LIMMAS: Linear models for mass spectrometry

Thomas Schwarzl[1,2], Elisa D Arcangelo[1], Norma Bargary[1], Matthias Malovits[1], and Desmond G Higgins[1]

[1] *Systems Biology Ireland, University College Dublin, Conway Institute, Dublin, Ireland*
[2] *European Molecular Biology Laboratory, Meyerhofstrasse 1, Heidelberg, Germany*

June 13, 2014

Contents

1 Introduction

Quantitative mass spectrometry (MS) allows the analysis of quantitative high-throughput proteomics data. For each protein analysed, a mass spectrum is generated which allows protein identification and assessment of protein abundance levels. Despite the remarkable power of MS technology, finding quantitative differences among proteins levels in distinct physiological conditions remains a challenging task (Bantscheff et al, 2007). This is especially true for data sets characterised by a high level of missing values, which is a characteristic frequently seen in MS output data sets (Ref). Statistical software used for the identification of differentially expressed proteins typically require the data to be complete. Methods for dealing with missing data are usually inappropriate in that substantial amounts of information is lost and the data distribution is skewed; other, more efficient techniques present themselves as complicated to implement and data set-specific (Ref, http://sites.stat.psu.edu/ jls/mifaq.html). Multiple imputation (MI) emerges as an interesting, easy to implement alternative for analysing incomplete data (Ref). MI is a Monte Carlo technique which substitutes missing values with m>1 estimated values (m typically amounts to 3-10), thereby producing $m > 1$ complete data sets. Each one of these is individually analysed with the statistical method of choice and the results are then pooled into estimates, whereby missing-data uncertainty is incorporated (Ref: http://sites.stat.psu.edu/ jls/mifaq.html). In our approach, we show how incomplete protein entries can be salvaged using MI and the full amount of MS information subsequently used for differential protein expression analysis. Analysis of differential protein expression is carried out with linear models and empiric Bayes method of the Linear Models for Microarray Data (limma) package (Ref). limma allows to fit a linear model to the expression data of each protein, whereby both simple experiments and experimental designs involving several groups and factors as well as time course experiments can be handled effectively.

First we load the library:

```
> library(limmas)
```

2 Data input

LIMMAS takes an object of class `ExpressionSet`, here called *eset*, as input. Basically, an `ExpressionSet` is an object containing expression values as well as the sample annotation (pheno) and feature annotation (protein or peptide information). Those are stored in slots of an `ExpressionSet`.

The pheno data stored in *eset* can be accessed with `pData(eset)` and should describe the experimental setup. `exprs(eset)` provides the protein intensities, `annotation(eset)` the annotation for the proteins, and `fData(eset)` the meta-information about the quantified proteins.

LIMMAS provides functions to create an object of ExpressionSets from tab-separated files or `data.frames` from any software for MS quantification.

Currently, functions for the processing of output files by the popular software `MaxQuant` are provided. LIMMAS directly reads *proteinGroups.txt* files as input, if the appropriate pheno data is provided manually.

2.1 Example data

To demonstrate our approach, we apply the functions in the LIMMAS package to a label-free quantitative (LFQ) MS data comparing the protein abundance of immunoprecipitated GEFH1 binding partners. 2 technical and 3 biological replicates each of sample groups were generated. The groups are: (1) antibody control, (2) GEFH1, and (3) GEFH1 + Mek Inhibitor. These were quantified using `MaxQuant` software (Ref) and technical replicates were also summed up by `MaxQuant`. Activation of the guanine exchange factor GEFH1 is dependent on the action of Mek (ref). This study aimed at detecting changes in protein binding dependent on the presence of a specific Mek Inhibitor.

The ExpressionSet containing this data can be loaded with

```
> # data(gefh1)
```

If you want to use the example data, you can continue to 3.

2.2 Read in `MaxQuant` data

Firstly, sample information called 'pheno file' specifying the experimental conditions and different factors of the study is needed as `SmartAnnotatedDataFrame`. This is then used to read in the raw protein intensities from the `MaxQuant` output *proteinGroups.txt* file. This will provide an `ExpressionSet` for further analysis.

2.2.1 pheno data

Here we show how to create a pheno file and read in pheno information as `SmartAnnotated-DataFrame`.

Creating a pheno file

A pheno file is a tab-separated text-file where each row describes a sample. Here, the rows start with the sample name, then the column name of the raw data file where the protein intensities can be found. The next column 'groups' describes the sample groups. Any other factors for the analysis can be added, one factor per column.

We call this pheno file "pheno.txt".

```
                  OrigNames             groups
Control.A         LFQ.intensity.ALEX_1  Control
Control.B         LFQ.intensity.ALEX_2  Control
Control.C         LFQ.intensity.ALEX_3  Control
GEFh1.A           LFQ.intensity.ALEX_4  Gefh1
GEFh1.B           LFQ.intensity.ALEX_5  Gefh1
GEFh1.C           LFQ.intensity.ALEX_6  Gefh1
GEFh1.MEKInhib.A  LFQ.intensity.ALEX_7  Gefh1Inhib
GEFh1.MEKInhib.B  LFQ.intensity.ALEX_8  Gefh1Inhib
GEFh1.MEKInhib.C  LFQ.intensity.ALEX_9  Gefh1Inhib
```

Reading from file

The pheno file can be directly read into the workspace by calling

```
> # pheno <- read.pheno("pheno.txt", originalNamesCol="OrigNames", sampleNamesCol="")
```

When reading in the protein intensity raw data, it is often convenient to re-name the samples. LIMMAS provides this function automatically when given original sample names and new sample names for renaming.

originalNamesCol defines the name of the column of the pheno data which keeps the raw sample names. This are the exact column names of the data file which hold the protein intensities and are read in in the next step. In the case of the here described data set, this would be for example “OrigNames”.

sampleNamesCol specifies the name of the column of the pheno data which keeps the new sample names. “” means simply, that the sample names are stored in the rownames.

In short, after reading the data file, the name for sample `LFQ.intensity.ALEX 1` will be `Control.A` in the resulting `ExpressionSet`. Should the user want to keep the original sample names, the same value must be specified in both arguments.

`read.pheno` creates pheno data of the class `SmartAnnotatedDataFrame`.

Reading from data frame

If the pheno information is already stored in a `data.frame`, here *pheno.input*, the following command will create the `SmartAnnotatedDataFrame` object `pheno`.

```
> #pheno <- SmartAnnotatedDataFrame(pheno.input)
```

SmartAnnotatedDataFrame functions

`SmartAnnotatedDataFrame` is a derived class from `AnnotatedDataFrame` and therefore has all functionalities of `AnnotatedDataFrame` objects, as well as additional functions.

An already existing method `pData` displays the raw pheno data

```
> #pData(pheno)
```

Sometimes samples have to be excluded because of quality issues, or because only a subset of the data set is analysed. In the `SmartAnnotatedDataFrame` object, factors will automatically re-level when samples are excluded. If for example, in the example data the control was to be excluded,

```
> #pheno.without.control <- pheno[,-c(1:3)]
```

the factor returned by pData(pheno.without.control)[,“groups”] would contain two, not three levels

```
> #pData(pheno.without.control)[,``groups'']
```

When successfully created an `SmartAnnotatedDataFrame` following functions will work:

`getOriginalNames` returns all original sample names.

```
> #getOriginalNames(pheno)
```

`getSampleNames` returns all new sample names.

```
> #getSampleNames(pheno)
```

getAnnotatedDataFrame returns an AnnotatedDataFrame without the original sample name column and the new samples as rownames.

```
> #getAnnotatedDataFrame(pheno))
```

2.2.2 protein intensity data

Given the SmartAnnotatedDataFrame, MaxQuant intensity data can easily read in.

From proteinGroups.txt file

The MaxQuant text output file proteinGroups.txt contains the intensities on protein level and a lot of meta information about the quantification process. read.maxQuant takes an SmartAnnotatedDataFrame and reads in the samples specified in the SmartAnnotatedDataFrame. It returns the ExpressionSet object we need for analysis.

```
> #data <- read.maxQuant(file= 'maxQuantFile.txt', pheno, splitIds= ';')
```

From data frame

The function createExpressionSetFromMaxQuant processes MaxQuant output and provides an object of the class ExpressionSet as utilised in the package *Biobase* (Ref). The function splits the data.frame in the tab-separated data file using the original sample name columns from the SmartAnnotatedDataFrame object as protein intensities in exprs(data). It assigns the sample names defined in pheno as column name of the intensities. Via the parameter protein annotation is retrieved from the data and stored in annotation(data). The remaining input data is stored in fData(data). This can contain meta-information about peptide counts, contaminants, etc.

In summary, the features of data can be viewed as follows:

- exprs(data) returns the protein intensity values
- pData(data) returns the corresponding pheno data
- annotation(data) views the protein ids
- fData(data) views other features included in the raw input experimental data

2.3 Other sources

For sources different from MaxQuant, ExpressionSet objects can be created easily with the provided function

```
> #data <- createExpressionSet(intensities, pheno, features, annotation)
```

- intensities is an expression matrix of protein intensities.
- pheno can be either an AnnotatedDataFrame or SmartAnnotatedDataFrame.

- `features` is `matrix` for feature data.
- `annotation` is a `character` string containing the protein annotation.

For manual creation of an `ExpressionSet`, please refer to the `ExpressionSet` manual pages.

3 Filtering

Three filtering steps strip the raw data from unwanted protein entries, including proteins identified on the basis of only one peptide fragment and positive protein hits in reverse- and contaminant databases. This information should be found in `fData(data)`.

3.1 Peptide Filter

`peptideFilter` represents the first filtering step. This function removes all protein entries of an object of the class `ExpressionSet`, which are characterised by a peptide count smaller than or equal to a user-defined cut-off in the column named "peptides".

```
> #dim(data)
> #data<-peptideFilter(data)
> #dim(data)
```

The default value for the parameter `peptideCutoff` within the function is 1. The amount of protein entries after filtering can easily be checked using `dim(data)`.

3.2 Reverse Filter

`reverseFilter` is a function responsible for the removal of all proteins shown to be positive in the reverse peptide database of choice. This is annotated in the data set with e.g. a '+' symbol in the column 'reverse'. Should the original data set contain a different symbol to indicate positive hits, the user is required to change the symbol description with the parameter 'symbol' within the function.

```
> #dim(data)
> #data<-reverseFilter(data)
> #dim(data)
```

3.3 Contaminant Filter

Finally, the `contaminantFilter` function removes proteins contained in the chosen contaminant database. The same remark concerning the symbol used to indicate positive hits in the column 'contaminant' is used as above.

```
> #dim(data)
> #data<-contaminantFilter(data)
> #dim(data)
```

4 Check missingness

Of great interest for the user and for the purpose of MI is the degree of missingness across the data set. The percentage of missing data in each sample can be checked using `checkMissingness(data)`:

```
> #checkMissingness(data)
```

The number of missing value-free rows across samples can also be determined by calling `checkCompleteRows(data)`

```
> #checkCompleteRows(data)
```

Missing values within a MS data set are the result of random or systematic detection errors. It is therefore often hard to determine, especially for proteins close to the detection limit, whether an entity is truly missing or simply not recorded.

Knowing how likely any protein entity is missing in a given sample is therefore a useful insight that aids in the decision of pinpointing a threshold for true negative detection. This can be accomplished utilising the function `getMaxProbabilityMissingByChance(data, "groups", 3)`. This command calculates the maximum probability of any number of observed values being missing by chance in all groups of replicates, whereby the argument âĂŸgroupsâĂŹ determines the column in the pheno file that defines individual groups; the number of missing values can be specified with a number <= number of replicates.

```
> #getMaxProbabilityMissingByChance(data, "groups", 3)
```

5 Preprocessing

Before imputation and statistical testing, the data is subjected to normalization, scaling and transformation. It has been observed that normalization is best carried out prior to MI (Ref). Normal data distribution is necessary for imputation and fitting the linear model in a later step. Quantile normalization is set as the default method.

```
> #data.normalized <- normalizeData(data, minIntensity=0)
```

The paramenter minIntensity is arbitrary and allows to set a cut-off for protein intensity values, below which the user deems the observed values to be inaccurate, e.g. due to known irregularities of the mass spectrometer. Alternative Normalization methods can be used and suggested functions for ExpressionSet objects are described in the package affyPLM (Ref) and include:

- normalize.ExpressionSet.quantiles

- normalize.ExpressionSet.loess
- normalize.ExpressionSet.contrasts
- normalize.ExpressionSet.qspline
- normalize.ExpressionSet.scaling

The above functions can be employed by passing them into the `normalizeData` function call with the parameter The function `scaleData` scales down the protein intensity values by dividing them by a user-defined scalefactor argument (here chosen to be 1000).

```
> #data.scaled <- scaleData(data.normalized, 1000)
```

Normal distribution is achieved via a final step of log-transformation, which is widely used in mass spectrometry data (ref). Other transformations can be passed as functions into the `transformData` function if needed with the parameter xxxx.

```
> #data.transformed <- transformData(data.scaled)
```

Normality is checked using normal quantile-quantile (Q-Q) plots. A 45 degree diagonal from left bottom to top right would mean a perfectly normal distributed sample.

```
> #par(mfrow=c(3,3))
> #for(i in 1:9) {
> #  qqnorm(exprs(data.transformed)[,i], main = paste("Normal Q-Q Plot: ",
> #  colnames(exprs(data.transformed))[i], sep=""))
> #}
```

6 Imputation

MI of the incomplete data set is carried out utilising the function `imputeIndependentGroupsWithAmelia`. As described in the *AmeliaII* package (Ref), this command imputes the missing values present in the data set on the basis of the variation of the experimental values recorded, thereby producing a user defined number ($m > 1$) of complete data sets. The default number of imputations is set to $m = 10$.

```
> #imp <- imputeIndependentGroupsWithAmelia(data.transformed, minTotalPresent=2, groupingC
```

`groupingCol` specifies the sets of replicates (groups) across which imputation is to be carried out, as seen in pheno. Importantly, the parameter `minTotalPresent` specifies the number of observed values required by the user in each set of replicates (group). The following rationale underlies this step: For any specific protein and for each group, if *number of observed values* => *minTotalPresent*, then both the present and missing values remain unaltered and can be imputed. If however *number of observed values* < *minTotalPresent*, all values for that protein in that particular group are replaced by NA values and can therefore not be imputed. This approach aids the imputation of likely true positives while hindering the propagation of likely false positive observations.

In the example data, we decided that of the three replicates in each group at least two were required to contain observed data points, in order for the protein intensity values in that specific group to be imputed. Contrarily, for protein entries where only one value out of three was observed in a group and two were missing, the observed value was regarded as a likely false positive and was replaced by NA, thereby completing a full set of three NAs. These data points cannot be imputed.

At this point the in-build *Amelia II* quality checks can be performed. Below is an example of overdispersed starting values diagnostics, specifically for the imputed data in the control group. This diagnostic tool gives some indication as to how sensitive the imputation process is for any particular data set, by visualising how the first and second (as dims=2) principal component change across m imputations (here $m = 10$). If print to screen (*p2s*) is set to 2, a diagnostic output is created. Please refer to the *Amelia II* package description for further details; however, if all lines in the overdispersed starting value plot converge at the same point, it is reflective of a well behaved likelihood of the data.

```
> #disperse(imp$imputation$Control, m=10,dims=2, p2s=2)
```

The resulting object `imp` is a list comprising an object of class MImputedExpressionSets , which is specific to *LIMMAS* and carries the imputed protein expression values, and a list containing further imputation parameters returned by *Amelia*.

```
> #summary(imp)
```

The former can be extracted from imp with this command:

```
> #data.imputed <- imp[[âĂIJdataâĂI■]]
```

The features of data.imputed can be explored as follows: