Package: glydet (via r-universe)

May 17, 2026

Title Describe Glycosylation Structural Properties in a Site Specific Manner

Version 0.11.0

Description Calculate and analyze glycosylation derived traits for site-specific glycoproteomics data. Derived traits are summary features that aggregate individual glycan abundances into biologically meaningful groups (e.g., galactosylation, sialylation, fucosylation, branching). This package provides functions to compute well-established derived traits from the literature, and features a domain-specific language for defining custom derived traits tailored to specific research needs.

License MIT + file LICENSE

Suggests knitr, rmarkdown, testthat (>= 3.0.0), glyexp (>= 0.12.3), glyclean (>= 0.12.0), glystats (>= 0.6.3), missForest, ellmer

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

URL https://glycoverse.github.io/glydet/

Imports checkmate, cli, dplyr, glymotif (>= 0.13.0), glyparse (>= 0.5.3), glyrepr (>= 0.10.0), lifecycle, memoise, purrr, rlang, stringr, tibble, tidyr, tidyselect

Depends R (>= 4.1)

VignetteBuilder knitr

Config/pak/sysreqs cmake libglpk-dev make libicu-dev libuv1-dev libxml2-dev

Repository https://glycoverse.r-universe.dev

Date/Publication 2026-05-04 03:36:51 UTC

RemoteUrl https://github.com/glycoverse/glydet

RemoteRef v0.11.0

RemoteSha 196d23ec9cdf44b8729f563bd33493777ef2b18e

Contents

`add_meta_properties` *Add Meta-Properties to Experiment*

Description

This function adds meta-properties to the variable information of a `glyexp::experiment()`. Under the hood, it uses `get_meta_properties()` to calculate the meta-properties on the "glycan_structure" column (or column specified by `struc_col`) of the variable information tibble, and then adds the result back as new columns.

Usage

```
add_meta_properties(
  exp,
  mp_fns = NULL,
  struc_col = "glycan_structure",
  overwrite = FALSE
)
```

Arguments

exp	An glyexp::experiment() object.
mp_fns	A named list of meta-property functions. Names of the list are the names of the meta-properties. Default is all_mp_fns(). A meta-property function should takes a glyrepr::glycan_structure() vector, and returns a vector of the meta-property values. purrr-style lambda functions are supported.
struc_col	The column name of the glycan structures in the variable information tibble. Default is "glycan_structure".
overwrite	Whether to overwrite the existing meta-property columns. Default is FALSE, raising an error if the existing columns are found.

Value

An glyexp::experiment() object with meta-properties added to the variable information.

See Also

get_meta_properties(), glyexp::experiment()

Examples

```
library(glyexp)

# Compare the columns in the variable information before and after adding meta-properties
exp <- real_experiment |>
  slice_sample_var(n = 10)
colnames(get_var_info(exp))

exp2 <- add_meta_properties(exp)
colnames(get_var_info(exp2))
```

---

all_mp_fns *Get All Meta-Property Functions*

Description

This function returns a named list of all meta-property functions:

- "Tp": n_glycan_type(): type of the glycan
- "B": has_bisecting(): whether the glycan has a bisecting GlcNAc
- "nA": n_antennae(): number of antennae
- "nF": n_fuc(): number of fucoses
- "nFc": n_core_fuc(): number of core fucoses
- "nFa": n_arm_fuc(): number of arm fucoses

- "nG": `n_gal()`: number of galactoses
- "nGt": `n_terminal_gal()`: number of terminal galactoses
- "nS": `n_sia()`: number of sialic acids
- "nM": `n_man()`: number of mannoses

Usage

```
all_mp_fns()
```

Value

A named list of meta-property functions.

`derive_traits` *Calculate Derived Traits*

Description

This function calculates derived traits from a `glyexp::experiment()` object. For glycomics data, it calculates the derived traits directly. For glycoproteomics data, each glycosite is treated as a separate glycome, and derived traits are calculated in a site-specific manner.

Usage

```
derive_traits(exp, trait_fns = NULL, mp_fns = NULL, mp_cols = NULL)
```

Arguments

`exp`	A `glyexp::experiment()` object. Before using this function, you should preprocess the data using the `glyclean` package. For glycoproteomics data, the data should be aggregated to the "gfs" (glycoforms with structures) level using `glyclean::aggregate()`. Also, please make sure that the `glycan_structure` column is present in the `var_info` table, as not all glycoproteomics identification softwares provide this information. "glycan_structure" can be a `glyrepr::glycan_structure()` vector, or a character vector of glycan structure strings supported by `glyparse::auto_parse()`.
`trait_fns`	A named list of derived trait functions created by trait factories. Names of the list are the names of the derived traits. Default is `NULL`, which means all derived traits in `traits_basic()` are calculated.
`mp_fns`	A named list of meta-property functions. This parameter is useful if your trait functions use custom meta-properties other than those in `all_mp_fns()`. Default is `NULL`, which means all meta-properties in `all_mp_fns()` are used.
`mp_cols`	A character vector of column names in the `var_info` tibble to use as meta-properties. If names are provided, they will be used as names of the meta-properties, otherwise the column names will be used. When `mp_cols` is specified, the selected columns overwrite meta-properties introduced by `mp_fns` with the same names, including built-in meta-properties. Default is `NULL`, which

means all columns in `var_info` are available as meta-properties by their existing names. In this default mode, meta-properties introduced by `mp_fns` take precedence over `var_info` columns with the same names.

Value

A new `glyexp::experiment()` object for derived traits. Instead of "quantification of each glycan on each glycosite in each sample", the new `experiment()` contains "the value of each derived trait on each glycosite in each sample", with the following columns in the `var_info` table:

- `variable`: variable ID
- `trait`: derived trait name
- `explanation`: a concise English explanation of the trait

For glycoproteomics data, with additional columns:

- `protein`: protein ID
- `protein_site`: the glycosite position on the protein

Other columns in the `var_info` table (e.g. `gene`) are retained if they have "many-to-one" relationship with glycosites (unique combinations of `protein`, `protein_site`). That is, each glycosite cannot have multiple values for these columns. `gene` is a common example, as a glycosite can only be associate with one gene. Descriptions about glycans are not such a column, as a glycosite can have multiple glycans, thus having multiple descriptions. Columns not having this relationship with glycosites will be dropped. Don't worry if you cannot understand this logic, as long as you know that this function will try its best to preserve useful information.

`sample_info` and `meta_data` are not modified, except that the `exp_type` field of `meta_data` is set to "traitomics" for glycomics data, and "traitproteomics" for glycoproteomics data.

See Also

`traits_basic()`, `traits_detailed()`

Examples

```
library(glyexp)
library(glyclean)

exp <- real_experiment |>
  auto_clean() |>
  slice_sample_var(n = 100)
trait_exp <- derive_traits(exp)
trait_exp

# By default, only basic traits are calculated
names(traits_basic())

# You can calculate detailed traits in `traits_detailed()`
more_trait_exp <- derive_traits(exp, trait_fns = traits_detailed())
more_trait_exp
```

`derive_traits_` *Calculate Derived Traits from Tidy Data*

Description

This function calculates derived traits from a tibble in tidy format. Use this function if you are not using the `glyexp` package. For glycomics data, it calculates the derived traits directly. For glycoproteomics data, each glycosite is treated as a separate glycome, and derived traits are calculated in a site-specific manner.

Usage

```
derive_traits_(tbl, data_type, trait_fns = NULL, mp_fns = NULL)
```

Arguments

`tbl`	A tibble in tidy format, with the following columns: • `sample`: sample ID • `glycan_structure`: glycan structures, either a `glyrepr::glycan_structure()` vector or a character vector of glycan structure strings supported by `glyparse::auto_parse()`. • `value`: the quantification of the glycan in the sample. For glycoproteomics data, additional columns are needed: • `protein`: protein ID • `protein_site`: the glycosite position on the protein The unique combination of `protein` and `protein_site` determines a glycosite. Other columns are ignored. Please make sure that the data has been properly preprocessed, including normalization, missing value handling, etc. Specifically, for glycoproteomics data, please make sure that the data has been aggregated to the "glycoforms with structures" level. That is the quantification of each glycan structure on each glycosite in each sample.
`data_type`	Either "glycomics" or "glycoproteomics".
`trait_fns`	A named list of derived trait functions created by trait factories. Names of the list are the names of the derived traits. Default is `NULL`, which means all derived traits in `traits_basic()` are calculated.
`mp_fns`	A named list of meta-property functions. This parameter is useful if your trait functions use custom meta-properties other than those in `all_mp_fns()`. Default is `NULL`, which means all meta-properties in `all_mp_fns()` are used.

Value

A tidy tibble containing the following columns:

- `sample`: sample ID

- `trait`: derived trait name
- `value`: the value of the derived trait
- `explanation`: a concise English explanation of the trait

For glycoproteomics data, with additional columns:

- `protein`: protein ID
- `protein_site`: the glycosite position on the protein

Other columns in the original tibble are not included.

See Also

`derive_traits()`, `traits_basic()`, `traits_detailed()`

Examples

```
# Create example tidy data
library(dplyr)
library(glyexp)
library(tibble)

tidy_data <- as_tibble(real_experiment2)

# Calculate traits
traits <- derive_traits_(tidy_data, data_type = "glycomics")
traits
```

`explain_trait` *Explain a Derived Trait*

Description

This function provides a human-readable English explanation of what a derived trait represents. It works with trait functions created by the trait factories `prop()`, `ratio()`, `wmean()`, `total()`, and `wsum()`, and works best with traits defined by built-in meta-properties.

Usage

```
explain_trait(
  trait_fn,
  use_ai = FALSE,
  custom_mp = NULL,
  provider = getOption("glydet.ai_provider", "deepseek"),
  model = getOption("glydet.ai_model", NULL),
  api_key = getOption("glydet.ai_api_key", NULL),
  base_url = getOption("glydet.ai_base_url", NULL)
)
```

Arguments

trait_fn	A derived trait function created by one of the trait factories.
use_ai	**[Experimental]** Whether to use a Large Language Model (LLM) to explain the trait. Default is FALSE. To use this feature, you need to install the ellmer package. DeepSeek is used by default for backward compatibility. Other ellmer providers can be selected with provider, model, and provider-specific API key configuration.
custom_mp	A named character vector of custom meta-properties. The names are the meta-property names, and the values are in the format "(type) description". Only used when use_ai = TRUE.
provider	AI provider passed to ellmer when use_ai = TRUE. One of "deepseek", "openai", "anthropic", "gemini", "openrouter", or "openai_compatible". "google_gemini" is accepted as an alias for "gemini". Defaults to getOption("glydet.ai_provider", "deepseek").
model	Model to use when use_ai = TRUE. Defaults to getOption("glydet.ai_model"), or "deepseek-chat" for DeepSeek and the provider default for other providers.
api_key	API key for the selected provider. If NULL, the provider specific environment variable is used. Defaults to getOption("glydet.ai_api_key").
base_url	Optional base URL for custom or OpenAI-compatible endpoints. Defaults to getOption("glydet.ai_base_url").

Value

A character string containing a concise English explanation of the trait.

Examples

```
# Explain built-in traits
explain_trait(traits_basic()$TM)
explain_trait(traits_basic()$GS)

# Explain custom traits
explain_trait(prop(nFc > 0))
explain_trait(prop(nFc > 0, within = (Tp == "complex")))
explain_trait(ratio(Tp == "complex", Tp == "hybrid"))
explain_trait(wmean(nA, within = (Tp == "complex")))
explain_trait(wmean(nS / nG, within = nA == 4 & nFc > 0))

# Explain total and wsum traits
explain_trait(total(Tp == "complex"))
explain_trait(wsum(nS))
explain_trait(wsum(nS, within = (Tp == "complex")))
```

`get_meta_properties` *Get Meta-Properties of Glycans*

Description

This function calculates the meta-properties of the given glycans. Meta-properties are properties describing certain structural characteristics of glycans. For example, the number of antennae, the number of core fucoses, etc.

Usage

```
get_meta_properties(glycans, mp_fns = NULL)
```

Arguments

`glycans`	A `glyrepr::glycan_structure()` vector, or a character vector of glycan structure strings supported by `glyparse::auto_parse()`.
`mp_fns`	A named list of meta-property functions. Names of the list are the names of the meta-properties. Default is `all_mp_fns()`. A meta-property function should takes a `glyrepr::glycan_structure()` vector, and returns a vector of the meta-property values. purrr-style lambda functions are supported.

Value

A tibble with the meta-properties. Column names are the names of the meta-properties.

See Also

`all_mp_fns()`

Examples

```
glycans <- c(
  "Man(a1-3)[Man(a1-6)]Man(b1-4)GlcNAc(b1-4)GlcNAc(b1-",
 "Fuc(a1-3)GlcNAc(b1-2)Man(a1-3)[GlcNAc(b1-2)Man(a1-6)]Man(b1-4)GlcNAc(b1-4)GlcNAc(?1-",
  "GlcNAc(b1-2)Man(a1-3)[Man(a1-6)]Man(b1-4)GlcNAc(b1-4)GlcNAc(?1-"
)

# Use default meta-property functions
get_meta_properties(glycans)

# Use custom meta-property functions
fns <- list(
  nN = ~ glyrepr::count_mono(.x, "HexNAc"),  # purrr-style lambda function
  nH = ~ glyrepr::count_mono(.x, "Hex")
)
get_meta_properties(glycans, fns)
```

make_trait *Use a Large Language Model (LLM) to create a derived trait function*

Description

[Experimental] This function allows you to create a derived trait function using natural language. Note that LLMs can be unreliable, so the result should be verified manually. If the description is not clear, an error will be raised. Try to read the descriptions of built-in traits to get ideas. Currently, only `prop()`, `ratio()`, and `wmean()` are supported. To use this feature, you need to install the `ellmer` package. DeepSeek is used by default for backward compatibility. Other `ellmer` providers can be selected with `provider`, `model`, and provider-specific API key configuration.

Usage

```
make_trait(
  description,
  custom_mp = NULL,
  max_retries = 2,
  verbose = FALSE,
  provider = getOption("glydet.ai_provider", "deepseek"),
  model = getOption("glydet.ai_model", NULL),
  api_key = getOption("glydet.ai_api_key", NULL),
  base_url = getOption("glydet.ai_base_url", NULL)
)
```

Arguments

`description`	A description of the trait in natural language.
`custom_mp`	A named character vector of custom meta-properties. The names are the meta-property names, and the values are in the format "(type) description". For example: `c(nE = "(integer) number of a2,6-linked sialic acids")`. These custom meta-properties will be available for the LLM to use. Note that defining the meta-properties here is not enough for you to use them. You need to define corresponding meta-property functions or specifying meta-property columns. For more information about custom meta-properties, see the vignette Custom Meta-Properties.
`max_retries`	Maximum number of reflection retries when the AI-generated formula's explanation doesn't match the original description. Default is 2.
`verbose`	Whether to print verbose output. Default is FALSE. This is useful for inspecting how LLMs generate trait functions.
`provider`	AI provider passed to `ellmer`. One of "deepseek", "openai", "anthropic", "gemini", "openrouter", or "openai_compatible". "google_gemini" is accepted as an alias for "gemini". Defaults to `getOption("glydet.ai_provider", "deepseek")`.
`model`	Model to use. Defaults to `getOption("glydet.ai_model")`, or "deepseek-chat" for DeepSeek and the provider default for other providers.

api_key	API key for the selected provider. If NULL, the provider specific environment variable is used. Defaults to getOption("glydet.ai_api_key").
base_url	Optional base URL for custom or OpenAI-compatible endpoints. Defaults to getOption("glydet.ai_base_url").

Value

A derived trait function.

Examples

```
# Sys.setenv(DEEPSEEK_API_KEY = "your_api_key")
# my_traits <- list(
#   nS = make_trait("the average number of sialic acids"),
#   nG = make_trait("the average number of galactoses")
# )

# The trait function can then be used in `derive_traits()`:
# derive_traits(exp, trait_fns = my_traits)
```

---

n_glycan_type *Determine N-Glycan Key Properties*

Description

These functions check key properties of an N-glycan:

- n_glycan_type(): Determine the N-glycan type.
- has_bisecting(): Check if the glycan has a bisecting GlcNAc.
- n_antennae(): Count the number of antennae.
- n_fuc(): Count the number of fucoses.
- n_core_fuc(): Count the number of core fucoses.
- n_arm_fuc(): Count the number of arm fucoses.
- n_gal(): Count the number of galactoses.
- n_terminal_gal(): Count the number of terminal galactoses.
- n_sia(): Count the number of sialic acids.
- n_man(): Count the number of mannoses.

All functions assume the glycans are N-glycans without validation, thus may return meaningless values for non-N-glycans. Therefore, please make sure to pass in N-glycans only.

All functions put minimum requirement on the glycans, i.e. they work with glycans with generic monosaccharides (e.g. "Hex", "HexNAc") and no linkage information. This type of structures are common in glycoproteomics and glycomics studies.

Usage

```
n_glycan_type(glycans)

has_bisecting(glycans)

n_antennae(glycans)

n_fuc(glycans)

n_core_fuc(glycans)

n_arm_fuc(glycans)

n_gal(glycans)

n_terminal_gal(glycans)

n_sia(glycans)

n_man(glycans)
```

Arguments

glycans	A `glyrepr::glycan_structure()` vector.

Value

- `n_glycan_type()`: A factor vector indicating the N-glycan type, either "highmannose", "hybrid", "complex", or "paucimannose".
- `has_bisecting()`: A logical vector indicating if the glycan has a bisecting GlcNAc.
- `n_antennae()`: An integer vector indicating the number of antennae.
- `n_fuc()`: An integer vector indicating the number of fucoses.
- `n_core_fuc()`: An integer vector indicating the number of core fucoses.
- `n_arm_fuc()`: An integer vector indicating the number of arm fucoses.
- `n_gal()`: An integer vector indicating the number of galactoses.
- `n_terminal_gal()`: An integer vector indicating the number of terminal galactoses.
- `n_sia()`: An integer vector indicating the number of sialic acids.
- `n_man()`: An integer vector indicating the number of mannoses.

`n_glycan_type()`: N-Glycan Types

Four types of N-glycans are recognized: high mannose, hybrid, complex, and paucimannose. For more information about N-glycan types, see Essentials of Glycobiology.

`has_bisecting()`: **Bisecting GlcNAc**

Bisecting GlcNAc is a GlcNAc residue attached to the core mannose of N-glycans.

```
     Man
        \
GlcNAc - Man - GlcNAc - GlcNAc -
~~~~~~  /
     Man
```

`n_antennae()`: **Number of Antennae**

The number of antennae is the number of branching GlcNAc to the core mannoses.

`n_fuc()`: **Number of Fucoses**

Number of fucoses. This function assumes the fucose is a dHex.

`n_core_fuc()`: **Number of Core Fucoses**

Core fucoses are those fucose residues attached to the core GlcNAc of an N-glycan.

```
Man             Fuc  <- core fucose
   \             |
    Man - GlcNAc - GlcNAc -
   /
Man
```

`n_arm_fuc()`: **Number of Arm Fucoses**

Arm focuses are those focuse residues attached to the branching GlcNAc of an N-glycan.

```
 Fuc  <- arm fucose
  |
GlcNAc - Man
           \
            Man - GlcNAc - GlcNAc -
           /
GlcNAc - Man
```

`n_gal()`: **Number of Galactoses**

This function seems useless and silly. It is, if you have a well-structured glycan with concrete monosaccharides. However, if you only have "Hex" or "H" at hand, it is tricky to know how many of them are "Gal" and how many are "Man". This function makes a simply assumption that all the rightmost "H" in a "H-H-N-H" unit is a galactose. The two "H" on the left are mannoses of the N-glycan core. The "N" is a GlcNAc attached to one core mannose.

`n_terminal_gal()`: Number of Terminal Galactoses

Terminal galactoses are those galactose residues on the non-reducing end without sialic acid capping.

```
         Gal - GlcNAc - Man
         ~~~              \
     terminal Gal          Man - GlcNAc - GlcNAc -
                           /
Neu5Ac - Gal - GlcNAc - Man
         ~~~
   not terminal Gal
```

`n_sia()`: Number of Sialic Acids

Number of sialic acids (Neu5Ac). Neu5Gc is not counted.

`n_man()`: Number of Mannoses

Number of mannoses. This function assumes the Hex of the N-glycan core is mannoses. Also, for high-mannose and paucimannose glycans, all Hex are mannoses. Finally, for hybrid glycans, all the rightmost (the side without branching HexNAc) are mannoses.

`prop` *Create a Proportion Trait*

Description

A proportion trait is the proportion of certain group of glycans within a larger group of glycans. For example, the proportion of sialylated glycans within all glycans, or the proportion of tetra-antennary glycans within all complex glycans. This type of traits is the most common type of glycan derived traits. It can be regarded as a special case of the ratio trait (see `ratio()`).

Usage

```
prop(cond, within = NULL, na_action = "keep")
```

Arguments

`cond`	Condition to use for defining the smaller group. An expression that evaluates to a logical vector. The names of all built-in meta-properties (see `all_mp_fns()`) and custom meta-properties can be used in the expression.
`within`	Condition to use for defining the larger group, with the same format as `cond`. If `NULL` (default), all glycans are used as the larger group.
`na_action`	How to handle missing values. • "keep" (default): keep the missing values as NA. • "zero": set the missing values to 0.

Value

A derived trait function.

How to use

You can use `prop()` to create proportion trait easily.

For example:

```
# Proportion of core-fucosylated glycans within all glycans
prop(nFc > 0)

# Proportion of complex glycans within all glycans
prop(Tp == "complex")

# Proportion of sialylated and fucosylated glycans within all glycans
prop(nS > 0 & nFc > 0)
```

Note that the last example uses `&` for logical AND. Actually, you can use any logical operator in the expression in R (e.g., `|`, `!`, etc.).

If you want to perform a pre-filtering before calculating the proportion, for example, you want to calculate the proportion of core-fucosylated glycans within only complex glycans, you can use `within` to define the denominator.

```
# Proportion of core-fucosylated glycans within complex glycans
prop(nFc > 0, within = (Tp == "complex"))

# Proportion of core-fucosylated glycans with tetra-antenary complex glycans
prop(nFc > 0, within = (Tp == "complex" & nA == 4))
```

The parentheses around the condition in `within` are optional, but it is recommended to use them for clarity.

Note about NA

All the internal summation operations ignore NAs by default. Therefore, NAs in the expression matrix and meta-property values will not result in NAs in the derived traits. However, as all derived traits calculate a ratio of two values, NAs will be introduced when:

1. The denominator is 0. This can happen when the `within` condition selects no glycans.
2. Both the numerator and denominator are 0.

Examples

```
# Proportion of core-fucosylated glycans within all glycans
prop(nFc > 0)

# Proportion of bisecting glycans within all glycans
prop(B)
```

```
# Proportion of sialylated and arm-fucosylated glycans within all glycans
prop(nS > 0 & nFa > 0)

# Proportion of bi-antennary glycans within complex glycans
prop(nA == 2, within = (Tp == "complex"))

# Proportion of sialylated glycans within core-fucosylated tetra-antennary glycans
prop(nS > 0, within = (nFc > 0 & nA == 4))
```

`quantify_motifs` *Quantify Motifs in an Experiment*

Description

This function quantifies motifs from glycomic or glycoproteomic profiles. For glycomics data, it calculates motif quantifications directly. For glycoproteomics data, each glycosite is treated as a separate glycome, and motif quantifications are calculated in a site-specific manner.

The function takes a `glyexp::experiment()` object and returns a new `glyexp::experiment()` object with motif quantifications. Instead of containing quantifications of individual glycans on each glycosite in each sample, the new experiment contains quantifications of each motif on each glycosite in each sample (for glycoproteomics data) or motif quantifications in each sample (for glycomics data).

Usage

```
quantify_motifs(
  exp,
  motifs,
  method = "relative",
  alignments = NULL,
  ignore_linkages = FALSE
)
```

Arguments

`exp`	A `glyexp::experiment()` object. Before using this function, you should preprocess the data using the `glyclean` package. For glycoproteomics data, the data should be aggregated to the "gfs" (glycoforms with structures) level using `glyclean::aggregate()`. Also, please make sure that the `glycan_structure` column is present in the `var_info` table, as not all glycoproteomics identification softwares provide this information. "glycan_structure" can be a `glyrepr::glycan_structure()` vector, or a character vector of glycan structure strings supported by `glyparse::auto_parse()`.
`motifs`	A character vector of motif names, glycan structure strings, a 'glyrepr_structure' object, or a motif specification from `glymotif::dynamic_motifs()` or `glymotif::branch_motifs()`. For glycan structure strings, all formats supported by `glyparse::auto_parse()`

are accepted, including IUPAC-condensed, WURCS, GlycoCT, and others. If the vector is named, the names will be used as motif names. Otherwise, IUPAC-condensed structure strings will be used as motif names. For motif specifications, motifs are extracted automatically from the glycan structures in the experiment, and their IUPAC-condensed strings are used as motif names.

`method` A character string specifying the quantification method. Must be either "absolute" or "relative". Default is "relative". See "Relative and Absolute Motif Quantification" section for details.

`alignments` A character vector specifying the alignment method for each motif. Can be "terminal", "substructure", "core", "whole" or "exact". Default is "substructure". See `glymotif::have_motifs()` for details.

`ignore_linkages`
A logical value. If `TRUE`, linkages will be ignored in the comparison. See `glymotif::have_motifs()` for details.

Value

A new `glyexp::experiment()` object containing motif quantifications. Instead of containing quantifications of individual glycans on each glycosite in each sample, the new experiment contains quantifications of each motif on each glycosite in each sample (for glycoproteomics data) or motif quantifications in each sample (for glycomics data).

The `var_info` table includes a `motif_structure` column containing the parsed glycan structure for each motif, allowing traceability of motif definitions.

For glycoproteomics data, with additional columns:

- `protein`: protein ID
- `protein_site`: the glycosite position on the protein

Other columns in the `var_info` table (e.g., `gene`) are retained if they have a "many-to-one" relationship with glycosites (unique combinations of `protein`, `protein_site`). That is, each glycosite cannot have multiple values for these columns. `gene` is a common example, as a glycosite can only be associated with one gene. Glycan descriptions are not such columns, as a glycosite can have multiple glycans, thus having multiple descriptions. Columns that do not have this relationship with glycosites will be dropped. Don't worry if you cannot understand this logic— just know that this function will do its best to preserve useful information.

The `sample_info` and `meta_data` tables are not modified, except that the `exp_type` field in `meta_data` is set to "traitomics" for glycomics data and "traitproteomics" for glycoproteomics data.

Relative and Absolute Motif Quantification

Motif quantification can be performed in two ways: absolute and relative. Do not confuse this with the absolute and relative quantification of glycans or glycopeptides.

In an omics context, absolute quantification means we can determine the actual concentration (e.g., in mg/L) or counts (e.g., mRNA copy numbers), while relative quantification means we can only compare the abundance of the same molecule between different samples, but the absolute values themselves are not meaningful. Label-free quantification is a relative quantification method.

In the context of motif quantification, absolute and relative refer to whether we normalize the motif quantifications by the total abundance of the glycome (or the mini-glycomes of individual glycosites).

For example, let's say we have a glycomics dataset with two samples, A and B. In each sample, three glycans (G1, G2, G3) are present, with the following abundances:

- Sample A: G1 = 10, G2 = 20, G3 = 30
- Sample B: G1 = 20, G2 = 40, G3 = 60

For simplicity, let's say the motif we want to quantify happens to appear once in all glycans.

For absolute motif quantification, we simply sum the abundances of the motif across all glycans:

- Sample A: $10 \times 1 + 20 \times 1 + 30 \times 1 = 60$ ($\times 1$ because the motif appears once in each glycan)
- Sample B: $20 \times 1 + 40 \times 1 + 60 \times 1 = 120$

The results are clearly different.

However, if we quantify the motif in a relative way:

- Sample A: $(10 \times 1 + 20 \times 1 + 30 \times 1) / (10 + 20 + 30) = 60 / 60 = 1$
- Sample B: $(20 \times 1 + 40 \times 1 + 60 \times 1) / (20 + 40 + 60) = 120 / 120 = 1$

The results are identical!

The absolute motif quantification answers the question "how many motifs are there in the sample?", while the relative motif quantification answers the question "if I take out one glycan molecule, how many motifs are on it in average?"

So which method should you use? It depends on both your data type and research objectives. Here are some general guidelines:

- For glycomics data, you should typically use relative motif quantification, because glycomics data is inherently compositional (search "compositional data" for more details).
- For glycoproteomics data, the choice depends on your research question. If you want to compare observed motif abundance across samples, use absolute motif quantification. Note that many factors can cause one sample to have higher values than another, such as upregulation of enzymes responsible for the motif, or higher overall glycosylation site occupancy. If you want to understand the underlying regulatory mechanisms, use relative motif quantification to correct for differences in site occupancy.

Relationship with Derived Traits

Motif quantification is a special type of derived trait. It is simply a weighted sum (`wsum()`) or weighted mean (`wmean()`) of the motif counts, for absolute and relative motif quantification, respectively. You can perform motif quantification manually using `derive_traits()`.

Let's perform absolute motif quantification manually using `derive_traits()` to better understand the process (we will use custom column meta-properties here):

```
# Add the meta-properties to the variable information tibble
motifs <- c(
  nLx = "Hex(??-?)[dHex(??-?)]HexNAc(??-",  # Lewis x antigen
```

```
  nSLx = "NeuAc(??-?)Hex(??-?)[dHex(??-?)]HexNAc(??-"  # Sialyl Lewis x antigen
)
exp_with_mps <- glymotif::add_motifs_int(exp, motifs)

# Define the traits
trait_fns <- list(Lx = wsum(nLx), SLx = wsum(nSLx))

# Calculate the traits
derive_traits(exp_with_mps, trait_fns = trait_fns)
```

The code snippet above is equivalent to:

```
# Quantify the motifs
quantify_motifs(exp, motifs, method = "absolute")
```

In fact, this is essentially the implementation of the `quantify_motifs()` function, except the actual implementation is more robust and user-friendly.

For relative motif quantification, simply replace `wsum()` with `wmean()`, and everything else remains the same.

See Also

`derive_traits()`, `glymotif::have_motifs()`

Examples

```
library(glyexp)
library(glyclean)

exp <- real_experiment |>
  auto_clean() |>
  slice_head_var(n = 10)

motifs <- c(
  nLx = "Hex(??-?)[dHex(??-?)]HexNAc(??-",  # Lewis x antigen
  nSLx = "NeuAc(??-?)Hex(??-?)[dHex(??-?)]HexNAc(??-"  # Sialyl Lewis x antigen
)

quantify_motifs(exp, motifs)

# Using dynamic motifs (auto-extracted from data)
quantify_motifs(exp, glymotif::dynamic_motifs(max_size = 3))

# Using branch motifs (auto-extracted from data)
quantify_motifs(exp, glymotif::branch_motifs())
```

`ratio` *Create a Ratio Trait*

Description

A ratio trait is the ratio of total abundance of two groups of glycans. For example, the ratio of complex glycans and hybrid glycans, or the ratio of bisecting and unbisecting glycans.

Usage

```
ratio(num_cond, denom_cond, within = NULL, na_action = "keep")
```

Arguments

`num_cond`	Condition to use for defining the numerator. An expression that evaluates to a logical vector. The names of all built-in meta-properties (see `all_mp_fns()`) and custom meta-properties can be used in the expression.
`denom_cond`	Condition to use for defining the denominator. Same format as `num_cond`.
`within`	Condition to set a restriction for the glycans. Same format as `num_cond`.
`na_action`	How to handle missing values. • "keep" (default): keep the missing values as NA. • "zero": set the missing values to 0.

Value

A derived trait function.

How to use

You can use `ratio()` to create ratio trait easily.

For example:

```
# Ratio of complex glycans and hybrid glycans
ratio(Tp == "complex", Tp == "hybrid")

# Ratio of bisecting and unbisecting glycans
ratio(B, !B)

# Ratio of core-fucosylated and non-core-fucosylated glycans within complex glycans
ratio(nFc > 0 & Tp == "complex", nFc == 0 & Tp == "complex")

# The above example can be simplified as:
ratio(nFc > 0, nFc == 0, within = (Tp == "complex"))  # more readable
```

Note that the last example uses `&` for logical AND. Actually, you can use any logical operator in the expression in R (e.g., `|`, `!`, etc.).

`prop()` is a special case of `ratio()`, i.e., `prop(cond, within)` is equivalent to `ratio(cond & within, within)`. We recommend using `prop()` instead of `ratio()` for clarity if possible.

Note about NA

All the internal summation operations ignore NAs by default. Therefore, NAs in the expression matrix and meta-property values will not result in NAs in the derived traits. However, as all derived traits calculate a ratio of two values, NAs will be introduced when:

1. The denominator is 0. This can happen when the `within` condition selects no glycans.
2. Both the numerator and denominator are 0.

Examples

```
# Ratio of complex glycans and hybrid glycans
ratio(Tp == "complex", Tp == "hybrid")

# Ratio of bisecting and unbisecting glycans within bi-antennary glycans
ratio(B, !B, within = (nA == 2))
```

---

total *Create a Total Abundance Trait*

Description

A total abundance trait is the total abundance of a group of glycans. For example, the total abundance of all complex glycans, or the total abundance of all tetra-antennary glycans.

Usage

```
total(cond)
```

Arguments

cond	Condition to use for defining the group of glycans. An expression that evaluates to a logical vector. The names of all built-in meta-properties (see `all_mp_fns()`) and custom meta-properties can be used in the expression.

Value

A derived trait function.

How to use

You can use `total()` to create total abundance trait easily.

For example:

```
# Total abundance of all complex glycans
total(Tp == "complex")

# Total abundance of all tetra-antennary glycans
total(nA == 4)
```

Examples

```
# Total abundance of all complex glycans
total(Tp == "complex")

# Total abundance of all tetra-antennary glycans
total(nA == 4)
```

`traits_basic` *Get Basic Derived Traits*

Description

These derived traits are the most basic and commonly used derived traits. They describe global properties of a glycome including the type of glycans, fucosylation level, sialylation level, galactosylation level, and branching level.

Usage

```
traits_basic(sia_link = FALSE)

basic_traits(sia_link = FALSE)
```

Arguments

`sia_link`	A boolean indicating whether to include sialic acid linkage traits. Default is `FALSE`.

Value

A named list of derived traits.

Descriptions of traits

The explanations of the derived traits are as follows:

- `TM`: Proportion of highmannose glycans
- `TH`: Proportion of hybrid glycans
- `TC`: Proportion of complex glycans
- `MM`: Average number of mannoses within highmannose glycans
- `CA2`: Proportion of bi-antennary glycans within complex glycans
- `CA3`: Proportion of tri-antennary glycans within complex glycans
- `CA4`: Proportion of tetra-antennary glycans within complex glycans
- `TF`: Proportion of fucosylated glycans
- `TFc`: Proportion of core-fucosylated glycans

- TFa: Proportion of arm-fucosylated glycans
- TB: Proportion of glycans with bisecting GlcNAc
- GS: Average degree of sialylation per galactose
- AG: Average degree of galactosylation per antenna
- TS: Proportion of sialylated glycans

Four additional sialic acid linkage traits are included if sia_link = TRUE.

- GE: Average degree of a2,6-linked sialylation per galactose
- GL: Average degree of a2,3-linked sialylation per galactose
- TE: Proportion of a2,6-linked sialylated glycans
- TL: Proportion of a2,3-linked sialylated glycans

Usage of sialic acid linkage traits

To use these sialic acid linkage traits, var_info of the input glyexp::experiment() must have the following columns:

- nE: Number of a2,6-linked sialic acids
- nL: Number of a2,3-linked sialic acids

Note that you have to add these two columns even if the glycan_structure column has intact linkages. This is because by convention all traits work with glycan structures with "basic" structure levels (i.e., with generic monosaccharides like "Hex" and "HexNAc" and no linkages specified).

Examples

```
traits_basic()
```

---

traits_clerc_2018 *Get Traits in Clerc et al. 2018*

Description

These traits are the ones used by Clerc et al. 2018 (https://doi.org/10.1053/j.gastro.2018.05.030). We generally don't recommend using these traits because they are either redundant or missing some important traits. We include this function because this paper is very influential in the field.

Usage

```
traits_clerc_2018(sia_link = FALSE)
```

Arguments

sia_link	A boolean indicating whether to include sialic acid linkage traits. Default is FALSE.

Value

A named list of derived traits.

Usage of sialic acid linkage traits

To use these sialic acid linkage traits, `var_info` of the input `glyexp::experiment()` must have the following columns:

- `nE`: Number of a2,6-linked sialic acids
- `nL`: Number of a2,3-linked sialic acids

Note that you have to add these two columns even if the `glycan_structure` column has intact linkages. This is because by convention all traits work with glycan structures with "basic" structure levels (i.e., with generic monosaccharides like "Hex" and "HexNAc" and no linkages specified).

Examples

```
traits_clerc_2018()[1:5]
```

---

traits_detailed *Get Detailed Derived Traits*

Description

This function returns a named list of detailed derived traits. Compared to `traits_basic()`, this function includes derived traits with more detailed `within` conditions.

Usage

```
traits_detailed(sia_link = FALSE)

all_traits(sia_link = FALSE)
```

Arguments

`sia_link`	A boolean indicating whether to include sialic acid linkage traits. Default is `FALSE`.

Details

The explanations of the derived traits are as follows:

- `A1F`: Proportion of fucosylated glycans within mono-antennary glycans
- `A2F`: Proportion of fucosylated glycans within bi-antennary glycans
- `A3F`: Proportion of fucosylated glycans within tri-antennary glycans
- `A4F`: Proportion of fucosylated glycans within tetra-antennary glycans
- `A1Fc`: Proportion of core-fucosylated glycans within mono-antennary glycans

- A2Fc: Proportion of core-fucosylated glycans within bi-antennary glycans
- A3Fc: Proportion of core-fucosylated glycans within tri-antennary glycans
- A4Fc: Proportion of core-fucosylated glycans within tetra-antennary glycans
- A1Fa: Proportion of arm-fucosylated glycans within mono-antennary glycans
- A2Fa: Proportion of arm-fucosylated glycans within bi-antennary glycans
- A3Fa: Proportion of arm-fucosylated glycans within tri-antennary glycans
- A4Fa: Proportion of arm-fucosylated glycans within tetra-antennary glycans
- A1SFa: Proportion of arm-fucosylated glycans within sialylated mono-antennary glycans
- A2SFa: Proportion of arm-fucosylated glycans within sialylated bi-antennary glycans
- A3SFa: Proportion of arm-fucosylated glycans within sialylated tri-antennary glycans
- A4SFa: Proportion of arm-fucosylated glycans within sialylated tetra-antennary glycans
- A1S0Fa: Proportion of arm-fucosylated glycans within asialylated mono-antennary glycans
- A2S0Fa: Proportion of arm-fucosylated glycans within asialylated bi-antennary glycans
- A3S0Fa: Proportion of arm-fucosylated glycans within asialylated tri-antennary glycans
- A4S0Fa: Proportion of arm-fucosylated glycans within asialylated tetra-antennary glycans
- A1B: Proportion of bisecting glycans within mono-antennary glycans
- A2B: Proportion of bisecting glycans within bi-antennary glycans
- A3B: Proportion of bisecting glycans within tri-antennary glycans
- A4B: Proportion of bisecting glycans within tetra-antennary glycans
- A1FcB: Proportion of bisecting glycans within core-fucosylated mono-antennary glycans
- A2FcB: Proportion of bisecting glycans within core-fucosylated bi-antennary glycans
- A3FcB: Proportion of bisecting glycans within core-fucosylated tri-antennary glycans
- A4FcB: Proportion of bisecting glycans within core-fucosylated tetra-antennary glycans
- A1Fc0B: Proportion of bisecting glycans within a-core-fucosylated mono-antennary glycans
- A2Fc0B: Proportion of bisecting glycans within a-core-fucosylated bi-antennary glycans
- A3Fc0B: Proportion of bisecting glycans within a-core-fucosylated tri-antennary glycans
- A4Fc0B: Proportion of bisecting glycans within a-core-fucosylated tetra-antennary glycans
- A1G: Average degree of galactosylation per antenna within mono-antennary glycans
- A2G: Average degree of galactosylation per antenna within bi-antennary glycans
- A3G: Average degree of galactosylation per antenna within tri-antennary glycans
- A4G: Average degree of galactosylation per antenna within tetra-antennary glycans
- A1Gt: Average degree of terminal galactose per antenna within mono-antennary glycans
- A2Gt: Average degree of terminal galactose per antenna within bi-antennary glycans
- A3Gt: Average degree of terminal galactose per antenna within tri-antennary glycans
- A4Gt: Average degree of terminal galactose per antenna within tetra-antennary glycans
- A1S: Average degree of sialylation per antenna within mono-antennary glycans
- A2S: Average degree of sialylation per antenna within bi-antennary glycans

- A3S: Average degree of sialylation per antenna within tri-antennary glycans
- A4S: Average degree of sialylation per antenna within tetra-antennary glycans
- A1GS: Average degree of sialylation per galactose within mono-antennary glycans
- A2GS: Average degree of sialylation per galactose within bi-antennary glycans
- A3GS: Average degree of sialylation per galactose within tri-antennary glycans
- A4GS: Average degree of sialylation per galactose within tetra-antennary glycans

These additional sialic acid linkage traits are included if sia_link = TRUE.

- A1E: Average degree of a2,6-linked sialylation per antenna within mono-antennary glycans
- A2E: Average degree of a2,6-linked sialylation per antenna within bi-antennary glycans
- A3E: Average degree of a2,6-linked sialylation per antenna within tri-antennary glycans
- A4E: Average degree of a2,6-linked sialylation per antenna within tetra-antennary glycans
- A1L: Average degree of a2,3-linked sialylation per antenna within mono-antennary glycans
- A2L: Average degree of a2,3-linked sialylation per antenna within bi-antennary glycans
- A3L: Average degree of a2,3-linked sialylation per antenna within tri-antennary glycans
- A4L: Average degree of a2,3-linked sialylation per antenna within tetra-antennary glycans
- A1GE: Average degree of a2,6-linked sialylation per galactose within mono-antennary glycans
- A2GE: Average degree of a2,6-linked sialylation per galactose within bi-antennary glycans
- A3GE: Average degree of a2,6-linked sialylation per galactose within tri-antennary glycans
- A4GE: Average degree of a2,6-linked sialylation per galactose within tetra-antennary glycans
- A1GL: Average degree of a2,3-linked sialylation per galactose within mono-antennary glycans
- A2GL: Average degree of a2,3-linked sialylation per galactose within bi-antennary glycans
- A3GL: Average degree of a2,3-linked sialylation per galactose within tri-antennary glycans
- A4GL: Average degree of a2,3-linked sialylation per galactose within tetra-antennary glycans

Value

A named list of derived traits.

Usage of sialic acid linkage traits

To use these sialic acid linkage traits, var_info of the input glyexp::experiment() must have the following columns:

- nE: Number of a2,6-linked sialic acids
- nL: Number of a2,3-linked sialic acids

Note that you have to add these two columns even if the glycan_structure column has intact linkages. This is because by convention all traits work with glycan structures with "basic" structure levels (i.e., with generic monosaccharides like "Hex" and "HexNAc" and no linkages specified).

Examples

```
traits_detailed()
```

traits_fu_2026 *Get Traits in Fu et al. 2026*

Description

These traits are the ones used by Fu et al. 2026 (https://doi.org/10.1038/s41467-026-68579-x). It is much like `traits_detailed()`, but doesn't differentiate core- and arm-fucosylation, and doesn't include the sialic acid linkage traits. Also, many traits are too specific to be useful and interpretable.

Usage

```
traits_fu_2026()
```

Value

A named list of derived traits.

Examples

```
traits_fu_2026()[1:5]
```

traits_li_2025 *Get Traits in Li et al. 2025*

Description

These traits are the ones used by Li et al. 2025 (https://doi.org/10.1038/s41467-025-57633-9). These traits were created based only on glycan compositions, not structures. Here the traits are remapped to use glycan structures, so they are not exactly the same as the ones in the paper. Generally we don't recommend using these traits because they capture too little information. We include this function because this paper is very influential in the field.

Usage

```
traits_li_2025()
```

Value

A named list of derived traits.

Examples

```
traits_li_2025()[1:5]
```

wmean *Create a Weighted-Mean Trait*

Description

A weighted-mean trait is the average value of some quantitative property within a group of glycans, weighted by the abundance of the glycans. For example, the average number of antennae within all complex glycans, or the average number of sialic acids within all glycans.

Usage

```
wmean(val, within = NULL, na_action = "keep")
```

Arguments

`val`	Expression to use for defining the value. An expression that evaluates to a numeric vector. The names of all built-in meta-properties (see `all_mp_fns()`) and custom meta-properties can be used in the expression.
`within`	Condition to set a restriction for the glycans. Same format as `val`.
`na_action`	How to handle missing values. • "keep" (default): keep the missing values as NA. • "zero": set the missing values to 0.

Value

A derived trait function.

How to use

You can use `wmean()` to create weighted-mean trait easily.

For example:

```
# Weighted mean of the number of sialic acids within all glycans
wmean(nS)

# Average degree of sialylation per antenna within all glycans
wmean(nS / nA)
```

Note that the last example uses `/` for division. Actually, you can use any arithmetic operator in the expression in R (e.g., `*`, `+`, `-`, etc.).

If you want to perform a pre-filtering before calculating the weighted-mean, for example, you want to calculate the average degree of sialylation per antenna within only complex glycans, you can use `within` to define the restriction.

```
# Average number of antennae within complex glycans
wmean(nA, within = (Tp == "complex"))
```

Note about NA

All the internal summation operations ignore NAs by default. Therefore, NAs in the expression matrix and meta-property values will not result in NAs in the derived traits. However, as all derived traits calculate a ratio of two values, NAs will be introduced when:

1. The denominator is 0. This can happen when the `within` condition selects no glycans.
2. Both the numerator and denominator are 0.

Examples

```
# Weighted mean of the number of sialic acids within all glycans
wmean(nS)

# Average degree of sialylation per antenna within all glycans
wmean(nS / nA)

# Average number of antennae within complex glycans
wmean(nA, within = (Tp == "complex"))
```

---

wsum *Create a Weighted Sum Trait*

Description

A weighted sum trait is the sum of a quantitative property within a group of glycans, weighted by the abundance of the glycans. For example, the sum of the number of sialic acids within all glycans, or the sum of the number of Lewis x antigens within all glycans.

Usage

```
wsum(val, within = NULL)
```

Arguments

`val`	Expression to use for defining the value. An expression that evaluates to a logical vector. The names of all built-in meta-properties (see `all_mp_fns()`) and custom meta-properties can be used in the expression.
`within`	Condition to set a restriction for the glycans. Same format as `val`.

Value

A derived trait function.

How to use

You can use `wsum()` to create weighted sum trait easily.

For example:

```
# Weighted sum of the number of sialic acids within all glycans
wsum(nS)
```

This can be regarded as the quantification of sialic acids. If some glycan has only one sialic acid, its abundance is added to the results. If another glycan has two sialic acids, its abundance is doubled before being added to the results.

You can also use `within` to restrict the weighted sum calculation to specific glycan subsets. For example, you can calculate the weighted sum of the number of sialic acids within complex glycans:

```
wsum(nS, within = (Tp == "complex"))
```

Examples

```
# Weighted sum of the number of sialic acids within all glycans
wsum(nS)

# Weighted sum of the number of sialic acids within complex glycans
wsum(nS, within = (Tp == "complex"))
```

Index