h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt

Fachbereich Informatik

A Pipeline for Differential Gene Expression Analysis

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

vorgelegt von

Markus Leipold

Referent:	Prof. Dr. Bernhard Humm
Korreferentin:	Prof. Dr. Bettina Harriehausen-Mühlbauer

Ausgabedatum:	12.11.2015
Abgabedatum:	12.05.2016

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.
Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.
Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 11.05.2016

ABSTRACT

Die differenzielle Genanalyse ist eine Komponente der Genforschung. Die Ergebnisse der Analyse können unter anderem helfen die Ursachen von phänotypischen Erscheinungen zu verstehen.

Es existieren bereits viele quelloffene Pakete für die Programmiersprache R, welche zum Auffinden von differenziell exprimierten Genen genutzt werden können. Der einzige Nachteil ist hierbei, dass wahrscheinlich nicht jeder Wissenschaftler über Programmierkenntnisse verfügt, welche aber für die Benutzung der Pakete vorausgesetzt werden.

Das Resultat von dieser Masterarbeit soll den Wissenschaftlern eine Lösung in Form einer Pipeline für differenzielle Genanalysen darbieten, welche einfach und ohne Programmierkenntnisse zu nutzen ist. Des weiteren sollen mehrere Pakete mit unterschiedlichen zugrundeliegenden statistischen Verfahren integriert werden. Über die Pakete hinweg genutzte Parameter sollen hierbei vereinheitlicht werden. Diese Maßnahme soll das Lesen von mehreren Handbüchern ersparen, so dass mehr Zeit zur Forschung verwendet werden kann.

ABSTRACT

The differential expression analysis is a component of the genetic research. The results of this analysis could amongst other things help to understand the origin of several phenotypic occurrences.

Already many open source packages for the programming language R exist, that could be used to find differentially expressed genes. The only disadvantage is that probably not every scientist is in charge of programming language skills, which are necessary to use those packages.

The outcome of this thesis should provide a solution in form of a pipeline for differential gene expression analysis to scientists which could be used easily and without having programming knowledge. Furthermore multiple packages with different statistical models should be integrated. All common used parameters should be unified across these packages. This measurement should spare the reading of multiple manuals and save time that could be used for research.

Table of Contents

1 Introduction....1
1.1 Why Developing a Differential Expression Analysis Pipeline?....1
1.2 Project Environment....2
1.2.1 SAGE-CARE....2
1.2.2 NSilico Life Science Ltd....3
1.2.3 Outline....3
2 Requirements....4
2.1 Functional Requirements....4
2.1.1 Aggregate Count Data....4
2.1.2 Aggregate Sample Data....4
2.1.3 Differential Expression Analysis Pipeline....4
2.2 Non-Functional Requirements....5
2.2.1 User's Perspective....5
2.2.2 Developer's Perspective....5
3 Background....6
3.1 What is Gene Expression?....6
3.2 Differential Expression Analysis....6
3.3 Sample Tissues....7
3.4 Python....7
3.4.1 Indentation....7
3.4.2 Objects....8
3.4.3 Data Types....8
3.5 R....10
3.5.1 Integrated Development Environment....10
3.5.2 Packages....11
3.5.3 Data Types....12
3.5.4 Classes....15
4 Concept of the Genetic Differential Expression Pipeline....19
4.1 Assemble Count Data....19
4.1.1 Output from RNA-Seq Methods....19
4.1.2 Result....20
4.1.3 Interface....21
4.2 Assemble Sample Data....21
4.2.1 Clinical Data Files....21
4.2.2 Result....22
4.2.3 Interface....22
4.3 Differential Expression Analysis....22
4.3.1 Data Import....23
4.3.2 Filter Count Data....26
4.3.3 Quality Assessment....26
4.3.4 Differential Gene Expression Analysis Packages....27
4.3.5 Comparison of Results....32
5 Implementation of the Differential Gene Expression Analysis Pipeline....33
5.1 assembleRNAseq2....33
5.1.1 main....33
5.1.2 assemble....35

5.1.3 __checkConsistency....35
5.1.4 __checkIntegrity....36
5.2 AssembleClinical....36
5.2.1 assemble....36
5.3 GeneticAnalysisPipeline....36
5.3.1 Environment....37
5.3.2 Classes....37
5.3.3 AnalysisResult....41
5.3.4 Functions....41
6 Evaluation of the Genetic Differential Expression Pipeline....63
6.1 assembleRNAseq2....63
6.2 assembleClinical....64
6.3 GeneticAnalysisPipeline....67
7 Conclusion....75
8 Appendix A....79

List of Figures

Figure 1: This screenshot shows the IDE RStudio with a common window setup for R developers. From left top to bottom right, we can see: The source editor, reachable functions in the global environment, the interactive R console and the window for the graphical output.....11
Figure 2: ER-Diagram (in Chen notation) showing relationship between barcodes and filenames. 19
Figure 3: Simplified flowchart which describes the process chain for the differential expression analysis pipeline (using norm DIN66001).....23
Figure 4: Visualization of the relationship between count data and sample data file. The schematic is oriented on the SummarizedExperiment class [SUME01], which should be used at the implementation.....24
Figure 5: Flowchart of the Differential Expression Analysis conception.(using norm DIN66001) 31
Figure 6: This UML class diagram shows the inheritance between SummarizedExperiment0 and the AnalysisDataSet. Only the most relevant methods and slots are listed. Furthermore the inheritance between SummarizedExperiment0 and the Vector class is omitted.....39
Figure 7: Bar plot generated by the function plotAnalysisInfo with customized theme and color. 71
Figure 8: The PCA plot of the rlog method generated by the function plotQualityAssurance.....72
Figure 9: A heat map "hmDist" is showing the results of the log2 transformation.....73
Figure 10: Venn diagram shows the intersections of the several determined differentially expressed genes between the used packages.....74

List of Tables

Table 1: The Native Data Types of Python.....9
Table 2: A typical package structure in R.....12
Table 3: Data types of R.....13
Table 4: Exemplary line out of FILE_SAMPLE_MAP.txt from TCGA RNA-Seq data.....20
Table 5: Exemplary line out of ID.rsem.gene.results file, with needed gene_id and raw_count column.....20
Table 6: Exemplary output row of the tabular file from the count data aggregation tool.....20
Table 7: This table is showing a part of one row out of the patient file from the clinical data of TCGA.....22
Table 8: Exemplary row from output of AssemblePheno.....22
Tabelle 9: Estimation of Size Factors: Methods and Terminology.....28
Table 10: Exemplary Differential Expression Analysis output of a package.....31
Tabelle 11: Terms with descriptions for the result files.....31
Table 12: Available plots of the function plotQualityAssurance.....47
Table 13: Available plots of the function plotAnalysisInfo.....48
Table 14: Description of the arguments used by the execution of assembleRNAseq2.py.....63
Table 15: Description of the arguments used by the execution of assembleClinical.py.....65

Listings

Listing 3.1: Python "Hello World" example, showing the indentation.....8
Listing 3.2: Operations with native data types in Python.....9
Listing 3.3: R "Hello World" example. No explicit declaration of a data type is needed..10
Listing 3.4: Code example from operations on different data types in R.....14
Listing 3.5: Code example of the usage of S3 R classes.....16
Listing 3.6: Code example of the usage of S4 R classes.....17
Listing 5.1: Usage of the ArgumentParser in assembleRNAseq2 script.....34
Listing 5.2: Detection of non-numeric values out of the detectDataType function.....43
Listing 5.3: Estimating the unambiguity of non-numeric values in the detectDataType function.....43
Listing 5.4: Coercion by the usage of own classes in the detectDataType function.....44
Listing 5.5: Selection of the sample data in the function createAnalysisInfo.....45
Listing 5.6: The filtering of sample data in the function createAnalysisInfo.....46
Listing 5.7: Usage of user given additional arguments for foreign functions in runQualityAssurance.....47
Listing 5.8: Method cpm in the function filterCountData.....50
Listing 5.9: Creation of the standard contrast list by the function ".createStandardContrastList".....51
Listing 5.10: Attachment of user given arguments, function call and creation of an AnalysisResult object with prioriObjects.....52
Listing 5.11: Instantiation of the DESeq2 object in the initDESeq2 wrapper function.....53
Listing 5.12: Attachment of the estimated size factors to the DESeq2 object in the estimateSizeFactorsDESeq2 wrapper function.....53
Listing 5.13: Attachment of the dispersion estimations to the DESeq2 object in the estimateDispersionsDESeq2 wrapper function.....54
Listing 5.14: Fitting of the model and testing of the coefficients in the fitModelTestDESeq2 wrapper function.....55
Listing 5.15: Calculation and constellation of DEA results in the decideTestsDESeq2 wrapper function.....56
Listing 5.16: Normalize results and create ordered subset in the normalizeResultsDESeq2 function.....56
Listing 5.17: Generate dispersion plot in plotPrioriDESeq2 function.....57
Listing 5.18: Generate log2 fold change plot in plotPosterioriDESeq2 function.....57
Listing 5.19: Building a list which contains the joined results out of the AnalysisResult objects list in the joinResults function.....60
Listing 6.1: Execution of assembleRNAseq2.py.....63
Listing 6.2: Execution of assembleClinical.py.....65
Listing 6.3: Execution of cli_createAnalysisDataSet.....69

1 INTRODUCTION

In medical context, these days it is not enough to fight only the symptoms and signs of a disease. To get a better understanding, it is necessary to have a deeper look into the cause of the problem. That this is not only approachable by physicians, but becoming more and more an interdisciplinary challenge, is shown up also by the history of the Human Genome Project (abbr.: HGP).
The HGP ran from 1990 to 2003 and was coordinated through the U.S. Department of Energy and the National Institutes of Health [1]. One of the goals was to sequence the human genome, which includes the assembling (determining of the complete 3 billion DNA base pairs) and furthermore annotation (finding the corresponding label and information to a DNA base pair) of the genomes [2]. To solve this, knowledge from biologists, physicists, chemists, computer scientists, mathematicians and engineers around the world was needed, to develop necessary advanced equipment and tools [3]. Since we nowadays know almost every gene in the human body, the question about what they are doing, becomes more interesting [4]. Through the HGP, the course was set for more lucrative Differential Expression Analysis methods, which could give answers dedicated to this question [5].
Currently there are existing many open source tools, that are able to detect differential expressed genes, by using different statistical approaches. The downside is that they are mostly running under the R environment only and therefore are not usable with having programming knowledge. [6]
The outcome of this work should support physicians by detecting differential expressed genes between samples, without any necessary programming knowledge.

1.1 WHY DEVELOPING A DIFFERENTIAL EXPRESSION ANALYSIS PIPELINE?

The open source and open development project Bioconductor, is providing many tools for the analysis and comprehension of high-throughput genomic data [7]. Just under the topic "DifferentialExpression" are 184 different packages listed (Bioconductor version 3.2). *[6]*

But the downside for physicians with no or low programming skills is, that they are almost all written in R programming language and need to be used under R environment. The amount of packages also could be a bit overwhelming.

Bioconductor has a large international community which can help if any questions or problems occur. Additionally every package should also contain an appropriate documented workflow. Because this practical example contains functional program code, the user could replicate it locally [8].

Unfortunately the terminology of every package can be slightly different. If the results of a package do not satisfy the users estimations, he or she has to search another package and read its documentation again. Additionally the examples are often fitted unique to already prepared input data. Data cleaning and data transformation could be a time consuming procedure.

All in one, the pipeline should make it possible to enable the usage of selected analysis packages from Bioconductor for everyone. The focus of this thesis lies on the integration of multiple adapted differential expression analysis (abbr.: DEA) packages with different underlying statistical models. They should be set up behind an interface, which standardizes parameters and makes the usage as uniform and package independent as possible.
The benefit would be, that the user needs to read only one manual, if any, for all integrated packages.
Standard steps which are belonging to an analysis, should also be integrated and accessible. Examples for this steps are the filtering of count and phenotypic data, the selection of experimental groups and the possibility to visualize data. Another benefit would be, that through the combination of many work process steps into one tool, many error factors would be reduced for the user. He could focus on the results, without worrying about the intermediate steps.
Furthermore, the abstract aspect which underlies the pipeline, should make it possible to integrate different interfaces, which can pass the data comfortable to a presentation layer. Therefore all features could be made accessible through an easy to use graphical user interface and no programming skills are required.
Additionally multiple modules could extend the pipeline with data analysis methods, such as clustering, which makes it possible to find completely new groups and discriminating variables for a hypothesis, or to make a quality assurance.

1.2 Project Environment

This thesis was developed within the ,,SemAntically integrating Genomics with Electronic health records for Cancer CARE" (acronym: SAGE-CARE) project, coordinated by the University of Applied Sciences in Darmstadt and funded by the European Commission [9]. One important partner of this project is the company NSilico Life Science Ltd (abbr.: NSilico), which has also supported actively the work of this thesis.

1.2.1 SAGE-CARE

SAGE-CARE consists out of an interdisciplinary team, with members from different universities and companies. Their main target is to develop software, which offers the possibility to create semantically and intelligent links between data from genetic analysis, or medical results of research and electronic health records [10].
The outcome from the practical part of this thesis, should be integrated into this platform. A use case could be similar to Simplicity™, a product from the participating company NSilico

[11].

1.2.2 NSilico Life Science Ltd

The company is located in the Dublin 4 district in Ireland. Dr. Paul Walsh is Founder and Chief Technology Officer from NSilico [12]. He is having the role of the project manager and was also the first contact person for arising specialist questions respective to genetic and statistical topics.
There are currently three products on the website from NSilico, which are offered for sale:

- SimplicityMDT™: A multi-Disciplinary Team management platform [13].
- MolPath: Serves an application for mobile devices, which could be used for molecular pathology test ordering [14].
- Simplicity™: This product enables the possibility to access remotely multiple open-source tools, to analysis raw sequence data. All necessary steps for the downstream analysis will be easy manageable over a web interface, without necessary coding knowledge [15].

1.2.3 Outline

This thesis contains the following chapters:

- Introduction
- Requirements: The functional and non-functional requirements of the practical part of this thesis.
- Background: All necessary information about the background of this thesis. Biological as well as technical.
- Concept of the Genetic Differential Expression Pipeline: Conceptional part of the developed tools.
- Implementation of the Genetic Differential Expression Pipeline: Here the most important implementation steps are described.
- Evaluation of the Genetic Differential Expression Pipeline: Describes the evaluation of the function and non-functional requirements.
- Conclusion: Explanation of the outcome of this thesis and further steps.

2 REQUIREMENTS

This chapter describes the functional and non-functional requirements of the three applications that should be developed in context of this thesis. The first two tools should have the task to aggregate the input data for the main application.
The variety of programming languages is restricted to R and Python, for all three tools.

2.1 FUNCTIONAL REQUIREMENTS

The following sections are describing the functional requirements of every tool.

2.1.1 Aggregate Count Data

Identification	Description
A.F1	Merge files with count data from separate samples into one file by using a key column.
A.F2	The several identifiers of a count shall be inserted into the unlabeled first column, representing the row names.
A.F3	The keys corresponding to the counts shall represent the column header.

2.1.2 Aggregate Sample Data

Identification	Description
B.F1	Merge files with phenotypic data from separate samples into one file by using a key column.
B.F2	The several keys shall be included as a column.
B.F3	The labels corresponding to the phenotypes shall represent the column header.

2.1.3 Differential Expression Analysis Pipeline

Identification	Description
C.F1	Tool C shall have a functionality to import data from Tool A and Tool B.
C.F2	Data types and missing data from imported data shall be detected automatically.
C.F3	A function shall be available to make it possible to filter and select phenotypic data. The count data must be fitted correspondingly to the alternation.
C.F4	To enable Exploratory Data Analysis for the clinical data, there shall be a possibility to generate and export plots. Minimum requirement: count bar plot for categorical and box plot for numeric values.
C.F5	To enable Exploratory Data Analysis for the count data, the tool shall have the possibility to generate a Principal Component Analysis plot and a heat map with hierarchical clustering.
C.F6	The possibility shall be given to filter count matrices, using common methods.
C.F7	The selection and configuration of different statistical models shall be possible.
C.F8	Significant differentially expressed genes shall be detected, using the previous configured models with the preprocessed data and a user given hypothesis.
C.F9	A comparison between results shall be possible, using a Venn diagram.
C.F10	Significant genes shall be exported to a file in a user given format.

2.2 Non-Functional Requirements

The non-functional requirements are tool independent. For a better readability they are divided into two subcategories. The first is describing all usage dependent requirements and the second all coding style dependent non-functional requirements.

2.2.1 User's Perspective

Identification	Description
NF1	The tools shall be usable without having programming knowledge.
NF2	No package dependent knowledge shall be necessary to use the tools.

2.2.2 Developer's Perspective

Identification	Description

NF3	Integrated differential expression packages shall be easy to maintain and exchange.
NF4	It shall be manageably possible to add other statistical packages.
NF5	System shall be integrable to foreign systems through an interface.

3 BACKGROUND

In this chapter, some basic information about the background and technological environment of this thesis is given. The focus relies here only on thesis relevant topics.

3.1 WHAT IS GENE EXPRESSION?

This chapter is not going too far into detail of genetics, but should just give a basic understanding for interested readers.
Mutations in our deoxyribonucleic acid (abbr.: DNA) are the source of our varied society. Though the complementary double helix is not the only matter of the phenotypic variation. For example, brain cells and liver cells from one individual person should normally have identical genotypes, but their phenotypes are remarkably different. The answer of the question how that could be, is not to find in the physical makeup of the genome, but in its expression.
The term gene expression describes the synthesis of protein, which in turn determines the phenotypic attributes. [4] This process is a fundamental dogma of molecular biology:

The DNA, specifying our genetic information in sections called genes, is converted into a ribonucleic acid (abbr.: RNA) copy, which then is potentially translated into protein. To quantify the gene expression of a tissue, a possible way would be, to count the occurrences of mRNA's (messenger RNA). ([16], p. 1 and 7)

3.2 DIFFERENTIAL EXPRESSION ANALYSIS

To absolve a DEA, it is necessary to quantify the gene occurrences in the sample tissues of interest. There exist different methods for this approach, but this thesis will only focus on the export from high-throughput sequencing machines and in the narrow sense, the method called RNA-sequencing (abbr.: RNA-seq). [17]

The following preparation steps have to be absolved, before the analysis can be started [18]:

1. A DNA Sequencer machine saves the read nucleotide sequences, together with a quality score in FASTQ formatted files.
2. The nucleotide sequences must be mapped to a reference genome or transcriptome. The results are then stored commonly in files having SAM or BAM format.
3. A script counts how many reads map to each feature (here genes) and saves the results in a given format.

Through the usage of the received count matrix, it is now possible to find differentially expressed genes, which means, genes with differences in expression level across experimental conditions [19].
One possible subject could be to get back to the example from the last section, to find genes that are significantly more expressed in a liver than in a brain.

3.3 Sample Tissues

The aim was to create user friendly and comfortable applications, which made it necessary to use real sample data for the development process. Through the development on user level, many unpredictable problems could be detected rapidly and eliminated.
One source of the used sample data is provided by the site of the project "The Cancer Genome Atlas" (abbr.: TCGA). It contains a public available database with sample collections from tissues of different cancer types.
The "Skin Cutaneous Melanoma" (abbr.: SKCM) data set, which was used for this thesis, includes 470 samples overall (see [20]) and corresponding 469 RNA-Seq result files ([21]).

3.4 Python

The programming language Python is developed under an open-source license which is making it free to use and to distribute [22].
Python is currently released in the two different major versions 2.x and 3.x. One of the main reasons why 2.x is still maintained and available, is to keep the support for legacy modules and for operation systems with 2.x installations. Python 3.x otherwise, is stated as the proper choice for a development in an actual environment. [23]
Python is an interpretable language, which means that no compilation and linking is necessary. This aspect enables the possibility to interpret input interactively on the fly. A shell made for this task, is already included by the installation [24].

3.4.1 Indentation

The language is designed to preserve software quality in focus to the code readability and coherence which for many developers sets Python apart from other more traditional scripting languages ([25], p. 3).
Functions have neither an explicit begin or end tag, nor any braces that are marking the begin and end. The only indication for the begin is a colon after the function call and the used indentation of the code itself (see listing 3.1).
As a consequence, the used spaces have to be equal in the whole source code, otherwise it will result in an indentation error. ([26], p. 17)

```
def sayHello(recipient):
    greeting = "Hello"
    print("{0} {1}!".format(greeting, recipient))

sayHello(recipient = "World")
```

Listing 3.1: Python "Hello World" example, showing the indentation.

3.4.2 Objects

The general-purpose programming language is often applied in scripting roles, but supports nevertheless object oriented programming ([25], p. 5). Besides the objects which could be created through classes, everything else is also an object due the object-definition of Python.
An object consist out of a value, a type and an identity. The objects identity (like an address) and type is unchangeable after creation, while the value could be changed if the object is mutable.
The mutability of objects is type dependent. Immutable objects are for example numbers, strings, instances and tuples. Mutable on the other hand are dictionaries and lists. Explicit deletion of objects is not supported, instead they could get deleted by the garbage collector if they are unreachable.[27]

3.4.3 Data Types

Python does not require an explicit data type declaration. Instead it will detect the data type automatically by the value assignment.
It follows a table of some native data types (see also [26], p. 23):

Table 1: *The Native Data Types of Python.*

Numeric Types	Could be int, float or complex.
Text Sequence Type	A string is an object of the class str in Python.
Boolean Values	Constant objects, written as True and False.
Lists	Like in other languages, lists are mutable sorted sequences that could store multiple items.
Tuples	Like lists, but immutable after creation.
Sets	Sets are unsorted collections of unique elements.
Dictionaries	A dictionary is a mutable unsorted set of key value pairs.

The listing 3.2 is showing some exemplary variable assignments with the more specific data types:

```
1  >>> v_list = ["this", "is", "a", "list"]
2  >>> v_list.append("!")
3  >>> print(v_list[1:5])
4  ['is', 'a', 'list', '!']
5  >>> v_tuple = ("this", "is", "a", "tuple")
6  >>> print(v_tuple[-1])
7  tuple
8  >>> v_set_1 = {1, 2, 3}
9  >>> v_set_2 = {1, 2}
10 >>> print(v_set_1 & v_set_2)
11 {1, 2}
12 >>> v_dict = {"key_1": "value"}
13 >>> print(v_dict["key_1"])
14 value
```

Listing 3.2: Operations with native data types in Python.

- Line 1 to 4 is showing the creation of a list with the very common appending function, which inserts the given element at the end. Additionally, a feature which is called splicing is used in line 3. Splicing allows to create a new object out of a given interval. The interval has to be in a range of zero to the length of the given object.
- Line 5 to 7 includes the assignment of a tuple and is showing up that negative values are also allowed for indexing.
- Line 8 to 11 has been used to demonstrate the creation of a set. For sets exist different methods which are common in the set-theory. In case of line 10, the intersection of the two sets was used.
- Line 12 to 14 is demonstrating the usage of directories in Python. The values could be accessed through the key values. The key values could also be stored as a set.

3.5 R

R is a free programming language and environment with the focus on statistical computing and graphics. It can be considered as different implementation of S, which was developed at Bell Laboratories. The source is freely available (under GNU General Public License) and could be installed on a wide variety of UNIX platforms and similar systems, Windows and Mac OS. [28]
R is having some parallels to Python. It is also an interpretable language and comes with its own interactive console. The programming language offers object oriented programming as well and detects data types internally (see listing 3.3). [29]

```
sayHello = function(recipient) {
        greeting = "Hello"
        print(paste(greeting, " ", recipient, "!", sep = ""))
}

sayHello(recipient = "World")
```

Listing 3.3: R "Hello World" example. No explicit declaration of a data type is needed.

3.5.1 Integrated Development Environment

A large arsenal of statistic tools is already included in the R core packages. The results of the analysis could be stored or directly shown by using graphical facilities.
Besides that, a help system is also integrated, which allows to read the documentation of functions by just typing "*?function*" into the interactive console.
To get all the advantages of this environment, it is recommended to use R on an X window system or in an integrated development environment (abbr.: IDE). [29] Possible IDE's are for example Rstudio (see figure 1) or Eclipse with the StatET plug-in.

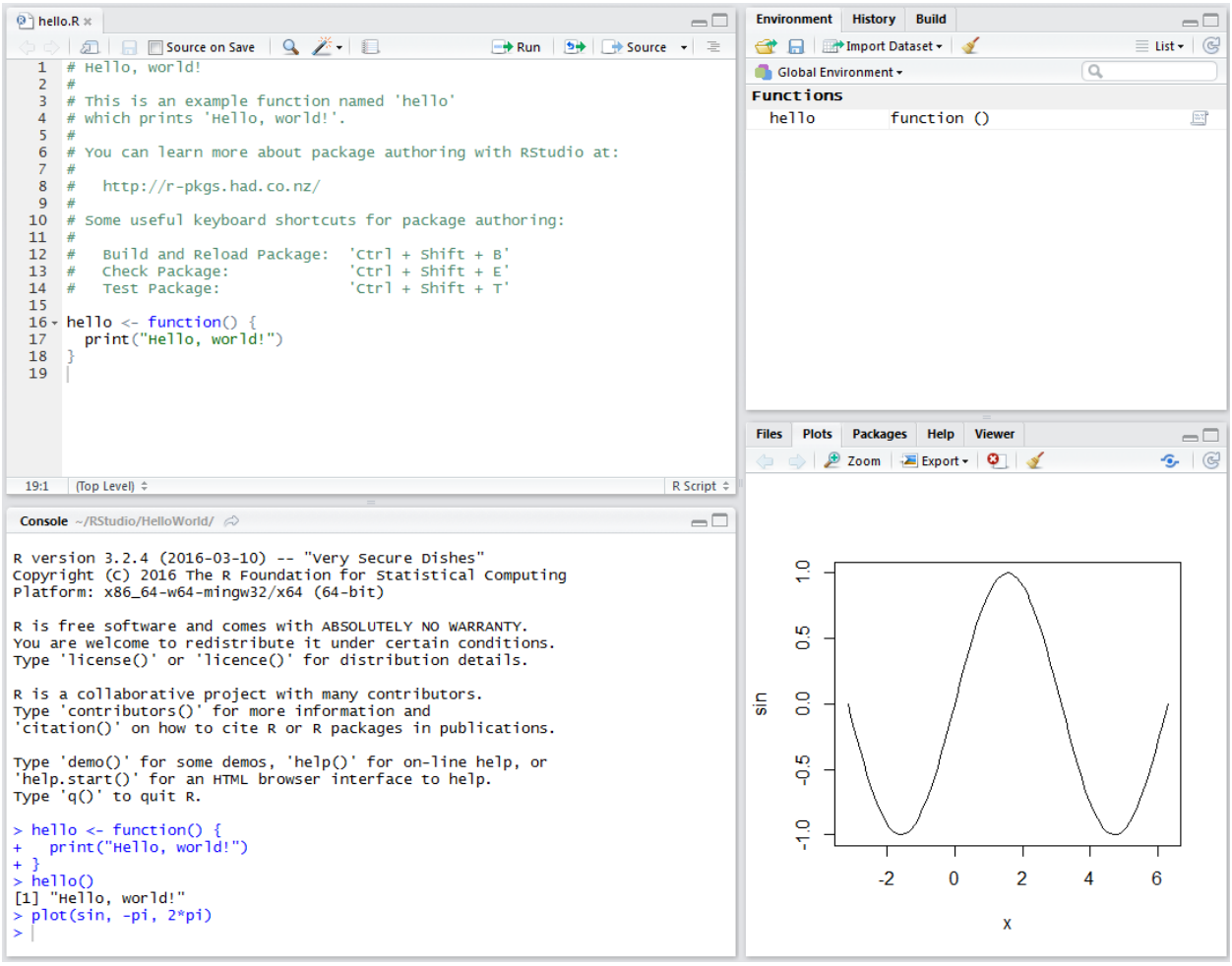

Figure 1: This screenshot shows the IDE RStudio with a common window setup for R developers. From left top to bottom right, we can see: The source editor, reachable functions in the global environment, the interactive R console and the window for the graphical output.

3.5.2 Packages

Many system default functions are written in R itself. That makes it easier for a user to follow the algorithmic choices made and also gives him good examples to provide his own functions.

If he wants to distribute his functions with the R community, one possibility would be to create a package and put it on one of the Comprehensive R Archive Network (abbr.: CRAN) families internet sites. [28]

Before a build package is committed to CRAN, it should be checked (with the flag "--as-cran") against warnings and errors. All necessary tools are already available after the installation of R. The package checker has a high granularity and checks even if the titles are capitalized correctly and examples in the documentation are working. [30]

Package Structure

The smallest possible package consist out of a folder called “R” and two files with the name “DESCRIPTION” and “NAMESPACE”. A more common used structure is described in table 2. [31]

Table 2: *A typical package structure in R.*

/data-raw/	Directory with files that should be excluded in the build package. Typically large source files that have been used for the creation of the compressed files in R format in the /data/ directory. The usage is optional.
/data/	Data which could be loaded by the users of the package. Typically used for examples. The usage is optional.
/man/	Directory which should contain the documentation of each function.
/R/	All source code files should be stored in that directory.
/tests/	This directory should contain all necessary tests.
DESCRIPTION	A file that holds general information about the package, the authors, the dependencies to R and other packages.
NAMESPACE	Defines which variables should be exported from the package, to be reachable by the user and additionally which variables should be imported from other packages.

3.5.3 Data Types

R has its main advantages in working with data structures. The easiest variant of such a structure here is the vector, which consist out of an ordered list of unique elements. These vectors could be used to create more complex structures, like a matrix.
The following table shows the common data types of R [29]:

Table 3: Data types of R.

Numeric Types	Could be integer, numeric or complex.
Text Sequence Type	A string is a vector of the type character.
Boolean Values	Constant objects, written as TRUE and FALSE.
NA Values	NA stands for “Not Available” and is a logical constant for missing values. Further reserved NA constants are NA_integer_, NA_real_, NA_complex_ and NA_character_.
Vector	Vectors are mutable ordered collections of elements, having an atomic basic type. The basic type could be integer, numeric, Boolean, character, complex or raw. It is possible to attach names, to access elements associative.
List	List are similar to vectors, but without restrictions to the type (non atomic).
Matrix	A matrix could be seen as two dimensional array, which consist out of vectors of the same length.
Data frame	Data frames are lists with additional restrictions and have the underlying class “data.frame”. They have to consist out of elements, equal in length, like vectors, factors, numeric matrices, lists, or other data frames. Character vectors will be coerced to factors by default.
Factor	A factor is an object which could be used to describe the discrete classification of a vector. Similar values in this factor will by grouped as a level.

The next figure shows the assignment and exemplary usage of some data types [29]:

```
1  >>> v_vector = c("this", "is", "a", "vector")
2  >>> v_vector[5] = "!"
3  >>> print(v_vector[1:5])
4  [1] "this"   "is"     "a"      "vector" "!"
5  >>> v_list = list("color" = c("green", "blue"), "count" = c(30, 53))
6  >>> print(v_list[2])
7  $count
8  [1] 30 53
9  >>> print(v_list[[1]])
10 [1] "green" "blue"
11 >>> v_matrix = matrix(c(1, 0, 1, 0, 1, 0), nrow = 2, ncol = 3)
12      [,1] [,2] [,3]
13 [1,]    1    1    1
14 [2,]    0    0    0
15 >>> v_data.frame = as.data.frame(v_list)
16 >>> print(v_data.frame)
17   color count
18 1 green    30
19 2  blue    53
20 >>> str(v_data.frame)
21 'data.frame':        2 obs. of  2 variables:
22  $ color: Factor w/ 2 levels "blue","green": 2 1
23  $ count: num  30 53
```

Listing 3.4: Code example from operations on different data types in R.

- Line 1 to 4 illustrates the creation and assignment of a vector to a variable. R allows a different variant of splicing, which is used in Python. Here a vector could be given, to define the index which is used to create a subset:
  - Vector of logical values: All TRUE values are included in the subset, others will be excluded.
  - Vector of positive integral quantities: The corresponding values of the given index will be returned in the given index order. Here the expression 1:5 creates a sequence from 1 to 5. The set has to be in the range of 1 to the length of the object which is used for sub-scripting.
  - Vector of negative integral quantities: Here the corresponding will be excluded in the subset.
  - Vector with names: Identical to the usage of the positive integral quantities, but with character strings which has to fit to the names of the values.
- Line 5 to 10 is showing the instantiation of a list with names. Important is here to remember that single brackets only create subsets, which will be a list in this case. To access single elements it could be used:
  - Double brackets around an index number (like in the example), or the name of the wanted value.
  - A dollar sign followed by the name of the wanted value (here *v_list$color*).
- Line 11 to 14, demonstrates the creation of a matrix. The parameters *nrow* and *ncol*

could be used to define the number of rows and columns of the matrix. R supports out of the box a wide spectrum of matrix calculations.

- Line 15 to 23 is showing how a list could be coerced to a data frame. With the help of the *str* function, which shows up the structure of the data frame, it could be easily shown that the character vector also has been coerced to a factor at the creation time.

3.5.4 Classes

Classes are a central aspect in OOP. In R they mainly define how objects will look like and act, describe the hierarchical context and relationship to other classes and they are used to re-reference methods. There exist three different class types in R, which enable OOP to developers. [32] This explanation will only focus on the more established S3 and S4 classes.

S3

By using S3 classes OOP is enabled by a simple generic function system. The generic functions take control about which method will be called. For this, typically the class of the first argument of these functions is used for the method dispatch.
The classes are more elementary, also in possibilities to define formalities.

```
1  >>> # Class definition
2  >>> myClass = function(txt) {
3  >>>         class(txt) = "myClass"
4  >>>         return(txt)
5  >>> }
6  >>> # Generic function f definition
7  >>> f = function(txt) UseMethod("f")
8  >>> # The f method for a myClass object
9  >>> f.myClass = function(txt) {
10 >>>         print("Method f for 'myClass' dispatched!")
11 >>>         print(txt)
12 >>> }
13 >>> # The print method for a myClass object
14 >>> print.myClass = function(x) {
15 >>>         print(paste0("Print ", class(x), ":"))
16 >>>         class(x) = NULL
17 >>>         NextMethod(x)
18 >>> }
19 >>> # Create object and call method f
20 >>> f(myClass("This is my txt!"))
21 [1] "Method f for 'myClass' dispatched!"
22 [1] "Print myClass:"
23 [1] "This is my txt!"
```

Listing 3.5: Code example of the usage of S3 R classes.

- Line 1 to 5, shows the class definition and assignment through a function. The usage of a function is not required, but enables the possibility of type checking and of tasks a constructor in other OOP languages would take care of. To apply inheritance, the class assignment in line 7 allows also vectors (an example for a vector could be here *class(myClass) = c("myClass", "parentClass")*).
- Line 7, is used for the definition and assignment of the generic function *f*. The function *UseMethod* dispatches the method, by using the class attribute(s) of the argument *txt*.
- Lino 9 to 12, illustrates how the methods of a class must be named, to be found by the dispatcher. Here the generic function *f* is attached to the *myClass* class.
- Line 14 to 18, are used to show the benefit of using the generic function system. The print function already exists and developers are now able to adapt an appropriate method for their class. The function *NextMethod* in line 17, calls the method for the next class, using the class vector. In this case the class is set to NULL and the fallback method *print.default* is called.

S4

The S4 classes, also known as formal classes, are also using a generic function system like the S3 classes, but supporting a more precise definition of their structure and environment.

The main differences are:

- The definition of a class includes its inheritance and fields.
- The dispatch system allows multiple arguments to find the accurate method.
- The operator @ is introduced to access fields of an object. [33]

```
1  >>> # Class definition
2  >>> setClass(Class = "myClass",
3  >>>                 slots = c(txt = "character"))
4  >>> # Generic function f definition
5  >>> setGeneric(name = "f", function(object) {
6  >>>                         standardGeneric("f")
7  >>>                 })
8  >>> # The f method for a myClass object
9  >>> setMethod(f = "f",
10 >>>                 signature = c("myClass"),
11 >>>                 definition = function(object) {
12 >>>                         print("Method f for 'myClass' dispatched!")
13 >>>                         print(object)
14 >>>                 })
15 >>> # The show method for a myClass object
16 >>> setMethod(f = "show",
17 >>>                 signature = c("myClass"),
18 >>>                 definition = function(object) {
19 >>>                         print(paste0("Print ", class(object), ":"))
20 >>>                         callNextMethod(object@txt)
21 >>>                 })
22 >>> # Create object
23 >>> myClassObj = new("myClass", txt = "This is my txt!")
24 >>> # Call method f
25 >>> f(myClassObj)
26 [1] "Method f for 'myClass' dispatched!"
27 [1] "Print myClass:"
28 [1] "This is my txt!"
```

Listing 3.6: Code example of the usage of S4 R classes.

- Line 1 to 3 shows the class definition and assignment through a function. S4 is much stricter as S3, which only uses the *class* function for a name and the inheritance. The developer has to define *slots*. These store the information about the names and possible classes of the fields, that will later be accessible. Further possible arguments are:
  - contains: To define parent classes, where this class is based on.
  - validity: That makes it possible to introduce validity checks.

    - prototype: To define default values for the slots.
- Line 5 to 14 presents how a new generic function is created and adapted in a method. The field signature in line 10 defines the classes, which should be used for the method dispatch.
- Line 16 to 21 describes, analog to the S3 example, the usage of an already existing generic function. Because *print* is based on S3, *show* should be used instead. Line 20 also shows the usage of the special @ operator, to access the slots of the object. [32]
- Line 23 is illustrating the usage of the more OOP typical *new* function, to create the *myClass* object.

4 Concept of the Genetic Differential Expression Pipeline

According to the requirements out of chapter 2, the pipeline has been divided into three independent components. The next two chapters will describe the conception of the two tools which should be developed to support data preparation steps. They could be used to transform data to the proper format according to the main application, whose conception will be described in the third chapter.

4.1 Assemble Count Data

This section describes the concept of the tool which should aggregate files which include count data. Its source files should contain already aligned and quantified gene counts, which are generated with tools like RSEM (abbr. for RNA-Seq by Expectation-Maximization, see [34]) or HTSeq-Count ([35]).
The purpose of the program is, to create one assembled RNA-Seq count file, which could be used together with the corresponding phenotypic file as input for the DEA Pipeline.

4.1.1 Output from RNA-Seq Methods

In the case of the TCGA sample data (see chapter 3.3 for more information), RSEM was used for the quantification of the gene and isoform abundances. After extracting the downloaded archive, results could be found in a subdirectory with the name “Level_3”. [36]
Every sequenced tissue corresponds to six different files:

1. <ID>.exon_quantification.txt
2. <ID>.junction_quantification.txt
3. <ID>.rsem.genes.results
4. <ID>.rsem.genes.normalized_results
5. <ID>.rsem.isoforms.results
6. <ID>.rsem.isoforms.normalized_results

The <ID> is here a replacement for a unique file identifier, which could be used to map the files to the corresponding tissue (with cardinality [1,n]:1), like shown in figure 2.

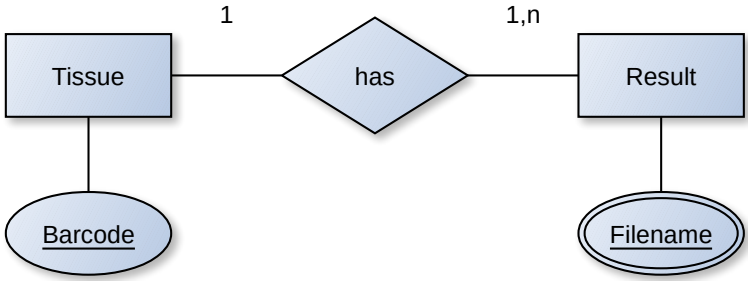

Figure 2: ER-Diagram (in Chen notation) showing relationship between barcodes and filenames.

The tissues also have unique identifiers (in TCGA context called barcodes), which consists out of 7 alphanumeric groups in the following form:

XXXX-XX-XXXX-YYY-YYY-YYYY-YY

The association between filenames and the barcode of the tissue is described in a file with name "FILE_SAMPLE_MAP.txt" (see table 4).

Table 4: Exemplary line out of FILE_SAMPLE_MAP.txt from TCGA RNA-Seq data.

filename	barcode(s)
ID.rsem.genes.results	TCGA-XX-XXXX-YYY-YYY-YYYY-YY

The first 3 parts from the code (marked as "X"), are matching to the patient-barcode, which is used in the clinical data sets. These data sets containing the corresponding phenotypic information from the tissue source. The "Y" part is not needed and could be ignored.

4.1.2 Result

The output from AssembleCounts should be a tabular file, which includes the raw gene count information of all samples. The genes should be able to be identified per rowname and the corresponding samples via column labels (see table 7). It should also be possible to map the phenotypic information gathered with sample data assembling tool, by using the corresponding identifiers in the table header.
In the case of the TCGA RNA-Seq data, the files with the ending ".rsem.genes.results" are containing the raw count information, together with the several gene label (see table 6).

Table 5: Exemplary line out of ID.rsem.gene.results file, with needed gene_id and raw_count column.

gene_id	**raw_count**	scaled_estimate	transcript_id
IGJ|3512	**23211.00**	0.000447497324805475	uc003hfn.3,uc010ihz.2

Together with the modified barcode (without the "Y" part) from the "FILE_SAMPLE_MAP.txt", all required information to write the target file is now available.
It should be noted, that the entity-relationship between barcode and patient_barcode is not always one to one, because in some cases multiple specimen exist.

Table 6: Exemplary output row of the tabular file from the count data aggregation tool.

	TCGA-XX-XXXX	TCGA-XX-XXXX	...
IGJ|3512	504	3431	...

4.1.3 Interface

All file dependent parameters have to be controllable by the user through an interface. The modification of the barcode is restricted to TCGA data only and shall therefore be optional.

4.2 Assemble Sample Data

This chapter includes the conception of the second tool, which should be responsible for the aggregation of the sample data.
The content describes the phenotypic information for a sample (in TCGA context stated as clinical data). Each sample tissue should be listed row-wise, whereby several attributes and informations are given per column. The outcome from this tool should be a tabular file, which corresponds to the tool described earlier in chapter 4.1. Both files are ready to use with the Differential Expression Pipeline.

4.2.1 Clinical Data Files

The TCGA clinical and biospecimen data files, which were used in the context of this thesis, are available in Extensible Markup Language (acronym XML) or tabular separated format.
As input for this tool the tabular files will be used, because they are already close to the defined output format and need less transformation steps. The XML files are redundant and could therefore be ignored.
The TCGA data is split by different topics, by using separate files. All files could be aggregated through a barcode, which is associated uniquely with the patients clinical data. But not every topic has a one to one relationship to the sample tissues, so that it is necessary to select files where only one row for one barcode exists, or none (1:0,1). [37]

Following list was aggregated for the evaluation, shown in chapter 6:

1. nationwidechildrens.org_clinical_patient_skcm.txt
2. nationwidechildrens.org_clinical_follow_up_v2.0_skcm.txt
3. nationwidechildrens.org_clinical_radiation_skcm.txt

The content of the main file (number 1, with patient information) is described in the table below:

Table 7: This table is showing a part of one row out of the patient file from the clinical data of TCGA.

bcr_patient_uuid	bcr_patient_barcode	form_completion_date	...
bcr_patient_uuid	bcr_patient_barcode	form_completion_date	...
CDE_ID:	CDE_ID:2673794	CDE_ID:3088528	...
XXXXXXXX-XXXX-(...)	TCGA-XX-XXXX	2014-1-1	...

4.2.2 Result

Generally the resulting tabular file should contain phenotypic information which is necessary for the analysis. One column has to include keys which could be used to map the corresponding column label from the assembled count file. Additionally the content of each columns should be described with a unique column name (example in table 9).
Especially for the TCGA files, the first and the third lines of the clinical data are reserved for additional column information and not needed for the further process (see table 7).
Therefore it should be optionally possible to remove this lines with the help of the tool.
Furthermore it should be possible to join other files through the given ID (in database terminology they would be "left joined"). In the case of the TCGA data, the main construct will be built out of the file "nationwidechildrens.org_clinical_patient_skcm.txt" and the ID will be the patient barcode.

Table 8: Exemplary row from output of AssemblePheno.

bcr_patient_uuid	bcr_patient_barcode	form_completion_date	...
XXXXXXXX-XXXX-(...)	TCGA-XX-XXXX	2014-1-1	...

4.2.3 Interface

The parameters should, analog to AssembleCounts, not be dependent to the TCGA data. See section 4.2.3 for more information.

4.3 Differential Expression Analysis

This chapter describes the high level concept of the main application of the Differential Expression Analysis pipeline. The application could be further divided into multiple segments, grouped by their function. All of these are oriented on a typical analysis workflow and should be dynamically controllable by the user. The aim is to offer more experienced

users the opportunity to create more complex experiments, by leaving also the possibility for simple standard setups.

The process steps are the following:

1. Data Import
2. Filter Count Data
3. Data Quality Assessment
4. Differential Expression Analysis
5. Result Comparison

In some cases, it could be necessary for the user to repeat several steps, until he gets a result that is more satisfying to his assumption. For example, it is not obviously how the dimensions of the count data will change, in respect to the set filter parameters. For this case, the user should be able take multiple loops between point 2 and point 3.
For the sake of efficiency, the output data of each process should be able to be stored. The simplified flowchart shown in figure 3, represents the resulting pipeline scheme and the additional entry points of the process steps.

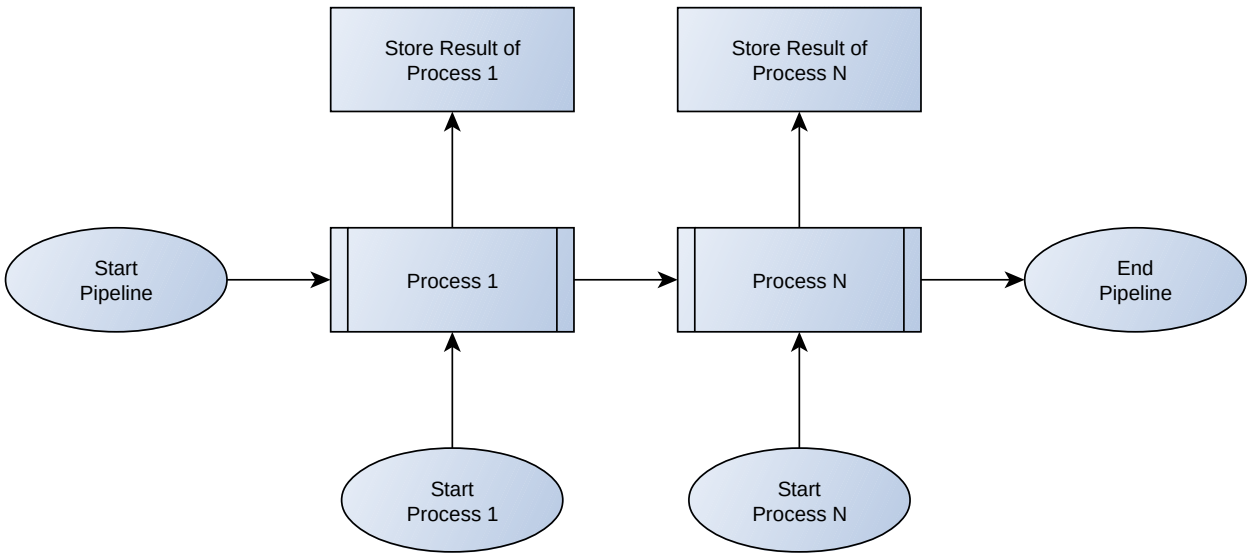

Figure 3: Simplified flowchart which describes the process chain for the differential expression analysis pipeline (using norm DIN66001).

4.3.1 Data Import

The first process step should adapt the source data to the pipeline. This consist out of one tabular file including the count data of each gene and another one that contains all necessary phenotypic information about the sequenced samples. Due to the requirements in chapter 2, the files should already be in an appropriate format, that is illustrated in figure 4.

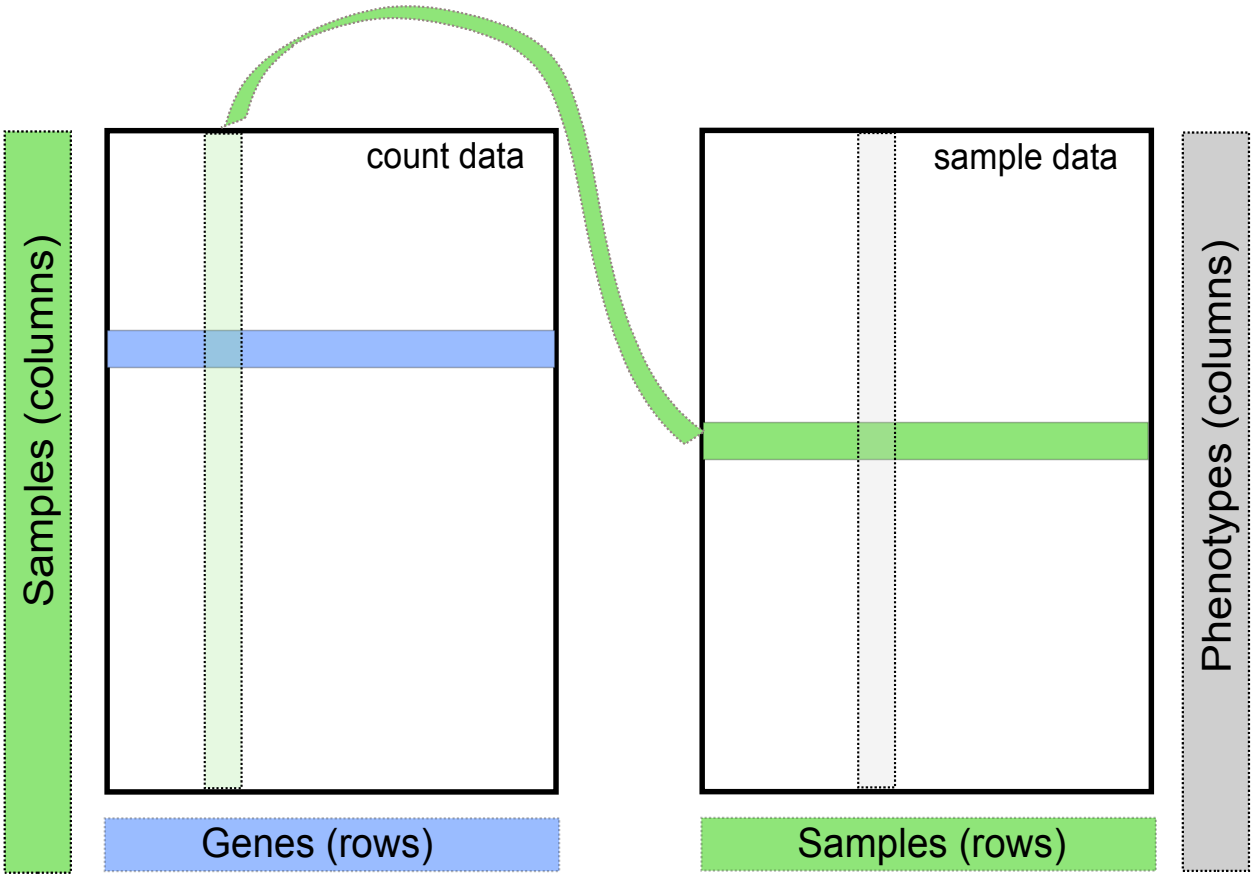

Figure 4: Visualization of the relationship between count data and sample data file. The schematic is oriented on the SummarizedExperiment class [SUME01], which should be used at the implementation.

More information about the file contents is given in chapter 4.1 and 4.2, where the conception of the assemble tools is described.

Data Type Detection

According to section 2.1.3, it is necessary to detect missing values in the sample data. This would also be the first process step, which has to be absolved before a data conversion could be applied. For the task of this pipeline, it is not relevant to cover all possible data types of R. The focus should rely on the following ones:

1. Factor
2. Numeric
3. Date

It is possible to convert numeric values and dates into factors without problems, but not vice versa. This in return means, that every column that should be converted wrongly, should be a factor in the end.
The downside on factors, is the detection of missing values (NA's). The risk to loose data

through a coded assumption, which coerces factor levels into NA values is too high. Therefore an optional parameter should exist, which is making it possible for the user to manual set several NA strings to support the conversion.
Through this convention, it should be only necessary to detect numeric, or date values. To do so, given thresholds could be used:

- The upper bound threshold, which defines the percentage of occurrences a data type should have to assume this data type for the whole column. All found words should be set as NA values.
- The lower bound threshold, that defines the minimum amount of occurrences a data type should have, to make further and more intensive investigations about the uniqueness of the other values (see NA tolerance below).
- The NA tolerance, which only should be used if the upper bound was missed, but the lower bound was reached. It defines how many different NA words are allowed per column and is compared to the uniqueness of the data type foreign values.

This combination of three thresholds makes it possible for the user to adjust the detection optimally to the precision of the sample data. Furthermore efficiency is maintained, by using the lower bound threshold.

Selection and Filtering

This section is about the filtering and selection of phenotypic data. The conception of the filter for the count data is described in chapter 4.3.2.
The constellation of the groups which should be tested against differences in gene expression is an important preparation step. The idea this thesis follows, is to coerce a user given string to a Boolean expression. The string should already consist out of the column name (or position) and one or multiple values. The connection between each word should be established by logical operators. The resulting expression could then be used as index for a given table, returning only the rows of interest.
The columns of the responding table have to cover all design formula attributes at minimum. This means that further columns will not be necessary and should be removed, to save memory space. The user should be able to switch between manual selection, which is for example recommended if the aim is only to produce principal component analysis plots, or automatic selection of the sample columns through a given formula. One or more columns should be selected through the column name or the column number.

Control Plots

To get some feedback about the current selected and filtered sample data for the user it should be possible to generate generic bar charts or box plots.
In the case of bar charts, the data for the x-axis should be generated out of a given column. The values of the y-axis, which are reflecting the occurrences of each x-axis level, has to be calculated automatically.
For box plots a factorial value should be able to be used with the x-axis, similar to the bar chart. But additionally a column name for a column with numeric content is needed, to

generate the boxes of every level stated on the x axis.

4.3.2 Filter Count Data

There is some discrepancy about the necessity of the need to filter RNA-Seq count data between the user guides of the packages edgeR, DESeq2 and limma. This is caused through some implemented automatism in DESeq2, which filters independently on the mean of normalized counts (see [38], p. 8).
However, filtering genes with a low count across all samples is recommended by all packages. This preparation step saves memory and will speed up the analysis calculations. This thesis is following the idea out of the limma and edgeR package documentation that instead of using only one threshold for the minimum count, includes the calculation of the *count-per-million* (abbr.: CPM) of every gene per sample. Furthermore, also a group dependent threshold is used.
The CPM is calculated similar to percentage, with the only difference that million is used instead of hundred:

$$CPM_{sample} = \frac{count_{sample}}{count_{total}} * 10^6$$

For example, two groups should be tested against differential expressed genes. One group consist out of 12 samples and the other out of 9. Through the calculation and the usage of two thresholds, it is now possible to let genes pass through, that have 9 times a count of zero but are otherwise highly expressed.

4.3.3 Quality Assessment

It should be possible for the user to inspect the count data under the aspects of data quality assessment and quality control. In context of this thesis and the related work the definition of the term quality is fitness for purpose. The purpose is the detection of differential expressed genes and the aim is to discover samples, whose experimental treatment suffered from an abnormality which biases negatively the detection (similar to the definition, used in the DESeq2 manual [39], p. 20).
Two possible ways to check the quality of the count data are the usage of *clustering methods* and the *principal component analysis*. To achieve better results it might be useful to work with transformed count data. One common choice for the transformation method is the logarithm. DESeq2 is also offering two alternatives which transform the count data with respect to the *library size*. (see [39], p. 17)
The term library size (or sequencing depth) is used for the result of the sum of all gene counts per sample (see [39], p. 10).

Log2

The binary logarithm is one approach to transform the count data. Since some genes could

have a count of zero, it is recommended to add a pseudo-count. In this thesis we use the following formula $y = \log(n+1)$. (see [39], p. 18)

Regularized Log Transformation

The input for the regularized log transformation function (abbr.: rlog) should be count data with attached estimated size factors and dispersion parameters (further information in chapter 4.3.4, under 'Estimation of Size Factors' and 'Estimation of Dispersion' for DESeq2). To accomplish blind estimations, which means that no information about the experimental groups is used within the calculations, the design formula is replaced with an intercept only (formula = ~ 1). The model fitting and therefore the results are based on the log2 scale.

Variance Stabilizing Transformation

The variance stabilizing transformation (abbr.: vst) makes, in the case of parametric fitted counts (the default), usage of a closed-form expression to transform the normalized count data. Large counts become asymptotic to the output of the logarithm to the base 2. The rlog function is more robust in the case when size factors vary widely. ([39], pp. 48, 49)

Control Plots

With the usage of the normalized count data and the help of clustering or principal component methods, several plots could be produced. This should be possible, analog to the other process steps, independent from the process.

4.3.4 Differential Gene Expression Analysis Packages

The section contains the conceptional integration of the analysis packages into the pipeline. Because of that the most packages are written in R, this part of the pipeline should be also written in this programming language.
The initial implementation will focus on the adaption of three packages with different underlying statistical approaches. The decision was made to use “edgeR”, “DESeq2” and “Limma”, which besides their difference to each other, are also having the best download statistics according due Bioconductor (see [40]).
To preserve modularity and a clean code structure at the implementation phase, it was necessary to concept a wrapper, that could be attached to each package. For this, similar process steps which are an integral part of the whole analysis, had to be uniformly grouped. These stages will be described in the following sections.

Initialization

At the initialization step, all necessary information for of the analysis should be used for instantiation of the package depending object. A typical experimental setup consist out of the following parameters:

1. Count data (optional filtered)
2. Phenotypic data (optional filtered and with necessary columns only)
3. Formula (describing the groups, using the columns from point 2)
4. [optional] Contrast (a vector, that specifies the hypothesis of interest)

The given formula is correlated with the phenotypic data and together they are defining the sample groups (also called replicates). Typically the result of this correlation is saved within a binary design matrix (also known as model matrix).
The contrast has to be a vector with the length of the coefficient count. Each position in this vector will then be used as factor of the corresponding coefficient (see [41], p. 28).

Estimation of Size Factors

The instantiated objects could be now used for the respective methods that are in charge of the count data normalization process. The task is handled differently across the packages and in most cases specialized in respect to the individual statistical model.
In the following table, the terminology and standard methods of the packages are described.

Tabelle 9: Estimation of Size Factors: Methods and Terminology.

Package	Terminology	Result
edgeR	Normalization Factor	Calculates the trimmed mean of M-values over all gene counts per sample (also called library) to a reference sample. If not set by user, the reference will be the sample whose upper quartile is closest to the mean upper quartile. [42]
DESeq2	Size Factor	Divides each gene count through the corresponding count of a reference sample, which is build out of the geometric mean per gene over all samples. The median of all quotients is the result for the library depth. ([39], p. 19)
Limma	See edgeR	Limma is making use of the edgeR normalization approach.

This thesis and the related implementation will use the “size factor” term for the normalization factors of the library sizes (see also [41], p. 13).

Estimation of Dispersion

While the estimation of the size factors could be used to normalize differing sequencing depths between samples, or other sample independent biases, the estimation of the dispersion could be used to account inner sample group variations. One possible cause of inter-library variation between replicates is the biological variation. ([41], p. 15)

Typically this variation is moderate between genetically identical animals, but can be very large in case of human samples [43].

edgeR

The package edgeR serves three different approaches for the differential expression analysis. One former exact method, also called *classic edgeR* and the latter, which are using a Generalized Linear Model, mostly referred as *glm edgeR ([41], p. 5)*.

1. *classic*: The classic approach makes usage of the quantile-adjusted conditional maximum likelihood (abbr.: qCML) method. This method could only be used for pairwise comparisons between groups. ([41], p. 17)
2. glm: The glm methods allow experiments with multiple factors by using the Cox-Reid profile-adjusted likelihood (abbr.: CR). So called tag-wise dispersion estimation is also supported, which weights the estimated gene counts individually. ([41], pp. 18-19)

DESeq2

Similar to edgeR, this package uses the CR method and calculates the dispersion tag-wise (see [44], p. 16).

Limma

Limma uses a linear model, which could be applied on data that has been generated with microarrays. Because of the different underlying distribution to RNA-Seq count data, it is necessary to adjust the data in a previous step. [45]
Voom (acronym for variance modeling at the observation level) normalizes the counts in respect to the library size by transforming them to log-counts per million (abbr.: log-cpm). Furthermore it modifies limma's empirical Bayes procedure, to incorporate a mean-variance trend. Besides that, the mean-variance trend is used to generate precision weights for each individual normalized log-count.
The result could now be used with the limma pipeline, or any other microarray pipeline that is precision weight aware. [46]

Fitting and Testing of the Statistical Model

This stage includes all steps that are necessary for the alignment of the model to the read counts for each gene. Additionally the elevation of gene-wise statistical tests for a given coefficient or coefficient contrast, is also included in this stage.

edgeR

1. *classic*: With the knowledge about the conditional distribution for the sum of counts in a group, p-values that have a probability less than the probability under the null hypothesis of the observed counts could be calculated. ([41], p. 18)
   Like already written in the last section, only pairwise comparison can be made with the classic methods. Through this restriction, the usage of a contrast parameter is not possible. Therefore the classic mode will be excluded in the initial part of the implementation and not further described in this thesis.
2. *glm*: EdgeR serves two generalized linear model approaches.
   1. The usage of a likelihood ratio test for one or more coefficients in a negative binomial generalized log-linear model.

2. Similar method, except that it uses quasi-likelihood (abbr.: QL) for the dispersion values and replaces likelihood ratio tests with empirical Bayes QL F-tests ([44], p. 67). The advantages are, that this method also reflects uncertainty in estimating the dispersion for each gene. Also it provides a more robust and reliable error rate control, in the case of a low replicate count. ([41], p. 20)

DESeq2

The DESeq2 package makes it also possible to decide between two methods:

1. As standard method, DESeq2 uses Wald p-value calculation by testing the estimated standard error of a log2 fold change against zero.
2. The additional method offered by the package uses a likelihood ratio test (abbr. LRT), similar to that which is used by edgeR. ([38], p. 26)

Limma

This package fits the linear model with the usage of the least squares method as standard and computes empirical Bayes moderated t-statistics for the hypothesis test ([47], pp. 98 and 60).

Multiple Testing and p-Value Adjustment

At the final stage, a classification of genes in respect to their significance should be made using the data from the previous steps. The standard false discovery rate (abbr.: FDR) has to be corrected with the Benjamin-Hochberg (abbr.: BH) procedure.

Output

To check the data quality, it is a common task to look at the data which has been generated in the "Estimation of Dispersion" stage. Special plots are offered by the packages, which are made exactly for this control step. With the help of this plots, a decision could be made to change the experiment parameters (for example the filter for the count data), or to step forward to the next stages and complete the analysis.
To enable the package depending plots, two entry and exit points have to be integrated in the package depended part of the pipeline (see flowchart in figure 5).

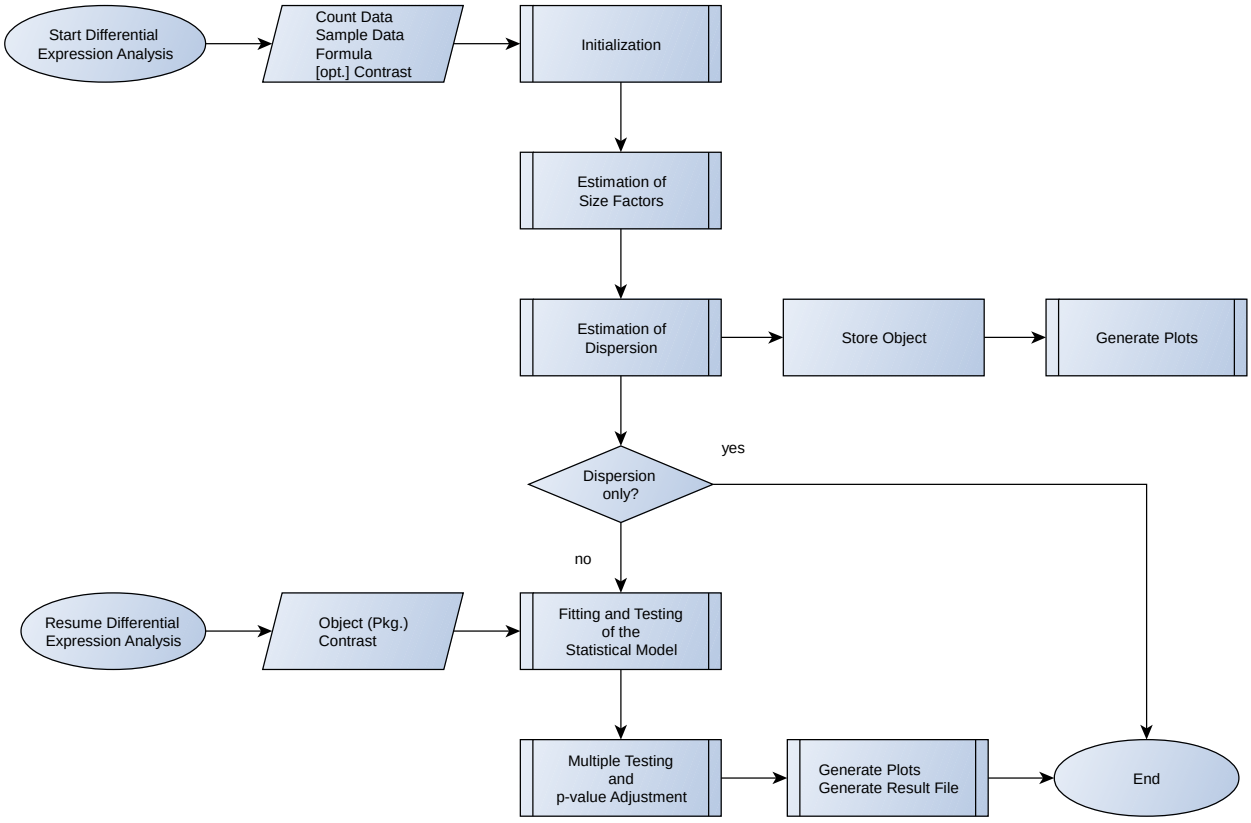

Figure 5: Flowchart of the Differential Expression Analysis conception.(using norm DIN66001)

At the end of a differential expression analysis, a result file for every used package should exist, containing the following exemplary data as minimum:

Table 10: Exemplary Differential Expression Analysis output of a package.

	log2FoldChange	pvalue	padj
LAMC2|3918	4.073	3.868e-09	5.133e-05

The terminology of the column labels should be consistent between packages and altered if necessary, by using the following list as orientation.

Tabelle 11: Terms with descriptions for the result files.

Term	Description
log2FoldChange	Logarithmic fold-change between the conditions.
pvalue	Probability value from test statistic (Abbr.: p-value).
padj	Adjusted p-value.
baseMean	Mean of normalized counts of all samples.

4.3.5 Comparison of Results

Another interesting aspect besides the result of every single package, is the intersection of significant genes between the packages.
For that the user should have the possibility to create one table with the union of all differentially expressed genes per package ($Genes_{edgeR} \cup Genes_{DESeq2} \cup Genes_{limma}$) and another table with only the intersection ($Genes_{edgeR} \cap Genes_{DESeq2} \cap Genes_{limma}$, similar to an inner join). For this, the gene labels have to be used as keys.
The same strategy should also fit for the creation of the Venn-diagram.

5 Implementation of the Differential Gene Expression Analysis Pipeline

In this chapter, the implementation phase of the three tools is described.
The first two tools, which have the task to aggregate the count and sample data, were written with the programming language Python. Python allows a fast and straight forward development and has a similar underlying function principle like R. This measure also clearly divorces the previous data preparation steps physically from the main application. Should other assembler tools be necessary in the future, they could be implemented parallelly to the two scripts and later chosen conditionally to the type of the source files.
As already mentioned in the concept chapter, it was evident to stay as close as possible to the most important part of the whole pipeline, the differential expression analysis packages. Therefore the main application was written in R programming language.

While all kinds of user interfaces are integrable into the three tools, in the related work of this thesis an exemplary commando line interface was used. As a result, the saved time from the reduced additional workload that would be necessary by using most of the other interfaces, could be used for more essential components. Furthermore the commando line could also be used by web-based platforms (for example Galaxy [48]).

5.1 assembleRNAseq2

The script assembleRNAseq2 imports the following Python modules (without modules that are required for tests):

- *os*: For system dependent tasks like file handling.
- *csv*: Serves features to read and write spreadsheets.
- *re*: Enables regular expression operations in Python.
- *argparse*: The classes ArgumentParser and RawDescriptionHelpFormatter supporting usage of a commando-line interface.

It mainly consist out of four important functions, that will be described in further detail in the next sections.

5.1.1 main

The main function currently includes the commando-line interface of the assembleRNAseq2 Python script and will be started only if it is called as main program by the interpreter.

The usage of the *ArgumentParser* class, helped a lot by the assignment and control of the altogether 14 possible arguments or options.
The ArgumentParser object automatically generates help messages and parses the user given arguments by using a previously defined rule set. The user arguments as default will be divided into two groups. The *optional arguments* (see line 7 to 11 in listing Fehler: Referenz nicht gefunden), which could be supplied by the user after a given option flag and *positional parameters* (see line 12 to 19 in listing Fehler: Referenz nicht gefunden), which will be evaluated through their position relative to the context.

```
1  . . .
2  # Initialize ArgumentParser
3  parser = ArgumentParser(description = prog_shortdesc,
4              formatter_class = RawDescriptionHelpFormatter,
5              epilog = prog_epilog)
6  # Setup input section
7  parser.add_argument(dest = "source_path",
8            metavar = "path",
9            type = str,
10           help = ("path to top source directory (including " +
11               "FILE_SAMPLE_MAP)[default: %(default)s]"))
12 parser.add_argument("-fmap",
13           "--file_sample_map",
14           dest = "filename_sample_map",
15           type = str,
16           default = "FILE_SAMPLE_MAP.txt",
17           metavar = "%s",
18           help = ("filename from file to sample mapping " +
19               "file [default: %(default)s]"))
20 . . .
21 # Parse arguments
22 args = parser.parse_args()
23 # Variable assignment
24 source_path = args.source_path
25 . . .
```

Listing 5.1: Usage of the ArgumentParser in assembleRNAseq2 script.

- Line 3 to 5, illustrates the instantiation of an ArgumentParser object. Here a description is supplied and will be shown by the automatic generated help, using the -h or - -help flag by the execution. Also the class *RawDescriptionHelpFormatter* which handles the formatting and an epilog with examples was added.
- Line 7 to 11, is showing how the method *add_argument* is used to add a positional argument. As default the count of positional arguments is estimated out of the given input type using the *type* parameter. However, in this case the only first given argument is stored in an object field called “source_path”, which is equal to the *dest* parameter. Furthermore, *metavar* is used for the typical usage text of the script and

the *help* attribute could be used for a short description. In that description a format specifier could be used to substitute a given default parameter (here "none").
- Line 12 to 19, describes an added optional argument. The parameters are analog to the parameters explained above.
- Line 22, is used to show the assignment of the object with the parsed attributes.
- Line 24, here the *source_path* attribute is assigned to a variable for further usage.

(see also [49])

5.1.2 assemble

The function assemble is used for all steps that are necessary to assemble the count data files. The functions *__checkIntegrity* and *__checkConsistency* are also included and could be activated by a user given Boolean parameter. By default all parameters fit to the TCGA data, but could of course be changed by the user.
At first, the source files have to be selected in the given directory. For the selection of the source files, the first column of FILE_SAMPLE_MAP.txt (see chapter 4.1.1) is checked against a regular expression (default: *^unc\\.edu\\..+\\.rsem.genes.results*). Matches will be stored as key in a dictionary with the corresponding patient barcode as value. The patient barcode is a substring of the given barcode in the second column (the X part with a length of 12).
The resulting dictionary includes now all necessary files with their corresponding patient barcodes. As next, the files could be optional checked for completeness and correctness using the *__checkIntegrity* and *__checkConsistency* function.
If the checks were successful, every file is read and the corresponding column with the count data is saved to list. Additionally, on the first run the defined column which contains the gene labels is also stored to a list. The *DictReader* class from the csv module allows an associative or numeric column selection, which in return offers the user a higher flexibility.
At the final step, the *writer* method from csv is used to insert the cached information into the resulting file. The first row has to be the header, so the list of the patient barcodes with a leading empty column is used for the insertion. The remaining information is inserted by using a loop over the gene labels (rows) and another which iterates across the count data of each sample (columns).

5.1.3 __checkConsistency

This internal function is making use of the dictionary which was generated out of the information of FILE_SAMPLE_MAP.txt by the function assemble. It is using the given source directory and the filenames (keys), to check if the files exist and are legit.
The similar function is also made available externally (*checkConsistency*), but this one has to create the dictionary again.

5.1.4 __checkIntegrity

The internal function *__checkIntegrity* is also making usage of the generated dictionary from assemble, like in *__checkConsistency*. Here the key column (the gene label) of the count data files is checked for integrity over all samples. This is necessary, because a wrong order will not be detected by the generation of the count data list in assemble and would lead consequently to wrong results.

5.2 ASSEMBLECLINICAL

The Python script *assembleClinical* uses the same module basis like *assembleRNAseq2* which was described in the last chapter. It allows the concatenation of multiple files to one base file by using a key (see chapter 4.2.1) and has the following restrictions:

1. Every column label has to be atomic, otherwise:
    1. In the same file, earlier entries for that column will be overwritten.
    2. In a different file, ambiguous entries will be ignored.
2. The ID (patient barcode) has to be atomic, otherwise earlier entries will be overwritten.
3. The first given file will build the basis of the aggregation and has to contain all necessary ID's.

The script consists out of a main function that contain the commando-line interface functionalities (see 5.1.1 for more information) and an assemble function which is explained more detailed in the next section.

5.2.1 assemble

Otherwise than in *assembleRNAseq2*, all source files are given by the user and are checked for existence directly.
After this control step, a list with unique fieldnames and a dictionary with the patient barcode as key and a further dictionary as value is created. The further dictionary includes the fieldnames and the corresponding values. Fieldnames that are no keys, could be extended with the usage of a user given prefix. This is done by changing the *fieldnames* field of the *DictReader* object.
Finally the list and the values of the dictionary, whose are dictionaries themselves, could be used to write the new file with the use of the *DictWriter* class.

5.3 GENETICANALYSISPIPELINE

The advantage by the implementation of the DEA part, was to preserve a high reusability of the source code by retaining the possibilities of the used analysis packages.
One way to make the future developers life easier was to create an own package. This decision has advantages for the user as well. For example it is now possible to use the install function of R, which also checks for dependencies. Furthermore the documentation could be accessed by using the help function.
The package with the name GeneticAnalysisPipeline consist out of 4 classes and mainly 21 functions (without the generic functions).

5.3.1 Environment

The R package was developed under the operating system Microsoft Windows 7 with the IDE Eclipse and the installed plug-in StatET. Furthermore Rtools and MiKTEX had been installed, which include necessary tools for the package compilation.
Also important for the development, was the installation of the following R packages:

- devtools: Includes a set of development tools, which for example allow the developer to load the package quite similar like it would has been installed already and initialized by the *library* function [50].
- roxygen2: Supports creation of the R documentation files (filename extension Rd) by using in-code documentation [51].
- testthat: Could be used for unit tests creation and evaluation [52].

5.3.2 Classes

Another approach to serve a better usability of the package and reusability of the code, was the introduction of four own classes. Therefore the S4 class system (see chapter 3.5.4) was chosen, because among other things, it has an integrated validation check for objects.
Based on the classes it was also possible to define the outcome of several pipeline stages precisely and to react dynamically if their objects are given to a function.

AnalysisDataSet

The class *AnalysisDataSet* was created to serve as container for the given input data. More precisely, it should contain the superset of the count and sample data.
For that, it inherits the *RangedSummarizedExperiment* class, whose superclass *SummarizedExperiment* was made for this kind of purpose (class diagram shown in figure 6). The developers of the package DESeq2 followed already a similar way by creating the *DESeqDataSet* class.
RangedSummarizedExperiment is more specialized in storing genomic coordinates and was chosen because of a name discrepancy. The new *SummarizedExperiment* class of the SummarizedExperiment package currently is named *SummarizedExperiment0*, but will renamed as soon as all existing relations to the old *SummarizedExperiment* of the

GenomicRanges package are cut.
SummarizedExerpimento uses the virtual *Vector* class from the S4Vector package as parent class, which has an extended semantic compared to the origin vector of R. Also from this package were used the *DataFrame* and *SimpleList* classes as data type for the slots *colData* and *assays*. [53]
The AnalysisDataSet class was modified to the needs of the pipeline. Multiple validation checks had been implemented additionally, to erase possible errors due incorrect data input. For the sample data a DataFrame or data.frame is expected, having samples per row and sample ID's as row names. On the other hand the corresponding count data has to be an integer matrix, with samples per columns and the sample ID's as header (conform to the preparation step, see figure Fehler: Referenz nicht gefunden).

Validation checks for the count matrix:

1. Count data is mandatory by object creation.
2. The count data has to be supplied in an integer matrix. Some used packages do generally not support floating point numbers (ambiguous gene counts). This restriction should keep this into the users awareness.
3. Count data matrices with negative numbers are not allowed.
4. Count data matrices with NA values are not allowed.
5. Column names for the count data matrix are mandatory.

Validation checks if sample data was supplied:

1. Row number of sample data has to be equal to column number of the count data matrix.
2. The row names of the sample data have to match the column names of the count data matrix.

Furthermore two getter and setter methods had been attached to the class. A *countData* method, which allows to set and get the attached count data matrix and likewise a *sampleData* method that allows to easily access and change the DataFrame object of the sample data.
The constructor of the class could be used with an object which inherits from the *SummarizedExperiment* class, or by using the *AnalysisDataSetFromMatrix* function.
Recommended for the creation is the function *createAnalysisDataSet*, which is the first stage in the R pipeline and will be explained in the next section.

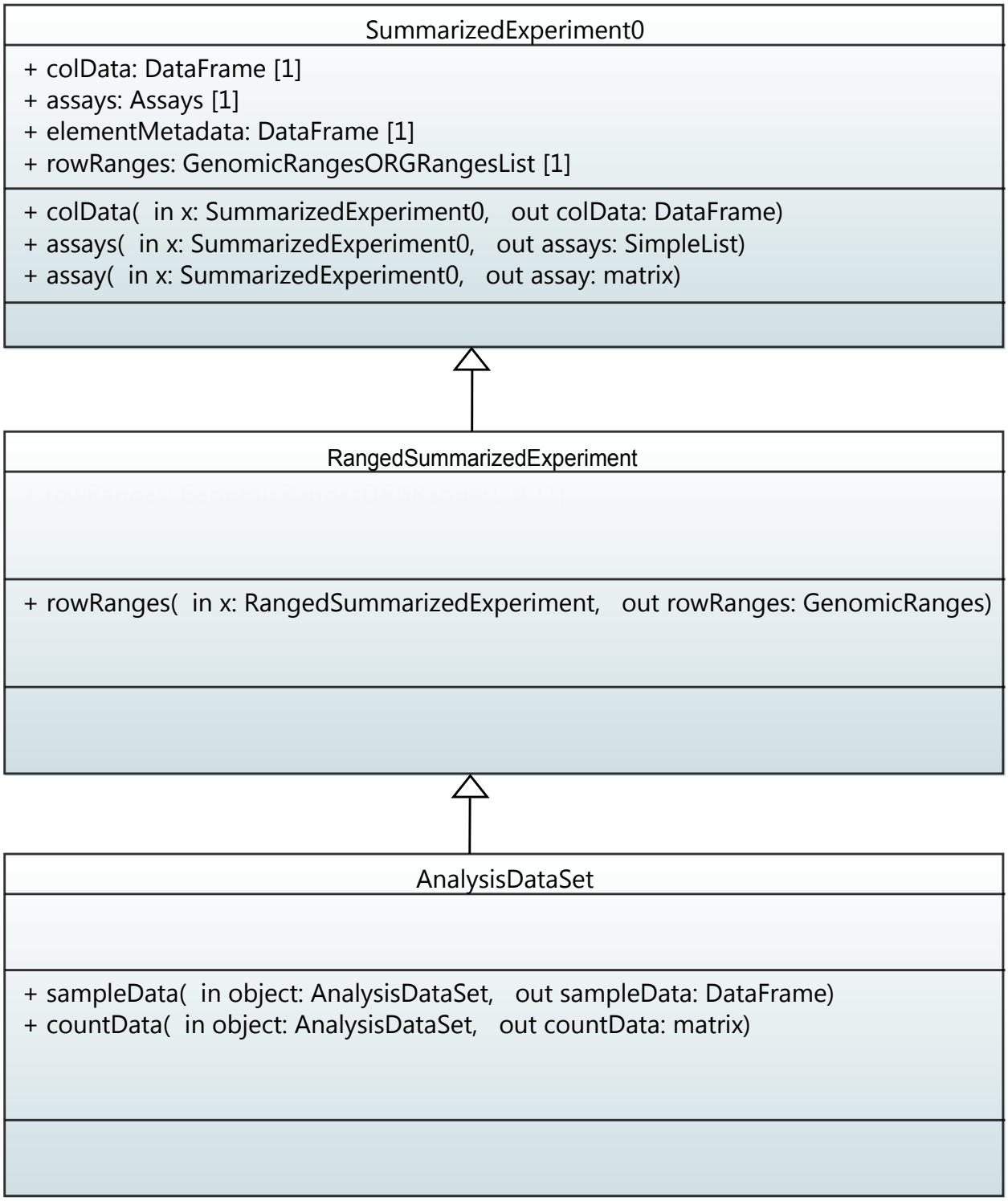

Figure 6: This UML class diagram shows the inheritance between SummarizedExperiment0 and the AnalysisDataSet. Only the most relevant methods and slots are listed. Furthermore the inheritance between SummarizedExperiment0 and the Vector class is omitted.

AnalysisInfo

After the superset was successfully transformed and saved with an AnalysisDataSet object,

it is now necessary to create a subset from this data. This subset has to fit to the hypothesis that the user wants to show with the analysis (more in chapter 4.3.1, Selection and Filtering). For this important step, also a new class was generated. The class *AnalysisInfo*, consists out of the following slots:

1. time: A character vector which could include the timestamp of the object creation.
2. name: Also a character vector, which makes it possible to store a name for the analysis.
3. formula: The formula slot has to contain the formula of the analysis. A formula represents the variables and their correlation that will be used with the statistical model.
4. model: In the model slot, the model matrix should be stored, that could be constructed out of the formula and the sample data.
5. group: Groups could also be created by the concatenation of all rows in the sample data of the environment. This could be used for the quality assurance.
6. contrast: The contrast has to be a list with numeric vectors. A contrast is defining the hypothesis, by using the model matrix.
7. envir: The envir slot contains an AnalysisDataSet object, which contains only the information relevant for this analysis.

Like in AnalysisDataSet, the validation is checked by creation and data assignment. This guarantees, among other things, that the contrast vector fits to the model matrix which on the other hand has to fit to the given formula and whereas finally has to match to the attached sample data.
If missing, the constructor automatically generates the group out of the sample data from the *AnalysisDataSet* object which is stored in the slot *envir*. Also the design matrix is generated by using the *model.matrix* function, which is available through the standard packages of R. Additionally a missing time would be set by the *date* function and a *name* build out of the formula and the time.
It is also important to note that the levels of the factors in the attached sample data are checked against existence and cleaned if necessary. Old levels could be remained by previous filtering steps, since factor levels are not updated automatically by R.
The object could be created directly or by using the *createAnalysisInfo* function.

QualityAssuranceResult

To store the results of the quality assurance part of the pipeline, the class *QualityAssuranceResults* was created. This container has the two slots *method* and *result*. The *method* slot could store a character vector, which should contain a string representation of the used normalization method. The *result* slot on the other hand should store the outcome of this method in form of a count matrix.
The object could be created manually or by using the function *runQualityAssurance*.

5.3.3 AnalysisResult

The class *AnalysisResult* could be used as container for the results of the DEA pipeline. It consist out of the four following slots:

1. name: A character vector which has to contain the identifier of the used package.
2. prioriObjects: Has to be a list that should contain all necessary information for the *plotPrioriObjects* function.
3. *posterioriObjects*: This slot should contain a list with a further list, which includes all necessary objects for the *plotPosterioriObjects* function per result/contrast.
4. *result*: The result slot should contain a list with all normalized results.

The validation check of this class ensures that every element in each list has a corresponding name. Further, if the list at the slot *posterioriObjects* has entries, the length must be equal to the list attached at the *result* slot.
Like in the other classes, all slots could be accessed through defined methods with an equal name.
Typically the object should be generated by the function *runDifferentialExpressionAnalysis*.

5.3.4 Functions

This section describes implementation of the most important functions of the DEA pipeline. They should support the user to create objects, tables and plots through a high amount of automatism.

createAnalysisDataSet

The function *createAnalysisDataSet* was implemented to support the user by the creation of the *AnalysisDataSet* object. For that, the path to the sample and count data file and several file customizations has to be given by the user.
If installed, the faster *fread* function of the *data.table* package is used, to read the count data file, otherwise the *read.table* function takes place from the default package utils.
The column names are checked for uniqueness, in case they are ambiguous, the user would be warned and only the first occurrence of the concerning names would be remained. Row names are also checked against duplicates by using the function *anyDuplicated*. But here, an error would be thrown as soon a duplicate is detected. After coercing the data.frame object to a matrix, the data type is checked and the function would be stopped if it is not numeric (means it is also not integer).
For this function, the sample data file is also mandatory. The user can enable the function *detectDataType* (explained in the next section) by a Boolean value. Otherwise the file is just read via *read.table*. The ID's are checked against duplicates, by using the parameter *keyColSampleData*, which should contain the corresponding column name or number.

Here ambiguous ID's would lead to an error, because whereas duplicate counts per sample could be explained through multiple sequencing processes of a tissue, phenotypic duplicates are uncommon and should be avoided. Finally the values are attached as row names to the *data.frame*.
Both objects are now ready for the alignment. Thereby a Boolean map is created by comparing the column names of the count matrix against the row names of the sample object. Losses on both sides will throw warnings, which contain the concerned ID's. This measure should support the user by detecting and eliminating faulty data rows.
At the end, the matrix is coerced to integer, if necessary. A warning is thrown and the fractional digits are cut off by changing the storage mode.

detectDataType

The function *read.table* in R already detects multiple data types by using the function *type.convert* internal. The first data type which could store all not missing values (unequal NA) will be chosen. How explained in chapter 4.3.1 in section Data Type Detection, the problem to solve is more to detect NA values. These detected values could be given to *read.table* via *na.strings* parameter. Of course, detected data types could also help to speed up the next *read.table* call by supporting the classes with the *colClasses* parameter. [33]

At the beginning of the function, the table is read commonly by using the *read.table* function. A vector with the name *colClasses* is created which should store the detected classes. Additionally a vector *naStrings* is created, if not already given by the user. This vector should store all detected NA strings.
A loop iterates over all columns of the *data.frame* object. Now the datatype of the column is checked:

- Numeric: The column seems to have no NA values, or they have been detected correctly already. The column class will be stored in *colClasses* and the iteration proceeds.
- Integer: Here the procedure is similar like it was by the numeric class. Integer columns should be coerced to numeric by the next read and to do so, "numeric" is stored into the *colClasses* vector. This reduces the chance of problems, caused by differing data types.
- Factor: A factor could also be a wrongly detected numeric with NA strings. Furthermore date formats are not recognized by the *read.table* function and it could also be a date.

The factors need to be checked more in detail. Because the amount of dates is properly lower than the amount of numerics, the former are checked first.
For this approach, the regular expression "^[-+]?([0-9]*\\.[0-9]+$|[0-9]+$)" is used against each factor level (see line 3 to 5 in listing 5.2). The proportion of non numeric values is calculated by subtracting the result of the length of the vector with values that do not match the expression divided through the row count of the *data.frame* from 1 (see line 7 to 9).

```
1  . . .
2  # Assign row numbers of detected non-numeric values to a vector.
3  v_matches = grep("^[-+]?([0-9]*\\.[0-9]+$|[0-9]+$)",
4      invert = TRUE,
5      levels(dfrm_file[, i])[dfrm_file[, i]])
6  # Count of matches
7  n_match_len = length(v_matches)
8  . . .
9  prop_match = 1.0 - (n_match_len / n_row)
10 . . .
```

Listing 5.2: Detection of non-numeric values out of the detectDataType function.

The resulting value *prop_match* has to be higher or equal the mentioned upper bound threshold which could be given by the user. If this is the case, "numeric" is added to the *colClasses* vector, the detected strings are added to the *naStrings* vector and the loop continues with the next column.
Otherwise the values have to be greater equal to the set lower bound threshold, or will checked against the date data type.
In the former case, the uniqueness of the detected non-numeric values is tested. Therefore the unique function is used on the factor level indices (see listing 5.3, line 3). This approach is faster as it would be by using the function directly on the factor (integer vector instead of character). Now the proportion could be calculated by dividing the count of ambiguous values through the length of the origin factor (line 10 in the listing below).
The result is compared against the *naTolerance* value, that could be also given by the user. If it is greater or equal to the parameter, the "numeric" is added to the *colClasses* vector and the iteration proceeds.

```
1  . . .
2  # Save vector with unique indices from detected factor levels
3  uniqueFactors = unique(c(dfrm_file[v_matches, i]))
4  # Save caught unique strings temporary
5  tmp_v_naStrings = levels(dfrm_file[v_matches, i])[uniqueFactors]
6  # Store amount of unique factor levels
7  n_naString_len = length(tmp_v_naStrings)
8  . . .
9  # Proportion of unique values (probably NA values) to matches
10 n_prop_unique = (n_match_len - n_naString_len) / n_match_len
11 . . .
```

Listing 5.3: Estimating the unambiguity of non-numeric values in the detectDataType function.

If, at this point, the class is still not detected, the column is tested against a regular expression that represents a date. The several steps are similar to the steps already

explained above and will therefore be omitted.
Finally, if a date was detected, "forceDate" is added to the *colClasses* list, otherwise "reduceFactor". Both are own classes that have special conversion methods. They are made for the *detectDataType* function only and will not be exported by the package.
The *reduceFactor* class replaces NA values by the usage of the *naStrings* vector. This step is necessary, because the *read.table* expects no whitespaces in *na.strings* and trims these automatically in case of numeric columns, but not in case of character columns.
The *forceDate* class is necessary, because the *as.date* function does not recognize date values automatically. To match the user given format, the values have to be coerced correctly.

```
1  . . .
2  # Coercion method for character to reduceFactor
3  setAs(from = "character", to = "reduceFactor",
4      function(from) {
5        # Boolean map with found NA strings
6        bln_isNA = from %in% naStrings
7        if (any(bln_isNA)) {
8          from[bln_isNA] = NA
9        }
10       as.factor(from)
11     }, where = environment())
12 # Coercion method for character to forceDate
13 setAs("character", "forceDate", function(from) {
14       as.Date(from, dateFormat)
15     }, where = environment())
16 . . .
```

Listing 5.4: Coercion by the usage of own classes in the detectDataType function.

At the end, the new *data.frame* object is returned by the *detectDataType* function.

createAnalyisInfo

The function *createAnalysisInfo* was made to support the user by selecting and filtering the data stored in the *AnalysisDataSet* object. The resulting *AnalysisInfo* object is a container, which includes all relevant information for every further analysis step.
At the very beginning of the function, the variables of the parameter formula, which should include the formula object, are extracted by using the standard function *all.vars* and stored in a vector.
If the length of this vector is greater zero, the variables are tested for existence in the given sample data of the *AnalysisDataSet* object. If the test fails, the function is stopped with an error. Otherwise the variables are used to select the column of the new *data.frame* (see listing 5.5, line 11). No further selection is needed and a Boolean variable *skipSelection* is set to TRUE. Furthermore, if given by the user, the reference value of the factor according to the last variable is set (line 14 to 17).

If otherwise the length of the vector was zero (how it would be for example by the formula ~ 1), the *skipSelection* variable is set to FALSE. In the case of a given parameter *selection*, which should be a character vector, the parameter is used to create the subset of the sample data.

```
1  . . .
2  # Assign sample data
3  sampleData = sampleData(analysisDataSet)
4  # Extract variables from formula
5  vars = all.vars(formula)
6  cntVars = length(vars)
7  # Check if formula with dependencies is given
8  if (cntVars > 0) {
9    . . .
10   # Create subset including only necessary colData
11   sampleData = sampleData[, vars, drop = FALSE]
12   # Automatic selection already done
13   skipSelection = TRUE
14   if (!missing(referenceValue) && !is.null(referenceValue)) {
15     # Set up reference value for last value in formula
16     sampleData[, vars[cntVars]] =
17       relevel(sampleData[, vars[cntVars]], ref = referenceValue)
18 . . .
```

Listing 5.5: Selection of the sample data in the function createAnalysisInfo.

Now the data is ready for filtering. For this purpose, the user could supply the parameter *condition*. This character vector has to use the syntax of R for the logical conjunctions. All columns will be fully qualified automatically by using a regular expression (see listing 5.6, line 3 to 10). After that, the resulting string is parsed and evaluated by using the R functions *parse* and *eval* (shown in line 13). The outcome is a Boolean vector, that indicates the positions of the filtered rows by TRUE values. In that case the given condition creates NA values, a warning is thrown which includes information about the concerned rows. This vector is then used for the selection (shown in line 19). The underlying *SummarizedExperiment* method assures, that the count matrix and the sample data is aligned respectively.

```
1  . . .
2  # Prefix and suffix for full qualification of column names
3  prefix = "sampleData[, "
4  suffix = "]"
5  # Build disjunction of possible column names for the regular
   ↪  expression
6  possibleColnames = paste0(colnamesSample, collapse = "|")
7  # Build regular expression
8  regex = paste0("(", possibleColnames, ")(\\s+(!=|==)\\s+)")
9  # Build fully qualified condition
10 condition = gsub(regex, paste0(prefix, "\"\\1\"", suffix, "\\2"),
   ↪  condition)
11 . . .
12 # Evaluate expression, inform user with a more specific error
13 eva = tryCatch(eval(parse(text = condition)),
14         error = function(e) {
15           stop("Error while parsing condition:\n", e)
16         })
17 . . .
18 # Create filtered subset
19 analysisDataSet = analysisDataSet[, !blnIsNa & eva, drop = FALSE]
20 . . .
```

Listing 5.6: The filtering of sample data in the function createAnalysisInfo.

The *createAnalysisInfo* function also allows to generate a pseudo random group of a given size out the resulting sample and count data. For this, the *sample* function of R is used on all possible groups.

runQualityAssurance

The function *runQualityAssurance* serves possibilities to normalize counts for quality assurance purposes (see also chapter 4.3.3 for the conception).
The user could choose between the already introduced methods vst, rlog, log2 and raw (which is the origin matrix). The choices could be made with a character vector which is checked for correctness by the R function *match.arg*. As default, all methods are selected.
To retain the full functionality of the foreign package functions that are in charge of the normalization, the user could supply the argument *addArguments*. This parameter has to be a list which contains the method identifier as name together with an *alist* object that includes the customizations. The *alist* is similar to a list, but does not evaluate its content ([33] p. 268).
The given *alist* will then be merged in another *alist* with the default parameters of this method (shown in figure 5.11, in line 3 to 9). The new list could now be used with the *do.call* function of R, which calls the corresponding function (shown in line 11 to 12).
Finally a *QualityAssuranceResult* object is created and inserted with the name of the used method into a vector (shown in line 14 to 16).

```
. . .
# Assign given arguments or NULL
addArgs = addArguments[["vst"]]
# Save standard arguments
args = alist(object = countData,
    blind = TRUE,
    ...)
# Add additional arguments
args = c(args, addArgs)
# Use variance stabilizing transformation from DESeq2 package
vstCountData = do.call(DESeq2::varianceStabilizingTransformation,
    args)
# Create object and add it to the vector
cResultsQA = c(cResultsQA,
    "vst" = QualityAssuranceResult(method = "vst",
        result = vstCountData))
. . .
```

Listing 5.7: Usage of user given additional arguments for foreign functions in runQualityAssurance.

plotQualityAssurance

The generated vector with the *QualityAssuranceResult* object from function *runQualityAssurance* could directly be used with the function *plotQualityAssurance* (the concept was described in chapter 4.3.3).
By the implementation the following plots were included:

Table 12: Available plots of the function plotQualityAssurance.

Identifier	Function (Package)	Description
sd	meanSdPlot (vsn)	Shows the standard deviation over rows (samples) versus the row means [54].
mds	plotMDS (limma)	MDS is the abbreviation for multidimensional scaling plot. The scatter plot shows approximated log2 fold changes between samples through distances in two dimensions [47].
pca	plotPCA (DESeq2)	Uses the *rowVars* function of the package genefilter to sort genes descending to their highest variance between samples. A given number (default: up to 500) is then used for a principal component analysis accomplished through the stats function *prcomp*. The result is furthermore used to generate a scatter plot, that indicates the correlations between sample groups. The function is originally from the package DESeq2, but had been exported and modified for the usage with standard

		matrices.
hmExpDesc	pheatmap (pheatmap)	Uses *rowMeans* function and selects the top hundred genes in descending order for the input to *pheatmap*. Could be used to detect sample outliers, which might have a big negative influence through the high expression on the analysis results.
hmExpAsc	pheatmap (pheatmap)	Similar to the description above, but here the genes will be sorted ascending order.
hmDist	pheatmap (pheatmap)	Uses the *dist* function on the transposed count data (equal to over rows) and uses the result as input for the pheatmap function and its *clustering_distance_rows* and *clustering_distance_cols* parameters. Could therefore also be used to detect groups and check the data quality.

It was also made possible for the user, to give additional arguments to several functions by using the *addArguments* parameter.

plotAnalysisInfo

The function *plotAnalysisInfo* should inform the user about the created analysis environment and includes currently three possible plots:

Table 13: Available plots of the function plotAnalysisInfo.

Identifier	Function (Package)	Description
na	plotNA	Generates a bar plot with information about the amount of NA values per column of the sample data (makes no sense if a filter has been used previously).
group	plotBar2dFacet	Shows the group counts by generating a fully customizable bar plot.
libsize	plotLibSizeDist	Generates a fully customizable bar plot, which shows the library sizes in millions. Could also be used to detect outliers.

Extra arguments for customizations could be supplied by the parameter *addArguments*.

plotBar2dFacet

The function offers an easy to use wrapper for the ggplot2 function *ggplot* to generate bar plots. It includes the *facet_wrap* function, which could be used to split the plots by given groups. An input given by the user has to be the parameter *data*, which should be a data.frame or a similar object and the parameter *x*, which should contain the name of the categorical variable of interest.
At first instance, NA values would be transformed into the string “NA”, because otherwise they would not be shown in the legend. Furthermore the object given by the variable data is

coerced into a data.frame, for further usage.
Now a ggplot object could be generated by using the *data* variable and evaluating the given *x* character vector in its context. Then the data for the bars is computed and added through the usage of the *geom_bar* function.
The function *plotBar2dFacet* also allows many customizations, as for example three different ways to color the bars.
If the Boolean parameter *simpleColor* has been set to TRUE by the user, ggplot's standard colorization would be applied to the bars. If the *ownColor* attribute was given by the user, the *scale_fill_manual* option of ggplot2 is used with the supplied colors. Additionally it is possible to use the diverging color palette of the RColorBrewer package, by using the *scale_fill_brewer* package. Of course the bars could be produced without colors too.
Further design choices could be made by using the *theme* parameter. The package ggplot2 is offering 8 different preconfigured themes that are able to be attached by the function. Themes could change the appearance of the data independent parts of the plot (see [55]).
Another feature was implemented, that rotates the labels of the x-axis at a certain length and additionally wraps them into two sentences (by concatenation with "\n"), if they have a length above 19. This should preserve the readability of the plot, even if uncommon long x labels exist.
The generated plot could by exported by using an R graphic device (default is *svg*), or just shown at the provided plot window.

plotBoxplot

This function could be used similar to the *plotBar2dFacet* function for manual plot generation. In this case a box plot could be generated out of a numeric value (y-axis) and a categorical value (x-axis).
Like in *plotBar2dFacet*, the ggplot package was used by the implementation. Through that the taken steps were almost the same, with the difference that here *geom_boxplot* was used instead of the geom_bar function.

filterCountData

Like already mentioned in the concept chapter 4.3.2, it is recommended to filter the count matrix before a DEA. This could be approached by using the function *filterCountData*.
The function requires an *AnalysisDataSet* object or a numeric count matrix as input. Different methods could, like in the other functions, be selected by a parameter, but currently only the so called "cpm" method had been implemented.
For the calculation, at first the CPM's are computed over every column by using the *cpm.default* method of the package edgeR.
The results are then compared against the given parameter *limColumn* by the logical operator greater than.
Now the logical result is given to the *rowSums* function, which finally is compared against *limColumn* by using the logical operator greater equal (line 9 in the listing below).
Finally the Boolean vector could be used to create a subset which contains the filtered genes only (line 11).
The similar method is also described in the edgeR manual (see [41], p. 11).

```
1  filterCountData = function(x,
2      method = c("cpm"),
3      limColumn,
4      limRow,
5      verbose = 2,
6      ...) {
7  . . .
8    # Build boolean white list indicating genes over the given
     ↪  thresholds
9    blnGenesOk = (rowSums(edgeR::cpm.default(x = countData(x)) >
     ↪  limColumn) >= limRow)
10   # Use SummarizedExperiment method to drop unwanted rows
11   x = x[blnGenesOk, , drop = FALSE]
12 . . .
```

Listing 5.8: Method cpm in the function filterCountData.

runDifferentialExpressionAnalysis

The function *runDifferentialExpressionAnalysis* takes control over the whole DEA process. Like described in chapter 4.3.4, the source code of this function would not be maintainable without encapsulating the three packages first. This had been achieved through the creation of the script files with the name myEdgeR, myDESeq2 and myLimma. The consistent structure of these wrapper script files is shown exemplary in myDESeq2 at the next section.
Like shown in figure 5, the DEA function could be exited in two different stages. The first stage has been named priori, because on this point no hypothesis tests have been made yet. The last stage on the other hand, has been named posteriori.
The DEA function expects an *AnalysisInfo* object whose count data is already filtered by using the function *filterCountData*.
The attribute *listAnalysisResult* could be used to resume the function at the posteriori stage. Therefore the necessary objects for each package would be extracted and all previous steps skipped by using an if clause.
The parameter *isPrioriOnly* could be set to TRUE by the user and indicates that only the dispersions should be estimated. If it was set to FALSE (what is the default) and no or an empty contrast list was supplied, the internal function "*.createStandardContrastList*" would be called.
This function creates a common contrast that compares the last column in the given model matrix versus the first column. Thereby, it also considers if an intercept was given or not (shown in line 2 in the figure below). In the former case the vector would have a leading zero (line 14), in the latter one a leading minus one (line 7). The last digit would be in both cases a one. All remaining digits would be set to zero trough the *rep* function of R.

```
1  .createStandardContrastList = function(formula, model) {
2    if (attr(terms.formula(formula), "intercept") == 0) {
3      # Last column vs first column
4      if(ncol(model) < 2) {
5        contrast = c(1)
6      } else {
7        contrast = c(-1, rep(0, times = ncol(model) - 2), 1)
8      }
9    } else {
10     # Last column vs intercept
11     if(ncol(model) < 2) {
12       contrast = c(1)
13     } else {
14       contrast = c(0, rep(0, times = ncol(model) - 2), 1)
15     }
16   }
17   return(list(contrast))
18 }
```

Listing 5.9: Creation of the standard contrast list by the function ".createStandardContrastList".

Similar like in the *runQualityAssurance* function, the user could supply additional arguments by using the *addArguments* parameter. But because of the high amount of possibilities, the list here is more complex. For example if the user wants to change the method of the *estimateDispersions* function from DESeq2 to "parametric", the given list has to be build in the following way: addArguments = list("DESeq2" = alist("estimateDispersions" = alist("method" = "parametric"))). Nevertheless, this approach ensures that the parameters reach the correct destination.
For every package given by the user with the *packages* parameter would the following functions now be called with the do.call function and the standard parameter together with the fitting additional parameters:

1. init<Package> (without addArguments)
2. estimateSizeFactors<Package>
3. estimateDispersions<Package> (shown in the figure below, at line 2 to 14)

The several results of the estimate dispersions would be stored now in a list at the *priorIObjects* slot of an *AnalysisResult* object with the name of the corresponding package (line 16 to 17).

```
# Save standard arguments
args = alist(dseqDataSet = dseqDataSet,
    verbose = verbose,
    ...
)
# Check if DESeq2 has got additional arguments
if (!is.null(addArgs)) {
  addArgsExtra = addArgs[["estimateDispersions"]]
  # Add additional arguments
  args = c(args, addArgsExtra)
}
# Estimate Dispersion
#####
dseqDataSet = do.call(estimateDispersionsDESeq2, args)
# Add dispersion result to AnalysisResult object
aResDESeq = AnalysisResult(name = "DESeq2",
    prioriObjects = list("estimateDispersions" = dseqDataSet))
```

Listing 5.10: Attachment of user given arguments, function call and creation of an AnalysisResult object with prioriObjects.

If the *prioriObjects* of all given packages are stored and *isPrioriOnly* had been set to TRUE, a list with the package names and the corresponding *AnalysisResult* objects would now be returned.
If it otherwise had been set to FALSE, the following functions would be called:

1. fitModelTest<Package>
2. decideTests<Package>
3. normalizeResults<Package>

The functions *decideTests* and *normalizeResults* need a contrast as argument and are therefore called within a loop that iterates over the supplied contrast list. The returned result objects are then saved in a separate list, by using the current position of the iteration as index.
Finally, if the loop was exited, both lists are assigned to the *AnalysisResult* object of the respective package. Additionally all *AnalysisResults* together with the package identifiers are stored into a list, which is returned at the end of the function.

myDESeq2.R

Because it would go beyond the constraints of the thesis to show the content of every wrapper script file, the DESeq2 script was chosen to be explained exemplary for all three files.
Following functions of the wrapper file myDESeq2 are used by the DEA function:

1. *initDESeq2*: This function could be used to generate all package dependent necessary

objects for the first stage of the DEA. In this case it wraps the DESeq2 function *DESeqDataSetFromMatrix* and uses instead an AnalysisInfo object as input.

```
1  initDESeq2 = function(analysisInfo, verbose = 2, ...) {
2  . . .
3    # Create DESeqDataSet
4    dseqDataSet = DESeq2::DESeqDataSetFromMatrix(countData = countData,
5        colData = sampleData,
6        design = formula,
7        ...)
8  . . .
```

Listing 5.11: Instantiation of the DESeq2 object in the initDESeq2 wrapper function.

2. *estimateSizeFactorsDESeq2*: The function generates the second possible package depending object in *runDifferentialExpressionAnalysis*. It should contain all necessary normalization steps that have to be applied on the given count matrix. In this case the *estimateSizeFactors* function of DESeq2 is used. The user could change the *type* attribute of the function through the parameter method (line 2 and 7).

```
1  estimateSizeFactorsDESeq2 = function(dseqDataSet,
2      method = c("ratio", "iterate"),
3      verbose = 2,
4      ...) {
5  . . .
6    # Attach estimated size factors
7    dseqDataSet = DESeq2::estimateSizeFactors(object = dseqDataSet,
8        type = method,
9        ...)
10 . . .
```

Listing 5.12: Attachment of the estimated size factors to the DESeq2 object in the estimateSizeFactorsDESeq2 wrapper function.

3. *estimateDispersonsDESeq2*: How already mentioned in the concept chapter, the packages also need a wrapper for the calculation of the dispersion. The outcome could also give first important information about the data quality and could therefore be used with the plotPrioriObjects function. The fitType of the estimateDispersion function from DESeq2 was also unified and could be changed by using the parameter method (line 2 and 7).

```
estimateDispersionsDESeq2 = function(dseqDataSet,
    method = c("parametric", "local", "mean"),
    verbose = 2,
    ...) {
. . .
  dseqDataSet = DESeq2::estimateDispersions(object = dseqDataSet,
      fitType = method,
      quiet = verbose < 1,
      ...)
. . .
```

Listing 5.13: Attachment of the dispersion estimations to the DESeq2 object in the estimateDispersionsDESeq2 wrapper function.

4. *fitModelTestDESeq2*: After the size factors and the dispersion were estimated successfully, the statistical model could be applied. Furthermore the coefficients could be tested for significance. As already mentioned, DESeq2 offers two different functions for the tests. The user could switch between the *nbinomWaldTest* and the *nbinomLRT* function by using the parameter method (line 10 to 13 and 16 to 19). The parameter *betaPrior* was set to FALSE by default, because the function is not supported in case of user supplied model matrices (see [38], p. 39). The further parameter *minReplicatesForReplace*, indicates the minimum threshold of samples a group must have, to enable outlier replacement (see [38], p. 36). This functionality is under normal circumstances only available by using the *DESeq* main function, but it was possible to adapt and reconstruct all necessary procedures.

```
fitModelTestDESeq2 = function(dseqDataSet,
    method = c("Wald", "LRT"),
    reducedFormula,
    betaPrior = FALSE,
    minReplicatesForReplace = 7L,
    verbose = 2,
    ...) {
. . .
  if (method == "Wald") {
    # perform Wald tests
    dseqDataSet = DESeq2::nbinomWaldTest(object = dseqDataSet,
        betaPrior = betaPrior,
        quiet = verbose < 1,
        ...)
  else if (method == "LRT") {
    # perform likelihood ratio tests
    dseqDataSet = DESeq2::nbinomLRT(object = dseqDataSet,
        betaPrior = betaPrior,
        quiet = verbose < 1,
        ...)
  }
. . .
```

Listing 5.14: Fitting of the model and testing of the coefficients in the fitModelTestDESeq2 wrapper function.

5. *decideTestsDESeq2*: The last package depending calculations are made in the function decideTestsDESeq2. In this case the *results* function of the package DESeq2 is called, which returns the result for the given *contrast* optimized for the FDR threshold *alpha* by using the correction given by the parameter method.

```
decideTestsDESeq2 = function(dseqDataSet,
    contrast,
    method = "BH",
    alpha = 0.05,
    verbose = 1,
    ...) {
. . .
  res = DESeq2::results(object = dseqDataSet,
      contrast = contrast,
      alpha = alpha,
      pAdjustMethod = method,
      ...)
  . . .
```

Listing 5.15: Calculation and constellation of DEA results in the decideTestsDESeq2 wrapper function.

6. *normalizeResultsDESeq2*: Finally the result object has to be normalized and transformed into a package independent *data.frame* object. Furthermore the *alpha* parameter is used to create a subset which contains only genes with a probability below the given value (line 10). Now the genes could be ordered by the given parameter *resSortBy* in the order which is given through *resSortDesc* (line 13 to 16).

```
normalizeResultsDESeq2 = function(result,
    alpha,
    resSortBy = c("padj"),
    resSortDesc = FALSE,
    verbose = 1,
    ...) {
. . .
  if (!missing(alpha)) {
    # Create subset with adjusted p-value < alpha
    result = SummarizedExperiment::subset(result, padj < alpha)
  }
  # Sort result
  orderResult = order(result[, resSortBy],
      decreasing = resSortDesc,
      na.last = TRUE)
  result = result[orderResult, , drop = FALSE]
  # Coerce result to a data.frame object
  result = as.data.frame(result)
. . .
```

Listing 5.16: Normalize results and create ordered subset in the normalizeResultsDESeq2 function.

Besides the wrapper functions that are made mainly for the *runDifferentialExpression* function, the script files contain also the necessary functions to plot parts of its results, the priori objects and posteriori objects. These functions are called by the functions *plotPrioriObjects* and *plotPosterioriObjects*, but could also be called manually.

1. *plotPrioriObjects*: This function could generally be used to generate a dispersion estimation plot (line 12 in the figure below). In this case the fitted mean-dispersion correlation is shown together with the per-gene dispersion estimates ([39], p. 32).

```
1  plotPrioriDESeq2 = function(lPrioriObj,
2      path = paste0(".", .Platform$file.sep),
3      export = TRUE,
4      file_extension = ".svg",
5      export_function = svg,
6      verbose = 2) {
7  . . .
8  # Assign result of estimateDispersionsDESeq2
9  deseq2Disp = lPrioriObj[["estimateDispersions"]]
10 # Plot dispersion estimates
11 DESeq2::plotDispEsts(deseq2Disp)
12 . . .
```

Listing 5.17: Generate dispersion plot in plotPrioriDESeq2 function.

2. plotPosterioriObjects: Handles analog to the *plotPrioriFunction*, but here the objects which have been saved at the posteriori stage are used. Here the *plotMA* function of DESeq2 is used to plot the log2 fold changes and their correlation to the mean of the normalized counts ([39], p. 33).

```
1  plotPosterioriDESeq2 = function(lPostObj,
2      result,
3      path = paste0(".", .Platform$file.sep),
4      export = TRUE,
5      file_extension = ".svg",
6      export_function = svg,
7      verbose = 2) {
8  . . .
9  # Assign result of decideTestsDESeq2
10 deseq2test = lPostObj[["decideTests"]]
11 # Plot log2 fold change vs mean of normalized counts
12 DESeq2::plotMA(deseq2test, main = "MA-Plot")
13 . . .
```

Listing 5.18: Generate log2 fold change plot in plotPosterioriDESeq2 function.

plotPrioriObjects

The function *plotPrioriObjects* could be used to plot the results from the priori stage of the function *runDifferentialExpressionAnalysis*.
The function expects a list with package identifiers and their corresponding *AnalysisResult* objects as input. Furthermore the regular arguments of other functions with plot tasks could be used.
The names in the list are checked for matches against the package names. If an entry has been found, the respective plotPriori function is called and the plot generated.

plotPosterioriObjects

This function was made to generate plots for the objects that are generated at the posteriori stage of the DEA function. The functional principle is similar to *plotPrioriObjects*, with the difference that every package has so many lists as contrasts were used at the analysis.
Through that, an additionally iteration step is necessary, which calls the respective *posterioriPlot* function.

saveResults

It was also necessary to build a function that could store the result tables in the list of the result slot of each *AnalysisResult* object. This was done by the implementation of the function *saveResults*, which relies on the same procedures explained in *plotPosterioriObjects*. The only difference is, that *saveResults* has to work with package independent *data.frame* objects and could therefore use one single R function *write.table* for all packages.

joinResults

According to section 4.3.5, it was also necessary to implement a function that could join the results. For that purpose the function *joinResults* was created.
The function expects the resulting list of *runDifferentialExpressionAnalysis* as input. Additionally, the user could define how the tables should be joined through the *joinMethod* argument. In the case of an outer join, all rows of every package will be exported. Missing values will be filled with the given *tableNA* parameter. Otherwise, if an inner join was chosen, only genes that were detected across all used packages would be exported. As default, both methods will be used.
At the very beginning of the function, an *apply* loop creates a list by iterating over the contrast count of the supplied result list (shown at line 3, in the listing below).
To ensure that the column names could be distinguished between the packages, the package name is concatenated as prefix with a customizable separator (line 12 to 13).
The first table is exported to the variables *innerJoinResult* and *outerJoinResult*, by using the iteration count as index (line 5 and 10).
Now another loop is necessary, that iterates over the used package count, but starting by two (line 17).
The result *data.frame* and name of the second package are now assigned to the variables *result* and *name*. Also the prefix is added to the column names of the object result.

The next step is to join the tables by using the methods selected by the user. This is made with the R function *merge* (line 21 to 26). The merge function moves the row names into a column, so the function has to reassign them, for the next iteration step (line 29 to 30).
After the inner loop over the packages has finished, the joined results of the first given contrast could be exported by the *write.table* function.
Finally the resulting *data.frame* objects are stored in a list which is returned to the apply loop, which could now proceed the iteration with the second contrast (line 34 to 36).
The result is a list of the form "list("inner" = innerJoinResult, "outer" = outerJoinResult)", whereby unused methods will be NULL.

```
. . .
# Iterate through results/contrasts and merge them across packages
lJoinedRes = lapply(seq_len(cntContrast), function(iContrast) {
  # Assign result table of first package
  result = lResult[[iContrast]]
  name = name(analysisRes)
  # Substitution of reference tables (inner join)
  if ("inner" %in% joinMethod) {
    isInnerJoin = TRUE
    innerJoinResult = result
    # Add user defined separator
    colnames(innerJoinResult) = paste0(name, prefixSeparator,
        colnames(innerJoinResult))
  }
  . . .
  # Iterate over remaining packages
  for (i in 2:(cntPackages)) {
    . . .
    if (isInnerJoin) {
      # Merge actual data.frame with next data.frame
      innerJoinResult = merge(x = innerJoinResult,
          y = result,
          all.x = FALSE,
          all.y = FALSE,
          by = "row.names"
      )
      # The function merge adds a column "row.names", we move this
      # information into real row.names again
      row.names(innerJoinResult) = innerJoinResult[, "Row.names"]
      innerJoinResult = innerJoinResult[, -1]
    }
    . . .
    # Attach available result to list
    tmpLjoinedRes = list("inner" = innerJoinResult,
        "outer" = outerJoinResult)
    return(tmpLjoinedRes)
    })
. . .
```

Listing 5.19: Building a list which contains the joined results out of the AnalysisResult objects list in the joinResults function.

plotComparison

The implementation of a Venn diagram is also necessary due the requirements (see section 2.1.3).

For the generation of visualizations regarding to the comparison of results between packages, the function *plotComparison* was implemented. Currently this function can only produce a Venn diagram, by using the *venn.diagram* function of the VennDiagram package.
For each contrast, a list is created, which contains the row names of the result for every used package together with the package name as identifier. This is approached by the usage of the *lapply* function of R.
This list could then be used directly with the function *plotVennDiagram*, which calls the *venn.diagram* function.

SerializeAnalysis

Because of the usage of well defined data structures, it is possible to serialize the outcome of the functions.
This measurement could ensure that several functions like *createAnalysisDataSet* are not executed unnecessarily multiple times. Consequently it saves the time of the user and the resources of the device it runs on.
The function *serializeAnalysis* makes usage of the R function *saveRDS*, which saves the data into a conform format, that could be read by *readRDS* used in *unserializeAnalysis*.
The function allows additionally compression, which is set to gzip by default.

unserializeAnalysis

The function *unserializeAnalysis* is the counterpart to *serializeAnalysis* and works therefore in the similar way. To read the R file, the function *readRDS* is used, which also detects the compression format automatically.
Finally the function returns the reattached object.

Commando-line interfaces

As already mentioned, the initial implementation will contain a commando-line interface, which could be used directly or for example controlled over a web-interface. The aim of the implementation was at first to satisfy the requirements in chapter 2. So not every functionality can be controlled over the interface yet.
Because of the high amount of options, the whole analysis was split into four different CLI's:

1. cli_createAnalyisDataSet.R: Could create and save the *AnalysisDataSet* object and creates optional a plot that shows the NA distribution over all columns.
2. cli_createAnalysisInfo.R: Offers the possibility to create and store the *AnalysisInfo* object by using the *createAnalysisInfo* function. Further the count data could be filtered by *filterCountData* and the *plotAnalysisInfo* function is used to create a selection of plots.
3. cli_createPlot.R: By using this CLI, the user has access to the functions *plotNA*, *plotBar2dFacet* and *plotBoxplot*.
4. cli_runDifferentialExpressionAnalysis.R Finally a DEA could be started with the function *runDifferentialExpressionAnalysis*. Additionally access is offered to the functions: *runQualityAssurance*, *plotQualityAssurance*, *saveResults*, *plotPrioriObjects*,

plotPosterioriObjects, *joinResults*, *plotComparison* and *compareResultsToRef*.

All CLI's are making use of the functions *serializeAnalysis* and *unserializeAnalysis* and have consequently to be executed in the correct order. As helper for the generation of the help text and the assignment of the arguments and options the function *make_option* and *optparse* of the package optparse have been used. The concept of the usage is oriented on the optparse module of Python, which is similar to the newer argparse module [56].

6 Evaluation of the Genetic Differential Expression Pipeline

This chapter describes the evaluation of the pipeline that has been developed in context of this thesis. In this case the requirements in chapter 2 build the foundation for the evaluation.

The test case should be the detection of differential expressed genes between the tumor tissue site distant metastasis and primary tumor. As data source should be used the SKCM data set from TCGA (see chapter 3.3). To hold up the overview in this experiment, both groups should be restricted to twenty pseudo randomly selected samples.
All commands were executed in Windows 7 by using the standard Command Prompt cmd.exe.

6.1 assembleRNAseq2

This section describes the evaluation of the functional requirements defined in section 2.1.1 based on the defined use case at the beginning of this chapter.
Therefore the Python script *assembleRNAseq2* is used with the downloaded and extracted mRNA data of the SKCM samples from the TCGA data portal [21].

Following command was thereby used for the script execution:

```
1  >>> assembleRNAseq2.py "./" "output/raw_counts.tsv"
    ↪  --source_file_col_key gene_id --verbose 3
2  Using filter: (^unc\.edu\..*\.rsem.genes.results)
3  Found 473 specified files in "./FILE_SAMPLE_MAP.txt"...
4  Consistency test succeed...
5  File successfully written!
```

Listing 6.1: Execution of assembleRNAseq2.py.

How shown in the listing 6.1, the following arguments and options have been used to generate the target file.

Table 14: Description of the arguments used by the execution of assembleRNAseq2.py.

Position	Value	Description
1	"./"	The first argument defines the source path. Here the script was executed in the directory of the extracted

		files.
2	“output/raw_counts.tsv”	The second positional argument defines the destination and filename of the generated file. The directory has to be generated previously by the user. Here, the target file “raw_counts.tsv” has been stored in a directory with the name “output”.
3	--source_file_col_key *gene_id*	This optional argument has set “gene_id” as key column for the aggregation. Its content has been also exported together with the column “raw_count” (default parameter) to the target file.
4	--verbose *3*	Through this option, the verbosity of the script was set to level 3. Level 1 would print warnings only.

The extracted 473 files has now been aggregated by using a key column through the script. The exported file has a size of 62.7 megabyte, contains 20532 rows (with header) and 474 columns (with row names) and is fitting to the format which was defined in the requirements.
Consequently the three functional requirements A.F1, A.F2, A.F3 has been fulfilled.

As next step the non-functional requirements will be evaluated (section 2.2).
Here, NF1 is given. A commando-line interface is not the user-friendliest approach, but it fulfills its purpose. Additionally a help has been implemented, that could guide the user through the usage.
The functional requirement NF4 is fulfilled already through the implementation of the CLI. Of course other interfaces could be also implemented.
The further requirements NF2 and NF3 are fulfilled, because of the usage of Python. This encapsulates the script completely from the main application and makes it independent.

6.2 ASSEMBLECLINICAL

Analog to *assembleRNAseq2*, in this section *assembleClinical* will be evaluated against the functional requirements defined in section 2.1.2.
Here the Python script is used with the downloaded and extracted clinical data corresponding to the mRNA data used in the last section [21].

Following command has been used with the CLI:

```
>>> Clinical>assembleClinical.py
 ↪  "./Clinical/Biotab/nationwidechildrens.org_clinical_patient_skcm.txt"
 ↪  "./Clinical/Biotab/nationwidechildrens.org_clinical_follow_up_v2.0_skcm.txt"
 ↪  "./Clinical/Biotab/nationwidechildrens.org_clinical_radiation_skcm.txt"
 ↪  "./Clinical/Biotab/nationwidechildrens.org_ssf_normal_controls_skcm.txt"
 ↪  "./Clinical/Biotab/nationwidechildrens.org_ssf_tumor_samples_skcm.txt"
 ↪  "./output/clinical_data.tsv" --foreign_key bcr_patient_barcode
 ↪  --column_prefix "" "flwup_" "rad_" "ctrl_" "smpl_" --verbose 3
... parsing
 ↪  ./Clinical/Biotab/nationwidechildrens.org_clinical_patient_skcm.txt
... caught 470 lines
... parsing
 ↪  ./Clinical/Biotab/nationwidechildrens.org_clinical_follow_up_v2.0_skcm.txt
... caught 389 lines
... parsing
 ↪  ./Clinical/Biotab/nationwidechildrens.org_clinical_radiation_skcm.txt
... caught 147 lines
... parsing
 ↪  ./Clinical/Biotab/nationwidechildrens.org_ssf_normal_controls_skcm.txt
... caught 472 lines
... parsing
 ↪  ./Clinical/Biotab/nationwidechildrens.org_ssf_tumor_samples_skcm.txt
... caught 473 lines
... writing file
File successfully written!
```

Listing 6.2: Execution of assembleClinical.py.

The following arguments have been used in listing 6.1:

Table 15: Description of the arguments used by the execution of assembleClinical.py.

Position	Value	Description
1-5	“*.txt”	The first five arguments select the source files that should be used for the aggregation.
6	“output/clinical_data.tsv”	Like in assembleRNAseq2, the last positional argument defines the destination and filename of the generated file. The directory has to be generated previously by the user. Here, the target file “clinical_data.tsv” has been stored in a directory with the name “output”.
7	--foreign_key *bcr_patient_barcode*	Through this optional argument, the key column for the aggregation has been set to “bcr_patient_barcode”.
8	*--column_prefix “” “flwup_”*	This arguments allows to set a prefix for every file

	“*rad_*” “*ctrl_*” “*smpl_*”	given in the first positional argument.
9	--verbose *3*	Through this option, the verbosity of the script was set to level 3. Level 1 would print warnings only.

Here the resulting file was aggregated out of 5 different files by using a key column and has finally a size of 980 kilobyte, contains 471 rows (with header) and 187 columns. Its content is also fitting to the format which was defined in the requirements.
Consequently the three functional requirements B.F1, B.F2, B.F3 has been fulfilled.

The evaluation of the non-functional requirements (section 2.2) which have be described in section 6.1 could be also applied on the tool assembleClinial.
Therefore the implementation fulfills again all given requirements.

6.3 GeneticAnalysisPipeline

This section describes the evaluation of the GeneticAnalysisPipeline against the given requirements in section 2.1.3. The two files whose creation has been described in the last sections has been used here as input.
Execution of the first CLI script:

```
>>> Rscript --no-environ cli_createAnalysisDataSet.R --countData "../../data-raw/raw_counts.tsv" --sampleData "../../data-raw/clinical_data.tsv" --keyColSampleData "bcr_patient_barcode" --detectDataType --outputData "../../data-raw/extscript/rdata/analysisData.RData"
Dimensions of supplied count matrix:
Rows: 20531. Columns: 474
Resulting dimensions of count matrix:
Rows: 20531. Columns: 469
Top 5 column names (keys to samples):
TCGA-ER-A2ND TCGA-QB-A6FS TCGA-D3-A2JH TCGA-WE-AAA4 TCGA-ER-A2NG
---
-Resulting changes---
Factor to Numeric: 26 columns.
Factor to Date: 3 columns.
Integer to Numeric: 3 columns.
---
Used NA word(s):
[Not Available]
. . .
Dimensions of supplied sample file:
Rows: 470. Columns: 187
Resulting dimensions of sample file:
Rows: 470. Columns: 186 (column bcr_patient_barcode removed)
Top 5 row names (keys to counts):
TCGA-D3-A1Q4 TCGA-FS-A1ZP TCGA-D9-A4Z6 TCGA-ER-A19F TCGA-D3-A3C3
---
Warnmeldungen:
1: In GeneticAnalysisPipeline::createAnalysisDataSet(fileCountData = countFile,  :
  Removed columns in count data! ID's in header from count data columns are not unique! Only the first occurrence will be retained in matrix.
2: In GeneticAnalysisPipeline::createAnalysisDataSet(fileCountData = countFile,  :
  Dimension of data sources differ! Samples without corresponding count data will be loss.
3: In GeneticAnalysisPipeline::createAnalysisDataSet(fileCountData = countFile,  :
  Count data seems to have floating point numbers (ignore this message otherwise). Ambiguous counts do not work properly with most statistical models and are therefore not supported!
4: In GeneticAnalysisPipeline::createAnalysisDataSet(fileCountData = countFile,  :
  Fractional digits will be cut off, which could lead to inaccurate or misleading results!
Object has been saved!
```

Listing 6.3: Execution of cli_createAnalysisDataSet.

The warnings in line 26 to 28 shown in listing 6.3 were caused of the existence of multiple samples per patient. Furthermore how shown in 29 to 32, TCGA has used ambiguous counts for the genes.
The execution of the first script already realizes the requirements C.F1 and C.F2. The detailed execution of the other three CLI scripts will not be further illustrated. They could all be used similar to the cli_createAnalysisDataSet script shown in listing 6.3 and serve every option that is needed to fulfill the functional requirements at minimum.

The figure 7 shows the distribution of the groups which are stored in the AnalysisInfo object.

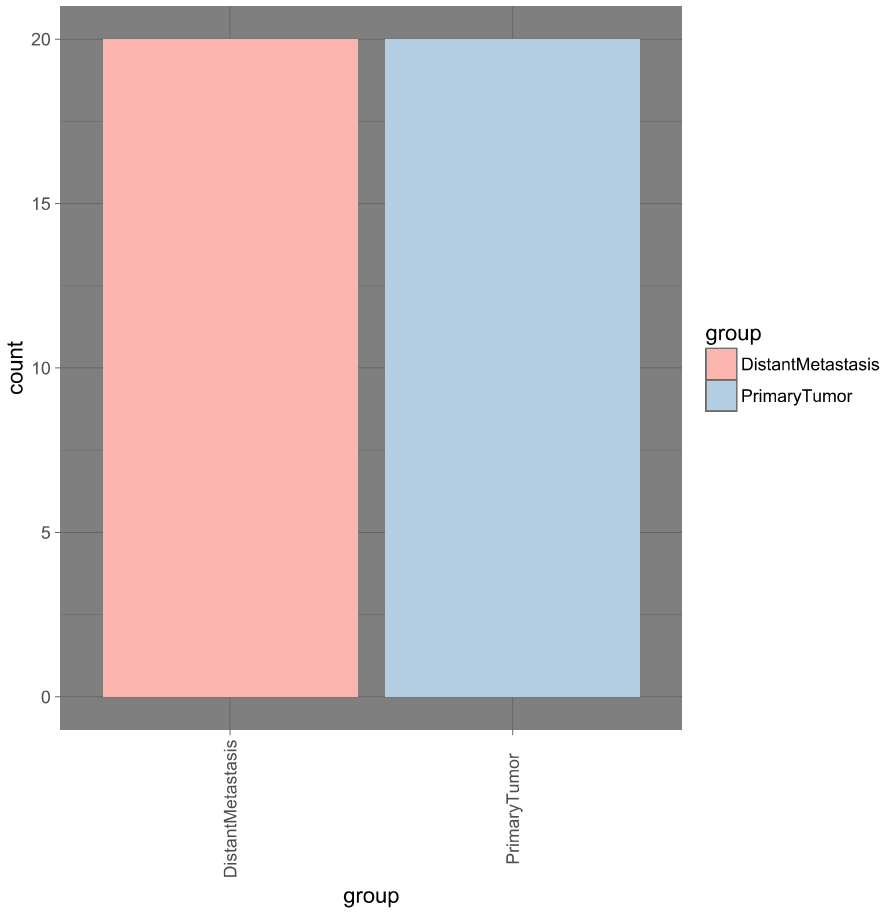

Figure 7: Bar plot generated by the function plotAnalysisInfo with customized theme and color.

The following PCA plot shows the result of the function *runQualityAssurance*. In an optimal case the two dots of the several groups would have a short distance to group members and a long distance to other groups. Therefore the count data in the test case seems not to be optimal.

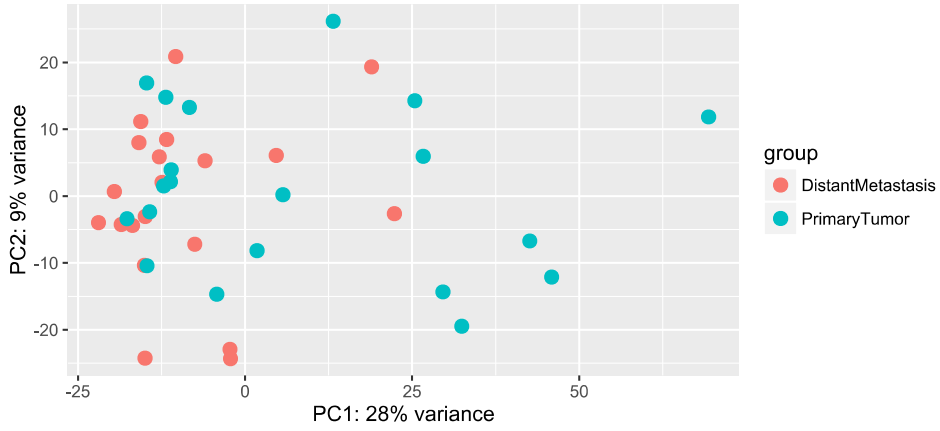

Figure 8: The PCA plot of the rlog method generated by the function plotQualityAssurance.

In figure 9 a heat map indicates several groups by clustering. Here the log2 transformed data was used.

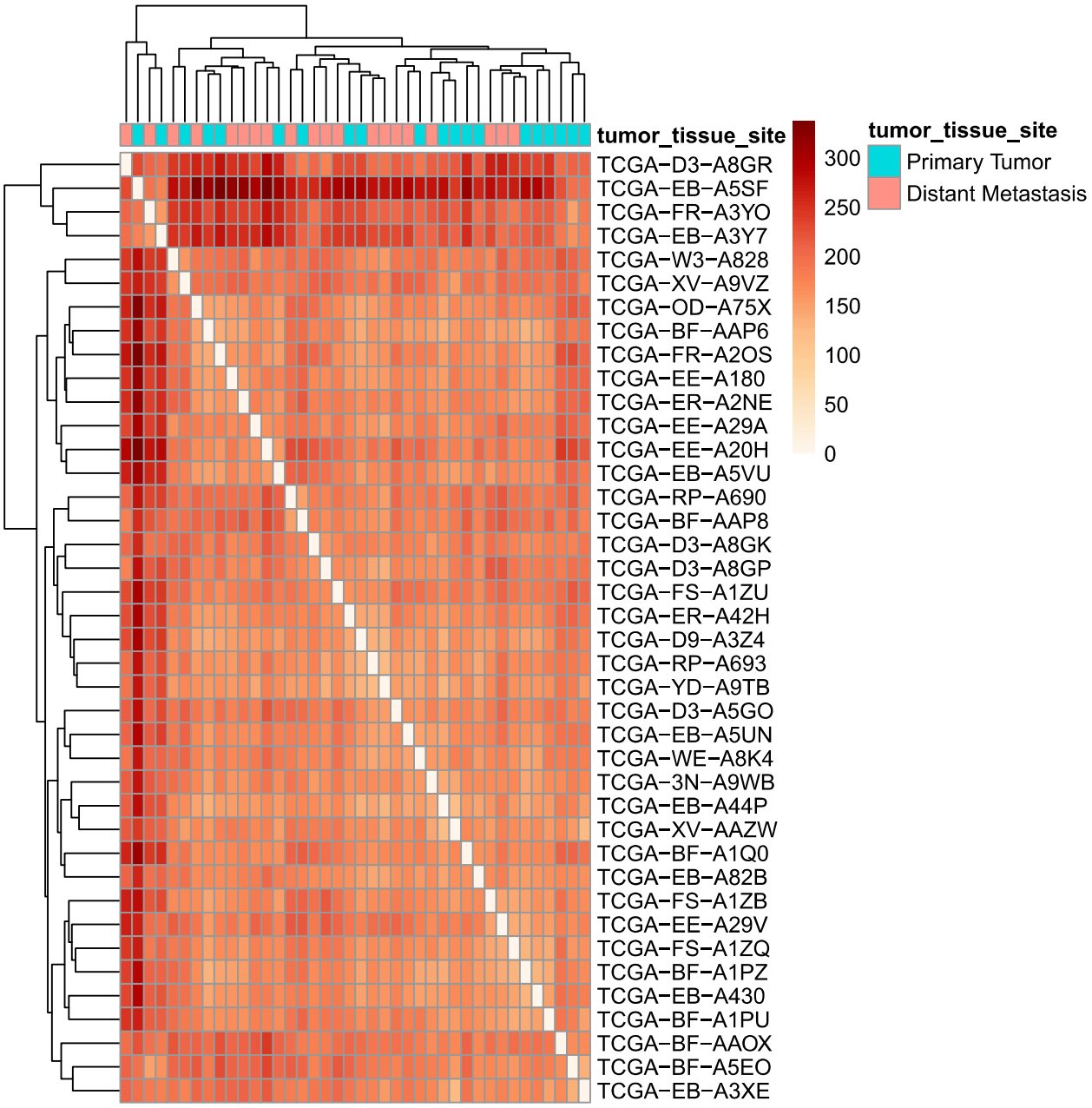

Figure 9: A heat map "hmDist" is showing the results of the log2 transformation.

Finally the Venn diagram is shown. In the case used for evaluation, the package edgeR has detected 71 differentially expressed genes, DESeq2 81 and limma 0. The relatively low result could be caused through the low sample count and also bad sample quality.

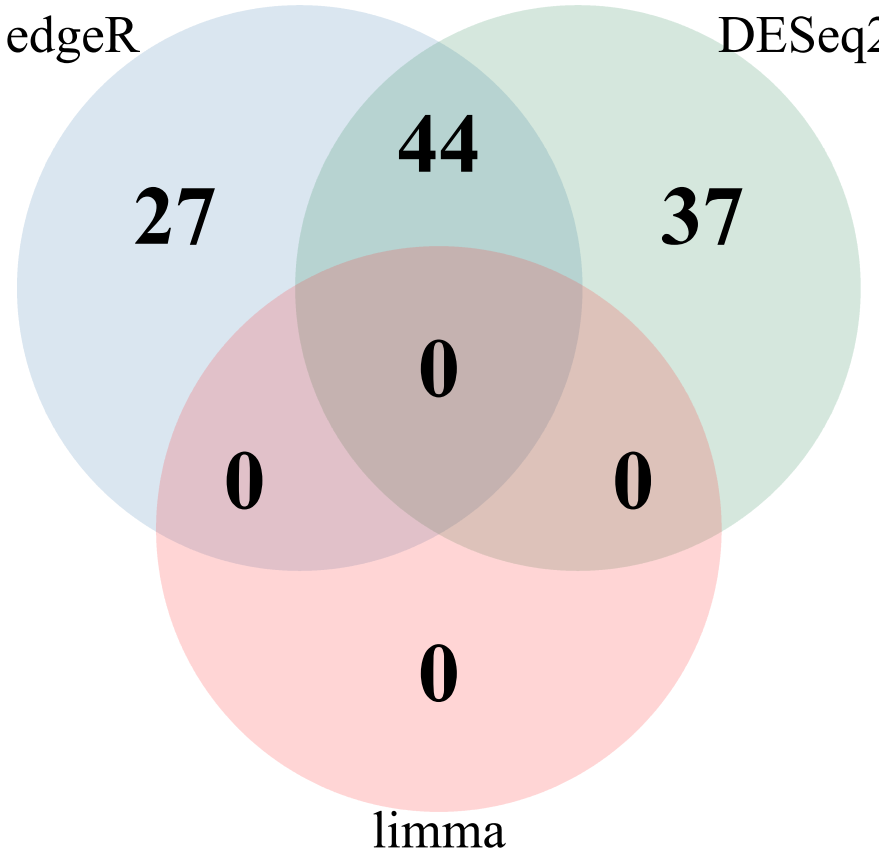

Figure 10: Venn diagram shows the intersections of the several determined differentially expressed genes between the used packages.

All other plots could also be produced, but will not be shown here for the sake of brevity. Additionally the files have been created properly and so all functional requirements has been fulfilled.
The non-functional requirements NF1 and partly NF2 has been fulfilled through the CLI. The usage of a package structure, the high abstraction of the pipeline and the strong modularization of the several used statistical packages finally satisfying the remaining requirements.

7 CONCLUSION

The tools *assembleRNAseq2*, *assembleClinical* and the *GeneticAnalysisPipeline* package whose have been implemented in the context of this thesis could help scientists to detect differential expressed genes between samples. Many additional features allowing a high customization of every analysis and their exports. Through the uniformed parameters across the packages, no additionally manual should be necessary. Furthermore the several stages of an analysis could be saved and resumed. Through the integrated CLI, no programming skills are necessary for the usage. Finally, the developed tools could save a lot of time.

The high modularization and abstraction are fitting very well to the principal concept of a pipeline. The strict separation of the data preparation step from the main application through the usage of Python enables a fast implementation of other data sources (for example a data base). Also the CLI interface could be exchanged quickly.
The same could be said about the main application, written in R. The packages are encapsulated and could be exchanged and extended without costing too much time.

In the future, another interface could be implemented to the pipeline. This could be for example a REST (abbr. for representational state transfer) interface, which is often used for web services [57].
Also possible is the extension of the GeneticAnalysisPipeline package. For example could the information from the principal component analysis be used to detect correlations between columns of the sample data automatically. This feature detection could also help the user by setting up the hypothesis.
Another great feature would be the possibility to make a look up to a gene ontology by using the labels of the differential expressed genes (see also [42], p 48). Through this more information could be retrieved.

Bibliography

1: U.S. Department of Energy Human Genome Project, Human Genome Project, 2014, https://web.ornl.gov/sci/techresources/Human_Genome/index.shtml (accessed 2016/04/07).
2: U.S. Department of Energy Human Genome Project, Major Events in the U.S. Human Genome Project and Related Projects, 2015, https://web.ornl.gov/sci/techresources/Human_Genome/project/timeline.shtml (accessed 2016/04/07).
3: Tripp, Simon and Grueber, M., economic impact of the human genome project, 2011, https://web.ornl.gov/sci/techresources/Human_Genome/publicat/BattelleReport2011.pdf (accessed 2016/04/07).
4: Avison, Matthew B., Measuring Gene Expression. Garland Science, 2005
5: Trevino, Victor et al., DNA Microarrays: a Powerful Genomic Tool for Biomedical and Clinical Research, 2007, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1933257/ (accessed 2016/04/07).
6: Bioconductor, All Packages, 2016, https://www.bioconductor.org/packages/release/BiocViews.html (accessed 2016/04/07).
7: Bioconductor, About, 2016, https://www.bioconductor.org/about/ (accessed 2016/04/07).
8: Huber, Wolfgang, et al, Orchestrating high-throughput genomic analysis with Bioconductor, 2015, https://www.nature.com/nmeth/journal/v12/n2/abs/nmeth.3252.html (accessed 2016/04/07).
9: CORDIS, SAGE-CARE, 2014, https://cordis.europa.eu/project/rcn/194165_en.html (accessed 2016/04/07).
10: Damm, Nico, Krebsbehandlung maßgeschneidert, 2015, https://www.h-da.de/fileadmin/h_da/Hochschule/Presse_Publikationen/campus_d/campus_d_nr15_WEB.pdf (accessed 2016/04/07).
11: NSilico Lifescience Ltd., Our Products, 2016, https://www.nsilico.com/Products (accessed 2016/04/07).
12: The University Of Edinburgh, Dr Paul Walsh, 2015, https://www.ed.ac.uk/pathway-medicine/our-staff/staff-profiles/paulwalsh (accessed 2016/04/07).
13: NSilico Lifescience Ltd., SimplicityMDT, 2016, https://www.nsilico.com/SimplicityMDT (accessed 2016/04/07).
14: NSilico Lifescience Ltd., MolPath, 2016, https://www.nsilico.com/Molpath (accessed 2016/04/07).
15: NSilico Lifescience Ltd., Simplicity, 2016, https://www.nsilico.com/Simplicity (accessed 2016/04/07).
16: Latchman, David, Gene regulation. Taylor & Francis, 2007
17: Oshlack, Alicia, Mark D. Robinson, and Matthew D. Young, From RNA-seq reads to differential expression results, 2010, https://www.biomedcentral.com/content/pdf/gb-2010-11-12-220.pdf (accessed).
18: Love, Michael, et al., RNA-seq workflow: gene-level exploratory analysis and differential expression, 2015, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4670015/ (accessed).
19: Soneson, Charlotte, and Mauro Delorenzi., A comparison of methods for differential expression analysis of RNA-seq data, 2013, https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-91 (accessed 2016/04/07).
20: TCGA, TCGA Data Portal Overview, 2016, https://tcga-data.nci.nih.gov/tcga/tcgaHome2.jsp (accessed 2016/04/07).

21: TCGA, Skin Cutaneous Melanoma: Case Counts, 2016, https://tcga-data.nci.nih.gov/tcga/tcgaCancerDetails.jsp?diseaseType=SKCM&diseaseName=Skin%20Cutaneous%20Melanoma (accessed 2016/04/07).
22: Python, About Python, , https://www.python.org/about/ (accessed 2016/04/07).
23: Python, Should I use Python 2 or Python 3 for my development activity?, 2015, https://wiki.python.org/moin/Python2orPython3 (accessed 2016/04/07).
24: Python, Whetting Your Appetite, 2016, https://docs.python.org/3/tutorial/appetite.html (accessed 2016/04/07).
25: Lutz, Mark, Learning Python. Beijing Sebastopol: O'Reilly, 2009
26: Pilgrim, Mark, Python 3 - Intensivkurs : Projekte erfolgreich realisieren. Imprint: Springer, 2010
27: Python, Objects, values and types, , https://docs.python.org/3.5/reference/datamodel.html#objects-values-and-types (accessed 2016/04/07).
28: R-Project, What is R?, , https://www.r-project.org/about.html (accessed 2016/04/07).
29: R-Project, An Introduction to R, , https://cran.r-project.org/doc/manuals/r-release/R-intro.html (accessed 2016/04/07).
30: R-Project, Writing R Extensions, , https://cran.r-project.org/doc/manuals/r-release/R-exts.html (accessed 2016/04/07).
31: Wickham, Hadley, Package structure, , https://r-pkgs.had.co.nz/package.html (accessed 2016/04/07).
32: Wickham, Hadley, OO field guide, , http://adv-r.had.co.nz/OO-essentials.html (accessed 2016/04/07).
33: R-Project, R: A Language and Environment for Statistical Computing, 2016, https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf (accessed 2016/04/07).
34: Bo Li, Colin Dewey, RSEM (RNA-Seq by Expectation-Maximization), 2016, https://deweylab.github.io/RSEM/ (accessed 2016/04/07).
35: Anders Simon, Counting reads in features with htseq-count, 2016, https://www-huber.embl.de/users/anders/HTSeq/doc/count.html (accessed 2016/04/07).
36: TCGA, RNASeq Version 2, 2013, https://wiki.nci.nih.gov/display/TCGA/RNASeq+Version+2 (accessed 2016/04/07).
37: TCGA, Biotab, 2014, https://wiki.nci.nih.gov/display/TCGA/Biotab (accessed 2016/04/07).
38: Love, Michael, Simon Anders, Wolfgang Huber, Differential analysis of count data - the DESeq2 package, , https://bioconductor.org/packages/release/bioc/vignettes/DESeq2/inst/doc/DESeq2.pdf (accessed 2016/04/07).
39: Love, Michael, Simon Anders, Wolfgang Huber, Package ‘DESeq2’, 2016, https://bioconductor.org/packages/release/bioc/manuals/DESeq2/man/DESeq2.pdf (accessed 2016/04/07).
40: Bioconductor, Download stats for Bioconductor Software packages, 2016, https://bioconductor.org/packages/stats/index.html (accessed 2016/04/07).
41: Yunshun Chen, Davis J. McCarthy, Robinson, Mark D., and Gordon K. Smyth, edgeR: differential expression analysis of digital gene expression data, 2015, https://bioconductor.org/packages/release/bioc/vignettes/edgeR/inst/doc/edgeRUsersGuide.pdf (accessed 2016/04/07).
42: Robinson, Mark D., Davis J. McCarthy, and Gordon K. Smyth., edgeR: a Bioconductor package for differential expression analysis of digital gene expression data, 2010, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2796818/ (accessed 2016/04/07).

43: Oberg, Ann L., et al., Technical and biological variance structure in mRNA-Seq data: life in the real world, 2012, https://bmcgenomics.biomedcentral.com/articles/10.1186/1471-2164-13-304 (accessed 2016/04/07).
44: Yunshun Chen, Davis J. McCarthy, Robinson, Mark D., and Gordon K. Smyth, Package ‘edgeR’, 2016, https://bioconductor.org/packages/release/bioc/manuals/edgeR/man/edgeR.pdf (accessed 2016/04/07).
45: Gordon K. Smyth, et al., limma: Linear Models for Microarray and RNA-Seq Data User's Guide, 2016, https://bioconductor.org/packages/release/bioc/vignettes/limma/inst/doc/usersguide.pdf (accessed 2016/04/07).
46: Law, Charity W., et al., Voom: precision weights unlock linear model analysis tools for RNA-seq read counts, 2014, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4053721/ (accessed 2016/04/07).
47: Gordon K. Smyth, et al., Package 'limma', 2016, https://bioconductor.org/packages/release/bioc/manuals/limma/man/limma.pdf (accessed 2016/04/07).
48: The Galaxy, Data intensive biology for everyone, , https://galaxyproject.org/ (accessed).
49: Python, argparse — Parser for command-line options, arguments and sub-commands, , https://docs.python.org/2/library/argparse.html (accessed 2016/04/07).
50: Wickham, Hadley, and Winston Chang, Package ‘devtools’, 2016, https://cran.r-project.org/web/packages/devtools/devtools.pdf (accessed 2016/04/07).
51: Wickham, H., P. Danenberg, and M. Eugster, roxygen2: In-Source Documentation for R, 2016
52: Wickham, Hadley, testthat: Unit Testing for R, 2016, https://cran.r-project.org/web/packages/testthat/index.html (accessed 2016/04/07).
53: Pagès, H., M. Lawrence, and P. Aboyoun, S4 implementation of vectors and lists, 2016, https://bioconductor.org/packages/release/bioc/html/S4Vectors.html (accessed 2016/04/07).
54: Huber, Wolfgang, Package ‘vsn’, 2016, https://bioconductor.org/packages/release/bioc/manuals/vsn/man/vsn.pdf (accessed 2016/04/07).
55: Wickham, Hadley, and Winston Chang, Help topics, , http://docs.ggplot2.org/current/ (accessed 2016/04/07).
56: Davis, L. Travor, optparse: Command Line Option Parser, 2015, https://cran.r-project.org/web/packages/optparse/index.html (accessed 2016/04/07).
57: Fielding, Roy Thomas, Architectural Styles and the Design of Network-based Software Architectures, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (accessed 2016/04/07).

8 APPENDIX A

Package 'GeneticAnalysisPipeline'

April 24, 2016

Title A Modular, Simple and Fast Pipeline for Differential Gene Expression Analysis

Version 0.0.0.9000

Author Bernhard Humm [cre], Paul Walsh [cre], Markus Leipold [aut]

Maintainer Bernhard Humm <bernhard.humm@h-da.de>, Paul Walsh <paul.walsh@nsilico.com>

Description
Create a Differential Expression Analysis with the help of an user friendly pipeline. You can control the analysis steps interactively via R console, or through a command-line interface.

Depends R (>= 3.2.4)

License What license is it under?

LazyData true

Suggests testthat, data.table, ggplot2, DESeq2, edgeR, limma, vsn, pheatmap, RColorBrewer, genefilter, grid, VennDiagram

Imports SummarizedExperiment, GenomicRanges, S4Vectors, methods

RoxygenNote 5.0.1

R topics documented:

AnalysisDataSet-class 2
AnalysisInfo-class 4
AnalysisResult-class 7
compareResultsToRef 9
createAnalysisDataSet 9
createAnalysisInfo 11
decideTestsDESeq2 12
decideTestsEdgeR 13
decideTestsLimma 15
detectDatatype 16
estimateDispersionsDESeq2 17
estimateDispersionsEdgeR 19
estimateDispersionsLimma 20
estimateSizeFactorsDESeq2 21
estimateSizeFactorsEdgeR 22
estimateSizeFactorsLimma 24
filterCountData 25
fitModelTestDESeq2 26

fitModelTestEdgeR . . . 27
fitModelTestLimma . . . 29
GeneticAnalysisPipeline . . . 30
initDESeq2 . . . 31
initEdgeR . . . 32
initLimma . . . 33
joinResults . . . 33
normalizeResultsDESeq2 . . . 34
normalizeResultsEdgeR . . . 35
normalizeResultsLimma . . . 35
plotAnalysisInfo . . . 36
plotBar2dFacet . . . 36
plotBoxplot . . . 38
plotComparison . . . 39
plotHeatMapDist . . . 39
plotHeatMapExp . . . 40
plotLibSizeDist . . . 41
plotMDS . . . 42
plotMeanSd . . . 42
plotNA . . . 43
plotPCA . . . 44
plotPosterioriDESeq2 . . . 45
plotPosterioriEdgeR . . . 45
plotPosterioriLimma . . . 46
plotPosterioriObjects . . . 46
plotPrioriDESeq2 . . . 47
plotPrioriEdgeR . . . 47
plotPrioriLimma . . . 48
plotPrioriObjects . . . 48
plotQualityAssurance . . . 49
plotVennDiagram . . . 49
QualityAssuranceResult-class . . . 50
runDifferentialExpressionAnalysis . . . 51
runQualityAssurance . . . 52
saveResults . . . 53
serializeAnalysis . . . 53
unserializeAnalysis . . . 54

Index **55**

`AnalysisDataSet-class` *AnalysisDataSet class and constructors*

Description

`AnalysisDataSet` is a subclass of `RangedSummarizedExperiment`, and storing all relevant input data for the Genetic Analysis pipeline. Additionally, this class checks for properly input and make it possible to easily access count matrix and sample data through correspondingly `countData` and `sampleData`. `AnalysisDataSet` checks also if order of input is correct and the matrix has non negative integer values.

Usage

```
AnalysisDataSet(object, ...)

AnalysisDataSetFromMatrix(countData, sampleData, ...)

countData(object, ...)

countData(object, ...) <- value

sampleData(object, ...)

sampleData(object, ...) <- value

## S4 method for signature 'SummarizedExperiment0'
AnalysisDataSet(object, ...)

## S4 method for signature 'SummarizedExperiment'
AnalysisDataSet(object, ...)

## S4 method for signature 'RangedSummarizedExperiment'
AnalysisDataSet(object, ...)

## S4 method for signature 'missing'
AnalysisDataSet(object, ...)

## S4 method for signature 'AnalysisDataSet'
sampleData(object)

## S4 replacement method for signature 'AnalysisDataSet,data.frame'
sampleData(object) <- value

## S4 replacement method for signature 'AnalysisDataSet,DataFrame'
sampleData(object) <- value

## S4 method for signature 'AnalysisDataSet'
countData(object)

## S4 replacement method for signature 'AnalysisDataSet,matrix'
countData(object) <- value

## S4 method for signature 'matrix,missing'
AnalysisDataSetFromMatrix(countData, sampleData, ...)

## S4 method for signature 'matrix,data.frame'
AnalysisDataSetFromMatrix(countData, sampleData,
  ...)

## S4 method for signature 'matrix,DataFrame'
AnalysisDataSetFromMatrix(countData, sampleData,
  ...)

## S4 method for signature 'matrix,ANY'
```

```
AnalysisDataSetFromMatrix(countData, sampleData, ...)
```

Examples

```
countData = matrix(1:100, ncol = 4, dimnames = list(1:25, 1:4))
sampleData = data.frame(sample = c("A", "B", "C", "D"))
sds = AnalysisDataSetFromMatrix(countData, sampleData)
```

AnalysisInfo-class *AnalysisInfo class and constructors*

Description

AnalysisInfo acts as container for analysis informations for the Genetic Analysis pipeline. It includes metadata like name, time, formula and a corresponding AnalysisDataSet with attached countData and sampleData. Additionally, this class checks for plausible input and make changes if necessary (see warnings).

Usage

```
AnalysisInfo(time, name, formula, model, group, contrast, envir, ...)

envir(object, ...)

envir(object, ...) <- value

time(x, ...)

time(object, ...) <- value

name(object)

name(object, ...) <- value

formula(x, ...)

formula(object, ...) <- value

model(object)

model(object, ...) <- value

group(object)

contrast(object)

contrast(object, ...) <- value


  ## S4 method for signature
```

```
## 'character,character,formula,matrix,missing,list,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'character,missing,formula,matrix,missing,list,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'character,character,formula,missing,missing,missing,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'missing,character,formula,matrix,missing,list,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'character,missing,formula,missing,missing,missing,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'missing,character,formula,missing,missing,missing,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'missing,missing,formula,matrix,missing,list,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)


  ## S4 method for signature
## 'missing,missing,formula,missing,missing,missing,AnalysisDataSet'
AnalysisInfo(time,
  name, formula, model, group, contrast, envir, ...)

## S4 method for signature 'AnalysisInfo'
envir(object)

## S4 replacement method for signature 'AnalysisInfo,AnalysisDataSet'
envir(object) <- value
```

```
## S4 method for signature 'AnalysisInfo'
formula(x)

## S4 replacement method for signature 'AnalysisInfo,formula'
formula(object) <- value

## S4 method for signature 'AnalysisInfo'
model(object)

## S4 replacement method for signature 'AnalysisInfo,formula'
model(object) <- value

## S4 method for signature 'AnalysisInfo'
group(object)

## S4 method for signature 'AnalysisInfo'
contrast(object)

## S4 replacement method for signature 'AnalysisInfo,list'
contrast(object) <- value

## S4 method for signature 'AnalysisInfo'
time(x)

## S4 replacement method for signature 'AnalysisInfo,character'
time(object) <- value

## S4 method for signature 'AnalysisInfo'
name(object)

## S4 replacement method for signature 'AnalysisInfo,character'
name(object) <- value

## S4 method for signature 'AnalysisInfo'
show(object)
```

Slots

`time` Character vector with creation time stamp.

`name` Character vector including the name of analysis.

`formula` Formula of interest.

`model` Optional model matrix, corresponding to given formula.

`group` A factor which should be built out of the concatenated values from the columns of interest.

`contrast` A list of contrast vectors.

`envir` A `link{AnalysisDataSet}` object, with sample and count data which is needed in experimental context.

See Also

AnalysisDataSet AnalysisDataSetFromMatrix

Examples

```
countData = matrix(1:100, ncol = 4, dimnames = list(1:25, 1:4))
sampleData = data.frame(sample = c("A", "B", "C", "B"))
analysisDataSet = AnalysisDataSetFromMatrix(countData, sampleData)
analysisInfo = AnalysisInfo(time = date(),
name = "example",
formula = ~ sample,
envir = analysisDataSet,
)
```

AnalysisResult-class *AnalysisResult class and constructors*

Description

`AnalysisResult` acts as container for analysis result data for the differential expression pipeline. It includes meta data like `name` for package name, `objects`, which is preserved for the package depending provisional results (e.g. dispersion estimations) together with the `result`. Additionally, this class checks for plausible input and make changes if necessary (see warnings).

Usage

```
AnalysisResult(name, priorObjects, posteriorObjects, result, ...)

result(object)

result(object, ...) <- value

priorObjects(object)

priorObjects(object, ...) <- value

posteriorObjects(object)

posteriorObjects(object, ...) <- value

## S4 method for signature 'character,list,list,list'
AnalysisResult(name, priorObjects,
  posteriorObjects, result, ...)

## S4 method for signature 'character,list,list,missing'
AnalysisResult(name, priorObjects,
  posteriorObjects, result, ...)

## S4 method for signature 'character,list,missing,missing'
AnalysisResult(name, priorObjects,
  posteriorObjects, result, ...)

## S4 method for signature 'character,missing,missing,missing'
AnalysisResult(name,
```

```
  prioriObjects, posterioriObjects, result, ...)

## S4 method for signature 'AnalysisResult'
name(object)

## S4 replacement method for signature 'AnalysisResult,character'
name(object) <- value

## S4 method for signature 'AnalysisResult'
result(object)

## S4 replacement method for signature 'AnalysisResult,list'
result(object) <- value

## S4 method for signature 'AnalysisResult'
prioriObjects(object)

## S4 replacement method for signature 'AnalysisResult,list'
prioriObjects(object) <- value

## S4 method for signature 'AnalysisResult'
posterioriObjects(object)

## S4 replacement method for signature 'AnalysisResult,list'
posterioriObjects(object) <- value

## S4 method for signature 'AnalysisResult'
show(object)
```

Slots

`name` Character vector including the name of the used package.

`prioriObjects` List of needed package depending objects which are needed in further context.

`posterioriObjects` List which includes lists of objects (per contrast) that could be used in further context.

`result` A `data.frame` object including normalized result informations.

See Also

AnalysisDataSet AnalysisDataSetFromMatrix

Examples

```
countData = matrix(1:100, ncol = 4, dimnames = list(1:25, 1:4))
sampleData = data.frame(sample = c("A", "B", "C", "B"))
sds = AnalysisDataSetFromMatrix(countData, sampleData)
sdsInfo = AnalysisInfo(time = date(),
name = "example",
formula = ~ sample,
envir = sds,
)
```

compareResultsToRef *Compare result tables against references.*

Description

Compare result tables from AnalysisResult object with a reference or multiple references.

Usage

```
compareResultsToRef(listAnalysisResult, lReferenceTables, refIDcolumn = 1,
  export = TRUE, separator = "\t", pathTables = paste0(".",
  .Platform$file.sep), file_extension = ".tsv", verbose = 2, ...)
```

Arguments

listAnalysisResult	A vector or list with AnalysisResult objects and attached result.
lReferenceTables	A list with attached reference data.frame objects. Example: list("maurerer" = data.frame()).
refIDcolumn	Integer with column number to ID's in reference table.
export	Boolean if table should be written to hard drive.
separator	Delimiter for columns used in resulting file.
pathTables	Path to directory, which should be used for export.
file_extension	File extension (should fit to separator).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

runDifferentialExpressionAnalysis, AnalysisResult

createAnalysisDataSet *Create easily an* AnalysisDataSet *object*

Description

First state of the GeneticAnalysisPipeline package.The function checks if supported files are appropriate for the further analysis and returns a AnalysisDataSet.
Additionally it is possible to normalize the sample data file. Through this automatically NA words and data types will be detected (see detectDatatype for more information).

Usage

```
createAnalysisDataSet(fileCountData, fileSampleData, sepCountData = "\t",
  sepSampleData = "\t", keyColSampleData = 1, detectDataType = TRUE,
  countDataHasRownames = TRUE, verbose = 2, ...)
```

Arguments

fileCountData	Path to file which is including the count data.
fileSampleData	Filepath leading to sample data.
sepCountData	Column separator used in count file.
sepSampleData	Delimiter for columns used in sample file.
keyColSampleData	String or integer with name or number from the key column of the supplied sample file.
detectDataType	Makes use of the function detectDatatype.
countDataHasRownames	True, if labels are supplied at the first column of the count table. Otherwise rows will be numbered automatically.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

A AnalysisDataSet.

See Also

fread which enables faster read and will be used for the count file, if the package is installed. detectDatatype to adjust automatic normalization with further optional arguments.

Examples

```
## Not run:
# Create exemplary sample data
sampleData = c(77, 1, 2, 3, "NOT AVAILABLE", 4, "NOT AVAILABLE", 9)
sampleData = data.frame(sampleData)
sampleCount = nrow(sampleData)
# Create count data
countData = matrix(data = rep(c(100:0), times = sampleCount), ncol = sampleCount)
colnames(countData) = 1:sampleCount

# Write temporary count file
tfCount = tempfile()
write.table(countData,
file = tfCount,
sep = "\t",
row.names = TRUE,
col.names = NA,
quote = FALSE)
# Write temporary sample file
tfSample = tempfile()
write.table(sampleData,
file = tfSample,
sep = "\t",
row.names = TRUE,
col.names = NA,
quote = FALSE)

# Create AnalysisDataSet which could be used in analysis pipeline
analysisDataSet = createAnalysisDataSet(fileCountData = tfCount,
```

```
fileSampleData = tfSample,
verbose = 2)

## End(Not run)
```

---

createAnalysisInfo *Second state in* GeneticAnalysisPipeline *pipeline.*

Description

The outcome from initalizeDataSet needs to be more specified for a common expression analysis. Relevant columns will be filtered automatically with usage of a given formula. In the case of a blind analysis (formula = ~ 1), target columns should be specified by selection argument. It is also possible to filter rows by using R's binary operators (see Comparison.

Usage

```
createAnalysisInfo(analysisDataSet, time, name, formula, contrast, selection,
  condition, referenceValue, rndGroupSize, verbose = 2, ...)
```

Arguments

analysisDataSet	A AnalysisDataSet object.
time	Timestamp from analysis.
name	Name from analysis.
formula	Formula for analysis.
contrast	Numeric vector, describing the factor for each coefficient (equal to the columns in the used model.matrix).
selection	Optional column selection (e.g.: for ~ 1 usage).
condition	Optional condition to select relevant rows only. countData will be automatically fitted to sampleData.
referenceValue	The last specified formula parameter should be the one of interest. Because the standard reference Factor's is internally chosen alphabetically, a reference should be given (equal to control group) to this parameter.
rndGroupSize	A integer used to select a pseudo random number of every result constellation.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

AnalysisInfo object.

See Also

AnalysisDataSet, AnalysisInfo.

Examples

```
# Create exemplary sample data
color = c("blue", "brown", "blue", "green")
gender = c("female", "male", "male", "female")
sampleData = data.frame(gender, color, row.names = letters[1:4])
# Create count data
countData = matrix(as.integer(c(1,2,3,4)), ncol = 4,
dimnames = list(1, letters[1:4]))
# Setup analysis
formula = ~ 1 + gender
name = "test"
time = date()
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)

# Create AnalysisInfo
analysisInfo = createAnalysisInfo(analysisDataSet = analysisDataSet,
time = time,
name = name,
formula = formula)

# Create AnalysisInfo, filter by gender
analysisInfo = createAnalysisInfo(analysisDataSet = analysisDataSet,
time = time,
name = name,
formula = formula,
condition = "gender == \"female\"",
verbose = 3)
```

---

decideTestsDESeq2 *Fifth state of* DESeq2 *package differential expression analysis.*

Description

This function is used as wrapper for the fifth possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. In this functions, results will be evaluated and extracted.

Usage

```
decideTestsDESeq2(dseqDataSet, contrast, method = "BH", alpha = 0.05,
  verbose = 1, ...)
```

Arguments

dseqDataSet	Object from DESeqDataSet class.
contrast	Matrix or numeric vector, indicating the coefficients that will be tested to be equal to zero.

method	Adjust methods: BH: Used in DESeq2 and edgeR standardly, also left as standard here. BY: Benjamini-Yekutieli (2001). holm: a less conservative correction by Holm (1988). none: pass through.
alpha	Significance cut-off value (default: 0.05)
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

DESeq2 function to classify the results. Different schemes allows multiple testing across contrasts or over genes.

Value

DESeqResults object with attached result.

See Also

DESeq2: results
edgeR: decideTestsDGE
limma: decideTests
For more adjust methods: p.adjust

Examples

```
# Create count data using DESeq2
dds = DESeq2::makeExampleDESeqDataSet()
countData = SummarizedExperiment::assay(dds)
sampleData = SummarizedExperiment::colData(dds)
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dseq = initDESeq2(analysisInfo)
dseq = estimateSizeFactorsDESeq2(dseq)
dseq = estimateDispersionsDESeq2(dseq)
dseq = fitModelTestDESeq2(dseq)
dseq = decideTestsDESeq2(dseq)
```

Description

This function is used as wrapper for the fifth possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. In this functions, results will be evaluated and extracted.

Usage

```
decideTestsEdgeR(edgeRdataSet, contrast, method = "BH", alpha = 0.05,
  verbose = 1, ...)
```

Arguments

edgeRdataSet	Object from DGELRT or DGEExact class.
contrast	Numeric vector, indicating the coefficients that will be tested to be equal to zero.
method	Adjust methods: BH: Used in DESeq2 and edgeR standardly, also left as standard here. BY: Benjamini-Yekutieli (2001). holm: a less conservative correction by Holm (1988). none: pass through.
alpha	Significance cut-off value (default: 0.05)
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

edgeR function to classify the results. Different schemes allows multiple testing across contrasts or over genes. Adapted from decideTests.

Value

TestResults object with attached result.

See Also

DESeq2: results
edgeR: decideTestsDGE
limma: decideTests
For more adjust methods: p.adjust

Examples

```
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
```

```
formula = ~ group,
envir = analysisDataSet)
dgelist = initEdgeR(analysisInfo)
dgelist = estimateSizeFactorsEdgeR(dgelist)
dgelist = estimateDispersionsEdgeR(dgelist)
dgelist = fitModelTestEdgeR(dgelist)
```

`decideTestsLimma` *Fifth state of* `limma` *package differential expression analysis.*

Description

This function is used as wrapper for the fifth possible step of all used differential expression analysis packages used in `GeneticAnalysisPipeline`. In this functions, results will be evaluated and extracted.

Usage

```
decideTestsLimma(limmaDataSet, contrast, method = "BH", alpha = 0.05,
  verbose = 1, ...)
```

Arguments

`limmaDataSet`	`mArrayLM` object from `DGELRT` or `DGEExact` class.
`contrast`	Numeric vector, indicating the coefficients that will be tested to be equal to zero.
`method`	Adjust methods: `BH`: Used in `DESeq2` and `edgeR` standardly, also left as standard here. `BY`: Benjamini-Yekutieli (2001). `holm`: a less conservative correction by Holm (1988). `none`: pass through.
`alpha`	Significance cut-off value (default: 0.05)
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

`limma` function to classify the results. Different schemes allows multiple testing across contrasts or over genes. Similar to `decideTestsDGE`.

Value

`TestResults` object with attached result.

See Also

limma: `decideTests`
DESeq2: `results`
edgeR: `decideTestsDGE`
For more adjust methods: `p.adjust`

Examples

```
requireNamespace("limma")
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ group,
envir = analysisDataSet)
dgelist = initLimma(analysisInfo)
dgelist = estimateSizeFactorsLimma(dgelist)
dgelist = estimateDispersionsLimma(dgelist)
mArrayLM = fitModelTestLimma(dgelist)
mArrayLM = decideTestsLimma(mArrayLM)
```

`detectDatatype` `detectDatatype` *is a helper function for the* `GeneticAnalysisPipeline` *pipeline*

Description

This function helps `read.table` to:

- detect missing values and replacing them with the `NA` constant from the `R` language.
- change integers to numerics (standardly).
- change factors to numerics, assuming a column with numerics.
- change factors to dates, assuming a column with dates.

Usage

```
detectDatatype(file, sep = "\t", header = TRUE, dec = ".", quote = "",
  nrows = -1L, naStrings, dateFormat = "%Y-%m-%d",
  shrinkFactors = TRUE, ubound = 0.8, lbound = 0.15, naTolerance = 0.8,
  skipFirstColumn = FALSE, verbose = 1, ...)
```

Arguments

`file`	The path or name of the file to read from.
`sep`	Field separator used in file.
`header`	Logical value indicating if the file contains column labels.
`dec`	Decimal point character of file.
`quote`	Quoting character. Set "" to disable.
`nrows`	Count of rows that should be read for interpretation.

naStrings	Vector with possible NA candidates.
dateFormat	String including a constellations of %d, %m, %y and %Y are supported.
shrinkFactors	Boolean value indicating if values of factors should be reduced using naStrings.
ubound	Percentage (decimal), of occurrences a class must have, so that no further estimations (lbound) are needed for conversion.
lbound	Absolute minimum amount of occurrences, that a class must have to make the deeper check on unique NA values naTolerance.
naTolerance	Percentage of uniqueness the estimated NA values must have serving the lbound.
skipFirstColumn	Logical value indicates if the first column should be passed through only (e.g. because of row names). This column will be coerced to class character.

Value

A list with the attributes colClasses and na.strings.

Info

useDataTablePkg !CurrentlyDeactivated! Boolean value. TRUE enables fread from data.table package instead of using read.table.

See Also

This is a helper function for read.table or otherwise if enabled and installed fread.

Examples

```
## Not run:
sample = c(77, 1, 2, 3, "NOT AVAILABLE", 4, "NOT AVAILABLE", 9)
sampleData = data.frame(someNumerics = sample)
tf = tempfile()
write.table(sampleData, tf, sep = "\t", row.names = FALSE, col.names = TRUE,
quote = FALSE)
readData = read.table(tf, sep = "\t", header = TRUE)
# Output: "factor"
is(readData[, 1], "factor")
sampleDataNumeric = detectDatatype(file = tf, verbose = 0)
# Output: "numeric"
is(sampleDataNumeric[, 1], "numeric")
## End(Not run)
```

---

estimateDispersionsDESeq2

Third state of DESeq2 *package differential expression analysis.*

Description

This function is used as wrapper for the third possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. Using the estimated size factors out of the second calculation step, it is possible to make dispersion estimations between samples of the same group, using the supplied formula. The outcome should be a fitted model.

Usage

```
estimateDispersionsDESeq2(dseqDataSet, method = c("parametric", "local",
  "mean"), verbose = 1, ...)
```

Arguments

dseqDataSet	Object from DESeqDataSet class.
method	Method for estimating dispersion trend: parametric: Fit a dispersion-mean relation. local: Log dispersion against log mean, weighted by normalized mean count. mean: Uses mean of gene wise dispersion estimates.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

DESeq2 function makes standardly usage of the Cox-Reid method, to estimate dispersion for Negative Binomial distributed data (like edgeR).

Value

DESeqDataSet object.

See Also

DESeq2: estimateDispersions
edgeR: estimateDisp
limma: voom

Examples

```
# Create count data using DESeq2
dds = DESeq2::makeExampleDESeqDataSet()
countData = SummarizedExperiment::assay(dds)
sampleData = SummarizedExperiment::colData(dds)
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dseq = initDESeq2(analysisInfo)
dseq = estimateSizeFactorsDESeq2(dseq)
dseq = estimateDispersionsDESeq2(dseq)
```

estimateDispersionsEdgeR
Third state of edgeR *package differential expression analysis.*

Description

This function is used as wrapper for the third possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. Using the estimated size factors out of the second calculation step, it is possible to make dispersion estimations between samples of the same group, using the supplied formula. The outcome should be a fitted model.

Usage

```
estimateDispersionsEdgeR(edgeRdataSet, method = c("locfit", "movingave",
  "loess", "none"), robust = TRUE, isClassicEdgeR = FALSE, verbose = 1,
  ...)
```

Arguments

edgeRdataSet	Object from DGEList class.
method	Method for estimating dispersion trend: locfit, movingave, loess and none
robust	Boolean value. TRUE robustifies prior.df against outliers.
isClassicEdgeR	Boolean value that could switch edgeR's classic mode on and off. Classic mode works with one factorial experiments only!
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

edgeR function makes standardly usage of the Cox-Reid method, to estimate dispersion for Negative Binomial distributed data (like DESeq2).

Value

DGEList object.

See Also

edgeR: estimateDisp
Classic mode: estimateCommonDisp
"GLM" mode: estimateGLMCommonDisp
limma: voom
DESeq2: estimateDispersions

Examples

```
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initEdgeR(analysisInfo)
dgelist = estimateSizeFactorsEdgeR(dgelist)
dgelist = estimateDispersionsEdgeR(dgelist)
```

---

estimateDispersionsLimma

Third state of edgeR *package differential expression analysis.*

Description

This function is used as wrapper for the third possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. Using the estimated size factors out of the second calculation step, it is possible to make dispersion estimations between samples of the same group, using the supplied formula. The outcome should be a fitted model.

Usage

```
estimateDispersionsLimma(limmaDataSet, method = c("none", "quantile"),
  verbose = 1, ...)
```

Arguments

limmaDataSet	Object from DGEList class.
method	Normalization method for logCPM's: none: Standard, because normally no further normalization is needed. quantile: In case of very noisy data "quantile" is proposed.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

limma's voom function transforms count data to log2 counts per million, estimates the mean-variance relationships and finally uses the results to calculate the weights. Opposite to edgeR and DESeq2, now linear statistical models could be used. See also normalizeBetweenArrays for informations about the methods.

Value

EList object. Caution: Old data will be loss!

See Also

limma: voom
edgeR: estimateDisp
DESeq2: estimateDispersions

Examples

```
requireNamespace("limma")
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initLimma(analysisInfo)
dgelist = estimateSizeFactorsLimma(dgelist)
dgelist = estimateDispersionsLimma(dgelist)
```

---

estimateSizeFactorsDESeq2

Second state of DESeq2 *package differential expression analysis.*

Description

This function is used as wrapper for the second possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. It estimates size factors (aka normalization factors) which could fit the count values properly for the several analysis method. Target is to get robust values with focus to dispersion and variance.

Usage

```
estimateSizeFactorsDESeq2(dseqDataSet, method = c("ratio", "iterate"),
  verbose = 1, ...)
```

Arguments

dseqDataSet	Object from DESeqDataSet class.
method	ratio: Uses standard median ratio. iterate: Likelihood optimization, using a ~ 1 model.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

DESeq2 standardly estimates size factors using the "median ratio method". It generates an artificial pseudo sample out of the geometric mean for each gene over all samples. Factors are calculated median ratios from a sample against the pseudo sample.
For more information see estimateSizeFactors and for even more details estimateSizeFactorsForMatrix.

Value

DESeqDataSet object.

See Also

DESeq2: estimateSizeFactors
edgeR, limma: calcNormFactors

Examples

```
# Create count data using DESeq2
dds = DESeq2::makeExampleDESeqDataSet()
countData = SummarizedExperiment::assay(dds)
sampleData = SummarizedExperiment::colData(dds)
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dseq = initDESeq2(analysisInfo)
dseq = estimateSizeFactorsDESeq2(dseq)
```

---

estimateSizeFactorsEdgeR

Second state of edgeR *package differential expression analysis.*

Description

This function is used as wrapper for the second possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. It estimates size factors (aka normalization factors) which could fit the count values properly for the several analysis method. Target is to get robust values with focus to dispersion and variance.

Usage

```
estimateSizeFactorsEdgeR(edgeRdataSet, method = c("TMM", "RKE",
  "upperquartile", "none"), verbose = 1, ...)
```

Arguments

edgeRdataSet	Object from DESeqDataSet class.
method	TMM: Weighted trimmed mean of M-values. RLE: Scaling factor method (like ratio in DESeq. upperquartile: Scale factors are calculated using the 75 the counts for each library. none: All factors are set to 1.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

With edgeR, the size factors are estimated using the weighted trimmed mean of M-values according to a given reference refColumn. If no reference is given (what would be in the most cases), the sample with nearest upper quartile to the mean upper quartile is used.
For more information see calcNormFactors.

Value

DGEList object.

See Also

edgeR, limma: calcNormFactors
DESeq2: estimateSizeFactors

Examples

```
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initEdgeR(analysisInfo)
dgelist = estimateSizeFactorsEdgeR(dgelist)
```

estimateSizeFactorsLimma *Second state of* limma *package differential expression analysis.*

Description

This function is used as wrapper for the second possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. It estimates size factors (aka normalization factors) which could fit the count values properly for the several analysis method. Target is to get robust values with focus to dispersion and variance.

Usage

```
estimateSizeFactorsLimma(limmaDataSet, method = c("TMM", "RKE",
  "upperquartile", "none"), verbose = 1, ...)
```

Arguments

method	TMM: Weighted trimmed mean of M-values. RLE: Scaling factor method (like ratio in DESeq. upperquartile: Scale factors are calculated using the 75 the counts for each library. none: All factors are set to 1.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

Limma is making usage from edgeR's normalization methods:
The size factors are estimated using the weighted trimmed mean of M-values according to a given reference refColumn. If no reference is given (what would be in the most cases), the sample with nearest upper quartile to the mean upper quartile is used.
For more information see calcNormFactors.

Value

DGEList object.

See Also

edgeR, limma: calcNormFactors
DESeq2: estimateSizeFactors

Examples

```
requireNamespace("limma")
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
```

```
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initLimma(analysisInfo)
dgelist = estimateSizeFactorsLimma(dgelist)
```

`filterCountData` *This function serves possibilities to filter count matrices.*

Description

Filtering counts is a recommended step before starting a differential expression analysis. The outcome should be genes, where a group of interest exists, which is having a gene count lying above a threshold (biological variance). Benefit should be a better False Discovery Rate and faster calculations. The easiest way, for example, would be to filter over a row sum.
`cpm`:
Filter genes below a given threshold using CPM function from EdgeR package. Calculation: $\frac{1e06 * count}{totalcount}$ per gene in a sample

Usage

```
filterCountData(x, method = c("cpm"), limColumn, limRow, verbose = 2, ...)
```

Arguments

`x`	`AnalysisDataSet` or integer matrix which should be filtered.
`method`	Method(s) which should be used to filter the count data. `cpm`: Counts per million will be calculated over columns, using a threshold (`limColumn`). A second threshold (`limRow`) will be used over row, telling the minimum amount that (`limColumn`) has to be met, to keep a gene in the matrix.
`limColumn`	Expression threshold per column (over all genes from one sample). Typically a gene should have a count of 5 to 10 in a library.
`limRow`	Minimum count of samples that should have $cpm > th_column$ (over all samples). Typically the minimum sum of samples per group of interest.

Value

The filtered count matrix.

See Also

`cpm`

Examples

```
# Create count data
cntA1 = c(0, 10000, 0, 10000, 1)
cntA2 = c(0, 12000, 0, 12000, 1)
cntA3 = c(0, 13000, 0, 13000, 1)
cntB1 = c(0, 0, 10000, 100, 1)
cntB2 = c(0, 0, 12000, 120, 1)
cntB3 = c(0, 0, 13000, 130, 1)
# Create simple count matrix example
countData = matrix(c(cntA1, cntA2, cntA3, cntB1, cntB2, cntB3), ncol = 6)
# We can use the function from edgeR directly to get a better feeling about
# the terms.
print(edgeR::cpm.default(countData))
# Filter removes first "zero" row, but not the last row with the nothing
# telling 1. This is caused by the low amount of rows and therefore high
# relative cpm value.
filterCountData(x = countData,
limColumn = 8,
limRow = 3)
```

---

fitModelTestDESeq2 *Fourth state of* DESeq2 *package differential expression analysis.*

Description

This function is used as wrapper for the fourth possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. In this functions, coefficients will be tested for significance.

Usage

```
fitModelTestDESeq2(dseqDataSet, method = c("Wald", "LRT"), reducedFormula,
  betaPrior = FALSE, minReplicatesForReplace = 7L, verbose = 1, ...)
```

Arguments

dseqDataSet	Object from DESeqDataSet class.
method	Kind of hypothesis test: Wald: Wald test against zero. LRT: Likelihood ratio tests. Two models are fitted to counts (ANODEV). A reduced formula must be supplied.
reducedFormula	A reduced formula for ANODEV test, needed for LRT method.
minReplicatesForReplace	Integer value for the minimum group size (sample replicates) for outlier replacement.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

DESeq2 function makes standardly usage of the nbinomWaldTest function, using estimated standard error of a log2 fold change. The beta prior feature is only available, if internal model.matrix is used.

Value

DESeqDataSet object with attached result.

See Also

DESeq2: Wald nbinomWaldTest
LRT nbinomLRT
Also for outlier replacement replaceOutliers.
edgeR: glmFit
glmLRT
glmQLFit
glmQLFTest
limma: contrasts.fit
eBayes

Examples

```
# Create count data using DESeq2
dds = DESeq2::makeExampleDESeqDataSet()
countData = SummarizedExperiment::assay(dds)
sampleData = SummarizedExperiment::colData(dds)
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dseq = initDESeq2(analysisInfo)
dseq = estimateSizeFactorsDESeq2(dseq)
dseq = estimateDispersionsDESeq2(dseq)
dseq = fitModelTestDESeq2(dseq)
```

---

fitModelTestEdgeR *Fourth state of* edgeR *package differential expression analysis.*

Description

This function is used as wrapper for the fourth possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. In this functions, coefficients will be tested for significance.

Usage

```
fitModelTestEdgeR(edgeRdataSet, method = c("Fit", "QL"), contrast = NULL,
  robust = (method == "QL"), verbose = 1, ...)
```

Arguments

edgeRdataSet	Object from DGEList class.
method	Kind of hypothesis test: Fit: Standard, using mglmLevenberg or mglmOneGroup for fitting and glmLRT for testing. QL: Quasi-likelihood method using Bayes for fitting and testing.
contrast	Matrix or numeric vector, indicating the coefficients that will be tested to be equal to zero.
robust	Logical, if TRUE it will estimate the prior QL dispersion robustly (recommended).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

edgeR uses standardly in case of one-way layout designs (one factor) mglmOneGroup in background, otherwise mglmLevenberg fitting is taking place.

Value

DGELRT object with attached result.

See Also

edgeR: glmFit
glmLRT
glmQLFit
glmQLFTest
limma: contrasts.fit
eBayes
DESeq2: Wald nbinomWaldTest
LRT nbinomLRT

Examples

```
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ group,
envir = analysisDataSet)
```

```
dgelist = initEdgeR(analysisInfo)
dgelist = estimateSizeFactorsEdgeR(dgelist)
dgelist = estimateDispersionsEdgeR(dgelist)
dgelist = fitModelTestEdgeR(dgelist)
```

`fitModelTestLimma` *Fourth state of* `limma` *package differential expression analysis.*

Description

This function is used as wrapper for the fourth possible step of all used differential expression analysis packages used in `GeneticAnalysisPipeline`. In this functions, coefficients will be tested for significance.

Usage

```
fitModelTestLimma(limmaDataSet, contrast, robust = FALSE, verbose = 1, ...)
```

Arguments

`limmaDataSet`	Object from `EList` class.
`contrast`	Numeric vector, indicating the coefficients that will be tested to be equal to zero.
`robust`	Logical, if `TRUE` it will estimate the prior QL dispersion robustly.
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

`limma` is making usage of `lmfit` or `contrasts.fit`. Last one rearranges the already fitted model, to enable the possibility of the contrast usage. In both cases, `eBayes` is used for the F-Tests.

Value

MArrayLM object with attached result.

See Also

edgeR: `glmFit`
`glmLRT`
`glmQLFit`
`glmQLFTest`
limma: `contrasts.fit`
`eBayes`
DESeq2: Wald `nbinomWaldTest`
LRT `nbinomLRT`

Examples

```
requireNamespace("limma")
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initLimma(analysisInfo)
dgelist = estimateSizeFactorsLimma(dgelist)
dgelist = estimateDispersionsLimma(dgelist)
mArrayLM = fitModelTestLimma(dgelist)
```

GeneticAnalysisPipeline

GeneticAnalysisPipeline: A package which includes an easy to use pipeline for differentially gene expression analysis

---

Description

The functions could be used interactively via R console, or through a command-line interface (abbr.: CLI).

Command-Line Interface

Scripts could be found in library directory under `./extscript/cl`.

R's interactive Console

Example workflow could be found in library directory under `./extscript/tcgaMelanoma.R`, or `./extscript/tcgaMelanoma.R`.

Typical Workflow

1. Create `link{AnalysisDataSet}` with `link{createAnalysisDataSet}` and store assembled clinical (phenotypic) and genetic data.
1.1 (optional) Generate plots with `link{plotNA}`, `link{plotBar2dFacet}`, `link{plotBoxplot}`.
2. Create `link{AnalysisInfo}` with `link{createAnalysisInfo}`.
2.1 (optional) Create plots using functions from 1.1 on the shrinked data, or using `link{plotAnalysisInfo}` function.
3. (optional) Use Quality Assurance and Feature Detection methods to qualify the experimental data, using `link{runQualityAssurance}` function together with `link{plotQualityAssurance}`.
4. (optional) Filter count data using `link{filterCountData}` function.
5. (optional) Loop between step 2., 3. and 4., until optimal data setup is approved.

6. Start Differentially Expression Analysis with link{runDifferentialExpressionAnalysis} function.
6.1 (optional) Stop by estimated dispersion and use link{plotPrioriObjects} for QA, loop between 2 and 6.1.
7. (optional) Plot results, using link{plotPrioriObjects}, link{plotPosteriorObjects}.
8. Create result tables using link{saveResults}.
9. (optional) Make a comparison between packages:
9.1 Join results using link{joinResults} method.
9.2 Plot Venn diagram with link{plotComparison}.
10. (optional) Make comparison to references, using link{compareResultsToRef} functionality.

`initDESeq2` *Initial state of* `DESeq2` *package.*

Description

This function is used as wrapper for the first possible step of all used differential expression analysis packages used in `GeneticAnalysisPipeline`. It generates a package typical container including sample informations and count data.

Usage

```
initDESeq2(analysisInfo, verbose = 2, ...)
```

Arguments

`analysisInfo`	Object from `AnalysisInfo` class.
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

`DESeqDataSet` object.

See Also

DESeq2: `DESeqDataSetFromMatrix`
edgeR, limma: `DGEList`

Examples

```
# Create count data using DESeq2
dds = DESeq2::makeExampleDESeqDataSet()
countData = SummarizedExperiment::assay(dds)
sampleData = SummarizedExperiment::colData(dds)
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
```

```
envir = analysisDataSet)
initDESeq2(analysisInfo)
```

`initEdgeR` *Initial state of* `edgeR` *package.*

Description

This function is used as wrapper for the first possible step of all used differential expression analysis packages used in `GeneticAnalysisPipeline`. It generates a package typical container including sample informations and count data.

Usage

```
initEdgeR(analysisInfo, verbose = 2, ...)
```

Arguments

`analysisInfo`	Object from `AnalysisInfo` class.
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

DGEList object.

See Also

edgeR, limma: `DGEList`
DESeq2: `DESeqDataSetFromMatrix`

Examples

```
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
initEdgeR(analysisInfo)
```

initLimma *Initial state of* limma *package.*

Description

This function is used as wrapper for the first possible step of all used differential expression analysis packages used in GeneticAnalysisPipeline. It generates a package typical container including sample informations and count data.

Usage

```
initLimma(analysisInfo, verbose = 2, ...)
```

Arguments

analysisInfo	Object from AnalysisInfo class.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

edgeR, limma: DGEList
DESeq2: DESeqDataSetFromMatrix

Examples

```
requireNamespace("limma")
requireNamespace("edgeR")
countData = matrix(rnbinom(1e04, mu=5, size=2), ncol=4)
# Ensure that we have an integer matrix
storage.mode(countData) = "integer"
sampleData = data.frame(group = factor(rep(LETTERS[1:2], each = 2)))
# Create AnalysisDataSet
analysisDataSet = AnalysisDataSetFromMatrix(countData = countData,
sampleData = sampleData)
# Create AnalysisInfo
analysisInfo = AnalysisInfo(time = date(),
name = "test",
formula = ~ 1,
envir = analysisDataSet)
dgelist = initLimma(analysisInfo)
```

joinResults *Joining result tables stored in a* AnalysisResult *object.*

Description

This function could be used for comparison between packages.

Usage

```
joinResults(listAnalysisResult, separator = "\t", prefixSeparator = "_",
  joinMethod = c("outer", "inner"), tableNA = "", export = TRUE,
  pathTables = paste0(".", .Platform$file.sep), file_extension = ".tsv",
  verbose = 2, ...)
```

Arguments

listAnalysisResult	A vector or list with AnalysisResult objects and attached result(s).
separator	Delimiter for columns used in resulting file.
prefixSeparator	Separator, which should be used between name of package and column label.
joinMethod	Method(s) which should be used for joining.
tableNA	Value for not existing values (outer join).
export	Boolean if table should be written to hard drive.
pathTables	Path to directory, which should be used for export.
file_extension	File extension (should fit to separator).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

Vector with joined results.

See Also

runDifferentialExpressionAnalysis, AnalysisResult

---

normalizeResultsDESeq2

Normalize DESeq2 *results.*

Description

This function could be used to normalize outputs, to keep differences between packages minimal. Target is to make synonyms and style uniform.

Usage

```
normalizeResultsDESeq2(result, alpha, resSortBy = c("padj"),
  resSortDesc = FALSE, verbose = 1, ...)
```

Arguments

result	Object from DESeqDataSet class.
alpha	If given, results will be filtered above or equal value.
resSortBy	Name of column(s) which should be used for the sort.
resSortDesc	If true, results will be sorted descending.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

Result as data.frame object.

---

normalizeResultsEdgeR *Normalize* edgeR *results.*

Description

This function could be used to normalize outputs, to keep differences between packages minimal. Target is to make synonyms and style uniform.

Usage

```
normalizeResultsEdgeR(result, alpha, resSortBy = c("padj"),
  resSortDesc = FALSE, verbose = 1, ...)
```

Arguments

result	Object from DESeqDataSet class.
alpha	If given, results will be filtered above or equal value.
resSortBy	Name of column(s) which should be used for the sort.
resSortDesc	If true, results will be sorted descending.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

Result as data.frame object.

---

normalizeResultsLimma *Normalize* limma *results.*

Description

This function could be used to normalize outputs, to keep differences between packages minimal. Target is to make synonyms and style uniform.

Usage

```
normalizeResultsLimma(result, alpha, resSortBy = c("padj"),
  resSortDesc = FALSE, verbose = 1, ...)
```

Arguments

result	Object from DESeqDataSet class.
alpha	If given, results will be filtered above or equal value.
resSortBy	Name of column(s) which should be used for the sort.
resSortDesc	If true, results will be sorted descending.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

Result as data.frame object.

---

plotAnalysisInfo *Plot Analysis Information Could be used after* link{makeAnalysisInfo}.

Description

Plot Analysis Information

Could be used after link{makeAnalysisInfo}.

Usage

```
plotAnalysisInfo(analysisInfo, plots = c("na", "group", "libsize"),
  addArguments, path = paste0(".", .Platform$file.sep),
  file_extension = ".svg", export_function = svg, verbose = 2, ...)
```

Arguments

analysisInfo	Object from AnalysisInfo class.
plots	Plot(s) which should be generated.
addArguments	List with additional arguments for plot depending functions. Example: list("plot" = alist("parameter" = "value"))
path	Path to directory, which should be used for export.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

AnalysisInfo, plotNA which is used for "na", plotBar2dFacet which is used for "group", plotLibSizeDist which is used for "libsize"

---

plotBar2dFacet *Wrapper to generate Barplots.*

Description

plotBar2dFacet could be used to generate multiple conditional barplots at once, using a main factor (x) and a formula (facets). This function kindly wraps ggplot with geom_bar and optional facet_wrap. Individual coloring is made possible with the functions scale_fill_manual and scale_fill_brewer.

Usage

```
plotBar2dFacet(data, x, facets, addCountLabels = FALSE, simpleColor = TRUE,
  ownColor, colorBrewerPalette = c("Accent", "Dark2", "Paired", "Pastel1",
  "Pastel2", "Set1", "Set2", "Set3"), theme = c("gray", "bw", "linedraw",
  "light", "dark", "minimal", "classic", "void"), export = TRUE,
  path = paste0(".", .Platform$file.sep), filename = paste0("barplot",
  file_extension), file_extension = ".svg", export_function = svg)
```

Arguments

data	A data.frame including the wanted factors.
x	String vector with the name of the factorial column for bottom x scale.
facets	Optional formula or vector with factor(s) for top x scale.
addCountLabels	Boolean value. A TRUE attaches count labels on bars.
simpleColor	Optional logical value to switch standardly colored bars on or off.
ownColor	Optional vector with colors to corresponding bars.
colorBrewerPalette	Optional character vector for RColorBrewer palette.
theme	Optional ggplot2 theme. See package for more information.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).

See Also

ggplot, geom_bar

Examples

```
# Create exemplary sample data
gender = c(rep("male", times = 3), rep("female", times = 7))
group = c(rep(month.abb[1:2], 5))
data = data.frame(gender = gender, group = group)

plotBar2dFacet(data = data,
x = "gender",
facets = ~ gender + group,
colorBrewerPalette = "Dark2")
```

plotBoxplot *Wrapper for Boxplots generation.*

Description

plotBoxplot could be used to generate multiple boxplots at once, using a factorial (x) and a numeric (y) attribute. This function kindly wraps ggplot with geom_boxplot. SimpleColoring could be chosen, if only the borders of the boxplots should be painted. Filling with individual colors is made possible through the functions scale_fill_manual and scale_fill_brewer.

Usage

```
plotBoxplot(data, x, y, simpleColor = TRUE, ownColor,
  colorBrewerPalette = c("Accent", "Dark2", "Paired", "Pastel1", "Pastel2",
  "Set1", "Set2", "Set3"), theme = c("gray", "bw", "linedraw", "light",
  "dark", "minimal", "classic", "void"), export = TRUE, path = paste0(".",
  .Platform$file.sep), filename = paste0("boxplot", file_extension),
  file_extension = ".svg", export_function = svg)
```

Arguments

data	A data.frame including the wanted factors.
x	String vector with the name of the factorial column for bottom x scale.
y	String vector with the name of the numerical column for bottom y scale.
simpleColor	Optional logical value to switch standardly colored boxborders on or off.
ownColor	Optional vector with colors to corresponding boxplots (fill).
colorBrewerPalette	Optional character vector for RColorBrewer palette (fill).
theme	Optional ggplot2 theme. See package for more information.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).

See Also

ggplot, geom_boxplot

Examples

```
# Create exemplary sample data
xData = c(rep("male", times = 3), rep("female", times = 7))
set.seed(100)
yData = runif(10)
sampleData = data.frame(xData, yData)

plotBoxplot(data = sampleData,
```

```
x = "xData",
y = "yData",
colorBrewerPalette = "Dark2")
```

`plotComparison` *Plot results from package comparison.*

Description

This function makes it possible to compare the results between packages on a visual way.

Usage

```
plotComparison(listAnalysisResult, path = paste0(".", .Platform$file.sep),
  plots = c("venn"), file_extension = ".svg", export_function = svg,
  verbose = 2, ...)
```

Arguments

`listAnalysisResult`	A `vector` or `list` with `AnalysisResult` objects and attached posteriori lists.
`path`	Path to directory, which should be used for export.
`plots`	Vector including the names of the plots, which should be generated.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

`runDifferentialExpressionAnalysis`, `AnalysisResult`

`plotHeatMapDist` *Plots heatmap according to gene expression distance*

Description

Wrapper for `pheatmap`. Calculate distance between columns in matrix and use output pheatmap. SVG not supported, so PDF is used instead.

Usage

```
plotHeatMapDist(countData, sampleData, method = c("euclidean", "maximum",
  "manhattan", "canberra", "binary", "minkowski"),
  colorBrewerPalette = c("Blues", "BuGn", "BuPu", "GnBu", "Greens", "Greys",
  "Oranges", "OrRd", "PuBu", "PuBuGn", "PuRd", "Purples", "RdPu", "Reds",
  "YlGn", "YlGnBu", "YlOrBr", "YlOrRd"), export = TRUE, path = paste0(".",
  .Platform$file.sep), filename = paste0("pheatmap_distance", file_extension),
  file_extension = ".pdf")
```

Arguments

countData	A data.frame including the wanted factors.
sampleData	A data.frame including the phenotypic data.
method	The method which should be used for distance calculation.
colorBrewerPalette	Optional character vector for RColorBrewer palette.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension.

See Also

pheatmap, dist

Examples

```
# If you want to see a specific RColorBrewerPalette,
# you can use for example the command: display.brewer.pal(9,"Greens")
```

---

plotHeatMapExp *Plots heatmap according to gene mean expression*

Description

Wrapper for pheatmap. SVG not supported, so PDF is used instead.

Usage

```
plotHeatMapExp(countData, sampleData, group, descending = TRUE,
  export = TRUE, path = paste0(".", .Platform$file.sep),
  filename = paste0(ifelse(descending, "pheatmap_HighExpression",
  "pheatmap_LowExpression"), file_extension), file_extension = ".pdf")
```

Arguments

countData	A data.frame including the wanted factors.
sampleData	A data.frame including the phenotypic data.
group	A factor describing the group.
descending	Top 100 highest, or lowest expressed gene (per mean).
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension.

See Also

pheatmap

plotLibSizeDist *Plot lib size in millions per library.*

Description

Plot lib size in millions per library.

Usage

```
plotLibSizeDist(countData, group, distMedianCnt, rotateX = TRUE,
  simpleColor = TRUE, ownColor, colorBrewerPalette = c("Accent", "Dark2",
  "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3"), theme = c("gray",
  "bw", "linedraw", "light", "dark", "minimal", "classic", "void"),
  addCountLabels = FALSE, title = "Library size per million",
  xlab = "Library", ylab = "Library size ppm", export = TRUE,
  path = paste0(".", .Platform$file.sep),
  filename = paste0("barplot_libsize_ppm", file_extension),
  file_extension = ".svg", export_function = svg)
```

Arguments

countData	A data.frame including the wanted factors.
group	A factor describing the group.
distMedianCnt	Integer value that, if set, shrinks the result by the top distMedianCnt higest distances to median over all libraries.
rotateX	Switches between rotation of x labels.
simpleColor	Optional logical value to switch standardly colored bars on or off.
ownColor	Optional vector with colors to corresponding bars.
colorBrewerPalette	Optional character vector for RColorBrewer palette.
theme	Optional ggplot2 theme. See package for more information.
addCountLabels	Boolean. Adds count labels on top of bars if true.
title	Title from plot.
xlab	X label from plot.
ylab	Y label from plot.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function
export_function	The function, which should be used for the export (e.g. svg).

Value

See ggplot.

See Also

ggplot

plotMDS *Plot gene expression distances between libraries*

Description

Wrapper for plotMDS.

Usage

```
plotMDS(countData, group, export = TRUE, path = paste0(".",
  .Platform$file.sep), filename = paste0("MDSplot", file_extension),
  file_extension = ".svg", export_function = svg)
```

Arguments

countData	A data.frame including the wanted factors.
group	A factor describing the group.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).

See Also

plotMDS

plotMeanSd *Plots standard deviation (variance) against mean.*

Description

Wrapper for meanSdPlot.

Usage

```
plotMeanSd(countData, theme = c("gray", "bw", "linedraw", "light", "dark",
  "minimal", "classic", "void"), export = TRUE, path = paste0(".",
  .Platform$file.sep), filename = paste0("meanSdplot", file_extension),
  file_extension = ".svg", export_function = svg)
```

Arguments

countData	A data.frame including the wanted factors.
theme	Optional ggplot2 theme. See package for more information.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).

See Also

meanSdPlot

---

plotNA *plotNA is generating a barplot, showing the amount of NA values per column from the supported dataframe.*

Description

plotNA is generating a barplot, showing the amount of NA values per column from the supported dataframe.

Usage

```
plotNA(x, export = TRUE, path = paste0(".", .Platform$file.sep),
  filename = paste0("barplot_NA", file_extension), file_extension = ".svg",
  export_function = svg)
```

Arguments

x	A data.frame including the NA values, which should be measured.
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function and will be ignored if filename is given.
export_function	The function, which should be used for the export (e.g. svg).

Value

See barplot.

See Also

barplot

Examples

```
# Create exemplary sample data
sampleData = c(77, 1, 2, 3, NA, 4, NA, 9)
sampleData = data.frame(sampleData)
plotNA(sampleData, export = FALSE)
```

`plotPCA` *DESeq2's plotPCA*

Description

The basis of this code is taken from plotPCA at `plotPCA`. Sorts genes by variance for Principal Component Analysis.

Usage

```
plotPCA(countData, group, ntop = 500, theme = c("gray", "bw", "linedraw",
  "light", "dark", "minimal", "classic", "void"), export = TRUE,
  path = paste0(".", .Platform$file.sep), filename = paste0("PCAplot",
  file_extension), file_extension = ".svg", export_function = svg)
```

Arguments

`countData`	A data.frame including the wanted factors.
`group`	A factor describing the group.
`ntop`	Maximal count of genes, which should be used for PCA calculations. Genes will be sorted internal from highest variance to lowest.
`theme`	Optional `ggplot2` theme. See package for more information.
`export`	Boolean value, indicating if the plot should be only shown or exported.
`path`	Path to directory, which should be used for export.
`filename`	Name of the file, which should be exported.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).

See Also

`plotPCA`

plotPosterioriDESeq2 *Plot posteriori* DESeq2 *objects.*

Description

Most packages serving plots for fitted model and similar. To enable this functionality, necessary objects could be stored in `AnalysisResult` in the posteriori list.

Usage

```
plotPosterioriDESeq2(lPostObj, result, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

`lPostObj`	List with posteriori object.
`result`	Result data.table object.
`path`	Path to directory, which should be used for export.
`export`	Boolean value, indicating if the plot should be only shown or exported.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

plotPosterioriEdgeR *Plot posteriori* edgeR *objects.*

Description

Most packages serving plots for fitted model and similar. To enable this functionality, necessary objects could be stored in `AnalysisResult` in the posteriori list.

Usage

```
plotPosterioriEdgeR(lPostObj, result, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

`lPostObj`	List with posteriori object.
`result`	Result data.table object.
`path`	Path to directory, which should be used for export.
`export`	Boolean value, indicating if the plot should be only shown or exported.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

plotPosterioriLimma *Plot posteriori* limma *objects.*

Description

Most packages serving plots for fitted model and similar. To enable this functionality, necessary objects could be stored in AnalysisResult in the posteriori list.

Usage

```
plotPosterioriLimma(lPostObj, result, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

lPostObj	List with posteriori object.
result	Result data.table object.
path	Path to directory, which should be used for export.
export	Boolean value, indicating if the plot should be only shown or exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

plotPosterioriObjects *Plot package depending posteriori information*

Description

Quality Assurance using objects stored in posteriori list in a AnalysisResult object.

Usage

```
plotPosterioriObjects(listAnalysisResult, path = paste0(".",
  .Platform$file.sep), file_extension = ".svg", export_function = svg,
  verbose = 2, ...)
```

Arguments

listAnalysisResult	A vector or list with AnalysisResult objects and attached posteriori lists.
path	Path to directory, which should be used for export.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

runDifferentialExpressionAnalysis, AnalysisResult

---

plotPrioriDESeq2 *Plot priori* DESeq2 *objects.*

Description

Most packages serving plots for dispersion estimation and similar. To enable this functionality, necessary objects could be stored in AnalysisResult in the priori list.

Usage

```
plotPrioriDESeq2(lPrioriObj, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

lPrioriObj	List with priori object.
path	Path to directory, which should be used for export.
export	Boolean value, indicating if the plot should be only shown or exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

---

plotPrioriEdgeR *Plot priori* edgeR *objects.*

Description

Most packages serving plots for dispersion estimation and similar. To enable this functionality, necessary objects could be stored in AnalysisResult in the priori list.

Usage

```
plotPrioriEdgeR(lPrioriObj, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

lPrioriObj	List with priori object.
path	Path to directory, which should be used for export.
export	Boolean value, indicating if the plot should be only shown or exported.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

plotPrioriLimma *Plot priori* limma *objects.*

Description

Most packages serving plots for dispersion estimation and similar. To enable this functionality, necessary objects could be stored in `AnalysisResult` in the priori list.

Usage

```
plotPrioriLimma(lPrioriObj, path = paste0(".", .Platform$file.sep),
  export = TRUE, file_extension = ".svg", export_function = svg,
  verbose = 1)
```

Arguments

`lPrioriObj`	List with priori object.
`path`	Path to directory, which should be used for export.
`export`	Boolean value, indicating if the plot should be only shown or exported.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

plotPrioriObjects *Plot package depending priori information*

Description

Quality Assurance using objects stored in priori list in a `AnalysisResult` object.

Usage

```
plotPrioriObjects(listAnalysisResult, path = paste0(".", .Platform$file.sep),
  file_extension = ".svg", export_function = svg, verbose = 2, ...)
```

Arguments

`listAnalysisResult`	A `vector` or `list` with `AnalysisResult` objects and attached priori lists.
`path`	Path to directory, which should be used for export.
`file_extension`	File extension (should fit to `export_function`).
`export_function`	The function, which should be used for the export (e.g. svg).
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

`runDifferentialExpressionAnalysis`, `AnalysisResult`

plotQualityAssurance *Plot Quality Assurance information*

Description

Quality Assurance of (typically) normalized genetic count data. For more information about the used plots, please visit the links shown at the bottom of this description.

Usage

```
plotQualityAssurance(listQA, sampleData, group, plots = c("sd", "mds", "pca",
  "hmExpDesc", "hmExpAsc", "hmDist"), addArguments, path = paste0(".",
  .Platform$file.sep), file_extension = ".svg", export_function = svg,
  verbose = 2, ...)
```

Arguments

listQA	A vector or list with QualityAssuranceResult objects.
sampleData	sampleData from corresponding AnalysisDataSet object.
group	group from corresponding AnalysisInfo object.
plots	Plot(s) which should be generated.
addArguments	List with additional arguments for plot depending functions. Example: list("plot" = alist("parameter" = "value"))
path	Path to directory, which should be used for export.
file_extension	File extension (should fit to export_function).
export_function	The function, which should be used for the export (e.g. svg).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

runQualityAssurance, AnalysisInfo, plotMeanSd, plotMDS, plotPCA, plotHeatMapExp, plotHeatMapDist

plotVennDiagram *Plots a venn diagram.*

Description

For additional parameters see venn.diagram.

Usage

```
plotVennDiagram(universe, export = TRUE, path = paste0(".",
  .Platform$file.sep), filename = paste0("vennDiagram", file_extension),
  file_extension = ".svg", export_function = svg, fill = c("steelblue",
  "seagreen", "firebrick1", "cornflowerblue"), col = "transparent",
  alpha = 0.2, fontfamily = "serif", fontface = "bold", cex = 4,
  cat.cex = 2.5, cat.fontfamily = "serif", ...)
```

Arguments

universe	List with package labels as names and id's as value. For example: list("edgeR" = c("ID1, "ID2"), "DESeq2" = c("ID2"), ...)
export	Boolean value, indicating if the plot should be only shown or exported.
path	Path to directory, which should be used for export.
filename	Name of the file, which should be exported.
file_extension	File extension (should fit to export_function
export_function	The function, which should be used for the export (e.g. svg).

See Also

venn.diagram

QualityAssuranceResult-class

QualityAssuranceResult class and constructors

Description

QualityAssuranceResult acts as container for processed quality assurance informations by using GeneticAnalysisPipeline. It simply stores the name from a used method together with the transformed count data.

Usage

```
QualityAssuranceResult(method, result, ...)

method(object)

method(object, ...) <- value

## S4 method for signature 'character,matrix'
QualityAssuranceResult(method, result, ...)

## S4 method for signature 'QualityAssuranceResult'
method(object)

## S4 replacement method for signature 'QualityAssuranceResult,character'
method(object) <- value

## S4 method for signature 'QualityAssuranceResult'
result(object)

## S4 replacement method for signature 'QualityAssuranceResult,matrix'
result(object) <- value

## S4 method for signature 'QualityAssuranceResult'
show(object)
```

Slots

`method` Character vector telling the used method for normalization.

`result` Numeric matrix with normalized counts.

Examples

```
countData = matrix(1:100, ncol = 4, dimnames = list(1:25, 1:4))
logCount = log2(countData)
qaResult = QualityAssuranceResult(method = "log2",
result = logCount,
)
```

---

runDifferentialExpressionAnalysis

Differential Expression Analysis

Description

This function is the entrance to all attached DEA packages form this pipeline. Previous steps that should, at minimum, have been done already:

- link{createAnalysisDataSet}
- link{createAnalysisInfo}
- link{filterCountData}

Usage

```
runDifferentialExpressionAnalysis(analysisInfo, listAnalysisResult,
  packages = c("DESeq2", "edgeR", "limma"), addArguments,
  isPrioriOnly = FALSE, alpha = 0.05, resSortBy = "padj",
  resSortDesc = FALSE, verbose = 2, ...)
```

Arguments

`analysisInfo`	Object from `AnalysisInfo` class.
`listAnalysisResult`	To resume a analysis which has been exited on priori stage.
`packages`	Package(s) which should be used for the analysis.
`addArguments`	List with additional arguments for each package depending functions. Example: list("DESeq2" = alist("estimateDispersions" = alist("method" = "parametric"))).
`isPrioriOnly`	If TRUE, function exits on priori (dispersion is known).
`alpha`	Alpha value for adjusted `p-value`.
`resSortBy`	Name of column(s) which should be used for the sort.
`resSortDesc`	Boolean value for descending ordering.
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Details

This function can be started and left in two different states:

- State 1: A dispersion estimation is stored in every `AnalysisResult` object at `priorObject`. This list could be used to print plots, or to resume the analysis.
- State 2: A dispersion estimation is stored in every `AnalysisResult` object at `priorObject`. Also information about the fitting model process could be stored into the `posteriorObject` and the standardized result table into the `result` field.

Value

A `vector` with `AnalysisResult` objects.

See Also

`AnalysisResult`

`runQualityAssurance` *Make count data ready for quality assurance.*

Description

This functions serves possibilities to normalize counts for Quality Assurance purposes. Besides `raw`, which is left to explore the original counts, the output could be used for clustering and machine learning algorithms.
`raw`:
Counts are left original. This option should be used with caution, because many algorithms and therefore plots will not work accurate.
`log2`:
Common way is to transform counts into log2 scale. Note that this option intensely amplifies small values.
`vst`:
The variance stabilizing transformation from the package `DESeq2` (see `varianceStabilizingTransformation`) is making usage of normalization factors to reduce variance along the mean values (to get homoscedastic data). Library sizes (column sums) will also be used for the transformation.
`rlog`:
`rlogTransformation` coming with the `DESeq2` package (see (see `rlog`). Data will be transformed to log2 scale with respect to library size. If size factors varying widely, this method will be work more robust as `vst`.

Usage

```
runQualityAssurance(x, method = c("raw", "log2", "vst", "rlog"), addArguments,
  verbose = 2, ...)
```

Arguments

`x`	Count matrix, for example from a `AnalysisInfo` object.
`method`	Method(s) that should be used.
`addArguments`	List with additional arguments for package depending functions. Example: list("method" = alist("parameter" = "value"))
`verbose`	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

A vector with QualityAssuranceResult objects. E.g.: list("rlog" = cntMatrix)

See Also

rlog, varianceStabilizingTransformation, AnalysisInfo, AnalysisResult

---

saveResults *Stores result tables from* AnalysisResult *object.*

Description

Stores result tables from AnalysisResult object.

Usage

```
saveResults(listAnalysisResult, separator = "\t", path = paste0(".",
  .Platform$file.sep), file_extension = ".tsv", verbose = 2, ...)
```

Arguments

listAnalysisResult	A vector or list with AnalysisResult objects and attached result.
separator	Delimiter for columns used in resulting file.
path	Path to directory, which should be used for export.
file_extension	File extension (should fit to export_function).
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

See Also

runDifferentialExpressionAnalysis, AnalysisResult

---

serializeAnalysis *Serialize an object from* GeneticAnalysisPipeline *pipeline.*

Description

This function wraps saveRDS and could be used to serialize objects. The file will be saved in binary format and could be compressed.

Usage

```
serializeAnalysis(object, file, compression = c("gzip", "bzip2", "xz"),
  verbose = 2, ...)
```

Arguments

object	Object for serialization.
file	Path to which the file should be saved.
compression	Optional compression method (gzip, bzip, xz.
verbose	Integer value describing the verbose level (silent = 0, warnings = 1, ...)

Value

Boolean status.

See Also

saveRDS, serialize.

---

unserializeAnalysis *Unserialize an object from* GeneticAnalysisPipeline *pipeline.*

Description

This function wraps readRDS and could be used to unserialize objects.

Usage

```
unserializeAnalysis(file, verbose = 2, ...)
```

Arguments

file	Path from which the file should be loaded.
verbose	Set verbosity level.

Value

Boolean status.

See Also

readRDS, unserialize.

Index

AnalysisDataSet, *10*, *11*
AnalysisDataSet (AnalysisDataSet-class), 2
AnalysisDataSet,missing-method (AnalysisDataSet-class), 2
AnalysisDataSet,RangedSummarizedExperiment-method (AnalysisDataSet-class), 2
AnalysisDataSet,SummarizedExperiment-method (AnalysisDataSet-class), 2
AnalysisDataSet,SummarizedExperiment0-method (AnalysisDataSet-class), 2
AnalysisDataSet-class, 2
AnalysisDataSetFromMatrix (AnalysisDataSet-class), 2
AnalysisDataSetFromMatrix,matrix,ANY-method (AnalysisDataSet-class), 2
AnalysisDataSetFromMatrix,matrix,data.frame-method (AnalysisDataSet-class), 2
AnalysisDataSetFromMatrix,matrix,DataFrame-method (AnalysisDataSet-class), 2
AnalysisDataSetFromMatrix,matrix,missing-method (AnalysisDataSet-class), 2
AnalysisInfo, *11*, *36*, *49*, *53*
AnalysisInfo (AnalysisInfo-class), 4
AnalysisInfo,character,character,formula,matrix,missing,list,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,character,character,formula,missing,missing,missing,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,character,missing,formula,matrix,missing,list,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,character,missing,formula,missing,missing,missing,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,missing,character,formula,matrix,missing,list,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,missing,character,formula,missing,missing,missing,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,missing,missing,formula,matrix,missing,list,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo,missing,missing,formula,missing,missing,missing,AnalysisDataSet-method (AnalysisInfo-class), 4
AnalysisInfo-class, 4
AnalysisResult, *9*, *34*, *39*, *47*, *48*, *52*, *53*
AnalysisResult (AnalysisResult-class), 7
AnalysisResult,character,list,list,list-method (AnalysisResult-class), 7
AnalysisResult,character,list,list,missing-method (AnalysisResult-class), 7
AnalysisResult,character,list,missing,missing-method (AnalysisResult-class), 7
AnalysisResult,character,missing,missing,missing-method (AnalysisResult-class), 7
AnalysisResult-class, 7

barplot, *43*

calcNormFactors, *22–24*
compareResultsToRef, 9
Comparison, *11*
contrast (AnalysisInfo-class), 4
contrast,AnalysisInfo-method (AnalysisInfo-class), 4
contrast<- (AnalysisInfo-class), 4
contrast<-,AnalysisInfo,list-method (AnalysisInfo-class), 4
contrasts.fit, *27–29*
countData (AnalysisDataSet-class), 2
countData,AnalysisDataSet-method (AnalysisDataSet-class), 2
countData<- (AnalysisDataSet-class), 2
countData<-,AnalysisDataSet,matrix-method (AnalysisDataSet-class), 2
cpm, *25*
createAnalysisDataSet, *9*
createAnalysisInfo, 11

data.frame, *35*, *36*
decideTests, *13–15*
decideTestsDESeq2, 12
decideTestsDGE, *13*, 15
decideTestsEdgeR, 13
decideTestsLimma, 15
DESeqDataSet, *18*, *22*, *27*, *31*
DESeqDataSetFromMatrix, *31–33*
DESeqResults, *13*
detectDatatype, *9*, *10*, 16
DGEList, *31–33*
dist, *40*

eBayes, *27–29*
envir *(AnalysisInfo-class)*, 4
envir,AnalysisInfo-method *(AnalysisInfo-class)*, 4
envir<- *(AnalysisInfo-class)*, 4
envir<-,AnalysisInfo,AnalysisDataSet-method *(AnalysisInfo-class)*, 4
estimateCommonDisp, *19*
estimateDisp, *18, 19*, *21*
estimateDispersions, *18, 19*, *21*
estimateDispersionsDESeq2, 17
estimateDispersionsEdgeR, 19
estimateDispersionsLimma, 20
estimateGLMCommonDisp, *19*
estimateSizeFactors, *22–24*
estimateSizeFactorsDESeq2, 21
estimateSizeFactorsEdgeR, 22
estimateSizeFactorsForMatrix, *22*
estimateSizeFactorsLimma, 24

facet_wrap, *36*
filterCountData, 25
fitModelTestDESeq2, 26
fitModelTestEdgeR, 27
fitModelTestLimma, 29
formula *(AnalysisInfo-class)*, 4
formula,AnalysisInfo-method *(AnalysisInfo-class)*, 4
formula<- *(AnalysisInfo-class)*, 4
formula<-,AnalysisInfo,formula-method *(AnalysisInfo-class)*, 4
fread, *10*, *17*

GeneticAnalysisPipeline, 30
GeneticAnalysisPipeline-package *(GeneticAnalysisPipeline)*, 30
geom_bar, *36, 37*
geom_boxplot, *38*
ggplot, *36–38*, *41*
glmFit, *27–29*
glmLRT, *27–29*
glmQLFit, *27–29*
glmQLFTest, *27–29*
group *(AnalysisInfo-class)*, 4
group,AnalysisInfo-method *(AnalysisInfo-class)*, 4

initDESeq2, 31
initEdgeR, 32
initLimma, 33

joinResults, 33

meanSdPlot, *42, 43*
method *(QualityAssuranceResult-class)*, 50
method,QualityAssuranceResult-method *(QualityAssuranceResult-class)*, 50
method<- *(QualityAssuranceResult-class)*, 50
method<-,QualityAssuranceResult,character-method *(QualityAssuranceResult-class)*, 50
model *(AnalysisInfo-class)*, 4
model,AnalysisInfo-method *(AnalysisInfo-class)*, 4
model<- *(AnalysisInfo-class)*, 4
model<-,AnalysisInfo,formula-method *(AnalysisInfo-class)*, 4

name *(AnalysisInfo-class)*, 4
name,AnalysisInfo-method *(AnalysisInfo-class)*, 4
name,AnalysisResult-method *(AnalysisResult-class)*, 7
name<- *(AnalysisInfo-class)*, 4
name<-,AnalysisInfo,character-method *(AnalysisInfo-class)*, 4
name<-,AnalysisResult,character-method *(AnalysisResult-class)*, 7
nbinomLRT, *27–29*
nbinomWaldTest, *27–29*
normalizeBetweenArrays, *20*
normalizeResultsDESeq2, 34
normalizeResultsEdgeR, 35
normalizeResultsLimma, 35

p.adjust, *13–15*
pheatmap, *39, 40*
plotAnalysisInfo, 36
plotBar2dFacet, *36*, 36
plotBoxplot, 38
plotComparison, 39
plotHeatMapDist, 39, *49*
plotHeatMapExp, 40, *49*
plotLibSizeDist, *36*, 41
plotMDS, *42*, 42, *49*
plotMeanSd, 42, *49*
plotNA, *36*, 43
plotPCA, *44*, 44, *49*
plotPosterioriDESeq2, 45
plotPosterioriEdgeR, 45
plotPosterioriLimma, 46
plotPosterioriObjects, 46
plotPrioriDESeq2, 47

plotPrioriEdgeR, 47
plotPrioriLimma, 48
plotPrioriObjects, 48
plotQualityAssurance, 49
plotVennDiagram, 49
posterioriObjects
(AnalysisResult-class), 7
posteriorObjects,AnalysisResult-method
(AnalysisResult-class), 7
posterioriObjects<-
(AnalysisResult-class), 7
posterioriObjects<-,AnalysisResult,list-method
(AnalysisResult-class), 7
prioriObjects (AnalysisResult-class), 7
prioriObjects,AnalysisResult-method
(AnalysisResult-class), 7
prioriObjects<- (AnalysisResult-class),
7
prioriObjects<-,AnalysisResult,list-method
(AnalysisResult-class), 7

QualityAssuranceResult
(QualityAssuranceResult-class),
50
QualityAssuranceResult,character,matrix-method
(QualityAssuranceResult-class),
50
QualityAssuranceResult-class, 50

read.table, *17*
readRDS, *54*
replaceOutliers, *27*
result (AnalysisResult-class), 7
result,AnalysisResult-method
(AnalysisResult-class), 7
result,QualityAssuranceResult-method
(QualityAssuranceResult-class),
50
result<- (AnalysisResult-class), 7
result<-,AnalysisResult,list-method
(AnalysisResult-class), 7
result<-,QualityAssuranceResult,matrix-method
(QualityAssuranceResult-class),
50
results, *13–15*
rlog, *52, 53*
runDifferentialExpressionAnalysis, *9*,
34, *39*, *47, 48*, 51, *53*
runQualityAssurance, *49*, 52

sampleData (AnalysisDataSet-class), 2
sampleData,AnalysisDataSet-method
(AnalysisDataSet-class), 2
sampleData<- (AnalysisDataSet-class), 2
sampleData<-,AnalysisDataSet,data.frame-method
(AnalysisDataSet-class), 2
sampleData<-,AnalysisDataSet,DataFrame-method
(AnalysisDataSet-class), 2
saveRDS, *53, 54*
saveResults, 53
scale_fill_brewer, *36*, *38*
scale_fill_manual, *36*, *38*
serialize, *54*
serializeAnalysis, 53
show,AnalysisInfo-method
(AnalysisInfo-class), 4
show,AnalysisResult-method
(AnalysisResult-class), 7
show,QualityAssuranceResult-method
(QualityAssuranceResult-class),
50

TestResults, *14, 15*
time (AnalysisInfo-class), 4
time,AnalysisInfo-method
(AnalysisInfo-class), 4
time<- (AnalysisInfo-class), 4
time<-,AnalysisInfo,character-method
(AnalysisInfo-class), 4

unserialize, *54*
unserializeAnalysis, 54

varianceStabilizingTransformation, *52,*
53
venn.diagram, *49, 50*
voom, *18, 19*, *21*