we observe to be larger than the number of permutations that we calculated:

$$P - value = \frac{\#|S^{permutations}| \geq |S^{obs}|}{\# of permutations} \tag{7.14}$$

A particularly important feature of this is the fact that the permutations will be uniformly distributed. The p-values almost always go to zero with sample size. The cutoff of 0.05 is a made up number: these should be reported in conjunction with estimates and variances. The concept of multiple hypothesis testing (MHT) is best emphasized by the 'jelly bean' phenomenon. For example: if you measure 10,000 genes, 500 will be false positive. Family wise error rate: Pr(# False Positives$\geq$1). False discovery rate: E[$\frac{\#FalsePositives}{\#ofDiscoveries}$] - this helps to quantify the noise level in the number of discoveries made. The way to do these corrections:

- **Bonferroni Correction** for the family wise error rate: p-values less than α/m are significant.
- **Benjamin-Hochberg Correction** for the false discovery rate: order the p-values from smallest to largest - p_1,...,p_m: if $p_i < \alpha \times \frac{i}{m}$ then it is significant: this is a linearly growing function in the number of tests performed.

Just using the raw-uncorrected p-values is only done when really analyzing one variable of interest. Type 1 errors are different to the family wise error rate and the false discovery rate. These all rely on the p-values being 'correct': but many things can go wrong - such as batch effects, bad specification, etc.

Week Four

When you get a list of SNPs or genes that you find interesting, how do you then communicate them? What do these genes or SNPs do? Do some genes have something in common with a list of differentially expressed genes? Take statistics for every gene calculated, and order from largest to smallest (or largest to smallest p-value). Then take some gene set that you care about and label them - calculate a running statistic that goes up every time you have a gene in the gene set - are they clustered in one region? This is gene set enrichment - the max deviation from zero - is it more than we would expect to see by chance? Again - permute this - recalculate the statistics and re-order them to get a reference set and again do a false discovery rate type test. This can be hard to interpret - especially if the categories are broad or vague - it is too easy to 'tell stories' if you aren't careful, and be careful to correct for the multiple testing problem. Enrichment analysis can be useful for summarizing a number of things though - not just gene sets - for example - two sets of results for SNPs labeled with one of two different analyses. For example: count eQTLs which are in a particular number of GWAS SNPs, and take another randomly selected bunch of hits to calculate eQTLs - again - a type of permutation scheme: are they independent of each other? Use Fishers exact test or chi-square test (possibly within minor allele frequency or GC content) - take into account common properties. Getting the null right in this case is extremely hard.

The process for RNA-seq: Recall that the central dogma - information passed from DNA to RNA to protein: imagine we have a fragmented RNA molecule using its poly-A tail. You can then reverse transcribe it into complimentary DNA and you can sequence that and turn it into the RNA.

1. Align the reads to the genome or transciptome, assuming it's known. You can align reads to the genome to account for splicing using software such as `HiSat`, `Rail`, `Star`, `Tophat2`, etc.

2. Then you need to count the values that correspond to a particular gene or transcript - use `HTSeq, featureCounts, kallisto (no alignment - quicker), derfinder`. You can also assemble and quantify: `StringTie, Cufflinks (where the reference genome is known), Trinity, RSEM`.

3. Then you need to normalize the data - `EDAseq, cqn, DESeq/edger, Ballgown, derfinder`. To remove batch effects: use `sva or RUVseq`.

4. Then you need to perform statistical tests/modelling - `DESeq2/edgeR, Ballgown, derfinder`.

5. Finally - gene set enrichment analysis to find gene sets or categories which are enriched: `goseq`, `SeqGSEA`, etc.

The process for CHip-Seq: useful for measuring the way in which proteins (regulate) interact with DNA - through binding of proteins to DNA (transcription factors which regulate the expression of particular genes) - how much are particular proteins attached to DNA - the first step is to crosslink proteins to DNA - then fragment the DNA. If you have an anti-body for the protein to which you have attached to the DNA, you can do an anti-body pulldown - enriches for a particular subset of the DNA associated with the proteins of interest. Sequence just those fragments - how do you analyze the DNA that just comes down from this pulldown experiment.

1. Align: you don't necessarily need to worry about doing anything other than a straight-ahead alignment to the genome, such as with `Bowtie2` or `BWA`.

2. Detect peaks: we've enriched particular sequences and big piles of reads corresponding to sequences : `CisGenome, MACS, PICS`.

3. Count the amount of reads which cover a particular peak - how quantitative is the CHip-Seq technology? It's useful to know how much of the reads fall into each of the peaks with `CisGenome,MACS, diffbind`.

4. A newer step is normalization – especially cross-sample normalization because of the increasing number of replicates over time. Use `diffbind, MAnorm` in Bioconductor.

5. Then you need to do statistical tests to identify differences between the cases you care about - binomial test or others: `CisGenome and MACS` focus on the two-sample case and cover the whole process, `diffbind` cover the whole spectrum.

6. What is the sequence motifs underlying this particular transcription factor? Annotate the transcription factor binding sites - `CisGenome`, `meme-suite`, `BioC Annotation Workflow`

The Process for DNA Methylation: one of the many epigenetic marks measured through NGS or microarrays. DNA methylation refers to a particular methyl group binding to CpG sites in the genome - where can we identify the places where the binding is occurring - one is bi-sulfite conversion followed by sequencing. Another way is through methylation arrays and hybridisation to a microarray to show how much methylation is happening at a specific locus.

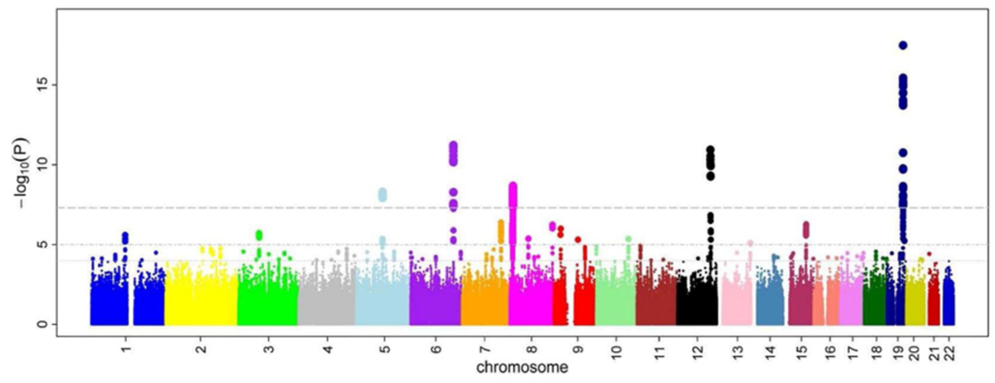
1. Normalization - detect whether there was methylation at that locus, comparing methylated dna to unmethylated dna whether that is through bi-sulfite conversion or hybridisation (`minfi` or `charm`).

2. Smoothing - find clumps of points above a particular level which shows they are highly methylated (`charm` or `bsseq`).

3. Region finding - fit a statistical model to find these regions or tissues (`charm` or `bsseq`).

4. Annotate the regions to the genome: particular categories specific to methylation - regions next to CpG islands, outside regions, etc.

Manhatten Plot Example



The Process for GWAS/WGS: One of the most common applications is WGS (previously called GWAS) - directly measuring variability in the DNA to find different types of varients - insertion or deletion in the genome. Population level inference. One way to do this is direct re-sequencing of the genome. Fragment it then sequence it - look for variations in the genome compared to the standard reference genome and identify if there is any variability. Should there be more than one reference genome? How do we define variability? How much do these variations associate with different outcomes? Once you've got the genome and your fragments, do many fragments have a C vs a G? Is this a heterozygote for a varient which doesnt necessarily feature in the reference sequence. The idea is that you can do the same sort of thing with a microarray - GWAS approach typically applied on microarrays - natural extension. You have fragmented samples and you compare them on homozygous reference alleles, etc, and identify which one it is to genotype samples.

1. Varient identification: compare the levels which you've observed for the different genotypes then make a call for each varient - (SNP-chip - `crlmm`) (sequencing - a little more complicated - `freeBayes, GATK`).
2. Population stratification - if youre looking for association between variants, the most common confounder is population structure. Try: `EIGENSOFT, snpStats`.
3. Once you have these confounders you do a set of statistical tests - SNP by SNP or varient by varient - test for association with outcome - with a logistic regression model potentially adjusted for principle components - with a p-value for every SNP. A typical way to do this is with Manhatten plots where the highest values looks like spikes above a threshold with BF corrections. Software: `PLINK and snpStats`.
4. Then analyze specific variants and particular regions to figure out which one might cause it, such as with `PLINK, Annotating Genomic Variants`.
5. Then a whole bunch of software is required to categorize the variants - `CADD, VariantAnnotation, Annotating Genomic Variants`. Are they in a splice site, etc. Ultimately you need to do downstream experiments to identify functional variation.

eQTL: expression quantitative trait losci - a common integrative analysis - an analysis where you try to identify variation in DNA which correlate with variations in RNA - what is the abundance of different RNA modules in the same samples - a whole class of problems associated with genomic types. Integrate those data together to identify cross-regulation between measurements. Identify relationship between expression levels as well as genotype levels - SNP/marker level associated with each gene in the genome (position of SNP) and information on a particular gene (how much is it expressed? where is it?) An association between all gene expression levels and all SNPs. Important to be wary of batch effects in the data which may or may not be biasing results. cis-eQTL: where SNP position are close to coding region to the gene, and trans-eQTL where the distance is far. Like GWAS, you need to adjust for population, batch effects and sequence artifacts. The basic idea is to do linear regressions relating gene expression information related to genotype information one by one.

There are a large number of steps in these analyses: this leads us to 'researcher degrees of freedom'. These undisclosed flexibilities in data collection and analysis allow us to craft significance from anything.

The question is contingent on what your model is. The whole idea is summarized by the ‘garden of forking paths’ - by Gelman. The key take home method - have a specific hypothesis, and pre-specify the analysis plan with training/testing sets. Only analyze your data once or report all analyses. Don’t add more and more things until you find the significance you want in order to ‘run into’ extra false positives. Throughout this course we’ve focused in inference, as opposed to prediction. The central dogma of prediction: use a training set to build a prediction function (e.g. a classifier). Inference and prediction can give hugely different answers - and rightly so. Inferences may or may not be ‘predictive’.