Transnational University Limburg is a unique collaboration of two universities in two countries: the University of Hasselt and Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculty of Sciences

School for Information Technology

Master of Statistics and Data Science

Master's thesis

An analytical pipeline for processing and analysis of proteomes of the S. cerevisiae Reference Assembly Panel (ScRAP)

Alvaro Gomez Perez

Thesis presented in fulfillment of the requirements for the degree of Master of Statistics and Data Science, specialization Bioinformatics

SUPERVISOR :
Prof. dr. Dirk VALKENBORG
De heer Frédérique VILENNE

SUPERVISOR :
Dr. Julia MUENZNER
Dr. Pauline TREBULLE

2023
2024

Faculty of Sciences
School for Information Technology
Master of Statistics and Data Science

Master's thesis

An analytical pipeline for processing and analysis of proteomes of the S. cerevisiae Reference Assembly Panel (ScRAP)

Alvaro Gomez Perez

Thesis presented in fulfillment of the requirements for the degree of Master of Statistics and Data Science, specialization Bioinformatics

SUPERVISOR :

Prof. dr. Dirk VALKENBORG

De heer Frédérique VILENNE

SUPERVISOR :

Dr. Julia MUENZNER

Dr. Pauline TREBULLE

HASSELT UNIVERSITY

MASTER THESIS

Development of pre-processing and analytical procedures to assess the proteomic impact of structural genomic variation across the *S. cerevisiae* species

Author:
Álvaro Gómez Pérez

Supervisor:
INTERNAL
prof. Dr. Dirk Valkenborg
Frédérique Vilenne

EXTERNAL
Dr. Julia Muenzner
Dr. Pauline Trébulle

A thesis submitted in fulfillment of the requirements for the Masters of Statistics and Data Science

in the

Ralser Lab
Insitute of Biochemistry at Charité Universitätsmedizin Berlin

August 20, 2024

"Try again, fail again, fail better."

Samuel B. Beckett

HASSELT UNIVERSITY

Abstract

Faculty of Sciences

Master of Statistics and Data Science

Development of pre-processing and analytical procedures to assess the proteomic impact of structural genomic variation across the *S. cerevisiae* species

by Álvaro Gómez Pérez

Numerous fields have been dependent on the use of reference genomes: this is, the genome of an idealized individual of a species, which has been assembled from - potentially multiple - high-quality sequencing runs, and which is used as a reference for the whole species. Nonetheless, reference genomes fail to capture the genetic diversity of a species. Recently, the concept of pangenomes has emerged. Pangenomes unify sequenced genomes corresponding to different strains, isolates, or individuals within a species, and thus better cover the genomic space of a species. Pangenomes can provide an insight into a species' genetic diversity, enabling, for example, evolutionary tracing, or improving genotype-to-phenotype mapping. Pangenomes have been assembled for multiple organisms, such as *Escherichia coli*, *Drosophila melanogaster*, or *Saccharomyces cerevisiae*. In addition to the use of pangenomes, long-read sequencing techniques have become available over the last few years which allow for gapless telomere-to-telomere assemblies of chromosomes. Recently, a species-representative panel of S. cerevisiae isolates has been selected to undergo such long-read sequencing in order to assess the effect of genomic structural variants within the species. This panel is known as the *Saccharomyces cerevisiae* Reference Assembly Panel (ScRAP). Strains in the ScRAP represent the genetic diversity of the *S. cerevisiae* species; this is, strains of different ploidies, zygosities, and strains containing complex aneuploidies are included. In this work, proteomics measurements were obtained for 134 of the ScRAP strains, with a median of roughly 2300 protein identifications per sample. In many cases, analysis of proteomics data based on a reference genome is sufficient to quantify and compare protein abundances across samples. However, in order to target questions such as allele-specific expression in diploid and polyploid strains, or the expression of proteins affected by structural variants, it is necessary to take each strain's actual genome into account. In this thesis, reference genome-based and strain-specific processing approaches for the ScRAP proteomic dataset are developed and compared. The strain-specific approach significantly increased the number of protein identifications per strain by an average of around 35%. Furthermore, the strain-specific processed data allowed for promising findings at the biological level: 51 proteins were found to be significantly differentially expressed between haplotypes in heterozygous diploid strains, and 16 proteins containing deletions or non-coding insertions were found to be significantly affected with regard to their expression. Thus, the conjunction of deep sequencing and high-throughput proteomics, followed by strain-specific processing of data, promises to be a powerful tool for unconvering the effect of genomic structural variants on protein expression, and strain-specific expression patterns.

Keywords: Data independent acquisition, haplotype, mass spectrometry, natural isolate collection, proteomics, structural variant.

Acknowledgements

I would like to express my gratitude to Prof. Dr. Markus Ralser, for welcoming me in his research group and providing me with such an interesting project.

I would also like to thank my external supervisors, Dr. Julia Muenzner and Dr. Pauline Trébulle, for their invaluable guidance throughout the project, as well as for their understanding, supportive and encouraging attitude.

I am also thankful to my internal supervisor Prof. Dr. Dirk Valkenborg and co-supervisor Frédérique Vilenne for their suggestions and feedback.

Additionally I would like to express my thanks to our collaborators Prof. Dr. Gilles Fischer and Andrea Tarabini, who graciously provided us with the isolates and genomic files studied in this project, and answered all of our questions pertaining them.

I would also like to express my appreciation to the Core Facility High Throughput Mass Spectrometry of the Charité for support in sample preparation, acquisition, and analysis, as well as to Lisa Henning for her cultivation of the strains and preparation of the samples herein analyzed.

I am also thankful to all the colleagues at the Ralser Lab who have shared their knowledge with me and helped me see this project through.

Lastly, I would like to thank my close friends and family for their unconditional support throughout the process of obtaining my master's degree.

Contents

List of Figures

List of Abbreviations

CA	**C**ommon peptide**A**pproach
CV	**C**oefficient of **V**ariation
DDA	**D**ata **D**ependent **A**cquisition
DIA	**D**ata **I**ndependent **A**cquisition
GDPF	**G**enome-**D**erived **P**rotein **F**ile
HP	**H**a**p**lotype
LC	**L**iquid **C**hromatography
MS	**M**ass **S**pectrometry
OD	**O**ptical **D**ensity
ORF	**O**pen **R**eading **F**rame
QC	**Q**uality **C**ontrol
RT	**R**etention **T**ime
ScRAP	**S**accharomyces **c**erevisiae **R**eference **A**ssembly **P**anel
SM	**S**ynthetic **M**inimal (medium)
SNP	**S**ingle **N**ucleotide **P**olymorphism
SPE	**S**olid **P**hase **E**xtraction
SSA	**S**train **S**pecific **A**pproach
SV	**S**tructural **V**ariant
TIC	**T**otal **I**on **C**ount

Chapter 1

Introduction

1.1 *Saccharomyces cerevisiae* and genetic diversity

Saccharomyces cerevisiae is a well-known model organism that has been extensively studied due to its numerous advantages: it is a non-pathogenic unicellular organism which easily grows under laboratory conditions, and which can be grown in media with very diverse compositions, allowing researchers to explore its response to different chemical and physical environments [1, 2]. Furthermore, it is a eukaryotic organism, which makes findings more easily generalizable to humans and other eukaryotic species.

Another interesting characteristic of *S. cerevisiae* is that it occurs with different ploidy states in nature; that is, how many full sets of chromosomes are present in each cell. As an example, it is known that a full set of chromosomes in humans contains 23 chromosomes, and all humans have two of such full sets in all of their cells (except for gametes or reproductive cells). The case of *S. cerevisiae* is quite different: a full set of chromosomes contains 16 of them, and different strains will contain different numbers of chromosome sets, as also happens in other eukaryotic species [3, 4]. Organisms containing a single set of chromosomes are known as haploid, those with two sets are diploids, and so forth; starting from three sets of chromosomes, organisms can be generally referred to as "polyploid". In the case of *S. cerevisiae*, strains have been observed ranging from haploid to tetraploid, although haploids and diploids are the most common [5].

In the case of organisms with more than a single set of chromosomes, the concept of zygosity appears: zygosity refers to the degree to which the information contained in one set of chromosomes is, evaluated gene by gene, the same as that in the other set(s) of chromosomes of an organism. Organisms with the same information across chromosome sets are known as homozygous, and those with differing information across them as heterozygous. In this way, an organism with three sets of chromosomes, all of them containing the same information for all genes, would be referred to as a homozygous triploid, while an organism with two sets of chromosomes with differing information would be a heterozygous diploid. It must be noted that in the case of humans, due to the mode of reproduction being exclusively sexual, all individuals are heterozygous; however, in other organisms with different means of reproduction, functional homozygous individuals do occur. Importantly, homo- and heterozygosity are terms that can also be used at the level of single genes: this is, even though an individual can be overall heterozygous, this does not preclude that it can have the exact same information across chromosome sets for some of its genes; in fact, this will almost always be the case. Hence, it can be said that an organism is homo- or heterozygous for a certain gene, meaning respectively that it carries the same or a different allele (this is, the same or a different version of the gene) in its different chromosome sets. The information contained in a single set of chromosomes (this is, the set of alleles present in it) is referred to as "haplotype".

Aneuploidy is also a common event in *S. cerevisiae* [6]. Aneuploidy refers to the presence of an aberrant number of chromosomes in the cell, which in the case of *S. cerevisiae* means more or less than a multiple of 16: this is, there is either at least 1 chromosome missing, or at least 1 extra chromosome present. In their large collection of *S. cerevisiae* species, containing 1,011

different natural isolates, Peter et al., 2018 [5] found 19.1% of them to contain some kind of aneuploidy.

S. cerevisiae is not only an extremely popular and useful model organism, it is also widely distributed across the world. A large number of different strains with their own phenotypic characteristics adapted to their particular biological niches [5] have been isolated and described. One way that strains are classified refers to their isolation source: domesticated strains are those used for the production of wine, sake, bioethanol..., while wild strains are isolated from the natural medium: trees, insects... Apart from these, there are human strains, which are isolated from the body of humans in a clinical setting and laboratory ones, strains adapted to growth in laboratory conditions toward research purposes. This large variety of strains within the species makes it a prime subject for the study of population genomics and within-species genetic diversity in general.

The study of genetic diversity within species has gained popularity in the last decades, due to its numerous benefits: firstly, it allows for the discovery of new phenotypic and genetic traits and the relationship between them, as well as for the better understanding of previously known ones. It also enables a deeper understanding of the species as a whole and even of its evolution and origin, as is precisely the case of *S. cerevisiae*, which was recently postulated to have a "single out-of-China origin" [5]. Furthermore, the study of the genetic diversity of a species, including as many and as varied of its strains as possible, permits for research conclusions to become increasingly generalizable. This is, conclusions based on a single laboratory strain could be extremely biased and might not apply to the whole of the species nor to other organisms, while conclusions drawn from research performed on a large set of strains of diverse origins might become much more generizable [6, 7, 5, 8].

Peter et al. [5] produced a collection of 1,011 *S. cerevisiae* strains from diverse ecological origins and performed whole-genome sequencing of them, with the intention of sampling as much of the species' genomic space as possible. This allowed to obtain a comprehensive view of *S. cerevisiae*'s genome evolution, taking into account differences in ploidy, aneuploidies and genetic variants, which had not been done before for such a wide panel in this species. This, together with their efforts to study the phenotypic characteristics of these strains as well, produced an extremely insightful study into the evolution of the species and the relationship between its genotype and phenotype. Based on this seminal study, the *Saccharomyces cerevisiae* Reference Assembly Panel (ScRAP) was developed, around which the present thesis project revolves.

1.2 The *Saccharomyces cerevisiae* Reference Assembly Panel

The ScRAP [9] was developed with the goal of deepening the discoveries made in the above presented study [5], by characterizing the structural variants (SVs) in the different strains, as well as their effects. It was based on a subset of the strains included in [5], selected specifically to maintain as much diversity as possible, both with respect to ecological niche of origin and ploidy. Concretely, 142 strains were selected, whose classification according to origin and ploidy is presented in Figure 1.1. These strains were newly sequenced making use of single-molecule long-read sequencing technologies, which enable the construction of deep, gapless, reference-quality genomes (so called "telomere-to-telomere assemblies"). This allowed for the identification of numerous SVs to an extent never achieved before in *S. cerevisiae*. Structural variants are genomic changes affecting more than one nucleotide in the DNA sequence, such as insertions, deletions, contractions or inversions, as opposed to SNPs (Single Nucleotide Polymorphisms), which affect solely one nucleotide in the sequence. A total of 36,459 SVs were found across the 141 non-reference strains as compared to the reference *S. cerevisiae* strain,

	Haploid	Monosporic haploid	Monosporic diploid	Homozygous diploid	Heterozygous diploid	Triploid	Tetraploid	Total
Wild	10	6	29	6	4	1		56
Domesticated	20	5	13	6	12	6	7	69
Human	5	2	0	1	5			13
Laboratory	3	1	0	0				4
Total	38	14	42	13	21	7	7	142

Figure 1.1: Classification of the strains in the ScRAP, according to both their niche of origin and their ploidy. "Monosporic" refers to the procedure through which the strain was isolated. Illustration obtained from *Telomere-to-telomere assemblies of 142 strains characterize the genome structural landscape in* Saccharomyces cerevisiae, by O'Donnell et al., 2023, Fig. 1a [9].

S288C. These were caused by 4,809 unique SVs, which were differently present across the strains.

An extremely interesting feature of this study is that the authors succeeded in performing haplotype phasing for heterozygous diploid strains. Haplotype phasing is a method which allows to resolve, after sequencing a diploid or polyploid organism, which sequenced fragments came from each of its sets of chromosomes, hence understanding what information was contained in each of the haplotypes. Haplotype phasing can be based on either experimental or computational methods, with the second being the most cost-effective, and the one that was used in this study [10]. This method is an extremely useful tool in order to understand an organism, its origins and evolution, as well as the relationship between its genotype and its phenotype: this is, it may allow to evaluate which of the observable characteristics of a strain reflect the information in each haplotype, as well as whether there is a dominance of one haplotype over the other when it comes to the expression of certain genes. O'Donnell et al. [9] also note in their study how the successful haplotype phasing of heterozygous diploid strains increased the number of SVs that were detected, and which would have remained hidden had this technique not been used. The practical consequence of the successful haplotype phasing for these 21 strains was that two separate genomic sequence FASTA files were produced for each of them, one with the information from each haplotype. It must also be noted that haplotype phasing was applied to heterozygous polyploid (triploid and tetraploid) strains in the ScRAP as well, although with limited success, meaning that a single genomic sequence FASTA file was produced for each of them, containing all sequenced alleles in the strain but with no identification of which haplotype each allele came from.

Aside from haplotype phasing, a number of interesting findings were presented in this article. One example is how SVs impact gene expression at the locations where they appear, which can occur due to them affecting the sequence of the open reading frame (ORF), modifying the regulatory elements or the number of copies of the gene. They also stated that these SVs affecting previously existing genes help create new ones, hence growing the gene repertoire of the species. Another of their findings was that SVs produce complex aneuploid chromosomes, with a large proportion of aneuploid chromosomes being associated to large SVs. However, the authors studied these strains solely at the genomic level. Even though the genome is the basis

for everything that happens inside an organism, it is known that there is a distance between the information contained in it and the actual phenotype of the organism: DNA expression is strongly and precisely regulated depending on numerous factors, which affects which genes get to be expressed (and in which quantities), and even after transcription and translation take place and the corresponding protein is produced from a gene, post-transcriptional modifications can change the structure and activity of the protein [11]. This is the reason why it is important to not only study organisms at the genomic level, but also at the level of their proteome; because the proteome is much closer to the actual phenotype of the organism.

Some questions that could be targeted based on the proteome of the ScRAP strains would be to evaluate the actual effect of different types of mutations on the expression of the genes they affect: is the protein encoded by a mutated gene produced at all? Does it have the expected sequence or is a new, chimeric protein produced by the fusion of two previously existing genes? Furthermore, the above presented phased haplotypes for heterozygous diploid strains are also a particularly interesting topic of study through proteomics, since they could allow to evaluate if the same amount of a certain protein is produced from both haplotypes, or if one is dominant over the other. These are just some examples which serve to illustrate the vast number of biological questions that could be targeted by a proteomic analysis of the ScRAP strains.

1.3 Proteomics and mass spectrometry

Proteomics is the discipline that focuses on the identification and quantification of proteins, but which can also be extended to study their structure, function, modifications and interactions [12]. Since proteins are some of the most versatile molecules and are present across all living beings, there is a long list of fields in which proteomics can be of use, including but not limited to clinical applications (identifying proteins that can serve as biomarkers for diseases), pathogenesis mechanisms research (identifying the means of infection of pathogens, which are usually protein-based) or, as in the present study, the analysis of metabolism and genetic diversity [13].

Originally, before the advent of -omics sciences, protein analysis was an extremely costly and labor-intensive procedure: a single type of protein that was the subject of the analysis had first of all to be isolated and purified from a sample, and then complex biochemical techniques, such as Edman degradation, had to be used in order to identify the amino acids making up the protein, one by one. Over the last few decades, a number of techniques have been developed that slowly eased the analysis of the proteins in a sample. Firstly, so-called chromatography techniques were developed to separate a complex mixture of proteins based on their physico-chemical characteristics, which then allowed to either directly quantify the amount of proteins with common characteristics, or to forward these fractionated samples to a further analysis. The basic principle of chromatography is that a sample containing proteins or peptides, is added to a certain surface to which these molecules can adsorb or bind according to their structure, charge, etc., known as the "stationary phase". Next, a solvent (the "mobile phase") with certain chemical characteristics is allowed to pass through the surface, which will progressively elute different peptides or proteins, based on their physico-chemical properties [14]. Some well-known chromatography techniques are ion-exchange chromatography, size exclusion chromatography, or affinity chromatography. Electrophoresis gel-based techniques were also frequently used (and still are) to fractionate complex protein samples, typically based on the isoelectric point and molecular mass of proteins [15]. However, the downside of all of these methods is that they do not allow to target specific proteins accurately, this is, they separate proteins based on their physico-chemical characteristics but do not allow to study the actual amino acid sequence of proteins.

On the other hand, antibody-based methods such as ELISA (Enzyme-Linked Immunosorbent Assay) were later developed, which use antibodies specific to a certain protein sequence or

epitope to detect whether a protein is present in a sample, and to quantify it. This fixed the lack of specificity of chromatography or gel-based techniques. Such methods are still heavily used today [16, 17]. Nevertheless, it has the obvious downside that a specific antibody is needed for each protein that should be detected. Nonetheless, antibody-based methods gave rise to the first true high-throughput technique for protein analysis (high-throughput meaning that it allows for the analysis of multiple samples at once with minimal effort): microarrays. Microarrays can be used for the analysis of DNA or RNA as well, but in all cases the principle is the same: they consist of a small surface where certain molecules are fixed, which will react with the molecules to be detected. In the case of protein microarrays, the first type developed where analytical microarrays, which were based on ELISA, but to a much larger scale on a smaller device: a large number of different antibodies specific to different protein sequences were fixed to the surface of the microarray, and emitted a signal when the corresponding protein was contained in the sample and bound to them. Later, other types of protein microarrays such as functional microarrays where developed [18]. However, despite their usefulness and the increased throughput, these techniques still require prior knowledge of the protein sequences for them to be detected at all.

The most important, and by far the most popular technique for high-throughput analysis of complex protein mixtures nowadays is mass spectrometry (MS), usually employed in tandem (this is, two MS steps right after each other) and preceded by some type of chromatography in order to fractionate the sample beforehand. The advantage of this technique is that the lack of prior information about a certain protein does not necessarily prevent its identification. This is the technique that is employed in the present study.

There are multiple types of mass spectrometry-based proteomics, such as bottom-up, top-down or cross-linking. The one employed in this project is bottom-up proteomics, where proteins are fragmented prior to the analysis and information at the protein level is afterwards reconstructed through the use of different algorithms [19].

The procedure generally starts with the digestion of proteins into peptides, normally performed with trypsin, an effective enzyme with a well-known restriction pattern (it will always cleave protein sequences at the C-terminal side of the amino acids lysine (K) and arginine (R), unless they are followed by a proline (P)). Subsequently, a chromatography step takes place, which allows to start from a complex peptide mixture and fractionate it based on specific characteristics of the peptide molecules. As an example, in liquid chromatography, the sample is added to a porous column, to which the peptides adsorb. Then, a liquid solvent is run through the column in a gradient; this is, the solvent might be 100% water at the beginning (which will hence elute polar peptide molecules, which are soluble in water), and will then progressively over the course of a defined time, reduce its content in water and increase its content in a non-polar solvent, for example acetonitrile, until it consist of 100% acetonitrile. The time over which this full gradient is run through the column varies widely, anywhere from a couple of minutes to several hours, and is an important characteristic of the proteomics procedure. This is due to the fact that the longer the gradient is run, the more separated the peptides will be from each other in this first dimension, and the more distanced in time they will go into the mass spectrometer, which will generally increase the resolution; this is, the ability to identify more peptides. The time at which each peptide leaves the chromatography column is known as its retention time (RT), and as mentioned above, is the first dimension of separation in the procedure.

After leaving the chromatography column, peptides are introduced into the mass spectrometer. There are numerous kinds of mass spectrometers that are used in proteomics, and although their description is beyond the scope of this thesis, their general working principle will be briefly described. All mass spectrometers are composed of three main sections: an ion source, a mass analyzer and a detector [20].

As they enter the mass spectrometer, peptides are directed to the ion source, where they are ionized. This is, they undergo a process through which they acquire an electric charge (which usually goes from +1 to +4, although this may vary). At this point, these molecules stop being referred to as peptides and start being referred to as precursors, which are nothing but peptides with a certain charge state. It is important to realize that a single peptide (a certain sequence of amino acids) can give rise to different precursors, depending on the charge state it acquires. There are several ionization methods used in MS, with electrospray ionization (ESI) and matrix-assisted laser desorption ionization (MALDI) being the most popular ones [21]. It must be noted that the success of the ionization process depends on the chemical characteristics of each peptide, and in fact some peptides do not ionize well and consequently cannot be detected [22]. This is due to the fact that after ionization, precursors will be sorted in the mass analyzer based on their behavior when subjected to an electric field, and if their charge state is 0, the electric field will have no effect on them. This step, the main one in the mass spectrometer (where precursors are separated based on their physico-chemical properties as well as their mass and structure), can take place in numerous different ways: time-of-flight (TOF), quadrupole, or trapped ion mobility spectrometry (TIMS) are just some of the examples. In this study, a tandem TIMS-TOF instrument was used and therefore, the two underlying techniques will be briefly covered.

The principle behind TIMS consists of keeping precursors in place within a chamber in the spectrometer by applying a certain electric field to them. At the same time, a current of inert gas moves through the chamber, and by finely regulating both the electric field and the flow of the inert gas, precursors with certain characteristics are slowly allowed to be carried by the gas current, and moved outside of the spectrometer [23]. Concretely, it is the most mobile ionized peptides that are more rapidly carried by the inert gas current, with a higher mobility being a consequence of, mainly, a larger charge, smaller size and compact structure. Between the two tandem MS steps, the precursors coming out of the first MS (MS1) are fragmented again. This, once again, can happen in different ways, with collision-induced dissociation (CID) being the most popular one: this is, introducing the precursors out of the MS1 in a collision chamber, where they are hit with molecules of an inert gas, causing them to fragment [24]. Precursor fragments enter then the second and final MS (MS2), which in the case of this study was a time-of-flight (TOF) mass spectrometer: this machine consists of a long chamber where a vacuum is induced, and through which the charged precursor fragments are accelerated by subjecting them to an electric field. This acceleration is proportional to both their mass and their charge, which is why, depending on the time they take to reach the detector at the end of the chamber, their mass-to-charge ratio (m/z) can be inferred.

Concluding the LC-MS/MS experiment, information will have been obtained at three levels for each precursor, represented in the scheme in Figure 1.2: first, the retention time at which it left the chromatography column (labelled (a) in the figure); second, the m/z at which it was detected in MS1 ((b) in the figure); and lastly, the spectrum of peaks detected in the MS2 for its fragments ((c) in the figure). Incidentally, the way in which precursors are selected at the end of MS1 to be introduced to MS2 is not trivial, and it will affect the interpretation of the final data: the two options are data-dependent acquisition (DDA) and data-independent acquisition (DIA), and they will be covered in the following section. All the above mentioned information obtained for each precursor is contained in the files produced by the mass spectrometer, which are to be provided to a proteomics software (in this case, DIA-NN [25]) that will return the information summarized at the precursor and protein level.

The way in which such proteomics software works will be detailed in Chapter 2, but an important point in this respect is that it requires the use of a spectral library, or a list of peptides or proteins from which one can be generated. It was mentioned before that an important advantage of mass spectrometry-based proteomics is that it can detect proteins even if no prior knowledge of them is available. This can be easily understood now that the procedure of such

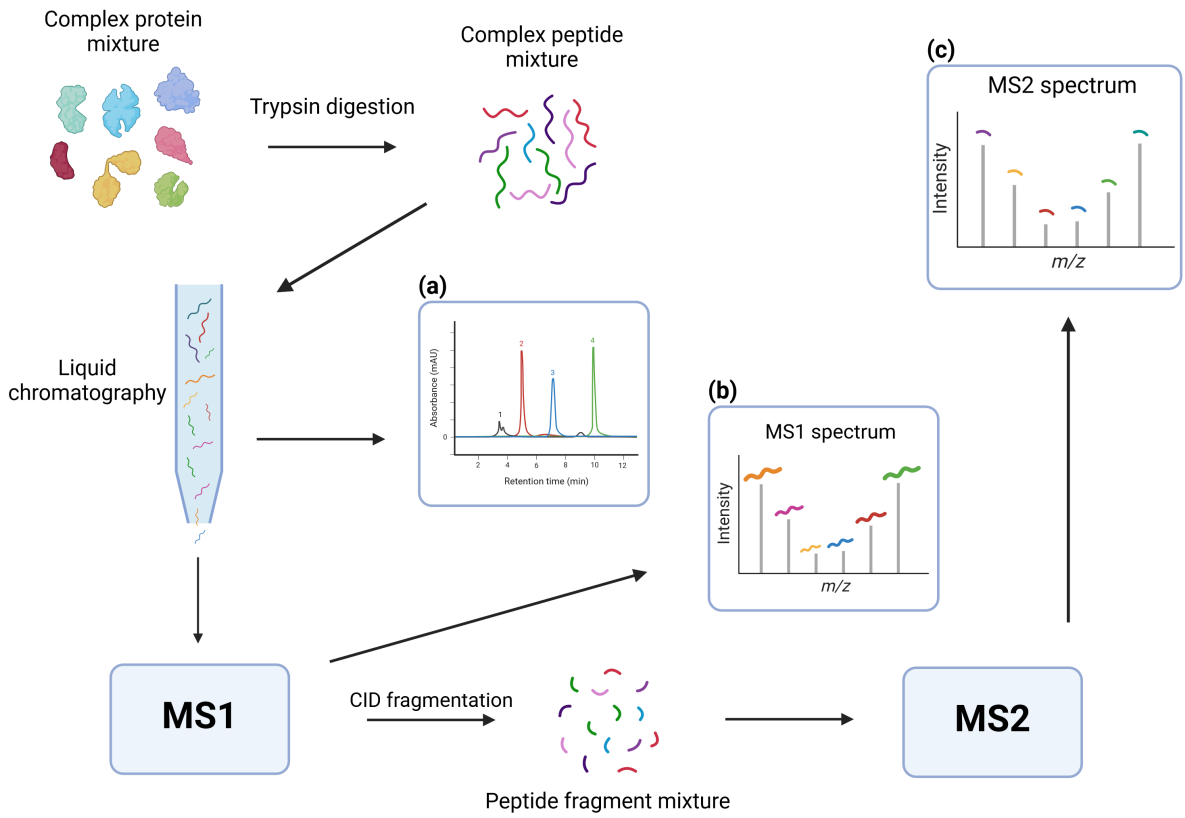

Figure 1.2: Simplified scheme of a LC-MS/MS procedure. Briefly, a complex protein mixture is digested with trypsin or another protease; the resulting peptides are separated by the chromatography step, and introduced into the first mass spectrometry step (MS1). As they come out of the MS1, precursors are further fragmented and introduced in the second mass spectrometer (MS2). Created with BioRender.

an experiment has been explained: no prior information is required by any of the steps, and all precursors derived from the proteins in the samples can be detected regardless of prior information being available on them. Nonetheless, the critical step for identification of precursors comes during the processing of MS files in the proteomics software. At this point, there are different approaches that can be followed in order to come up with the sequences of the detected peptides: database searching, spectral library searching and *de novo* methods. While the last focuses on the identification of previously unknown peptide sequences [26], the first two are both dedicated to identifying peptides based on previously available information, with spectral library searching being generally accepted as having a higher accuracy and sensitivity [27]. In the case of spectral library searching, spectra for the peptides that are expected to be found in the sample are usually directly provided to the software, however, current software also allows for the input of a set of proteins or peptides, that it then turns into a spectral library itself. This is the method that was followed in this project, providing DIA-NN with FASTA files containing the sequences of the peptides that were expected to be found in the samples.

1.4 Data Dependent Acquisition vs. Data Independent Acquisition

As indicated in the previous section, DIA and DDA are two different mass spectrometry techniques, and their difference lays in how the precursors that come out of MS1 are selected to be provided to MS2 [28].

As shown in Figure 1.3, in the case of DDA, only the most abundant precursors are selected and then individually introduced in the MS2. This provides high sensitivity and specificity for these highly abundant precursors, while also allowing for the MS2 spectra to be more easily interpretable, since they will contain fragments originating from a single precursor. However, DDA has the downside that many precursors from MS1 are ignored in this way, and hence much information is lost.

The basis of DIA is that all precursors in the MS1 spectra should go into the MS2. This is achieved by separating them in windows, and allowing all precursors contained in a certain window to go into the MS2 at once. It must be noted that while in Figure 1.3 it seems that these windows are defined based on m/z (the mass to charge ratio of the precursors) this is only one of many DIA techniques, known as SWATH (Sequential Window Acquisition of All Theoretical Mass Spectra) [30]. In fact, the technique that was used in this project in particular is known as PASEF (Parallel Accumulation–Serial Fragmentation) [31], and is characterized by its MS1 consisting of a trapped ion mobility spectrometry (TIMS) step. As described in the previous section, this means that precursors are separated based on their size, shape and charge in the gas phase, and accumulated, to then be sequentially let into the MS2. This parallel accumulation and serial fragmentation are part of what make of this technique such a useful tool, since they strongly increase the throughput of the method without missing almost any precursors along the way [31].

Therefore, the advantage of DIA over DDA consists in a significant reduction of information loss durint the analysis (in the form of MS1 precursors). However, as can also be observed from Figure 1.3 (b), this also causes MS2 spectra to consist of fragments of several different MS1 precursors, which left the MS1 at the same time and were thus fragmented and introduced in the MS2 togehter. This makes MS2 interpretation much more complex, since these spectra need to be deconvoluted first. In fact, there is also the possibility that co-eluting MS1 precursors (this is, precursors that leave MS1 at the same time) produce the exact same fragment in the MS2, which is known as interference and produces multiplexed spectra, which become even harder to deconvolute. Precisely the deconvolution of such MS2 spectra is

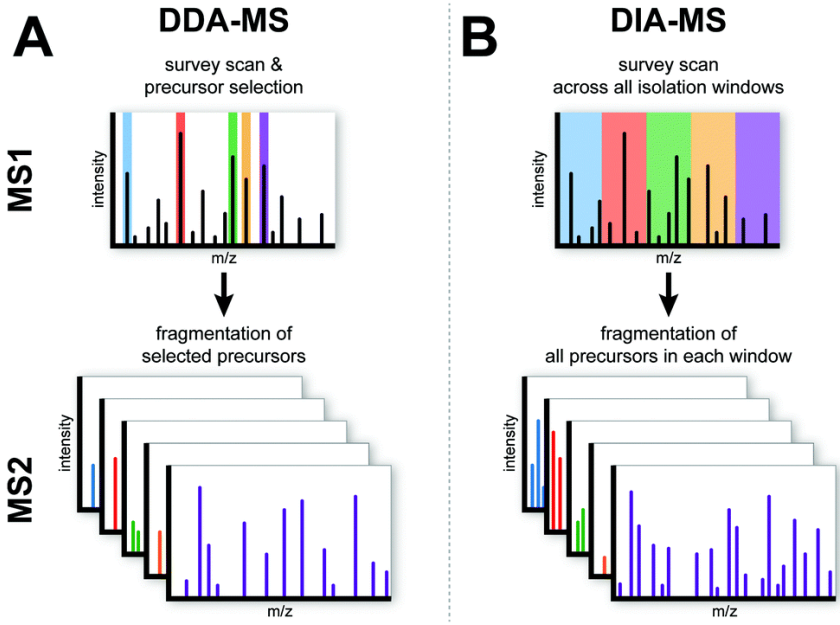

Figure 1.3: (A) shows the principle for DDA-MS, illustrating how individual precursors at the MS1 level are selected based on their abundance to be individually introduced in the MS2, producing simple MS2 spectra where all fragments are known to belong to the same precursor. (B) on the other hand illustrates the DIA-MS case, where precursors at the MS1 level are grouped based on a certain metric, and then introduced together into the MS2, producing more complex MS2 spectra. Figure obtained from *Data-independent acquisition mass spectrometry (DIA-MS) for proteomic applications in oncology*, by Lukas Krasny and Paul H. Huang, Fig. 1 [29].

This brings us to DIA-NN, Data Independent Acquisition Neural Networks [25], a software suite that uses neural networks to deconvolute DIA MS2 spectra, and which will be used in this project. It will be further introduced in Chapter 2.

To summarize, what limits protein identification in the case of DDA is precursor selection in MS1, while on the other hand, the limiting factor in DIA is the sensitivity of the mass spectrometer and the ability of the analysis software to deconvolute the signal in the MS2 spectra.

1.5 Strain-specific approach to the proteomic analysis

It was previously indicated that one of the main characteristics of proteomics software is that their performance is improved by providing them with a spectral library containing the sequences of the peptides that are expected to be present in the samples. Traditionally, such libraries would precisely be generated from the corresponding reference genome of the species, but as discussed in section 1.1, a reference genome cannot truly represent a full species, and is even less appropriate for this use as a spectral library in the particular case of the ScRAP [9], since there is such a wide variety of strains contained in it with such different backgrounds. Because of this, and especially in order to target some of the biological questions mentioned above, it was deemed appropriate that this proteomic analysis should be ran with strain-specific libraries. This will hopefully allow for much more accurate detection of strain-specific proteins that would go undetected were the analysis to be run with a single spectral library derived from the *S. cerevisiae* isolate S288c reference genome. Besides this, with an appropriate preparation of the libraries for the heterozygous diploid strains, it should be possible to make use of the phased haplotypes and recover information regarding allele-specific expression. In order to be able to compare the strain-specific proteomic analysis to the more commonly used reference proteome-based one, all samples were also run together in DIA-NN against a library

built from the reference genome of the reference *S. cerevisiae* strain, S288C. This approach is referred to in this thesis as "common approach".

Given the fact that this strain-specific approach is relatively new, there are also, apart from the necessary data processing and library creation, other questions that arise from it: the first and most obvious one is if this approach truly results in a significantly increased number of protein identifications as compared to the common approach mentioned above. However, another important question is regarding the processing of the proteomics software's output at the precursor level: this output contains information from all precursors identified in the samples (the structure of this output will also be covered in depth in Chapter 2), and needs to be filtered before a peptide-to-protein quantification is performed to obtain the final version of the data, so that protein identifications are reliable. This processing has been extensively performed before for typical "common approaches" where all samples in an experiment are ran through DIA-NN together with a library generated from a reference genome, however the same cannot be said for this strain-specific approach. Hence, it will have to be evaluated how to adapt the steps of this processing to the strain-specific approach. At the same time, since the steps of this processing are dependent on the origin and quality of the data, as well as on the objective of the study, the processing of the common approach data is also presented in this report, and will serve as a basis that will then be modified as necessary towards the strain-specific approach.

1.6 Research questions

Multiple questions, both at the methodological and biological levels, are addressed in this thesis. First, the strain-specific approach is set up, which includes the creation of the strain-specific files that are to be used as spectral libraries for each strain. This procedure is described in Chapter 2.

Subsequently, the first methodological questions are addressed: firstly, what is the appropriate way of processing the data at the precursor level in the common approach, and how should this processing be adapted to the case of the strain-specific approach? Secondly, does the strain-specific approach truly result in an increased number of protein identifications as compared to the common approach?

Finally, two of the biological questions that arised from the ScRAP [9] and that were mentioned in section 1.2 are targeted as well. The first of these is related to allele-specific expression: libraries for heterozygous diploid strains were built with this in mind, labelling proteins with different alleles between the two haplotypes of a strain so they could be differentiated. The process for creation of these libraries is detailed in Chapter 2. This should allow to detect the amount of a certain protein that is produced based on each of the two sets of chromosomes, and hence evaluate whether one of the haplotypes is dominant over the other, or if any other patterns can be observed at this level. A schematic representation of this biological question is presented in Figure 1.4. Finally, I also evaluate in this thesis the effect of insertions and deletions on protein expression. This is, based on information obtained by our collaborators at the genomic level, we knew which strains contained exactly which deletions and non-coding insertions in which of their genes. Consequently, for each protein that contained one of these mutations in at least one strain, the quantification values for the protein were grouped on the one hand for the strains carrying the mutation, on the other hand for the strains not carrying the mutation. Finally, these values were turned into binary data, representing whether the protein was detected or not in each sample, and a proportion test was performed between the two groups described above. The objective behind this was to evaluate whether deletions and non-coding insertions affect the expression of the protein encoded by the section of the DNA where they appear

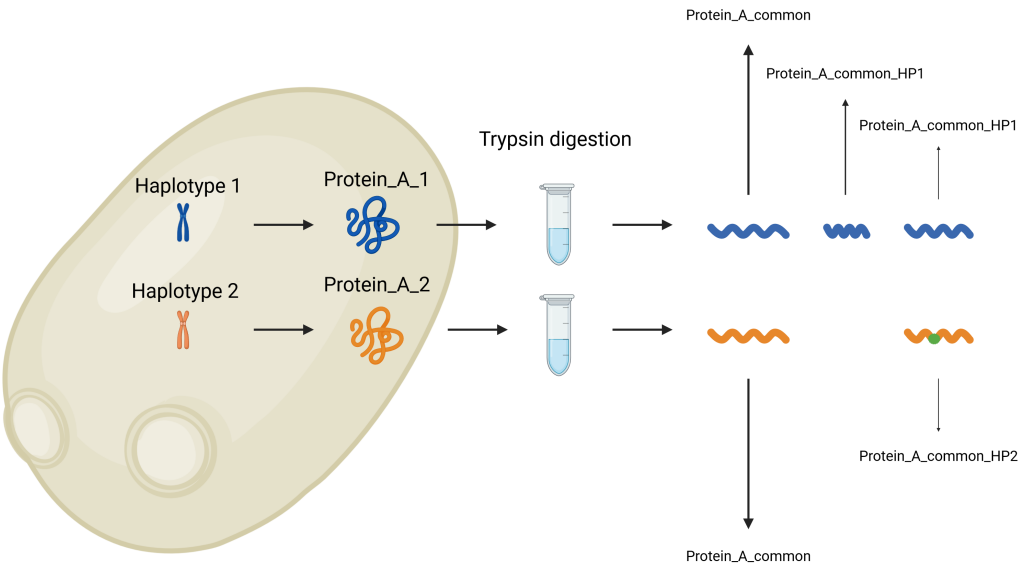

Figure 1.4: Scheme representing the principle behind the allele-specific expression question. Inside the *S. cerevisiae* cell of a heterozygous diploid strain, two different versions of the same proteins are produced, each from one of the haplotypes. The differences between these two versions of the protein are evidenced when performing an in-silico digestion of their sequences with trypsin: we observe that the first peptide obtained from this fragmentation is exactly the same between the two versions, while the second one is only present in one of them, and the third one is present in both of them but contains a mutation (represented as a small green dot) in the case of haplotype 2. These peptides are assigned the names observed next to them by following the principle presented in Figure 2.2. It must be noted that some peptides receive the exact same name because as described in the aforementioned section, peptides are labelled with the name of the protein from which they come, all of them with exactly the same name, as DIA-NN [25] is able to interpret them in this way. Created with BioRender.

Chapter 2

Materials and methods

2.1 Data

2.1.1 Proteomics raw files acquisition

We received 134 of the 142 strains belonging to the ScRAP from our collaborator, then randomized them to six 96-well plates, with four replicates of each strain. Randomization of the samples to the wells was performed in R [32], and made effective through the use of the Singer PIXL. These plates also contained 30 replicates (five per plate) of a laboratory strain (BY4741-ki), extremely similar to the reference *S. cerevisiae* strain S288C, as well as ten empty wells across the six plates. Hence, all 576 wells across the six plates were occupied. The final distribution of the samples across the plates can be seen in Figure 2.1.

These strain isolates, once randomized to the different wells throughout the plates, were grown on synthetic minimal medium (SM medium, as described in [6]) on agar for 48 hours, followed by a liquid overnight culture also in SM medium. Optical density (OD, at 600 nm) measurements of the cultures were then performed, known from this point on as the pre-culture OD. Afterwards, these pre-cultures were back-diluted 10x in SM medium (140 μL overnight culture + 1400 μL SM), and the resulting samples were cultivated at 30ºC with shaking. After 9 hours it was deemed that cells had reached the exponential growth phase, their OD was then measured again (referred to as harvest OD) and 1.2 mL were harvested from each well. The cells were centrifuged, the supernatant discarded, and the pellets were frozen at -80ºC.

In order to prepare the samples for mass spectrometry, cells were resuspended and their lysis was performed with a 200 μL 7M urea lysis buffer, followed by 2 cycles of genogrinder: samples were placed in new plates, with each well in the plates containing a borosilicate glass bead. Plates were then covered and centrifuged for 5 minutes at 1500 rpm, followed by 5 minutes of rest on ice, and this process was repeated twice. Pellets were subsequently resuspended, and protein digestion was performed in a solution of 2M urea, using 2 μg trypsin/LysC per sample. The following day, trypsin was inactivated by adding formic acid, and samples were run through solid phase extraction (SPE) columns in order to isolate the peptides and remove all other substances. After this, peptides were dried as all solvent was evaporated and resuspended. A pool of all samples was created to be used as a technical control. The peptide concentration of this pool was determined via a fluorimetric assay, and its OD measured. Based on this and on the OD measurements of the samples, the peptide concentration of the samples was interpolated.

Finally, based on the estimated concentration of each well after resuspension of peptides, samples were taken from each of them containing 2 μg of peptides, and these were analyzed on a TimsTOF HT Pro3 mass spectrometer, with a 5 minute active gradient and a technical control (a small aliquot of the sample pool) being ran every 30 samples.

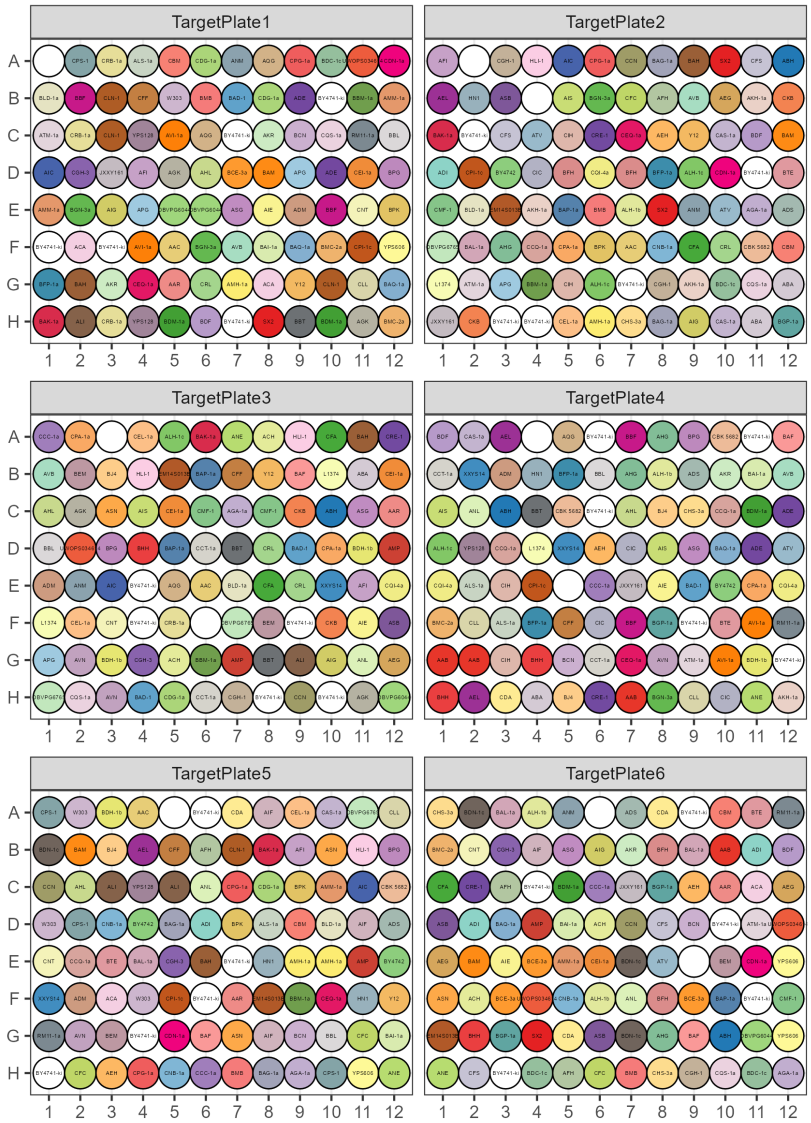

Figure 2.1: Scheme representing the distribution of the 576 samples to the wells across six 96-well plates.

2.1.2 Telomere-to-telomere proteomic assemblies

Single-molecule long-read sequencing technologies allow to obtain gapless genome assemblies, which over the last few years has contributed to a great increase in quality and contiguity in the reference genomes of multiple model organisms, as well as humans [9]. This technology was used by our collaborators to perform the sequencing of the 142 strains in the ScRAP, hence expecting to cover the entire genomic space of the species [9]. In the case of heterozygous diploid strains, these genomes were also subjected to haplotype phasing. Briefly, haplotype phasing is an algorithmic procedure that allows to resolve to which of the two haplotypes of a certain strain each genomic sequence belongs, and hence seamlessly reconstruct the two haplotypes separately. Haplotype phasing was also performed for the triploid and tetraploid strains, but without the same level of success. As a result, the sequences coming from the different haplotypes during the sequencing process were collected together into a single file, referred to as a "collapsed" genome.

Hence, we received from our collaborator a set of FASTA files containing haplotype-resolved and/or collapsed telomere-to-telomere genome assemblies for the 142 strains in the ScRAP, both for the nuclear and mitochondrial genomes. Apart from these, we received another version of these files, where the genomic sequences were translated to protein sequences, and each protein annotated with its systematic name, although annotation procedure was not perfect. One of these genome-derived protein sequence files (GDPFs) was received for each strain, except for heterozygous diploid strains, for which a file was received for each haplotype. I processed these files and generated a new set of FASTA files that were to be used as reference libraries to run the DIA-NN software in a strain-specific manner; this is, processing the mass spectrometry files corresponding to the replicates of each of the strains separately with their individual strain library. The processing consisted mainly of bringing together haplotypes in the case of heterozygous diploid strains, and dealing with the collapsed assemblies for polyploid strains, and will be covered in the next section. Details regarding the methodology used are available in section 2.2.1.

2.1.3 Structural variants and heterozygosity information

With the goal of targeting some biological questions based on the processed proteomics data, some extra files were provided by our collaborator. These included files with information on which structural variants (SVs) were found to be present in each strain, at which location in its genome and affecting which genes, as well as more information regarding these SVs. A total of 4809 SVs were found, each affecting usually more than one of the strains.

2.1.4 Databases

As mentioned above, most of the reference spectral libraries used during this project were generated from the GDPFs obtained by our collaborator [9] (as obtained from `https://www.evomicslab.org/db/ScRAPdb/`, accessed on 05.05.2024). However, in some instances, the S288C reference genome as obtained from UniProt [33] (accessed on 18.06.2024) was used, and protein annotations were obtained, when necessary, from the Saccharomyces Genome Database [34] (accessed on 10.06.2024).

2.2 Methodology

2.2.1 FASTA files preparation

As already covered in the previous section, there were three different types of strains among the ScRAP strains:

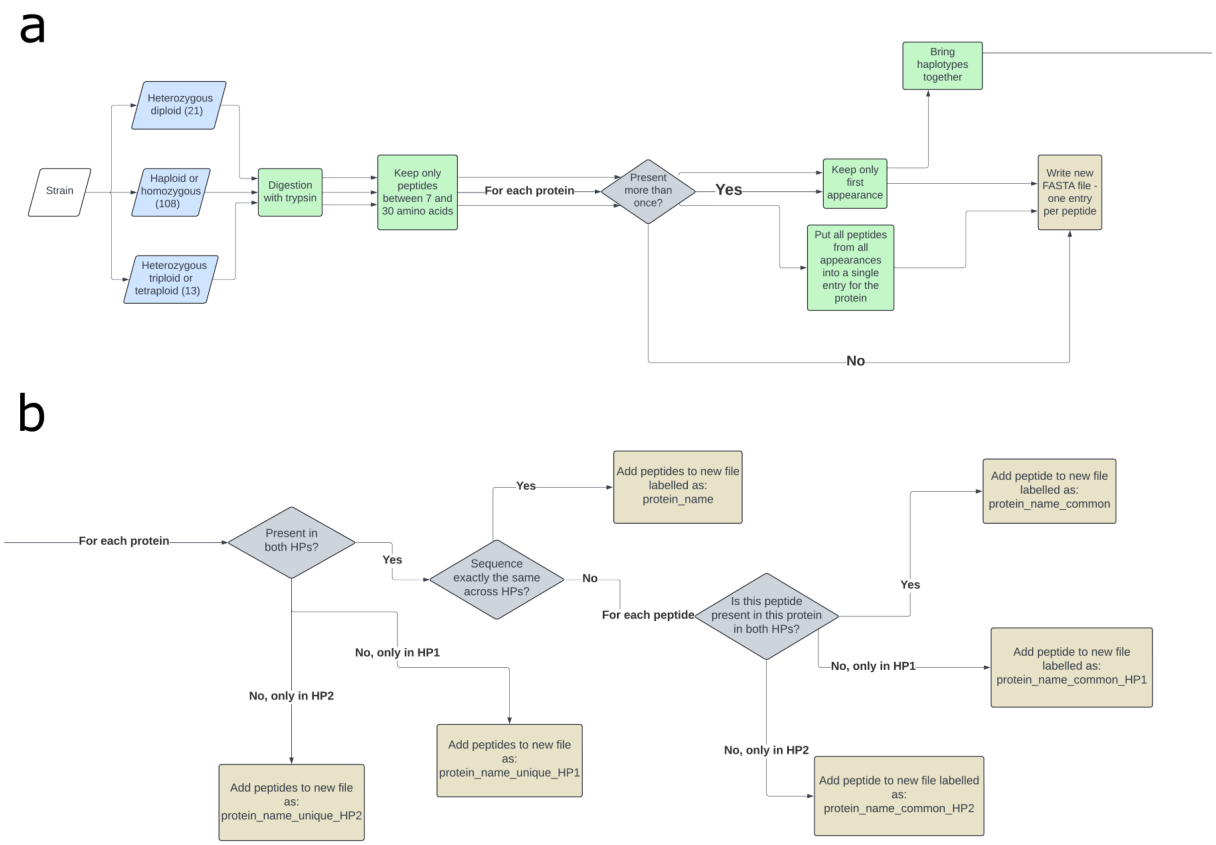

Figure 2.2: Two schemes representing the decision tree used to generate strain-specific FASTA files from the GDPFs, to be used as spectral libraries. It is noteworthy that the starting files contain a full protein sequence in each of its entries, while the final files contain a single peptide sequence in each of its entries, where peptides originating from the same protein have the same header (except in the special case of some proteins in the heterozygous diploid strains). (a) shows the full processing steps for haploid and homozygous strains, as well as for heterozygous triploid and tetraploid strains, in addition to the processing of heterozygous diploid strains that is common to the previous two. (b) is a continuation of (a), which shows the further processing that is specific to heterozygous diploid strains.

- Haploids and homozygous: a single GDPF was received for each strain.
- Heterozygous diploids: genomes from each haplotype were succesfully phased, resulting in a separate GDPF available for each of them.
- Polyploids: haplotype phasing was not successful, a single GDPF with all proteins identified in the strain was provided, without them being linked to a particular haplotype.

The processing undergone by these files was minimal in the case of haploid/homozygous and polyploid strains, while more complex in the case of the heterozygous diploid strains. The steps common to all strains are described below, and are also shown in Figure 2.2:

- Perform in silico tryptic digestion of each protein sequence.
- Remove any generated peptides not between 7 and 30 amino acids in length (due to the settings used during mass spectrometry we know they could not be detected).
- If 2 protein sequences are present in the file that were annotated with the same protein name, only the first appearance is kept, while the second one is saved in a separate file for later reference. It must be noted that in most cases the differences between sequences were minimal.

- Each obtained peptide sequence was written to the new version of the file with its header being the name of the protein that it originated from. This is, so that all peptides coming from the same protein had exactly the same header. This was necessary towards the use of these files in DIA-NN.

While the following steps were unique to the corresponding strains:

- Heterozygous diploid strains: after the steps presented above, the two haplotypes had to be brought together into a single file (described in Figure 2.2 (b)):
  - For proteins present in both haplotypes and with the exact same sequence across them, their peptides were annotated with just the protein name, as explained above.
  - For proteins present in both haplotypes but with different sequences across them, their peptides were respectively tagged as: ProteinName_common (if the peptide was present in both haplotypes), or ProteinName_common_HP1 or ProteinName_common_HP2, if the peptide was present only in haplotype 1 or haplotype 2, respectively.
  - For proteins present in only one haplotype, their peptides were tagged as Protein1_unique_HP1 or Protein1_unique_HP2 respectively.
- In the case of triploid and tetraploid strains, the only difference with the general procedure described above was that when a protein was repeated within the GDPF, we got its peptides that were not already present in the first appearance of that protein in the new file, and added them to it. The goal of this is to be able to detect any of the copies of the protein, despite not knowing which haplotype it came from.

At this point, the files are ready to be used by DIA-NN.

During this processing, information regarding the theoretical number of proteins present in each strain was collected. In addition, the proteins were compared to those of the reference strain S288C to identify proteins unique to each strains or potential difference in their sequences. This information is reflected in Figure 2.3.

2.2.2 DIA-NN software

DIA-NN (Data Independent Acquisition Neural Networks) [25] is a software suite for the analysis of DIA data which, through the usage of neural network ensembles, is able to deconvolute multiplexed DIA MS2 spectra, hence providing reliable, robust and quantitatively accurate interpretations of these data.

2.2.2.1 Algorithm

DIA-NN was originally published in 2020, and has since then underwent several improvements and updates as part of new version releases. However, the basic features which make it such a useful tool in the analysis of DIA data remain the same. Before briefly going into the algorithm itself, it is important to note that this software frequently works at the level of MS1 precursors, meaning that these should be defined: a precursor is a peptide with a certain charge, as it goes into the MS2. Hence, for a certain peptide (this is, simply a certain sequence of amino acids) multiple precursors can exist, since each peptide can usually acquire different charge states.

The DIA-NN algorithm is based on a target-decoy approach: apart from the proteomics raw files it must be provided with either a spectral library or a set of proteins or peptides

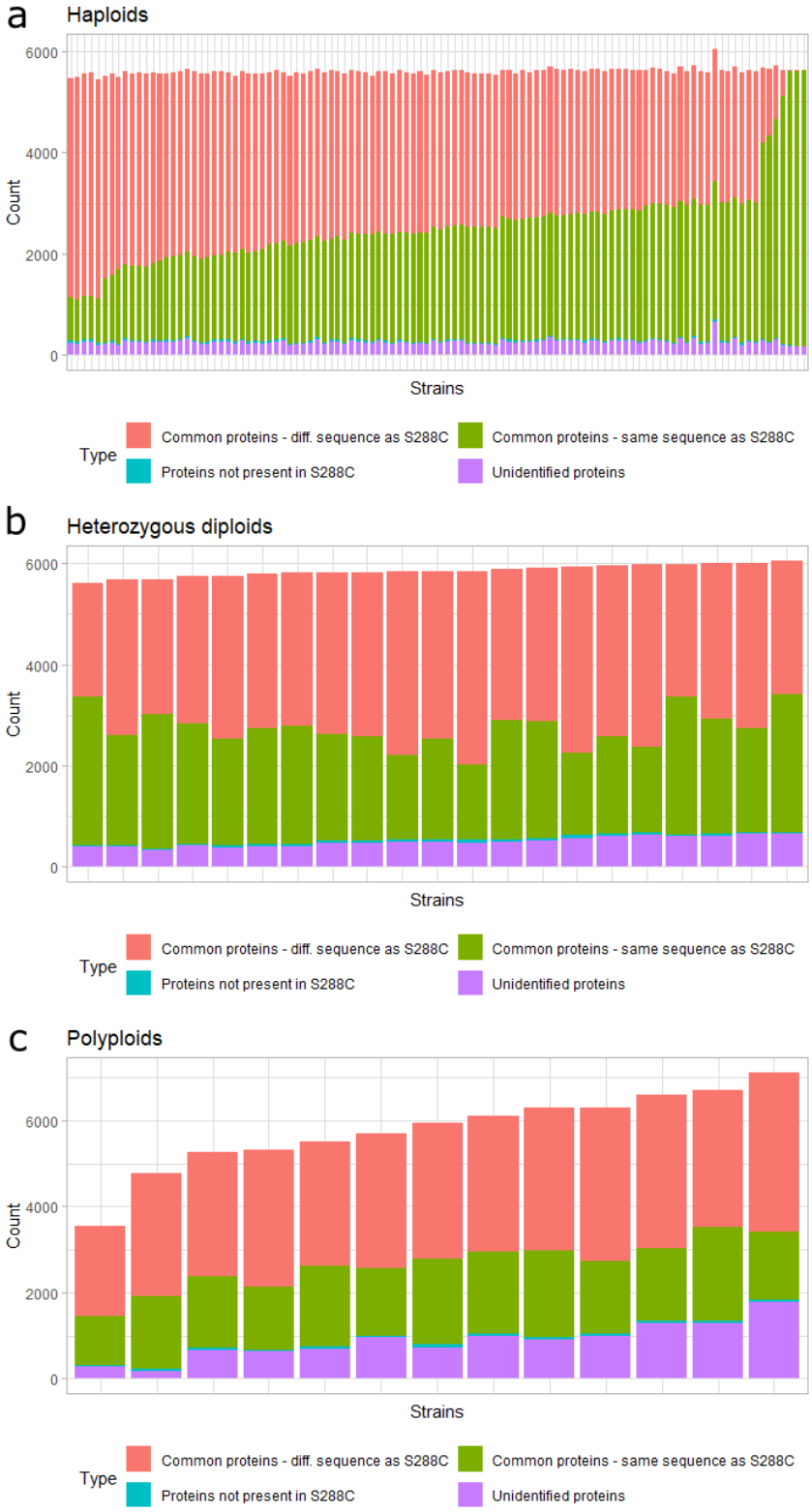

Figure 2.3: Barplots showing the abundance of different types of proteins in each strain in the ScRAP based on the GDPFs, as per their presence in the reference strain S288C: for each non-reference strain, proteins were counted that were present in S288C with the exact same sequence as in that strain, those that were present in S288C but with some difference in their sequence, those that were not present in S288C at all, and those proteins in the strain that were simply not identified. (a), (b) and (c) contain this information respectively for haploid, heterozygous diploid, and polyploid strains. Of particular interest are the proteins common with S288C but with a different sequence, which are present in large quantities in most strains, and whose identification and quantification might be improved by the strain-specific approach.

from which one can be generated, which was the case here. For each peptide in the library, a decoy peptide is generated, following a certain mutation pattern and keeping similar physicochemical properties. For each of these theoretical peptides, regardless of them being target or decoy, a theoretical fragmentation is performed, simulating that which would take place in the experimental setting. Then, out of the resulting theoretical fragments, one is selected to be representative of this peptide, based on various metrics. For each theoretical peptide, the fragment selected as representative is then compared to the real, observed fragments in the experimental MS2 spectrum at the corresponding RT and m/z, and the match between the theoretical fragment and each of the observed peaks is characterized by 73 scores (described in detail in the Supplementary Materials of [25]).

These 73 scores are then provided as input for an ensemble of 12 deep, feed-forward, fully connected neural networks. These consist of 5 tanh hidden layers, with the ith hidden layer containing 5·(6 - i) neurons, with i = 1, ..., 5, and a final softmax output layer. Cross-entropy is used as the loss function. These neural networks are trained for a single epoch to produce an outcome in the 0-1 range for each set of 73 scores provided, which represents the likelihood of the corresponding theoretical peptide being a target peptide. The 12 values coming from the different neural networks for the same theoretical peptide are averaged, and this final value for each peptide is what is used in order to calculate the Q-values. FDR is conservatively estimated as presented in Equation 2.1.

$$FDR = \frac{Decoy\ peptides}{Target\ peptides} \tag{2.1}$$

For inference at the protein level, only target precursors which are proteotypic (this is, that are specific to that concrete protein) are considered, so proteins without any proteotypic precursors identified automatically receive a Q-value of 1.

It must be noted that no batch normalization or dropout were used in the neural networks, at least in the original version of the software, since they did not seem to improve its performance [25]. It is also noteworthy that the values specified in the previous paragraph (number of DNNs in the ensemble, number of layers and of training epochs) are the default parameters, which can be modified, although for this project they were kept at these defaults.

In the original publication it was stated that regarding quantification of each precursor, DIA-NN estimated the intensities of all fragment ions associated to it by using an interference-removal algorithm. An advantage of this algorithm was that it did not depend on the spectral library in order to come up with a reference intensity value for each fragment, so its performance was independent of the quality of the spectral library provided. However, the method used to quantifiy each precursor involved, to explain it very briefly, bringing together the information from several of its fragment ions, which were selected in a cross-run manner, and summing their respective signals in each run. As stated above, this approach allowed to get rid of signals that were strongly affected by interference, however it was still subject to errors in individual acquisitions, and most importantly, it was realized that it discarded potentially useful information, mainly that obtained at the level of the MS1 for the full precursor. This is why QuantUMS [35] was developed: an improved version of this algorithm, which now brings together the information for a precursor at MS1 level and for its fragment ions at MS2 level in order to produce more accurate precursor quantifications.

Finally, another interesting feature of DIA-NN is the match-between-runs (MBR) mode. This consists in, for each sample that is processed by the software, creating a corresponding empirical spectral library with all the peptides found in the sample. This is done for all samples within an experiment, and this empirical spectral libraries are brought together into a single experiment-wide spectral library, which can then be used to run all samples against it again.

This allows for high sensitivity and the ability to detect any peptide that is abundant in at least one of the samples, in any other sample even at low amounts.

2.2.2.2 Running DIA-NN

DIA-NN can be ran both from its own GUI or from the command line, which was the case for this project. Mass spectrometry files were provided as .d directories, each directory containing multiple files in different formats, containing the information for one sample. With respect to the spectral libraries, these were provided as FASTA files, as covered in section 2.2.1.1. As already explained, two different approaches were taken when running DIA-NN: first was the so-called common peptide approach (CA), where all samples from all strains were ran with the same spectral library, coming from the reference genome of the reference *S. cerevisiae* strain. Secondly, for the strain-specific approach (SSA), the samples from each strain were ran separately, against a library built specifically for that strain, based on its sequenced genome.

Many different parameters and options are available when it comes to performing an analysis using DIA-NN, but here I will summarize the values that were used for this project: minimum and maximum peptide lengths were set to 7 and 30 amino acids respectively, since it was known from mass spectrometer technicians that this is the range of peptides that are detectable for such an experiment as was performed here. Missed cleavages were set to 0. Minimum and maximum precursor charges were set to 1 and 4 respectively.

Out of a single DIA-NN run, multiple output files are generated: the structure of the main report consist of one entry per row, corresponding to a specific precursor in a given sample, with columns specifying the sample and precursor IDs, as well as other characteristics such as charge state, stripped peptide sequence, different types of Q-values (at the precursor level, protein level, etc.), and of course columns with the quantification values for each precursor, both raw and normalized. This is just a brief overview of the main columns of the report that are used as part of the present analysis, but the full description of the report columns can be found at `https://github.com/vdemichev/DiaNN?tab=readme-ov-file#main-output-reference`. Alongside the main DIA-NN report, other output files are produced. Among them, the "unique_genes" file is based on the general report presented above, but information is collapsed at the protein level, so that this file contains a protein in each row and a sample in each column, ready for further analysis. This file is produced from the main report by simply removing all non-proteotypic precursors and using the maxLFQ [36] algorithm for peptide-to-protein quantification. This is important since it is one of the goals of this project to show how this process is improved by further filtering at the precursor level, prior to peptide-to-protein quantification, and how this allows for more confident and robust protein identifications. The last of the output files produced by DIA-NN that will be covered here is the "stats_file". This file contains each of the samples in the DIA-NN run as a row, with the columns containing different statistics for them, such as the total amount of precursors detected in that sample, the total MS1 signal as well as the total MS2 signal, the total count of ions detected in the sample, and so forth.

Regarding the maxLFQ algorithm, it must be noted that apart from being automatically used by the most recent versions of DIA-NN to create the "unique_genes" file, it is also the algorithm of choice for peptide-to-protein quantification throughout this project, thus it is deemed necessary to briefly introduce it. MaxLFQ is a popular generic algorithm for label-free protein quantification which is generally applicable to proteomics data, and which solved prior issues of this type of quantification. Before its publication, stable isotope-based labeling methods were the reference when it came to protein quantification, and the available software for label-free quantification was either created to function only in very specific experiments under concrete experimental conditions, or simply didn't provide such accurate quantification. MaxLFQ solved these issues by, briefly, performing quantification based on bringing together

peptide signals available in a number of different samples, as well as by introducing a "delayed normalization", which makes it compatible with any experimental separation technique employed [36]. MaxLFQ is available as part of the MaxQuant software, but also as a function within the DIA-NN R package, which was the one used in this project.

2.2.2.3 Processing DIA-NN output

As mentioned above, one of the goals of the present project was to compare the results from running DIA-NN in a common vs. a strain-specific manner. However, in order to do this, it was first necessary to perform a proper pre-processing of DIA-NN output at the precursor level prior to peptide-to-protein quantification, with the goal of posterior protein identifications being more reliable. This pre-processing of DIA-NN output a the precursor level has been extensively performed before, consequently it was only adapted to the present study in the case of the common approach. However, due to the strain-specific approach being more of a novelty, more importance fell on the task of adapting this pre-processing to the strain-specific setting. The final pipelines for both approaches will be covered in Chapter 3.

2.2.3 Software versions

This project was performed using DIA-NN version 1.8.1 [25], R version 4.3.2 (2023-10-31) [32] and Python version 3.12.2. [37].

Chapter 3

Results

3.1 Processing of DIA-NN common approach output

The steps of this processing are summarized in the scheme in Figure 3.1. In the following sub-sections the reasoning behind each step, as well as how they were performed, will be covered. The following section will deal with the adaptation of these processing steps towards the strain-specific approach. However it is important to first stress how this processing pipeline was developed for this particular dataset, with a certain reasoning in mind at each step, and that it is important to make such considerations again when adapting it to a new dataset or project, since parameters might need to be changed, some steps dropped altogether and others included, all depending on the origin of the data and the goals of the analysis, as mentioned in the introduction.

3.1.1 Remove empty or low OD samples

As covered in the materials and methods, the OD600 was measured for each sample both after the pre-culture and at harvest time. This measure is considered as a good proxy of cell growth in a culture, so it provides important information on the amount of proteins that could potentially be found. Hence, it was deemed appropriate to, first of all, remove from the dataset the samples with extremely low OD values at harvest time, since this indicates a lack of cell growth. Figure 3.2 (a) shows boxplots for the OD of each strain, and based on this observation and on prior knowledge from the team, the decision was made to set the minimal threshold for the OD at 0.12. This led to the removal of 15 strains from the data, with a total of 72 samples.

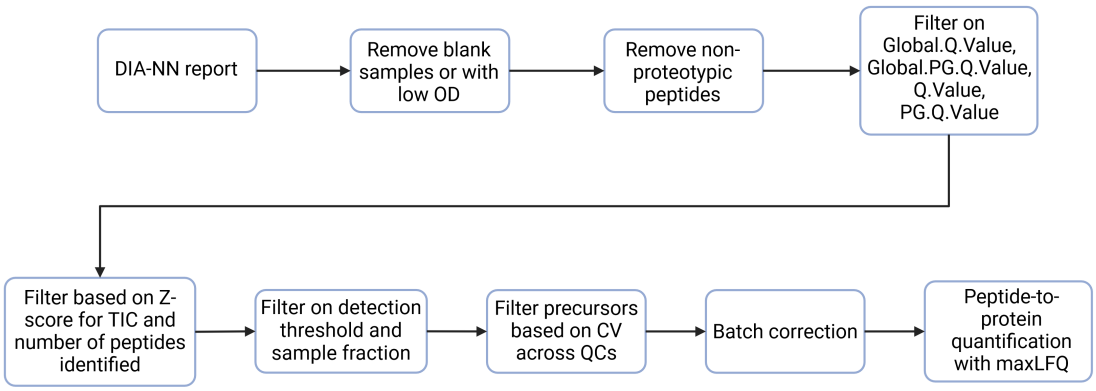

Figure 3.1: Scheme showing the steps of the processing underwent by the raw DIA-NN report from the common approach, up to the point of peptide-to-protein quantification, which will later be adapted to the strain-specific approach.

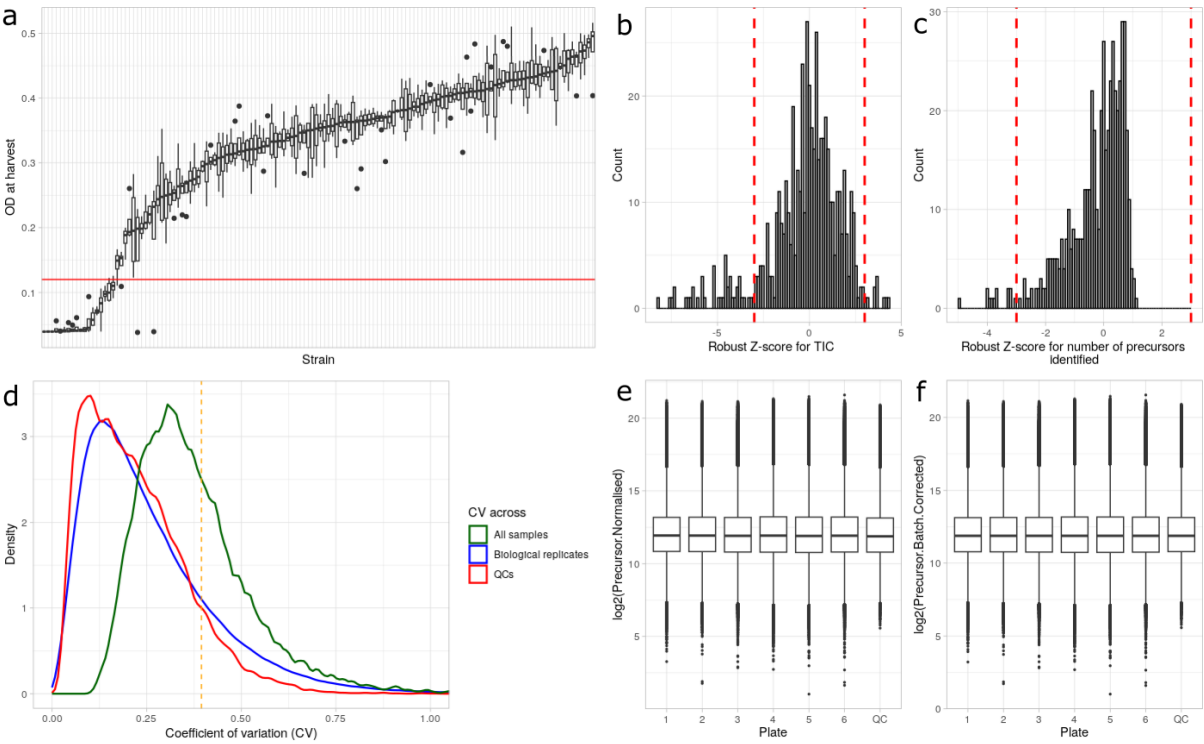

Figure 3.2: Figure containing plots corresponding to different steps of the processing of the DIA-NN report from the common approach: (a) contains boxplots for the OD measured at harvest for each strain, with a horizontal red line at OD = 0.12, where the cutoff was set for discarding samples below this value. (b) and (c) show, respectively, the distribution of the robust Z-scores for total ion count (TIC) and number of identified precursors for all samples remaining after the previous steps. Vertical dashed red lines represent the cutoffs, at -3 and 3 in both plots. (d) contains the density plots for the coefficient of variation (CV) calculated for each precursor left in the report in 3 manners: across all samples (green curve), across biological replicates (in blue) and across quality control samples (QCs, in red). The dashed orange vertical line represents the top 10th percentile of the red curve, which serves as a threshold for removal of all peptides above it from all samples in the dataset. (e) and (f) show boxplots per plate for the $\log_2$Precursor.Normalised before and after batch correction respectively, showing the quality of the dataset and the lack of batch effects.

It must be noted that following this step, all non-proteotypic precursors were also filtered out. This is a step that is common to many such processing pipelines, since later protein quantification based solely on proteotypic peptides will provide more reliable quantifications.

3.1.2 Remove precursors with non-significant Q-values

Out of the multiple Q-values present in the DIA-NN report, this filtering step focused on four of them:

- Q.Value: Calculated separately for the precursors in each sample, one Q-value being assigned to each precursor. These Q-values are assigned after ranking all precursors in the sample based on the score produced for them by the ensemble of neural networks, which represents their likelihood of being a target precursor, as opposed to a decoy.
- PG.Q.Value: Calculated at the Protein Group level, also separately for each sample. This means that the precursors corresponding to a certain set of proteins that are considered to have closely related sequences are grouped together, and the same Q-value is assigned to all of them. Non-proteotypic precursors are included in this case as well.
- Global.Q.Value: Calculated over all precursors across all samples in the DIA-NN run, again at the precursor level.
- Global.PG.Q.Value: Again at the Protein Group level, but in this case over all the samples in the DIA-NN run.

Filtering was performed for these four types of Q-values at $\alpha = 0.01$, so as to maximize the robustness of the protein quantifications.

3.1.3 Filter based on TIC and number of identified peptides

This step was performed based on the stats_file, where each row is a brief summary of each sample in the experiment. One of the columns in this file is the total ion count (TIC) that was detected in each sample, and which represents the total amount of peptides present in the sample, both identified and unidentified. It is a measure of the total protein or peptide amount in each sample. The second parameter used here is the number of identified peptides, which doesn't depend only on the sample and the instrument used anymore, but also on the spectral library used.

In order to remove outlying samples, with extremely high or extremely low protein concentration, a robust Z-score was calculated for each sample for each of these 2 variables, according to Equation 3.1:

$$Robust\ Z - score_i = \frac{X_i - median(X)}{MAD} \tag{3.1}$$

Where:

$$MAD = median(X) \cdot |X - median(X)| \tag{3.2}$$

The robust Z-score was used instead of the traditional Z-score due to the fact that the TIC and the total number of identified peptides can take quite extreme values in outliers samples. Thus, it was deemed appropriate to use a more robust version of the score, which uses the median instead of the mean and is hence not so affected by these outliers.

Samples were removed that had robust Z-scores over 3 or below -3. This step allows for the removal of samples with extremely large or small amounts of protein detected. It must be noted that only samples with extremely low TIC and number of peptides identified were filtered out at this step.

3.1.4 Filter based on detection threshold and sample fraction

As was covered in Chapter 2, there were initially four replicates for each of the strains in the experiment. However, the filtering performed in the previous steps of the processing might have caused this number to drop in the case of some strains. Hence, in this step, any strain with less than three replicates left was dropped, as a lower number of replicates would not provide enough information nor statistical power, or be properly representative of the strain.

Subsequently, for each strain, precursors which were not present in at least 65% of the samples (this is, in 3/4 or 2/3 samples) were also dropped, in order to make the quantification of each protein within each strain even more robust.

3.1.5 Filter based on coefficient of variation

The coefficient of variation (CV), as defined in Equation 3.3, was calculated for the normalized quantity of each precursor in the report in three different ways: across all samples, across biological replicates (samples belonging to the same strain) and across quality control samples (QCs).

$$CV = \frac{\sigma}{\mu} \tag{3.3}$$

The distribution of the CVs of these 3 different types can be seen in Figure 3.2 (d). The first noticeable characteristic of this plot is that the curve for the CV across biological replicates is quite close to that for the QCs, suggesting that the samples belonging to the same strain are indeed similar to each other, which indicates that the preparation and processing of the samples were correctly performed, introducing only a minimal amount of technical variability between them. The curve for the CV across all samples is, as expected, shifted to the right, since it contains as well the biological variability across the different strains.

The goal of this step is to remove precursors with a high technical variability associated to them, which would make them highly variable across samples without any association to the biological signal, and which might hence confound the final results at protein level and complicate their interpretation. The assumption made is that, since the different QC samples are different aliquots of the same mixture ran on the mass spectrometer at different times, the variability across them should be minimal. Hence, the precursors in the higher 10th percentile of the CV across QC samples were identified, and eliminated from all samples in the experiment, with the intention of reducing this technical variability.

3.1.6 Batch correction

Samples in this experiment were contained in six different 96-well plates, which were ran in the mass spectrometer in two batches. Consequently, it was decided to evaluate batch effects at the plate level. As can be seen in Figure 3.2 (e), no significant differences are observed at the level of the normalized quantity of precursor detected coming from each plate. Still, median normalization was performed, where the normalized precursor quantities coming from each plate were multiplied by the ratio between that plate's median quantity and the quality controls median quantity. As can be seen in Figure 3.2 (f), this did not cause any noticeable difference

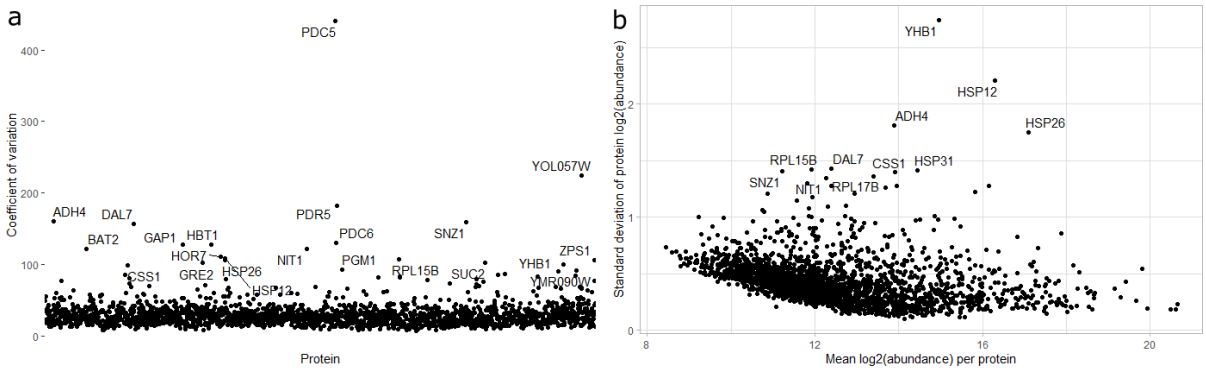

Figure 3.3: Assessment of protein abundance and variability across all samples after processing and peptide-to-protein quantification of the common approach dataset.(a) shows all proteins across the X axis, ordered alphabetically, and their coefficient of variation across all samples in the Y axis. The vast majority of proteins have a relatively low CV, only those above 100 are labelled with their name. (b) shows the mean $\log_2$(abundance) for each protein across all samples in the X axis, against the standard deviation of this $\log_2$(abundance) on the Y axis. A light trend can be identified of more abundant proteins having a lower standard deviation.

with respect to the distribution observed before batch correction. We also did not notice any additional batch effect associated with the mass spectrometer batches.

3.1.7 Peptide to protein quantification - maxLFQ

Finally, after extensive filtering at the precursor level, these were used for protein quantification, which was done with the maxLFQ algorithm [36] directly in R using the *diann* R package [38].

3.1.8 Resulting dataset

After these steps, the resulting dataset at the protein level contained 2329 proteins and 432 samples, corresponding to 104 strains. Some exploration into the features of this dataset was performed, and although it cannot be fully included here due to it not being the main topic of this thesis, some observations are highlighted in Figure 3.3. Figure 3.3 (a) shows the CV across all samples for each protein, where it can be observed that the majority of proteins have a relatively stable presence across the different strains, while some others such as PDC5, an isoform of the pyruvate decarboxylase, a key enzyme in alcoholic fermentation, show clear variation in their abundance. Such variations suggest that these proteins´ abundances might be tied to differences in the strains metabolism or their natural ecological niches. 3.3 (b) shows a light tendency of less abundant proteins to being more variable in their abundance, which is suspected to be due to the mass spectrometer being able to perform more accurate quantification at higher abundances and is consistent with prior observations. However, proteins with high abundances and high variability such as HSP12, HSP26, YHB1 or ADH4 are likely to have biological significance in the strains in which their abundance varied.

3.2 Processing of DIA-NN strain-specific approach output

The processing steps presented in Figure 3.1 were evaluated regarding their relevance to the strain-specific DIA-NN reports, as compared to the common approach one, and it was deemed that the majority of them were still relevant and applicable. Only the following ones presented difficulties that required their adaptation to the strain-specific approach:

3.2.1 Filter based on TIC and number of identified peptides

This step was now performed separately for each strain, which means that it will likely not be as stringent as when performed together for all of them. In this case, it will only allow to remove one of the replicates of a certain strain when it is extremely different in its total ion count or number of identified peptides with respect to the rest of them.

3.2.2 Filter based on coefficient of variation

This step is the most affected by the change to the strain-specific approach: this is due to the fact that in the common approach, precursors are filtered out from all samples based on them having a large CV across the quality control samples. Yet, in the case of the strain-specific approach, the precursors identified in the QCs and in each of the strains will not be exactly the same, which makes this approach not appropriate anymore. This could potentially be fixed by including the QC samples in each of the strain-specific runs, so that they are run in DIA-NN with each of the strain-specific libraries and hence the precursors detected in them would be much closer to those in the samples of each strain. However this would cause other issues, such as the number of QCs being much larger than the number of actual samples from that strain in each strain-specific DIA-NN run. Therefore, we resorted to the characteristic of this dataset that was mentioned when describing Figure 3.2 (d): that the CV for the precursors across biological replicates is quite close to that across QCs, meaning that it can be assumed that the variability captured by the CV across biological replicates is, in its majority, technical variability. This justifies the filtering of precursors with a high technical variability associated to them based on the CV across biological replicates. Hence, in the case of the strain-specific approach, precursors were filtered out in each strain that were among the higher 10th percentile of the CV distribution, with the CV being calculated solely across the samples belonging to that particular strain.

3.2.3 Batch correction

The previous approach to this step was rendered inapplicable for the strain-specific approach since each strain is now processed separately in DIA-NN. This is due to the fact that DIA-NN automatically performs a normalization of the detected quantity of precursors, and this normalization happens across all samples that are run together in DIA-NN. Hence, in this case, this happens separately for each strain, meaning that their precursor quantities are not comparable across strains anymore.

In this particular project, the solution to this was to simply not perform a batch correction, due to the lack of batch effects as shown in Figure 3.2 (e). Nonetheless, we are aware that this is a very specific case in which the dataset is of a great quality, and a strain-specific-applicable batch normalization approach is required. More about this will be discussed in Chapter 4.

3.3 Assessment of the strain-specific approach

As mentioned in Chapter 1, one of the goals of this study was for the strain-specific approach to allow to delve deeper into the proteome of each of the *S. cerevisiae* strains, identifying strain-specific proteins that could hardly be identified otherwise. In this section we evaluate how successful this was.

Figure 3.4 (a) illustrates the comparison of the number of proteins identified in each of the two approaches: each dot represents a strain, while the X axis shows the difference between number of proteins identified in the strain-specific approach and in the common approach, and

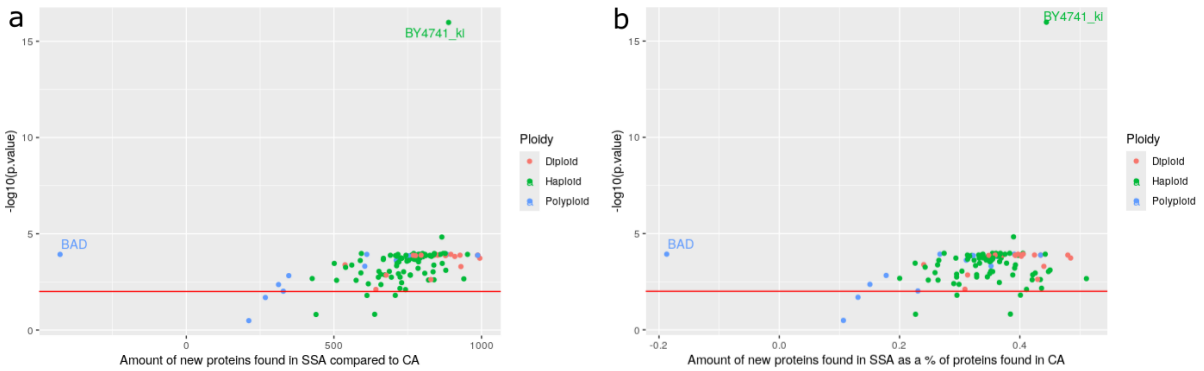

Figure 3.4: Two plots containing information about the difference in the number of proteins found for each strain in the strain-specific approach as compared to the common approach. This information is reflected on the X axis, in plot (a) as the raw difference, and in plot (b) as a percentage of the number of proteins found in the common approach. In both plots, each dot represents a strain, and they have been colored based on their ploidy. The Y axis shows the -$\log_{10}$(p-value) for the t-test between the number of proteins detected in each approach, and the horizontal red line is located at the equivalent to $\alpha = 0.01$.

the Y axis contains the -$\log_{10}$(p-value) for this comparison. Figure 3.4 (b) contains the same information, with the difference that the X axis has been changed to represent the difference in the number of proteins found as a percentage of the number of proteins identified in the common approach. Both these figures show that a significant increase in the number of identified proteins is achieved by the strain-specific approach.

It must be noted that both figures show two important outliers: the BY4741-ki and BAD strains. The case of BY4741-ki is easily explainable: since it is the laboratory strain, which was present in 30 replicates (as opposed to the 3-4 replicates for all other strains), it is expected to have such a large -$\log_{10}$(p-value) compared to the rest of the strains. On the other hand, the case of BAD is not so straightforward: it was later noticed that the GDPF for this strain contained around 3000 proteins, while most strains contain around 6000; this can be observed in Figure 2.3 (c), where BAD is represented by the first bar on the left. This justifies the lower number of identifications, however, it remains to be discussed with our collaborator if this was due to an error in the sequencing, or to an event of biological relevance occurring in this strain.

3.4 Biological questions

3.4.1 Allele-specific expression

As mentioned in Chapter 1, one of the main interest of this project on the biological side was to take advantage of the successful haplotype phasing in the heterozygous diploid strains included in the ScRAP in order to target haplotype-specific biological questions. One of such questions is allele-specific expression: this is, for proteins whose sequence is present in both haplotypes, is the same amount of this protein produced from each haplotype? Or is there a dominance of one of the haplotypes? Nevertheless, there is the limitation that this difference is only possible to evaluate for proteins which exhibit sequence differences between the two haplotypes (otherwise it is impossible to recognize from which haplotype each copy of the protein was produced). This is the reason why, as described in section 2.2.1, peptides from such proteins were specifically labelled to represent whether they are present in the version of the protein coming from one haplotype, the other, or both of them, as represented in Figure 1.4. This means that both DIA-NN and maxLFQ will interpret these three versions as three independent proteins, and quantify them separately: if we were dealing with a protein named Protein_1 (which, again, was present in both haplotypes but with a different sequence between them) we

would obtain quantification results for three different versions of it: Protein_1_common, Protein_1_common_HP1 and Protein_1_common_HP2. This then allows to test the abundances of the last two against each other to resolve whether more copies of the protein are produced from one of the two haplotypes. More about the accuracy of the quantification of proteins in this way will be covered in the corresponding section of Chapter 4.

The results of this testing are presented in Figure 3.5: (a) shows, in blue, the number of total proteins whose sequence should be present in both haplotypes with some difference between them, based on the original GDPF for that strain. The red bars represent the number of these proteins in each strain that were actually detected in the final reports as coming from both haplotypes, and the green bars shows the number of them where a significant difference was found in the abundance of the protein coming from one haplotype vs. the other one. (b) shows the same information but without the total number of proteins based on the GDPF, for a better view of the other quantities.

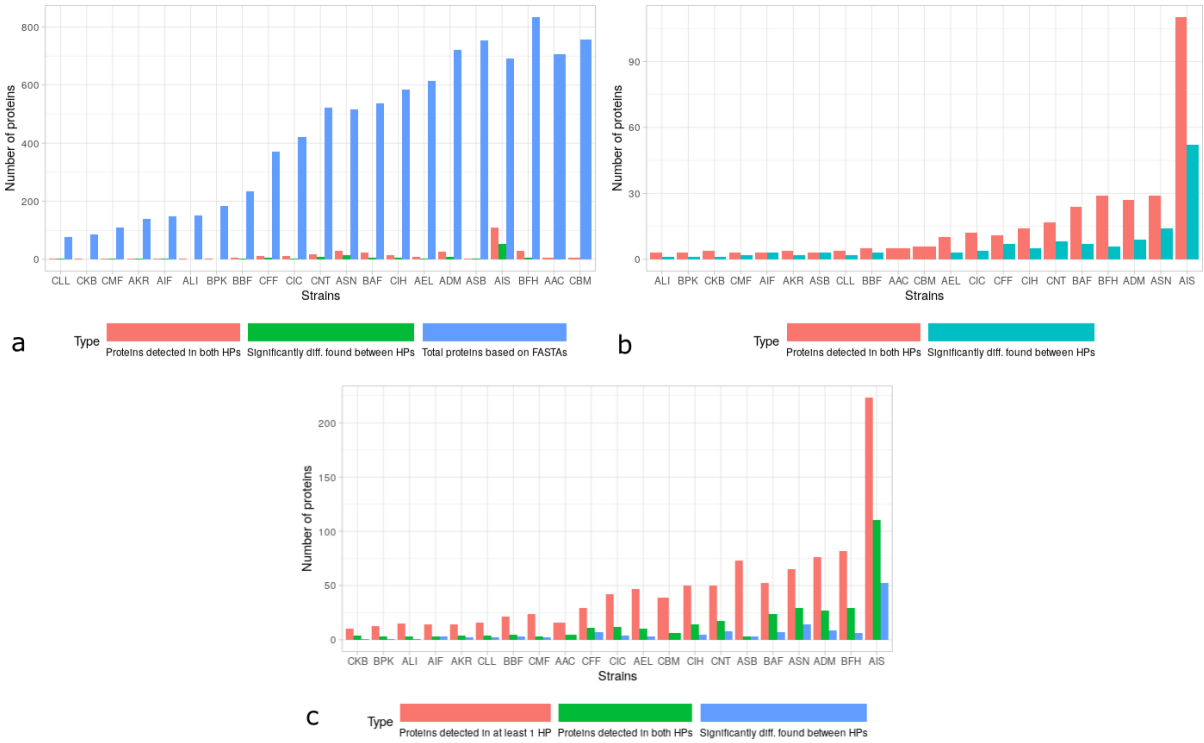

Figure 3.5: Three barplots containing information on the number of proteins found to be significantly differentially abundant between haplotypes, in each of the heterozygous diploid strains in the ScRAP. Plot (a) contains, for each strain, the total number of proteins whose sequence is expected to be present in both haplotypes with some difference between them, based on the GDPFs (in blue). Then in red, the number of proteins out of these which are actually experimentally detected, based on the dataset, and in green the number of those proteins for which a significant differential abundance across haplotypes is found. Plot (b) contains the exact same information, only the bars for the total theoretical number of proteins have been deleted so as to better appreciate the other two. (c) contains the same information as (b), but adds for each strain a column representing the number of proteins that were detected across only one of the haplotypes for each strain.

These results show that an extremely small amount of these proteins is actually detected with respect to those that were expected based on the GDPFs. Nevertheless, Figure 3.5 (b) shows that for those strains in which such proteins are detected in both haplotypes, the proportion of them that is found to be significantly differentially expressed between the haplotypes is not negligible, pointing to the existence of an actual difference in the amount of protein copies (of a certain protein) that are produced from each haplotype, at least for some proteins. Still, low detection prevents further conclusions at this point.

In summary, a total of 51 proteins were found to be significantly differentially abundant between haplotypes across a total of 12 strains (out of the total of 21 heterozygous diploid strains), with most of these proteins showing significance in a single strain. The list of these proteins is presented in Table A.1, in Appendix A. A gene ontology enrichment analysis was performed on these 51 proteins, using as background the total set of proteins detected for each strain in the analysis, but showed no significant enrichment. This is, proteins were shown to be mostly related to general metabolism and amino acid metabolism, but due to the tendency of the employed experimental setting to detect mostly such proteins (due to their large abundance in the cells), these results were not significant.

3.4.2 Effect of insertions and deletions on protein expression

As covered in section 2.1.3, a set of files were provided by our collaborators, based on the telomere-to-telomere sequencing of the strains, and containing information about different SVs in the ScRAP strains. In this case, we looked at insertions and deletions: the corresponding files contained information about 279 insertions and 525 deletions, each of them affecting a concrete gene (or genes), and found in usually a few of the strains. Consequently, it was decided to, for each insertion or deletion, test for the presence of the protein affected by it between strains containing and not containing the mutation. This is, for each sample from each of the strains, a value was produced for the protein, 1 or 0 respectively if the protein was or was not identified in that sample. Then, a proportion test was performed on these values between the strains that contained the mutation and those that didn't.

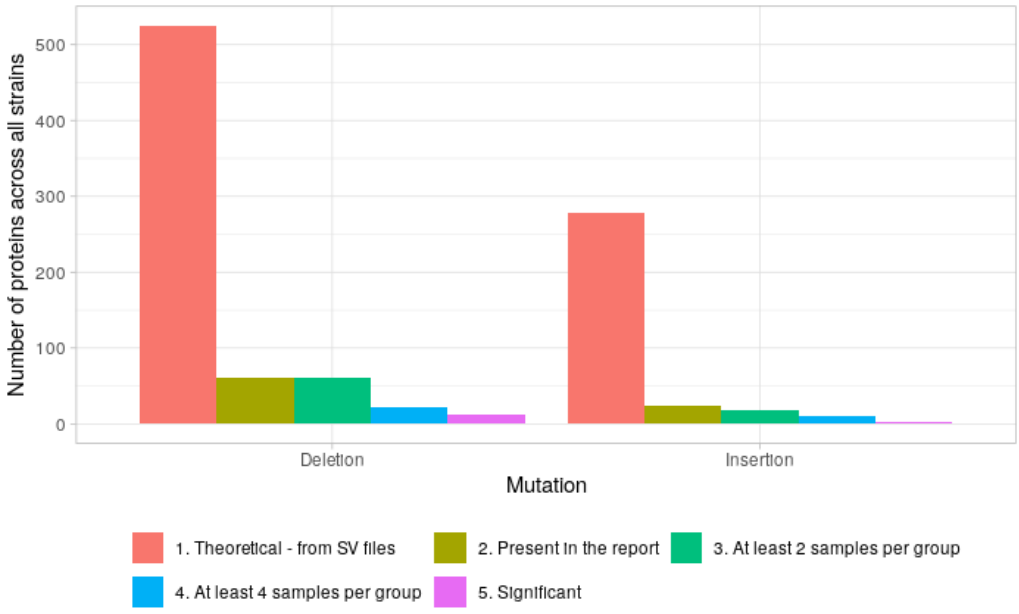

Figure 3.6: Barplot containing information about the number of proteins containing insertions or deletions in some strains, that were found to be significantly differentially present between the strains containing the corresponding mutation and those that didn't. This information is presented over all strains in the ScRAP together. Red bar represents the total number of proteins supposed to have each type of mutation in at least some of the strains, based on the sequencing performed by our collaborators [9]. The khaki bar represents the number of these proteins that were detected in the experimental data, while the following 2 bars (green and blue) represent the number of these proteins for which there were at least 2 and 4 samples (respectively) in both groups to be compared (this is, strains with and without the mutation). Finally, the purple bar represents the amount of these proteins that were found to be significantly differentially present between mutated and non-mutated strains.

The results from this testing are presented in Figure 3.6. The first column represents the number of theoretical proteins containing each type of mutation based on the SV files generated based on the sequencing information. The second bar represents the number of proteins which are actually detected. Again the same issue as in the previous section arises: most of the theoretical proteins are not actually detected in practice, likely due to the fact that their abundance is low and the experimental method employed here tends to favor highly abundant proteins. This complicates the drawing of conclusions, and even more so what is represented in the bright green and blue bars in the figure: the number of proteins for which there are at least two or four (respectively) samples in both groups prior to the testing. This is, the groups being the two types of strains which are being tested against each other, those carrying the mutation and those which don't. Two and four were chosen as the numbers of samples to be shown here because two is the minimum sample size necessary to be able to perform the testing, while four was chosen to illustrate the large amount of proteins for which testing is possible, but is occurring based on extremely small sample sizes. This reflects the biggest issue in this section: most mutations are only present in a couple of strains, and not present in all other strains. This means that the sample sizes for the testing are going to be extremely different, with the one for the group containing the mutation being much smaller. If on top of this, some of the strains containing the mutation have been dropped from the data during filtering (or at least some of their samples), sample sizes for this group end up being dramatically low, which not only directly prevents the possibility of testing in some cases (when less than 2 samples are present in this group) but also strongly decreases the power when testing is possible. This explains the extremely small amount of significantly differentially detected proteins between mutated and non-mutated strains for both insertions and deletions.

The list of proteins differentially expressed when affected by a certain deletion or insertion in certain strains is given in Table A.2, in Appendix A. Looking into the biological relevance or potentially affected pathways was out of the scope of this thesis, but will be followed up in further work.

Chapter 4

Discussion

4.1 Processing of DIA-NN output

The herein developed processing pipeline is deemed to have successfully removed low-quality and outlying samples from the dataset, as well as unreliable precursors, which should result in confident and robust protein quantifications that are representative of the true protein content of each of the species. The pipeline is also considered to have been properly adapted from the common to the strain-specific approach, in order to deal with the particular idiosyncrasies of the latter. Nonetheless, some further considerations are required pertaining some of the sections of the processing.

Regarding the filtering of all non-proteotypic precursors, it is important to realize that even though it allows for more reliable protein identifications, their removal causes a loss of information. Hence, this trade-off needs to be taken into account. Possibly in further approaches, retaining of non proteotypic peptides until later stages of the processing could be considered. Furthermore, the development of an algorithm that could include them during protein quantification would prove extremely useful as well.

As mentioned in Chapter 2, the DIA-NN output contains several types of Q-values, not only the ones that are used to filter in this particular case. Even though the filtering performed here is quite stringent and should allow for reliable quantification, it might be interesting to take other Q-values into account. Particularly for the strain-specific approach, the Library Q-values might be useful. According to the DIA-NN Github note, these are Q-values calculated for each library entry. Therefore, filtering for the library-specific Q-values might be appropriate in the strain-specific processing since libraries of different sizes are being used for each strain, and precisely these different library sizes might affect the number of precursors that receive a significant Q-value.

The objective of filtering based on the robust Z-score for TIC and number of identified peptides is to remove outlying samples based on these two variables. Thus, it was considered as an option to still perform this filtering for all samples from all strains together, instead of doing it separately for the three or four replicates of each strain, which might result in a more biased selection of samples based on the TIC and number of peptides patterns in each particular strain. This was considered as an option because it was thought that these two variables would be absolute, in the sense that they are not normalized across each DIA-NN run but that they are strictly dependent only on the actual number of ions and identified peptides in a sample, respectively. However, after further discussion, it was not clear that this is the case, so it was decided to keep this step separate for each strain for now. It must be noted though, as mentioned already in Chapter 3, that this filtering will be less stringent than its equivalent in the common approach.

Finally, concerning batch correction, the reasons why the method employed in the common approach was rendered inapplicable to the strain-specific approach were already discussed in Chapter 3 and are due to the separate normalization of precursor quantities by DIA-NN for each DIA-NN run. As options for a strain-specific-proof batch correction method, it was

proposed to use the batch correction ratios calculated for each plate in the common approach, and while this would be appropriate, it would require the extra work of running a common approach apart from the strain-specific approach in all future studies. Consequently, it was instead thought to calculate batch correction ratios based on the values for the laboratory strain BY4741-ki, for which, as mentioned in Chapter 2, five replicates were present in each plate. All 30 replicates of this strain are run together in DIA-NN, so they should provide an accurate way of estimating batch effects, and ratios to correct for them. Quality control samples could also be useful to this end, however these are only divided into Batches 1 and 2 (since they were not included in the plates but ran separately in between samples), while BY4741-ki was indeed present in all six plates.

In spite of this, further considerations arise regarding batch correction in this setting: ongoing work in our group is being carried out to asses the importance of batch effects introduced by separate DIA-NN runs, since some colleagues have reported strong such effects in some particular contexts. Hence, the pipeline presented here will be reviewed and adapted based on further findings on this topic.

4.2 Assessment of the strain-specific approach

As reported in Chapter 3, the strain-specific approach resulted in an average increase in the number of proteins identified per strain of around 35% with respect to the common approach, which is quite encouraging and supports prior assumptions. Previous findings in similar experiments in the literature support this, such as the study by Sun et al. [39], where the use of a larger spectral library built from healthy and cancerous prostate tissue outperformed a previous, smaller and less specific prostate library, with almost a 20% increase in protein identifications. Similar findings were reached in [40], although in this case, the procedure was slightly different: the new, more specific spectral library was in this case generated by performing a database search of the mass spectrometry files first. Then the protein identifications from the database search alone and from the spectral library search with this new library were put together, achieving an increase in protein identifications with respect to the database search alone that ranged from 20 to 156%. Another study reached congruent results after building a detailed spectral library of the guinea pig proteome by bringing together spectra generated from proteomic analysis of samples of different body parts, which also resulted in an increase in protein identifications [41]. It must be noted that no references to such a strain-specific approach in yeast were found.

Thus, even though no prior studies have been performed evaluating exactly the same as is presented here, namely the use of strain-specific spectral libraries, it does seem from both this study and previous literature that the more information that is contained in a spectral library and the more specific that this information is to the analyzed species, the more protein identifications that will be obtained. This, together with sequencing technologies becoming progressively cheaper every year, opens up the field for the creation of more strain-specific libraries for the analysis of new samples, and even for the re-analysis of older samples, with the prospect of new findings from them.

With respect to the results in this particular project, it would also be of interest, based on the plots in Figure 3.4, to study the relationship between the increase in the number of protein identifications and the ploidy of the strains, although this might also be affected by the poor haplotype phasing in the case of polyploid strains. Moreover, results presented in the same Figure are only based on the comparison between the number of proteins identified for each strain in each approach, but the newly identified proteins have not been further studied yet. They are of course expected to be proteins that are present in that concrete strain but not in the

reference strain S288C, but a further analysis of them might be interesting, to see if there is any pattern to be seen regarding their function or other characteristics.

4.3 Biological questions

Despite the complications presented by a relatively low identification of allele-specific pseudo-proteins and by the small amount of samples containing mutations within different proteins respectively, insightful results were obtained regarding both alelle-specific expression and the effect of deletions and non-coding insertions in protein sequences. In the case of allele-specific expression, a total of 51 proteins were identified as differentially abundantly produced from the two haplotypes of heterozygous diploid strains, across the 21 strains in this category. Regarding deletions and non-coding insertions, 16 proteins were found to be differentially detected in those strains where they carried such mutations as compared to those where they did not. The shortcomings of these approaches and the obtained results, as well as ideas for their improvement, will be addressed in the following paragraphs.

There are several considerations to be taken into account with respect to the question of allele-specific expression. Firstly, it was already covered how proteins which are present in both haplotypes but with different sequences between them are being identified and quantified as three different proteins, as covered in section 2.2.1 and Figures 1.4 and 2.2: Protein_1_common (from the peptides which are present in both versions of the protein), Protein_1_common_HP1 (from the peptides which are unique to haplotype 1) and Protein_1_common_HP2 (from the peptides unique to haplotype 2). This raises the question of how precise the identification, but particularly the quantification, can be in this setting given that it is based, for each of these "pseudo-proteins", on fewer precursors than it should typically be for the whole, original protein. This could potentially be evaluated by, apart from comparing the quantification of Protein_1_common_HP1 and Protein_1_common_HP2 to each other, comparing also both of them to Protein_1_common. Since the later should presumably be quantified based on more precursors than the previous two and hence more reliably. it could be used as a reference in order to check whether their detected quantities are in the correct range. If this showed that indeed the quantities detected for Protein_1_common_HP1 and Protein_1_common_HP2 were in a different order of magnitude compared to Protein_1_common for the same protein, this would confirme that there are too few precursors specific to Protein_1_common_HP1 and Protein_1_common_HP2 for their quantification to be accurate. If this were the case, an intuitive solution would be to include Protein_1_common precursors (this is, precursors that are common to both versions of the protein) into both Protein_1_common_HP1 and Protein_1_common_HP2, so as to improve the accuracy of their quantification. However, further consideration is necessary regarding how this would affect proteotypicity.

Another consideration about the question of allele-specific expression is that, as mentioned in Chapter 3, testing for significantly differentially abundant proteins across haplotypes became difficult due to the extremely low amount of proteins that were detected coming from both haplotypes. Nevertheless, the number of proteins that were detected coming from a single one of the haplotypes is considerably higher, as can be seen in Figure 3.5 (c). This leads us to suspect that these proteins, which are being ignored as there is no possibility to test for them across haplotypes, might be biologically meaningful, representing full dominance of one of the haplotypes, with this concrete protein being generated exclusively from said haplotype. This will be the subject of further investigation.

A further, important consideration on this question is that, at the step of strain-specific

library creation for these heterozygous diploid strains, when a peptide is classified as Protein_1_common_HP1 or Protein_1_common_HP2, it is done on a direct comparison of the sequences, simply checking if they are exactly identical to each or not. Therefore, it is not taken into account whether the difference between them might be a single amino acid change or several of them. Moveover, here come into play also the degree to which the physico-chemical properties of some amino acids are much more similar than others, hence the mutation of some amino acids to others being more or less impactful. This might strongly influence the results and their interpretation, so there is an interest in looking into this effect and taking it into account further down the line.

Finally, with respect to allele-specific expression, it would be a possibility in order to obtain more significant results to re-run the mass spectrometry experiment for the samples of the 21 heterozygous diploid strains with a longer gradient in the chromatography step. As covered in Chapter 1, this increases the proteomic depth of the analysis, and should allow for the identification of more proteins, which could potentially help detect these haplotype-specific precursors better. Furthermore, the samples could also be re-run on an even more sensitive mass spectrometer, again increasing the depth of the acquisition.

Regarding the results for the biological question on the effect of insertions and deletions on protein expression, again one of the main issue is the lack of statistical power. The most straightforward way to improve this would be to obtain more replicates for the strains carrying mutations, or at least for those accumulating the most mutations. However, it does not seem like this will be an option, at least not in the near future.

Another improvement that could be added to this section is to, instead of the proportion test with the binary version of the data, try to use a mixed model to model the missing data. This might be a topic of further investigation not before long.

4.4 Ethical thinking, societal relevance, and stakeholder awareness

The organism used in this study, *S. cerevisiae*, is unicellular and non-pathogenic. In addition, none of the used strains was genetically modified; instead, this study employed a collection of naturally occurring yeast isolates to answer basic questions regarding the effect of structural variants on gene expression. Therefore, there are no ethical concerns regarding the experimental part of this project.

In fact, there are some ethical advantages to the approach taken in this project: firstly, the raw proteomic files obtained for the samples will be kept and in due time made publicly available, since DIA provides very deep, rich datasets which are by no means exhausted by the analysis performed here. Hence, it will be possible to come back to them and re-analyze them at no extra experimental cost, for example in the case that more advanced software is developed.

Secondly, the very promising results found with the strain-specific approach might encourage the re-analysis of previously acquired samples or raw proteomic files in virtually any field. Just through the preparation of a new spectral library that is more specific to the sample, a number of new protein identifications could potentially be made, without the need for any further harvesting of samples from animals nor humans. Looking further ahead, the success of the strain-specific approach could be helpful in the development of personalized medicine approaches, since it shows how more accurate prior knowledge on the studied individual allows for better and more accurate findings.

Lastly, in a more general way, the present analysis targets basic biological questions (e.g. the occurrence and consequences of structural variants across the genome) using a non-mammalian, non-higher organism, which might nonetheless be extrapolated to higher order organisms. This is an ethical advantage in itself, which observes the "3 Rs rule in animal research" [42], concretely towards the replacement of animals by other organisms or cell cultures.

Chapter 5

Conclusion

Pangenomes and reference panels such as the ScRAP are extremely useful tools when it comes to the study of any species since they allow to take into account its genetic diversity, thus paving the way for more generalizable results. Together with multi-omics approaches, starting from genomic information and using it to direct further proteomic (or even potentially metabolomic) analysis will help considerably to increase our understanding of a species and the concept of species itself.

Here, in the shape of the strain-specific approach, a method was presented to take advantage of such pangenomes at the level of proteomic research; this is, it was proven that the creation of spectral libraries which are as specific as possible to the analyzed organisms result in a significant increase in the number of protein identifications. Importantly, as part of this master thesis, a pipeline for the strain-specific processing of DIA proteomics data was developed. Given the obtained success, evidenced by an average increase in the number of protein identifications of around 35%, this might be a useful tool for future similar analyses based on pangenomes, specially those of microorganisms where many strains can be collected. Furthermore, given its relative ease of implementation, it might encourage researchers in other fields to also perform their proteomic analyses with new libraries that are more specific to the studied organism or tissue, since this would increase the number of protein identifications they would obtain.

This approach does come with some downsides with respect to a common approach, such as the need for further preparation as well as posterior processing, for example regarding the open question of batch correction in the strain-specific approach, or the difficulty in performing direct comparisons between the obtained protein quantities for each strain. Nonetheless, and despite some further development being necessary, the strain-specific approach is considered a useful and promising tool.

The herein developed approach allowed us to start to address interesting biological questions regarding protein expression that could not be studied with prior strategies. Despite low detection of allele-specific proteins, the first analysis attempt of allele-specific expression in heterozygous diploid strains found 51 proteins to be produced in significantly different amounts from the two haplotypes in such strains, and this number could possibly be increased when implementing some of the improvements suggested in Chapter 4, such as including proteins detected to be expressed exclusively from one of the alleles. Regarding the question on the effect of deletions and non-coding insertions on protein expression, a modest 16 significantly differentially present proteins were found between strains where they carried the mutation and those where they did not. However, further research and discussion with our collaborators is necessary regarding the interpretation and meaningfulness of this result. It is however necessary to remember that particularly the targeting of the allele-specific question would have been impossible without the strain-specific approach.

In summary, strain-specific processing approaches are a promising tool in the field of proteomics, particularly as we are heading into an era where pangenomes will slowly replace reference genomes, making it easier to obtain the necessary strain-specific spectral libraries.

Bibliography

[1] André Goffeau, Bart G Barrell, Howard Bussey, Ronald W Davis, Bernard Dujon, Heinz Feldmann, Francis Galibert, Jörg D Hoheisel, Claude Jacq, Michael Johnston, et al. "Life with 6000 genes". In: *Science* 274.5287 (1996), pp. 546–567.

[2] Jens Nielsen. "Yeast systems biology: model organism and cell factory". In: *Biotechnology journal* 14.9 (2019), p. 1800421.

[3] JENNIFER A TATE, DOUGLAS E SOLTIS, and PAMELA S SOLTIS. "Polyploidy in plants". In: *The evolution of the genome*. Elsevier, 2005, pp. 371–426.

[4] Warren Albertin and Philippe Marullo. "Polyploidy in fungi: evolution after whole-genome duplication". In: *Proceedings of the Royal Society B: Biological Sciences* 279.1738 (2012), pp. 2497–2509.

[5] Jackson Peter, Matteo De Chiara, Anne Friedrich, Jia-Xing Yue, David Pflieger, Anders Bergström, Anastasie Sigwalt, Benjamin Barre, Kelle Freel, Agnès Llored, et al. "Genome evolution across 1,011 Saccharomyces cerevisiae isolates". In: *Nature* 556.7701 (2018), pp. 339–344.

[6] Julia Muenzner, Pauline Trébulle, Federica Agostini, Henrik Zauber, Christoph B Messner, Martin Steger, Christiane Kilian, Kate Lau, Natalie Barthel, Andrea Lehmann, et al. "Natural proteome diversity links aneuploidy tolerance to protein turnover". In: *Nature* (2024), pp. 1–9.

[7] Audrey P Gasch, Bret A Payseur, and John E Pool. "The power of natural variation for model organism biology". In: *Trends in Genetics* 32.3 (2016), pp. 147–154.

[8] Julia Muenzner, Pauline Trébulle, Federica Agostini, Christoph B Messner, Martin Steger, Andrea Lehmann, Elodie Caudal, Anna-Sophia Egger, Fatma Amari, Natalie Barthel, et al. "The natural diversity of the yeast proteome reveals chromosome-wide dosage compensation in aneuploids". In: *BioRxiv* (2022), pp. 2022–04.

[9] Samuel O'donnell, Jia-Xing Yue, Omar Abou Saada, Nicolas Agier, Claudia Caradec, Thomas Cokelaer, Matteo De Chiara, Stéphane Delmas, Fabien Dutreux, Téo Fournier, et al. "Telomere-to-telomere assemblies of 142 strains characterize the genome structural landscape in Saccharomyces cerevisiae". In: *Nature Genetics* 55.8 (2023), pp. 1390–1399.

[10] Sharon R Browning and Brian L Browning. "Haplotype phasing: existing methods and new developments". In: *Nature Reviews Genetics* 12.10 (2011), pp. 703–714.

[11] Robin D Dowell, Owen Ryan, An Jansen, Doris Cheung, Sudeep Agarwala, Timothy Danford, Douglas A Bernstein, P Alexander Rolfe, Lawrence E Heisler, Brian Chin, et al. "Genotype to phenotype: a complex problem". In: *Science* 328.5977 (2010), pp. 469–469.

[12] Bruno Domon and Ruedi Aebersold. "Mass spectrometry and protein analysis". In: *science* 312.5771 (2006), pp. 212–217.

[13] Bilal Aslam, Madiha Basit, Muhammad Atif Nisar, Mohsin Khurshid, and Muhammad Hidayat Rasool. "Proteomics: technologies and their applications". In: *Journal of chromatographic science* (2016), pp. 1–15.

[14] Ozlem Coskun. "Separation techniques: chromatography". In: *Northern clinics of Istanbul* 3.2 (2016), p. 156.

[15] François Chevalier. "Highlights on the capacities of" Gel-based" proteomics". In: *Proteome science* 8.1 (2010), p. 23.

[16] Rudolf M Lequin. "Enzyme immunoassay (EIA)/enzyme-linked immunosorbent assay (ELISA)". In: *Clinical chemistry* 51.12 (2005), pp. 2415–2418.

[17] Paula Ciaurriz, Fátima Fernández, Edurne Tellechea, Jose F Moran, and Aaron C Asensio. "Comparison of four functionalization methods of gold nanoparticles for enhancing the enzyme-linked immunosorbent assay (ELISA)". In: *Beilstein journal of nanotechnology* 8.1 (2017), pp. 244–253.

[18] FX Reymond Sutandy, Jiang Qian, Chien-Sheng Chen, and Heng Zhu. "Overview of protein microarrays". In: *Current protocols in protein science* 72.1 (2013), pp. 27–1.

[19] Emmalyn J Dupree, Madhuri Jayathirtha, Hannah Yorkey, Marius Mihasan, Brindusa Alina Petre, and Costel C Darie. "A critical review of bottom-up proteomics: the good, the bad, and the future of this field". In: *Proteomes* 8.3 (2020), p. 14.

[20] Ankit Sinha and Matthias Mann. "A beginner's guide to mass spectrometry–based proteomics". In: *The Biochemist* 42.5 (2020), pp. 64–69.

[21] Hanan Awad, Mona M Khamis, and Anas El-Aneed. "Mass spectrometry, review of the basics: ionization". In: *Applied Spectroscopy Reviews* 50.2 (2015), pp. 158–175.

[22] Piia Liigand, Karl Kaupmees, and Anneli Kruve. "Influence of the amino acid composition on the ionization efficiencies of small peptides". In: *Journal of Mass Spectrometry* 54.6 (2019), pp. 481–487.

[23] Mark E Ridgeway, Markus Lubeck, Jan Jordens, Mattias Mann, and Melvin A Park. "Trapped ion mobility spectrometry: A short review". In: *International journal of mass spectrometry* 425 (2018), pp. 22–35.

[24] J Mitchell Wells and Scott A McLuckey. "Collision-induced dissociation (CID) of peptides and proteins". In: *Methods in enzymology* 402 (2005), pp. 148–185.

[25] Vadim Demichev, Christoph B Messner, Spyros I Vernardis, Kathryn S Lilley, and Markus Ralser. "DIA-NN: neural networks and interference correction enable deep proteome coverage in high throughput". In: *Nature methods* 17.1 (2020), pp. 41–44.

[26] Christopher Hughes, Bin Ma, and Gilles A Lajoie. "De novo sequencing methods in proteomics". In: *Proteome Bioinformatics* (2010), pp. 105–121.

[27] Xin Zhang, Yunzi Li, Wenguang Shao, and Henry Lam. "Understanding the improved sensitivity of spectral library searching over sequence database searching in proteomics data analysis". In: *Proteomics* 11.6 (2011), pp. 1075–1085.

[28] Alex Hu, William S Noble, and Alejandro Wolf-Yadlin. "Technical advances in proteomics: new developments in data-independent acquisition". In: *F1000Research* 5 (2016).

[29] Lukas Krasny and Paul H Huang. "Data-independent acquisition mass spectrometry (DIA-MS) for proteomic applications in oncology". In: *Molecular omics* 17.1 (2021), pp. 29–42.

[30] Christina Ludwig, Ludovic Gillet, George Rosenberger, Sabine Amon, Ben C Collins, and Ruedi Aebersold. "Data-independent acquisition-based SWATH-MS for quantitative proteomics: a tutorial". In: *Molecular systems biology* 14.8 (2018), e8126.

[31] Florian Meier, Scarlet Beck, Niklas Grassl, Markus Lubeck, Melvin A Park, Oliver Raether, and Matthias Mann. "Parallel accumulation–serial fragmentation (PASEF): multiplying sequencing speed and sensitivity by synchronized scans in a trapped ion mobility device". In: *Journal of proteome research* 14.12 (2015), pp. 5378–5387.

[32] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2023. URL: `https://www.R-project.org/`.

[33] Alex Bateman, Maria-Jesus Martin, Sandra Orchard, Michele Magrane, Shadab Ahmad, Emanuele Alpi, Emily H Bowler-Barnett, Ramona Britto, Austra Cukura, Paul Denny, et al. "UniProt: the universal protein knowledgebase in 2023". In: *Nucleic acids research* 51.D 1 (2023), pp. D523–D531.

[34] Edith D Wong, Stuart R Miyasato, Suzi Aleksander, Kalpana Karra, Robert S Nash, Marek S Skrzypek, Shuai Weng, Stacia R Engel, and J Michael Cherry. "Saccharomyces genome database update: server architecture, pan-genome nomenclature, and external resources". In: *Genetics* 224.1 (2023), iyac191.

[35] Franziska Kistner, Justus L Grossmann, Ludwig R Sinn, and Vadim Demichev. "QuantUMS: uncertainty minimisation enables confident quantification in proteomics". In: *BioRxiv* (2023), pp. 2023–06.

[36] Jürgen Cox, Marco Y Hein, Christian A Luber, Igor Paron, Nagarjuna Nagaraj, and Matthias Mann. "Accurate proteome-wide label-free quantification by delayed normalization and maximal peptide ratio extraction, termed MaxLFQ". In: *Molecular & cellular proteomics* 13.9 (2014), pp. 2513–2526.

[37] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[38] Vadim Demichev. *diann: Report processing and protein quantification for MS-based proteomics*. R package version 1.0.1, commit af538f6e2cd5ab715e1381632e17cb8f234ebf53. 2020. URL: `https://github.com/vdemichev/diann-rpackage`.

[39] Rui Sun, Mengge Lyu, Shuang Liang, Weigang Ge, Yingrui Wang, Xuan Ding, Cheng Zhang, Yan Zhou, Shanjun Chen, Lirong Chen, et al. "A prostate cancer tissue specific spectral library for targeted proteomic analysis". In: *Proteomics* 22.7 (2022), p. 2100147.

[40] Erik Ahrné, Alexandre Masselot, Pierre-Alain Binz, Markus Müller, and Frederique Lisacek. "A simple workflow to increase MS2 identification rate by subsequent spectral library search". In: *Proteomics* 9.6 (2009), pp. 1731–1736.

[41] Pawel Palmowski, Rachael Watson, G Nicholas Europe-Finner, Magdalena Karolczak-Bayatti, Andrew Porter, Achim Treumann, and Michael J Taggart. "The generation of a comprehensive spectral library for the analysis of the Guinea Pig proteome by SWATH-MS". In: *Proteomics* 19.15 (2019), p. 1900156.

[42] William Moy Stratton Russell, Rex Leonard Burch, Charles Westley Hume, et al. *The principles of humane experimental technique*. Vol. 238. Methuen London, 1959.

Appendix A

General appendix

A.1 Table for allele-specific expression proteins

Table A.1: Table containing the information on the proteins that were found to be significantly differentially produced by the 2 haplotypes in heterozygous diploid strains. The first column contains the name of the strain where the significant difference for this protein was found, the following columns consist of the names of the protein, systematic and standard one (when available). Finally, the raw and FDR-corrected p-values are presented in the last two columns.

Strain	Protein systematic name	Protein standard name	p-value	FDR
AEL	YGR187C	HGH1	0.010	0.029
AEL	YGL039W		0.006	0.029
AEL	YIR003W	AIM21	0.011	0.029
AIF	YOR042W	CUE5	0.009	0.013
AIF	YER063W	THO1	0.003	0.009
AIS	YHR020W		0.006	0.047
AIS	YFR052W	RPN12	0.006	0.047
AIS	YGL043W	DST1	0.004	0.042
AIS	YGR253C	PUP2	0.000	0.002
AIS	YGL049C	TIF4632	0.001	0.014
AIS	YGR207C	CIR1	0.000	0.002
AIS	YGR048W	UFD1	0.004	0.047
AIS	YGL062W	PYC1	0.002	0.027
AIS	YGR012W	MCY1	0.005	0.047
AIS	YFL014W	HSP12	0.003	0.040
AIS	YGL012W	ERG4	0.000	0.003
AIS	YFR016C	AIP5	0.007	0.049
AIS	YFL022C	FRS2	0.006	0.047
AIS	YGR264C	MES1	0.001	0.014
AIS	YGL037C	PNC1	0.003	0.042
ASN	YHR020W		0.000	0.000
ASN	YGR005C	TFG2	0.002	0.017
ASN	YLL026W	HSP104	0.000	0.001
ASN	YLR058C	SHM2	0.006	0.038
BAF	YDR212W	TCP1	0.000	0.003
BBF	YEL020W-A	TIM9	0.021	0.035
BBF	YLR044C	PDC1	0.006	0.015
BBF	YMR186W/YPL240C	HSC82	0.000	0.000
BPK	YAL005C/YLL024C	SSA1	0.007	0.020
CFF	YGL147C/YNL067W	RPL9A	0.002	0.006

CFF	YER091C	MET6	0.000	0.005
CFF	YER143W	DDI1	0.020	0.045
CFF	YHL033C/YLL045C	RPL8A	0.002	0.006
CFF	YER006W	NUG1	0.002	0.006
CIC	YBL017C	PEP1	0.022	0.039
CIC	YOR251C	TUM1	0.033	0.046
CIC	YCR005C	CIT2	0.022	0.039
CIC	YBR031W/YDR012W	RPL4A	0.012	0.039
CIC	YBR011C	IPP1	0.000	0.000
CKB	YJL172W	CPS1	0.000	0.001
CLL	YHR146W	CRP1	0.001	0.001
CLL	YNL138W	SRV2	0.000	0.001
CMF	YMR194W/YPL249C-A	RPL36A	0.020	0.030
CMF	YLR441C/YML063W	RPS1A	0.030	0.030
CNT	YMR039C	SUB1	0.000	0.000
CNT	YMR092C	AIP1	0.001	0.003
CNT	YMR038C	CCS1	0.001	0.006
CNT	YOL097C	WRS1	0.007	0.018
CNT	YDL075W/YLR406C	RPL31A	0.005	0.016
CNT	YEL020W-A	TIM9	0.000	0.000
CNT	YML057W	CMP2	0.006	0.017

A.2 Table for mutation-containing proteins

Table A.2: Table with all proteins that were found to be significantly differentially detected between the strains where they carried a mutation and those where they did not. Proteins are named both with their systematic name and their standard name (when available). The mutation type is indicated in the next column, and then the raw and FDR-corrected p-values.

Protein systematic name	Protein standard name	Mutation type	p-value	FDR
YHR188C	GPI16	Deletion	0.00762	0.04087
YMR099C		Deletion	0.00380	0.02242
YMR105C	PGM2	Deletion	0.00001	0.00009
YMR108W	ILV2	Deletion	0.00000	0.00000
YMR116C	ASC1	Deletion	0.00000	0.00000
YCR053W	THR4	Deletion	0.00000	0.00000
YCR083W	TRX3	Deletion	0.00833	0.04094
YCR084C	TUP1	Deletion	0.00000	0.00000
YCR088W	ABP1	Deletion	0.00000	0.00000
YHR013C	ARD1	Deletion	0.00000	0.00000
YIR035C	NRE1	Deletion	0.01003	0.04551
YPL152W	RRD2	Deletion	0.00000	0.00000
YCL018W	LEU2	Deletion	0.00000	0.00000
YIL169C	CSS1	Insertion	0.00266	0.01598
YJL020C	BBC1	Insertion	0.00000	0.00000
YCL026C-B	HBN1	Insertion	0.00000	0.00000

Appendix B

Appendix for R code

B.1 Creating functions to be used later

```r
#' Create the correspondence dataframe for the full DIA-NN report
#'
#' This function takes as input the unique_genes matrix from DIA-NN, and a file with the structure of the samples on the plates. This is, in this second file, each
#' row corresponds to a sample, and there are columns describing: the plate, the well, the batch and the strain that was in that sample. What this function does
#' is to match the information from these 2 dataset based on the positions on the plates, and to create a new dataframe, based on this second one, but with
#' columns containing: the file names, Well ID, Sample, Strain, Batch ID and Plate ID. This allows to then name the columns in the unique_genes file and in the
#'  DIA-NN report based on the "Sample" column, which specifies which replicate of each strain each sample is (for example, "AAB_3").
#'
#' @param df The unique_genes matrix from DIA-NN (as a dataframe)
#' @param structure A dataframe containing a sample in each row, and columns with information about their position in the plate, the strain that was in it...
#' @return A nicer structure dataframe, containing the following columns: file names, Well ID, Sample, Strain, Batch ID and Plate ID.
#'
create_sample_correspondence_dataset <- function(df, structure)  {
  # Reconstruct ID for each plate in the same way as in the column names
  out = c()
  for (i in 1:nrow(structure)) {
    if (structure$strain[i] != "QC") {
      # Plate number
      plate = paste("P0", substr(structure$plate[i], 12, 12), sep = "")

      # Well ID
      if (nchar(structure$column96[i]) == 1) {
        num = paste("0", structure$column96[i], sep="")
      }
      else {num = structure$column96[i]}
      well = paste(structure$row96[i], num, sep = "")

      # Put it all together
      ID = paste(plate, well, sep = "_")
      out =  c(out, ID)
    }
    else {out <- c(out, NA)}
  }
  structure$ID = out

  df_unique <- data.frame(File.Name = colnames(df))

  # Create the strain replicate names
  strain_replicates = c()
  well_IDs = c()
```

```
strains = c()
for (i in 1:length(colnames(df))) {
  og_colname = colnames(df)[i]

  # For QCs
  if (grepl("QC", og_colname) & grepl("Batch2", og_colname)) {
    start <- str_locate(og_colname, '_P00_')[2]
    end <- str_locate(og_colname, '.d')[1]
    new <- substr(og_colname, start + 1, end - 1)
    new <- gsub("\\.", "-", new)
    new <- paste(new, "2", sep = "_")
    old = new
    strain = "QC"
  }
  else if (grepl("QC", og_colname)) {
    start <- str_locate(og_colname, '_P00_')[2]
    end <- str_locate(og_colname, '.d')[1]
    new <- substr(og_colname, start + 1, end - 1)
    new <- gsub("\\.", "-", new)
    old = new
    strain = "QC"
  }

  # For the rest of wells
  else {
    for (j in 1:length(structure$ID)) {
      ID = structure$ID[j]
      if (grepl(ID, og_colname)) {
        strain = structure$strain[j]
        if (sum(grepl(strain, strain_replicates)) >= 1) {
          num = sum(grepl(strain, strain_replicates)) + 1
          new = paste(strain, as.character(num), sep = "_")
          old = ID
        }
        else {
          new = paste(strain, 1, sep = "_")
          old = ID
        }
      }
    }
  }
  strain_replicates = c(strain_replicates, new)
  well_IDs = c(well_IDs, old)
  strains = c(strains, strain)
}

# Create the batch indicator
batch_ID = c()
for (i in 1:length(colnames(df))) {
  og_colname = colnames(df)[i]

  # For QCs
  if (grepl("QC", og_colname) & grepl("Batch2", og_colname)) {
    batch_ID = c(batch_ID, 2)
  }
  else if (grepl("QC", og_colname)) {
    batch_ID = c(batch_ID, 1)
  }

  # For the rest of wells
  else {
    for (j in 1:length(rownames(structure))) {
      if (grepl(structure$ID[j], og_colname)) {
```

```
          plate = structure$plate[j]
          plate_num = as.numeric(substr(plate, nchar(plate), nchar(plate)))
          if (plate_num <= 3) {batch_ID = c(batch_ID, 1)}
          else if (plate_num > 3) {batch_ID = c(batch_ID, 2)}
        }
      }
    }
  }

  # Create the plate indicator
  plate_ID = c()
  for (i in 1:length(colnames(df))) {
    og_colname = colnames(df)[i]

    # For QCs
    if (grepl("QC", og_colname)) {
      plate_ID = c(plate_ID, "QC")
    }

    # For the rest of wells
    else {
      for (j in 1:length(rownames(structure))) {
        if (grepl(structure$ID[j], og_colname)) {
          plate = structure$plate[j]
          plate_num = as.numeric(substr(plate, nchar(plate), nchar(plate)))
          plate_ID = c(plate_ID, plate_num)
        }
      }
    }
  }

  # Bring together the dataframe
  sample_correspondence = data.frame(df_unique$File.Name, well_IDs, strain_replicates, strains, batch_ID, plate_ID)
  colnames(sample_correspondence) = c("File_Name", "Well_ID", "Sample", "Strain", "Batch_ID", "Plate_ID")

  # Add names in ms03 computer - in order to be able to run DIA-NN
  ms03_names = c()
  for (i in 1:nrow(sample_correspondence)) {
    name = sample_correspondence$File_Name[i]
    loc_1 = str_locate(name, "Projects.")[2]
    loc_2 = str_locate(name, ".d")[1]
    name_ms03 = substr(name, loc_1+1, loc_2-1)
    name_ms03 = gsub(".", "-", name_ms03, fixed = T)
    name_ms03 = paste(name_ms03, ".d", sep = "")
    ms03_names = c(ms03_names, name_ms03)
  }
  sample_correspondence$names_in_ms03 <- ms03_names

  return(sample_correspondence)
}


#' Match between correspondence dataset and OD report
#'
#' Add a column with the Plate ID to the OD dataset so that its rows can be matched to the ones in the large dataset. Also add a column indicating whether each
#' well generated or not measurements, and hence is or not included in the DIA-NN report.
#'
#' @param OD A dataframe containing a sample in each row, columns with the OD of each sample, but also a column indicating the plate in which the sample was, and
#' another one indicating the position of the sample within the plate
```

```r
#' @param sample_correspondence The dataframe created by "create_sample_correspondence_dataset_from_full_report()", so with the following columns: file names,
#' Well ID, Sample, Strain, Batch ID and Plate ID.
#' @param missing A vector with the names of the wells containing samples that didn't produce a single measurement (since they are included in the OD dataframe,
#' but not in the sample_correspondence one, because this one is built based on the DIA-NN report and this one doesn't contain samples with 0 measurements).
#' @return The OD dataframe with an extra column containing the Plate ID, and another extra column containing whether each well is or not included in the DIA-NN
#' report (wells with extremely low ODs sometimes don't generate any measurements in the MS and hence are not included in the DIA-NN report I think, something
#' like that).
#'
match_OD_info_to_sample = function(OD, sample_correspondence, missing = c(NA)) {
  # Create well ID for the OD table
  out = c()
  for (i in 1:length(OD$plate)) {
    # Plate number
    plate = paste("P0", OD$plate[i], sep = "")

    # Well ID
    if (nchar(OD$position[i]) == 2) {
      num = paste(substr(OD$position[i], 1, 1), "0", substr(OD$position[i], 2, 2), sep="")
    }
    else {num = OD$position[i]}

    # Put it together
    well = paste(plate, num, sep = "_")
    out = c(out, well)
  }
  OD$ID = out

  # Instead of removing the rows with wells where the samples didn´t produce any measurement, add an extra column containing this information (they are not
  # included in the main DIA-NN report so we need to be aware of them when trying to match them later).
  if (sum(is.na(missing)) == 0) {
    present_or_missing <- c()
    for (i in 1:nrow(OD)) {
      if (OD$ID[i] %in% missing) {
        present_or_missing <- c(present_or_missing, "Missing")
      }
      else {
        present_or_missing <- c(present_or_missing, "Present")
      }
    }
    OD$Presence <- present_or_missing
  }
  return(OD)
}


#' Add columns with sample information to report dataset
#'
#' Add columns to the DIA-NN report based on the sample_correspondence dataset: Well_ID, Sample, Strain, Batch_ID, Plate_ID. Had to add the if right at the
#' beginning because otherwise when using this function on the strain-specific reports, I get an error for strain CPS, which has an empty report.
#'
#' @param data The DIA-NN report as a dataframe
```

```r
#' @param sample_correspondence The dataframe created by "create_sample_correspondence_dataset_from_full_report()", so with the following columns: file names,
#' Well ID, Sample, Strain, Batch ID and Plate ID.
#' @param OD A dataframe containing a sample in each row, columns with the OD of each sample, but also a column indicating the plate in which the sample was, and
#' another one indicating the position of the sample within the plate
#' @param column_to_use Do we want to do this based on the name_in_ms03 column (used for SS report) or on File.Name (used for CA report)
#' @return The DIA-NN report as a dataframe, with the mentioned extra columns
#'
add_correspondence_columns_to_report = function(data, sample_correspondence, OD, column_to_use) {
  if (nrow(data) > 0 & column_to_use == "names_in_ms03") {
    # Create empty columns to fill in
    data$Well_ID = data$Sample = data$Strain = data$Batch_ID = data$Plate_ID = data$OD_at_harvest = data$OD_preculture = NA

    # Fill in these empty columns based on the created sample_correspondence
    for (i in 1:nrow(sample_correspondence)) {
      bool = data$names_in_ms03 == sample_correspondence$names_in_ms03[i]

      data$Well_ID[bool] = sample_correspondence$Well_ID[i]
      data$Sample[bool] = sample_correspondence$Sample[i]
      data$Strain[bool] = sample_correspondence$Strain[i]
      data$Batch_ID[bool] = sample_correspondence$Batch_ID[i]
      data$Plate_ID[bool] = sample_correspondence$Plate_ID[i]
    }

    # Fill in the emtpy OD columns based on the OD data
    for (i in 1:nrow(OD)) {
      bool = data$Well_ID == OD$ID[i]
      data$OD_at_harvest[bool] = OD$OD_at_harvest[i]
      data$OD_preculture[bool] = OD$OD_preculture[i]
    }
    return(data)
  }

  else if (nrow(data) > 0 & column_to_use == "File.Name") {
    # Create empty columns to fill in
    data$Well_ID = data$Sample = data$Strain = data$Batch_ID = data$Plate_ID = data$OD_at_harvest = data$OD_preculture = NA

    # Fill in these empty columns based on the created sample_correspondence
    for (i in 1:nrow(sample_correspondence)) {
      bool = data$File.Name == sample_correspondence$File_Name[i]

      data$Well_ID[bool] = sample_correspondence$Well_ID[i]
      data$Sample[bool] = sample_correspondence$Sample[i]
      data$Strain[bool] = sample_correspondence$Strain[i]
      data$Batch_ID[bool] = sample_correspondence$Batch_ID[i]
      data$Plate_ID[bool] = sample_correspondence$Plate_ID[i]
    }

    # Fill in the emtpy OD columns based on the OD data
    for (i in 1:nrow(OD)) {
      bool = data$Well_ID == OD$ID[i]
      data$OD_at_harvest[bool] = OD$OD_at_harvest[i]
      data$OD_preculture[bool] = OD$OD_preculture[i]
    }
    return(data)
  }

```

```
  else {
    return(data)
  }

}


#' Add OD information to sample_correspondence
#'
#' Add a column with the OD to the sample_correspondence dataset
#'
#' @param sample_correspondence The dataframe created by "create_sample_correspondence_dataset_from_full_report()", so with the following columns: file names,
#' Well ID, Sample, Strain, Batch ID and Plate ID.
#' @param OD A dataframe containing a sample in each row, columns with the OD of each sample, but also a column indicating the plate in which the sample was, and
#' another one indicating the position of the sample within the plate
#' @return The sample_correspondence dataset with an extra column containing the OD values
#'
add_OD_to_sample_correspondence = function(sample_correspondence, OD) {
  OD_at_harvest = c()
  OD_preculture = c()
  for (i in 1:nrow(sample_correspondence)) {
    if (sample_correspondence$Strain[i] == "QC") {
      OD_at_harvest = c(OD_at_harvest, NA)
      OD_preculture = c(OD_preculture, NA)
    }
    else {
      OD_at_harvest = c(OD_at_harvest, OD$OD_at_harvest[OD$ID == sample_correspondence$Well_ID[i]])
      OD_preculture = c(OD_preculture, OD$OD_preculture[OD$ID == sample_correspondence$Well_ID[i]])
    }
  }
  sample_correspondence$OD_at_harvest = OD_at_harvest
  sample_correspondence$OD_preculture = OD_preculture

  return(sample_correspondence)
}


#' Match systematic to standard protein names
#'
#' We provide a dataframe or a vector with systematic protein names, and the output is either a vector of (or a dataframe where one of the columns is) the corresponding
#' standard protein names. It is important to notice that when there is no standard name in the database for a certain protein, the systematic name is returned instead.
#'
#' @param data This can be a vector with the systematic protein names, or a dataframe where one column has the systematic protein names. If it is a dataframe, the name
#' of this column must be "Gene.secondaryIdentifier"
#' @param yeastmine A dataframe with the databse information for protein names in S. cerevisiae, as downloaded from _________.
#' @param simplify A boolean value indicating if we want the output to be simply a vector with the standard protein names (TRUE), or the input dataframe where the standard
#' protein names are added as a new column (FALSE).
#' @param add_extra_columns A boolean value indicating, if simplify == FALSE, whether we only want to add to the dataframe the column with the standard protein names
#' (FALSE) or also all other columns in the provided yeastmine dataframe.
#'

match_systematic_to_standard_protein_names <- function(data,
```

```
                                                    yeastmine,
                                                    simplify = FALSE,
                                                    add_extra_columns = FALSE) {

  # First of all, if we have received a vector as input, turn it into a dataframe and work from there
  if (class(data) == "character") {
    data <- data.frame(data)
    colnames(data) <- c("Gene.secondaryIdentifier")
  }

  # Match the names to the YeastMine ones
  df <- left_join(data, yeastmine, by = join_by(Gene.secondaryIdentifier))

  # Create the new column we'll keep as output, where we take standard gene names, but if this is not present, we fill it in with the systematic one
  df <- df %>%
    mutate(Final.Ids = case_when(Gene.symbol == "" ~ Gene.secondaryIdentifier,
                                 is.na(Gene.symbol) ~ Gene.secondaryIdentifier,
                                 TRUE ~ Gene.symbol))

  # Prepare the output according to the specifications provided when calling the function
  if (simplify == TRUE) {
    out <- as.character(df$Final.Ids)
  }
  else {
    if (add_extra_columns == TRUE) {
      out <- df
    }
    else if (class(data) == "data.frame") {
      colnames_to_remove <- colnames(yeastmine)
      colnames_to_remove <- colnames_to_remove[!colnames_to_remove %in% c("Gene.secondaryIdentifier")]
      out <- df %>%
        select(-c(colnames_to_remove))}
    else {
      out <- df %>%
        select(Gene.secondaryIdentifier, Final.Ids)
    }
  }

  # Return output
  return(out)
}
```

B.2 Data preparation

```
# Packages
library(data.table)
library(dplyr)
library(readODS)
source("/~/0. prepare_data_functions.R")


# 1. Load data
## 1.1. Original DIA-NN dataframe
data <- fread("/~/30-0107_SamplesBatch0102.tsv")
data <- as.data.frame(data)

## 1.2. Unique dataframe
unique <- fread("/~/30-0107_SamplesBatch0102.unique_genes_matrix.tsv")
```

```r
unique <- as.data.frame(unique)
row.names(unique) <- unique$Genes
unique <- unique[, -1]

## 1.3. OD data
OD <- read.csv("/~/231130_scrap_ODs_multi_read.txt", sep = "\t")

## 1.4. Structure
structure <- read_ods("/~/new_library_reformatting_alvaro.ods", 1)
structure$column96 <- as.character(structure$column96)


# 2. Data preparation for the main DIA-NN report
## 2.1. Create the sample_correspondence dataframe
sample_correspondence <- create_sample_correspondence_dataset(unique, structure)


## 2.2. Match between correspondence dataset and OD report
### Add a column with the Plate ID to the OD dataset so that its rows can be matched to the ones in the large dataset
missing <- c("P01_A01", "P01_C04", "P01_E06", "P01_H04", "P05_A01", "P06_C03", "P06_D04", "P06_E09")   # Not using this anymore
OD <- match_OD_info_to_sample(OD, sample_correspondence)

### Add OD information to sample_correspondence dataset
sample_correspondence <- add_OD_to_sample_correspondence(sample_correspondence, OD)


## 2.3. Add all previously created columns to the large dataset, as well as another column with the OD
data <- add_correspondence_columns_to_report(data, sample_correspondence, OD)


## 2.4. Save the new (matched) version of the DIA-NN report and its sample_correspondence dataframe
### Save the modified report file
fwrite(data, file <- "/~/30-0107_SamplesBatch0102_matched.tsv", quote=FALSE, sep='\t')

### Save the sample correspondence file
fwrite(sample_correspondence, file = "/~/sample_correspondence.tsv", quote=FALSE, sep='\t')


# 3. Data preparation for the unique_genes matrix
## 3.1. Apply new column names to the unique matrix
colnames(unique) <- sample_correspondence_unique$Sample

## 3.3. Save the modified unique file
fwrite(unique, file <- "/~/unique_matched.tsv", quote=FALSE, sep='\t', row.names = T)

```

B.3 Processing DIA-NN report for common approach

```
Packages
```{r}
library(data.table)
library(dplyr)
library(readODS)
library(kableExtra)
library(gridExtra)
library(ggplot2)
library(glue)
```
```

```
library(gt)
library(ggvenn)
library(ggrepel)
library(diann)
library(ggpubr)
library(forcats)
```


Load data
```{r}
data = fread('/~/30-0107_SamplesBatch0102_matched.tsv')
data = as.data.frame(data)

unique_genes = fread("/~/unique_matched.tsv")
unique_genes = as.data.frame(unique_genes)

sample_correspondence = fread("/~/sample_correspondence.tsv")
sample_correspondence = as.data.frame(sample_correspondence)

stats_file = fread("/~/30-0107_SamplesBatch0102.stats.tsv")
stats_file = as.data.frame(stats_file)
```


0. Set up parameters
```{r}
OD_threshold = 0.12
Q_values_threshold = 0.01
min_samples_per_strain = 3
percentage_of_samples_per_precursor = 0.65
SD_limit_for_TIC_filtering = 2.5
quantile_limit_QC_CV = 0.9
```


1. Remove samples with low OD
```{r}
# Create function
filter_based_on_OD = function(data, OD_threshold) {
  data = data %>% filter(OD_at_harvest > OD_threshold | Strain == "QC")
  return(data)
}

# Run filtering
data_filtered_OD = filter_based_on_OD(data, OD_threshold)
```


2. Remove non-proteotypic peptides
```{r}
# Create function
filter_proteotypic = function(data) {
  data = data %>% filter(Proteotypic == 1)
  return(data)
}

# Run filtering
data_filtered_proteotypic = filter_proteotypic(data_filtered_OD)
```


3. Filter based on Q-values
```
```

```
```{r}
# Create function
filter_Q_values = function(data, Q_values_threshold) {
  data = data %>% filter(Q.Value < Q_values_threshold,
                         PG.Q.Value < Q_values_threshold,
                         Global.Q.Value < Q_values_threshold,
                         Global.PG.Q.Value < Q_values_threshold)
  return(data)
}

# Run filtering
data_filtered_Q = filter_Q_values(data_filtered_proteotypic, Q_values_threshold)
```


4. Filter based on z-score of TIC and number of precursors identified
4.0. Remove first all samples that were already removed by this point?
```{r}
stats_file = stats_file[stats_file$File.Name %in% data_filtered_Q$File.Name,]
```

4.1. Exploration regarding TIC and number of identified precursors
Calculate z-score and robust z-score for TIC as new columns in the stats file
```{r}
stats_file = stats_file %>% mutate(z_score_tic = (MS1.Signal - mean(MS1.Signal))/sd(MS1.Signal))
stats_file = stats_file %>% mutate(robust_z_score_tic = (MS1.Signal - median(MS1.Signal))/mad(MS1.Signal))
stats_file = stats_file %>% mutate(QC = as.factor(case_when(data$Strain[match(File.Name, data$File.Name)] == "QC" ~ 1,
                        TRUE ~ 0)))

# Establish a coloring by which samples have been removed already - not used in the end
stats_file = stats_file %>% mutate(Previously.Removed = case_when(File.Name %in% data_filtered_Q$File.Name ~ FALSE,
                                                                  TRUE ~ TRUE))

ggplot(data = stats_file, aes(x = robust_z_score_tic)) +
  geom_histogram(color = "black", fill = "grey", bins = 100) +
  theme_light() +
  theme(legend.position = "none") +
  xlab("Robust Z-score for TIC") +
  ylab("Count") +
  #geom_vline(aes(xintercept=mean(robust_z_score_tic)),
            #color="blue", linetype="dashed", linewidth=1) +
  geom_vline(aes(xintercept=mean(robust_z_score_tic)-2.5*sd(robust_z_score_tic)),
            color="red", linetype="dashed", linewidth=1) +
  geom_vline(aes(xintercept=mean(robust_z_score_tic)+2.5*sd(robust_z_score_tic)),
            color="red", linetype="dashed", linewidth=1) #+
  #annotate("text", x = -6, y = 18, label = "Mean - 2.5*SD", angle = 90, color = "red") +
  #annotate("text", x = -1.1, y = 18, label = "Mean", angle = 90, color = "blue") #+
  #annotate("text", x = 3.80, y = 18, label = "Mean + 2.5*SD", angle = 90, color = "red")
```

Calculate z-score and robust z-score for number of precursors identified
```{r}
stats_file = stats_file %>% mutate(z_score_pept_num = (Precursors.Identified - mean(Precursors.Identified))/sd(Precursors.Identified))
stats_file = stats_file %>% mutate(robust_z_score_pept_num = (Precursors.Identified - median(Precursors.Identified))/sd(Precursors.Identified))

ggplot(data = stats_file, aes(x = robust_z_score_pept_num)) +
  geom_histogram(color = "black", fill = "grey", bins = 100) +
  theme_light() +
```
```

```
  theme(legend.position = "none") +
  xlab("Robust Z-score for number of precursors identified") +
  ylab("Count") +
  geom_vline(aes(xintercept=mean(robust_z_score_pept_num)),
             color="blue", linetype="dashed", lwd=1) +
  geom_vline(aes(xintercept=-3),
             color="red", linetype="dashed", lwd=1) +
  geom_vline(aes(xintercept=3),
             color="red", linetype="dashed", lwd=1) +
  #annotate("text", x = -2.5, y = 25, label = "Mean - 2.5*SD", angle = 90, color = "red") +
  annotate("text", x = -0.5, y = 25, label = "Mean", angle = 90, color = "blue") #+
  #annotate("text", x = 1.5, y = 25, label = "Mean + 2.5*SD", angle = 90, color = "red")
```

4.2. Perform the filtering

```{r}
# Create function
filter_TIC_and_peptide_number = function(data, stats_file, SD_limit_for_TIC_filtering) {
  # Filter on the stats file
  stats_file_filtered = stats_file %>% filter(robust_z_score_tic > -3 & robust_z_score_tic < 3,
                                              robust_z_score_pept_num > -3 &
                                              ↪ robust_z_score_pept_num < 3)

  # Filter on the actual dataset based on the stats file
  data = data[data$File.Name %in% stats_file_filtered$File.Name,]

  return(data)
}

# Run filtering
data_filtered_TIC = filter_TIC_and_peptide_number(data_filtered_Q, stats_file,
↪ SD_limit_for_TIC_filtering)
```

5. Filter based on detection threshold/sample fraction

5.1. Perform filtering based on number of samples present per strain

```{r}
data_filtered_replicate_num = data_filtered_TIC %>%
  group_by(Strain) %>%
  mutate(sample_count = length(unique(Sample))) %>%
  filter(sample_count >= 3)
```

5.2. Remove, for each strain, those precursors which are not present in at least 3/4 or 2/3 ↪ replicates

```{r}
# Create function
remove_uncommon_precursors_per_strain = function(data, percentage_of_samples_per_precursor) {

  # Set up the filter
  filterSF <- data %>%
    group_by(Precursor.Id, Strain) %>%
    summarise(count = n()) %>%
    ungroup() %>%
    group_by(Strain) %>%
    mutate(maxCount=max(count))

  # Apply filter
  out = data %>% left_join(filterSF) %>% filter(count >=
  ↪ percentage_of_samples_per_precursor*maxCount)

  return(out)
```

```
}

# Filter
data_filtered_prec_per_strain = remove_uncommon_precursors_per_strain(data_filtered_replicate_num, percentage_of_samples_per_precursor)
```


6. Filter based on precursor CV
6.1. First of all I need to calculate the CV for each precursor across: QCs, biological replicates, and all samples, and plot their densities.
```{r}
create_CV_data = function(data) {
  CV_data = data %>%
    group_by(Strain, Precursor.Id) %>%
    mutate("SD_strain" = sd(Precursor.Normalised, na.rm = T), "CV_strain" = sd(Precursor.Normalised, na.rm = T)/mean(Precursor.Normalised, na.rm = T))
  CV_data = CV_data %>% ungroup() %>%
    group_by(Precursor.Id) %>%
    mutate("SD_all_samples" = sd(Precursor.Normalised, na.rm = T), "CV_all_samples" = sd(Precursor.Normalised, na.rm = T)/mean(Precursor.Normalised, na.rm = T))

  return(CV_data)
}

CV_data = create_CV_data(data_filtered_prec_per_strain)
```

Density plot
```{r}
QC_CV_dist = CV_data$CV_strain[CV_data$Strain == "QC"]

ggplot() +
  geom_density(aes(x = CV_data$CV_strain[CV_data$Strain != "QC"], color = "Biological replicates"), linewidth = 0.8) +
  geom_density(aes(x = CV_data$CV_strain[CV_data$Strain == "QC"], color = "QCs"), linewidth = 0.8) +
  geom_density(aes(x = CV_data$CV_all_samples, color = "All samples"), linewidth = 0.8) +
  scale_color_manual("CV across", values = c("Biological replicates" = "blue", "QCs" = "red", "All samples" = "darkgreen")) +
  xlab("Coefficient of variation (CV)") +
  ylab("Density") +
  theme_light() +
  coord_cartesian(xlim = c(0, 1)) +
  geom_vline(xintercept = quantile(QC_CV_dist, probs = c(0.9)), linetype = "dashed", col = "orange")
  #annotate("text", x = 0.7, y = 3.3, label = "90% quantile of CV across QCs", col = "orange") #+
  #geom_vline(xintercept = quantile(QC_CV_dist, probs = c(0.95)), linetype = "dashed", col = "lightblue") +
  #annotate("text", x = 0.7, y = 2.7, label = "95% quantile of CV across QCs", col = "lightblue")
#ggsave("/data/gpfs-1/users/algo12_c/work/Images_for_thesis/CVs.png", plot = plot)
#quantile(QC_CV_dist, probs = c(0.9))
#quantile(QC_CV_dist, probs = c(0.95))
```


6.2. Filtering
Remove from all samples the precursors which have a large CV in the QCs
```{r}
filter_CV = function(CV_data, quantile_limit_QC_CV) {
```

```
  QC_CV_dist = CV_data$CV_strain[CV_data$Strain == "QC"]
  keep_precursors = CV_data$Precursor.Id[CV_data$Strain == "QC" & CV_data$CV_strain <= quantile(QC_CV_dist, probs = c(quantile_limit_QC_CV))]
  data_filtered_by_QC_CV = CV_data[CV_data$Precursor.Id %in% keep_precursors,]
  return(data_filtered_by_QC_CV)
}

data_filtered_by_QC_CV = filter_CV(CV_data, quantile_limit_QC_CV)
```


7. Batch correction
Check differences between plates
```{r}
# By plate
batch_correction_1 <- ggplot(data = data_filtered_by_QC_CV, aes(x = Plate_ID, y = log2(Precursor.Normalised), group = Plate_ID)) +
  geom_boxplot(outlier.size = 0.5) +
  xlab("Plate") +
  theme_light()

# By well
#ggplot(data = data_filtered_by_QC_CV, aes(x = Well_ID, y = log2(Precursor.Normalised), color = Plate_ID)) +
#  geom_boxplot(outlier.shape = NA) +
#  theme(axis.text.x=element_blank(),
#        axis.ticks.x=element_blank())
```

Correct for batch effect
```{r}
batch_correct = function(data) {
  # Find the median of the QCs across plates
  target_median = median(data$Precursor.Normalised[data$Strain == "QC"])

  # Next we iterate over the plates and for each we get a normalization factor that we apply to its measurements afterwards
  data$Precursor.Batch.Corrected = NA
  for (i in 1:6) {
    tmp = data %>% filter(Plate_ID == i)
    plate_median = median(tmp$Precursor.Normalised)
    norm_factor = plate_median/target_median
    data$Precursor.Batch.Corrected[data$Plate_ID == i] = data$Precursor.Normalised[data$Plate_ID == i]/norm_factor
  }
  data$Precursor.Batch.Corrected[data$Plate_ID == "QC"] = data$Precursor.Normalised[data$Plate_ID == "QC"]
  return(data)
}

data_batch_corrected = batch_correct(data_filtered_by_QC_CV)

# Get new boxplots by plate and see if batch correction changed anything
ggplot(data = data_batch_corrected, aes(x = Plate_ID, y = log2(Precursor.Batch.Corrected), group = Plate_ID)) +
  geom_boxplot(outlier.size = 0.5) +
  labs(title = "After batch correction") +
  xlab("Plate") +
  theme_light()
```

```
# 8. Number of precursors per protein
Do not filter based on this, but have a look at the distribution of the number of precursors per protein

## Calculate the amount of precursor per protein
```{r}
check_number_of_precursors_per_protein = function(data) {
  data$Precursor.Id = as.factor(data$Precursor.Id)
  data = data %>%
    group_by(Protein.Ids) %>%
    mutate(Precursor.Per.Protein = length(unique(Precursor.Id))) %>%
    ungroup()
  return(data)
}

precursors_per_protein = check_number_of_precursors_per_protein(data_batch_corrected)
```

Obtain a version of this data to create plots from, and generate the plots
```{r}
# Get plotting dataset
temp = precursors_per_protein %>%
  distinct(Protein.Ids, .keep_all = T) %>%
  select(Protein.Ids, Precursor.Per.Protein, Genes)

# Precursor for each protein
ggplot(data = temp, aes(x = Protein.Ids, y = Precursor.Per.Protein)) +
  geom_point(size = 0.5) +
  theme_light() +
  theme(axis.text.x=element_blank(),
        axis.ticks.x=element_blank()) +
  geom_text_repel(data = subset(temp, Precursor.Per.Protein >= 40),
                  aes(x = Protein.Ids, y = Precursor.Per.Protein, label = Genes)) +
  xlab("Proteins") +
  ylab("Precursors per protein")

# Histogram of precursor per protein
ggplot(data = temp, aes(x = Precursor.Per.Protein)) +
  geom_histogram(bins = 92, col = "black", fill = "grey") +
  theme_light() +
  xlab("Precursors per protein") +
  ylab("Count") +
  geom_vline(xintercept = mean(precursors_per_protein$Precursor.Per.Protein), col = "blue") +
  geom_vline(xintercept = median(precursors_per_protein$Precursor.Per.Protein), col = "red") +
  annotate("text", x = 20, y = 450, label = "Mean", col = "blue") +
  annotate("text", x = 6, y = 450, label = "Median", col = "red")

table(temp$Precursor.Per.Protein)
```


9. Peptide-to-protein quantification using maxLFQ
```{r}
protein_quantified = diann_maxlfq(data_batch_corrected,
                                  sample.header = "File.Name",
                                  group.header = "Genes",
                                  id.header = "Precursor.Id",
                                  quantity.header = "Precursor.Batch.Corrected")
protein_quantified_df = data.frame(protein_quantified)
protein_quantified_df$Genes = rownames(protein_quantified_df)
```
```

B.4 Processing DIA-NN reports for strain-specific approach

```
Packages
```{r}
library(data.table)
library(dplyr)
library(readODS)
library(kableExtra)
library(gridExtra)
library(Cairo)
library(ggplot2)
library(glue)
library(gt)
library(ggvenn)
library(ggrepel)
library(diann)
```


Load data
```{r}
# Reports
files_path = '/~/matched_precursor_reports/'
files = list.files(files_path, full.names = T)

## Grab the names of the dataframes (the strain names)
names <- c()
full_new_names <- c()
for (file in files) {
  start = str_locate(file, "06062024_")[2] + 1
  end = str_locate(file, "_matched.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}

datas <- lapply(files, fread)
datas <- lapply(datas, as.data.frame)
names(datas) <- names

# Sample correspondences
sample_correspondence <- fread("/~/sample_correspondence.tsv")
sample_correspondence <- as.data.frame(sample_correspondence)
sample_correspondences <- rep(list(sample_correspondence), length(datas))
names(sample_correspondences) <- names

# Stats files
files_path = '/~/stats_files/'
files = list.files(files_path, full.names = T)

stats_files <- lapply(files, fread)
stats_files <- lapply(stats_files, as.data.frame)
names(stats_files) <- names
```


0. Set up
0.1. Parameters
```{r}
OD_threshold = 0.12
Q_values_threshold = 0.01
```
```

```
min_samples_per_strain = 3
percentage_of_samples_per_precursor = 0.65
z_score_limit = 3
quantile_limit_QC_CV = 0.9
```


1. Remove samples with low OD
```{r}
# Create function
filter_based_on_OD = function(data, OD_threshold) {
  data = data %>% filter(OD_at_harvest > OD_threshold | Strain == "QC")
  return(data)
}

# Run filtering
datas_filtered_OD = lapply(datas, filter_based_on_OD, OD_threshold)
```


2. Remove non-proteotypic peptides
```{r}
# Create function
filter_proteotypic = function(data) {
  data = data %>% filter(Proteotypic == 1)
  return(data)
}

# Run filtering
datas_filtered_proteotypic = lapply(datas_filtered_OD, filter_proteotypic)
```


3. Filter based on Q-values
```{r}
# Create function
filter_Q_values = function(data, Q_values_threshold) {
  data = data %>% filter(Q.Value < Q_values_threshold,
                         PG.Q.Value < Q_values_threshold,
                         Global.Q.Value < Q_values_threshold,
                         Global.PG.Q.Value < Q_values_threshold)
  return(data)
}

# Run filtering
datas_filtered_Q = lapply(datas_filtered_proteotypic, filter_Q_values, Q_values_threshold)
```


4. Filter based on z-score of TIC and number of precursors identified
4.0. Remove all samples that were already removed by this point
```{r}
remove_filtered_samples_from_stats_file = function(stats_file, data_filtered_Q) {
  stats_file = stats_file[stats_file$File.Name %in% data_filtered_Q$File.Name,]
  return(stats_file)
}

modified_stats_files <- mapply(FUN = remove_filtered_samples_from_stats_file, stats_file =
↪  stats_files,  data_filtered_Q = datas_filtered_Q, SIMPLIFY = F)
```

4.1. Exploration regarding TIC and number of identified precursors
Calculate z-score and robust z-score for TIC as new columns in the stats file

```{r}
modify_stats_file_add_z_scores = function(stats_file, data) {
  # Z-scores for TIC
  stats_file = stats_file %>%
    mutate(MS1.Signal = as.numeric(MS1.Signal)) %>%
    mutate(z_score_tic = (MS1.Signal - mean(MS1.Signal))/sd(MS1.Signal)) %>%
    mutate(robust_z_score_tic = (MS1.Signal - median(MS1.Signal))/mad(MS1.Signal)) %>%
    mutate(QC = as.factor(case_when(data$Strain[match(File.Name, data$File.Name)] == "QC" ~ 1,
                          TRUE ~ 0)))

  # Z-scores for number of precursors identified
  stats_file = stats_file %>%
    mutate(Precursors.Identified = as.numeric(Precursors.Identified)) %>%
    mutate(z_score_pept_num = (Precursors.Identified - mean(Precursors.Identified))/sd(Precursors.Identified)) %>%
    mutate(robust_z_score_pept_num = (Precursors.Identified - median(Precursors.Identified))/sd(Precursors.Identified))
}

modified_stats_files <- mapply(FUN = modify_stats_file_add_z_scores, stats_file = modified_stats_files, data = datas_filtered_Q, SIMPLIFY = F)
```

4.2. Perform the filtering
```{r}
# Create function
filter_TIC_and_peptide_number = function(data, stats_file, SD_limit_for_TIC_filtering) {
  # Filter on the stats file
  stats_file_filtered = stats_file %>% filter(robust_z_score_tic < z_score_limit & robust_z_score_tic > -z_score_limit,
                                              robust_z_score_pept_num < z_score_limit & robust_z_score_pept_num > -z_score_limit)

  # Filter on the actual dataset based on the stats file
  data = data[data$File.Name %in% stats_file_filtered$File.Name,]

  return(data)
}

# Run filtering
datas_filtered_TIC = mapply(FUN = filter_TIC_and_peptide_number, data = datas_filtered_Q, stats_file = modified_stats_files, SIMPLIFY = F)
```


5. Filter based on detection threshold/sample fraction
5.1. Perform filtering based on number of samples present per strain
```{r}
filter_detection_threshold <- function(data_filtered_TIC, min_samples_per_strain) {
  test_replicate_num = data_filtered_TIC %>%
    group_by(Strain) %>%
    mutate(sample_count = length(unique(Sample))) %>%
    filter(sample_count >= min_samples_per_strain)
  return(test_replicate_num)
}

datas_filtered_replicate_num <- mapply(filter_detection_threshold, data_filtered_TIC = datas_filtered_TIC, min_samples_per_strain = min_samples_per_strain, SIMPLIFY = F)
```

5.4. Remove, for each strain, those precursors which are not present in at least 3/4 or 2/3 replicates
```{r}
```

```
# Create function
remove_uncommon_precursors_per_strain = function(data, percentage_of_samples_per_precursor) {

  # Set up the filter
  filterSF <- data %>%
    group_by(Precursor.Id, Strain) %>%
    summarise(count = n()) %>%
    ungroup() %>%
    group_by(Strain) %>%
    mutate(maxCount=max(count))

  # Apply filter
  out = data %>% left_join(filterSF) %>% filter(count >= percentage_of_samples_per_precursor*maxCount)

  return(out)
}

# Filter
datas_filtered_prec_per_strain <- mapply(remove_uncommon_precursors_per_strain, data = datas_filtered_replicate_num, percentage_of_samples_per_precursor = percentage_of_samples_per_precursor, SIMPLIFY = F)
```


6. Filter based on precursor CV
6.1. First of all I need to calculate the CV for each precursor in each strain
```{r}
create_CV_data = function(data) {
  CV_data = data %>%
    group_by(Precursor.Id) %>%
    mutate("SD" = sd(Precursor.Normalised, na.rm = T), "CV" = sd(Precursor.Normalised, na.rm = T)/mean(Precursor.Normalised, na.rm = T))
  return(CV_data)
}

CV_datas = mapply(create_CV_data, data = datas_filtered_prec_per_strain, SIMPLIFY = F)
```

Density plot
```{r}
# First of all create a dataframe from which I can plot this
CV_data <- data.frame(matrix(nrow = max(as.numeric(lapply(CV_datas, nrow))), ncol = length(CV_datas)))
colnames(CV_data) <- names(datas)
for (i in 1:length(CV_datas)) {
  strain <- names(CV_datas)[i]
  CV_data[,i] <- c(CV_datas[[strain]]$CV, rep(NA, nrow(CV_data) - nrow(CV_datas[[strain]])))
}

CV_data_long <- CV_data %>% pivot_longer(cols = everything(), names_to = "Strain", values_to = "Counts")
CV_data_long <- na.omit(CV_data_long)

# Plot
ggplot(data = CV_data_long) +
  geom_density(aes(x = Counts, color = Strain)) +
  xlab("CV") +
  ylab("Density") +
  theme_light() +
  theme(legend.position = "none")

# Do the same but actually color QC, BY4741-ki, and then all other strains
```

```
CV_data_long <- CV_data_long %>%
  mutate(Strain_original = Strain) %>%
  mutate(Strain = case_when(Strain_original == "QC" ~ "QC",
                            Strain_original == "BY4741_ki" ~ "BY4741_ki",
                            TRUE ~ "Other"))
ggplot(data = CV_data_long) +
  geom_density(aes(x = Counts, color = Strain)) +
  xlab("CV") +
  ylab("Density") +
  theme_light()
```


6.2. Filtering
Remove from all samples the precursors which have a large CV in the QCs
```{r}
# I have to do this separately so as to be able to save these values
produce_limit_CV_values_per_strain <- function(data) {
  limit_value <- quantile(data$CV, probs = c(quantile_limit_QC_CV))
  return(limit_value)
}
CV_cutoffs <- mapply(produce_limit_CV_values_per_strain, data = CV_datas, SIMPLIFY = F)


# Now actually perform the filtering
filter_CV <- function(data) {
  limit_value <- quantile(data$CV, probs = c(quantile_limit_QC_CV))
  data <- data %>% filter(CV <= limit_value)
  return(data)
}
datas_CV_filtered <- mapply(filter_CV, data = CV_datas, SIMPLIFY = F)

# Make a plot of the CV cutoffs
CV_cutoffs_df <- as.data.frame(t(as.data.frame(CV_cutoffs)))
CV_cutoffs_df$Strain <- rownames(CV_cutoffs_df)
colnames(CV_cutoffs_df) <- c("cutoffs", "Strain")
CV_cutoffs_df <- CV_cutoffs_df %>%
  mutate(QC = case_when(Strain == "BY4741_ki" ~ "QC",
                        TRUE ~ "Other"))
ggplot(data = CV_cutoffs_df, aes(x = Strain, y = cutoffs)) +
  geom_point() +
  theme_light() +
  theme(axis.text.x=element_blank(),
        axis.ticks.x=element_blank()) +
  geom_text_repel(data = subset(CV_cutoffs_df, cutoffs > 0.6),
                  aes(x = Strain, y = cutoffs, label = Strain)) +
  ylab("CV cutoff")
```


7. Run maxlfq and save the resulting dataframes
```{r}
## Create the function which will run maxlfq and write the corresponding output to the
↪ corresponding directory
run_maxlfq_strain_specific <- function(data, strain_name, output_dir_path) {
  if (nrow(data) > 0) {
    protein_quantified <- diann_maxlfq(data,
                                       sample.header = "Sample",
                                       group.header = "Protein.Names",
                                       id.header = "Precursor.Id",
                                       quantity.header = "Precursor.Normalised")
    protein_quantified_df = data.frame(protein_quantified)
    protein_quantified_df$Genes = rownames(protein_quantified_df)
```

```

    # Come up with the path and name of where I will save this file
    output_file <- paste0(output_dir_path, strain_name, "_protein_level", ".tsv")

    # Save new protein-level data
    fwrite(protein_quantified_df, output_file)
  }
  else {print(glue('Strain {strain_name} could not be processed since its report file is empty. A unique_genes dataset for this strain was not generated.'))}
}

## Run the function, doesn't show any output but it runs maxlfq and writes the files to the specified directory
output_dir_path <- "/~/protein_level_reports/"
mapply(run_maxlfq_strain_specific, data = datas, strain_name = names(datas), output_dir_path = output_dir_path)
```

B.5 Compare number of identified proteins between approaches

```
Packages
```{r}
library(dplyr)
source("/~/0. prepare_data_functions.R")
```

0. Load data
```{r}
# Independently pre-processed strain-specific reports
## Get file names
files_path = '/~/protein_level_reports'
files = list.files(files_path, full.names = T)

## Load them and grab the strain names
names <- c()
for (file in files) {
  start = str_locate(file, "protein_level_reports/")[2] + 1
  end = str_locate(file, "_protein_level.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}
ss_datas <- lapply(files, fread)
ss_datas <- lapply(ss_datas, as.data.frame)
names(ss_datas) <- names


# Independently pre-processed common approach reports
## Get file names
files_path = '/~/individual_reports_per_strain_CA_after_maxlfq'
files = list.files(files_path, full.names = T)

## Load them and grab the strain names
names <- c()
for (file in files) {
  start = str_locate(file, "individual_reports_per_strain_CA_after_maxlfq/")[2] + 1
  end = str_locate(file, "_per_strain_CA_after_maxlfq.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}
ca_datas <- lapply(files, fread)
ca_datas <- lapply(ca_datas, as.data.frame)
```
```

```
names(ca_datas) <- names

## I need to turn the Gene column into rownames
genes_to_rownames <- function(data) {
  rownames(data) <- data$Genes
  data <- data %>% dplyr::select(-Genes)
  return(data)
}
ca_datas <- lapply(ca_datas, genes_to_rownames)


# Sample correspondence
sample_correspondence <- fread("/~/sample_correspondence.tsv")
sample_correspondence <- as.data.frame(sample_correspondence)


# Load the data about ploidy and process it a bit
load('/~/strains_in_each_type_vectors.Rdata')
diploid_strains <- unique(diploid_strains)
haploid_strains <- c(haploid_strains, "BY4741_ki", "QC")

ploidy_info <- data.frame(c(haploid_strains, diploid_strains, polyploid_strains),
↪  c(rep("Haploid", length(haploid_strains)), rep("Diploid", length(diploid_strains)),
↪  rep("Polyploid", length(polyploid_strains))))
colnames(ploidy_info) <- c("Strain", "Ploidy")
```

1. Come up with the results table

```{r}
p.vals <- c()
direction <- c()
CA_mean <- c()
SS_mean <- c()

for (i in 1:length(ca_datas)) {
  strain <- names(ca_datas)[i]

  temp_ca <- ca_datas[[strain]]
  temp_ss <- ss_datas[[strain]]

  counts_ca <- apply(temp_ca, 2, function(x) sum(!(is.na(x))))
  counts_ss <- apply(temp_ss, 2, function(x) sum(!is.na(x)))

  # t-test and save p-value, also direction of difference
  if (length(counts_ca) > 1 & length(counts_ss) > 1) {

    p.vals <- c(p.vals, t.test(counts_ca, counts_ss)$p.value)
    CA_mean <- c(CA_mean, mean(counts_ca))
    SS_mean <- c(SS_mean, mean(counts_ss))

    if (mean(counts_ca) > mean(counts_ss)) {
      direction <- c(direction, "CA")
    }
    else {
      direction <- c(direction, "SS")
    }
  }
  else {
    p.vals <- c(p.vals, NA)
    direction <- c(direction, NA)
    CA_mean <- c(CA_mean, NA)
    SS_mean <- c(SS_mean, NA)
  }
```

```
}

results_processed_separately <- data.frame(names(ca_datas), p.vals, direction, CA_mean, SS_mean)
colnames(results_processed_separately) <- c("Strain", "p.val", "Approach_with_more_identified_proteins", "Mean_proteins_in_CA", "Mean_proteins_in_SS")
results_processed_separately$p.vals.corrected <- p.adjust(results_processed_separately$p.val, method = "BH")
results_processed_separately$Effect_size <- results_processed_separately$Mean_proteins_in_CA - results_processed_separately$Mean_proteins_in_SS
results_processed_separately$log10pval <- -log10(results_processed_separately$p.vals.corrected)
```

2. Add information about the ploidy of each strain
```{r}
results_processed_separately <- left_join(results_processed_separately, ploidy_info, by = join_by(Strain))
results_processed_separately$Ploidy[results_processed_separately$Strain == "QC"] <- "QC"
results_processed_separately$Ploidy[results_processed_separately$Strain == "BY4741_ki"] <- "Haploid"
```

3. Plot
Plot of the p-values for each strain, colored per which approach discovers more proteins
```{r}
ggplot(data = results_processed_separately, aes(x = Strain, y = p.vals.corrected, col = Approach_with_more_identified_proteins)) +
  geom_point() +
  geom_hline(yintercept = 0.05) +
  theme_light() +
  theme(axis.text.x=element_blank(),
        axis.ticks.x=element_blank(),
        legend.position = "none")
```

4. Create volcano plots
```{r}
results_processed_separately <- results_processed_separately %>%
  mutate(Effect_size_SS_positive = -Effect_size) %>%
  mutate(Effect_size_SS_positive_perc = Effect_size_SS_positive/Mean_proteins_in_CA)

results_processed_separately_final <- results_processed_separately %>%
  filter(Strain != "QC")

ggplot(data = results_processed_separately_final, aes(x = Effect_size_SS_positive, y = log10pval, col = Ploidy)) +
  geom_point() +
  geom_hline(yintercept = -log10(0.01), col = "red") +
  ylab("-log10(p.value)") +
  xlab("Amount of new proteins found in SSA compared to CA") +
  #labs(title = "Absolute value") +
  geom_text_repel(data = subset(results_processed_separately_final, Effect_size_SS_positive < 0 | log10pval > 10),
                  aes(x = Effect_size_SS_positive, y = log10pval, col = Ploidy, label = Strain))

ggplot(data = results_processed_separately_final, aes(x = Effect_size_SS_positive_perc, y = log10pval, col = Ploidy)) +
  geom_point() +
  geom_hline(yintercept = -log10(0.01), col = "red") +
  ylab("-log10(p.value)") +
  xlab("Amount of new proteins found in SSA as a % of proteins found in CA") +
  #labs(title = "Percentage of total proteins found in CA") +
  geom_text_repel(data = subset(results_processed_separately_final, Effect_size_SS_positive_perc < 0 | log10pval > 10),
```

```
                    aes(x = Effect_size_SS_positive_perc, y = log10pval, col = Ploidy, label =
                    ↪  Strain))
```

B.6 Allele-specific expression

```
# 0. Load data and get it ready
## 0.1. Load all the dataframes as a list
```{r}
# Reports
files_path = '/~/matched_precursor_reports'
files = list.files(files_path, full.names = T)

## Grab the names of the dataframes (the strain names)
names <- c()
for (file in files) {
  start = str_locate(file, "matched_precursor_reports/Run_1_test_06062024_")[2] + 1
  end = str_locate(file, "_matched.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}

## Actually load the dataframes
setwd(files_path)
datas <- lapply(files, fread)
datas <- lapply(datas, as.data.frame)
names(datas) <- names

# Repeat this for this information already turned to protein level
files_path = "/~/protein_level_reports"
files = list.files(files_path, full.names = T)

## Grab the names of the dataframes (the strain names)
names <- c()
for (file in files) {
  start = str_locate(file, "protein_level_reports/")[2] + 1
  end = str_locate(file, "_protein_level.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}

## Actually load the dataframes
setwd(files_path)
datas_protein_level <- lapply(files, fread)
datas_protein_level <- lapply(datas_protein_level, as.data.frame)
names(datas_protein_level) <- names

## Set protein names as rownames
for (i in 1:length(datas_protein_level)) {
  rownames(datas_protein_level[[i]]) <- datas_protein_level[[i]]$Genes
  datas_protein_level[[i]] <- datas_protein_level[[i]] %>% select(-Genes)
}


# Sample correspondence
sample_correspondence <- fread("/~/sample_correspondence.tsv")
sample_correspondence <- as.data.frame(sample_correspondence)


# Stats files
files_path = '/~/stats_files/'
```
```

```
files = list.files(files_path, full.names = T)

setwd(files_path)
stats_files <- lapply(files, fread)
stats_files <- lapply(stats_files, as.data.frame)
names(stats_files) <- names

# Remove unnecessary variables
rm(list = c("end", "file", "files", "files_path", "start", "strain"))
```

0.2. Load information on which strains are haploid, diploid or polyploid

```{r}
load('/~/strains_in_each_type_vectors.Rdata')
```

0.3. Create separate lists for haploid, diploid and polyploid strains

```{r}
# Remember that QCs and BY4741-ki are not included in any of these!!
datas_haploid <- datas[names(datas) %in% haploid_strains]
datas_diploid <- datas[names(datas) %in% diploid_strains]
datas_polyploid <- datas[names(datas) %in% polyploid_strains]
```

1. Allele-specific expression - Proteins different across haplotypes in heterozygous diploid strains

1.1. Load and prepare data

Load the reference JSON file

```{r}
diploids_dict <- fromJSON(file = "/~/final_diploids_dict.json", simplify = FALSE)
```

1.2. Look at the unique peptides to each HP and those common to both, for each protein in each strain

1.2.1. Collect the information from the strain-specific reports and put it into nested lists

```{r}
# Define the list where I´ll collect my output
results_diploids_list <- list()

# Iterate over strains
strains = intersect(names(diploids_dict), names)
for (i in 1:length(strains)) {
  strain <- strains[i]
  strain_list <- list()

  # Get the common proteins for this strain
  common_proteins <- names(diploids_dict[[strain]][["common_prots_diff"]])

  # For each of these proteins, get 3 vectors, containing the respective peptides of this protein, classified in the 3 types
  for (j in 1:length(common_proteins)) {
    protein <- common_proteins[j]
    peptides_common <- unlist(diploids_dict[[strain]][["common_prots_diff"]][[protein]][["common_peptides"]])
    peptides_hp1 <- unlist(diploids_dict[[strain]][["common_prots_diff"]][[protein]][["common_HP1_peptides"]])
    peptides_hp2 <- unlist(diploids_dict[[strain]][["common_prots_diff"]][[protein]][["common_HP2_peptides"]])

```

```
    # Create 2 datasets by filtering the report of this strain based on Stripped.Sequence: one with sequences from the peptides unique to HP1 and the other for HP2
    temp_hp1 <- datas[[strain]] %>%
      filter(Stripped.Sequence %in% peptides_hp1) %>%
      filter(Proteotypic == 1)
    temp_hp2 <- datas[[strain]] %>%
      filter(Stripped.Sequence %in% peptides_hp2) %>%
      filter(Proteotypic == 1)
    temp_common <- datas[[strain]] %>%
      filter(Stripped.Sequence %in% peptides_common) %>%
      filter(Proteotypic == 1)

    # For HP1 report, if it is not empty, get the values of Precursor.Quantity across these peptides
    if (nrow(temp_hp1) > 0) {
      hp1_Precursor.Quantity <- as.numeric(temp_hp1$Precursor.Quantity)
      hp1_Precursor.Quantity <- data.frame(hp1_Precursor.Quantity, temp_hp1$Stripped.Sequence, temp_hp1$Precursor.Id, temp_hp1$Modified.Sequence, temp_hp1$File.Name)
      colnames(hp1_Precursor.Quantity) <- c("Precursor.Quantity", "Stripped.Sequence", "Precursor.Id", "Modified.Sequence", "File.Name")
    }
    else {
      hp1_Precursor.Quantity <- c(0)
    }

    # Same for HP2
    if (nrow(temp_hp2) > 0) {
      hp2_Precursor.Quantity <- as.numeric(temp_hp2$Precursor.Quantity)
      hp2_Precursor.Quantity <- data.frame(hp2_Precursor.Quantity, temp_hp2$Stripped.Sequence, temp_hp2$Precursor.Id, temp_hp2$Modified.Sequence, temp_hp2$File.Name)
      colnames(hp2_Precursor.Quantity) <- c("Precursor.Quantity", "Stripped.Sequence", "Precursor.Id", "Modified.Sequence", "File.Name")
    }
    else {
      hp2_Precursor.Quantity <- c(0)
    }

    # Same for common peptides
    if (nrow(temp_common) > 0) {
      common_Precursor.Quantity <- as.numeric(temp_common$Precursor.Quantity)
      common_Precursor.Quantity <- data.frame(common_Precursor.Quantity, temp_common$Stripped.Sequence, temp_common$Precursor.Id, temp_common$Modified.Sequence, temp_common$File.Name)
      colnames(common_Precursor.Quantity) <- c("Precursor.Quantity", "Stripped.Sequence", "Precursor.Id", "Modified.Sequence", "File.Name")
    }
    else {
      common_Precursor.Quantity <- c(0)
    }

    # Save these values to the strain list
    strain_list[[protein]] <- list(HP1 = hp1_Precursor.Quantity, HP2 = hp2_Precursor.Quantity, common = common_Precursor.Quantity)
  }

  # Save the list created for this strain to the full list
  results_diploids_list[[strain]] <- strain_list
}
```

- Keep only proteins for which we detect common peptides, peptides from HP1 and peptides from HP2

```{r}
```

```
results_diploids_list_filtered <- list()

for (i in 1:length(results_diploids_list)) {
  strain <- names(results_diploids_list[i])
  strain_list <- list()
  for (j in 1:length(results_diploids_list[[strain]])) {
    protein <- names(results_diploids_list[[strain]])[j]
    hp1_peptides <- results_diploids_list[[strain]][[protein]][["HP1"]]
    hp2_peptides <- results_diploids_list[[strain]][[protein]][["HP2"]]
    common_peptides <- results_diploids_list[[strain]][[protein]][["common"]]
    if (class(hp1_peptides) == "data.frame" & class(hp2_peptides) == "data.frame" & class(common_peptides) == "data.frame") {
      strain_list[[protein]] <- results_diploids_list[[strain]][[protein]]
    }
  }
  results_diploids_list_filtered[[strain]] <- strain_list
}
```

- Keep only proteins for which we detect peptides from HP1 and peptides from HP2 (do not care about common ones anymore)

```{r}
results_diploids_list_only_hps <- list()
precursors_found_for_each_protein_in_each_hp <- c()

for (i in 1:length(results_diploids_list)) {
  strain <- names(results_diploids_list[i])
  strain_list <- list()
  for (j in 1:length(results_diploids_list[[strain]])) {
    protein <- names(results_diploids_list[[strain]])[j]
    hp1_peptides <- results_diploids_list[[strain]][[protein]][["HP1"]]
    hp2_peptides <- results_diploids_list[[strain]][[protein]][["HP2"]]
    if (class(hp1_peptides) == "data.frame" & class(hp2_peptides) == "data.frame") {
      strain_list[[protein]] <- results_diploids_list[[strain]][[protein]][names(results_diploids_list[[strain]][[protein]]) != "common"]
      precursors_found_for_each_protein_in_each_hp <- c(precursors_found_for_each_protein_in_each_hp,
         length(unique(results_diploids_list[[strain]][[protein]][["HP1"]]$Precursor.Id)),
         length(unique(results_diploids_list[[strain]][[protein]][["HP2"]]$Precursor.Id)))
    }
  }
  if (length(strain_list) > 0) {
   results_diploids_list_only_hps[[strain]] <- strain_list
  }
}
```

- Keep proteins where any peptide is detected at all

```{r}
results_diploids_detected <- list()

for (i in 1:length(results_diploids_list)) {
  strain <- names(results_diploids_list[i])
  strain_list <- list()
  for (j in 1:length(results_diploids_list[[strain]])) {
    protein <- names(results_diploids_list[[strain]])[j]
    hp1_peptides <- results_diploids_list[[strain]][[protein]][["HP1"]]
    hp2_peptides <- results_diploids_list[[strain]][[protein]][["HP2"]]
    if (class(hp1_peptides) == "data.frame" | class(hp2_peptides) == "data.frame" | class(common_peptides) == "data.frame") {
      strain_list[[protein]] <- results_diploids_list[[strain]][[protein]]
    }
```

```
  }
  results_diploids_detected[[strain]] <- strain_list
}
```

Create some barplots which show how many proteins we are keeping and how many we are removing because there are no peptides recognised for them from both HPs

```{r}
# Create empty dataframe
kept_proteins_og <- data.frame(matrix(nrow = 0, ncol = 5))
colnames(kept_proteins_og) <- c("Strain", "Total proteins based on FASTAs", "Total_proteins_detected", "Proteins_with_observed_peptides_from_both_HPs", "Proteins_with_observed_peptides_from_both_HPs_and_common")

# Iterate over strains
for (i in 1:length(diploids_dict)) {
  strain <- names(diploids_dict)[i]

  if (strain %in% names(results_diploids_detected)) {
    # Figure out the number of proteins at different points for this strain
    total_prots <- length(diploids_dict[[strain]][["common_prots_diff"]])
    kept_prots_detected <- length(results_diploids_detected[[strain]])
    kept_prots_HPs <- length(results_diploids_list_only_hps[[strain]])
    kept_prots_HPs_and_common <- length(results_diploids_list_filtered[[strain]])

    # Bring these together and add them as a new row to the output dataframe
    kept_proteins_og[nrow(kept_proteins_og)+1,] <- c(strain, total_prots, kept_prots_detected, kept_prots_HPs, kept_prots_HPs_and_common)
  }
}

# Change colnames for legend
colnames(kept_proteins_og) <- c("Strain", "Total proteins based on FASTAs", "Total proteins with at least 1 precursor detected", "Proteins for which peptides are observed coming from both HPs", "Proteins for which peptides are observed coming from both HPs, and also common")

# Get dataframe into longer format
kept_proteins_og <- kept_proteins_og %>% pivot_longer(!Strain, names_to = "Type", values_to = "Count")
kept_proteins_og$Count <- as.numeric(kept_proteins_og$Count)

# Plot this
ggplot(data = kept_proteins_og, aes(x = reorder(Strain, Count), y = Count, fill = Type)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  theme_light() +
  theme(legend.position = "none") +
  labs(title = "Number of proteins present in both HPs") +
  xlab("Strains") +
  ylab("Number of proteins")
```


 # 1.2.2. Compare the actual amounts of Precursor.Quantity that I find for the precursors coming from each HP for each protein (within each strain of course)

Come up with a list where each entry is a strain, and for it we have a dataframe with, in each row a protein, and the p-values and corrected p-values from testing the Precursor.Quantitys we have for that protein between HPs

```{r}
numerical_comparison_list <- list()

# Iterate over strains
for (i in 1:length(results_diploids_list_only_hps)) {
  strain <- names(results_diploids_list_only_hps)[i]
```

```
  pvals_strain <- c()
  protein_names <- c()
  most_abundant_hp <- c()

  # Iterate over proteins
  for (j in 1:length(results_diploids_list_only_hps[[strain]])) {
    protein <- names(results_diploids_list_only_hps[[strain]])[j]

    # Get vectors with the Precursor.Quantity values found for this protein in each HP
    hp1 <- as.numeric(results_diploids_list_only_hps[[strain]][[protein]][["HP1"]]$Precursor.Quantity)
    hp2 <- as.numeric(results_diploids_list_only_hps[[strain]][[protein]][["HP2"]]$Precursor.Quantity)

    # If both have more than 1 value then we can do a t-test, otherwise not :(
    if (length(hp1) > 1 & length(hp2) > 1) {
      p <- t.test(hp1, hp2)$p.value                 # Need to store these somewhere and correct them together for multiple testing
      pvals_strain <- c(pvals_strain, p)
      protein_names <- c(protein_names, protein)

      # Check which HP has the highest abundance for this protein so as to record it for later
      if (mean(hp1) > mean(hp2)) {most_abundant_hp <- c(most_abundant_hp, "HP1")}
      else if (mean(hp2) > mean(hp1)) {most_abundant_hp <- c(most_abundant_hp, "HP2")}
    }
  }
  # Apply multiple testing correction for this strain
  if (length(pvals_strain) > 0) {
    corrected_pvals <- p.adjust(pvals_strain, method = "BH")
  }
  strain_df <- data.frame(pvals_strain, corrected_pvals, protein_names, most_abundant_hp)
  colnames(strain_df) <- c("pvals", "pvals_corrected", "proteins", "hp_with_higher_abundance")
  numerical_comparison_list[[strain]] <- strain_df
}
```

Now check the p-values and save the proteins and strains for which we have obtained significant p-values

```{r}
# Create empty dataframe for output
significant_proteins_df <- data.frame(matrix(ncol = 5, nrow = 0))
colnames(significant_proteins_df) <- c(colnames(numerical_comparison_list[[1]]), "Strain")

# Iterate over strains and check which proteins had significant p-values, then add these to the dataframe created above
for (i in 1:length(numerical_comparison_list)) {
  strain <- names(numerical_comparison_list)[i]
  temp <- numerical_comparison_list[[i]]
  for (j in 1:nrow(temp)) {
    if (temp$pvals_corrected[j] < 0.05) {
      significant_proteins_df[nrow(significant_proteins_df)+1,] <- c(temp[j,], strain)
    }
  }
}
```

Get the gene names of these proteins

```{r}
## Load the reference table from SGD
yeastmine_tab <- fread(file = "/~/yeastmine_results.tsv",
                       sep="\t",
                       fill=T,
                       header=T)
```

```

# Turn protein_1/protein_2 protein names into only protein_1, just so that they are taken into account for the GO analysis - also add a column which serves as indicator for which proteins we did this to, since otherwise we would lose this information
significant_proteins_df <- significant_proteins_df %>%
  mutate(Gene.secondaryIdentifier = case_when(grepl("/", proteins) ~ substr(proteins, 0, str_locate(proteins, "/")-1),
                                              TRUE ~ proteins)) %>%
  mutate(was_more_than_1_isoform = case_when(grepl("/", proteins) ~ "Yes",
                                              TRUE ~ "No"))

# Use the function I created in a different file to get the gene names
significant_proteins_df <- match_systematic_to_standard_protein_names(significant_proteins_df, yeastmine_tab, simplify = F, add_extra_columns = T)
```

Test and get p-values
```{r}
# Create another version of this list, where for each strain we only keep the proteins which are differentiated between HPs - this filters out all strains which are not heterozygous diploids
datas_protein_unnormalized_HPs <- list()
for (i in 1:length(datas_protein_level)) {
  strain <- names(datas_protein_level)[i]
  df <- datas_protein_level[[i]]
  df <- df[grepl("_common_", rownames(df)),]
  if (nrow(df) > 0) {
   datas_protein_unnormalized_HPs[[strain]] <- df
  }
}

# For each strain, go through the rownames (protein names) and remove the _common_HP part, leave only the protein name. Then iterate through them and for those for which we have both versions, perform a t-test on the amounts found
diploids_results_final <- list()
for (i in 1:length(datas_protein_unnormalized_HPs)) {
  strain <- names(datas_protein_unnormalized_HPs)[i]
  df <- datas_protein_unnormalized_HPs[[i]]

  # Get unique protein names
  full_protein_names <- rownames(df)
  protein_names <- c()
  for (i in 1:length(full_protein_names)) {
    protein_name <- str_match(full_protein_names[i], "(.*)_common")
    protein_names <- c(protein_names, protein_name)
  }
  protein_names <- unique(protein_names)

  # Iterate over the unique protein names
  strain_df <- data.frame(matrix(ncol = 3, nrow = 0))
  colnames(strain_df) <- c("protein", "p", "higher_hp")
  for (i in 1:length(protein_names)) {
    protein_name_1 <- paste(protein_names[i], "_common_HP1", sep = "")
    protein_name_2 <- paste(protein_names[i], "_common_HP2", sep = "")

    # If the version of the protein for both haplotypes is present, perform a t-test and add a row to the df for this strain
    if (protein_name_1 %in% full_protein_names & protein_name_2 %in% full_protein_names) {
      hp1_values <- na.omit(as.numeric(df[rownames(df) == protein_name_1,]))
      hp2_values <- na.omit(as.numeric(df[rownames(df) == protein_name_2,]))
      if (length(hp1_values) > 1 & length(hp2_values) > 1) {
        p <- t.test(hp1_values, hp2_values)$p.value
        if (mean(hp1_values) > mean(hp2_values)) {higher_hp <- "HP1"}
```

```
        else if (mean(hp2_values) > mean(hp1_values)) {higher_hp <- "HP2"}
        strain_df[nrow(strain_df)+1,] <- c(protein_names[i], p, higher_hp)
      }
    }
  }
  strain_df$p.corrected <- p.adjust(strain_df$p, method = "BH")
  diploids_results_final[[strain]] <- strain_df
}

# Create a subset of this list with only the significant p-values
diploids_results_final_significant <- list()
for (i in 1:length(diploids_results_final)) {
  strain <- names(diploids_results_final)[i]
  df <- diploids_results_final[[i]]

  if (sum(df$p.corrected < 0.05) > 0) {
    df_new <- df %>% filter(p.corrected < 0.05)
    diploids_results_final_significant[[strain]] <- df_new
  }
}
```

Plot this

```{r}
# Add the number of significant proteins to the first barplot from before, so we see how few of them we have
kept_proteins_final <- kept_proteins_og
for (i in 1:length(diploids_results_final_significant)) {
  strain <- names(diploids_results_final_significant)[i]
  row_number <- nrow(kept_proteins_final)+1
  kept_proteins_final[row_number, 1] <- strain
  kept_proteins_final[row_number, 2] <- "Significantly diff. found between HPs"
  kept_proteins_final[row_number, 3] <- length(diploids_results_final_significant[[strain]][["protein"]])
}

temp <- kept_proteins_final %>% filter(Type != "Proteins detected in at least 1 HP")

# Plot
ggplot(data = temp, aes(x = reorder(Strain, Count), y = Count, fill = Type)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  theme_light() +
  theme(legend.position = "bottom",
        legend.text.position = "bottom") +
  xlab("Strains") +
  ylab("Number of proteins")


# Repeat the same but without the number of theoretical proteins based on the FASTAs
kept_proteins_final_no_FASTAs <- kept_proteins_final %>% filter(Type != "Total proteins based on FASTAs",
                                                                Type != "Proteins detected in at least 1 HP")

ggplot(data = kept_proteins_final_no_FASTAs, aes(x = reorder(Strain, Count), y = Count, fill = Type)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  theme_light() +
  theme(legend.position = "bottom",
        legend.text.position = "bottom") +
  xlab("Strains") +
  ylab("Number of proteins")

```

```
# Last plot I need for the discussion I thin
temp <- kept_proteins_final %>% filter(Type != "Total proteins based on FASTAs")

ggplot(data = temp, aes(x = reorder(Strain, Count), y = Count, fill = Type)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  theme_light() +
  theme(legend.position = "bottom",
        legend.text.position = "bottom") +
  xlab("Strains") +
  ylab("Number of proteins")
```


1.3. Gene ontology enrichment analysis
Using all S288C genes as background
```{r}
# Load data
entrez_db <- fread("C:/~/entrez_reference.txt")
go_df <- fread("C:/~/genes_to_be_GO_analyzed.tsv")

# Run GO analysis
my_universe <- as.character(entrez_db$`NCBI gene (formerly Entrezgene) ID`)
go_results <- enrichGO(gene = go_df$`NCBI gene (formerly Entrezgene) ID`, OrgDb = "org.Sc.sgd.db", keyType = "ENTREZID", ont = "BP", universe = my_universe)
go_results <- as.data.frame(go_results)
```

Using as reference unique() of all the proteins detected over all strain-specific runs separately
```{r}
# Load data
entrez_db <- fread("C:/~/entrez_reference.txt")
go_df <- fread("C:/~/genes_to_be_GO_analyzed.tsv")

# Process data
background_genes <- data.frame(total_proteins_observed_over_all_strains_ss_new)
colnames(background_genes) <- c("Genes")
background_genes <- left_join(background_genes, entrez_db, by = c("Genes" = "Protein stable ID"))

# Run GO analysis
my_universe <- as.character(background_genes$`NCBI gene (formerly Entrezgene) ID`)
go_results <- enrichGO(gene = go_df$`NCBI gene (formerly Entrezgene) ID`, OrgDb = "org.Sc.sgd.db", keyType = "ENTREZID", ont = "BP", universe = my_universe)
go_results <- as.data.frame(go_results)
```

B.7 Proteins with insertions and deletions

```
# 0. Load data and get it ready
## 0.1. Load all the dataframes as a list
```{r}
# Reports
files_path = '/~/matched_precursor_reports'
files = list.files(files_path, full.names = T)

## Grab the names of the dataframes (the strain names)
names <- c()
for (file in files) {
```
```

```
  start = str_locate(file, "matched_precursor_reports/Run_1_test_06062024_")[2] + 1
  end = str_locate(file, "_matched.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}

## Actually load the dataframes
setwd(files_path)
datas <- lapply(files, fread)
datas <- lapply(datas, as.data.frame)
names(datas) <- names

# Repeat this for this information already turned to protein level
files_path = "/~/protein_level_reports"
files = list.files(files_path, full.names = T)

## Grab the names of the dataframes (the strain names)
names <- c()
for (file in files) {
  start = str_locate(file, "protein_level_reports/")[2] + 1
  end = str_locate(file, "_protein_level.tsv")[1] - 1
  strain = substr(file, start, end)
  names <- c(names, strain)
}

## Actually load the dataframes
setwd(files_path)
datas_protein_level <- lapply(files, fread)
datas_protein_level <- lapply(datas_protein_level, as.data.frame)
names(datas_protein_level) <- names

## Set protein names as rownames
for (i in 1:length(datas_protein_level)) {
  rownames(datas_protein_level[[i]]) <- datas_protein_level[[i]]$Genes
  datas_protein_level[[i]] <- datas_protein_level[[i]] %>% select(-Genes)
}


# Sample correspondence
sample_correspondence <- fread("/~/sample_correspondence.tsv")
sample_correspondence <- as.data.frame(sample_correspondence)


# Stats files
files_path = '/~/stats_files/'
files = list.files(files_path, full.names = T)

setwd(files_path)
stats_files <- lapply(files, fread)
stats_files <- lapply(stats_files, as.data.frame)
names(stats_files) <- names

# Remove unnecessary variables
rm(list = c("end", "file", "files", "files_path", "start", "strain"))
```

0.2. Load information on which strains are haploid, diploid or polyploid

```{r}
load('/~/strains_in_each_type_vectors.Rdata')
```

0.3. Create separate lists for haploid, diploid and polyploid strains

```{r}
```

```
# Remember that QCs and BY4741-ki are not included in any of these!!
datas_haploid <- datas[names(datas) %in% haploid_strains]
datas_diploid <- datas[names(datas) %in% diploid_strains]
datas_polyploid <- datas[names(datas) %in% polyploid_strains]
```


1.1.1. Load data and get it ready
Report and file with information about indels
```{r}
# Load data
indels_per_strain <- read.csv("/~/indels_per_strain.csv")
ss_report_normalized <- fread("/~/protein_level_full_report.tsv")
ss_report_normalized <- as.data.frame(ss_report_normalized)
rownames(ss_report_normalized) <- ss_report_normalized$Genes
ss_report_normalized <- ss_report_normalized %>% select(-Genes)

source("/~/0. prepare_data_functions.R")

# Fix protein names
new_rownames <- c()
for (i in 1:nrow(ss_report_normalized)) {

  ## For multiple protein names, grab only the first one
  rowname <- rownames(ss_report_normalized)[i]
  if (grepl("/", rowname)) {
    end <- str_locate(rowname, "/") - 1
    new_rowname <- substr(rowname, 0, end)
    new_rownames <- c(new_rownames, new_rowname)
  }
  else {
    new_rownames <- c(new_rownames, rowname)
  }
}
```

Process the file with the information about indels: create 2 separate files, one for insertions and one for deletions, and in each of them have one protein per row, and then the strains in which there is an insertion/deletion in that protein, this will make it much easier afterwards - actually 4 files, we do this with both systematic and standard protein names
```{r}
# Remove S288C because it does not have any proteins with deletions
indels_per_strain <- indels_per_strain %>% filter(Strain != "S288C")

insertions <- list()
deletions <- list()

for (i in 1:nrow(indels_per_strain)) {
  proteins_with_insertions <- unique(str_split_1(indels_per_strain$Proteins_with_insertion[i], ", "))
  proteins_with_deletions <- unique(str_split_1(indels_per_strain$Proteins_with_deletion[i], ", "))

  # Proteins with insertions
  for (j in 1:length(proteins_with_insertions)) {
    protein <- proteins_with_insertions[j]
    if (!(protein %in% names(insertions))) {
      insertions[[protein]] <- c(indels_per_strain$Strain[i])
    }
    else {
      insertions[[protein]] <- c(insertions[[protein]], indels_per_strain$Strain[i])
    }
  }
```

```
  # Proteins with deletions
  for (j in 1:length(proteins_with_deletions)) {
    protein <- proteins_with_deletions[j]
    if (!(protein %in% names(deletions))) {
      deletions[[protein]] <- c(indels_per_strain$Strain[i])
    }
    else {
      deletions[[protein]] <- c(deletions[[protein]], indels_per_strain$Strain[i])
    }
  }
}
```

1.1.2. Tests

Test for each protein with insertions or deletions (separately) if the abundance of this protein is significantly different between the strains which have the mutation and those which do not

Insertions

```{r}
# Insertions
## Get a smaller version of the dataset which only contains the proteins with insertions - there are only 41 of the 279 that are actually detected :(
temp_insertions <- ss_report_normalized[rownames(ss_report_normalized) %in% names(insertions),]

## Create dataframe for p-values
proteins_tested <- c()
p.values.bin <- c()
p.values.cont <- c()
non_na_values_mutated <- c()
non_na_values_non_mutated <- c()
total_values_mutated <- c()
total_values_non_mutated <- c()

## Now actually go through the proteins and test for those which are present
for (i in 1:length(insertions)) {
  protein <- names(insertions)[i]

  # If this protein is found in the report
  if (protein %in% rownames(temp_insertions)) {
    # Come up with a vector of booleans which indicates in which columns are the samples of the strains that contain insertions in this protein
    # We use this to obtain both vectors we will be using for testing
    columns_condition <- rep(FALSE, ncol(temp_insertions))
    for (strain in insertions[[protein]]) {
      columns_condition <- columns_condition | grepl(strain, colnames(temp_insertions))
    }
    mutated <- temp_insertions[rownames(temp_insertions) == protein, columns_condition]
    non_mutated <- temp_insertions[rownames(temp_insertions) == protein, !columns_condition]

    # Perform the testing
    if (length(mutated) > 1 & length(non_mutated > 1)) {
      # Turn the data into presence/absence and do a proportion test instead
      mutated_bin <- as.numeric(!(is.na(mutated)))
      non_mutated_bin <- as.numeric(!(is.na(non_mutated)))
      my_mat <- matrix(c(sum(mutated_bin == 1), sum(mutated_bin == 0),
                         sum(non_mutated_bin == 1), sum(non_mutated_bin == 0)),
                         ncol = 2, byrow = T)
      colnames(my_mat) <- c("Present", "Absent")
      rownames(my_mat) <- c("Mutated", "Non-mutated")

      # Add the tested protein and its p-value to the output vectors
```

```
      proteins_tested <- c(proteins_tested, protein)
      p.values.bin <- c(p.values.bin, prop.test(my_mat)$p.value)

      # Check how many values different from NA we have in each of the vectors, and save that
      non_na_values_mutated <- c(non_na_values_mutated, sum(!(is.na(mutated))))
      non_na_values_non_mutated <- c(non_na_values_non_mutated, sum(!(is.na(non_mutated))))

      # Check how many values in total we have in each of the vectors and also save it
      total_values_mutated <- c(total_values_mutated, length(mutated))
      total_values_non_mutated <- c(total_values_non_mutated, length(non_mutated))

      # Perform a test keeping the data as continuous and save that p-value as well
      mutated_cont <- na.omit(as.numeric(mutated))
      non_mutated_cont <- na.omit(as.numeric(non_mutated))
      if (length(mutated_cont) > 1 & length(non_mutated_cont) > 1) {
        p.values.cont <- c(p.values.cont, t.test(mutated_cont, non_mutated_cont)$p.value)
      }
      else {
        p.values.cont <- c(p.values.cont, NA)
      }
    }
  }
}

results_insertions_final <- data.frame(proteins_tested, p.values.bin, p.values.cont,
↪  non_na_values_mutated, non_na_values_non_mutated, total_values_mutated,
↪  total_values_non_mutated)
colnames(results_insertions_final) <- c("Protein", "p.val.bin", "p.val.cont",
↪  "Non_NA_values_mutated", "Non_NA_values_non_mutated", "total_values_mutated",
↪  "total_values_non_mutated")
results_insertions_final$p.adj.bin <- p.adjust(results_insertions_final$p.val.bin, method =
↪  "BH")
results_insertions_final$p.adj.cont <- p.adjust(results_insertions_final$p.val.cont, method =
↪  "BH")
```

Deletions

```{r}
## Get a smaller version of the dataset which only contains the proteins with deletions - there
↪  are only 41 of the 279 that are actually detected :(
temp_deletions <- ss_report_normalized[rownames(ss_report_normalized) %in% names(deletions),]

## Create dataframe for p-values
proteins_tested <- c()
p.values.bin <- c()
p.values.cont <- c()
non_na_values_mutated <- c()
non_na_values_non_mutated <- c()
total_values_mutated <- c()
total_values_non_mutated <- c()

## Now actually go through the proteins and test for those which are present
for (i in 1:length(deletions)) {
  protein <- names(deletions)[i]

  # If this protein is found in the report
  if (protein %in% rownames(temp_deletions)) {
    # Come up with a vector of booleans which indicates in which columns are the samples of the
    ↪  strains that contain deletions in this protein
    # We use this to obtain both vectors we will be using for testing
    columns_condition <- rep(FALSE, ncol(temp_deletions))
    for (strain in deletions[[protein]]) {
      columns_condition <- columns_condition | grepl(strain, colnames(temp_deletions))
```

```
    }
    mutated <- temp_deletions[rownames(temp_deletions) == protein, columns_condition]
    non_mutated <- temp_deletions[rownames(temp_deletions) == protein, !columns_condition]

    # Perform the testing
    if (length(mutated) > 1 & length(non_mutated > 1)) {
      # Turn the data into presence/absence and do a proportion test instead
      mutated_bin <- as.numeric(!(is.na(mutated)))
      non_mutated_bin <- as.numeric(!(is.na(non_mutated)))
      my_mat <- matrix(c(sum(mutated_bin == 1), sum(mutated_bin == 0),
                         sum(non_mutated_bin == 1), sum(non_mutated_bin == 0)),
                         ncol = 2, byrow = T)
      colnames(my_mat) <- c("Present", "Absent")
      rownames(my_mat) <- c("Mutated", "Non-mutated")

      # Add the tested protein and its p-value to the output vectors
      proteins_tested <- c(proteins_tested, protein)
      p.values.bin <- c(p.values.bin, prop.test(my_mat)$p.value)

      # Check how many values different from NA we have in each of the vectors, and save that
      non_na_values_mutated <- c(non_na_values_mutated, sum(!(is.na(mutated))))
      non_na_values_non_mutated <- c(non_na_values_non_mutated, sum(!(is.na(non_mutated))))

      # Check how many values in total we have in each of the vectors and also save it
      total_values_mutated <- c(total_values_mutated, length(mutated))
      total_values_non_mutated <- c(total_values_non_mutated, length(non_mutated))

      # Perform a test keeping the data as continuous and save that p-value as well
      mutated_cont <- na.omit(as.numeric(mutated))
      non_mutated_cont <- na.omit(as.numeric(non_mutated))
      if (length(mutated_cont) > 1 & length(non_mutated_cont) > 1) {
        p.values.cont <- c(p.values.cont, t.test(mutated_cont, non_mutated_cont)$p.value)
      }
      else {
        p.values.cont <- c(p.values.cont, NA)
      }
    }
  }
}

results_deletions_final <- data.frame(proteins_tested, p.values.bin, p.values.cont,
↪ non_na_values_mutated, non_na_values_non_mutated, total_values_mutated,
↪ total_values_non_mutated)
colnames(results_deletions_final) <- c("Protein", "p.val.bin", "p.val.cont",
↪ "Non_NA_values_mutated", "Non_NA_values_non_mutated", "total_values_mutated",
↪ "total_values_non_mutated")
results_deletions_final$p.adj.bin <- p.adjust(results_deletions_final$p.val.bin, method = "BH")
results_deletions_final$p.adj.cont <- p.adjust(results_deletions_final$p.val.cont, method =
↪ "BH")
```


1.1.3. Come up with some barplots which show how the number of proteins of each type
↪ decreases along the steps we take here

```{r}
# Create dataset
protein_numbers <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(protein_numbers) <- c("Mutation", "Step", "Protein_number")

## Total proteins with each type of mutation
protein_numbers[nrow(protein_numbers)+1,] <- c("Insertion", "1. Theoretical - from Gilles SV
↪ files", as.character(length(insertions)))
```

```
protein_numbers[nrow(protein_numbers)+1,] <- c("Deletion", "1. Theoretical - from Gilles SV files", as.character(length(deletions)))

## Proteins that show up in the report (that is already pre-processed)
protein_numbers[nrow(protein_numbers)+1,] <- c("Insertion", "2. Present in the report", as.character(sum(names(insertions) %in% rownames(ss_report_normalized))))
protein_numbers[nrow(protein_numbers)+1,] <- c("Deletion", "2. Present in the report", as.character(sum(names(deletions) %in% rownames(ss_report_normalized))))

## Proteins that could be tested
protein_numbers[nrow(protein_numbers)+1,] <- c("Insertion", "3. Could be tested - at least 2 samples in each group", as.character(nrow(results_insertions_final)))
protein_numbers[nrow(protein_numbers)+1,] <- c("Deletion", "3. Could be tested - at least 2 samples in each group", as.character(nrow(results_deletions_final)))

## Proteins for which we had more than 4 observations in both vectors compared
temp <- results_insertions_final %>% filter(total_values_mutated > 4 & total_values_non_mutated > 4)
protein_numbers[nrow(protein_numbers)+1,] <- c("Insertion", "4. More than 4 replicates per group", as.character(nrow(temp)))
temp <- results_deletions_final %>% filter(total_values_mutated > 4 & total_values_non_mutated > 4)
protein_numbers[nrow(protein_numbers)+1,] <- c("Deletion", "4. More than 4 replicates per group", as.character(nrow(temp)))

## Proteins that are found to be significantly differentially present/absent between mutated and non-mutated strains
protein_numbers[nrow(protein_numbers)+1,] <- c("Insertion", "5. Significant", as.character(sum(results_insertions_final$p.adj.bin < 0.05)))
protein_numbers[nrow(protein_numbers)+1,] <- c("Deletion", "5. Significant", as.character(sum(results_deletions_final$p.adj.bin < 0.05, na.rm = T)))

## Turn last column to numeric - if you try to add rows with different data types you get an error
protein_numbers$Protein_number <- as.numeric(protein_numbers$Protein_number)


# Plot
ggplot(data = protein_numbers, aes(x = Mutation, y = Protein_number, fill = Step)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  theme_light() +
  ylab("Number of proteins across all strains") +
  theme(legend.position = "bottom",
        legend.title = element_blank()) +
  guides(fill=guide_legend(nrow=2,byrow=TRUE))
```

Appendix C

Appendix for Python code

C.1 Create dictionaries from original FASTA files

```
# Diploid strains
## Define directory which contains the files
directory = "C:\~\Diploids"

# 1. Iterate over files in directory, create fragmentation_dict and a few others
# Define dictionaries we want to end up with
full_id_dict = {}
fragmentation_dict = {}
repeated_across_strains = {}


# Iterate over files in the directory
for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    strain = filename[0:filename.find(".")]
    HP = filename[filename.find(".nuclear")-3:filename.find(".nuclear")]
    tag = strain + "_" + HP


    # Open file
    with open(path) as handle:
        peptides_per_protein_dict = {}
        all_prot_ids = []
        full_ids = {}
        repeated = {}
        full_protein_seqs_strain = {}

        # In each file, iterate over the proteins
        for seq_id, seq in SimpleFastaParser(handle):
            # Get what is going to be the protein ID. Also append it to the "repeated" list if
            ↪  we´ve seen that ID before in this file
            limit = seq_id.rfind("|")
            last_chunk = seq_id[limit + 1:len(seq_id)]
            first_chunk = seq_id[0:seq_id.find("|")]

            if last_chunk == first_chunk:
                prot_id = last_chunk[last_chunk.rfind("_") + 1:len(last_chunk)] + "_" + tag

            else:
                prot_id = last_chunk
                if prot_id in all_prot_ids:
                    if prot_id not in list(repeated.keys()):
                        repeated[prot_id] = [full_protein_seqs_strain[prot_id], seq]
                    else:
                        repeated[prot_id].append(seq)
                else:
                    full_protein_seqs_strain[prot_id] = seq
```

```python

            # Perform fragmentation
            peptides_pre = re.sub(r'(?<=[RK])(?=[^P])', '\n', seq)
            peptides_pre = list(peptides_pre.split("\n"))
            peptides = []
            for peptide in peptides_pre:
                if 7 <= len(peptide) <= 30:
                    peptides.append(peptide)

            if prot_id not in list(repeated.keys()):
                # Add to dictionary
                peptides_per_protein_dict[prot_id] = peptides

                # All prot_ids
                all_prot_ids.append(prot_id)

                # Full IDs
                full_ids[prot_id] = seq_id

        # Add to final dictionaries
        full_id_dict[tag] = full_ids
        fragmentation_dict[tag] = peptides_per_protein_dict
        repeated_across_strains[tag] = repeated


# 2. Save dictionaries
## 2.1. Save the dictionary of dictionaries for the repeated proteins in each haplotype to a JSON file for later reference
json_file = os.path.join('C:\~\Dictionaries\\', 'repeated_proteins_diploids.json')
with open(json_file, 'w') as fp:
    json.dump(repeated_across_strains, fp)

## 2.2. Save the fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Diploids_original.json')
with open(json_file, 'w') as fp:
    json.dump(fragmentation_dict, fp)

## 2.3. Save the full ID dictionary
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_diploids.json')
with open(json_file, 'w') as fp:
    json.dump(full_id_dict, fp)



### Haploid strains
# 1. Create fragmentation dictionary
## Define directory which contains the files
directory = "C:\~\Haploids"

## Define dictionary we want to end up with
fragmentation_dict_haploids = {}
repeated_across_strains_haploids = {}
full_id_dict = {}
empty_peptides = {}

## Iterate over files in the directory
for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    strain = filename[0:filename.find(".")]

    # Open file
    with open(path) as handle:
        peptides_per_protein_dict = {}
```

```
        all_prot_ids = []
        repeated = {}
        full_ids = {}
        empty_peptides_strain = {}
        full_protein_seqs_strain = {}

        # In each file, iterate over the proteins
        for seq_id, seq in SimpleFastaParser(handle):
            # Get what is going to be the protein ID. Also append it to the "repeated" list if
            ↪  we´ve seen that ID before in this file
            limit = seq_id.rfind("|")
            last_chunk = seq_id[limit + 1:len(seq_id)]
            first_chunk = seq_id[0:(len(seq_id) - limit - 1)]

            if last_chunk == first_chunk:
                prot_id = last_chunk[last_chunk.rfind("_") + 1:len(last_chunk)] + "_" + strain

            else:
                prot_id = last_chunk
                if prot_id in all_prot_ids:
                    if prot_id not in list(repeated.keys()):
                        repeated[prot_id] = [full_protein_seqs_strain[prot_id], seq]
                    else:
                        repeated[prot_id].append(seq)
                else:
                    full_protein_seqs_strain[prot_id] = seq

            # Perform fragmentation
            peptides_pre = re.sub(r'(?<=[RK])(?=[^P])', '\n', seq)
            peptides_pre = list(peptides_pre.split("\n"))
            peptides = []
            empty_peptides_protein_list = []
            for peptide in peptides_pre:
                if 7 <= len(peptide) <= 30:
                    peptides.append(peptide)
                else:
                    empty_peptides_protein_list.append(len(peptide))
            if peptides == []:
                empty_peptides_strain[prot_id] = empty_peptides_protein_list

            if prot_id not in list(repeated.keys()):
                # Add to dictionary
                peptides_per_protein_dict[prot_id] = peptides

                # All prot_ids
                all_prot_ids.append(prot_id)

                # Full IDs
                full_ids[prot_id] = seq_id

        # Add to final dictionary
        fragmentation_dict_haploids[strain] = peptides_per_protein_dict
        repeated_across_strains_haploids[strain] = repeated
        full_id_dict[strain] = full_ids
        empty_peptides[strain] = empty_peptides_strain

# 2. Save created dictionaries
## 2.1. Save the fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Haploids_original.json')
with open(json_file, 'w') as fp:
    json.dump(fragmentation_dict_haploids, fp)
```

```
## 2.2. Save the dictionary of dictionaries for the repeated proteins in each haplotype to a JSON file for later reference
json_file = os.path.join('C:\~\Dictionaries\\', 'repeated_proteins_haploids.json')
with open(json_file, 'w') as fp:
    json.dump(repeated_across_strains_haploids, fp)

## 2.3. Save the dictionary of full IDs, I use this to create the new FASTA files
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_haploids.json')
with open(json_file, 'w') as fp:
    json.dump(full_id_dict, fp)



### Polyploid strains
# 1. Create fragmentation dictionary
## Define directory which contains the files
directory = "C:\~\Polyploids_HP"

## Define dictionary we want to end up with
fragmentation_dict_polyploids = {}
repeated_across_strains_polyploids = {}
full_id_dict = {}
empty_peptides = {}

## Iterate over files in the directory
for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    strain = filename[0:filename.find(".")]

    # Open file
    with open(path) as handle:
        peptides_per_protein_dict = {}
        all_prot_ids = []
        repeated = {}
        full_ids = {}
        empty_peptides_strain = {}
        full_protein_seqs_strain = {}

        # In each file, iterate over the proteins
        for seq_id, seq in SimpleFastaParser(handle):
            # Get what is going to be the protein ID. Also append it to the "repeated" list if we´ve seen that ID before in this file
            limit = seq_id.rfind("|")
            last_chunk = seq_id[limit + 1:len(seq_id)]
            first_chunk = seq_id[0:(len(seq_id) - limit - 1)]

            if last_chunk == first_chunk:
                prot_id = last_chunk[last_chunk.rfind("_") + 1:len(last_chunk)] + "_" + strain

            else:
                prot_id = last_chunk

            # This is new here: we put this outside the above "else" because in this case proteins tagged as "G00000010" can also be repeated
            if prot_id in all_prot_ids:
                if prot_id not in list(repeated.keys()):
                    repeated[prot_id] = [full_protein_seqs_strain[prot_id], seq]

                else:
                    repeated[prot_id].append(seq)

            else:
                full_protein_seqs_strain[prot_id] = seq
```

```
230
231             # Perform fragmentation
232             peptides_pre = re.sub(r'(?<=[RK])(?=[^P])', '\n', seq)
233             peptides_pre = list(peptides_pre.split("\n"))
234             peptides = []
235             empty_peptides_protein_list = []
236             for peptide in peptides_pre:
237                 if 7 <= len(peptide) <= 30:
238                     peptides.append(peptide)
239                 else:
240                     empty_peptides_protein_list.append(len(peptide))
241             if peptides == []:
242                 empty_peptides_strain[prot_id] = empty_peptides_protein_list
243
244             if prot_id not in list(repeated.keys()):
245                 # Add to dictionary
246                 peptides_per_protein_dict[prot_id] = peptides
247
248                 # All prot_ids
249                 all_prot_ids.append(prot_id)
250
251                 # Full IDs
252                 full_ids[prot_id] = seq_id
253
254             else:
255                 # Add to dictionary any new peptides we´ve found for this protein
256                 for peptide in peptides:
257                     if peptide not in peptides_per_protein_dict[prot_id]:
258                         peptides_per_protein_dict[prot_id].append(peptide)
259
260         # Add to final dictionary
261         fragmentation_dict_polyploids[strain] = peptides_per_protein_dict
262         repeated_across_strains_polyploids[strain] = repeated
263         full_id_dict[strain] = full_ids
264         empty_peptides[strain] = empty_peptides_strain
265
266 # 2. Save created dictionaries
267 ## 2.1. Save the fragmentation dictionary
268 json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Polyploids_original.json')
269 with open(json_file, 'w') as fp:
270     json.dump(fragmentation_dict_polyploids, fp)
271
272 ## 2.2. Save the dictionary of dictionaries for the repeated proteins in each haplotype to a
    ↪  JSON file for later reference
273 json_file = os.path.join('C:\~\Dictionaries\\', 'repeated_proteins_polyploids.json')
274 with open(json_file, 'w') as fp:
275     json.dump(repeated_across_strains_polyploids, fp)
276
277 ## 2.3. Save the dictionary of full IDs, I use this to create the new FASTA files
278 json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_polyploids.json')
279 with open(json_file, 'w') as fp:
280     json.dump(full_id_dict, fp)
281
282
283
284 ### Add information from mitochondrial assemblies
285 # 1. Define directory where our files are
286 dir = "C:\~\mitochondrial"
287
288
289 # 2. Go through the files creating a fragmentation dictionary for each
290 full_id_dict_mito = {}
291 fragmentation_dict_mito = {}
```

```
repeated_across_strains_mito = {}

# Iterate over files in the directory
for filename in os.listdir(dir):
    path = os.path.join(dir, filename)
    strain = filename[0:filename.find(".")]
    tag = strain

    # Open file
    with open(path) as handle:
        peptides_per_protein_dict = {}
        all_prot_ids = []
        full_ids = {}
        repeated = {}
        full_protein_seqs_strain = {}

        # In each file, iterate over the proteins
        for seq_id, seq in SimpleFastaParser(handle):
            # Get what is going to be the protein ID. Also append it to the "repeated" list if we´ve seen that ID before in this file
            limit = seq_id.rfind("|")
            last_chunk = seq_id[limit + 1:len(seq_id)]
            first_chunk = seq_id[0:seq_id.find("|")]

            if last_chunk == first_chunk:
                prot_id = last_chunk[last_chunk.rfind("_") + 1:len(last_chunk)] + "_" + tag

            else:
                prot_id = last_chunk

            # This is new here: we put this outside the above "else" because in this case proteins tagged as "G00000010" can also be repeated - as in polyploids
            if prot_id in all_prot_ids:
                if prot_id not in list(repeated.keys()):
                    repeated[prot_id] = [full_protein_seqs_strain[prot_id], seq]
                else:
                    repeated[prot_id].append(seq)
            else:
                full_protein_seqs_strain[prot_id] = seq

            # Perform fragmentation
            peptides_pre = re.sub(r'(?<=[RK])(?=[^P])', '\n', seq)
            peptides_pre = list(peptides_pre.split("\n"))
            peptides = []
            for peptide in peptides_pre:
                if 7 <= len(peptide) <= 30:
                    peptides.append(peptide)

            if prot_id not in list(repeated.keys()):
                # Add to dictionary
                peptides_per_protein_dict[prot_id] = peptides

                # All prot_ids
                all_prot_ids.append(prot_id)

                # Full IDs
                full_ids[prot_id] = seq_id

            else:
                # Add to dictionary any new peptides we´ve found for this protein - from polyploids, allows us to have all fragments from all versions of a protein in the entry for that protein
```

```
                # (in this case it only applies to one of the polyploids, CDN_1a, and this doesn´t even affect it, but okay)
                for peptide in peptides:
                    if peptide not in peptides_per_protein_dict[prot_id]:
                        peptides_per_protein_dict[prot_id].append(peptide)

        # Add to final dictionaries
        full_id_dict_mito[tag] = full_ids
        fragmentation_dict_mito[tag] = peptides_per_protein_dict
        repeated_across_strains_mito[tag] = repeated

# 2. Save created dictionaries
## 2.1. Save the fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Mitochondrial.json')
with open(json_file, 'w') as fp:
    json.dump(fragmentation_dict_mito, fp)

## 2.2. Save the dictionary of dictionaries for the repeated proteins in each haplotype to a JSON file for later reference
json_file = os.path.join('C:\~\Dictionaries\\', 'repeated_proteins_mitochondrial.json')
with open(json_file, 'w') as fp:
    json.dump(repeated_across_strains_mito, fp)

## 2.3. Save the dictionary of full IDs, I use this to create the new FASTA files
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_mitochondrial.json')
with open(json_file, 'w') as fp:
    json.dump(full_id_dict_mito, fp)
```

C.2 Create new FASTA files

```
### Haploid strains
# 1. Load necessary dictionaries
## 1.1. Haploid fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Haploids_original.json')
with open(json_file) as f_in:
    fragmentation_dict = json.load(f_in)

## 1.2 Full ID dictionary
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_haploids.json')
with open(json_file) as f_in:
    full_id_dict = json.load(f_in)

# 2. Write new FASTAs
## 2.1. Define directory which contains the files
directory = "C:\~\Data\\DIA-NN"
new_dir = os.path.join(directory, "New haploid files")
os.makedirs(new_dir)

## 2.2. First of all iterate over strains
strains = list(fragmentation_dict.keys())
for strain in strains:
    # Create FASTA file and start writing into it
    new_filename = strain + '_HPO_nuclear' + ".fasta"
    file_out = os.path.join(new_dir, new_filename)
    with open(file_out, "w") as f_out:
        for protein in list(fragmentation_dict[strain].keys()):
            full_id = full_id_dict[strain][protein]
            for peptide_seq in fragmentation_dict[strain][protein]:
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

```

```python


### Diploid strains
# 1. Load necessary dictionaries
## 1.1. Diploid fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Diploids_original.json')
with open(json_file) as f_in:
    fragmentation_dict = json.load(f_in)
del (f_in, json_file)

## 1.2. Full ID dictionary
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_diploids.json')
with open(json_file) as f_in:
    full_id_dict = json.load(f_in)


# 2. Get a list of the haplotypes and create a dictionary that maps each strain to its 2 haplotypes
haplotypes = list(fragmentation_dict.keys())

strain_to_HP_dict = {}
for haplotype in haplotypes:
    strain = haplotype[0:3]
    strain_to_HP_dict[strain] = [haplotype for haplotype in haplotypes if haplotype[0:3] == strain]


# 3.
## 3.1. Get a list of the strains and iterate over them. For each, we get the two haplotypes and get the intersection of their proteins,
## those which are present in both of them. Then we iterate over these proteins and compare their fragments, to see if they are
## exactly the same protein or not.

## 3.2 I´ve decided to use this loop to also create a dict with an entrance for each strain, which is also a dict,
## with an entrance for each protein, which is also a dict, where then I have the following entrances:
## common peptides between HPs, peptides only in HP1, peptides only in HP2
## This I should probably be able to use also to construct the final FASTA files
strains = list(strain_to_HP_dict.keys())

strain_summary_dict = {}
dict_common_prots_per_strain = {}

for strain in strains:
    strain_dict_goal_1 = {}
    prot_dict_goal_2 = {}
    haplotype_1, haplotype_2 = strain_to_HP_dict[strain]
    proteins_hp_1 = list(fragmentation_dict[haplotype_1].keys())
    proteins_hp_2 = list(fragmentation_dict[haplotype_2].keys())
    strain_dict_goal_1["unique_HP1"] = list(set(proteins_hp_1) - set(proteins_hp_2))
    strain_dict_goal_1["unique_HP2"] = list(set(proteins_hp_2) - set(proteins_hp_1))

    common_proteins = list(set(proteins_hp_1).intersection(proteins_hp_2))
    common_equal = []
    common_diff = []

    for prot in common_proteins:
        peptides_hp_1 = fragmentation_dict[haplotype_1][prot]
        peptides_hp_2 = fragmentation_dict[haplotype_2][prot]
        if peptides_hp_1 == peptides_hp_2:
            common_equal.append(prot)
```

```
        else:
            common_diff.append(prot)
            peptides_dict_goal_2 = {"common": list(set(peptides_hp_1).intersection(set(peptides_hp_2))),
                                    "unique_hp_1": list(set(peptides_hp_1) - set(peptides_hp_2)),
                                    "unique_hp_2": list(set(peptides_hp_2) - set(peptides_hp_1))}

            prot_dict_goal_2[prot] = peptides_dict_goal_2

    # Add lists to the strain dictionary
    strain_dict_goal_1["common_equal"] = common_equal
    strain_dict_goal_1["common_diff"] = common_diff

    # Add strain dict to the general dict with all strains
    strain_summary_dict[strain] = strain_dict_goal_1
    dict_common_prots_per_strain[strain] = prot_dict_goal_2


# 4. Write new FASTAs
## 4.1. Define directory which contains the files
directory = "C:\~\Data\\DIA-NN"
new_dir = os.path.join(directory, "New diploid files")
os.makedirs(new_dir)

## 4.2. First of all iterate over strains
strains = list(strain_to_HP_dict.keys())
for strain in strains:
    # Create FASTA file and start writing into it
    new_filename = strain + "_HP1_HP2_nuclear" + ".fasta"
    file_out = os.path.join(new_dir, new_filename)
    with open(file_out, "w") as f_out:
        # For proteins unique to HP1
        for protein in list(strain_summary_dict[strain]["unique_HP1"]):
            full_id = full_id_dict[strain+"_HP1"][protein] + "_unique_HP1"
            for peptide_seq in fragmentation_dict[strain+"_HP1"][protein]:
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

        # For proteins unique to HP2
        for protein in list(strain_summary_dict[strain]["unique_HP2"]):
            full_id = full_id_dict[strain + "_HP2"][protein] + "_unique_HP2"
            for peptide_seq in fragmentation_dict[strain + "_HP2"][protein]:
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

        # For proteins common to both HPs and with the same sequence
        for protein in list(strain_summary_dict[strain]["common_equal"]):
            full_id = full_id_dict[strain + "_HP1"][protein]                    # Just the original full ID, I could grab it from either HP1 or HP2 dictionary since they are the same
            for peptide_seq in fragmentation_dict[strain + "_HP1"][protein]:
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

        # For proteins common to both HPs but with different sequences
        for protein in list(strain_summary_dict[strain]["common_diff"]):
            id = full_id_dict[strain + "_HP1"][protein]                         # Same
            # Common peptides
            for peptide_seq in list(set(fragmentation_dict[strain + "_HP1"][protein]).intersection(set(fragmentation_dict[strain + "_HP2"][protein]))):
```

```
                full_id = id + "_common"
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

            # Peptides only in HP1
            for peptide_seq in list(set(fragmentation_dict[strain + "_HP1"][protein]) -
            ↪   set(fragmentation_dict[strain + "_HP2"][protein])):
                full_id = id + "_common_HP1"
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)

            # Peptides only in HP2
            for peptide_seq in list(set(fragmentation_dict[strain + "_HP2"][protein]) -
            ↪   set(fragmentation_dict[strain + "_HP1"][protein])):
                full_id = id + "_common_HP2"
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)



### Polyploid strains
# 1. Load necessary dictionaries
## 1.1. Polyploid fragmentation dictionary
json_file = os.path.join('C:\~\Fragmentation dictionaries\\', 'Polyploids_original.json')
with open(json_file) as f_in:
    fragmentation_dict = json.load(f_in)

## 1.2 Full ID dictionary
json_file = os.path.join('C:\~\Dictionaries\\', 'full_IDs_polyploids.json')
with open(json_file) as f_in:
    full_id_dict = json.load(f_in)


# 2. Write new FASTAs
## 2.1. Define directory which contains the files
directory = "C:\~\Data\\DIA-NN"
new_dir = os.path.join(directory, "New polyploid files")
os.makedirs(new_dir)

## 2.2. First of all iterate over strains
strains = list(fragmentation_dict.keys())
for strain in strains:
    # Create FASTA file and start writing into it
    new_filename = strain + '_HP_nuclear' + ".fasta"
    file_out = os.path.join(new_dir, new_filename)
    with open(file_out, "w") as f_out:
        for protein in list(fragmentation_dict[strain].keys()):
            full_id = full_id_dict[strain][protein]
            for peptide_seq in fragmentation_dict[strain][protein]:
                entry = ">" + full_id + "\n" + peptide_seq + "\n"
                f_out.write(entry)
```

C.3 Create stacked barplots - diploid strains as example

```
## 1. Start from here, load the fragmentation dictionary from a JSON file
json_file = os.path.join("C:\~\Fragmentation dictionaries\\", 'Diploids_original.json')
with open(json_file) as f_in:
    fragmentation_dict_diploids = json.load(f_in)
del(f_in, json_file)


```

```
# 2. Get the data from S288C, load it from the corresponding dictionary
json_file = os.path.join('C:\~\Dictionaries\\', 'S288C_fragmentation_dict.json')
with open(json_file) as fp:
    S288C_dict = json.load(fp)
S288C_prots = list(S288C_dict.keys())


# 2.1. Create 3 dictionaries, all of them with strains as keys, and as values more dictionaries with:
#   - Proteins that are in that strain and not in S288C (keys), and lists with the fragments from each (values)
#   - Proteins common to that strain and S288C, then the fragments that are unique to this strain w.r.t. S288C
#   - Proteins common to that strain and S288C, then the fragments that are common to both
strains = list(fragmentation_dict_diploids.keys())

proteins_unidentified = {}
proteins_non_common_dict_diploids = {}
proteins_common_dict_diploids = {}
proteins_common_equal_dict_diploids = {}

for strain in strains:
    strain_unidentified = {}
    strain_common_to_288 = {}
    strain_common_to_288_equal = {}
    strain_diff_from_288 = {}
    strain_prots = list(fragmentation_dict_diploids[strain].keys())

    for prot in strain_prots:
        if "G0" in prot:
            strain_unidentified[prot] = fragmentation_dict_diploids[strain][prot]
        elif prot in S288C_prots:
            peptides_this_strain = fragmentation_dict_diploids[strain][prot]
            peptides_288 = S288C_dict[prot]
            temp_list = list(set(peptides_this_strain) - set(peptides_288))
            if peptides_this_strain != peptides_288:
                strain_common_to_288[prot] = []
            else:
                strain_common_to_288_equal[prot] = fragmentation_dict_diploids[strain][prot]
        else:
            strain_diff_from_288[prot] = fragmentation_dict_diploids[strain][prot]

    proteins_unidentified[strain] = strain_unidentified
    proteins_non_common_dict_diploids[strain] = strain_diff_from_288
    proteins_common_dict_diploids[strain] = strain_common_to_288
    proteins_common_equal_dict_diploids[strain] = strain_common_to_288_equal
del(peptides_288, peptides_this_strain, prot, strain, strain_prots, temp_list)


# Create a stacked barplot summarizing all of this
## Create a Pandas dataframe from which to plot
df_data = [list(proteins_common_dict_diploids.keys()), [len(x) for x in proteins_common_equal_dict_diploids.values()], [len(x) for x in proteins_common_dict_diploids.values()], [len(x) for x in proteins_non_common_dict_diploids.values()], [len(x) for x in proteins_unidentified.values()]]
df_for_stacked_barplot_diploids = pd.DataFrame(df_data, index=['Strains', 'Common proteins between this strain and S288C - same sequence', 'Common proteins between this strain and S288C - different sequence', "Proteins in this strain not present in S288C", 'Unidentified proteins in this strain']).T

## Re-order so the barplot looks better
```

```
df_for_stacked_barplot_diploids = df_for_stacked_barplot_diploids.sort_values(by = ["Common proteins between this strain and S288C - same sequence"])

## Come up with the tags for the columns
x_tags = []
for index, row in df_for_stacked_barplot_diploids.iterrows():
    x_tags.append(row['Strains'])

## Plot
ax = df_for_stacked_barplot_diploids.plot(kind = 'bar', stacked=True, title='Comparison of proteins present in each strain with respect to S288C')
ax.set_xticklabels(x_tags, fontsize=8)

```