Alice Gudem

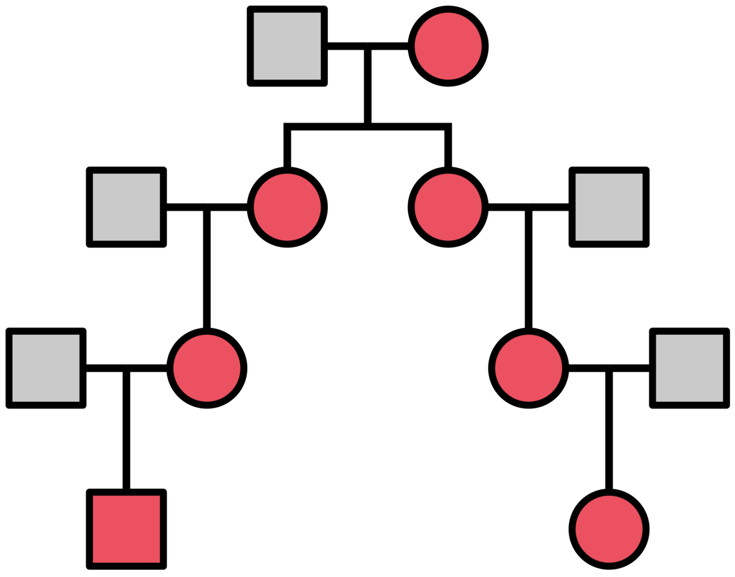
Mitodetect

Estimating common maternal ancestry in the HUNT study

Master's thesis in Computer Science
Supervisor: Prof. Magnus Hetland, Prof. Pål Sætrom
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



NTNU
Norwegian University of
Science and Technology

Alice Gudem

Mitodetect

Estimating common maternal ancestry in the HUNT study

Master's thesis in Computer Science
Supervisor: Prof. Magnus Hetland, Prof. Pål Sætrom
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Summary

Mitochondrial DNA is maternally inherited, as opposed to nuclear DNA which is inherited from both parents. This means that distantly related individuals who have almost no DNA in common, might have close to identical mtDNA if they have a direct maternal line to a common maternal ancestor.

There exist many methods that can estimate relatedness between individuals and reconstruct pedigrees, based on their nuclear DNA, but little work has been put into looking at mitochondrial DNA and to whether it is possible to estimate if two individuals are ”mitochondrially related”, i.e. that they do in fact have a direct maternal line to a common maternal ancestor.

This further raises questions like how different mitochondrial DNA is between individuals in a population, if you also take their degree of relatedness into account.

To address this question, I developed the function `mitomatch` which calculates mitochondrial distances between pairs of individuals. These distances were further compared with the individuals’ degree of relatedness estimated by DRUID, and this comparison showed that for individuals in the HUNT study, the distributions of mitochondrial distances of 6th degree relatives and above are indistinguishable from the distributions of mitochondrial distances between unrelated individuals.

Furthermore, mitochondrial distances between father/child-pairs – with the mother and father being 7th degree relatives or more distantly related – were used to estimate p-values that indicate the chance that two individuals have similar mtDNA simply by coincidence.

These p-values were used in a hypothesis test to decide whether individuals are mitochondrially related.

`mitomatch` and the hypothesis test is available as the software Mitodetect.

Sammendrag

Mitokondrie-DNA blir arvet fra mor til barn, i motsetning til vanlig, autosomt DNA som blir arvet fra begge foreldre. Dette betyr at personer som er fjernt i slekt vil ha veldig lite vanlig DNA til felles, men de kan ha nesten identisk mitokondrie-DNA hvis de har en direkte moderlinje til en felles stammor.

Det finnes mange metoder som kan estimere grad av slektskap mellom individer og rekonstruere familietrær basert på autosomt DNA, men det finnes ikke mye arbeid som fokuserer på mitokondrie-DNA, og som ser på om det er mulig å estimere om to personer er i ”mitokondrie-slekt”, altså at de faktisk har en direkte moderlinje til en felles stammor.

Hvis man i tillegg sammenligner mitokondrie-DNA mellom personer i en befolkning, hvordan vil disse forskjellene være hvis man også tar hensyn til personenes grad av slektskap?

For å adressere dette spørsmålet, har jeg utviklet funksjonen `mitomatch`, som beregner avstander på mitokondrie-DNA mellom par, og disse avstandene har blitt sammenlignet med parenes grad av slektskap, estimert av programmet DRUID. Denne sammenligningen viste at for personer som har vært med i HUNT-studien, så er distribusjonene av 6. grads slektninger lik distribusjonen av distanser mellom personer som ikke er i slekt.

Videre så har mitokondrie-avstander blitt beregnet på far/barn-par – der mor og far er 7. grads slektninger eller fjernere i slekt – og distribusjonen av disse avstandene har blitt brukt til å estimere p-verdier, som skal indikere sjansen for at to personer har likt mitokondrie-DNA ved en tilfeldighet, og ikke fordi de er i slekt.

Disse p-verdiene ble brukt i en hypotesetest for å avgjøre om personer er i ”mitokondrie-slekt”.

Denne hypotesetesten, sammen med funksjonen `mitomatch` er tilgjengelig som programvaren Mitodetect.

Preface

This report is a master thesis within the field of Computer science, written during the Spring of 2019 at the Norwegian University of Science and Technology. And it is a continuation of my project thesis, written fall 2018.

I would like to thank my supervisors, Prof. Magnus Hetland and Prof. Pål Sætrom for providing exceptional guidance throughout the two semesters, as well as PhD Candidate and researcher Anne Heidi Skogholt for helping with access to the HUNT data set and Post. Doc. Laurent Thomas for providing trio data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The mitochondria are organelles located our cells, and they are often labeled ”the powerhouses of the cells” – needless to say, they are responsible for generating most of the cells’ energy. The mitochondrion has its own DNA which is inherited maternally, from our mothers – as opposed to ”normal” nuclear DNA, which is inherited from both parents. Mutations in the mitochondrial DNA is known to cause an array of diverse disorders (Schon, DiMauro, & Hirano, 2012), which further encourages research on the mitochondrial DNA.

In genetics, it is essential to know the relationship between individuals when studying their DNA. At one end, close relatives are needed in family association- and linkage-studies, and at the other end, case-control studies require non-relatives (Daly & Day, 2001).

There already exists established protocols for accounting for relatedness, when studying nuclear DNA. One might make use of reported pedigrees (e.g. family connections taken from health registers), but it is nonetheless normal to apply some sort of pedigree reconstruction method or relatedness estimation based on the individuals’ DNA, to ensure one knows the truest relationship between the individuals, and to avoid inflated false positive rates due to cryptic relationships (Voight & Pritchard, 2005).

However, there is no established protocols when it comes to studying the mitochondrial DNA of individuals data sets. Assume you have a large biobank where thousands of individuals have participated, and their mitochondrial DNA has been sampled. You want to do a statistical analysis on their mitochondrial DNA, but you need samples to be independent, i.e. unrelated. Would it be enough to only apply a relatedness estimation method on the individuals, when the relatedness estimation only takes their nuclear DNA into account? This question arises because mitochondrial DNA is inherited in its entirety from mothers, while nuclear DNA is inherited from both parents and are subject to recombination in each generation.

To illustrate this with an example, consider the two following figures: Figure 1.1 shows a pedigree spanning four generations with the colors indicating nuclear DNA inheritance. The couple A and B are the great-grandparents of C and D, who are second cousins. As can be seen, both C and D will have inherited on average 25 % of their DNA from A and B (or 12.5 % from each of them). However, as C and D are second cousins (and hence 5th degree relatives), they will only share on average about 3% of their DNA with each other.

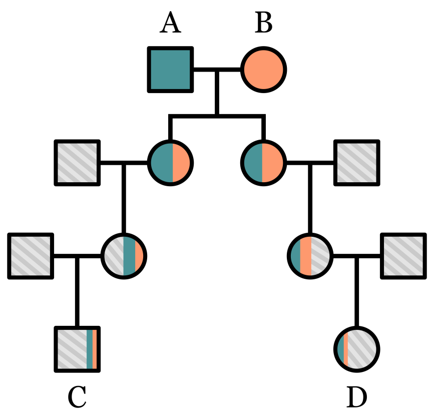


Figure 1.1: The figure depicts a pedigree spanning four generations, where squares represent males and circles represent females. A and B's DNA is indicated by their respective colors, and the pedigree shows how much of their DNA is inherited by their descendants. More specifically, their great-grandchildren have inherited about 25% of their DNA combined.

In contrast, Figure 1.2 shows the mitochondrial inheritance, which is inherited maternally. Both C and D have a direct maternal line to their great-grandmother, B, and – assuming no mutations have occurred – C and D, therefore, have the exact same mitochondrial DNA.

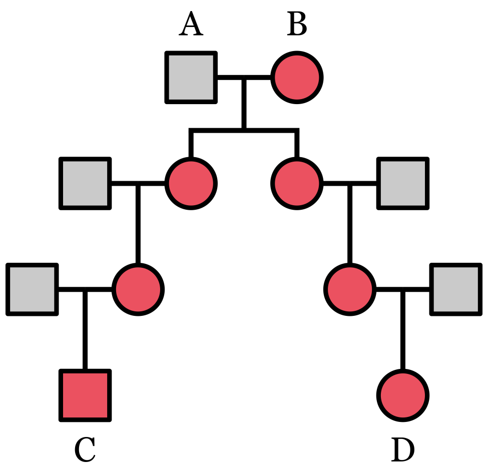


Figure 1.2: This shows the same pedigree as in Figure 1.1, but instead of showing nuclear DNA inheritance, the figure depicts mitochondrial DNA inheritance, which follows the maternal line.

A lot of effort has been put into creating large biobanks in later years (Ramstetter et al., 2018), with thousands of genotyped individuals, but the situation shown in Figure 1.2, with four generations or more being genotyped, is not necessarily common. It is more likely you only have the DNA of C and D (and perhaps their parents), and you could still perform some relatedness estimation on these individuals to find out that they are related, but would it be possible to find out whether they both have a direct maternal line – a series of mothers – to a common maternal ancestor, as indicated in Figure 1.3?

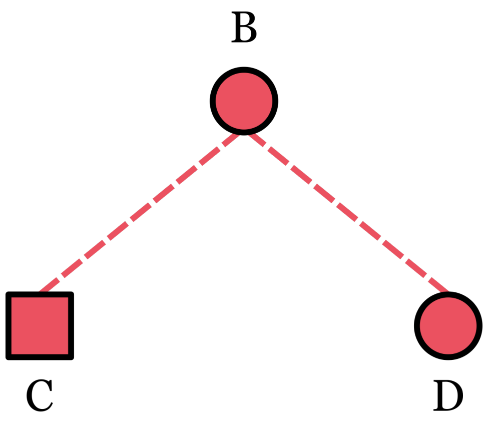


Figure 1.3: In this figure, we simply know that C and D are relatives but not how distant. In addition, they have a direct maternal line to a common female ancestor.

This leads up to my goal, which is to look into how can you find out whether two randomly selected individuals are "mitochondrially related", i. e. they have a direct maternal line to a common maternal ancestor. A part of this goal will be to create a software that will try to estimate this.

Given a population, it is already possible to infer whether some of the individuals are mitochondrially related. Namely, mother/child pairs, grandmother/grandchild pairs, full siblings and half siblings on the mother's side are examples of individuals who share mitochondrial DNA due to common maternal ancestry – and these relationships might be possible to infer with pedigree reconstruction-methods. I will therefore test some pedigree reconstruction methods, to see how they perform on the HUNT set:

> **Question 1:** Which pedigree reconstruction-methods can be used to find individuals that have a direct maternal line to a maternal ancestor for close enough relatives?

However, it is not possible to use pedigree reconstruction methods or relatedness estimation methods alone to infer maternal relationships, if there are missing links between two individuals or if they are distantly related. E.g. if you have a grandmother/grandchild relationship, you cannot know if they are on the same maternal line without knowing if the grandmother is on the mother's side. However, you can use the mitochondrial differences between pairs of individuals – if the grandmother and grandchild have identical mtDNA, this is a strong indicator for the missing link between them being the mother.

So a way to infer whether two individuals share a direct maternal line to a maternal ancestor, would be to compare their DNA. Additionally, you can take the degree of relatedness between C and D into account, because if you know that C and D are related in some way, this would naturally increase the chance that they have are mitochondrially related. Besides, the chances for them having a direct maternal line to a common maternal ancestor would be higher if they are 4th degree relatives, as opposed to 7th degree relatives. So how different are mtDNA between individuals when taking their relatedness into account? A way to answer this question would be to take a group of genotyped individuals, measure the difference between their mitochondrial DNA in some way, and compare this to their degree of relatedness:

> **Question 2:** At what degree of relatedness do the distributions of mtDNA differences look like those of unrelated individuals?

At last, we need to measure the difference between individuals' mtDNA somehow, and a way would be to calculate the distance between pairs of individuals. The Hamming distance is one opportunity – calculate the number of different variants – and a weighted distance is another opportunity. But to calculate a weighted distance, weights are needed, and to obtain weights, we can look at which variants in the mitochondrial DNA of the population varies a lot, and which do not:

> **Question 3:** What is the probability that two individuals will not share a given mitochondrial variant?

In order to answer these questions, I have had access to the HUNT biobank, a part of a large health study conducted in Nord-Trøndelag in Norway, where about 70 000 of the participants have provided DNA samples. Therefore, it is worth mentioning that the information I find will be valid for the HUNT population, but not necessarily all other populations. The HUNT study has a high number of participants, as well as a high coverage of the population in Nord-Trøndelag, which likely results in a lot of relatives among the participants. Indeed, a study of 1000 randomly selected individuals from the HUNT biobank showed that each individual was related to on average 970 other individuals in the subset, when considering up to 12th degree relatedness (Gudem, 2018). This might affect how distributions of mitochondrial differences look like when taking the degree of relatedness into account. Besides, different populations will have different mitochondrial variants that vary a lot, which again will affect the probabilities that two individuals will not share a variant.

Chapter 2 starts with an introduction to some topics in genetics and biology which might be necessary to have a grasp of when reading the rest of this paper and chapter 3 talks about probabilities, weights and ways to estimate whether two individuals are mitochondrially related. In addition, the chapter has information about the HUNT study and the software that has been used, namely DRUID (Ramstetter et al., 2018) and Sequoia (Huisman, 2017).

In chapter 5, you will find the results and discussions of the analyses that I have run on the HUNT set and the software I have made. chapter 4 describes how the analyses has been run, and how to run the software.

At last, the conclusions of my findings is found in chapter 6.

Chapter 2

Genetics

This chapter has been taken in its entirety from my project thesis (Gudem, 2018).

This chapter will contain a brief introduction to biology and genetics that is necessary to have a grasp of when reading the rest of the paper. Some parts will have a rather brief and superficial explanation, while others will go more in depth. It is also recommended to read this chapter sequentially for readers who are not familiar with concepts within these subjects.

Section 2.1 will explain relevant topics within biology and genetics, including the cell, DNA, mitochondria, gene and chromosomes to name a few. Section 2.2 will further explain concepts within heredity, including how DNA is inherited, what identity by descent means and the concepts degree of relatedness, pedigrees and cryptic relatedness.

2.1 Biology

2.1.1 Cells

A *cell* is the basic structural, functional and biological unit of all living organisms. It is enclosed within a membrane and contains several *organelles*, which is a specialized subunit that serves some special function.

Eukaryotic cells and the nucleus

A *eukaryotic cell* is a cell that has a nucleus. The *nucleus* is one of the larger organelles in a cell, and it contains the cell's genetic material, i.e. this is where most of an organism's DNA is located. The nucleus maintains the integrity of the DNA and controls the activities of the cell by regulating the gene expression. In other words, it works like the cell's control center.

A simplified eukaryotic cell is shown in Figure 2.1.

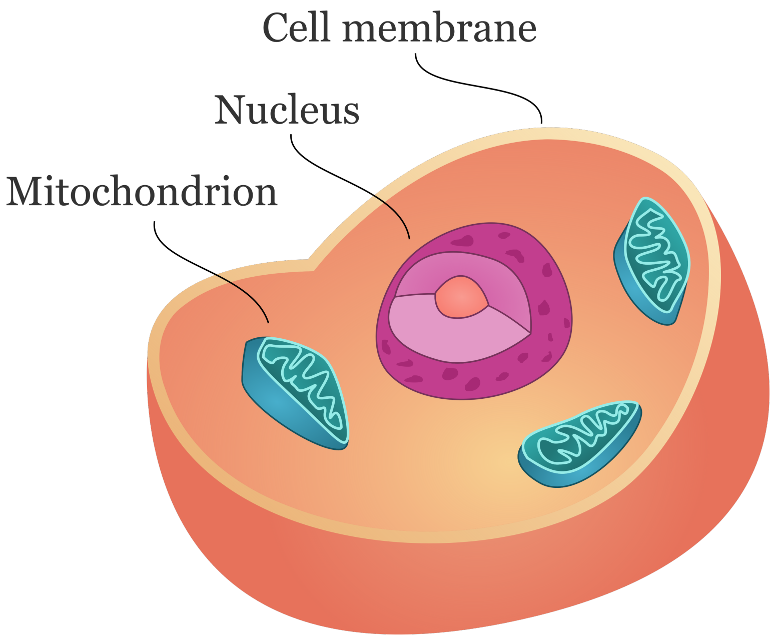


Figure 2.1: This figure shows a simplified eukaryotic cell – all the stuff that is not necessary for this report has simply been taken out, and what remains is the cell with its nucleus and its mitochondria, enclosed within the cell membrane.

Mitochondria

Another important organelle of the cell is the *mitochondrion* (plural *mitochondria*). It is often nicknamed the “Powerhouse of the Cell”, mainly because it is responsible for supplying cellular energy by converting energy from food into a form that cells can use.

2.1.2 DNA and the genome

DNA is the hereditary material in humans and nearly all other organisms. All DNA in a human is referred to as the *genome*.

The DNA is arranged in two long strands of nucleotides that form a spiral called a *double helix*. This double helix structure looks like a spiraling ladder as can be seen in Figure 2.2.

A *nucleotide* consists of a sugar molecule, a phosphate molecule and one of four chemical bases: Adenine (A), thymine (T), guanine (G) and cytosine (C). As shown in Figure 2.2, each base is paired up with another base in the two connected strands: Adenine is paired with thymine and cytosine is paired with guanine. One of the strands is, therefore, a “mirror” of the other.

The order of these chemical bases determines the information available for building and maintaining organisms, and hence DNA can be expressed as strings (e.g. “TATA” would be a sequence of nucleotides with the bases thymine, adenine, thymine and ade-

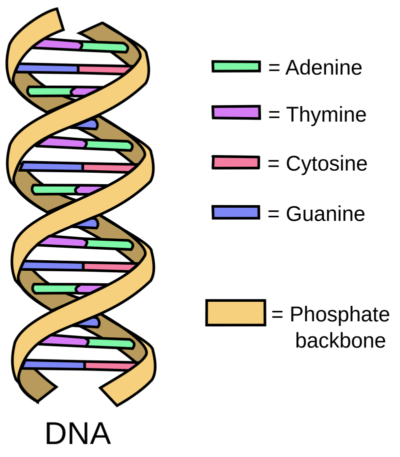


Figure 2.2: The image depicts two strands of DNA and how they are connected in a double helix with the four chemical bases. Source: user:Forluvoft / Wikimedia Commons / Public Domain

nine). A specific sequence (e.g. a gene) at a specific location in the DNA will, therefore, act as a code for some function (e.g. whether you have blue or brown eyes). Human DNA consists of about 3 billion bases, and more than 99% of these are the same in all people.

Nucleus DNA

The DNA located in the nucleus is called the *nucleus DNA* or simply *DNA*.

Mitochondrial DNA

The *Mitochondrial DNA* or *mtDNA* is a DNA found in the mitochondria. It is inherited directly from the mother, and can thus be used to trace the maternal line.

2.1.3 Loci

A locus (plural *loci*) is a fixed position on a chromosome, like the position of a gene or a genetic marker.

2.1.4 Genes

A gene is a sequence of the DNA string and is the basic physical and functional unit of hereditary. A gene can be coding or noncoding, whereas the former acts as instructions to make protein sequences and constitutes about 1% of all human DNA (it is the proteins that do the actual work of the cell). The noncoding genes perform other tasks.

2.1.5 Chromosomes

In the nucleus of the cell, the DNA molecule is packed into thread-like structures called *chromosomes*. Each chromosome is made up of DNA coiled tightly together many times around proteins called histones that support their structure.

Each person normally has 23 pairs of chromosomes, where 22 of these pairs are called *autosomes* and look the same in both males and females. The 23rd pair, the *sex chromosomes* (or *allosome*), differ between males and females. Females have two copies of the *X-chromosome* while males have one X- and one *Y-chromosome*. The chromosomes are numbered after size in decreasing order.

Chromosome pair 1-22 are called *homologeous chromosomes* or *homologs*, meaning that they have the same size and code for the same genes – e.g. if one homolog contains a specific gene at a specific locus, the other homolog will have the same gene at the same locus, but they can have different alleles. Besides, a woman's X-chromosomes are homologs, but a man's Y and X-chromosomes are not.

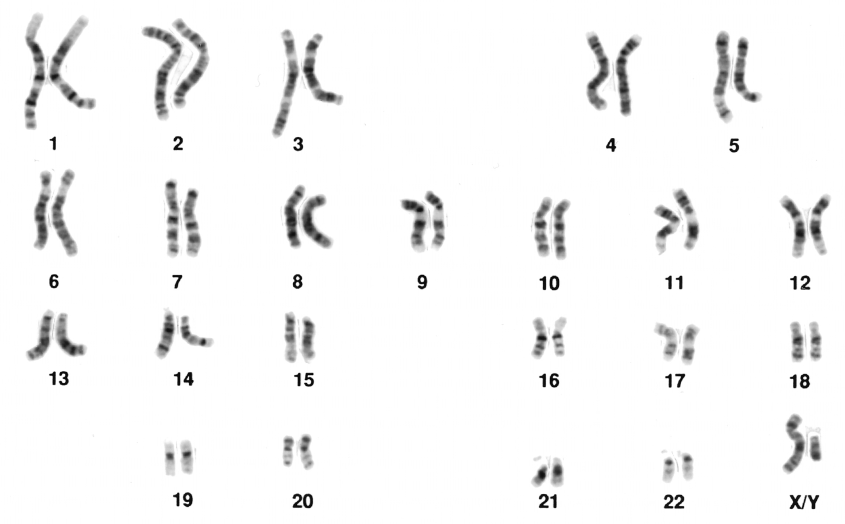


Figure 2.3: The chromosomes in a male individual. It shows the 22 chromosomes and a pair of sex chromosomes, numbered according to decreasing size. Source: National Cancer Institute / Wikimedia Commons / Public Domain .

2.1.6 Alleles, genotypes and phenotypes

An *allele* is the form a gene can take – where the gene is the variable, an allele is its value encoded in the DNA string.

A *genotype* is the alleles located at the same locus in a pair of homologs. More specifically, a genotype can be on the form Aa, where A is the allele on one of the homologs and a is on the other. A could be the allele for brown eye color, and a the allele for blue eye color. The actual manifested physical trait is called the *phenotype*.

When individuals are genotyped (their DNA is sampled), the genotypes are often unphased, meaning that one doesn't know which homolog has what allele. It is possible to infer the phasing of genotypes by looking at whole populations.

2.1.7 Haplotypes and haplogroups

A *haplotype* is a group of alleles that have been inherited together from a single parent, and hence will be located on one of the homologs. A *haplogroup* is a set of different haplotypes at different chromosome regions that are closely linked and are usually inherited together. Two examples of haplogroups are the Y-chromosome haplogroup and the mitochondrial haplogroup.

2.1.8 Single-nucleotide Polymorphisms

A *Single-nucleotide polymorphism*, often shortened SNP is a variation in a specific position on the genome where each variation occurs above some threshold for a population (e.g. more than 1%). For example, an SNP may replace the nucleotide cytosine (C) with thymine (T) on a certain stretch of DNA in about 20% of the population.

2.2 Heredity

As stated in subsection 2.1.5, each person has 23 pairs of chromosomes. For each of these pairs, one chromosome is inherited from the mother and one is inherited from the father.

This essentially means that each person has inherited roughly half their DNA from their mum and the other half from their dad.

2.2.1 Meiosis

So how is DNA from a parent duplicated to their offspring? It is done during *meiosis*, the process of creating *gametic cells* (gametic cells are either egg cells or sperm cells). As opposed to normal copying of DNA, which is called *mitosis*, where you want to copy the DNA exactly, gametic cells contain chromosomes that are a mixture of the two homologous chromosomes in the original DNA, for each of the 22 pairs of chromosomes. As can be seen in Figure 2.4, four different chromosomes have been created based on one pair of chromosomes, and everyone is a little bit different. These four chromosomes could be represented as four siblings, and the process explains why siblings are different.

This also shows that a person will inherit roughly half of one of their parents DNA, and roughly half of that again will be from one of their grandparents.

Sex chromosomes

With the sex chromosomes, things are a bit different. The X-chromosome that each person inherits from their mother, will have been created the same way, with two homologs mixing, because a woman has two X-chromosomes that can be mixed.

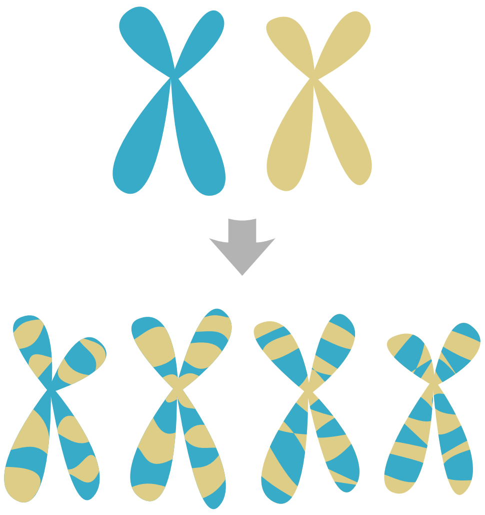


Figure 2.4: An example of how chromosomes are created based on two homologs during meiosis.

However, a man's X-chromosome and Y-chromosome, which are not homologous, are not compatible with this mixing process, and therefore they are passed unchanged to the next generation (roughly half of a man's sperm cells will have his Y-chromosome and the other half will have his X-chromosomes, also implicating that the sperm cells decides the gender of the offspring). In addition, this means that a man's Y-chromosome has been passed relatively unchanged through his paternal line of male ancestors.

Mitochondria

How then, is the mtDNA passed down the generations? Egg cells are a bit more complicated than sperm cells. In a way, egg cells are a kind of starter pack for an embryo to grow and develop, and one of these necessary items in the starter pack is the mitochondria, while sperm cells mainly contain DNA.

So the mitochondrion is passed directly and unchanged from the mother, during meiosis. This, in turn, means that the mtDNA has been passed relatively unchanged through the direct maternal line of a person's female ancestors.

Identity-By-Descent

When two people have inherited DNA from the same ancestor, and they have identical segments of DNA, those segments are said to be identity-by-descent or IBD. Essentially, there has been no intermediate recombination of the segments through the generations. If two people have identical segments of DNA because of coincidences (i. e. they haven't inherited the same segment from a common ancestor), those segments are said to be identity-by-state (IBS).

Thompson (2013) states: "there is no absolute measure of IBD; IBD are always relative to some ancestral reference population". Which means that IBD are measured relative to some point in time and the founder population at that point in time.

Besides, there are different IBD levels. IBD0 is the likelihood that two individuals are unrelated, IBD1 is the likelihood that the individuals are IBD on one haplotype and IBD2 is the likelihood that the individuals are IBD on both haplotypes (Browning & Browning, 2012).

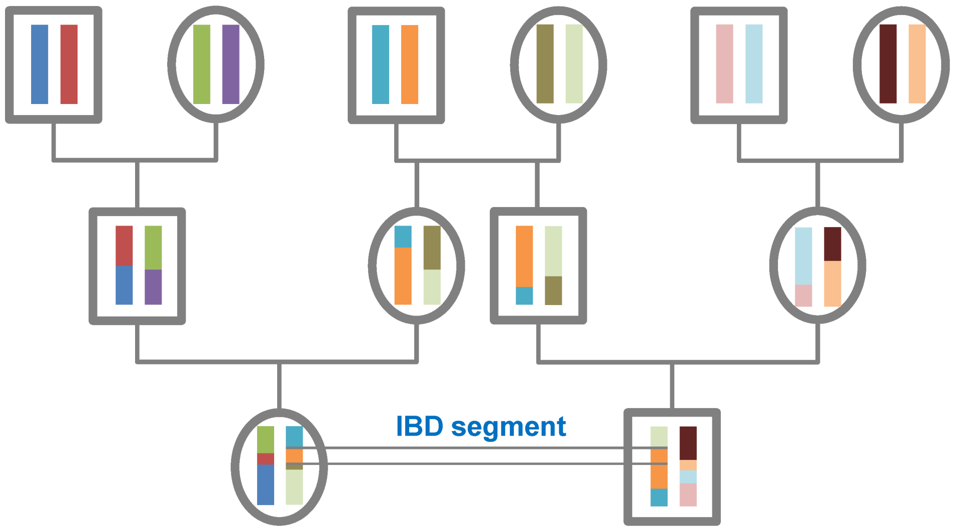


Figure 2.5: This is a pedigree with 12 individuals. Males are shown with squares and females with circles. The two vertical bars in each node represent two homologous chromosomes (e.g. chromosome 1), and the colors indicate how these chromosomes are inherited. It shows how the pair of cousins in the bottom share an IBD segment inherited from their shared grandfather. Source: user:Gklambauer / Wikimedia Commons / CC-BY-CA-3.0 / GFDL

2.2.2 Degree of relatedness

The degree of relatedness follows the average sharing of DNA between two relatives, where first degree relatives share 50% of their genes on average, and these are parent/child- and full sibling-relationships. Second-degree relatives share about 25% of their genes and include an individual's grandparents, aunts, uncles, nephews, nieces and double cousins. 3rd degree relatives share about 12.5% of their genes and include an individual's great-grandparents, great-grandchildren and first cousins.

2.2.3 Pedigrees and cryptic relationships

A *pedigree* is a family tree or DNA tree. It might have been made based on reported relationships, but it can also be inferred from DNA data.

A *cryptic relationship* is an unknown relationship between two individuals. E.g. a reported pedigree on a data set that hasn't included a specific relationship between two individuals.

Chapter 3

Background

3.1 The probability of two individuals having different mitochondrial alleles

Although a SNP can take up to four different alleles, all mitochondrial variants in the HUNT study are one of two possible alleles (e.g. a certain SNP will either have allele 'A' or allele 'C'). This means that when you want to count the number of pairs in a population that do not share a variant, this number is upper bounded, and the highest possible number of pairs is achieved when half of the population has one allele and the other half has the other.

If a population consists of n individuals, where $d = n/2$ has allele A and the same number has allele C, then for every individual with allele A, there will be d individuals with allele C. Another way to view this, is to have two subgraphs with d nodes, and edges between every node in the two subgraph (but none in between nodes in the same subgraph), and then count the number of edges, as shown in Figure 3.1. The total number of pairs with different alleles then becomes:

$$d^2 = \frac{n^2}{2^2} = \frac{n^2}{4}$$

Furthermore, that also affects the probability that two individuals do not share a variant, given two equal sized groups with two distinct alleles. The total number of pairs in a set of n individuals is $n(n-1)/2$, and the upper bound for the probability becomes:

$$\text{P(Individuals do not share marker)} \leq \frac{n^2}{4} \cdot \frac{2}{n(n-1)} = \frac{n}{2(n-1)}$$

This function converges towards 0.5 for increasing n.

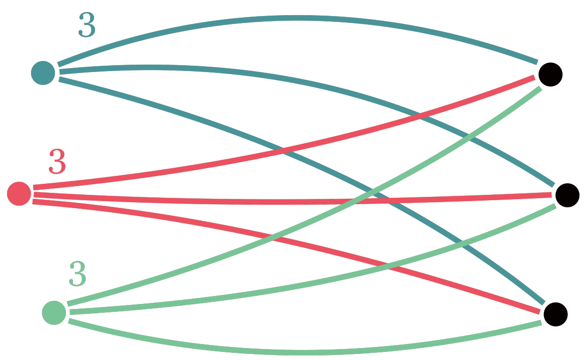


Figure 3.1: An example of a population with six individuals, where an edge between two individuals indicate different alleles on a given marker. The graph is divided in two, with three individuals on each side: The group on the left has one allele and the one on the right has another. As can be seen, for every individual in the left group, there are three edges to individuals in the right group, i.e. a total of 3×3 edges.

3.2 A common maternal ancestor

Assume you pick to individuals at random from a population – how likely is it that they both have a direct maternal line to a maternal ancestor?

3.2.1 Using mitochondrial distance

A way to estimate this, would be to compare their mitochondrial DNA: Two individiuals that have a direct maternal line to a common female ancestor necessarily needs to have similar mitochondrial DNA, because it is the mtDNA that are passed down from mother to child.

Let A and B be two randomly selected individuals from a population, and let the null hypothesis and alternative hypothesis be:

$$H_0 : \text{A and B does not have a direct maternal line to a common maternal ancestor}$$

$$H_1 : \text{A and B does have a direct maternal line to a common maternal ancestor}$$

Furthermore, let $d_{A,B}$ be the mitochondrial DNA distance between them. The idea is that the distance $d_{A,B}$ must be low enough, so that we reject H_0 and accept H_1 – but at what point will the distance be low enough? We can use p-values for this, where the p-value will indicate the probability that two individuals have similar mtDNA simply by chance and not due to common maternal ancestry.

To estimate p-values, we can look at the probability distribution of distances between father/child relationships. The reason being that father and child share environment because of their close relatedness (they share autosomal DNA), but, assuming that father and mother are non-related, their mitochondria DNA is independent. The probability distribution of distances between father/child relationships will therefore serve as our p-values, where this probability distribution is denoted by $f_{\text{father/child}}(x)$. To find the p-value for the distance $d_{A,B}$, the following equation is used:

$$p_{d_{A,B}} = \int_0^{d_{A,B}} f_{\text{father/child}}(x)dx$$

This value denotes the possibility that individuals A and B have similar mitochondria distances simply by chance, and will be the area under the probability distribution of mitochondrial distances in father/child relationships, as shown in Figure 3.2.

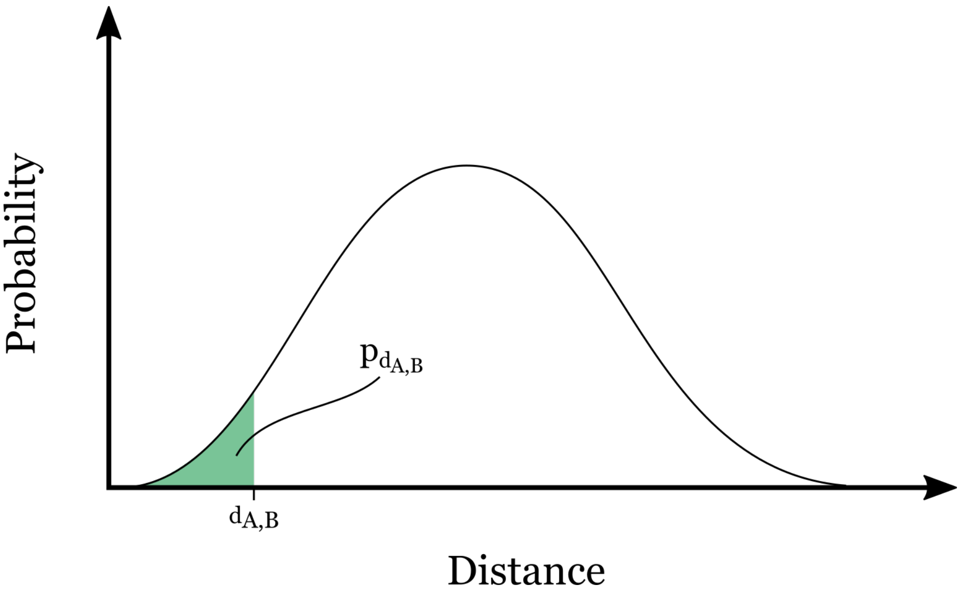


Figure 3.2: An example of a probability distribution on distances between father/child pairs, with the p-value given by the distance $d_{A,B}$ being the area under the distribution.

We can set a threshold α, for what chance we are willing to accept, e.g. if we set $\alpha = 0.05$ we will reject H_0 and accept H_1 with a maximum of 5 % chance that the individuals have similar mitochondrial DNA due to coincidence.

More formally, we reject H_0 and accept H_1 if:

$$p_{d_{A,B}} \leq \alpha$$

Another way to frame this, would be to find the critical value d_c, such that

$$p_{d_c} = \alpha$$

and then we would reject H_0 and accept H_1 if:

$$d_{A,B} \leq d_c$$

3.2.2 Using the degree of relatedness

If you knew their degree of relatedness, it would be possible to estimate the probability that they do have a direct maternal line to a maternal ancestor, with some assumptions. First, assume that there has been no inbreeding in the population, and second, assume that there are no half-siblings in the set – all siblings in the population share the same parents. With these assumptions, the degree of relatedness is the same as the number of parents on the path between the two individuals, as shown in Figure 3.3. The figure shows 2nd through 4th degree relationships, and the number of parents on the paths between the different kinds of relationships is the same as the degree.

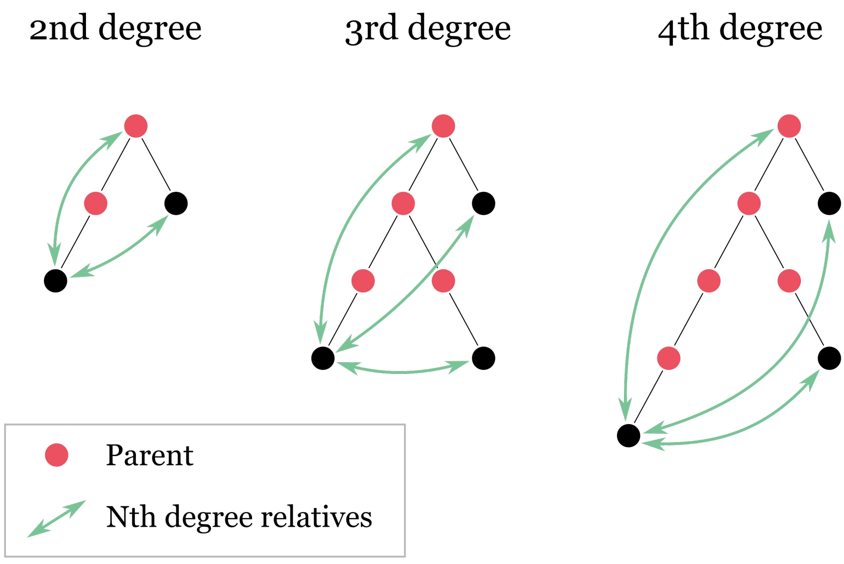


Figure 3.3: An example of how there are n parents on the paths between nth degree relatives, when you assume that all siblings share the same parents and no inbreeding has occured.

The probability that two nth degree relatives have a maternal line to a common maternal ancestor is the same as the probability that all parents on the path between them are female. A parent is either a mother or a father, so the probability A that a parent is female is $P(A) = 0.5$. If you look at the cousins among the 3rd degree relatives (the two bottom leaves in the middle tree), there are three parents between them, and all of these needs to be female. The probability then becomes:

$$P(\text{All parents are female}) = P(A)P(A)P(A) = 0.5^3$$

More specifically: For all D parents on the path between two individuals where D is the degree of relatedness, the probability X_D that the two individuals have a common female ancestor and a direct maternal line to this ancestor is:

$$P(X_D) = P(A)^D = 0.5^D$$

3.3 A weighted distance

In order to calculate a weighted distance on the mitochondrial DNA between two individuals, some kind of weights are needed, and the probability that two individuals do not share a variant can be used for this.

Ideally, we want variants that has a low probability of being different to have high weights, because these variants are typically family specific, and the variants with a high probability of being different should get a lower weight. In order to achieve this from the probabilities, we take the negative logarithm of the probabilities. More formally, for a variant v and the probability, p_v, that two individuals will not share this variant, the weight will be:

$$w_v = -\log(p_v)$$

Furthermore, if the function $f(v)$ is defined as:

$$f(v) = \begin{cases} 1 & \text{The two individuals do not share the variant } v \\ 0 & \text{The two individuals share the variant } v \end{cases}$$

Then the weighted distance $d_{A,B}$ between two individuals A and B for variants $v = 1...n$ is defined as:

$$d_{A,B} = \sum_{v=1}^{n} w_v f(v)$$

If you did not take the negative logarithm of the probabilities and instead multiplied the probabilities together, this would be the same as getting the probabilities that two individuals do not share the specific combination of variants.

3.4 The HUNT study

The ”Nord-Trøndelag Health Study” (HUNT) is a large norwegian health study started in 1984, where about 120 000 people from Nord-Trondelag has participated and provided health information. The study has been through three iterations, the first in the eighties, the second in the nineties and the last in the 2006-08. In addition, a state-of-the-art biobank was created in the latest survey, with DNA from about 70 000 of the participants gathered, which includes both nuclear DNA and mitochondrial DNA.

Information about the specification of the HUNT set is found in Appendix F.

3.5 Software

3.5.1 Sequoia

Sequoia (Huisman, 2017) is a pedigree reconstruction method, presented as an R package, that considers up to third degree relationships, and it can handle data sets with non-genotyped individuals, inbreeding and unknown birth years. The method assigns first-, second- and third-degree relationships between individuals, by considering the likelihoods between all these relationships – including their inbred derivatives – and the alternative of being unrelated. It takes a hill-climbing approach to the problem, by dividing the problem in several iterations:

- It starts with parentage assignment, which is done in several iterations until the total likelihood asymptotes. It also makes use of filtering to speed things up, e.g. by looking at opposite homozygosity to exclude pairs of individuals as potential parent-offspring pairs, and subsequently reduce the amount of parent-offspring pairs to consider (i.e. if one individual has genotype AA on a marker, and another has CC, then one of them cannot have inherited an allele from the other).
- It continues with finding clusters of half-siblings with one ungenotyped parent and assigns dummy-parents to these clusters. Individuals may be assigned as parents to the dummy-parents (i.e. if the individuals are grandparents of the sibship-clusters). This is also done in an iterative fashion.
- It makes use of sex- and age-information, to ensure that (i) an individual cannot be its own ancestor, (ii) ancestors are older than their descendants and (iii) parents are of opposite sex.

For full reconstruction down to 200 independent SNPs, they reported a low error-rate ($< 0.3\%$) and high assignment rates ($> 90\%$) in a limitied computation time (< 1 h).

3.5.2 DRUID

DRUID (Ramstetter et al., 2018) can do pedigree reconstruction and relatedness estimation. It is an IBD-based method that uses close relatives to infer an ungenotyped ancestor's IBD sharing profile, that way finding up to 12th degree relatives. More specifically, DRUID leverages multi-way relatedness by taking several individuals into account when determining relationships. An example of this is how the method combines siblings' DNA to infer what their ungenotyped parent's DNA looks like – a pair of siblings will has more information about an ungenotyped parent than only one of them. This information is further used to find aunts and uncles of the siblings – the authors reported recovering 92.2 % of real aunts/uncles with zero false positives and $>$ 79 % of 10th degree relatives correctly or within one degree.

Chapter 4

Methods

4.1 Analysis

Ten subsets were created with 1500 individuals randomly selected from the dataset. The mitochondria of all pairs of individuals in the subsets where compared, and these results were further compared with the degree of relatedness of the individuals, estimated by DRUID (Ramstetter et al., 2018).

This section will further describe how the analysis of the data was executed.

4.1.1 Selecting subsets

The complete script for selecting random subsets is found in Listing D.3.

Individuals were randomly chosen from those in the HUNT study that had their mitochondrial DNA sampled and an inbreeding coefficient below 0.025, since DRUID assumes no inbreeding (see subsection B.2.1 for calculation of the inbreeding coefficient) (Ramstetter et al., 2018).

The selecting was done with Python's function `random.sample()`, which will choose `k` unique values from the provided list, as seen in Listing 4.1.

```
    matrix = []
    for i in range(num_sets):
        matrix.append(random.sample(list(ids_choose.index), k=num_ind))
```

Listing 4.1: An excerpt from `get-random-subsets.py`, where the random selection of individuals is done. The complete script is found in Listing D.3.

Furthermore, these individuals' phased genotypes had to be extracted from the `.vcf` file created by the phasing program BEAGLE (for use with DRUID – see the next section). Since `.vcf` files has one column for each individual and one row for each variant, this was done with the `cut` command in Linux (in addition, this file is very big – opening it in e.g. Python would probably be impossible). To get the correct columns, the header of the BEAGLE `.vcf` file was read into the script, and the column number of each individual for each subset was written out to a file. The corresponding code for this is found in lines 45–71 in Listing D.3.

4.1.2 Running DRUID on all subsets

To run DRUID on the subsets, I followed the same procedure as described in my project thesis (Gudem, 2018). Furthermore, as I'd already used BEAGLE to phase the complete HUNT set, I reused these results (I spent a rocking three weeks waiting for BEAGLE to complete and then it crashed – no point repeating that endeavor). Because of the BEAGLE-crash in the middle of chromosome 20 (due to how the sex chromosomes where structured), I reran BEAGLE on chromosomes 20–22 on the complete HUNT set, extracted chromosomes 1–19 from the old file and put the old and new files together.

The complete script for running DRUID on all subsets is found in Listing D.4. This pipeline includes:

- Cutting out individuals from the BEAGLE `.vcf` file.
- Running RefinedIBD.
- Running the druid-bakeoff script provided by the authors of DRUID.
- Lastly, running DRUID.

4.1.3 Calculating mitochondrial distances

The function, `mitomatch`, has been created to compare the mitochondrial DNA between all pairs of individuals in a dataset.

The function requires an instance of `MitochondriaData` and (optionally) a list of individuals' IIDs. Listing 4.2 shows how to initiate an instance of `MitochondriaData`, which requires the path prefix to a `.ped` and `.map` file (see PLINKs file format reference for more information on the `.ped` and `.map` format).

The complete source code of `MitochondriaData` is found in Listing C.3.

```
mito_data = MitochondriaData("/prefix/to/ped-and-map-file")
```

Listing 4.2: An example of how to initiate MitochondriaData.

Hamming distance

`mitomatch`'s default calculation type is the hamming distance, but it can nonetheless be explicitly defined by setting `calculation_type='hamming'`. Listing 4.3 shows hos this was done on the ten subsets.

```
    print("> Reading mitochondria data")
    mito_data = MitochondriaData("/prefix/to/mtdna-ped-and-map")

    print("> Calculating distance for subsets")
    for subset in range(1, 11):
        print("> Subset {}".format(subset))
        ids = pd.read_csv(ID_PATH_PREFIX + str(subset),
            header=None,
            names=["FID", "IID", "ids_beagle_format"],
            sep='\s+')

        # Calculate hamming distances
        distances = mitomatch(mito_data,
            individuals=ids.IID,
            calculation_type='hamming')
        distances.to_csv(DISTANCES_PATH_PREFIX + str(subset) + ".txt",
```

Listing 4.3: How hamming distances for each pair of individuals has been calculated. The full script is located in Listing D.2.

Weighted distance

To calculate a weighted distance on a pair of individuals' mtDNA, `calculation_type` was set to `'weighted'` and weights were provided as a dictionary with variant ids as the key and weights as the value. The script located in Listing D.2 calculates weighted differences for the subsets, and an excerpt is shown in Listing 4.4.

```
# Calculate weighted distances
weighted_distances = mitomatch(mito_data,
    individuals=ids.IID,
    calculation_type='weighted',
    weights=variant_probabilities.variant_log_probabilities)
```

Listing 4.4: How mitomatch is called

The weights, `variant_log_probabilities` were set to be the negative logarithm of the probability p that two individuals don't share a marker, as described in section 3.3:

$$w_i = -\log(p)$$

The approach to find these p-values for every variant has been described in subsection 4.1.6.

T-test on weighted distances

A Student's T-test was applied between 5th–12th degree relatives and the unrelated individuals. For each degree of relatedness, the median values from distances across all subsets were used.

The function `ttest_ind` from scipy.stats was used to run the T-test, as shown in Listing 4.5, and it calculates the T-test for two independent samples.

```
tstats, p_val = stats.ttest_ind( a, b )
```

Listing 4.5: A t-test on the median values of distances between groups of individuals with different degrees of relatedness, where `a` is a list of median values for one kind of relationship and `b` is a list of median values for another kind of relationship, e.g. fifth degree relatives and unrelated individuals.

4.1.4 Comparing mitochondrial differences with DRUID output

When DRUID was run on all subsets and distances were calculated, these results needed to be merged together. More specifically, the output of DRUID had to be merged together with the output of `mitomatch`.

On thing to note, however, is that for a given pair of individuals with ids `a` and `b`, their ids could be stored as `a | b` in the DRUID output and as `b | a` in the `mitomatch` output or vice versa. Therefore, as shown in Listing 4.6, joining `pairwise_distances` with `druid` is done in two turns, once with `right_on` set to `['ind1', 'ind2']` and once with it set to `['ind2', 'ind1']`.

```
result = pd.merge(druid, pairwise_distances,
    left_on=['ind1', 'ind2'], right_on=['ind1', 'ind2'], how='inner')
result = result.append( pd.merge(
    druid, pairwise_distances,
    left_on=['ind1', 'ind2'], right_on=['ind2', 'ind1'], how='inner'),
    sort=False)
```

Listing 4.6: This listing shows how the output from DRUID and mitomatch is merged together into one dataframe. For every subset, the output from DRUID and mitomatch has been read into the variables `druid` and `pairwise_distances`, respectively.

4.1.5 Calculating missingness in the data set

PLINK can be used to calculate the missingness in `.ped` and `.map` files as shown in Listing 4.7.

```
$ plink2 --bfile mtdna-ped-and-map-prefix --keep individuals.inds
    --missing --out missing
```

Listing 4.7: Calculating missingness in a genotype data set with PLINK.

This will produce two files, `missing.imiss` and `missing.lmiss`, the former describing the amount of missingness for each individual listed in the file `individuals.inds` and the latter for each variant.

This was calculated separately for each subset of individuals.

4.1.6 The probability of mtDNA variant differences

To find the probability that two individuals don't share a mtDNA variant, the `mitomatch` function was used, with the argument `calculation_type` set to `"true-false-vector"`, as shown in Listing 4.8.

```
mitochondria_differences = mitomatch(mito_data, individuals=subset,
    calculation_type="true-false-vector")
```

Listing 4.8: An example of getting mitochondria differences with `mitomatch`.

This will produce a dataframe with one row for each pair of individual, and one column for each variant, where the columns will be `True` if the individuals share an allele or

False if they do not. This was done for each subset, and the script that does this for all subsets and writes the results out to a file is found in Listing D.2.

Furthermore, the number of False values were counted for each variant, to get the total number of pairs who do not share alleles on the markers, as shown in Listing 4.9.

```python
# For each subset, mitochondria_differences has been read in from file
# We don't need ids
mitochondria_differences = mitochondria_differences.iloc[:, 2:]

# Count number of different values
counted_differences = mitochondria_differences.apply(
lambda column: column.value_counts()
).fillna(0).loc[False]
```

Listing 4.9: Counting the number of mitochondrial variants that differ. An excerpt taken from Listing D.1.

To get the probabilities of each subset, the number of counted differences for every variant were added by a pseudocount of 1 and divided by the total number of possible pairs, which is $n(n-1)/2$ where n is the number of individuals in the subset (i.e. 1500). The final probabilities were set to be the average probability over all subsets, i.e. the sum of all subsets' probabilities divided by 10. This is shown in Listing 4.10.

```python
num_pairs = 1500*1499 / 2
probabilities_all_subsets = counted_differences_all_subsets.apply( lambda
     row: (row + 1) / num_pairs, axis=1 )
average_probabilities = probabilities_all_subsets.mean()
```

Listing 4.10: Getting the probabilities for all subsets. An excerpt taken from Listing D.1.

The script that reads in mitochondria differences calculated by mitomatch and calculates the probabilities are found in Listing D.1.

4.2 Trio data

I had access to a data set of mother/father/child trios in the HUNT study, that had been estimated with (Manichaikul et al., 2010). This set initally consisted of about 16 000 trios, but only the trios where all three had their mitochondrial DNA sampled were used, which resulted in 7777 trios, with about 5000 pairs of parents (and hence, some siblings as well). Furthermore, DRUID were run on all mother/father pairs, and mitochondrial distances were calculated on mother/child- and father/child-pairs, as described in the next sections.

4.2.1 Estimating the degree of relatedness between parents

DRUID were estimated between all mothers and fathers in the trios. However, some additional filtering were needed. DRUID will estimate the relatedness between all pairs in a set, but I was only interested in the degree of relatedness between the pairs of parents.

Before running DRUID, the parents were split into ten subsets, ensuring that if a mother of a child were in a subset, the child's father would also be in the subset. The same approach and script described in subsection 4.1.4 were used to run DRUID on these subset, with some slight modifications to the script.

When DRUID completed, every pair of parents were extracted from the results, ignoring all degrees of relatedness calculated between pairs of individuals that were not parents of the same child. The degree of relatedness estimated by DRUID were added to the trio-data.

4.2.2 The mitochondrial distance between parents and their children

`mitomatch` were used to calculate the weighted distance between father/child- and mother/child-relationships. `mitomatch` also calculates the distance between all pairs of a set, so a script were written that runs `mitomatch` on small subsets to speed things up, and extracts only the wanted pairs from the results. In addition, to ensure that the parents were unrelated, only the trios where the parents were seventh degree relatives or above were used. The script for doing this is found in Listing D.5.

Both Hamming distances and weighted distances were calculated, with the weights being negative log-probabilities, as described in section 3.3.

4.2.3 Estimating a density function

To estimate a density function on the distribution of distances between father/child relationships, the `KernelDensity` class from the scikit-learn library was used. The code for doing this is shown in Listing 4.11.

```
    pairwise_distances = pd.read_csv('/path/to/trio-distances',
        header=0, sep='\t')

    df = pairwise_distances.loc[
        pairwise_distances.relationship == 'father/child', 'distance'
        ]
    max_distance = max(df.values)
    distances = df.values.reshape(-1,1)

    kde = KernelDensity(
        kernel='gaussian',
        bandwidth=BANDWITH).fit(distances)

    scale = 10
    x = np.linspace(0, max_distance + PADDING,
        num=scale*(max_distance + PADDING))

    log_probabilities = kde.score_samples(x.reshape(-1,1))
    probabilities = np.exp(log_probabilities)
```

Listing 4.11: Estimating a density function based on the distribution of distances between father/child pairs with the `KernelDensity` class from scikit-learn.

A Gaussian kernel was used for fitting the density function to the data, and the probabilities are stored in the `probabilities` variable. This was done on both hamming distances and weighted distances, with a `BANDWITH` of 1 and 5 and a `PADDING` of 3 and 20, respectively.

4.3 Mitodetect

Mitodetect calculates mitochondrial distances between individuals in a set and does the hypothesis test described in subsection 3.2.1 on the pairs of individuals, to decide whether they are likely to have a direct maternal line to a common maternal ancestor.

To run Mitodetect, a `.ped` and `.map` file with mitochondrial genotype data is needed, and either a list of individuals or the a pedigree calculated by Sequoia is needed. If it gets a list of individuals, it will compare the mitochondrial distance between all pairs listed in the file. If it gets a Sequoia-pedigree, it will find all maternal founders in the pedigree, and calculate the mitochondrial distance between these.

To run Mitodetect with a list of individuals, the following command can be issued:

```
$ python mitodetect.py prefix/to/ped-and-map-file path/to/output
    --individuals path/to/individuals
```

The file with individuals must be whitespace-delimited, with the first two columns having family ids and individual ids (FID and IID).

To run Mitodetect with a pedigree calculated by Sequoia, the following command can be issued:

```
$ python mitodetect.py prefix/to/ped-and-map-file path/to/output
    --pedigree path/to/pedigree.txt
```

In addition, it is possible to give the following flags:

`--alpha`
The threshold for when to reject H_0 and accept H_1 as described in subsection 3.2.1 (default: 0.05).

`--hamming`
Calculate Hamming distances.

`--p_value_file`
A distribution to use for estimating a kernel density function. If this is not provided, a pickled kernel density function will be used, which is based on father/child mitochondrial distance (either Hamming distances or weighted distances) where mother and father has a degree of relatedness of 7 or above.

4.4 Sequoia

4.4.1 Installation and documentation

Sequoia is available from CRAN, and can be installed in R with the command `install.packages("sequoia)`. It is also possible to download the source code from https://github.com/JiscaH/sequoia and compile it oneself.

Furthermore, the R command `vignette("sequoia")` gives the documentation.

4.4.2 Execution

Sequoia's pipeline is shown in Figure 4.1. Sequoia requires genotype data as a numeric `GenoM` matrix, with one line for each individual and one column per SNP, coded as zero, one or two copies of the reference allele (e.g. if C is the reference allele, then genotype AA would be coded as 0, AC as 1 and CC as 2). In addition, the authors recommend to prune the data to about 500-700 independent SNPs.

Sequoia will also use age and sex-information if available.

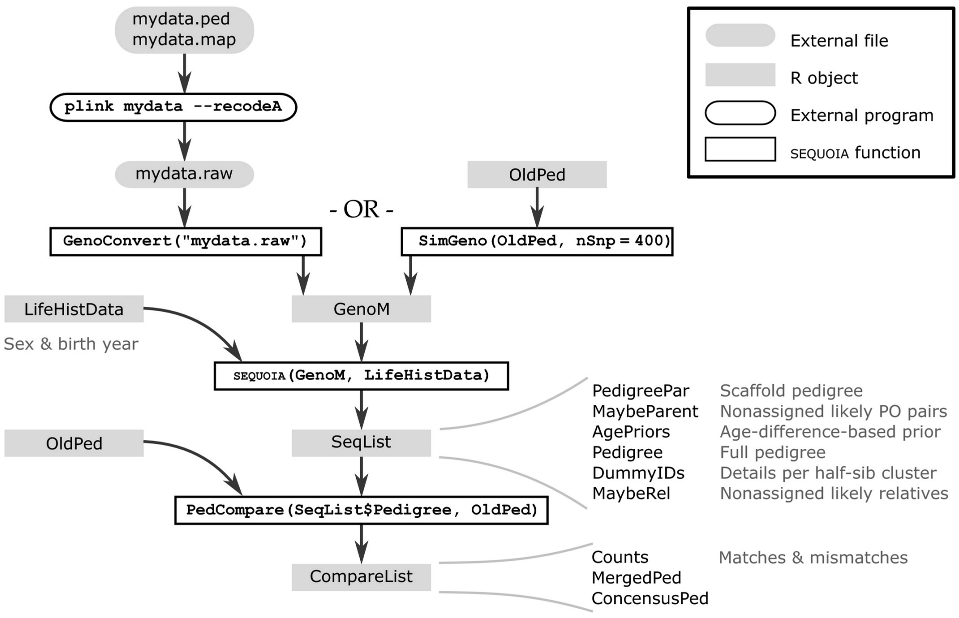

mydata.ped
mydata.map
plink mydata --recodeA
mydata.raw
GenoConvert("mydata.raw")
- OR -
OldPed
SimGeno(OldPed, nSnp = 400)
External file
R object
External program
SEQUOIA function
LifeHistData
Sex & birth year
GenoM
SEQUOIA(GenoM, LifeHistData)
OldPed
SeqList
PedCompare(SeqList$Pedigree, OldPed)
CompareList
PedigreePar Scaffold pedigree
MaybeParent Nonassigned likely PO pairs
AgePriors Age-difference-based prior
Pedigree Full pedigree
DummyIDs Details per half-sib cluster
MaybeRel Nonassigned likely relatives
Counts Matches & mismatches
MergedPed
ConcensusPed

Figure 4.1: The pipeline of SEQUOIA. Source: Huisman, 2017 / CC-BY-CA-4.0

Preprocessing

I found that using the following flags and values gave the desired number of SNPs on the HUNT set:

`--maf 0.4985`, which is the minor-allele frequency – it was this parameter that affected the resulting number of SNPs the most.

`--geno 0.1`

`--indep-pairwise 200 20 0.05`

To prune the data set and extract the wanted SNPs, the commands shown in Listing 4.12 was issued.

```
$ plink2 --bfile genotyped --geno 0.1 --maf 0.4985 --indep-pairwise 200 20
     0.05 --out genotyped-pruned
$ plink2 --bfile genotyped --extract genotyped-pruned.prune.in --out
    genotyped-geno-0.1-maf-0.4985-indep-pairwise-200-20-0.05 --make-bed
```

Listing 4.12: Commands showing how to prune and extract SNPs from the HUNT set, resulting in a set with 500-700 SNPs.

The required coding of the SNPs can be obtained by using PLINK and the parameter `--recode A` on the pruned data set. To extract the wanted subset of individuals and get the file ready for Sequoia's GenoConvert function, the following command was issued:

```
$ plink2 --bfile genotyped-geno-0.1-maf-0.4985-indep-pairwise-200-20-0.05
    --keep individuals-to-keep.txt --recode A --out recode-A
```

Listing 4.13: A command to filter out individuals listed in the file `individuals-to-keep.txt` and recode SNPs to show the count of the reference allele.

Life history data

The life history data should be an R dataframe with columns ID, sex and birth year. The sex column codes female as 1 and male as 2 – note that this is opposite of e.g. PLINK. Column names are ignored, and therefore the order of the columns are critical.

This dataframe doesn't have to include information of all individuals, it can contain more individuals than the genotype data, and it doesn't need to be ordered.

One thing to note is that Sequoia will by default not use family ids (FID), but if you want it to use family ids, the ids specified in `lifeHistData.txt` should be on the format `<FID>__<IID>` – FID and IID separated by two underscores.

Running Sequoia

The R-script shown in Listing 4.14 will run Sequoia in two steps – first assigning parents to the individuals in the data set, and then assign more relatives, as described in subsection 3.5.1.

```
# Read in the genoM
genoM <- GenoConvert(InFile='/path/to/genotype-files.raw',
    UseFID=TRUE)

# Read in life history data
lifeHistData <- read.table('/path/to/lifeHistData.txt', header=FALSE)

# Assign parents
parOUT <- sequoia(GenoM=genoM, LifeHistData=lifeHistData, MaxSibIter=0)

# Find pedigree
seqOUT <- sequoia(GenoM=genoM, SeqList=parOUT, MaxSibIter=5)

# Save results in designated folder:
writeSeq(SeqList=seqOUT,
    GenoM=genoM,
    folder='path/to/sequoia-output-folder')
```

Listing 4.14: The script for running Sequoia in R.

Chapter 5

Results and discussion

This chapter starts with the results of the quantitative analyses run on ten subsets of 1500 individuals randomly chosen from the HUNT study, in section 5.1. More specifically, Hamming distances has been calculated on all pairs of individuals' mtDNA. These Hamming distances were further used to estimate probabilities that individuals will not share a given mitochondrial variant, and the probabilities were used to estimate weighted mitochondrial distances. In addition, genotype missingness in the subsets – the amount of genotypes that are missing due to e.g. errors when sampling – were looked into, described in subsection 5.1.4.

The chapter then continues with the results of the quantitative analyses run on father/-mother/child trios, in section 5.2. Hamming distances and weighted distances were calculated between father/child- and mother/child pairs, were father and mother were 7th degree relatives or more distantly related. In addition, the distribution of mitochondrial distances between father/child pairs were used to estimate p-values for use in the hypothesis test described in subsection 3.2.1.

The chapter then ends with a description of the program Mitodetect in section 5.3 and how the function `mitomatch` was optimized in section 5.4. Mitodetect uses `mitomatch` to calculate mitochondrial distances between individuals in a data set and applies the hypothesis test described in subsection 3.2.1, to decide whether the individuals are related or not.

5.1 Mitochondrial DNA comparisons and the degree of relatedness

Similarities in autosomal variants are quickly lost with increasing degrees of relatedness such that 5th degree relatives have only about 3% inherited variants in common. As mitochondrial DNA instead is maternally inherited, it would be interesting to see how mitochondrial DNA differ compared with the degree of relatedness – more specifically, if one has compared the mtDNA between all pairs of individuals in a population, how would it look like, when also taking their degree of relatedness into account? Naturally, one would expect the distributions to indicate more similar mtDNA for close relatives, but at what point – at what degree of relatedness – do the distributions of mtDNA differences look like those of unrelated individuals?

To answer these two questions, ten subsets with 1500 individuals were randomly chosen from the HUNT study, mitochondria DNA were compared for every pair of individual and their degree of relatedness were estimated with DRUID (Ramstetter et al., 2018).

5.1.1 Hamming distance

Using the hamming distance when comparing two individuals' mtDNA was the logical place to start. Given two individuals' mtDNA, the number of different variants was counted, meaning that if the hamming distance between two individuals' mtDNA were 10, they would have 10 variants with different alleles. The approaches used to find the hamming distance and compare the results with DRUID results are found in section 4.1.3 and subsection 4.1.4.

Figure 5.1 shows a letter value plot (Hofmann, Kafadar, & Wickham, 2011) of the distribution of mitochondrial distances for different degrees of relatedness in subset 1, where letter values are used to approximate more quantiles to give more information about the tails of the distributions. The plots for the remaining subsets are found in section E.1.

As seen in Figure 5.1, the distributions are moving upwards from 1st degree relatedness and seems to stabilize around 5th-7th degree relatedness. The tail of 6th degree relatives is slightly bigger compared to 7th degree relatedness, a tendency shown for all the subset plots, found in section E.1. The results for 12th degree relatives are somewhat surprising though, but these results are most likely due to the sample size and coincidences, discussed more thoroughly below.

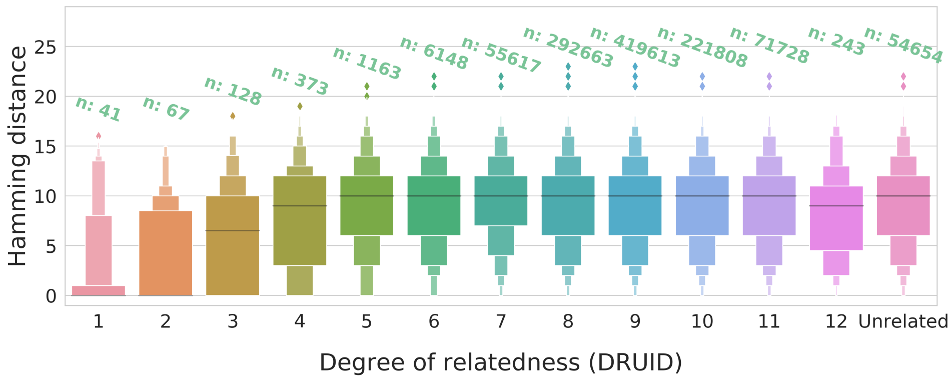


Figure 5.1: The plot shows the distributions of Hamming distances over all degrees of relatedness for subset 1, where n is set to be the number of relatives found for each degree. The distributions are shown as letter value plots (Hofmann, Kafadar, & Wickham, 2011), where letter values are used to approximate more quantiles, and hence show more information about the tails of the distribution. The black horizontal lines indicate the median, and the inner boxes correspond to the first upper and lower quantiles (i.e. 25% and 75%). The dots represent outliers, defined to be a proportion of 0.007.

Figure 5.2 shows the result of the comparison between mitochondrial distances and the degree of relatedness, over all subsets. More specifically, the plot shows boxplots for the upper quantiles (75%), median values and lower quantiles (25%) over all subsets.

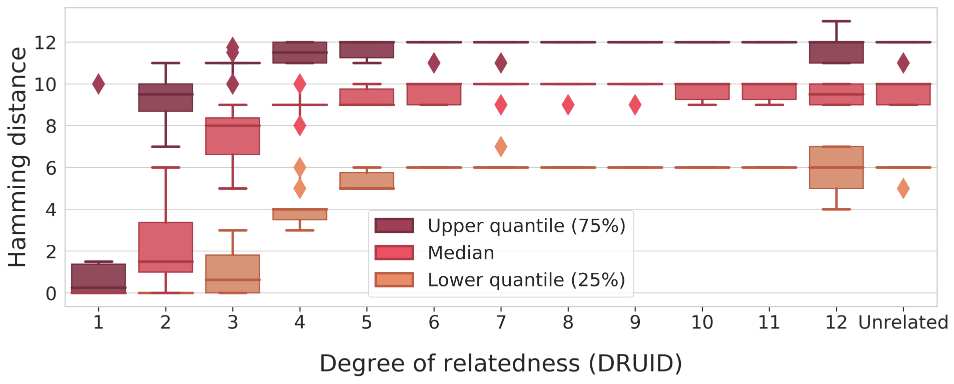


Figure 5.2: This plot shows the Hamming distance on the y axis and the degree of relatedness on the x axis, along with the distribution of the upper and lower quantiles (75% and 25% respectively) and the median values for all ten subsets.

The median (and lower quantile) is 0 for first degree relationships, which includes full siblings, mother/child and father/child relationships – DRUID found considerably more siblings than parent/offspring relationships in the subsets, and siblings are expected to share mitochondrial DNA. The median then increases and stabilizes on distance 10 for 5th degree relatedness. The upper and lower quantiles are also increasing and seem to stabilize from 6th degree relatedness.

It is worth mentioning that 12th degree relatedness has a bigger variation, especially on the lower quantile. However, DRUID found a considerably lower number of 12th degree relationships over all subsets (around 200–300) compared to fifth through 11th degree relatedness and ”unrelated” (which ranged from three to five orders of magnitude, as can be seen in Figure 5.1). This lower number of samples has probably created more variance over the subsets, simply due to coincidence.

5.1.2 Mitochondrial DNA variant probabilities

When comparing hamming distances with the degree of relatedness, it was not clear whether differences were more likely to occur at certain positions compared to others. This information could further be used to calculate weighted distances, and it was therefore interesting to see how the variants in the mitochondria DNA varies over the population, e.g. which mitochondrial markers will be shared across the population, and which will vary across the population. The methods used to find the number of differences between individuals mtDNA and the corresponding probabilities are described in subsection 4.1.6.

Figure 5.3 shows a heatmap with subsets on the y-axis and mtDNA markers on the x-axis, where the color indicates the number of pairs that have different alleles on a marker. What is especially interesting to see, is how uniform the distributions are for a given variant

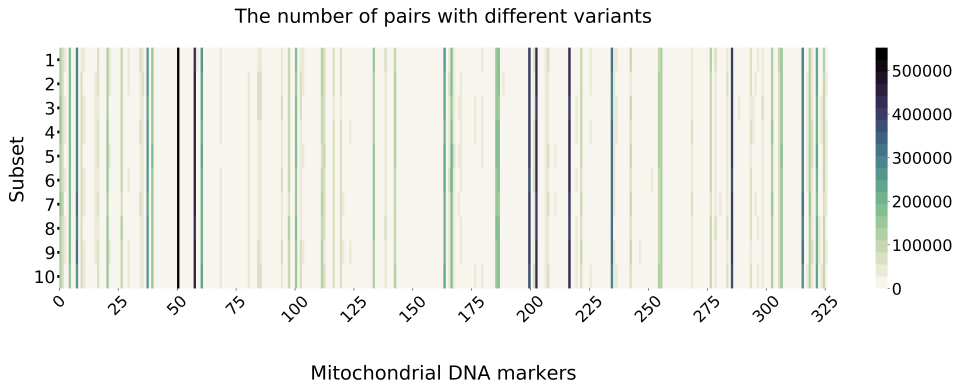


Figure 5.3: This figure shows a heatmap with the ten subsets over the y-axis and the 326 mitochondrial markers on the x-axis. The colors indicate the number of pairs in a subset that has different alleles on a mitochondrial marker, where darker color indicated a higher number.

over all subsets, i.e. a variant that varies a lot over one of the subsets also varies a lot for all the other populations. Note that the number of different variants is upper bounded, as described in section 3.1. When $n = 1500$, the upper bound is $n^2/4 = 562\,500$.

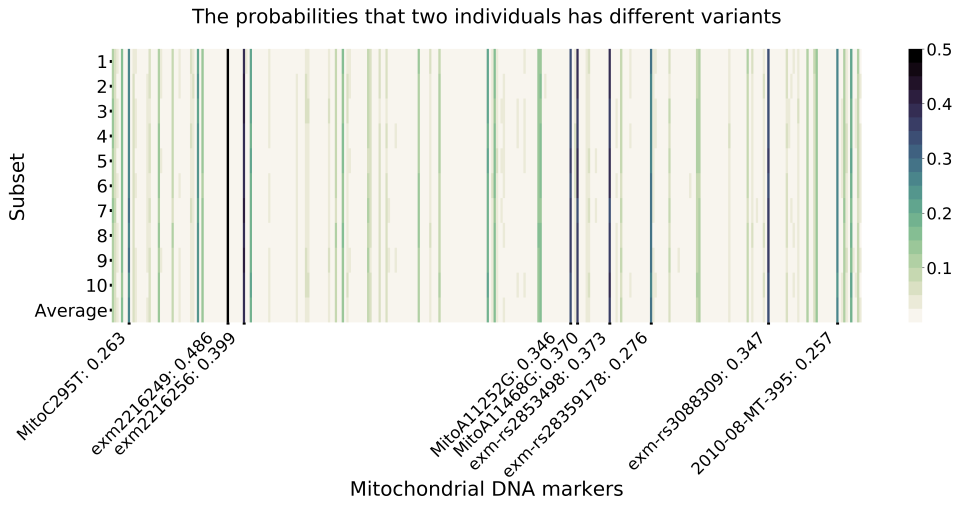


Figure 5.4: This heatmap shows the probabilities that two individuals don't share a variant in the 10 subsets, and the average over all subsets has also been added on the bottom. The variants with an average probability above 0.25 has been marked together with their probabilities.

Figure 5.4 shows a heatmap of the actual probabilities that two markers are different in all subsets. The upper probability is bounded by 0.5003 when $n = 1500$ as described in section 3.1.

This figure shows the same tendencies as Figure 5.3, which is expected, as the highest number of different markers are close to the upper bound of the possible number of pairwise differences.

5.1.3 Weighted distance

By using the probabilities found above, weights were created and used when calculating the distance, where variants with a lower probability of being different where weighted higher, and variants with a higher probability where weighted lower. The methods used to get these results are described in section 4.1.3.

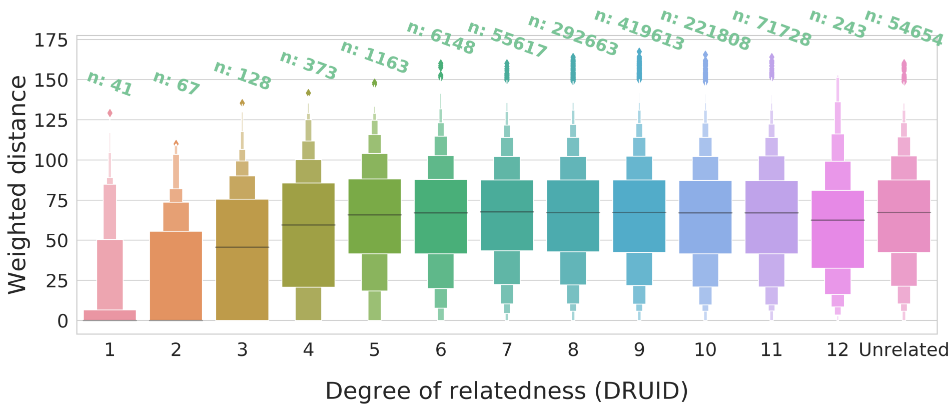


Figure 5.5: The plot shows the distributions of weighted distances over all degrees of relatedness for subset 1, along with the number of relatives found for each degree.

The results for subset 1 are shown as letter value plots in Figure 5.5 and the distributions show some of the same patterns as those for the hamming distance (Figure 5.1), especially when looking at the tails of the distributions. The plots showing weighted distances for all subsets are found in section E.2.

Figure 5.6 shows boxplots for medians, upper quantiles (75 %) and lower quantiles (75 %) over all subsets. Again, note that the median and lower quantile is 0 for 1st degree relatives.

In these results, it is slightly clearer that the distributions for 7th degree relatives and above look more like the distributions for non-relatives. There is not a big difference between sixth and seventh degree relatives, other than the fact that the lower quantile is slightly lower for 6th degree relatives – there is a bigger ratio of pairs with similar mtDNA among the 6th degree relatives.

In order to find out at what degree of relatedness the distributions look like those of unrelated individuals, Student's T-tests has been applied. More specifically, the median values for all subsets of 5th–12th degree relatives were compared with the median values for all subsets of unrelated individuals.

The null-hypothesis is that there is no significant difference the average of the median-values, while the alternative hypothesis is that there is a difference. A small p-value (e.g. $p \leq 0.05$) will indicate that if there is a difference, then there is a 5 % chance for it being

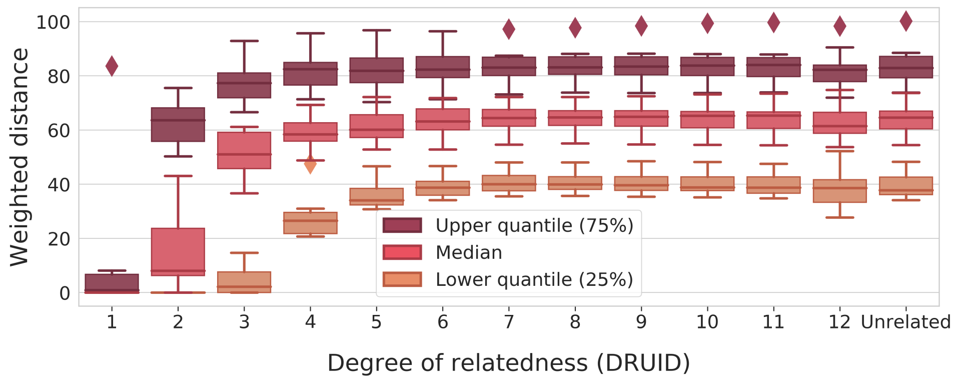


Figure 5.6: This plot shows the weighted distance on the y axis and the degree of relatedness on the x axis, along with the distribution of the upper and lower quantiles (75% and 25% respectively) and the median values for all ten subsets.

Table 5.1: The results of t-tests on the distribution of distances between 5th-, 6th-, 7th-degree relatives and unrelated individuals

	t-value	**p-value**
	Unrelated	
5th degree	3.29	0.004
6th degree	1.27	0.222
7th degree	0.03	0.975
8th degree	−0.21	0.833
9th degree	−0.40	0.696
10th degree	−0.41	0.689
11th degree	−0.33	0.744
12th degree	2.73	0.014

coincidence, and hence we will accept the null-hypothesis for small p-values. The results are shown in Table 5.1.

As can be seen in the table, the p-values are very small for 5th degree relatives versus unrelated individuals, and it is above the threshold of 0.05 for 6th degree relatives. The p-value is also small for 12th degree relatives, but this is most likely due to the low sampling size.

5.1.4 Genotype missingness

One thing to note is how genotype missingness has been handled. When measuring DNA, genotypes might not be sampled due to some sampling error, and this will be indicated in the `.ped` file with a 0. If you have two individuals, A and B, where A has a missing allele for a marker, while B has not – what should you do with the distance? The conservative approach is to assume that A has the same allele and add 0 to the distance, while the alternative is to assume A has a different allele and add 1. `mitomatch` used the conservative

Table 5.2: Genotype missingness in all ten subsets, where the first column shows how many genotypes each individual are missing on average, the second column shows how many individuals who does not have any genotype missingness and the last column shows the three highest numbers of missing genotypes found in individuals for every subset.

Subset	Average number of missing genotypes per individual	Individuals with no missingness (%)	Top 3 missing genotypes
1	0.36	73.40	5, 4, 4
2	0.35	74.60	5, 5, 4
3	0.33	75.47	4, 4, 4
4	0.35	75.07	14, 5, 4
5	0.32	75.67	5, 4, 4
6	0.34	75.27	4, 4, 4
7	0.36	73.20	4, 4, 4
8	0.33	75.53	4, 4, 4
9	0.33	75.07	5, 5, 5
10	0.35	74.40	14, 4, 4

alternative, because assuming missing genotypes are different results in outliers, especially among closely related individuals – a mother and child might get a high distance simply due to missing genotypes.

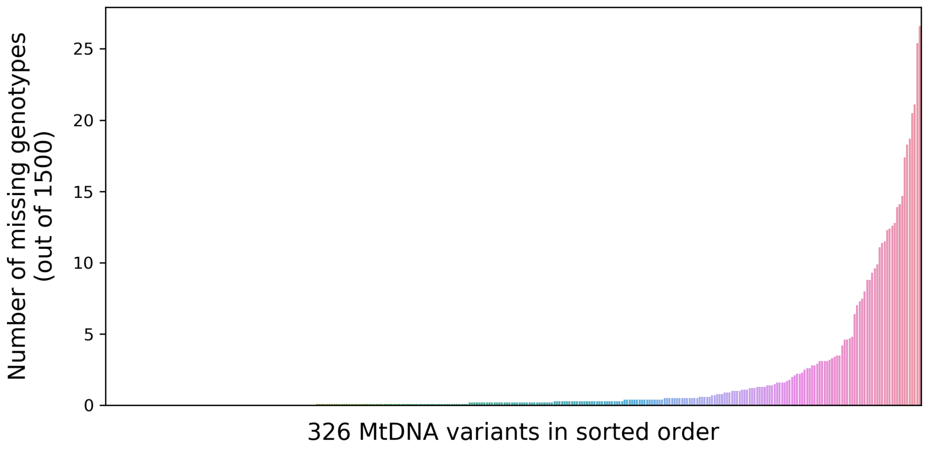


Figure 5.7: The average number of missing genotypes per variant over all subsets, sorted in increasing order.

Table 5.2 shows more specific information about missingness in the ten subsets, namely the average number of missing genotypes for each individuals, the percentage of individuals with no missing genotypes and the top three individuals in each set with the highest number of missing genotypes (the individual with 14 missing genotypes in subset 4 and 10 is the same). Besides, Figure 5.7 shows the average number of missing genotypes per variant over all subsets, and as seen, most mitochondrial variants are successfully genotyped in all individuals. Furthermore, in the most extreme cases, genotyping has failed for about 1.7 % of the individuals on average on all ten subsets.

5.2 Looking at trios

DRUID and `mitomatch` were run on a set of about 7000 trios of mothers, fathers and their children as described in section 4.2. More specifically, the hamming distances and weighted distances were calculated on the mitochondrial DNA between every father/child and mother/child pair, where the mother and father had an estimated degree of relatedness of seven or above, and the results of the distributions are shown in Figure 5.8.

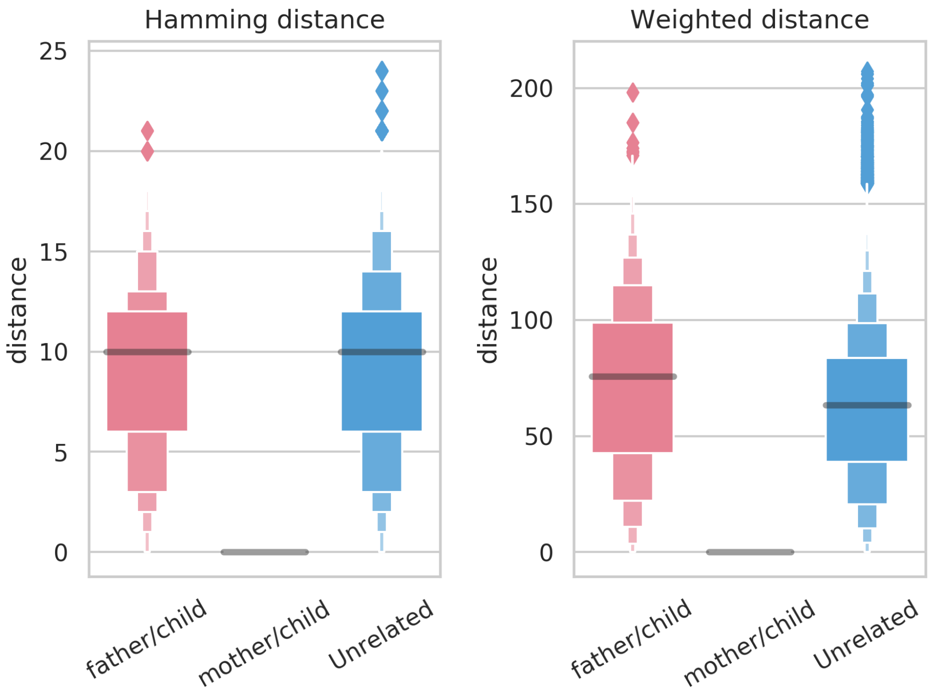


Figure 5.8: This shows the results from calculating Hamming distances and weighted distances on father/child and mother/child relationships, where mother and father are 7th degree relatives or more distantly related, as estimated by DRUID. In addition, the distribution of distances between all unrelated individuals found in the ten subsets has been added for comparison. There were 6953 pairs in the father/child- and mother/child-distributions and 486 394 pairs of individuals in the unrelated distribution.

Every mother/child pair has a distance of 0 between them, while father/child pairs have a distribution more similar to that of unrelated individuals – the hamming distance distribution is almost exactly the same, while the weighted distance distribution lies above for father/child-pairs. The fact that there exist no mother/child pairs with different mtDNA is somewhat interesting – it might indicate that mutations when inheriting mtDNA happen more rarely than in 1 out of 7000 cases. However, it is also worth mentioning that missing genotypes were ignored and assumed similar, which introduces a slight uncertainty.

As discussed in subsection 3.2.1, the distribution of mitochondrial distances between father and their children could be used as p-values, when deciding whether two randomly selected individuals have a direct maternal line to a common maternal ancestor. Figure 5.9 shows the histogram and results of estimating a kernel density function based on the distribution of weighted distances between fathers and their children in the trio data. The method used to estimate a kernel density function has been described in subsection 4.2.3.

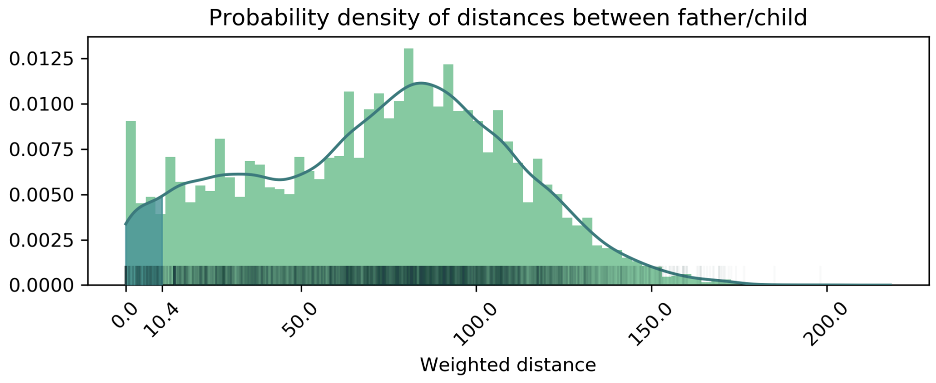


Figure 5.9: This shows the distribution of weighted distances between father/child pairs, where the line indicates an estimated density function. The shaded area under the density-function is equal to $\alpha = 0.05$.

It does seem like there is an overweight of cases with distance 0 in the weighted distribution, which is why a kernel density function has been applied on the Hamming distances between father/child pairs as well, to prove that this is not the case, as shown in Figure 5.10. The overweight of 0-distances in the weighted distribution is due to the fact that all cases where no variants differ between father and their children get distance 0, while cases where e.g. one variant differ between father/child pairs will result in different distances when the variants have different weights. So if there are three pairs that have a Hamming distance of 0 and six pairs with a hamming distance of 1, the same three pairs will get a weighted distance of 0, while the pairs with one differing variant might yield one pair with a weighted distance of 1.5, two with a distance of 2, two with a distance of 3.5 and one with a distance of 4.0.

A weakness with using a kernel density function is the fact that the lowest possible p-value for father/child pairs is 0.02, shown in Figure 5.10. This should be the same for weighted distances, but due to the smoothing of the kernel density function, the lowest p-value for weighted distances lie a little below 0.003, as shown in Figure 5.9.

It is also worth mentioning that there might exist some father/child-pairs with a direct maternal line to a common maternal ancestor among these results, which further increases the uncertainty.

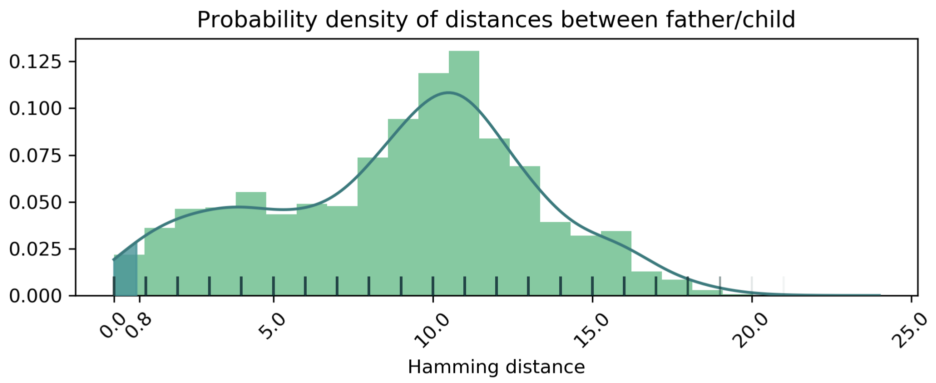


Figure 5.10: The distribution of hamming distances between father/child pairs, where the line indicates an estimated density function. The shaded area under the density-function is equal to $\alpha = 0.05$. The histogram has as many bins as discrete values.

In addition, both plots show a shaded area, which is the area under the distributions that sums to $\alpha = 0.05$. The critical values are 10.2 and 0.8 for weighted distances and Hamming distances respectively and would be the thresholds for when to accept or reject H_0 if $\alpha = 0.05$ given two individuals A and B with distance $d_{A,B}$, as described in subsection 3.2.1.

5.3 Mitodetect

Mitodetect is a program that calculates the mitochondrial distance between a group of individuals, by using the function `mitomatch` (section 5.4) and tests whether the pairs are likely to have a direct maternal line to a common maternal ancestor, by looking at the mitochondrial DNA distances. More specifically, it can calculate the weighted distance or the hamming distance between individuals as described in subsection 4.1.3 and applies the hypothesis test described in subsection 3.2.1 by using the estimated p-values found in section 5.2.

A description of how to use Mitodetect is found in section 4.3.

Mitodetect was tested on all ten subsets, with the weighted distance and with $\alpha = 0.05$. As described above, this yielded a critical value of 10.2, meaning that a pair of individuals with a mitochondrial weighted distance below this value were deemed to have a direct maternal line to a common maternal ancestor. Figure 5.11 shows the results of these runs – more specifically, it shows the average number of individuals that had a distance above the threshold and the number that had a distance below, divided into the degree of relatedness.

For sixth degree relatives and above, there are a little more than 10 times as many pairs of individuals where the null-hypothesis as been withheld, as opposed to the null-hypothesis being rejected with the given alpha value, and this ratio is somewhat stable for all degrees of relatedness above 6th degree.

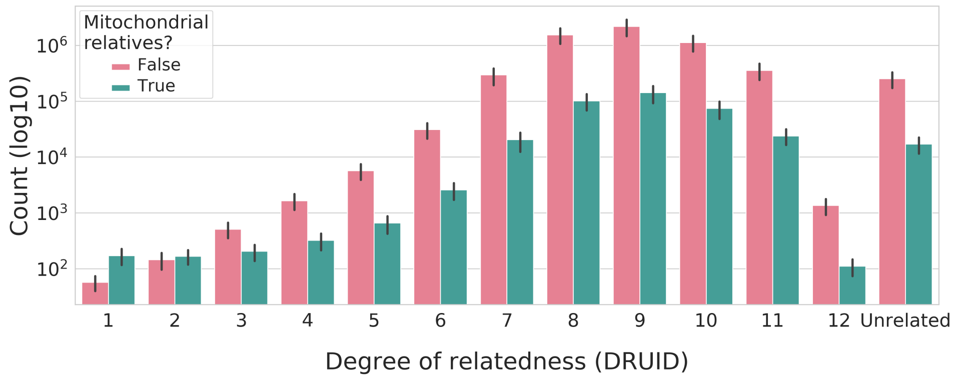


Figure 5.11: The average number of individuals that have and does not have a direct maternal line to a maternal ancestor across all subsets, estimated by Mitodetect. The y-axis shows the count scaled with log10, and the x-axis shows the degree of relatedness. The black vertical lines shows the confidence intervals for the average values.

One important thing to note, is that Mitodetect does not automatically correct for multiple testing. This means that when the alpha-value is set to 0.05, about 5% of the pairs that are deemed to have a direct maternal line to a common maternal ancestor might be false positives. And this might result in a substantial amount of individuals.

It is not necessarily straightforward to correct for multiple testing, because you might end up being too stringent as well. However, one approach could be to use nuclear DNA to determine all groups of individuals who share mitochondrial DNA due to maternal ancestry, like mother/child-pairs and full sibling-pairs, and then use one representative from each of these groups. Mitodetect already identifies these groups for mother/child-pairs if you provide a pedigree reconstructed by Sequoia, and will only calculate mitochondrial distances between maternal founders in the set. However, the ratio of mother/child pairs compared to the total number of pairs in a given subset is very low, making this approach not very effective. Determining groups of individuals based on full siblings and other close relatives as well, might have been an improvement.

5.4 `mitomatch`

The function, `mitomatch`, calculates pairwise mitochondrial differences between individuals. It can calculate the hamming distance, a weighted distance, and output True/False vectors indicating whether a pair of individuals share a variant or not.

5.4.1 Code optimizations

This section describes how I made `mitomatch` run 10 times faster by simply doing code optimizations, from spending about 30 minutes to about 3 minutes on a set of 1500 individuals. The initial code is shown in Listing 5.1. Note that timing and printing have been removed in the code for clarity, and timing was split into total time, access (getting

Table 5.3: The timings and speedup of each iteration of the code optimizations.

	1st iteration	2nd iteration	3rd iteration	4th iteration
Total	29 minutes	21.5 minutes	11.3 minutes	3 minutes
Access:	34.73 %	40.28 %	74.10 %	2.59 %
Calculation:	65.11 %	59.50 %	25.52 %	55.47 %
Storing:	0.06 %	0.08 %	0.12 %	0.31 %
For-loop	NA	NA	NA	41.35 %
Speedup	NA	1.34	2.57	2.98

values from the DataFrame), storage (storing values in a matrix) and calculation (doing the comparison of mtDNA).

Timings and speedup for all the iterations of the implementations are shown in Table 5.3, and as seen in the table, the initial iteration spent about 29 minutes to calculate the hamming distance between every pair in a set of 1500 individuals.

```python
import pandas as pd
import numpy as np

def mitomatch(mito_data, individuals):
    mito_data.set_subset(individuals)
    pairwise_thing = pd.DataFrame(columns=["id1", "id2", "distance"])
    ped_df = mito_data.ped_df
    gsi = mito_data.genotype_start_index
    matrix = []

    for index1 in ped_df.index:
        mito_dna1 = ped_df.iloc[index1, gsi:].values
        id1 = ped_df.iloc[index1, 1]

        for index2 in range( index1 + 1, len(ped_df.index) ):
            mito_dna2 = ped_df.iloc[index2, gsi:].values
            id2 = ped_df.iloc[index2, 1]

            temp = np.equal(mito_dna1, mito_dna2)
            distance = len(temp) - sum(temp)

            matrix.append( [id1, id2, distance] )
    pairwise_thing = pd.DataFrame(matrix)

    return pairwise_thing
```

Listing 5.1: The initial code of `mitomatch`.

Filter out variants shared across entire set

The individuals' mitochondria DNA has been sampled on 326 variants, but many of the variants are shared across the entire subsets. My first idea was, therefore, to simply filter out those variants which were shared across the entire set – why spend time counting differences where there are none?

Listing 5.2 shows the additions to the code. The variant filtering is implemented in the class `MitochondriaData.py`, shown in Listing C.3. Note that the filtering itself was not a part of the timing.

```python
def mitomatch(mito_data, individuals, filter_out_variants=True):
    mito_data.set_subset(individuals)
    mito_data.filter_out_variants() if filter_out_variants else None
```

Listing 5.2: Filtering out variants shared across the entire set.

This resulted in a speedup of 1.34, with the time spent in calculation somewhat improved.

Using functools.reduce()

At this point, the `mitomatch` spent most of its time doing calculations, so that was where I focused my next optimization. Listing 5.3 shows how I tried using `reduce` from the standard library `functools`.

```python
            distance = reduce(
                    lambda total, x: total + int(x[0] != x[1]),
                    zip(mito_dna1, mito_dna2), 0 )
```

Listing 5.3: Using `functools.reduce()` to speed up calculation time.

`reduce` takes one pass over a list, and does some calculations and yields a single result, as opposed to using a combination of `np.equal` and `sum`, which will first calculate and create a true/false numpy array based on the two arrays, and then this true/false-array is passed to the `sum` function – basically, there is more overhead with this, compared to doing a single pass over a list. As seen in Table 5.3, the amount of time spent doing calculations was reduced to 25.52 % and with a speedup of 2.57.

Iterating over DataFrame rows with `.iterrows()`

At last, I looked at whether there was possible to access the DataFrame differently – and there was. Instead of getting the mtDNA and ids with indexing, I used the method `iterrows`, when iterating over the dataframes, as shown in Listing 5.4.

```python
    for index1, row1 in ped_df.iterrows():
        mito_dna1 = row1.values[gsi:]
        id1 = row1.values[1]

        for index2, row2 in ped_df.iloc[index1 + 1:].iterrows():
            mito_dna2 = row2.values[gsi:]
            id2 = row2.values[1]

            distance = reduce(
                    lambda total, x: total + int(x[0] != x[1]),
                    zip(mito_dna1, mito_dna2), 0 )
```

Listing 5.4: Spending more of the time in the for-loop, but reducing access time substantially.

As seen in Table 5.3, this led to another speedup of 2.98, and access time was reduced to a staggering 2.59 % – or, is that true? Well, no. At this point, the for-loop itself had to be timed as well (before this, the time spent handling for loop variables was negligible, and therefore not timed). But by using `iterrows`, accessing the dataframe became a part of the for loop-logic, and as seen in the table, 41.35 % of the time was spent in the for-loop.

All in all, the total speedup ended up being 10.26, all by implementing things a little differently. However, the current implementation of `mitomatch` uses about 5 minutes on a subset of 1500 individuals, due to more features.

5.5 Pedigree reconstruction- and relatedness estimation-methods

In order to answer my first research question, namely to see which pedigree reconstruction methods could be used to find individuals with a direct maternal line to a common ancestor, I tested the methods CLAPPER (Huisman, 2017), DRUID (Ramstetter et al., 2018) and Sequoia (Huisman, 2017).

CLAPPER failed, and PADRE was never tried out, because the program ERSA (Huff et al., 2011) – which PADRE depends upon – failed. The information about how I run these programs and what went wrong is described in Appendix B.

The first author of the CLAPPER software (Ko & Nielsen, 2017) recommended I try the R package Sequoia (Huisman, 2017), which did succeed. The methods for how Sequoia can be run on the HUNT set is described in section 4.4. Sequoia scaled well on a subset of 10 000 individuals, but was never tried out on the full set. However, according to the documentation, a stand-alone version of the algorithm exists, written in Fortran, which does not rely on the same memory limitations as R.

In addition, I tested DRUID (Ramstetter et al., 2018) in my project thesis, and although it doesn't scale well for large sets (Gudem, 2018), it runs well for subsets of 1500 individuals.

However, when selecting subsets of individuals, the chances of the subsets containing close relatives is rather small, and you would therefore not find that many of these relatives that way. Because of this, I focused more on using mitochondrial distances to estimate whether individuals had a direct maternal line to a common maternal ancestor instead. A combination of using pedigrees within subsets and mitochondrial distances could provide a stronger hypothesis test, and would be an interesting thing to look further into.

Chapter 6

Conclusion

As described in the introduction, my goal was to find a way to estimate whether two individuals have a direct maternal line to a common maternal ancestor, and my proposal to this problem is the software Mitodetect. Mitodetect can calculate either the weighted distance or Hamming distance between individuals in a data set, and runs the hypothesis test described in section 5.3 to decide whether the individuals are likely to have a direct maternal line to a common maternal ancestor.

In addition, I have had three research questions to answer:

Question 1: Which pedigree reconstruction-methods can be used to find individuals that have a direct maternal line to a maternal ancestor for close enough relatives?

Question 2: At what degree of relatedness do the distributions of mtDNA differences look like those of unrelated individuals?

Question 3: What is the probability that two individuals will not share a given mitochondrial variant?

The programs Sequoia (Huisman, 2017) and DRUID (Ramstetter et al., 2018) both do pedigree reconstruction, and can be run on the HUNT set. Sequoia scales well for 10 000 individuals, but it has not been tried on the full set – however, this might be possible if one uses the stand-alone program written in Fortran. DRUID is also able to do pedigree reconstruction, but does not scale well for large data sets. Anyhow, the chances of finding close relatives in small subsets are rather low, which made this approach not that effective, and I therefore focused more on looking into mitochondrial distances.

To answer questions 2 and 3, I did quantitative analyzes on ten subsets with 1500 individuals from the HUNT study. More specifically, Hamming distances were calculated between all pairs of individuals, and probabilities that a given mitochondrial variant will be different for a pair of individuals were estimated based on these Hamming distances. The probabilities were further used to calculate weighted distances. T-tests indicated that for 6th degree relatedness and above, the distributions of the weighted distances looked like those of unrelated individuals.

In addition, quantitative analyses were run on father/mother/child trios, in order to estimate the p-values for use in Mitodetect. Both Hamming distances and weighted distances were calculated between the father/child-pairs and the mother/child-pairs, where the father and mother were 7th degree relatives or more distantly related. All mother/child pairs had identical mtDNA, which might indicate that the chances of mutations when inheriting mtDNA might be less than 1 out 7000. The Hamming distribution of father/child pairs were almost identical to the distribution of Hamming distances between unrelated individuals and the distribution of weighted distances between father/child-pairs lay above that of unrelated individuals.

Mitodetect was tested on all subsets, with an alpha of 0.05, and the results showed that for close relatives, more pairs of individuals were deemed to be mitochondrially related for closer relationships, while for 6th degree relatives and above, there were slightly more than 10 times as many non-mitochondrial relatives as there were mitochondrial relatives. However, Mitodetect does not automatically correct for multiple testing, which can give 5 % false positives when $\alpha = 0.05$.

Further work includes making Mitodetect correct for multiple testing, and a way to do this would be to look into using pedigree reconstruction methods more actively to determine groups of individuals that share mitochondrial DNA due to common maternal ancestry. These relationships include mother/child, full siblings and grandmother/grandchild to name a few. That way, you could have one individual per group who acts as a ”mitochondrial representative” for that group – if two mitochondrial representatives are deemed to have a direct maternal line to a common maternal ancestor, this would indicate that all individuals in one group would have a direct maternal line to all individuals in the other group.

It would also be interesting to leverage degrees of relatedness between individuals in addition to mitochondrial distances, to determine whether they share maternal ancestry. If two individuals are closely related and in addition share mitochondrial DNA, this proves as strong evidence that they have a direct maternal line to a common maternal ancestor. But how would this be for more distantly related individuals? At one end, if two individuals are distantly related and share mitochondrial DNA, this can be a strong evidence of them being mitochondrially related. At the other end, the probability that two individuals have a direct line to a common maternal ancestor will decrease when the individuals are more distantly related, as described in subsection 3.2.2 – would this increase the probability that they share mitochondrial DNA by coincidence?

At last, it would be interesting to look at more levels of maternally related individuals, to find the probability that mutations occur when inheriting mitochondrial DNA. No mutations were found between 7000 mother/child pairs – how would it look like for grandmother/grandchild pairs, or great-grandmother/grandchild pairs? If you knew these probabilities, they could be combined with the degree of relatedness.

Bibliography

Auton, A. (2013). Genetic maps generated from the 1000g phased omni data. Retrieved February 13, 2019, from ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20130507_omni_recombination_rates/

Browning, S. R., & Browning, B. L. (2012). Identity by descent between distant relatives: Detection and applications. *Annual Review of Genetics*, *46*(1), 617–633. doi:10.1146/annurev-genet-110711-155534

Daly, A. K., & Day, C. P. (2001). Candidate gene case-control association studies: Advantages and potential pitfalls. *British journal of clinical pharmacology*, *52*(5), 489–499. doi:10.1046/j.0306-5251.2001.01510.x

Gazal, S., Génin, E., & Leutenegger, A. L. (2016). Relationship inference from the genetic data on parents or offspring: A comparative study. *Theoretical Population Biology*, *107*, 31–38. doi:10.1016/j.tpb.2015.09.002

Gudem, A. (2018). Relatedness estimation and pedigree reconstruction on large datasets. Retrieved from http://www.github.com/aliccce/project-thesis-2018

Hofmann, H., Kafadar, K., & Wickham, H. (2011). *Letter-value plots: Boxplots for large data*. had.co.nz.

Huff, C. D., Witherspoon, D. J., Simonson, T. S., Xing, J., Watkins, W. S., Zhang, Y., … Jorde, L. B. (2011). Maximum-likelihood estimation of recent shared ancestry (ersa). *Genome Research*, *21*(5), 768–774. doi:10.1101/gr.115972.110

Huisman, J. (2017). Pedigree reconstruction from snp data: Parentage assignment, sibship clustering and beyond. *Molecular Ecology Resources*, *17*(5), 1009–1024. doi:10.1111/1755-0998.12665

Inbreeding. (n.d.). Retrieved April 28, 2019, from http://abri.une.edu.au/online/pages/inbreeding_coefficient_help.htm

Ko, A., & Nielsen, R. (2017). Composite likelihood method for inferring local pedigrees. *PLoS Genetics*, *13*(8). doi:10.1371/journal.pgen.1006963

Manichaikul, A., Mychaleckyj, J. C., Rich, S. S., Daly, K., Sale, M., & Chen, W.-M. (2010). Robust relationship inference in genome-wide association studies. *Bioinformatics*, *26*(22), 2867–2873. doi:10.1093/bioinformatics/btq559

Ramstetter, M. D., Shenoy, S. A., Dyer, T. D., Lehman, D. M., Curran, J. E., Duggirala, R., … Williams, A. L. (2018). Inferring identical-by-descent sharing of sample ancestors promotes high-resolution relative detection. *American Journal of Human Genetics*, *103*(1), 30–44. doi:10.1016/j.ajhg.2018.05.008

Schon, E. A., DiMauro, S., & Hirano, M. (2012). Human mitochondrial dna: Roles of inherited and somatic mutations. *Nature Reviews Genetics*, *13*, 878. doi:10.1038/nrg3275

Staples, J., Qiao, D., Cho, M. H., Silverman, E. K., Nickerson, D. A., & Below, J. E. (2014). Primus: Rapid reconstruction of pedigrees from genome-wide estimates of identity by descent. *American Journal of Human Genetics*, *95*(5), 553–564. doi:10.1016/j.ajhg.2014.10.005

Staples, J., Witherspoon, D. J., Jorde, L. B., Nickerson, D. A., Below, J. E., & Huff, C. D. (2016). Padre: Pedigree-aware distant-relationship estimation. *American Journal of Human Genetics*, *99*(1), 154–162. doi:10.1016/j.ajhg.2016.05.020

Thompson, E. A. (2013). Identity by descent: Variation in meiosis, across genomes, and in populations. *Genetics*, *194*(2), 301–26. doi:10.1534/genetics.112.148825

Voight, B. F., & Pritchard, J. K. (2005). Confounding from cryptic relatedness in case-control association studies. *PLoS genetics*, *1*(3). doi:10.1371/journal.pgen.0010032

Weir, B. S., Anderson, A. D., & Hepler, A. B. (2006). Genetic relatedness analysis: Modern data and new challenges. *Nat Rev Genet*, *7*(10), 771–80. doi:10.1038/nrg1960

Appendices

Appendix A

Genetic mapping file

The genetic mapping that has been used is based on the Finnish population. The positions were taken from (Auton, 2013). The original data has four fields: Physical position (bp), recombination rate (cM/Mb), genetic position (cM) and a fourth field named ”filtered”.

Only the first three fields are needed, and to clean up the file, the necessary columns were extracted and whitespace characters were replaced by a space. Since the genetic mapping are stored seperately for each chromosome, this must be done on all files. A simple bash-script that converts the files to the desired format is shown below (Listing A.1).

```
#!/bin/bash

for chr in {1..22}
do
    sed '1d' FIN-$chr-final.txt | sed 's/ \+//g' | cut -f 1-3 | sed 's/\t/
      /g' > gmap-FIN-$chr.txt
done
```

Listing A.1: A simple script that creates genetic mapping files.

When using this in combination with PLINK, the `--cm-map` flag is used, and the file path is specified with the wildcard character `@` for the chromosome number:

```
plink2 --bfile input --cm-map gmap-FIN-@.txt --recode --out output
```

Listing A.2: An example of how the genetic mapping files are used in PLINK.

This will yield a `.map` file with positions in centimorgans in the third column, which is often necessary in relatedness analyses (otherwise, these fields are normally zeroed out).

Appendix B

Programs

My plan was to test and use CLAPPER and PADRE. However, I hit some issues that wasn't possible to solve without help from the authors or some access to source code (if I were to do the debugging myself). Unfortunately, neither was available. I will nonetheless document what I did up to the point of failure for the programs I tested.

This appendix will describe what I did and what issues that was in my power to solve, up to the point of failure in both CLAPPER and PADRE.

B.1 Specifications and equipment

The plan was to start with a 1,000 individuals with an inbreeding coefficient below 0.025 (see subsection B.2.1) for both CLAPPER and PADRE, and with initially about 350,000 markers. In both programs, markers ended up being filtered.

The programs and their versions are shown in Table B.1.

Table B.1: The programs and their versions that has been used.

Program	**Version**	**Last changed**
PLINK	1.90p 64-bit	March 25th, 2016
CLAPPER	0.9	March 13th, 2018
SHAPEIT	v2 r904	January 17th, 2018
GERMLINE	1.5.3	October 10th, 2018
ERSA	2.1	No information
PRIMUS	1.9.0	No information

B.2 CLAPPER

The writers of CLAPPER advice users to check whether the population is inbred beforehand, since CLAPPER assumes an outbred population (Ko & Nielsen, 2017).

B.2.1 Inbreeding coefficient

The inbreeding coefficient is the probability that an individual carries two identical-by-descent alleles at a locus, which also means the probability that the parents' respective alleles are identical-by-descent (Weir, Anderson, & Hepler, 2006).

There are several ways of checking inbreeding levels, including approaches that rely on allele frequencies at each marker considered independently (single-point approaches), approaches that rely on IBD segments (multi-point approaches) and approaches that combine the two. PLINK's `--het` option is a single-point approach that uses method of moments-estimation. (Gazal, Génin, & Leutenegger, 2016), and it was chosen because of its simplicity to use.

Calculating the inbreeding coefficient was performed on both a pruned and non-pruned data set.

To prune the data set, the following commands were issued in PLINK:

```
$ plink2 --bfile prefix-to-bed-file --indep-pairwise 50 5 .05 --out pruned
$ plink2 --bfile prefix-to-bed-file --extract pruned.pruned.in --make-bed
    --out pruned-bed
```

Listing B.1: Pruning a data set in PLINK.

And to calculate the inbreeding coefficient, the following command was issued:

```
$ plink2 --bfile prefix-to-bed-file --het --out inbreeding-coefficients
```

Listing B.2: Calculate the inbreeding coefficient in PLINK.

Listing B.2 outputs a `.het` file, which contains one row per individual in the data set, with fields as shown in Table B.2.

Table B.2: The columns of the `.het` file output when running `plink --het`.

FID	Family ID
IID	Individual ID
O(HOM)	Observed number of homozygotes
E(HOM)	Expected number of homozygotes
N(NM)	Number of non-missing genotypes
F	F inbreeding coefficient estimate

More specifically, F is the inbreeding coefficient, which we are interested in.

The results where further processed by finding the median and average inbreeding coefficient:

```
$ awk '{ NR > 1 && total += $6 } END { print total/(NR-1) }'
    inbreeding-coefficients.het
```

Listing B.3: The average of the inbreeding coefficients

```
$ sort -gk 6 inbreeding-coefficients.het | awk '{ a[i++]=$6; } END { print
      a[int(i/2)]; }'
```

Listing B.4: The median of the inbreeding coefficients

The average values for the non-pruned and pruned data sets, were 0.0029484 and 0.00290772 respectively, and the median values were 0.001528 and 0.001635 (again, respectively).

Since both the median and average values are about zero, it is safe to say that the data set as a whole is not inbred. The histogram in Figure B.1 shows the distribution of the inbreeding coefficients based on the pruned data set.

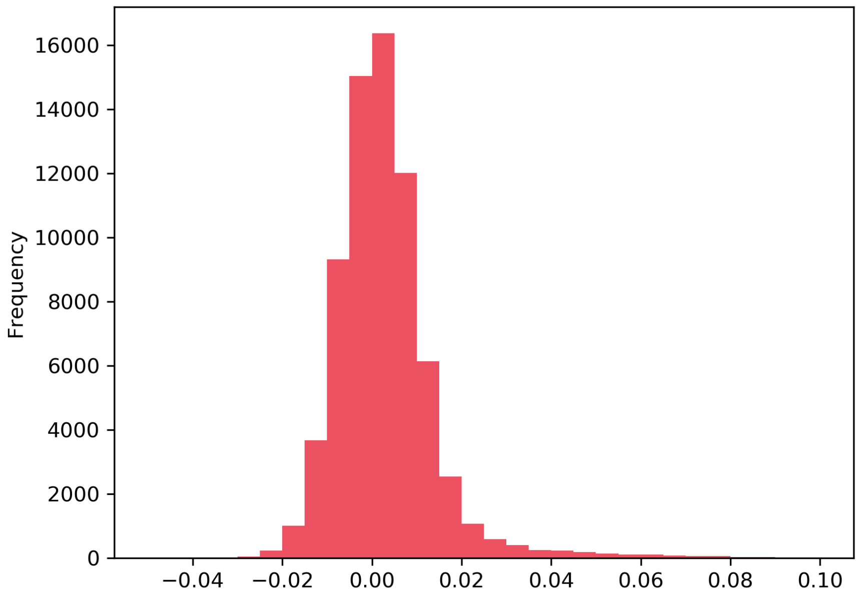


Figure B.1: The histogram of the inbreeding coefficients calculated with PLINK based on a pruned data set.

For further use with CLAPPER, individuals with an inbreeding coefficient above 0.025 were removed from the set, as levels below 0.03 (or 3%) can be considered a low level of inbreeding ("Inbreeding", n.d.).

B.2.2 Input files for clapper

CLAPPER takes `.tped` and `.tfam` files as input, which can be created with PLINK, with `--recode transpose`.

However, a few trial-and-error rounds were needed when trying to run the program. This was issues mainly to do with the input files. To recap:

1. The input files need physical base point positions.
2. Clapper does not accept several markers on the same position.
3. CLAPPER accepts SNPs only, in the form of 'A', 'C', 'G' and 'T' alleles – there exists variants in the HUNT set that are multi-nucleotide alleles (markers with several alleles, e.g. like 'tgaggc').
4. CLAPPER does not support missing genotypes, which are coded as a pair of 0's in the `.tped` file.

Issue 1), 3) and 4) were solved with PLINK, while issue 2) was solved after PLINK had created the `.tped` and `.tfam` files.

Issue 1, 3 and 4

Issue 1) and 3) are (more or less) straightforward to solve. To ensure that the `.tped` has physical positions, the flag `--cm-map` must be provided together with files containing genetic mappings separated into one file for each chromosome (where `@` serves as the wildcard character for chromosome number, as seen in Listing B.5). The creation of these files are described in A. Furthermore, to ensure that the file only contains SNPs (markers with only one allele), one simply adds the flag `--snps-only` to the PLINK command. This will remove markers like 'tgaggc'.

Issue 4) can be tackled in three ways: Either remove the individuals that has missing genotypes, remove the markers that has missing genotypes or do both. Since the HUNT study has lot of individuals genotyped on a lot of markers, doing one or the other will reduce the set substantially. Therefore, I did a combination of the two. The flags `--mind X` and `--geno Y` will filter out individuals and markers with a missingness exceeding X and Y respectively. The values `--mind 0.001` and `--geno 0.001` were sufficient to ensure no missing genotypes in the resulting subset.

The resulting PLINK command is shown in Listing B.5. The `--keep` flag simply keeps the individuals listed in `individuals-to-keep.txt`, which should be space- or tab-delimited, with the family-id in the first column and the individual id in the second column.

```
$ plink2 --bfile genotype-data --keep individuals-to-keep.txt --cm-map
    gmap-FIN-@.txt --mind 0.001 --geno 0.001 --snps-only --recode
    transpose --out output-prefix
```

Listing B.5: The PLINK command that solves issue 1), 3) and 4).

Issue 2

Issue 2 is dependent on the precision of the map-file, so a possible solution would be to increase the precision or create a new genetic mapping based on the population in Nord-Trondelag. However, creating a genetic mapping is not straightforward, and since I was more interested in getting a successful run of CLAPPER, I chose to simply remove all but one marker for each position.

The `.tped` file contains no header line and $2N + 4$ columns, where N is the number of individuals. The first four columns of each row contain chromosome codes, variant identifiers, positions in centimorgans and base-pair coordinates. There are as many rows as there are variants. The variants are sorted in increasing position (both on the third and fourth field), and it is the third column that has duplicate positions.

The following command was issued to filter out duplicate positions:

```
$ sort -guk 3,3 input.tped | sort -g -k 1,1 -k 3,3 -k 4,4 > output.tped
```

Listing B.6: Command to remove duplicate centimorgan positions

The command will first filter out all unique values in the third column, and then it will sort the entire file on chromosome order, centimorgan positions and physical positions.

B.2.3 Running CLAPPER

Clapper needs an optionfile with arguments. The option file is shown in Listing B.7.

```
/path/prefix/to/tped-and-tfam-files #fileName
/path/to/age-file.txt #ageFileName
```

Listing B.7: Optionfile for CLAPPER

As can be seen, a file containing ages is also provided. This file must be white space-delimited, with three columns: Family ID, individual ID and the age of the individual.

B.2.4 Results

Unfortunately, I hit a roadblock at this point, as CLAPPER failed to execute. The program reported a NullPointerException, which was not possible for me to debug, as I have no access to the source code. I sent an email to the first author, but they did not have the opportunity to help me at the time. They did, however, recommend checking out SEQUOIA or COLONY. Because of this, CLAPPER was dropped.

B.3 Padre

PADRE is a relatedness estimation method that combines the use of PRIMUS (Staples et al., 2014) and ERSA (Huff et al., 2011), by estimating three generational pedigrees with PRIMUS, and using ERSA to estimate the degree of relationship between the founders of the pedigrees (Staples et al., 2016). PADRE has been able to detect up to 13th degree relationships, which is accomplishes by the use of multi-way estimation – to best find the relationship between two founders of a pedigree, it leverages all individiuals in both pedigrees in a composite likelihood framework.

Figure B.2 shows what programs were necessary to run PADRE. ERSA and PRIMUS are standalone programs and needs to be run beforehand, and the output must be given to PADRE.

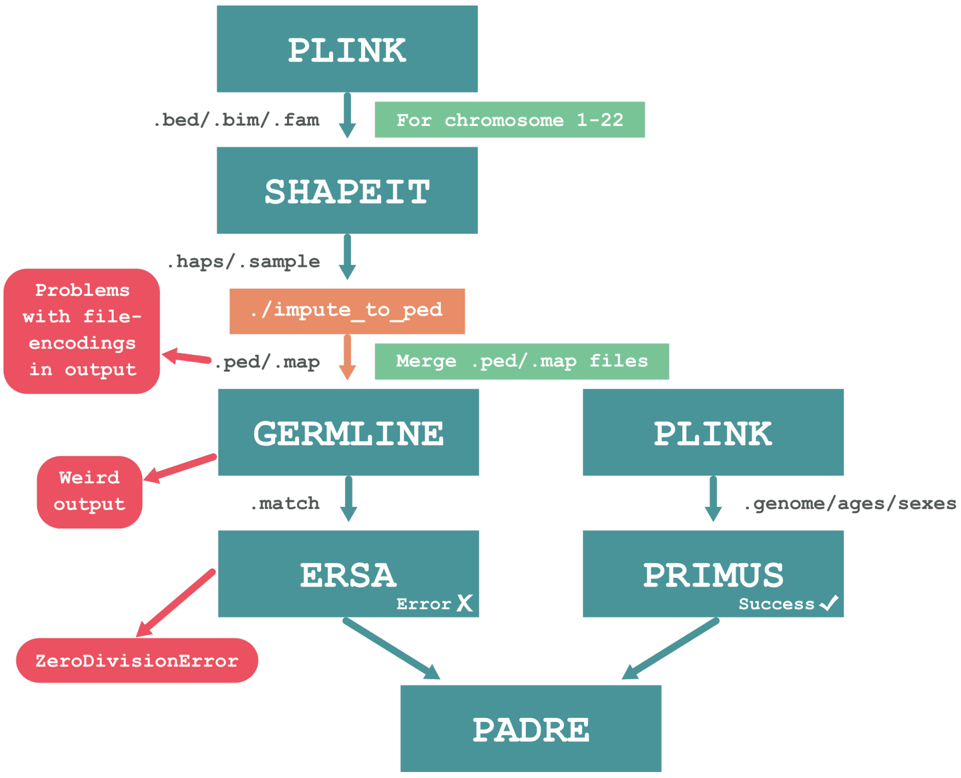


Figure B.2: The method flow of PADRE.

PRIMUS had a pretty straightforward pipeline, and executed successfully. ERSA requires more programs in the pipeline, and unfortunately, I did not succeed with running it, which led to PADRE being dropped.

This section begins with an explanation of how the data should be preprocessed before running any of the programs, and then it continues with explaining the executions of PRIMUS and ERSA – and what went wrong with the latter.

B.3.1 PLINK preprocessing

First and foremost, both SHAPEIT and PRIMUS requires input data prepped by PLINK. In addition, GERMLINE requires `.ped` and `.map` files, and has some of the same expectations of the data as CLAPPER. To ensure that the data set is the same for all programs – that it includes the same individuals and the same markers – and that the data is filtered correctly, the data should be preprocessed as shown in Listing B.8.

```
plink2 --bfile genotype-data --keep individuals-to-keep.txt --cm-map
    gmap-FIN-@.txt --mind 0.001 --geno 0.001 --maf 0.05 --snps-only
    --recode --out temp-prefix
sort -guk 3,3 output-prefix.map | sort -g -k 1,1 -k 3,3 -k 4,4 | cut -f 2
    > variants-to-extract.txt
plink2 --file temp-prefix --extract variants-to-extract --recode --out
    preprocessed
```

Listing B.8: A short script that will make `.ped` and `.map` files with positions in centimorgans, no multi-nucleotide alleles and no missing genotypes. Furthermore, it will find all variants with unique genetic positions and filter these out. The resulting files are `preprocessed.ped` and `preprocessed.map`, which should be used in further processing.

B.3.2 PRIMUS

PRIMUS (Staples et al., 2014) requires IBD estimates to be calculated by PLINK. These estimates can be aquired as shown in Listing B.3.2.

```
$ plink2 --file preprocessed --genome --out plink-ibd-estimates
```

It is also possible to provide the age and sex of the individuals in the study, which PRIMUS will use to decrease the amount of possible pedigrees. The age file must be space delimited, with the columns family id, individual id and age in that order. Similarly, the sex file must be space delimited with the columns family id, individual id and sex, where 1 = male and 2 = female. These files are given to PRIMUS with the flags `--age_file` and `--sex_file`. The command to run PRIMUS is shown in Listing B.3.2.

```
$ Run_PRIMUS.pl --p plink-ibd-estimates.genome --age_file age-file.txt
    --sex_file sex-file.txt --out primus-output-directory
```

B.3.3 ERSA

As shown in Figure B.2, ERSA takes IBD estimates from GERMLINE as input, and the data was phased with SHAPEIT before running GERMLINE. This section will begin with and explanation of how the data was phased, before it continues with a description of how to convert from SHAPEIT output to GERMLINE input, and it will complete with and explanation of how to run GERMLINE and ERSA, and what went wrong.

Phasing: SHAPEIT

SHAPEIT was used for phasing of the data, and it takes PLINK's .ped and .map format or the binary coded .bed, .bim and .fam format. In addition, it requires a genetic mapping. I ran SHAPEIT with .bed/.bim/.fam input files and genetic mappings as described in A.

Phasing with SHAPEIT can only be done on one chromosome at a time, so I made a script as shown in Listing B.9, which runs the phasing for each chromosome. This ended up taking about 67 hours on roughly 1000 individuals, and produced a .haps and .sample file for each chromosome.

```
#!/bin/bash
for chr in {1..22}
do
    # Extract chromosome $chr from the ped and map file
    plink2 --file preprocessed --chr $chr --make-bed --out
    preprocessed-$chr

    # Run shapeit on chromosome $chr
    ./shapeit --input-bed preprocessed-$chr.bed preprocessed-$chr.bim
    preprocessed-$chr.fam -M gmap-FIN-$chr.txt -O shapeit-$chr-phased

    # Remove all bed/bim/fam files for chromosome $chr
    rm preprocessed-$chr*
done
```

Listing B.9: A script that runs shapeit for chromosomes 1-22.

To be fair, I tried to use phased output from BEAGLE first, since I already had used BEAGLE to phase genotypes of all individuals in the complete data set. However, BEAGLE outputs a .vcf file with phased data, and when I converted this file to a .ped and .map file with plink, the phasing was lost because the .ped format is not intended for storing phased genotypes, as described more closely in the next section.

Converting from SHAPEIT output to GERMLINE input

GERMLINE takes .ped and .map files as input and the authors recommend phasing the data with SHAPEIT or EAGLE. As described in the last section, I used SHAPEIT for the phasing.

One thing to note is that GERMLINE uses PLINK's .ped and .map format for the input files. This is somewhat problematic, as the .ped and .map format are not meant to store phased data at all. Meaning that if you do any kind of work with these files in PLINK prior to using GERMLINE, phasing information will be lost. This makes file-handling a bit more tedious, because one has to manually change files, to e.g. extract individuals or variants. There are other filetypes, like the .vcf format, which does retain phasing information, has its own suite of tools and would be better suited for this kind of program.

To give an example, Listing B.10 shows how phased data are stored in a .vcf. More specifically, you can see that both individuals A and B have heterozygote alleles at marker 1:123:C:T, but on opposite chromosomes, indicated by "|". C is also heterozygote on the marker, but it is uncertain which chromosome has what allele, as indicated by

”\”.

```
#CHROM  POS   ID          REF ALT   QUAL  FILTER  INFO  FORMAT  A_A  B_B  C_C
1       123   1:123:C:T   C    T     .      PASS    .     GT       1|0  0|1  1\0
```

Listing B.10: An example of a `.vcf` file, where genotypes are stored as shown in the last three columns. Whether genotypes are phased or not, are indicated with a | or a \– where the former indicates phased genotypes and the latter indicates unphased genotypes.

In the corresponding `.ped` file, shown in Listing B.11, there is nothing that indicates whether genotypes has been phased or not, and if you use these files with PLINK, the alleles will be reordered – e.g. the alleles could all be coded as `CT` for `A`, `B` and `C` as seen in the listing.

```
#FID    IID    SIRE   DAM    SEX    PHENOTYPE   ALLELE1   ALLELE2
 A      A      0      0      0      -9          C         T
 B      B      0      0      0      -9          C         T
 C      C      0      0      0      -9          C         T
```

Listing B.11: An example of a .ped file, and how a pair of alleles are stored for a marker. The .ped file will simply show whether the marker is heterozygote or homozygote, but not whether it has been phased or not.

In addition, GERMLINE has many of the same expectations of the data set as CLAPPER, but there is little information about this in the documentation. More specifically, the program will complain about markers on the same position, markers not being biallelic (this might be due to missing genotypes) and markers not being SNPs (as opposed to multi-nucleotide alleles). These issues can be solved with PLINK, but it should be done before phasing (i.e. before using SHAPEIT), otherwise the issues must be solved with file handling, to not lose the phasing information. When you cannot use PLINK for this, it makes things more complicated.

Nonetheless, at this point, I still had 22 `.haps` and `.sample` files which needed to be converted to `.ped` and `.map` format. GERMLINE has provided a script that does this, namely `impute_to_ped`. This must be done for each chromosome, and the resulting `.ped` and `.map` files for all chromosomes must be merged into one `.ped` file and one `.map` file. A script that does this is shown in Listing B.12. The `impute_to_ped` script might output some characters that can cause problems, so the extra perl-command shown in lines 16 and 20 ensures that only valid characters are printed to avoid this.

```
1  #!/bin/bash
2
3  # Use impute_to_ped to convert from .haps and .sample to .ped and .map for
       each chromosome
4  for chr in {1..22}
5  do
6      ./impute_to_ped shapeit-$chr-phased.haps shapeit-$chr-phased.sample
       shapeit-$chr-phased
7
8  # Merge .map files
9  touch shapeit-phased.map
10 for chr in {1..22}
11 do
12         cat shapeit-$chr-phased.map >> shapeit-phased.map
```

```
done

# Merge .ped files
cat shapeit-1-phased.ped > shapeit-phased.ped

for chr in {2..22}
do
    cut -f 7- -d ' ' shapeit-$chr-phased.ped > temp
    paste -d ' ' shapeit-phased.ped temp > temp2
    cat temp2 > shapeit-phased.ped
done
```

Listing B.12: A script that uses GERMLINE's script, impute_to_ped to convert from haps/sample format to ped/map forimat, and merges all ped and map files into a a single ped and map file.

The last thing that had to be done with the files before running GERMLINE, was to update the `.map` file with genetic positions in cM. These positions will be lost from the SHAPEIT phasing and the `impute_to_ped` script, resulting in all zeros.

Why are these genetic positions important? If the `.map` file does not contain genetic positions, GERMLINE will output the genetic length of segments in megabases as opposed to centimorgans. ERSA will not accept GERMLINE output with the genetic length in megabases. There is little in the documentation about this, both regarding GERMLINE and ERSA. The former does not explicitly say what makes the genetic length be in centimorgans or megabases and ERSA does not inform of this requirement, other than an error message when you first run the program.

The command shown in Listing B.13 will take the `.ped` and `.map` file generated by the script above, and update the genetic positions.

```
plink2 --file shapeit-phased --cm-map gmap-FIN-@.txt --recode --out
    shapeit-phased-cm-map
```

Listing B.13: Getting genetic positions in the .map file.

The command will create the two files `shapeit-phased-cm-map.ped` and `shapeit-phased-cm-map.map`. It is important to note that only the `.map` file should be used, because the `.ped` file will have lost its phasing information. So, to be more specific, the input files to germline will be `shapeit-phased.ped` and `shapeit-phased-cm-map.map`.

Running GERMLINE and ERSA

At last, GERMLINE can be run, as shown in Listing B.14.

```
./germline -input shapeit-phased.ped shapeit-phased-cm-map.map  -output
    germline-ibd-estimates
```

Listing B.14: ZeroDivisionError in ERSA

GERMLINE completed successfully, and ERSA was executed as shown in Listing B.15.

```
python2 ersa --segment_files=germline-ibd-estimates.match --output_file=
    ersa-results
```

Listing B.15: ZeroDivisionError in ERSA

At this point, ERSA failed. More specifically with a ZeroDivisionError, as shown in Listing B.16.

```
File "ersa", line 398, in get_cm
return cm * float(end_position-begin_position/(segment_end_position-
    segment_begin_position))
```

Listing B.16: ZeroDivisionError in ERSA

The reason seems to be the GERMLINE output. More specifically, GERMLINE outputs a `.match` file, where each line contains information of pairwise shared IBD segments between two individuals. Essentially, this means that there will be several lines for each pair of individuals, one line for each shared segment. For some reason, almost all these shared segments consisted of only one marker.

”Should this be a problem?” One would ask. Well, as seen in Listing B.16, this results in a ZeroDivisionError. I am guessing that `segment_end_position` and `segment_begin_position` is taken from the fields `segment start (bp)` and `segment end (bp)` in the `.match` file, which would have the same value if the segment only contains one marker. Ultimately, I cannot be sure, since I don't have access to ERSA's source code.

To avoid this error, the code in ERSA should probably add 1 in the denominator, as a segment containing one marker should have length 1:

$$\frac{\text{end\_position} - \text{begin\_position}}{\text{segment\_end\_position} - \text{segment\_begin\_position} + 1}$$

But nonetheless, since this does result in an error, I suspect that there might be something wrong with the output in GERMLINE. My reasoning being that if it was ”normal” to have segments of length one in the `.match` file, the ZeroDivisionError would've been discovered by the authors of ERSA. I sent an email to both the authors of GERMLINE and the authors of ERSA, explaining my problem, but unfortunately I didn't get any answers from neither of them.

I had to give up on this point, and move on to another program, because continuing working with and fixing this would be outside of the scope of my thesis and too time consuming.

Appendix C

Mitodetect

This appendix contains the source code of `Mitodetect`.

C.1 Source code

Listing C.1: main.py

```python
import argparse
import pandas as pd
from MitochondriaData import MitochondriaData
from Pedigree import Pedigree
from MitoMatch import mitomatch
from variant_log_probabilities import variant_log_probabilities as vlp
from critical_value import get_critical_value,
    get_critical_value_from_kernel_density
import pickle


if __name__ == '__main__':

    parser = argparse.ArgumentParser(description="yaya")
    parser.add_argument('mitochondria', metavar='ped/map-prefix', type=str
    , nargs=1,
        help="Path prefix to mitochondrial DNA in PLINK's ped/map format."
    )
    parser.add_argument('output', metavar='output-path', type=str, nargs
    =1,
        help="Path to output")
    parser.add_argument('--pedigree', metavar='Sequoia-pedigree', type=str
    , nargs=1,
        help="Pedigree output from sequoia.")
    parser.add_argument('--individuals', metavar='individuals.txt', type=
    str, nargs=1,
        help="The list of IDs to the individuals whom should be included
    in the estimation.")
```

```python
parser.add_argument('--alpha', metavar='float', type=float, nargs=1,
    help="The value for when to accept whether two individuals are mitochondrial relatives or not.")
parser.add_argument('--hamming', action='store_true',
    help="Whether to calculate a hamming distance, as opposed to a weighted distance.")
parser.add_argument('--p_value_file', type=str, nargs=1,
    help="A file with distributions to use for estimating p-values. \n \
        If this is not provided, a pickled density function will be used, \n\
            based on a distribution of weighted/hamming distances between \n\
                father/child pairs where father and mother have a degree of relatedness of 7 or above..")

args = parser.parse_args()

PEDIGREE_FILE = args.pedigree[0] if args.pedigree else False
INDIVIDUALS_FILE = args.individuals[0] if args.individuals else False
P_VALUE_FILE = args.p_value_file[0] if args.p_value_file else False
PEDMAP_PREFIX = args.mitochondria[0]
OUTPUT_PATH = args.output[0]
ALPHA =  args.alpha[0] if args.alpha else 0.05
HAMMING = args.hamming

if not PEDIGREE_FILE and not INDIVIDUALS_FILE:
    print("You must either specify the path to a sequoia output-pedigree or a list of individuals.")
    exit()

print("\nRunning MitoDetect.")
print("\nOptions:\n")

print("Ped/map file prefix: ", PEDMAP_PREFIX, sep='\t')
print("Pedigree-filepath:   ", PEDIGREE_FILE, sep='\t') if PEDIGREE_FILE else None
print("Individuals filepath:", INDIVIDUALS_FILE, sep='\t') if INDIVIDUALS_FILE else None
print("Output path:         ", OUTPUT_PATH, sep='\t')
print("Alpha:               ", ALPHA, sep='\t')
print("Distance type:       ", "weighted (default)" if not HAMMING else "Hamming", sep='\t')
print("p-values from:       ", P_VALUE_FILE) if P_VALUE_FILE else None

if PEDIGREE_FILE:
    pedigree = Pedigree(PEDIGREE_FILE)
    individuals = pedigree.founders.id.str.split('__', 1, expand=True)
    individuals.columns = ["FID", "IID"]
    print("Number of founders:", len(pedigree.founders))
else:
    individuals = pd.read_csv(INDIVIDUALS_FILE, sep='\s+', header=None, usecols=[0, 1])
    individuals.columns = ['FID', 'IID']

print("\n> Reading ped/map files")
mito_data = MitochondriaData(PEDMAP_PREFIX)
```

```python

    print("> Running mitomatch\n")
    if HAMMING:
        distances = mitomatch(mito_data, individuals=individuals.IID, calculation_type="hamming" )
    else:
        distances = mitomatch(mito_data, individuals=individuals.IID, calculation_type="weighted", weights=vlp )

    if P_VALUE_FILE:
        trio_path = P_VALUE_FILE
        trio_distances = pd.read_csv(trio_path, header=0, sep='\t')

        df = trio_distances.loc[trio_distances.relationship == 'father/child', 'distance' ]
        bandwidth = 5 if not HAMMING else 1
        critical_value = get_critical_value(df, bandwidth=bandwidth, alpha=ALPHA)
    else:
        pickled_kernel_density = 'data/father-child-log-prob-weighted-kernel-density-druid-above-7.pickle'
        start, end = 0, 300
        if HAMMING:
            pickled_kernel_density = 'data/father-child-hamming-kernel-density-druid-above-7.pickle'
            start, end = 0, 30

        with open(pickled_kernel_density, 'rb') as file:
            kde = pickle.load(file)

        critical_value = get_critical_value_from_kernel_density(kde, start=start, end=end, alpha=ALPHA)

    print("\n> Running hypothesis tests, with alpha = {} and critical value = {:.3f}.".format( ALPHA, critical_value ))
    decisions = []

    for row in distances.itertuples():
        if row.distance <= critical_value:
            decisions.append(True)
        else:
            decisions.append(False)

    distances.loc[decisions, 'mito_relatives'] = True
    distances.mito_relatives = distances.mito_relatives.fillna(False)
    print("Number of mitochondrial relatives:", len(distances.loc[distances.mito_relatives.isin( [True] )]))
    distances.to_csv(OUTPUT_PATH, index=False, sep='\t')
```

Listing C.2: MitoMatch.py

```python
import pandas as pd
import numpy as np
import time
from functools import reduce
import warnings

```

```python

def mitomatch(mito_data, individuals=None, filter_out_variants=True, calculation_type='hamming', weights=None, ignore_missing=True, verbose=True):
    """ mitomatch will compare mitochondrial DNA between all pairs of individuals
    in mito_data or those defined individuals. It can calculate the hamming-distance,
    a weighted distance or a true/false vector for all variants.

    Arguments:
        mito_data {MitochondriaData} -- An instance of MitochondriaData.

    Keyword Arguments:
        individuals -- A list of individuals' IID's (default: {None})
        filter_out_variants {bool} -- If this keyword is set to true, all
        variants shared by all individuals in the set will be filtered away
        to improve execution time. (default: {True})
        calculation_type {str} -- The type of calculation to be specified.
        Choose between a hamming, weighted or true-false-vector (default: {'hamming'})
        weights {dict} -- A dictionary with weights for all variant_ids (default: {None})

    Returns:
        [pandas.Dataframe] -- A dataframe that consists of all pairs of ids, and
        the result for each pair of individuals in the subsequent columns.
    """
    a = time.time() # Total execution timing

    # Timing variables
    access, calculation, store, for_loop = 0, 0, 0, 0

    # Variables necessary for calculation
    variant_ids = mito_data.map_df.variant_id.values
    columns = ["id1", "id2", "distance"]

    # Determine the calculation_type and set calculate_func accordingly
    if calculation_type == 'hamming':
        if ignore_missing:
            def hamming_distance(mito_dna1, mito_dna2):
                return [ reduce(
                    lambda total, x: total + int(x[0] != x[1]) if 0 not in (x[0], x[1]) else total,
                    zip(mito_dna1, mito_dna2),
                    0 ) ]
        else:
            raise NotImplementedError()

        calculate_func = hamming_distance

    elif calculation_type == 'weighted':
        if weights is None:
            raise ValueError("weights are needed if calculation_type='weighted'.")

        if ignore_missing:
            def weighted_distance(mito_dna1, mito_dna2):
                return [ reduce(
                    lambda total, x: total + weights[x[0]] if x[1] != x[2] and 0 not in (x[1], x[2]) else total + 0,
                    zip(variant_ids, mito_dna1, mito_dna2),
                    0 ) ]
        else:
            raise NotImplementedError()

        calculate_func = weighted_distance

    elif calculation_type == 'true-false-vector':
        if not weights is None:
            warnings.warn("Weights will not be applied when True/False vectors are calculated.")

        if ignore_missing:
            def true_false_vector(mito_dna1, mito_dna2):
                return [x == y if 0 not in (x, y) else True for x,y in zip(mito_dna1, mito_dna2)]
        else:
            def true_false_vector(mito_dna1, mito_dna2):
                return np.equal(mito_dna1, mito_dna2).tolist()

        calculate_func = true_false_vector
        columns = ["id1", "id2"] + variant_ids.tolist()
    else:
        raise ValueError("Unknown calculation_type: {}\n\
            Choose between 'hamming', 'weighted' and 'true-false-vector'".format(calculation_type))

    print("Using calculation_type:", calculation_type) if verbose else None

    # Ready up data set
    mito_data.set_subset(individuals) if not individuals is None else None
    mito_data.filter_out_variants() if filter_out_variants and calculation_type != 'true-false-vector' else None

    # Output
    matrix = []

    # Progress variables
    n = len(mito_data)
    tenth = n // 10

    # Do the calculations!! :D
    x = time.time() # For loop timing
    for index1, row1 in mito_data.ped_df.iterrows():
        y = time.time()
        for_loop += (y - x)

        if tenth > 0 and verbose:   # Show progress
            print("... {} individuals remaining.".format(n - index1)) if index1 % tenth == 0 else None

        x = time.time() # Access timing
        mito_dna1 = row1.iloc[mito_data.genotype_start_index:]
        id1 = row1.values[1]
        y = time.time()
        access += (y - x)

        x = time.time() # For loop timing
        for index2, row2 in mito_data.ped_df.iloc[index1 + 1:].iterrows():
            y = time.time()
            for_loop += (y - x)

            x = time.time() # Access timing
            mito_dna2 = row2.iloc[mito_data.genotype_start_index:]
            id2 = row2.values[1]
            y = time.time()
            access += (y - x)

            x = time.time() # Calculation timing
            comparison = calculate_func(mito_dna1, mito_dna2)
            matrix.append( [id1, id2] + comparison )
            y = time.time()

            calculation += (y - x)

            x = time.time() # For loop timing
        x = time.time() # For loop timing

    pairwise_thing = pd.DataFrame(matrix)
    pairwise_thing.columns = columns

    b = time.time()
    total = b - a

    # Timing calculation and printing
    access_ratio = access / total
    calculation_ratio = calculation / total
    for_loop_ratio = for_loop / total
    store_ratio = store / total

    print( "Total time spent: {} seconds\n\
        Access: {}%\n\
        Calculation: {}%\n\
        For-loop: {}%\n\
        Store: {}%".format(total, access_ratio*100, calculation_ratio*100,
            for_loop_ratio*100, store_ratio*100 )) if verbose else None

    mito_data.unset_subset()
    return pairwise_thing
```

Listing C.3: MitochondriaData.py

```
import pandas as pd
import numpy as np
import warnings
import re

```

```python
class MitochondriaData:

    def __init__(self, file_path_prefix, individuals=None):
        self.genotype_start_index = 6
        self.shared_genotypes = None

        self.map_df = self.open_map(file_path_prefix + '.map')
        self.ped_df_complete = self.open_and_clean_ped(file_path_prefix + '.ped')
        self.ped_df = self.ped_df_complete

        self.num_variants = len(self.ped_df.columns) / 2 - self.genotype_start_index

        self.subset_active = True if individuals != None else False
        self.set_subset(individuals) if individuals != None else None


    def open_map(self, file_path):
        try:
            map_df = pd.read_csv(file_path,
                sep='\s+',
                header=None,
                names=['chr', 'variant_id', 'genomic_pos', 'physical_pos'],
                dtype={'chr': str, 'variant_id': str, 'genomic_pos': np.float64, 'physical_pos': np.int64},
                )

        except FileNotFoundError as e:
            self.print_file_not_found_message(e)
            exit()

        return map_df


    def open_and_clean_ped(self, file_path):
        try:
            ped_df = pd.read_csv(file_path, sep='\s+', header=None, dtype=str)

        except FileNotFoundError as e:
            self.print_file_not_found_message(e)
            exit()

        assert len(self.map_df.index) * 2 + 6 == len(ped_df.columns), \
            "The number of columns in the ped-file does not match the number of variants.\n\
                There should be 6 fixed fields + 2v fields where v is the number of variants\
                (corresponding to the number of rows in the map-file).\n\
                See https://www.cog-genomics.org/plink2/formats#ped for more about the ped-file format."

        ped_df = self.clean_ped_file(ped_df)
        ped_df.columns = self.create_ped_column_names()

        ped_df.sex = pd.to_numeric(ped_df.sex).astype(np.uint8)
        ped_df = self.convert_ped(ped_df)

        return ped_df


    def create_ped_column_names(self):
        # A ped file should contain 6 fixed fields (as seen in names, below) + 2v fields where v is
        # the number of variants. The number of variants is extracted from the map file.
        # See https://www.cog-genomics.org/plink2/formats#ped for more about the format.
        names = ['FID', 'IID', 'sire', 'dam', 'sex', 'phenotype'][:self.genotype_start_index]
        variants = self.map_df.variant_id.values.tolist()

        return names + variants


    def clean_ped_file(self, ped_df):
        variants_to_keep = ped_df.iloc[:, self.genotype_start_index::2]

        self.genotype_start_index -= 1 # We don't need phenotype column
        fixed_fields = ped_df.iloc[:, :self.genotype_start_index]

        return pd.concat( [fixed_fields, variants_to_keep], axis=1 )


    def convert_ped(self, ped_df):
        x = self.genotype_start_index
        ped_df.iloc[:, x:] = ped_df.iloc[:, x:].applymap( MitochondriaData.allele_map ).astype(np.uint8)
        return ped_df


    def get_mtdna(self, i_index):
        return self.ped_df.iloc[i_index, self.genotype_start_index:].values


    def get_id(self, i_index):
        return self.ped_df.iloc[i_index, 1]


    def set_subset(self, individuals):
        self.ped_df = self.ped_df_complete.loc[ self.ped_df_complete.IID.isin(individuals) ].reset_index(drop=True)
        self.subset_active = True
        assert len(self.ped_df) <= len(individuals)


    def unset_subset(self):
        self.subset_active = False
        self.ped_df = self.ped_df_complete


    def filter_out_variants(self):
        if not self.subset_active:
            warnings.warn("You are currently filtering out variants on the complete dataset. The full dataset is still retained in ped_df_complete.")

        columns_to_drop = []
        for column in self.ped_df.iloc[:, self.genotype_start_index: ]:
            if len(set(self.ped_df[column])) == 1:
                columns_to_drop.append(column)

        self.ped_df = self.ped_df.drop(columns=columns_to_drop)
        self.shared_genotypes = len(columns_to_drop)
        print("... {} variants are shared by all individuals.".format(len(columns_to_drop)))



    def print_file_not_found_message(self, e):
        print("The file(s) was not found.\n",
            "Ensure that only the prefix to the .ped and .map file is provided, ",
            "and that they have the same name.", sep=’’)


    def __len__(self):
        return len(self.ped_df)


    @staticmethod
    def allele_map(x):
        mapping = { ’0’: 0, ’A’: 1, ’C’: 2, ’G’: 3, ’T’: 4 }
        return mapping[x]
```

Listing C.4: Pedigree.py

```
import pandas as pd
import numpy as np

class Pedigree:
    dtypes = {
        "id": str,
        "dam": str,
        "sire": str,
        "LLRdam": np.float64,
        "LLRsire": np.float64,
        "LLRpair": np.float64
    }

    def __init__(self, pedigree_file_path, sequoia_specs_path=None, iterations=10, debug=True):
        self.debug = debug
        self.pedigree = pd.read_csv(
            pedigree_file_path,
            sep=’\s+’,
            header=0,
            dtype=self.dtypes
```

```python
            )
        self.specs = self.read_specs(sequoia_specs_path) if sequoia_specs_path else None

        self.find_dummies()
        self.find_missing_link_mums()
        self.find_founders()


    def read_specs(self, path):
        specs = {}
        with open(path, 'r') as file:
            for line in file:
                key, value = line.split(",")
                specs[ key.strip() ] = value.strip()
        return specs

    def find_dummies(self):
        self.dummies = self.pedigree.loc[ self.pedigree.id.str.contains("^M|F[0-9]+") ].copy()
        print("Number of dummies: ", len(self.dummies)) if self.debug else None

    def find_missing_link_mums(self):
        self.missing_link_mums = self.dummies.loc[ self.dummies['dam'].notnull() ].copy()
        print("Number of missing-link-mums: ", len(self.missing_link_mums)) if self.debug else None

    def find_founders(self):
        """ An individual is not a founder if:
        1. The individual is a dummy
        2. The individual has a mum which is not a dummy
        2. The individuals dummy-mum has a mum

        Basically, drop individual if:

        is_dummy or has_mum and (not mum_is_dummy or dummy_has_mum)
        """
        drop = self.pedigree.loc[ self.pedigree['id'].isin(self.dummies.id)
                | self.pedigree['dam'].notnull() & (
                    ~self.pedigree['dam'].isin(self.dummies.id)
                    | self.pedigree['dam'].isin(self.missing_link_mums.id)
                )
            ]

        self.founders = self.pedigree.drop(index=drop.index).copy()
```

C.2 Tests

Unit tests were written for the two classes `Pedigree` and `MitochondriaData` as well as the function, `mitomatch`.

Listing C.5: test_mito_match.py

```python
import unittest
from unittest.mock import MagicMock
from unittest.mock import patch
import pandas
#from MitochondriaData import MitochondriaData
from MitoMatch import mitomatch

class TestMain(unittest.TestCase):

    def setUp(self):
        self.mito_data = MagicMock()
        self.mito_data.map_df = self.valid_map()
        self.mito_data.genotype_start_index = 5

    def test_output_has_correct_individuals_with_no_duplicates(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 1, 2],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )

        valid_pairs = [
            ('a', 'b'), ('b', 'a'),
            ('a', 'c'), ('c', 'a'),
            ('b', 'c'), ('c', 'b')
            ]

        results = mitomatch(self.mito_data, 'dontcare')

        for index, row in results.iterrows():
            pair = (row['id1'], row['id2'])
            self.assertIn(pair, valid_pairs)

        self.assertEqual(len(results), 3)

    def test_mito_match_hamming_distance_without_filter_out_variants(self)
    :
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 1, 2],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )

        expected_1 = ['a', 'b', 2]
        expected_2 = ['a', 'c', 2]
        expected_3 = ['b', 'c', 1]
        results = mitomatch(self.mito_data, 'dontcare', calculation_type='
    hamming')

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())

    def test_mito_match_hamming_distance_with_filter_out_variants(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 4],
            ['b', 'b', '0', '0', 2, 3, 2],
            ['c', 'c', '0','0', 2, 4, 2]]
        )

        expected_1 = ['a', 'b', 2]
        expected_2 = ['a', 'c', 2]
        expected_3 = ['b', 'c', 1]
        results = mitomatch(self.mito_data, 'dontcare', filter_out_variants=True, calculation_type='hamming')

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())

    def test_mito_match_hamming_distance_adds_0_when_ignores_missing_is_true(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 0, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 1, 2],
            ['c', 'c', '0','0', 2, 4, 1, 0]]
        )

        expected_1 = ['a', 'b', 1]
        expected_2 = ['a', 'c', 0]
        expected_3 = ['b', 'c', 1]
        results = mitomatch(self.mito_data, 'dontcare', calculation_type='hamming', ignore_missing=True)

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())


    def test_weighted_distance_adds_0_when_ignores_missing_is_true(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 0, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 1, 2],
            ['c', 'c', '0','0', 2, 4, 1, 0]]
        )
        # {'2010-08-MT-841': 0.5, '2010-08-MT-981': 0.25, '2010-08-MT-550': 0.75}
        expected_1 = ['a', 'b', 0 + 0 + 0.75]
        expected_2 = ['a', 'c', 0 + 0 + 0]
        expected_3 = ['b', 'c', 0.5 + 0 + 0]
        results = mitomatch(self.mito_data, 'dontcare', calculation_type='weighted', ignore_missing=True, weights=self.get_weights())

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())

    def test_true_false_vector_sets_true_when_ignores_missing_is_true(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 0, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 1, 2],
            ['c', 'c', '0','0', 2, 4, 1, 0]]
        )

        expected_1 = ['a', 'b', True, True, False]
        expected_2 = ['a', 'c', True, True, True]
        expected_3 = ['b', 'c', False, True, True]
        results = mitomatch(self.mito_data, 'dontcare', calculation_type='true-false-vector', ignore_missing=True)

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())


    def test_mito_match_weighted_distance_without_filter_out_variants(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 2, 2],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )
        self.mito_data.ped_df.columns = ['FID', 'IID', 'dc', 'dc', 'dc',
            '2010-08-MT-841', '2010-08-MT-981', '2010-08-MT-550']

        expected_1 = ['a', 'b', 0.5 + 0.25 + 0.75]
        expected_2 = ['a', 'c', 0.5 + 0 + 0.75]
        expected_3 = ['b', 'c', 0.5 + 0.25 + 0]
        results = mitomatch(self.mito_data, 'dontcare', weights=self.get_weights(), calculation_type='weighted')

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())

    def test_mito_match_weighted_distance_without_weights_raises_error(self):
        with self.assertRaises(ValueError) as cm:
            results = mitomatch(self.mito_data, 'dontcare', weights=self.get_weights(), calculation_type='weighted')
            self.assertEqual(cm.exception.code, 1)

    def test_equal_mitochondria_gives_distance_zero(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 2, 1, 4],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )
        self.mito_data.ped_df.columns = ['FID', 'IID', 'dc', 'dc', 'dc',
            '2010-08-MT-841', '2010-08-MT-981', '2010-08-MT-550']

        expected = ['a', 'b', 0]

        results = mitomatch(self.mito_data, 'dontcare', calculation_type='hamming')
        self.assertListEqual(expected, results.iloc[0].values.tolist(), "Equal mtdna doesn't give distance zero (unweighted)")

        results = mitomatch(self.mito_data, 'dontcare', weights=self.get_weights(), calculation_type='weighted')
        self.assertListEqual(expected, results.iloc[0].values.tolist(), "Equal mtdna doesn't give distance zero (weighted)")

    def test_true_false_vector_gives_right_output(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 2, 2],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )

        expected_1 = ['a', 'b', False, False, False ]
        expected_2 = ['a', 'c', False, True, False ]
        expected_3 = ['b', 'c', False, False, True ]

        results = mitomatch(self.mito_data, 'dontcare', calculation_type='true-false-vector')

        self.assertListEqual(expected_1, results.iloc[0].values.tolist())
        self.assertListEqual(expected_2, results.iloc[1].values.tolist())
        self.assertListEqual(expected_3, results.iloc[2].values.tolist())

    def test_assert_warning_when_true_false_vector_and_weights_are_provided(self):
        self.mito_data.ped_df = pandas.DataFrame(
            [['a', 'a', '0', '0', 2, 2, 1, 4],
            ['b', 'b', '0', '0', 2, 3, 2, 2],
            ['c', 'c', '0','0', 2, 4, 1, 2]]
        )

        with self.assertWarns(Warning):
            results = mitomatch(self.mito_data, 'dontcare', calculation_type='true-false-vector', weights=self.get_weights())

    def valid_map(self):
        valid_map = pandas.DataFrame(
            [['26', '2010-08-MT-841', 100, 72],
            ['26', '2010-08-MT-981', 100, 93],
            ['26', '2010-08-MT-550', 100, 215]],
            columns=['chr', 'variant_id', 'genomic_pos', 'physical_pos'],
            )
        return valid_map

    def get_weights(self):
        return {'2010-08-MT-841': 0.5, '2010-08-MT-981': 0.25, '2010-08-MT-550': 0.75}

    def individuals(self):
        return ['a', 'b', 'c']

if __name__ == '__main__':
    unittest.main()
```

Listing C.6: test_mitochondria_data.py

```python
import unittest
from unittest.mock import patch
from MitochondriaData import MitochondriaData
import pandas
import numpy as np


class TestMitochondriaData(unittest.TestCase):

    @patch('pandas.read_csv')
    def test_ped_file_not_found_exits(self, mock_read_csv):
        mock_read_csv.side_effect = [None, FileNotFoundError]

        with self.assertRaises(SystemExit) as cm:
            mito_data = MitochondriaData("dontcare")
            self.assertEqual(cm.exception.code, 1)

    @patch('pandas.read_csv')
    def test_map_file_not_found_exits(self, mock_read_csv):
        mock_read_csv.side_effect = [FileNotFoundError, None]

        with self.assertRaises(SystemExit) as cm:
            mito_data = MitochondriaData("dontcare")
            self.assertEqual(cm.exception.code, 1)

    @patch('pandas.read_csv')
    def test_map_and_ped_not_compatible_assertion(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped().
    iloc[:, :-2]]
        with self.assertRaises(AssertionError):
            mito_data = MitochondriaData("dontcare")


    @patch('pandas.read_csv')
    def test_correct_ped_column_names(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        expected_names = ['FID', 'IID', 'sire', 'dam', 'sex', '2010-08-MT
    -841', '2010-08-MT-981', '2010-08-MT-550']

        mito_data = MitochondriaData("dontcare")

        self.assertListEqual( expected_names, list(mito_data.ped_df.
    columns) )

    @patch('pandas.read_csv')
    def test_correct_converting(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        expected_data = [['a', 'a', '0', '0', 2, 2, 1, 4],
                         ['b', 'b', '0', '0', 2, 3, 1, 2]]
        mito_data = MitochondriaData("dontcare")
        self.assertListEqual( expected_data, mito_data.ped_df.values.
    tolist() )
        self.assertEqual( np.uint8, mito_data.ped_df.iloc[:, -1].dtype ) #
     last column is genotype
        self.assertEqual( np.uint8, mito_data.ped_df.iloc[:, 4].dtype ) #
    sex column

    @patch('pandas.read_csv')
    def test_giving_individuals_to_constructor_subsets_properly(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        mito_data = MitochondriaData("dontcare", individuals=['a'])
        expected_data = [['a', 'a', '0', '0', 2, 2, 1, 4]]

        self.assertListEqual(expected_data, mito_data.ped_df.values.tolist())
        self.assertEqual(mito_data.subset_active, True)

    @patch('pandas.read_csv')
    def test_set_and_unset_subset_works_properly(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        mito_data = MitochondriaData("dontcare")
        expected_data = [['a', 'a', '0', '0', 2, 2, 1, 4]]

        mito_data.set_subset(['a'])
        self.assertListEqual(expected_data, mito_data.ped_df.values.tolist(), "Set subset fails")
        self.assertEqual(mito_data.subset_active, True)
        self.assertEqual(len(mito_data), 1)

        expected_data = [['a', 'a', '0', '0', 2, 2, 1, 4],
                         ['b', 'b', '0', '0', 2, 3, 1, 2]]

        mito_data.unset_subset()
        self.assertListEqual(expected_data, mito_data.ped_df.values.tolist(), "Unset subset fails")
        self.assertEqual(mito_data.subset_active, False)
        self.assertEqual(len(mito_data), 2)


    @patch('pandas.read_csv')
    def test_get_id_and_mitdna_from_index_works_properly(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        mito_data = MitochondriaData("dontcare")

        expected_mtdna_complete_set = [2, 1, 4]
        expected_id_complete_set = 'a'

        expected_mtdna_subset = [3, 1, 2]
        expected_id_subset = 'b'

        self.assertEqual( mito_data.get_id(0), expected_id_complete_set, "ID fails on complete set" )
        self.assertListEqual( mito_data.get_mtdna(0).tolist(), expected_mtdna_complete_set, "Mtdna fails on complete set" )

        mito_data.set_subset(['b'])

        self.assertEqual( mito_data.get_id(0), expected_id_subset, "ID fails on subset" )
        self.assertListEqual( mito_data.get_mtdna(0).tolist(), expected_mtdna_subset, "Mtdna fails on subset" )

        mito_data.unset_subset()

        self.assertEqual( mito_data.get_id(0), expected_id_complete_set, "ID fails after unsetting subset" )
        self.assertListEqual( mito_data.get_mtdna(0).tolist(), expected_mtdna_complete_set, "Mtdna fails after unsetting subset" )


    @patch('pandas.read_csv')
    def test_filter_out_variants_where_all_individuals_share_genotypes(self, mock_read_csv):
        extra_ind = ['c', 'c', '0', '0', '2', '-9', 'T', 'T', 'A', 'A', 'C', 'C']
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped(extra_ind) ]
        mito_data = MitochondriaData("dontcare", individuals=['a', 'b'])

        expected_data = [['a', 'a', '0', '0', 2, 2, 4],
                         ['b', 'b', '0', '0', 2, 3, 2]]

        self.assertEqual( mito_data.shared_genotypes, None )

        mito_data.filter_out_variants()
        self.assertListEqual( mito_data.ped_df.values.tolist(), expected_data, "Filter fails on ped_df" )
        self.assertListEqual( mito_data.get_mtdna(0).tolist(), [2, 4], "Filter fails on get_mtdna")
        self.assertEqual( mito_data.shared_genotypes, 1, "Filter doesn't show correct amount of shared genotypes" )


    @patch('pandas.read_csv')
    def test_assert_filter_out_variants_raises_warning_if_not_subset(self, mock_read_csv):
        mock_read_csv.side_effect = [self.valid_map(), self.valid_ped()]
        mito_data = MitochondriaData("dontcare")

        with self.assertWarns(Warning):
            mito_data.filter_out_variants()

    def valid_map(self):
        valid_map = pandas.DataFrame(
            [['26', '2010-08-MT-841', 100, 72],
            ['26', '2010-08-MT-981', 100, 93],
            ['26', '2010-08-MT-550', 100, 215]],
            columns=['chr', 'variant_id', 'genomic_pos', 'physical_pos'],
            )
        return valid_map

    def valid_ped(self, extra_ind=None):
        temp = [['a', 'a', '0', '0', '2', '-9', 'C', 'C', 'A', 'A', 'T', 'T'],
            ['b', 'b', '0', '0', '2', '-9', 'G', 'G', 'A', 'A', 'C', 'C']]
        temp.append(extra_ind) if not extra_ind is None else None

        valid_ped = pandas.DataFrame(
            temp,
            dtype=str
            )

        return valid_ped



if __name__ == '__main__':
    unittest.main()
```

Listing C.7: test_pedigree.py

```
import unittest
from unittest.mock import patch, mock_open
from Pedigree import Pedigree
import pandas

class TestPedigree(unittest.TestCase):

    @patch('pandas.read_csv')
    def test_read_specs(self, mock_read_csv):
        read_data = "Genofile        ,        Geno.txt\n\
                    LHfile   ,        LifeHist.txt\n\
                    nIndLH   ,        998"

        with patch("builtins.open", mock_open(read_data=read_data)) as m:
            pedigree = Pedigree("dontcare", "specs", debug=False)

        assert len(pedigree.specs) == 3
        assert pedigree.specs['nIndLH'] == '998'


    @patch('pandas.read_csv')
    def test_finds_dummies_correctly(self, mock_read_csv):
        dummies = ['F000', 'M003', 'F0100', 'M0']
        mock_read_csv.return_value = pandas.DataFrame(
            self.make_data(id=['1','2','4','5'] + dummies)
            )
        pedigree = Pedigree("dontcare", debug=False)
        assert all( [dummie in set(pedigree.dummies.id) for dummie in
    dummies] )


    @patch('pandas.read_csv')
    def test_finds_missing_link_moms_correctly(self, mock_read_csv):
        ids = ['1', '2', 'F0']
        dams = ['F0', None, '2']
        mock_read_csv.return_value = pandas.DataFrame(
            self.make_data(id=ids, dam=dams)
        )

        pedigree = Pedigree("dontcare", debug=False)

        assert len(pedigree.missing_link_mums) == 1
        assert "F0" in set(pedigree.missing_link_mums.id)

    @patch('pandas.read_csv')
    def test_finds_founders_no_dummies(self, mock_read_csv):
        mock_read_csv.return_value = pandas.DataFrame(
            self.make_data(id=["1","2","3","4","5"], dam=["2","3",None,"5"
    ,None])
        )

        pedigree = Pedigree("dontcare", debug=False)
        assert len(pedigree.founders) == 2
        assert all( [founder in set(pedigree.founders.id) for founder in [
    "3","5"]] )


    @patch('pandas.read_csv')
    def test_finds_founders_keeps_individuals_with_dummy_mums(self,
    mock_read_csv):
        dummy = "F3"
        mock_read_csv.return_value = pandas.DataFrame(
            self.make_data(
                id= ["1", "2", "F3",  "4", "5"],
                dam=["2", "F3", None, "5", None])
        )

        pedigree = Pedigree("dontcare", debug=False)
        assert len(pedigree.founders) == 2
        assert all( [founder in set(pedigree.founders.id) for founder in [
    "2","5"]] )


    @patch('pandas.read_csv')
    def
    test_finds_founders_drops_individuals_with_grandparent_and_missing_mum
    (self, mock_read_csv):
        mock_read_csv.return_value = pandas.DataFrame(
            self.make_data(
                id= ["1", "F2", "3"],
                dam=["F2", "3", None])
        )

        pedigree = Pedigree("dontcare", debug=False)
        assert len(pedigree.founders) == 1
        assert "3" in set(pedigree.founders.id)


    def make_data(self, id=None, dam=None, sire=None, LLRdam=None, LLRsire
    =None, LLRpair=None):
        data = {}

        args = (id, dam, sire, LLRdam, LLRsire, LLRpair)
        for arg in args:
            if arg != None:
                n = len(arg)

        data["id"] = id if id != None else [-1]*n
        data["dam"] = dam if dam != None else [-1]*n
        data["sire"] = sire if sire != None else [-1]*n
        data["LLRdam"] = LLRdam if LLRdam != None else [-1]*n
        data["LLRsire"] = LLRsire if LLRsire != None else [-1]*n
        data["LLRpair"] = LLRpair if LLRpair != None else [-1]*n

        return data


if __name__ == '__main__':
    unittest.main()
```

Appendix D

Scripts

Listing D.1: mitochondria-variant-probabilities.py

```python
import pandas as pd
import numpy as np

DIFFERENCES_PATH_PREFIX = "/path/to/pairwise-mtdna-differences-subset-1500-10-"

if __name__=='__main__':
    counted_differences_all_subsets = pd.DataFrame()

    for subset in range(1, 11):
        print("> Subset {}".format(subset))
        mitochondria_differences = pd.read_csv(
            DIFFERENCES_PATH_PREFIX + str(subset) + '.txt',
            header=0, sep='\t')

        # We don't need ids
        mitochondria_differences = mitochondria_differences.iloc[:, 2:]

        # Count number of different values
        counted_differences = mitochondria_differences.apply(
            lambda column: column.value_counts()
        ).fillna(0).loc[False]

        counted_differences_all_subsets = counted_differences_all_subsets.append(
            counted_differences)[counted_differences.index]

    counted_differences_all_subsets.index = range(1,11)

    # Calculate probabilities for subsets and average probabilities
    n = 1500*1499 / 2
    probabilities_all_subsets = counted_differences_all_subsets.apply(lambda row: (row + 1) / n, axis=1 )
    average_probabilities = probabilities_all_subsets.mean()

    average_probabilities.to_csv("/path/to/variant_probabilities.txt")
```

Listing D.2: compare-mitochondria.py

```python
from MitoMatch import mitomatch
from MitochondriaData import MitochondriaData
import pandas as pd
import variant_probabilities

ID_PATH_PREFIX = "/path/to/ids/subset-1500-10-"
DISTANCES_PATH_PREFIX = "/path/to/pairwise-hamming-distance-subset
    -1500-10-"
WEIGHTED_DISTANCES_PATH_PREFIX = "/path/to/pairwise-weighted-distance-
    subset-1500-10-"
DIFFERENCES_PATH_PREFIX = "/path/to/pairwise-mtdna-differences-subset
    -1500-10-"

if __name__=='__main__':
    print("> Reading mitochondria data")
    mito_data = MitochondriaData("/prefix/to/mtdna-ped-and-map")

    print("> Calculating distance for subsets")
    for subset in range(1, 11):
        print("> Subset {}".format(subset))
        ids = pd.read_csv(ID_PATH_PREFIX + str(subset),
            header=None,
            names=["FID", "IID", "ids_beagle_format"],
            sep='\s+')

        # Calculate hamming distances
        distances = mitomatch(mito_data,
            individuals=ids.IID,
            calculation_type='hamming')
        distances.to_csv(DISTANCES_PATH_PREFIX + str(subset) + ".txt",
            index=False, sep='\t')
        del distances

        # Calculate weighted distances
        weighted_distances = mitomatch(mito_data,
            individuals=ids.IID,
            calculation_type='weighted',
            weights=variant_probabilities.variant_probabilities)
        weighted_distances.to_csv(WEIGHTED_DISTANCES_PATH_PREFIX,
            index=False, sep='\t'
            )
        del weighted_distances

        # Calculate true/false vectors
        differences = mitomatch(mito_data,
            individuals=ids.IID,
            calculation_type='true-false-vector')
        differences.to_csv(DIFFERENCES_PATH_PREFIX + str(subset) + ".txt",
            index=False, sep='\t')
        del differences
```

Listing D.3: get-random-subsets.py

```python
import pandas as pd
import random

if __name__ == '__main__':
    num_sets = 10
    num_ind = 1500

    ids = pd.read_csv('/path/to/individual/ids/in/mtdna-set.inds', header=None, names=['IID', 'FID'], dtype=str, sep=' ')
    ids_not_inbred = pd.read_csv('/path/to/individuals/with/low/inbreeding.inds', header=None, names=['IID', 'FID'], dtype=str, sep=' ')

    ids_beagle_format = ids.IID.copy()
    ids_beagle_format = ids_beagle_format.map(lambda x: '{}_{}'.format(x, x))
    ids['ids_beagle_format'] = ids_beagle_format

    print("Number of people with mtDNA: {}".format(len(ids)))
    ids_choose = ids.loc[ids.IID.isin(ids_not_inbred.IID)].copy()
    print("Inbreeding coefficient below 0.25: {}".format(len(ids_choose)))

    matrix = []
    for i in range(num_sets):
        matrix.append(random.sample(list(ids_choose.index), k=num_ind))

    total = 0
    n = 0
    for subset1 in matrix:
        subset1 = set(subset1)
        for subset2 in matrix:
            subset2 = set(subset2)

            if subset1 == subset2:
                continue

            total += len(subset1.intersection(subset2))
            n += 1

    print("Average intersection of each pair of subset: {0:.2f}".format(total / n))
    print("Percentage average intersection: {0:.2f}%".format(total/n/num_ind*100))
    print("Number of individuals: {}".format(num_ind))
    print("Number of subsets: {}".format(num_sets))

    for i, subset in enumerate(matrix):
        df = ids.loc[subset]
        df.to_csv("/path/to/subset-output-{}-{}-{}".format(num_ind, num_sets, i+1), sep=' ', header=False, index=False)

    # Because the beagle-file is HELLA big, it is easier to use the cut-command
    # instead of reading the entire file in python (95Gig, just sayin)
    # Therefore, we're gonna read in the header of the beagle-file instead,
    # and write out the columns for use in the cut-command.

    beagle = pd.read_csv('/path/to/beagle/file.vcf',
        nrows=1,
        dtype=str,
        header=9,
        sep='\t'
        )

    print("..Finding column numbers in beagle-file")
    column_numbers = []
    for i, subset in enumerate(matrix):
        ids_beagle_format = set(ids.loc[subset, 'ids_beagle_format'])
        temp = '1-9,'
        for col_num, id in enumerate(beagle.columns):
            if id in ids_beagle_format:
                temp += '{},'.format(col_num + 1)
        column_numbers.append(temp[:-1])

    filepath = 'run-druid-col-numbers.temp'
    print("..Writing column numbers to {}".format(filepath))
    with open(filepath, 'w') as file:
        for col_num in column_numbers:
            file.write(col_num + '\n')
```

Listing D.4: run-druid-with-subsets.sh

```bash
#!/bin/bash
NUM_SETS=10
SET_SIZE=1500
BEAGLE_FILE_PATH_PREFIX="/path/to/
    beagle-$SET_SIZE-inds-chr-1-22-ne-100000-seed-1564-subset"
REFINED_IBD_FILE_PATH_PREFIX="/path/to/
    refined-ibd-$SET_SIZE-inds-chr-1-22-ne-100000-seed-1564-subset"
DRUID_BAKEOFF_FILE_PATH_PREFIX="/path/to/
    druid-bakeoff-$SET_SIZE-inds-chr-1-22-ne-100000-seed-1564-subset"
DRUID_FILE_PATH_PREFIX="/path/to/
    druid-$SET_SIZE-inds-chr-1-22-ne-100000-seed-1564-subset"
MAP_FILE="/path/to/genotyped-chr-1-22-FIN-cm.map"
BIG_BEAGLE="/path/to/beagle-phased-ne-100000-seed-1564-chrom-1-to-22.vcf"

# Uncomment this if this is not already done.
# python get-random-subsets.py

FILE_NUMBER=1

date
echo "Extract from beagle-files"
while read p;
do
    BEAGLE_FILE_PATH="$BEAGLE_FILE_PATH_PREFIX-$FILE_NUMBER.vcf"
    echo $BEAGLE_FILE_PATH
    cut -f $p $BIG_BEAGLE > $BEAGLE_FILE_PATH
    FILE_NUMBER=$((FILE_NUMBER+1))
done < run-druid-col-numbers.temp # This file is created by
    get-random-subsets.py

date
```

```
echo "Run refinedIBD on beagle-subsets:"
for file_num in {1..10}
do
    BEAGLE_FILE_PATH="$BEAGLE_FILE_PATH_PREFIX-$file_num.vcf"
    REFINED_IBD_FILE_PATH="$REFINED_IBD_FILE_PATH_PREFIX-$file_num"
    echo $REFINED_IBD_FILE_PATH
    java -Xmx32g -jar ~/localfiles/refinedIBD/refined-ibd.12Jul18.a0b.jar
    gt=$BEAGLE_FILE_PATH map=$MAP_FILE out=$REFINED_IBD_FILE_PATH > /dev/
    null
done

date
echo "Run druid-bakeoff script on refinedibd-files:"
for file_num in {1..10}
do
    REFINED_IBD_FILE_PATH="$REFINED_IBD_FILE_PATH_PREFIX-$file_num.ibd"
    gunzip -c "$REFINED_IBD_FILE_PATH.gz" > "$REFINED_IBD_FILE_PATH"

    DRUID_BAKEOFF_FILE_PATH="$DRUID_BAKEOFF_FILE_PATH_PREFIX-$file_num"
    echo $DRUID_BAKEOFF_FILE_PATH
    python ~/localfiles/druid-bakeoff/getIBD.py -f $REFINED_IBD_FILE_PATH
    $REFINED_IBD_FILE_PATH $REFINED_IBD_FILE_PATH -m $MAP_FILE -s 1 -t 1
    -o $DRUID_BAKEOFF_FILE_PATH
    rm $REFINED_IBD_FILE_PATH
done

date
echo "Run druid:"
for file_num in {1..10}
do
    DRUID_BAKEOFF_FILE_PATH="$DRUID_BAKEOFF_FILE_PATH_PREFIX-$file_num"
    DRUID_FILE_PATH="$DRUID_FILE_PATH_PREFIX-$file_num"
    DRUID_BAKEOFF_FILE_PATH="$DRUID_BAKEOFF_FILE_PATH_PREFIX-$file_num"
    echo $DRUID_FILE_PATH
    python ~/localfiles/druid/DRUID.py -o $DRUID_FILE_PATH -i
    $DRUID_BAKEOFF_FILE_PATH.ibd12 -s $DRUID_BAKEOFF_FILE_PATH.seg -m
    $MAP_FILE
done
```

Listing D.5: run-mitomatch-on-trios.py

```
from MitoMatch import mitomatch
from MitochondriaData import MitochondriaData
import variant_probabilities
import pandas as pd
import time
from variant_log_probabilities import variant_log_probabilities as vlp

relationship_to_degree = {
    'FS': 1,
    'P': 1,
    'C': 1,
    'HS': 2,
    'GP': 2,
    'GC': 2,
    'AU': 2,
    'NN': 2,
```

```python
    'PC': 1,
    'DC': 2,
    'UN': 100,
}

if __name__=='__main__':
    print("> Reading mitochondria data")
    mito_data = MitochondriaData("/prefix/to/mtdna-ped-and-map")
    trio_data = pd.read_csv("/path/to/trio-data-with-druid-degrees", sep='\t', header=0, usecols=['OFS.ID', 'PAT.ID', 'MAT.ID', 'DRUID'])
    trio_data.columns = ['child', 'father', 'mother', 'druid']

    #Filter out seventh degree relatives and above
    trio_data.druid = trio_data.druid.map(lambda x: relationship_to_degree[x] if x in relationship_to_degree else x)
    trio_data = trio_data.loc[trio_data.druid.astype(int) >= 7]
    trio_data.druid = trio_data.druid.map(lambda x: "unrelated" if x == 100 else x)

    # Print out the total number of mothers, fathers and children
    children = set(trio_data.child)
    mothers = set(trio_data.mother)
    fathers = set(trio_data.father)
    total = children.union(mothers).union(fathers)
    print("Children: {}, fathers {}, mothers {}, total {}".format(len(children), len(fathers), len(mothers), len(total)))


    distances_total = pd.DataFrame()
    step_size = 30
    start = time.time()
    for i in range(0, len(trio_data), step_size):
        print("Row... ", i) if i % 1000 == 0 else None

        sub_df = trio_data.iloc[i:i + step_size]
        individuals = set(sub_df.child).union(sub_df.father).union(sub_df.mother)

        distances = mitomatch(mito_data,
            individuals=individuals, calculation_type='weighted',
            weights=vlp, verbose=False)

        # filter out stuff:
        mother_child = [(id1, id2) for id1, id2 in zip(sub_df.mother.values, sub_df.child.values)]
        father_child = [(id1, id2) for id1, id2 in zip(sub_df.father.values, sub_df.child.values)]

        filter_moms = distances.apply(
            lambda r: (r.id1, r.id2) in mother_child or (r.id2, r.id1) in mother_child,
            axis=1)
        distances.loc[filter_moms, 'relationship'] = "mother/child"

        filter_dads = distances.apply(
            lambda r: (r.id1, r.id2) in father_child or (r.id2, r.id1) in father_child,
            axis=1)
        distances.loc[filter_dads, 'relationship'] = "father/child"

        distances = distances.dropna()

        distances_total = distances_total.append(distances)

    end = time.time()
    print("Distances, total:", len(distances_total))
    print("trio_data, total:", len(trio_data))
    print("Time spent:", (end-start)/60)

    distances_total.to_csv("/path/to/output.txt", index=False, sep='\t')
```

Appendix E

Plots

E.1 Hamming distance and degree of relatedness (DRUID)

This section contains letter value plots showing the Hamming distance on the y-axis and the degree of relatedness calculated by DRUID on the x-axis, for all subsets.

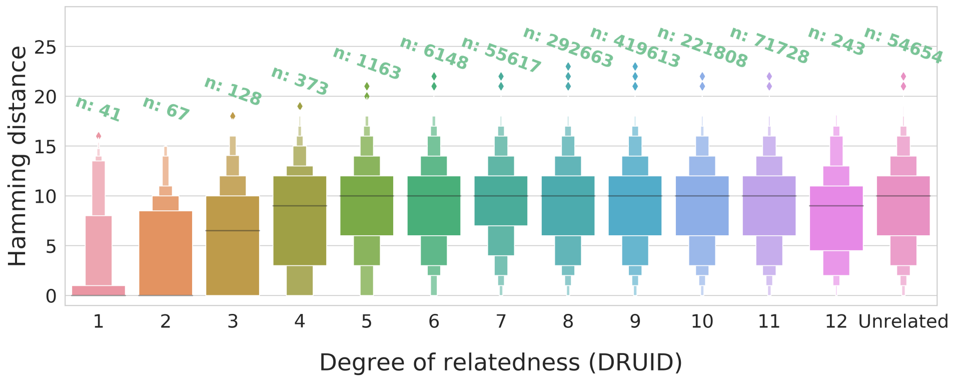


Figure E.1: Hamming distances: Subset 1

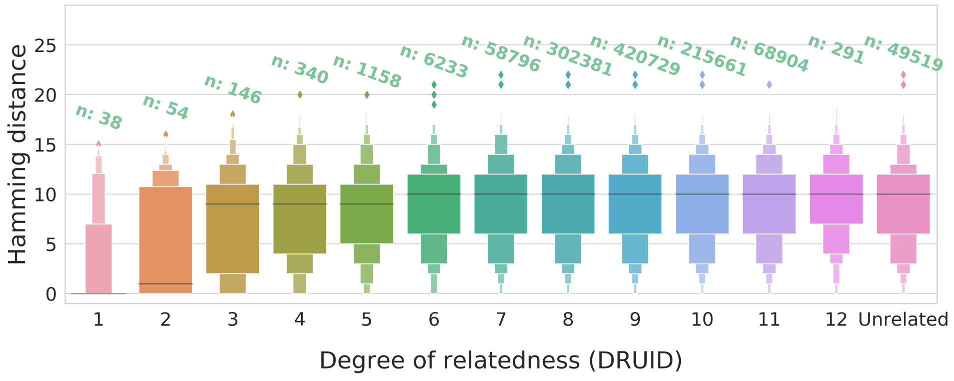


Figure E.2: Hamming distances: Subset 2

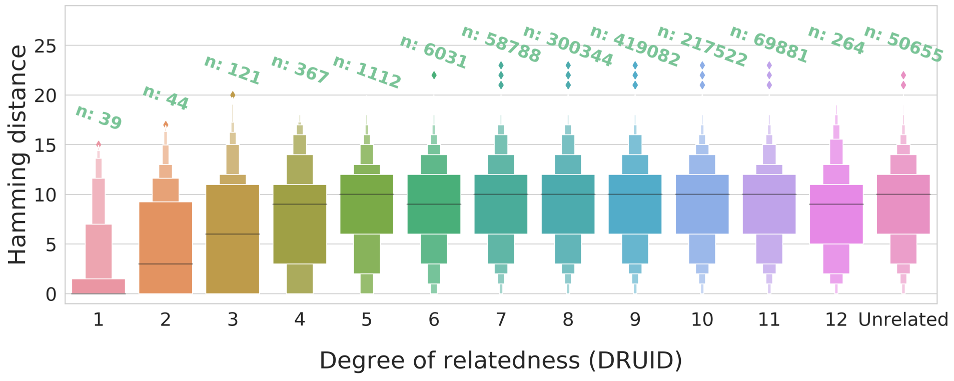


Figure E.3: Hamming distances: Subset 3

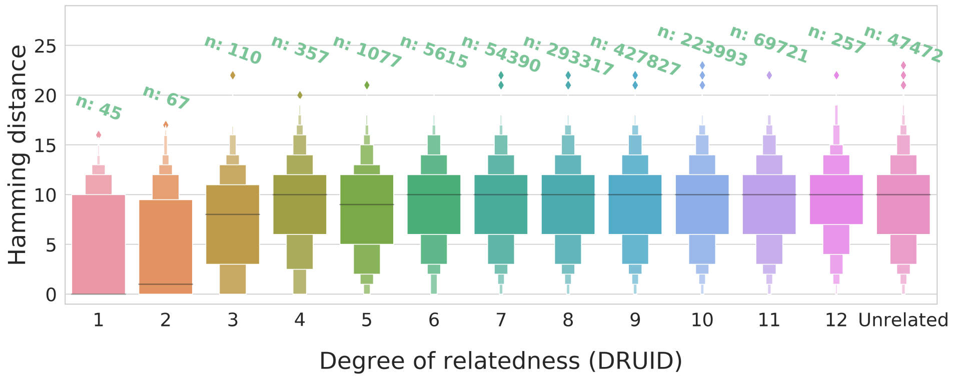


Figure E.4: Hamming distances: Subset 4

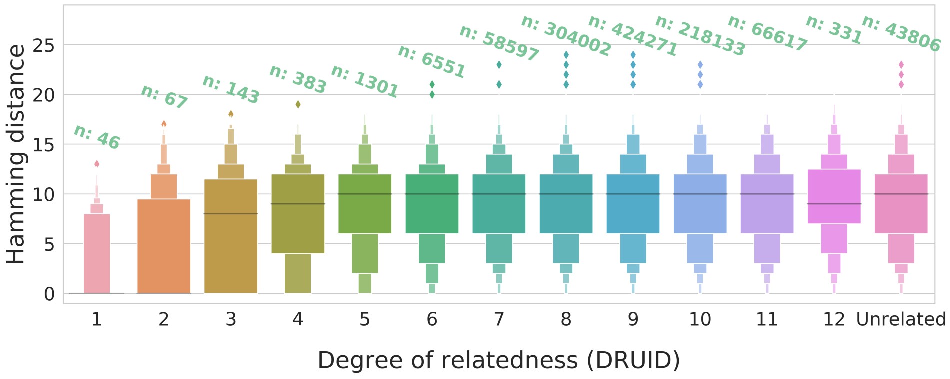


Figure E.5: Hamming distances: Subset 5

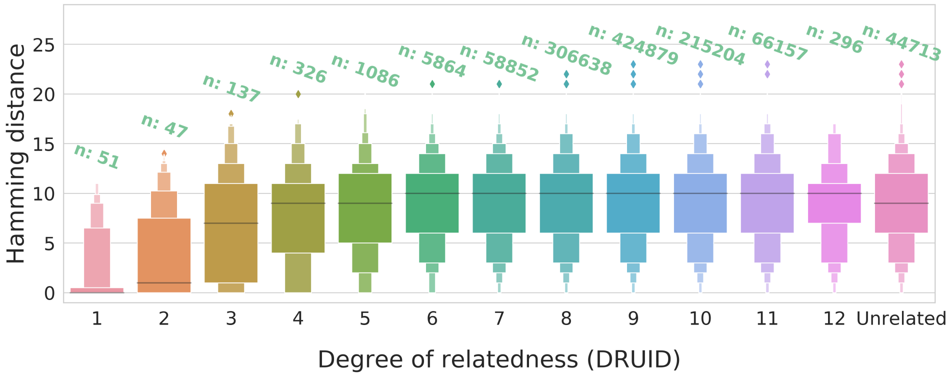


Figure E.6: Hamming distances: Subset 6

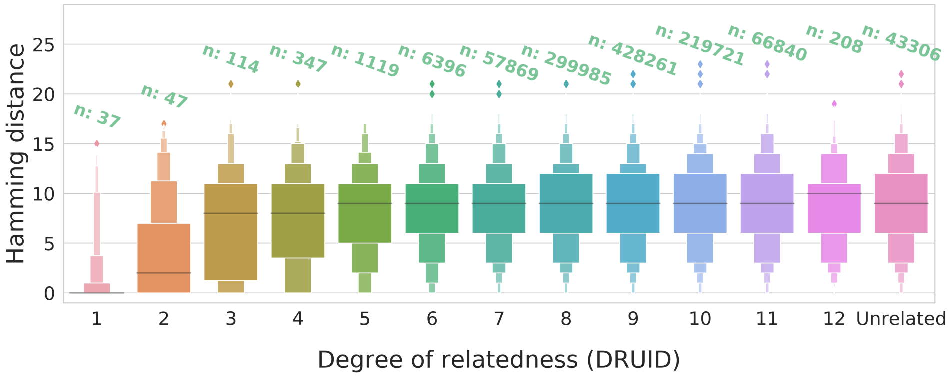


Figure E.7: Hamming distances: Subset 7

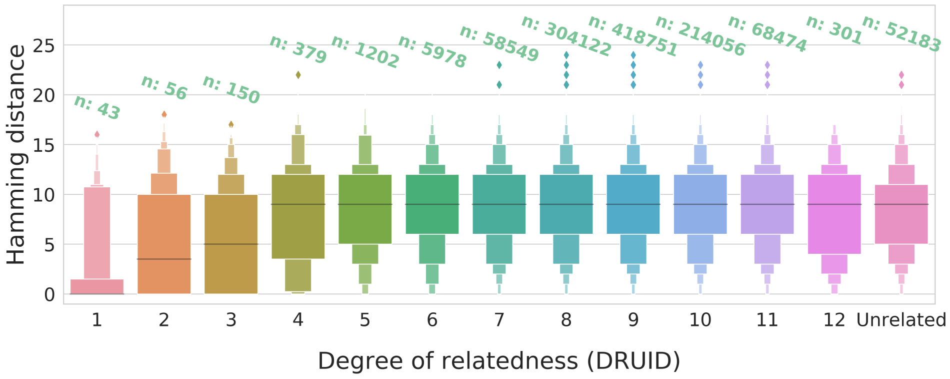


Figure E.8: Hamming distances: Subset 8

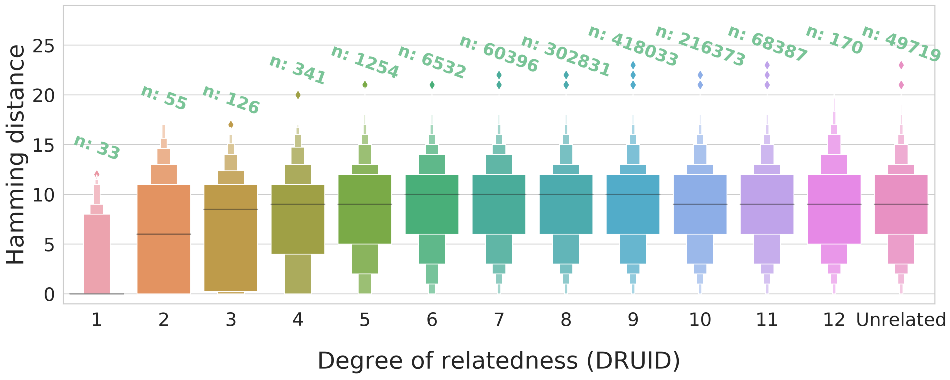


Figure E.9: Hamming distances: Subset 9

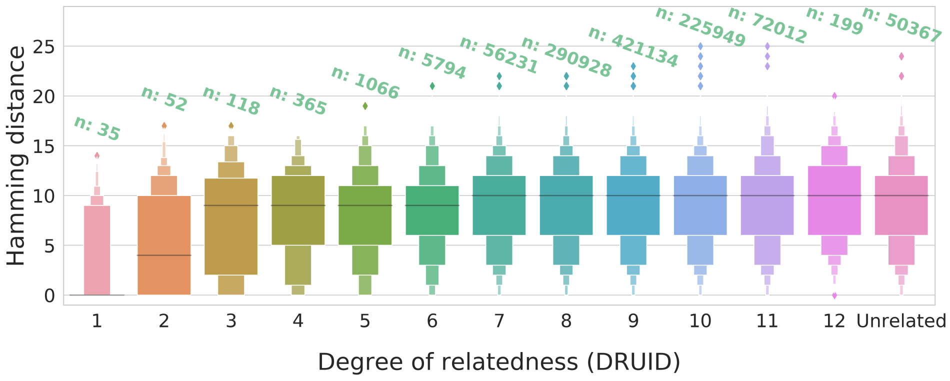


Figure E.10: Hamming distances: Subset 10

E.2 Weighted distance and degree of relatedness (DRUID)

This section contains letter value plots for all subsets, where a weighted distance has been calculated and compared with the degree of relatedness estimated by DRUID.

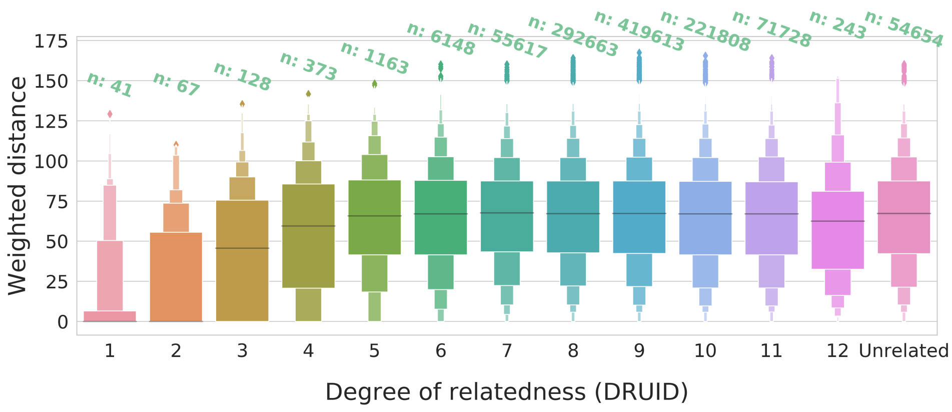


Figure E.11: Weighted distances: Subset 1

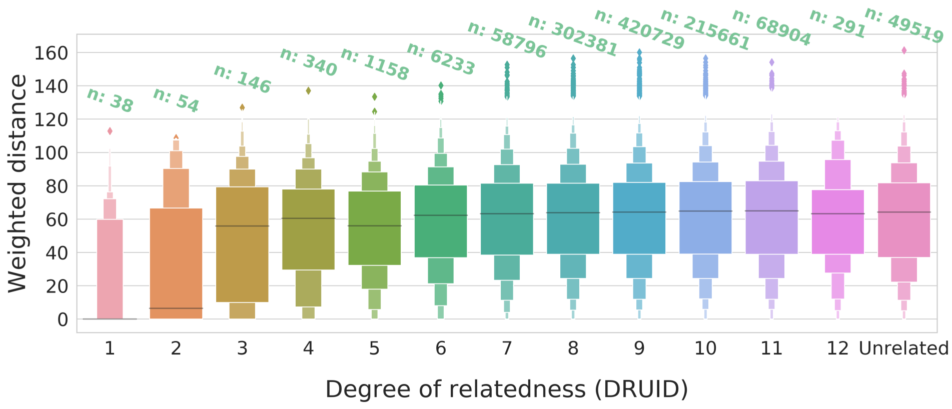


Figure E.12: Weighted distances: Subset 2

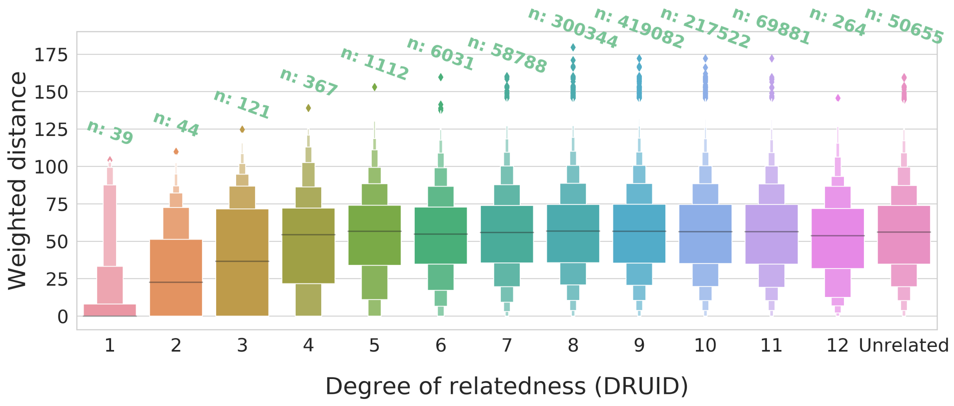


Figure E.13: Weighted distances: Subset 3

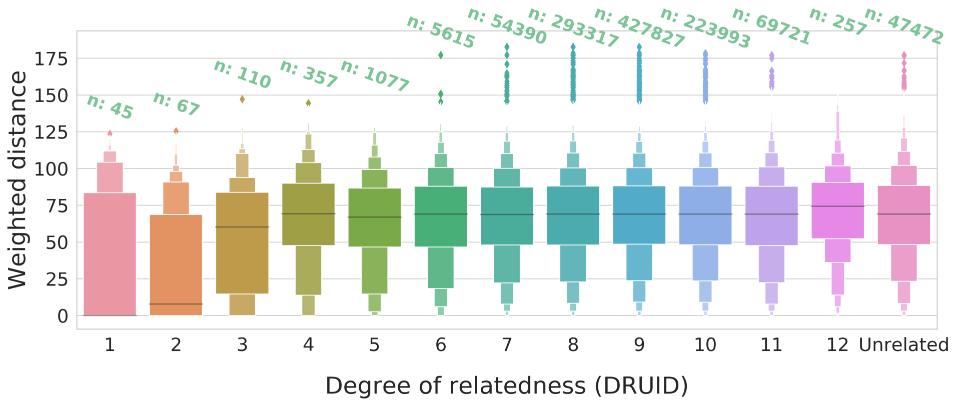


Figure E.14: Weighted distances: Subset 4

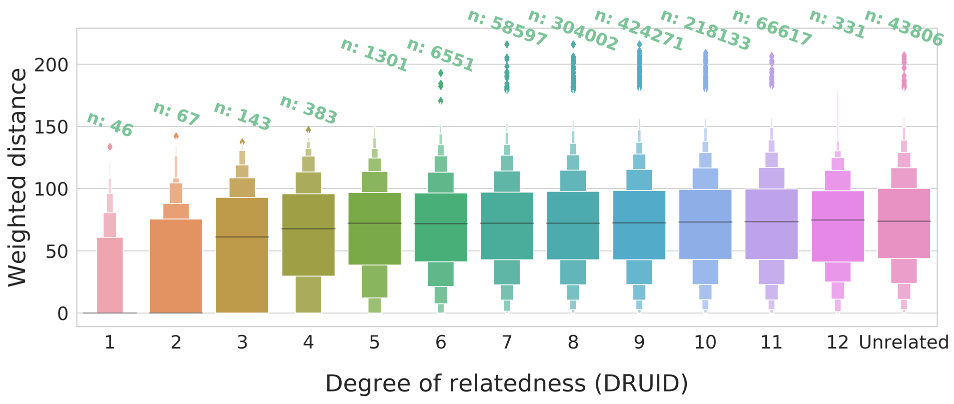


Figure E.15: Weighted distances: Subset 5

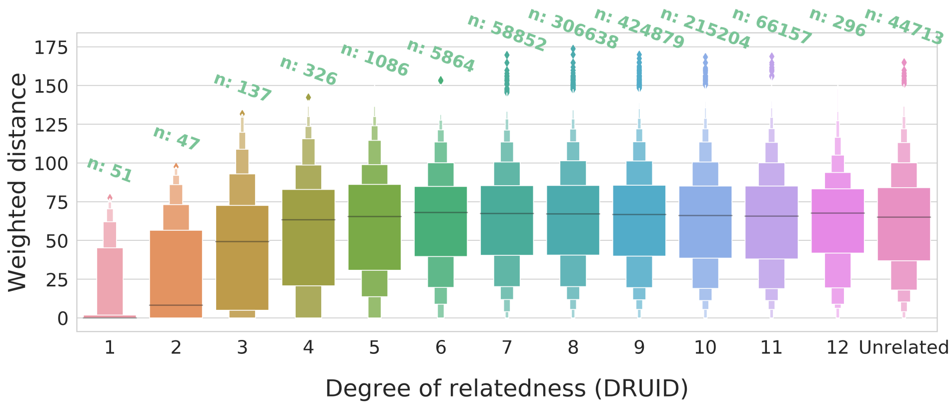


Figure E.16: Weighted distances: Subset 6

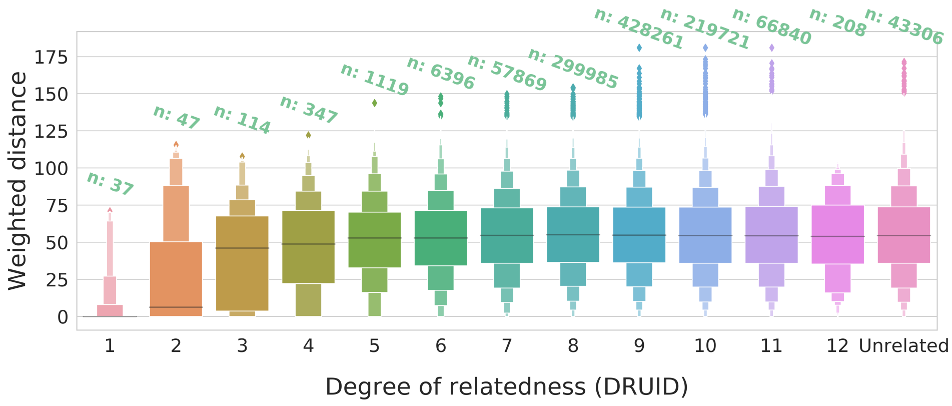


Figure E.17: Weighted distances: Subset 7

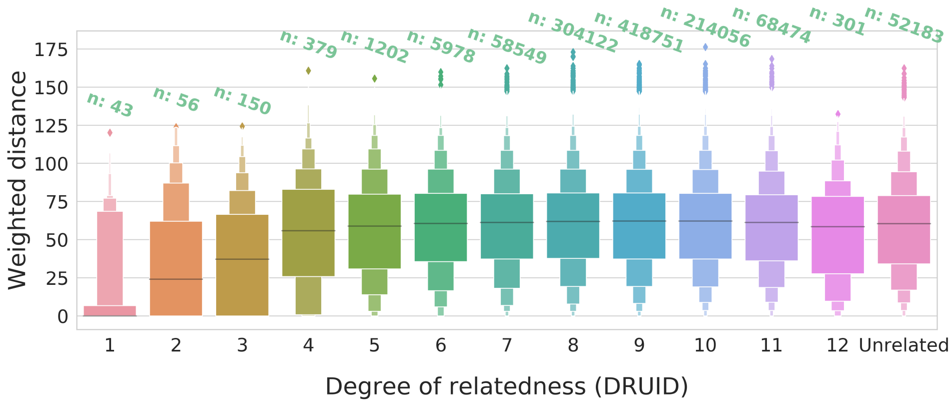


Figure E.18: Weighted distances: Subset 8

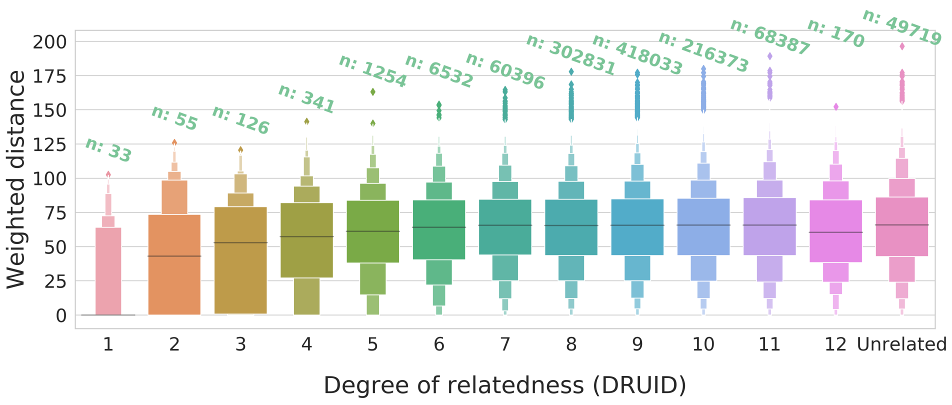


Figure E.19: Weighted distances: Subset 9

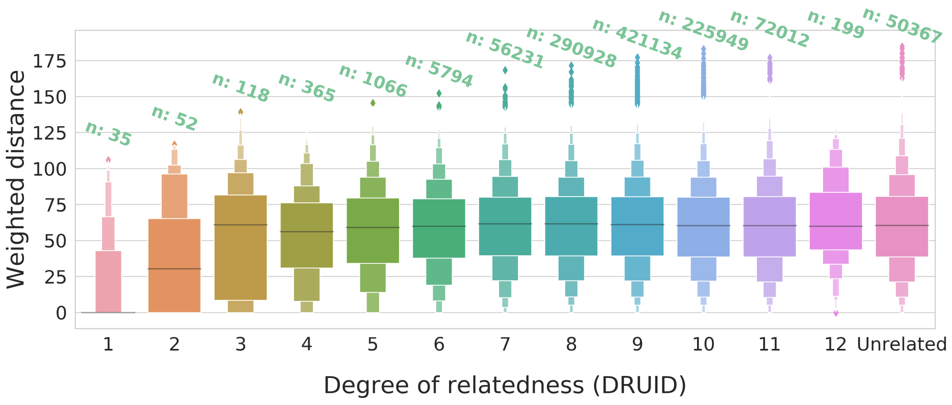


Figure E.20: Weighted distances: Subset 10

Appendix F

Information about the HUNT data set

NTNU
K.G. Jebsen Center for
Genetic Epidemiology

ALL-IN

GWAS-data

SNPs

This document provides a description of the

handling of the all-in genotyped data.

Document version 2018-09-07

Table of content

Changes

Documentation for genotyped data

This document provides a brief description of the handling of the all-in genotyped data from genotyping through QC. The purpose of the document is to provide background information for research using single or multiple SNP's which have been extracted and made available from the total dataset.

Background

From 2012-2015 the HUNT-Michigan (HUNT-MI) collaboration genotyped approximately 72.000 individuals from the HUNT biobank. The genotyping effort was a research collaboration between researchers at NTNU and the University of Michigan. Every individual with a DNA sample with a suitable DNA concentration was selected for genotyping. Samples were picked at random and genotyped in batches. All genotyping was performed at the Genomics-Core Facility (GCF) at the Norwegian University of Science and Technology, NTNU.

Contact list

Kristian Hveem	Leader, K.G. Jebsen center for genetic epidemiology	kristian.hveem@ntnu.no
Maiken E. Gabrielsen	Research coordinator, K.G. Jebsen center for genetic epidemiology	maiken.e.gabrielsen@ntnu.no
Anne Heidi Skogholt	Analysis coordinator, K.G. Jebsen center for genetic epidemiology	anne.heidi.skogholt@ntnu.no
Ben M. Brumpton	Senior Researcher, K.G. Jebsen center for genetic epidemiology	ben.brumpton@ntnu.no
Oddgeir L. Holmen	Leader, HUNT data center	oddgeir.l.holmen@ntnu.no

Acknowledgements

When using gwas - SNP´s in publications, please acknowledge the following:
The genotyping was financed by the National Institute of health (NIH), University of Michigan, The Norwegian Research council, and Central Norway Regional Health Authority and the Faculty of Medicine and Health Sciences, Norwegian University of Science and Technology (NTNU).
The genotype quality control and imputation has been conducted by the K.G. Jebsen center for genetic epidemiology, Department of public health and nursing, Faculty of medicine and health sciences, Norwegian University of Science and Technology (NTNU).

Quick overview:

Genotyping platform: Illumina

Chip: HumanCoreExome arrays:

- HumanCoreExome 12 v.1.0
- HumanCoreExome 12 v.1.1
- UM HUNT Biobank v1.0 (HumanCoreExome 24 with custom content)

Imputation: Human reference consortium (HRC) and custom panel including 2200 HUNT individuals with low pass WGS

About the data-set:

The data-set has been produced and quality controlled at the K.G. Jebsen center for genetic epidemiology, NTNU, in collaboration with Associate Professor Willer and Professor Abecasis at the University of Michigan. Below you will find the description of the handling of the original data-set.

All SNPs in the file have been imputed (including the also genotyped SNPs). Genotypes are coded as dosage. The value given is the dosage of the alternative allele.

Quality control

In total, DNA from 71,860 HUNT samples was genotyped using one of three different Illumina HumanCoreExome arrays (HumanCoreExome12 v1.0, HumanCoreExome12 v1.1 and UM HUNT Biobank v1.0). Samples that failed to reach a 99% call rate, had contamination > 2.5% as estimated with BAF Regress (Jun *et al*., 2012), large chromosomal copy number variants, lower call rate of a technical duplicate pair and twins, gonosomal constellations other than XX and XY, or whose inferred sex contradicted the reported gender, were excluded. Samples that passed quality control were analysed in a second round of genotype calling following the Genome Studio quality control protocol described elsewhere (Guo *et al*., 2014). Genomic position, strand orientation and the reference allele of genotyped variants were determined by aligning their probe sequences against the human genome (Genome Reference Consortium Human genome build 37 and revised Cambridge Reference Sequence of the human mitochondrial DNA; http://genome.ucsc.edu) using BLAT (Dunham et al., 2012). Variants were excluded if (1) their probe sequences could not be perfectly mapped to the reference genome, cluster separation was < 0.3, Gentrain score was < 0.15, showed deviations from Hardy Weinberg equilibrium in unrelated samples of European ancestry with p-value < 0.0001), their call rate was < 99%, or another assay with higher call rate genotyped the same variant.

Ancestry/Population structures

Ancestry of all samples was inferred by projecting all genotyped samples into the space of the principal components of the Human Genome Diversity Project (HGDP) reference panel (938 unrelated individuals; downloaded from http://csg.sph.umich.edu/chaolong/LASER/) (Li et al., 2008; Wang et al., 2014), using

PLINK v1.90 (Chang *et al.*, 2015). Recent European ancestry was defined as samples that fell into an ellipsoid spanning exclusively European populations of the HGDP panel. The different arrays were harmonized by reducing to a set of overlapping variants and excluding variants that showed frequency differences > 15% between data sets, or that were monomorphic in one and had MAF > 1% in another data set. The resulting genotype data were phased using Eagle2 v2.3 (Loh *et al.*, 2016).

Imputation

Imputation was performed on the 69,716 samples of recent European ancestry using Minimac3 (v2.0.1, http://genome.sph.umich.edu/wiki/Minimac3) (Das *et al.*, 2016) with default settings (2.5 Mb reference based chunking with 500kb windows) and a customized Haplotype Reference consortium release 1.1 (HRC v1.1) for autosomal variants and HRC v1.1 for chromosome X variants (McCarthy *et al.*, 2016). The customized reference panel represented the merged panel of two reciprocally imputed reference panels: (1) 2,201 low-coverage whole-genome sequences samples from the HUNT study and (2) HRC v1.1 with 1,023 HUNT WGS samples removed before merging. We excluded imputed variants with Rsq < 0.3 resulting in over 24.9 million well-imputed variants.

References

Chang, C. C., Chow, C. C., Tellier, L. C., Vattikuti, S., Purcell, S. M., & Lee, J. J. (2015). Second-generation PLINK: rising to the challenge of larger and richer datasets. *Gigascience, 4*, 7. doi:10.1186/s13742-015-0047-8

Das, S., Forer, L., Schonherr, S., Sidore, C., Locke, A. E., Kwong, A., . . . Fuchsberger, C. (2016). Next-generation genotype imputation service and methods. *Nat Genet*. doi:10.1038/ng.3656

Dunham, I., Kundaje, A., Aldred, S. F., Collins, P. J., Davis, C. A., Doyle, F., . . . Lochovsky, L. (2012). An integrated encyclopedia of DNA elements in the human genome. *Nature, 489*(7414), 57-74. doi:nature11247 [pii] 10.1038/nature11247

Guo, Y., He, J., Zhao, S., Wu, H., Zhong, X., Sheng, Q., . . . Long, J. (2014). Illumina human exome genotyping array clustering and quality control. *Nat Protoc, 9*(11), 2643-2662.

Jun, G., Flickinger, M., Hetrick, K. N., Romm, J. M., Doheny, K. F., Abecasis, G. R., . . . Kang, H. M. (2012). Detecting and estimating contamination of human DNA samples in sequencing and array-based genotype data. *Am J Hum Genet, 91*(5), 839-848. doi:10.1016/j.ajhg.2012.09.004

Li, J. Z., Absher, D. M., Tang, H., Southwick, A. M., Casto, A. M., Ramachandran, S., . . . Myers, R. M. (2008). Worldwide human relationships inferred from genome-wide patterns of variation. *Science, 319*(5866), 1100-1104. doi:10.1126/science.1153717

Loh, P.-R., Danecek, P., Palamara, P. F., Fuchsberger, C., Reshef, Y. A., Finucane, H. K., . . . Price, A. L. (2016). Reference-based phasing using the Haplotype Reference Consortium panel. *bioRxiv*. doi:http://dx.doi.org/10.1101/052308

McCarthy, S., Das, S., Kretzschmar, W., Delaneau, O., Wood, A. R., Teumer, A., . . . Haplotype Reference, C. (2016). A reference panel of 64,976 haplotypes for genotype imputation. *Nat Genet, 48*(10), 1279-1283. doi:10.1038/ng.3643

Wang, C., Zhan, X., Bragg-Gresham, J., Kang, H. M., Stambolian, D., Chew, E. Y., . . . Abecasis, G. R. (2014). Ancestry estimation and control of population stratification for sequence-based association studies. *Nat Genet, 46*(4), 409-415. doi:10.1038/ng.2924

Alice Gudem

Mitodetect: Estimating common maternal ancestry in the HUNT study

NTNU
Kunnskap for en bedre verden