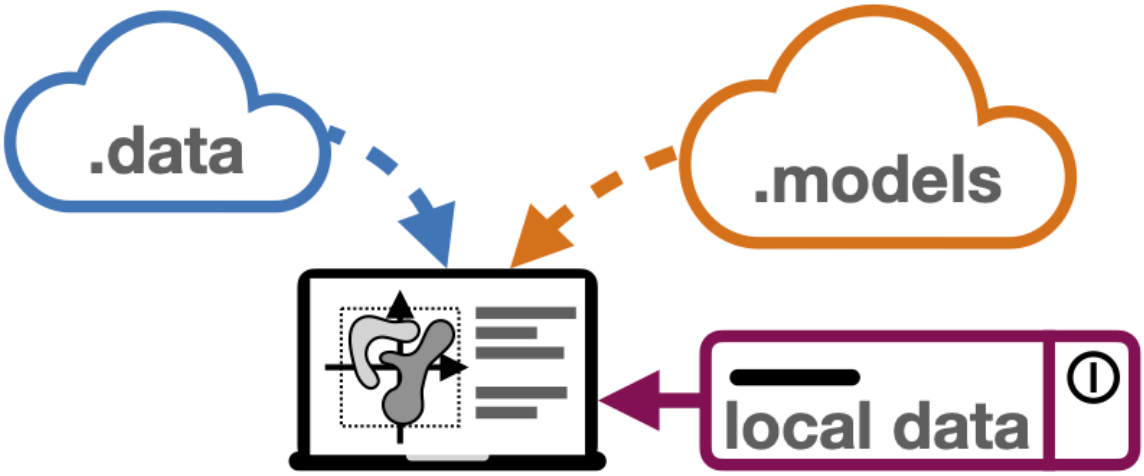
sfaira Documentation

Release v0.3.12+0.g8ebd63b

Leander Dony, David S. Fischer, Lukas Heumos

Feb 15, 2022

CONTENTS



sfaira is a model and a data repository in a single python package. We provide an interactive overview of the current state of the zoos on sfaira-portal.

sfaira fits into an environment of many other project centred on making data and models accessible.

CHAPTER

ONE

DATA ZOO

We focus on providing a python interface to interact with locally stored data set collections without requiring dedicated data reading and annotation harmonisation scripts: These code blocks are absorbed into our data zoo backend and can be conveniently triggered with short commands.

CHAPTER

TWO

MODEL ZOO

A large body of recent research has been devoted to improving models that learn representation of cell captured with single-cell RNA-seq. These models include embedding models such as autoencoders and cell type prediction models. Many of these models are implemented in software packages and can be deployed on new data sets. In many of these cases, it also makes sense to use pre-trained models to leverage previously published modelling results. We provide a single interface to interact with such pre-trained models which abstracts model settings into a API so that users can easily switch between different pre-trained models. Importantly, model execution is performed locally so that data does not have to be uploaded to external servers and model storage is decentral so that anybody can contribute models easily. Users benefit from easy, streamlined access to models that can be used in analysis workflows, developers benefit from being able to deploy models to a large community of users without having to set up a model zoo.

2.1 News

No news yet, stay tuned!

2.2 Latest additions

2.2.1 Installation

sfaira is pip installable.

PyPI

To install a sfaira release directly from PyPi, run:

```
pip install sfaira
```

Install a development version

To install a specific branch `target_branch` of sfaira from a clone, run:

```
cd target_directory
git clone https://github.com/theislab/sfaira.git
cd sfaira
git checkout target_branch
git pull
pip install -e .
```

In most cases, you would install one of the following: You may choose the branch `release` if you want to use a relatively stable version which is similar to the current release but may have additional features already. You may choose the branch `dev` if you want newer features than available from `release`. You may choose a specific feature branch if you want to use or improve that feature before it is reviewed and merged into `dev`. Note that the `master` branch only contains releases, so every installation based on the `master` branch can also be performed via PyPi.

2.2.2 API

Import sfaira as:

```
import sfaira
```

Data: `data`

Data loaders

The sfaira data zoo API.

Dataset representing classes used for development:

data.DatasetBase([data_path, meta_path, ...])	
data.DatasetGroup(datasets[, collection_id])	Container class that co-manages multiple data sets, removing need to call Dataset() methods directly through wrapping them.
data.DatasetGroupDirectoryOriented(file_base)	
data.DatasetSuperGroup(dataset_groups)	Container for multiple DatasetGroup instances.

sfaira.data.DatasetBase

class sfaira.data.**DatasetBase**(*data_path: Optional[str] = None*, *meta_path: Optional[str] = None*, *cache_path: Optional[str] = None*, *load_func=None*, *dict_load_func_annotation=None*, *yaml_path: Optional[str] = None*, *sample_fn: Optional[str] = None*, *sample_fns: Optional[List[str]] = None*, *additional_annotation_key: Optional[str] = None*, ***kwargs*)

Attributes

additional_annotation_key
annotated
assay_differentiation
assay_sc

continues on next page

Table 2 – continued from previous page

assay_type_differentiation	
author	
bio_sample	
bio_sample_obs_key	
cache_fn	
cell_line	
cell_type	
celltypes_universe	
citation	Return all information necessary to cite data set.
data_dir	
default_embedding	
development_stage	
directory_formatted_doi	
disease	
doi	All publication DOI associated with the study which are the journal publication and the preprint.
doi_cleaned_id	
doi_journal	The prepring publication (secondary) DOI associated with the study.
doi_main	The main DOI associated with the study which is the journal publication if available, otherwise the preprint.
doi_preprint	The journal publication (main) DOI associated with the study.
download_url_data	Data download website(s).
download_url_meta	Meta data download website(s).
ethnicity	
feature_reference	
feature_type	
id	
individual	
loaded	return: Whether DataSet was loaded into memory.

continues on next page

Table 2 – continued from previous page

meta
meta_fn
ncells
ontology_class_maps
organ
organism
primary_data
sample_source
sex
source
source_doi
state_exact
tech_sample
tech_sample_obs_key
title
year
adata
data_dir_base
meta_path
cache_path
genome
supplier
layer_counts
layer_processed
layer_spliced_counts

continues on next page

Table 2 – continued from previous page

layer_spliced_processed
layer_unspliced_counts
layer_unspliced_processed
layer_velocity
gm
treatment
assay_sc_obs_key
assay_differentiation_obs_key
assay_type_differentiation_obs_key
cell_type_obs_key
development_stage_obs_key
disease_obs_key
ethnicity_obs_key
gm_obs_key
individual_obs_key
organ_obs_key
organism_obs_key
sample_source_obs_key
sex_obs_key
source_doi_obs_key
state_exact_obs_key
treatment_obs_key
feature_id_var_key
feature_reference_var_key
feature_symbol_var_key

continues on next page

Table 2 – continued from previous page

feature_type_var_key
spatial_x_coord_obs_key
spatial_y_coord_obs_key
spatial_z_coord_obs_key
vdj_vj_1_obs_key_prefix
vdj_vj_2_obs_key_prefix
vdj_vdj_1_obs_key_prefix
vdj_vdj_2_obs_key_prefix
vdj_c_call_obs_key_suffix
vdj_consensus_count_obs_key_suffix
vdj_d_call_obs_key_suffix
vdj_duplicate_count_obs_key_suffix
vdj_j_call_obs_key_suffix
vdj_junction_obs_key_suffix
vdj_junction_aa_obs_key_suffix
vdj_locus_obs_key_suffix
vdj_productive_obs_key_suffix
vdj_v_call_obs_key_suffix
load_raw
mapped_features
remove_gene_version
subset_gene_type
streamlined_meta
sample_fn

sfaira.data.DatasetBase.additional_annotation_key

property DatasetBase.**additional_annotation_key**: Union[None, str]

sfaira.data.DatasetBase.annotated

property DatasetBase.**annotated**: Optional[bool]

sfaira.data.DatasetBase.assay_differentiation

property DatasetBase.**assay_differentiation**: Union[None, str]

sfaira.data.DatasetBase.assay_sc

property DatasetBase.**assay_sc**: Union[None, str]

sfaira.data.DatasetBase.assay_type_differentiation

property DatasetBase.**assay_type_differentiation**: Union[None, str]

sfaira.data.DatasetBase.author

property DatasetBase.**author**: str

sfaira.data.DatasetBase.bio_sample

property DatasetBase.**bio_sample**: Union[None, str]

sfaira.data.DatasetBase.bio_sample_obs_key

property DatasetBase.**bio_sample_obs_key**: Union[None, str]

sfaira.data.DatasetBase.cache_fn

property DatasetBase.**cache_fn**

sfaira.data.DatasetBase.cell_line

property DatasetBase.**cell_line**: Union[None, str]

sfaira.data.DatasetBase.cell_type

property DatasetBase.**cell_type**: Union[None, str]

sfaira.data.DatasetBase.celltypes_universe

property DatasetBase.**celltypes_universe**

sfaira.data.DatasetBase.citation

property DatasetBase.**citation**

Return all information necessary to cite data set.

Returns

sfaira.data.DatasetBase.data_dir

property DatasetBase.**data_dir**

sfaira.data.DatasetBase.default_embedding

property DatasetBase.**default_embedding**: Union[None, str]

sfaira.data.DatasetBase.development_stage

property DatasetBase.**development_stage**: Union[None, str]

sfaira.data.DatasetBase.directory_formatted_doi

property DatasetBase.**directory_formatted_doi**: str

sfaira.data.DatasetBase.disease

property DatasetBase.**disease**: Union[None, str]

sfaira.data.DatasetBase.doi

property DatasetBase.**doi**: List[str]
All publication DOI associated with the study which are the journal publication and the preprint. See also .doi_preprint, .doi_journal.

sfaira.data.DatasetBase.doi_cleaned_id

property DatasetBase.**doi_cleaned_id**

sfaira.data.DatasetBase.doi_journal

property DatasetBase.**doi_journal**: str
The prepring publication (secondary) DOI associated with the study. See also .doi_journal.

sfaira.data.DatasetBase.doi_main

property DatasetBase.**doi_main**: str
The main DOI associated with the study which is the journal publication if available, otherwise the preprint. See also .doi_preprint, .doi_journal.

sfaira.data.DatasetBase.doi_preprint

property DatasetBase.**doi_preprint**: str
The journal publication (main) DOI associated with the study. See also .doi_preprint.

sfaira.data.DatasetBase.download_url_data

property DatasetBase.**download_url_data**: Union[Tuple[List[str]], Tuple[List[None]]]
Data download website(s).

Save as tuple with single element, which is a list of all download websites relevant to dataset. :return:

sfaira.data.DatasetBase.download_url_meta

property DatasetBase.**download_url_meta**: Union[Tuple[List[str]], Tuple[List[None]]]
Meta data download website(s).

Save as tuple with single element, which is a list of all download websites relevant to dataset. :return:

sfaira.data.DatasetBase.ethnicity

property DatasetBase.**ethnicity**: Union[None, str]

sfaira.data.DatasetBase.feature_reference

property DatasetBase.**feature_reference**: str

sfaira.data.DatasetBase.feature_type

property DatasetBase.**feature_type**: str

sfaira.data.DatasetBase.id

property DatasetBase.**id**: str

sfaira.data.DatasetBase.individual

property DatasetBase.**individual**: Union[None, str]

sfaira.data.DatasetBase.loaded

property DatasetBase.**loaded**: bool
return: Whether DataSet was loaded into memory.

sfaira.data.DatasetBase.meta

property DatasetBase.**meta**: Union[None, pandas.core.frame.DataFrame]

sfaira.data.DatasetBase.meta_fn

property DatasetBase.**meta_fn**

sfaira.data.DatasetBase.ncells

property DatasetBase.**ncells**: Union[None, int]

sfaira.data.DatasetBase.ontology_class_maps

property DatasetBase.**ontology_class_maps**: Dict[str, pandas.core.frame.DataFrame]

sfaira.data.DatasetBase.organ

property DatasetBase.**organ**: Union[None, str]

sfaira.data.DatasetBase.organism

property DatasetBase.**organism**: Union[None, str]

sfaira.data.DatasetBase.primary_data

property DatasetBase.**primary_data**: Union[None, bool]

sfaira.data.DatasetBase.sample_source

property DatasetBase.**sample_source**: Union[None, str]

sfaira.data.DatasetBase.sex

property DatasetBase.**sex**: Union[None, str]

sfaira.data.DatasetBase.source

property DatasetBase.**source**: str

sfaira.data.DatasetBase.source_doi

property DatasetBase.**source_doi**: str

sfaira.data.DatasetBase.state_exact

property DatasetBase.**state_exact**: Union[None, str]

sfaira.data.DatasetBase.tech_sample

property DatasetBase.tech_sample: Union[None, str]

sfaira.data.DatasetBase.tech_sample_obs_key

property DatasetBase.tech_sample_obs_key: Union[None, str]

sfaira.data.DatasetBase.title

property DatasetBase.title

sfaira.data.DatasetBase.year

property DatasetBase.year: Union[None, int]

sfaira.data.DatasetBase.adata

DatasetBase.adata: Union[None, anndata._core.anndata.AnnData]

sfaira.data.DatasetBase.data_dir_base

DatasetBase.data_dir_base: Union[None, str]

sfaira.data.DatasetBase.meta_path

DatasetBase.meta_path: Union[None, str]

sfaira.data.DatasetBase.cache_path

DatasetBase.cache_path: Union[None, str]

sfaira.data.DatasetBase.genome

DatasetBase.genome: Union[None, str]

sfaira.data.DatasetBase.supplier

DatasetBase.**supplier**: str

sfaira.data.DatasetBase.layer_counts

DatasetBase.**layer_counts**: Union[None, str]

sfaira.data.DatasetBase.layer_processed

DatasetBase.**layer_processed**: Union[None, str]

sfaira.data.DatasetBase.layer_spliced_counts

DatasetBase.**layer_spliced_counts**: Union[None, str]

sfaira.data.DatasetBase.layer_spliced_processed

DatasetBase.**layer_spliced_processed**: Union[None, str]

sfaira.data.DatasetBase.layer_unspliced_counts

DatasetBase.**layer_unspliced_counts**: Union[None, str]

sfaira.data.DatasetBase.layer_unspliced_processed

DatasetBase.**layer_unspliced_processed**: Union[None, str]

sfaira.data.DatasetBase.layer_velocity

DatasetBase.**layer_velocity**: Union[None, str]

sfaira.data.DatasetBase.gm

DatasetBase.**gm**: Union[None, str]

sfaira.data.DatasetBase.treatment

DatasetBase.**treatment**: Union[None, str]

sfaira.data.DatasetBase.assay_sc_obs_key

DatasetBase.**assay_sc_obs_key**: Union[None, str]

sfaira.data.DatasetBase.assay_differentiation_obs_key

DatasetBase.**assay_differentiation_obs_key**: Union[None, str]

sfaira.data.DatasetBase.assay_type_differentiation_obs_key

DatasetBase.**assay_type_differentiation_obs_key**: Union[None, str]

sfaira.data.DatasetBase.cell_type_obs_key

DatasetBase.**cell_type_obs_key**: Union[None, str]

sfaira.data.DatasetBase.development_stage_obs_key

DatasetBase.**development_stage_obs_key**: Union[None, str]

sfaira.data.DatasetBase.disease_obs_key

DatasetBase.**disease_obs_key**: Union[None, str]

sfaira.data.DatasetBase.ethnicity_obs_key

DatasetBase.**ethnicity_obs_key**: Union[None, str]

sfaira.data.DatasetBase.gm_obs_key

DatasetBase.**gm_obs_key**: Union[None, str]

sfaira.data.DatasetBase.individual_obs_key

DatasetBase.**individual_obs_key**: Union[None, str]

sfaira.data.DatasetBase.organ_obs_key

DatasetBase.**organ_obs_key**: Union[None, str]

sfaira.data.DatasetBase.organism_obs_key

DatasetBase.**organism_obs_key**: Union[None, str]

sfaira.data.DatasetBase.sample_source_obs_key

DatasetBase.**sample_source_obs_key**: Union[None, str]

sfaira.data.DatasetBase.sex_obs_key

DatasetBase.**sex_obs_key**: Union[None, str]

sfaira.data.DatasetBase.source_doi_obs_key

DatasetBase.**source_doi_obs_key**: Union[None, str]

sfaira.data.DatasetBase.state_exact_obs_key

DatasetBase.**state_exact_obs_key**: Union[None, str]

sfaira.data.DatasetBase.treatment_obs_key

DatasetBase.**treatment_obs_key**: Union[None, str]

sfaira.data.DatasetBase.feature_id_var_key

DatasetBase.**feature_id_var_key**: Union[None, str]

sfaira.data.DatasetBase.feature_reference_var_key

DatasetBase.**feature_reference_var_key**: Union[None, str]

sfaira.data.DatasetBase.feature_symbol_var_key

DatasetBase.**feature_symbol_var_key**: Union[None, str]

sfaira.data.DatasetBase.feature_type_var_key

DatasetBase.**feature_type_var_key**: Union[None, str]

sfaira.data.DatasetBase.spatial_x_coord_obs_key

DatasetBase.**spatial_x_coord_obs_key**: Union[None, str]

sfaira.data.DatasetBase.spatial_y_coord_obs_key

DatasetBase.**spatial_y_coord_obs_key**: Union[None, str]

sfaira.data.DatasetBase.spatial_z_coord_obs_key

DatasetBase.**spatial_z_coord_obs_key**: Union[None, str]

sfaira.data.DatasetBase.vdj_vj_1_obs_key_prefix

DatasetBase.**vdj_vj_1_obs_key_prefix**: Union[None, str]

sfaira.data.DatasetBase.vdj_vj_2_obs_key_prefix

DatasetBase.**vdj_vj_2_obs_key_prefix**: Union[None, str]

sfaira.data.DatasetBase.vdj_vdj_1_obs_key_prefix

DatasetBase.**vdj_vdj_1_obs_key_prefix**: Union[None, str]

sfaira.data.DatasetBase.vdj_vdj_2_obs_key_prefix

DatasetBase.**vdj_vdj_2_obs_key_prefix: Union[None, str]**

sfaira.data.DatasetBase.vdj_c_call_obs_key_suffix

DatasetBase.**vdj_c_call_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_consensus_count_obs_key_suffix

DatasetBase.**vdj_consensus_count_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_d_call_obs_key_suffix

DatasetBase.**vdj_d_call_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_duplicate_count_obs_key_suffix

DatasetBase.**vdj_duplicate_count_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_j_call_obs_key_suffix

DatasetBase.**vdj_j_call_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_junction_obs_key_suffix

DatasetBase.**vdj_junction_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_junction_aa_obs_key_suffix

DatasetBase.**vdj_junction_aa_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_locus_obs_key_suffix

DatasetBase.**vdj_locus_obs_key_suffix: Union[None, str]**

sfaira.data.DatasetBase.vdj_productive_obs_key_suffix

DatasetBase.**vdj_productive_obs_key_suffix:** Union[None, str]

sfaira.data.DatasetBase.vdj_v_call_obs_key_suffix

DatasetBase.**vdj_v_call_obs_key_suffix:** Union[None, str]

sfaira.data.DatasetBase.load_raw

DatasetBase.**load_raw:** Union[None, bool]

sfaira.data.DatasetBase.mapped_features

DatasetBase.**mapped_features:** Union[None, str, bool]

sfaira.data.DatasetBase.remove_gene_version

DatasetBase.**remove_gene_version:** Union[None, bool]

sfaira.data.DatasetBase.subset_gene_type

DatasetBase.**subset_gene_type:** Union[None, str]

sfaira.data.DatasetBase.streamlined_meta

DatasetBase.**streamlined_meta:** bool

sfaira.data.DatasetBase.sample_fn

DatasetBase.**sample_fn:** Union[None, str]

Methods

clear()	Remove loaded .adata to reduce memory footprint.
collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
get_ontology(k)	

continues on next page

Table 3 – continued from previous page

load([load_raw, allow_caching])	**param remove_gene_version** Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
load_meta(fn)	
project_free_to_ontology(attr)	Project free text cell type names to ontology based on mapping table.
read_ontology_class_maps(fns)	Load class maps of free text class labels to ontology classes.
set_dataset_id([idx])	
show_summary()	
streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance to a defined output schema.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.
write_meta([fn_meta, dir_out])	Write meta data object for data set.
write_ontology_class_maps(fn, attrs[, ...])	Load class maps of ontology-controlled field to ontology classes.

sfaira.data.DatasetBase.clear

DatasetBase.**clear**()
Remove loaded .adata to reduce memory footprint.

Returns

sfaira.data.DatasetBase.collapse_counts

DatasetBase.**collapse_counts**()
Collapse count matrix along duplicated index.

sfaira.data.DatasetBase.download

DatasetBase.**download**(***kwargs*)

sfaira.data.DatasetBase.get_ontology

DatasetBase.**get_ontology**(*k*) → Optional[*sfaira.versions.metadata.base.OntologyHierarchical*]

sfaira.data.DatasetBase.load

DatasetBase.**load**(*load_raw: bool = False*, *allow_caching: bool = True*, ***kwargs*)

Parameters

- **remove_gene_version** – Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
- **match_to_reference** – Reference genomes name or False to keep original feature space.
- **load_raw** – Loads unprocessed version of data if available in data loader.
- **allow_caching** – Whether to allow method to cache adata object for faster re-loading.

sfaira.data.DatasetBase.load_meta

DatasetBase.**load_meta**(*fn: Optional[Union[os.PathLike, str]]*)

sfaira.data.DatasetBase.project_free_to_ontology

DatasetBase.**project_free_to_ontology**(*attr: str*)

Project free text cell type names to ontology based on mapping table.

ToDo: add ontology ID setting here.

sfaira.data.DatasetBase.read_ontology_class_maps

DatasetBase.**read_ontology_class_maps**(*fns: List[str]*)

Load class maps of free text class labels to ontology classes.

Parameters **fns** – File names of tsv to load class maps from.

Returns

sfaira.data.DatasetBase.set_dataset_id

DatasetBase.**set_dataset_id**(*idx: int = 1*)

sfaira.data.DatasetBase.show_summary

DatasetBase.**show_summary**()

sfaira.data.DatasetBase.streamline_features

DatasetBase.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. This also adds missing ensid or gene symbol columns if match_to_reference is not set to False and removes all adata.var columns that are not defined as gene_id_ensembl_var_key or gene_id_symbol_var_key in the dataloader.

Parameters

- **match_to_release** – Which genome annotation release to map the feature space to. Note that assemblies from ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be:
  - str: Provide the name of the release.
  - **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.
- **remove_gene_version** – Whether to remove the version number after the colon sometimes found in ensembl gene ids.
- **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:
  - None: All genes in assembly.
  - "protein_coding": All protein coding genes in assembly.

sfaira.data.DatasetBase.streamline_metadata

DatasetBase.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance to a defined output schema.

Output format are saved in ADATA_FIELDS* classes.

Note on ontology-controlled meta data: These are defined for a given format in `ADATA_FIELDS*.ontology_constrained`. They may appear in three different formats:

- original (free text) annotation
- ontology symbol
- ontology ID

During streamlining, these ontology-controlled meta data are projected to all of these three different formats. The initially annotated column may be any of these and is defined as "{attr}_obs_col". The resulting three column per meta data item are named:

- ontology symbol: "{ADATA_FIELDS*.attr}"

- ontology ID: {ADATA_FIELDS*.attr}_{ADATA_FIELDS*.onto_id_suffix}"
- original (free text) annotation: "{ADATA_FIELDS*.attr}_{ADATA_FIELDS*.onto_original_suffix}"

Parameters

- **schema** – Export format. - "sfaira" - "cellxgene"
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.
- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.
- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.DatasetBase.subset_cells

DatasetBase.**subset_cells**(*key*, *values*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - "assay_sc" points to self.assay_sc_obs_key
  - "assay_differentiation" points to self.assay_differentiation_obs_key
  - "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
  - "cell_line" points to self.cell_line
  - "cell_type" points to self.cell_type_obs_key
  - "developmental_stage" points to self.developmental_stage_obs_key
  - "ethnicity" points to self.ethnicity_obs_key
  - "organ" points to self.organ_obs_key
  - "organism" points to self.organism_obs_key
  - "sample_source" points to self.sample_source_obs_key
  - "sex" points to self.sex_obs_key
  - "state_exact" points to self.state_exact_obs_key
- **values** – Classes to overlap to.

Returns

sfaira.data.DatasetBase.write_distributed_store

DatasetBase.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: Optional[dict] = None*, *chunks: Optional[int] = None*, *shuffle_data: bool = False*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets.

Parameters

- **dir_cache** – Directory to write cache in.
- **store_format** – Disk format for objects in cache. Recommended is "dao".
  - **"h5ad": Allows access via backed .h5ad.**

    Note on compression: .h5ad supports sparse data with is a good compression that gives fast row-w
    access if the files are csr, so further compression potentially not necessary.
  - "dao": Distributed access optimised format, recommended for batched access in optimisation, for example.
- **dense** – Whether to write sparse or dense store, this will be homogenously enforced.
- **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format=="h5ad", see also anndata.AnnData.write_h5ad:
  - compression,
  - compression_opts.

  For store_format=="dao", see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:
  - compressor
  - overwrite
  - order

  and others.
- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store=="dao". The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.
- **shuffle_data** – If True -> shuffle ordering of cells in datasets before writing store

sfaira.data.DatasetBase.write_meta

DatasetBase.**write_meta**(*fn_meta: Optional[str] = None*, *dir_out: Optional[str] = None*)

Write meta data object for data set.

Does not cache data and attempts to load raw data.

Parameters

- **fn_meta** – File to write to, selects automatically based on self.meta_path and self.id otherwise.
- **dir_out** – Path to write to, file name is selected automatically based on self.id.

Returns

sfaira.data.DatasetBase.write_ontology_class_maps

DatasetBase.**write_ontology_class_maps**(*fn*, *attrs: List[str]*, *protected_writing: bool = True*, ***kwargs*)

Load class maps of ontology-controlled field to ontology classes.

TODO: deprecate and only keep DatasetGroup writing?

Parameters

- **fn** – File name of tsv to write class maps to.
- **attrs** – Attributes to create a tsv for. Must correspond to *_obs_key in yaml.
- **protected_writing** – Only write if file was not already found.

Returns

sfaira.data.DatasetGroup

class sfaira.data.**DatasetGroup**(*datasets: dict*, *collection_id: str = 'default'*)

Container class that co-manages multiple data sets, removing need to call Dataset() methods directly through wrapping them.

Example:

#query loaders lung #from sfaira.dev.data.loaders.lung import DatasetGroupLung as DatasetGroup #dsg_humanlung = DatasetGroupHuman(path='path/to/data') #dsg_humanlung.load_all(match_to_reference='Homo_sapiens_GRCh38_97') #dsg_humanlung[some_id] #dsg_humanlung.adata

Attributes

adata

adata_ls

additional_annotation_key
"

continues on next page

Table 4 – continued from previous page

collection_id	
doi	Propagates DOI annotation from contained datasets.
ids	
ontology_celltypes	use might be replaced by ontology_container_sfaira in the future.
ontology_container_sfaira	
supplier	Propagates supplier annotation from contained datasets.
datasets	

sfaira.data.DatasetGroup.adata

property DatasetGroup.**adata**

sfaira.data.DatasetGroup.adata_ls

property DatasetGroup.**adata_ls**

sfaira.data.DatasetGroup.additional_annotation_key

property DatasetGroup.**additional_annotation_key: Dict[str, Union[None, str]]**

” Return dictionary of additional_annotation_key for each data set with ids as keys.

sfaira.data.DatasetGroup.collection_id

property DatasetGroup.**collection_id**

sfaira.data.DatasetGroup.doi

property DatasetGroup.**doi: List[str]**

Propagates DOI annotation from contained datasets.

sfaira.data.DatasetGroup.ids

property DatasetGroup.**ids**

sfaira.data.DatasetGroup.ontology_celltypes

property DatasetGroup.**ontology_celltypes**
use might be replaced by ontology_container_sfaira in the future.

Type # TODO

sfaira.data.DatasetGroup.ontology_container_sfaira

property DatasetGroup.**ontology_container_sfaira**

sfaira.data.DatasetGroup.supplier

property DatasetGroup.**supplier: List[str]**
Propagates supplier annotation from contained datasets.

sfaira.data.DatasetGroup.datasets

DatasetGroup.**datasets: Dict[str, *sfaira.data.dataloaders.base.dataset.DatasetBase*]**

Methods

collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
load([annotated_only, load_raw, ...])	Load all datasets in group (option for temporary loading).
ncells([annotated_only])	
ncells_bydataset([annotated_only])	
obs_concat([keys])	Returns concatenation of all .obs.
project_celltypes_to_ontology([...])	Project free text cell type names to ontology based on mapping table.
show_summary()	

continues on next page

Table 5 – continued from previous page

streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. :param match_to_release: Which genome annotation release to map the feature space to. Note that assemblies from ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be: - str: Provide the name of the release. - dict: Mapping of organism to name of the release (see str format). Chooses release for each data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids. :param subset_genes_to_type: Type(s) to subset to. Can be a single type or a list of types or None. Types can be: - None: All genes in assembly. - "protein_coding": All protein coding genes in assembly.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance in each data set to output format.
subset(key, values)	Subset list of adata objects based on sample-wise properties.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_backed(adata_backed, genome, idx[, ...])	Loads data set group into slice of backed anndata object.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.
write_ontology_class_maps(fn, attrs[, ...])	Write cell type maps of free text cell types to ontology classes.

sfaira.data.DatasetGroup.collapse_counts

DatasetGroup.**collapse_counts**()

Collapse count matrix along duplicated index.

sfaira.data.DatasetGroup.download

DatasetGroup.**download**(***kwargs*)

sfaira.data.DatasetGroup.load

DatasetGroup.**load**(*annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*, *processes: int = 1*, *func=None*, *kwargs_func: Optional[dict] = None*, *verbose: int = 0*, ***kwargs*)

Load all datasets in group (option for temporary loading).

Note: This method automatically subsets to the group to the data sets for which input files were found.

This method also allows temporarily loading data sets to execute function on loaded data sets (supply func). In this setting, datasets are removed from memory after the function has been executed.

param annotated_only

param load_raw

param allow_caching

param processes Processes to parallelise loading over. Uses python multiprocessing if > 1, for loop otherwise.

param func Function to run on loaded datasets. map_fun should only take one argument, which is a Dataset instance. The return can be empty:

def func(dataset, **kwargs_func): # code manipulating dataset and generating output x. return x

param kwargs_func Kwargs of func.

param verbose Verbosity of description of loading failure.

- 0: no indication of failure
- 1: indication of which data set failed in warning
- 2: 1 with error report in warning
- 3: reportin as in 2 but aborts with OSError

Parameters

- **remove_gene_version** – Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
- **match_to_reference** – Reference genomes name or False to keep original feature space.
- **load_raw** – Loads unprocessed version of data if available in data loader.
- **allow_caching** – Whether to allow method to cache adata object for faster reloading.

sfaira.data.DatasetGroup.ncells

DatasetGroup.**ncells**(*annotated_only: bool = False*)

sfaira.data.DatasetGroup.ncells_bydataset

DatasetGroup.**ncells_bydataset**(*annotated_only: bool = False*) → numpy.ndarray

sfaira.data.DatasetGroup.obs_concat

DatasetGroup.**obs_concat**(*keys: Optional[list] = None*)

Returns concatenation of all .obs.

Uses union of all keys if keys is not provided.

Parameters **keys** –

Returns

sfaira.data.DatasetGroup.project_celltypes_to_ontology

DatasetGroup.**project_celltypes_to_ontology**(*adata_fields: Optional[sfaira.consts.adata_fields.AdataIds] = None*, *copy=False*)

Project free text cell type names to ontology based on mapping table. :return:

sfaira.data.DatasetGroup.show_summary

DatasetGroup.**show_summary**()

sfaira.data.DatasetGroup.streamline_features

DatasetGroup.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. :param match_to_release: Which genome annotation release to map the feature space to. Note that assemblies from

ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be:

- str: Provide the name of the release.
- **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids.

Parameters **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:

- None: All genes in assembly.
- ”protein_coding”: All protein coding genes in assembly.

sfaira.data.DatasetGroup.streamline_metadata

DatasetGroup.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance in each data set to output format. Output format are saved in ADATA_FIELDS* classes.

Parameters

- **schema** – Export format. - "sfaira" - "cellxgene"
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.
- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.
- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.DatasetGroup.subset

DatasetGroup.**subset**(*key*, *values: Union[list, tuple, numpy.ndarray]*)

Subset list of adata objects based on sample-wise properties.

These keys are properties that are available in lazy model. Subsetting happens on .datasets.

Parameters

- **key** – Property to subset by.
- **values** – Classes to overlap to. Return if elements match any of these classes.

Returns

sfaira.data.DatasetGroup.subset_cells

DatasetGroup.**subset_cells**(*key*, *values: Union[str, List[str]]*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - "assay_differentiation" points to self.assay_differentiation_obs_key
  - "assay_sc" points to self.assay_sc_obs_key
  - "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
  - "cell_line" points to self.cell_line
  - "cell_type" points to self.cell_type_obs_key
  - "developmental_stage" points to self.developmental_stage_obs_key
  - "ethnicity" points to self.ethnicity_obs_key
  - "organ" points to self.organ_obs_key
  - "organism" points to self.organism_obs_key
  - "sample_source" points to self.sample_source_obs_key
  - "sex" points to self.sex_obs_key
  - "state_exact" points to self.state_exact_obs_key
- **values** – Classes to overlap to.

Returns

sfaira.data.DatasetGroup.write_backed

DatasetGroup.**write_backed**(*adata_backed: anndata._core.anndata.AnnData*, *genome: str*, *idx: List[numpy.ndarray]*, *annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*)

Loads data set group into slice of backed anndata object.

Subsets self.datasets to the data sets that were found. Note that feature space is automatically formatted as this is necessary for concatenation.

Parameters

- **adata_backed** – Anndata instance to load into.
- **genome** – Genome container target genomes loaded.
- **idx** – Indices in adata_backed to write observations to. This can be used to immediately create a shuffled object. This has to be a list of the length of self.data, one index array for each dataset.
- **annotated_only** –
- **load_raw** – See .load().
- **allow_caching** – See .load().

Returns New row index for next element to be written into backed anndata.

sfaira.data.DatasetGroup.write_distributed_store

DatasetGroup.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: dict = {}*, *chunks: Optional[int] = None*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets. This method writes a separate file for each data set in this object.

Parameters

- **dir_cache** – Directory to write cache in.
- **store_format** – Disk format for objects in cache. Recommended is "dao".
  - **"h5ad": Allows access via backed .h5ad.**

    Note on compression: .h5ad supports sparse data with is a good compression that gives fast ro
    access if the files are csr, so further compression potentially not necessary.
  - "dao": Distributed access optimised format, recommended for batched access in optimisation, for example.
- **dense** – Whether to write sparse or dense store, this will be homogenously enforced.
- **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format=="h5ad", see also anndata.AnnData.write_h5ad:
  - compression,
  - compression_opts.

  For store_format=="dao", see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:
  - compressor
  - overwrite
  - order

  and others.
- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store=="dao". The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.

sfaira.data.DatasetGroup.write_ontology_class_maps

DatasetGroup.**write_ontology_class_maps**(*fn*, *attrs: List[str]*, *protected_writing: bool = True*, ***kwargs*)

Write cell type maps of free text cell types to ontology classes.

Parameters

- **fn** – File name of tsv to write class maps to.
- **attrs** – Attributes to create a tsv for. Must correspond to *_obs_key in yaml.
- **protected_writing** – Only write if file was not already found.

sfaira.data.DatasetGroupDirectoryOriented

class sfaira.data.**DatasetGroupDirectoryOriented**(*file_base: str*, *data_path: Optional[str] = None*, *meta_path: Optional[str] = None*, *cache_path: Optional[str] = None*)

Attributes

adata	
adata_ls	
additional_annotation_key	"
collection_id	
doi	Propagates DOI annotation from contained datasets.
ids	
ontology_celltypes	use might be replaced by ontology_container_sfaira in the future.
ontology_container_sfaira	
supplier	Propagates supplier annotation from contained datasets.

sfaira.data.DatasetGroupDirectoryOriented.adata

property DatasetGroupDirectoryOriented.**adata**

sfaira.data.DatasetGroupDirectoryOriented.adata_ls

property DatasetGroupDirectoryOriented.**adata_ls**

sfaira.data.DatasetGroupDirectoryOriented.additional_annotation_key

property DatasetGroupDirectoryOriented.**additional_annotation_key: Dict[str, Union[None, str]]**

" Return dictionary of additional_annotation_key for each data set with ids as keys.

sfaira.data.DatasetGroupDirectoryOriented.collection_id

property DatasetGroupDirectoryOriented.**collection_id**

sfaira.data.DatasetGroupDirectoryOriented.doi

property DatasetGroupDirectoryOriented.**doi: List[str]**

Propagates DOI annotation from contained datasets.

sfaira.data.DatasetGroupDirectoryOriented.ids

property DatasetGroupDirectoryOriented.**ids**

sfaira.data.DatasetGroupDirectoryOriented.ontology_celltypes

property DatasetGroupDirectoryOriented.**ontology_celltypes**

use might be replaced by ontology_container_sfaira in the future.

Type # TODO

sfaira.data.DatasetGroupDirectoryOriented.ontology_container_sfaira

property DatasetGroupDirectoryOriented.**ontology_container_sfaira**

sfaira.data.DatasetGroupDirectoryOriented.supplier

property DatasetGroupDirectoryOriented.**supplier: List[str]**

Propagates supplier annotation from contained datasets.

Methods

clean_ontology_class_maps()	Finalises processed class maps of free text labels to ontology classes.
collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
load([annotated_only, load_raw, ...])	Load all datasets in group (option for temporary loading).
ncells([annotated_only])	
ncells_bydataset([annotated_only])	
obs_concat([keys])	Returns concatenation of all .obs.
project_celltypes_to_ontology([...])	Project free text cell type names to ontology based on mapping table.
show_summary()	
streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. :param match_to_release: Which genome annotation release to map the feature space to. Note that assemblies from ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be: - str: Provide the name of the release. - dict: Mapping of organism to name of the release (see str format). Chooses release for each data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids. :param subset_genes_to_type: Type(s) to subset to. Can be a single type or a list of types or None. Types can be: - None: All genes in assembly. - "protein_coding": All protein coding genes in assembly.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance in each data set to output format.
subset(key, values)	Subset list of adata objects based on sample-wise properties.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_backed(adata_backed, genome, idx[, ...])	Loads data set group into slice of backed anndata object.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.
write_ontology_class_maps(fn, attrs[, ...])	Write cell type maps of free text cell types to ontology classes.

sfaira.data.DatasetGroupDirectoryOriented.clean_ontology_class_maps

DatasetGroupDirectoryOriented.**clean_ontology_class_maps**()

Finalises processed class maps of free text labels to ontology classes.

Checks that the assigned ontology class names appear in the ontology. Adds a third column with the corresponding ontology IDs into the file.

Returns

sfaira.data.DatasetGroupDirectoryOriented.collapse_counts

DatasetGroupDirectoryOriented.**collapse_counts**()

Collapse count matrix along duplicated index.

sfaira.data.DatasetGroupDirectoryOriented.download

DatasetGroupDirectoryOriented.**download**(***kwargs*)

sfaira.data.DatasetGroupDirectoryOriented.load

DatasetGroupDirectoryOriented.**load**(*annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*, *processes: int = 1*, *func=None*, *kwargs_func: Optional[dict] = None*, *verbose: int = 0*, ***kwargs*)

Load all datasets in group (option for temporary loading).

Note: This method automatically subsets to the group to the data sets for which input files were found.

This method also allows temporarily loading data sets to execute function on loaded data sets (supply func). In this setting, datasets are removed from memory after the function has been executed.

param annotated_only

param load_raw

param allow_caching

param processes Processes to parallelise loading over. Uses python multiprocessing if > 1, for loop otherwise.

param func Function to run on loaded datasets. map_fun should only take one argument, which is a Dataset instance. The return can be empty:

def func(dataset, **kwargs_func): # code manipulating dataset and generating output x. return x

param kwargs_func Kwargs of func.

param verbose Verbosity of description of loading failure.

- 0: no indication of failure
- 1: indication of which data set failed in warning

- 2: 1 with error report in warning
- 3: reportin as in 2 but aborts with OSError

Parameters

- **remove_gene_version** – Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
- **match_to_reference** – Reference genomes name or False to keep original feature space.
- **load_raw** – Loads unprocessed version of data if available in data loader.
- **allow_caching** – Whether to allow method to cache adata object for faster reloading.

sfaira.data.DatasetGroupDirectoryOriented.ncells

DatasetGroupDirectoryOriented.**ncells**(*annotated_only: bool = False*)

sfaira.data.DatasetGroupDirectoryOriented.ncells_bydataset

DatasetGroupDirectoryOriented.**ncells_bydataset**(*annotated_only: bool = False*) → numpy.ndarray

sfaira.data.DatasetGroupDirectoryOriented.obs_concat

DatasetGroupDirectoryOriented.**obs_concat**(*keys: Optional[list] = None*)

Returns concatenation of all .obs.

Uses union of all keys if keys is not provided.

Parameters **keys** –

Returns

sfaira.data.DatasetGroupDirectoryOriented.project_celltypes_to_ontology

DatasetGroupDirectoryOriented.**project_celltypes_to_ontology**(*adata_fields: Optional[sfaira.consts.adata_fields.AdataIds] = None*, *copy=False*)

Project free text cell type names to ontology based on mapping table. :return:

sfaira.data.DatasetGroupDirectoryOriented.show_summary

DatasetGroupDirectoryOriented.**show_summary**()

sfaira.data.DatasetGroupDirectoryOriented.streamline_features

DatasetGroupDirectoryOriented.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. :param match_to_release: Which genome annotation release to map the feature space to. Note that assemblies from

ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be:

- str: Provide the name of the release.
- **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids.

Parameters **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:

- None: All genes in assembly.
- "protein_coding": All protein coding genes in assembly.

sfaira.data.DatasetGroupDirectoryOriented.streamline_metadata

DatasetGroupDirectoryOriented.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance in each data set to output format. Output format are saved in ADATA_FIELDS* classes.

Parameters

- **schema** – Export format. - "sfaira" - "cellxgene"
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.
- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.

- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.DatasetGroupDirectoryOriented.subset

DatasetGroupDirectoryOriented.**subset**(*key*, *values: Union[list, tuple, numpy.ndarray]*)

Subset list of adata objects based on sample-wise properties.

These keys are properties that are available in lazy model. Subsetting happens on .datasets.

Parameters

- **key** – Property to subset by.
- **values** – Classes to overlap to. Return if elements match any of these classes.

Returns

sfaira.data.DatasetGroupDirectoryOriented.subset_cells

DatasetGroupDirectoryOriented.**subset_cells**(*key*, *values: Union[str, List[str]]*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key
  - ”sample_source” points to self.sample_source_obs_key
  - ”sex” points to self.sex_obs_key
  - ”state_exact” points to self.state_exact_obs_key
- **values** – Classes to overlap to.

Returns

sfaira.data.DatasetGroupDirectoryOriented.write_backed

DatasetGroupDirectoryOriented.**write_backed**(*adata_backed: anndata._core.anndata.AnnData*, *genome: str*, *idx: List[numpy.ndarray]*, *annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*)

Loads data set group into slice of backed anndata object.

Subsets self.datasets to the data sets that were found. Note that feature space is automatically formatted as this is necessary for concatenation.

Parameters

- **adata_backed** – Anndata instance to load into.
- **genome** – Genome container target genomes loaded.
- **idx** – Indices in adata_backed to write observations to. This can be used to immediately create a shuffled object. This has to be a list of the length of self.data, one index array for each dataset.
- **annotated_only** –
- **load_raw** – See .load().
- **allow_caching** – See .load().

Returns New row index for next element to be written into backed anndata.

sfaira.data.DatasetGroupDirectoryOriented.write_distributed_store

DatasetGroupDirectoryOriented.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: dict = {}*, *chunks: Optional[int] = None*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets. This method writes a separate file for each data set in this object.

Parameters

- **dir_cache** – Directory to write cache in.
- **store_format** – Disk format for objects in cache. Recommended is "dao".
  - **"h5ad": Allows access via backed .h5ad.**

    Note on compression: .h5ad supports sparse data with is a good compression that gives fast ro
    access if the files are csr, so further compression potentially not necessary.
  - "dao": Distributed access optimised format, recommended for batched access in optimisation, for example.
- **dense** – Whether to write sparse or dense store, this will be homogenously enforced.
- **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format=="h5ad", see also anndata.AnnData.write_h5ad:
  - compression,
  - compression_opts.

For store_format==”dao”, see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:

- compressor
- overwrite
- order

and others.

- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store==”dao”. The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.

sfaira.data.DatasetGroupDirectoryOriented.write_ontology_class_maps

DatasetGroupDirectoryOriented.**write_ontology_class_maps**(*fn*, *attrs: List[str]*, *protected_writing: bool = True*, ***kwargs*)

Write cell type maps of free text cell types to ontology classes.

Parameters

- **fn** – File name of tsv to write class maps to.
- **attrs** – Attributes to create a tsv for. Must correspond to *_obs_key in yaml.
- **protected_writing** – Only write if file was not already found.

sfaira.data.DatasetSuperGroup

class sfaira.data.**DatasetSuperGroup**(*dataset_groups: Union[None, List[sfaira.data.dataloaders.base.dataset_group.DatasetGroup], List[sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup]]*)

Container for multiple DatasetGroup instances.

Used to manipulate structured dataset collections. Primarly designed for this manipulation, convert to DatasetGroup via flatten() for more functionalities.

Attributes

adata	
adata_ls	
additional_annotation_key	"
datasets	Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.
ids	

continues on next page

Table 8 – continued from previous page

fn_backed
dataset_groups

sfaira.data.DatasetSuperGroup.adata

property DatasetSuperGroup.**adata**

sfaira.data.DatasetSuperGroup.adata_ls

property DatasetSuperGroup.**adata_ls**

sfaira.data.DatasetSuperGroup.additional_annotation_key

property DatasetSuperGroup.**additional_annotation_key**: List[Dict[str, Union[None, str]]]

” Return list (by data set group) of dictionaries of additional_annotation_key for each data set with ids as keys.

sfaira.data.DatasetSuperGroup.datasets

property DatasetSuperGroup.**datasets**: Dict[str, *sfaira.data.dataloaders.base.dataset.DatasetBase*]

Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.

Returns

sfaira.data.DatasetSuperGroup.ids

property DatasetSuperGroup.**ids**

sfaira.data.DatasetSuperGroup.fn_backed

DatasetSuperGroup.**fn_backed**: Union[None, os.PathLike]

sfaira.data.DatasetSuperGroup.dataset_groups

DatasetSuperGroup.**dataset_groups**: Union[list, List[*sfaira.data.dataloaders.base.dataset_group.DatasetGroup*], List[*sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup*]]

Methods

collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
extend_dataset_groups(dataset_groups)	
flatten()	Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.
get_gc([genome])	
load([annotated_only, load_raw, ...])	Loads data set homosapiens into anndata object.
load_config(fn)	Load a config file and recreates a data sub-setting.
ncells([annotated_only])	
ncells_bydataset([annotated_only])	List of list of length of all data sets by data set group.
ncells_bydataset_flat([annotated_only])	Flattened list of length of all data sets.
project_celltypes_to_ontology([...])	Project free text cell type names to ontology based on mapping table.
remove_duplicates([supplier_hierarchy])	Remove duplicate data loaders from super group, e.g.
set_dataset_groups(dataset_groups)	
show_summary()	
streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance in each group and each data set to output format.
subset(key, values)	Subset list of adata objects based on match to values in key property.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.

sfaira.data.DatasetSuperGroup.collapse_counts

DatasetSuperGroup.**collapse_counts**()
Collapse count matrix along duplicated index.

sfaira.data.DatasetSuperGroup.download

DatasetSuperGroup.**download**(***kwargs*)

sfaira.data.DatasetSuperGroup.extend_dataset_groups

DatasetSuperGroup.**extend_dataset_groups**(*dataset_groups: Union[List[*sfaira.data.dataloaders.base.dataset_group.DatasetGroup*], List[*sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup*]]*)

sfaira.data.DatasetSuperGroup.flatten

DatasetSuperGroup.**flatten**() → *sfaira.data.dataloaders.base.dataset_group.DatasetGroup*

Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.

Returns

sfaira.data.DatasetSuperGroup.get_gc

DatasetSuperGroup.**get_gc**(*genome: Optional[str] = None*)

sfaira.data.DatasetSuperGroup.load

DatasetSuperGroup.**load**(*annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*, *processes: int = 1*, ***kwargs*)

Loads data set homosapiens into anndata object.

Parameters

- **annotated_only** –
- **load_raw** – See .load().
- **allow_caching** – See .load().
- **processes** – Processes to parallelise loading over. Uses python multiprocessing if > 1, for loop otherwise. Note: parallelises loading of each dataset group, but not across groups.

Returns

sfaira.data.DatasetSuperGroup.load_config

DatasetSuperGroup.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting.

Parameters **fn** – Output file.

sfaira.data.DatasetSuperGroup.ncells

DatasetSuperGroup.**ncells**(*annotated_only: bool = False*)

sfaira.data.DatasetSuperGroup.ncells_bydataset

DatasetSuperGroup.**ncells_bydataset**(*annotated_only: bool = False*)
List of list of length of all data sets by data set group. :return:

sfaira.data.DatasetSuperGroup.ncells_bydataset_flat

DatasetSuperGroup.**ncells_bydataset_flat**(*annotated_only: bool = False*)
Flattened list of length of all data sets. :return:

sfaira.data.DatasetSuperGroup.project_celltypes_to_ontology

DatasetSuperGroup.**project_celltypes_to_ontology**(*adata_fields: Optional[sfaira.consts.adata_fields.AdataIds] = None*, *copy=False*)
Project free text cell type names to ontology based on mapping table. :return:

sfaira.data.DatasetSuperGroup.remove_duplicates

DatasetSuperGroup.**remove_duplicates**(*supplier_hierarchy: str = 'cellxgene,sfaira'*)
Remove duplicate data loaders from super group, e.g. loaders that map to the same DOI.

Any DOI match is removed (pre-print or journal publication). Data sets without DOI are removed, too. Loaders are kept in the hierarchy indicated in supplier_hierarchy. Requires a super group with homogenous suppliers across DatasetGroups, throws an error otherwise. This is given for sfaira maintained libraries but may not be the case if custom assembled DatasetGroups are used.

Parameters **supplier_hierarchy** – Hierarchy to resolve duplications by. Comma separated string that indicates which data provider takes priority. Choose "cellxgene,sfaira" to prioritise use of data sets downloaded from cellxgene. Choose "sfaira,cellxgene" to prioritise use of raw data processing pipelines locally.

- cellxgene: cellxgene downloads
- sfaira: local raw file processing

Returns

sfaira.data.DatasetSuperGroup.set_dataset_groups

DatasetSuperGroup.**set_dataset_groups**(*dataset_groups: Union[sfaira.data.dataloaders.base.dataset_group.DatasetGroup, sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup, List[sfaira.data.dataloaders.base.dataset_group.DatasetGroup], List[sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup]]*)

sfaira.data.DatasetSuperGroup.show_summary

DatasetSuperGroup.**show_summary**()

sfaira.data.DatasetSuperGroup.streamline_features

DatasetSuperGroup.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.

Parameters

- **match_to_release** – Which genome annotation release to map the feature space to. Note that assemblies from ensembl are usually named as Organism.Assembly.Release, this is the Release string. Can be:
  - str: Provide the name of the release.
  - **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids.
- **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:
  - None: All genes in assembly.
  - "protein_coding": All protein coding genes in assembly.

sfaira.data.DatasetSuperGroup.streamline_metadata

DatasetSuperGroup.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance in each group and each data set to output format. Output format are saved in ADATA_FIELDS* classes.

Parameters

- **schema** – Export format. - "sfaira" - "cellxgene"
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.

- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.
- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.DatasetSuperGroup.subset

DatasetSuperGroup.**subset**(*key*, *values*)

Subset list of adata objects based on match to values in key property.

These keys are properties that are available in lazy model. Subsetting happens on .datasets.

Parameters

- **key** – Property to subset by.
- **values** – Classes to overlap to.

Returns

sfaira.data.DatasetSuperGroup.subset_cells

DatasetSuperGroup.**subset_cells**(*key*, *values: Union[str, List[str]]*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key

- ”sample_source” points to self.sample_source_obs_key
- ”sex” points to self.sex_obs_key
- ”state_exact” points to self.state_exact_obs_key

• **values** – Classes to overlap to.

Returns

sfaira.data.DatasetSuperGroup.write_config

DatasetSuperGroup.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting.

Parameters **fn** – Output file.

sfaira.data.DatasetSuperGroup.write_distributed_store

DatasetSuperGroup.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: dict = {}*, *chunks: Optional[int] = None*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets. This method writes a separate file for each data set in this object.

Parameters

• **dir_cache** – Directory to write cache in.

• **store_format** – Disk format for objects in cache. Recommended is “dao”.

- **”h5ad”: Allows access via backed .h5ad.**

  Note on compression: .h5ad supports sparse data with is a good compression that gives fast ro
  access if the files are csr, so further compression potentially not necessary.

- ”dao”: Distributed access optimised format, recommended for batched access in optimisation, for example.

• **dense** – Whether to write sparse or dense store, this will be homogenously enforced.

• **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format==”h5ad”, see also anndata.AnnData.write_h5ad:

- compression,
- compression_opts.

For store_format==”dao”, see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:

- compressor
- overwrite
- order

and others.

- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store=="dao". The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.

Interactive data class to use a loaded data object in the context sfaira tools:

data.DatasetInteractive(data[, ...])	

sfaira.data.DatasetInteractive

class sfaira.data.**DatasetInteractive**(*data: anndata._core.anndata.AnnData*, *feature_symbol_col: Optional[str] = 'index'*, *feature_id_col: Optional[str] = None*, *feature_type_col: Optional[str] = None*, *dataset_id: str = 'interactive_dataset'*, *data_path: Optional[str] = '.'*, *meta_path: Optional[str] = '.'*, *cache_path: Optional[str] = '.'*)

Attributes

additional_annotation_key	
annotated	
assay_differentiation	
assay_sc	
assay_type_differentiation	
author	
bio_sample	
bio_sample_obs_key	
cache_fn	
cell_line	
cell_type	
celltypes_universe	
citation	Return all information necessary to cite data set.
data_dir	
default_embedding	

continues on next page

Table 11 – continued from previous page

development_stage	
directory_formatted_doi	
disease	
doi	All publication DOI associated with the study which are the journal publication and the preprint.
doi_cleaned_id	
doi_journal	The prepring publication (secondary) DOI associated with the study.
doi_main	The main DOI associated with the study which is the journal publication if available, otherwise the preprint.
doi_preprint	The journal publication (main) DOI associated with the study.
download_url_data	Data download website(s).
download_url_meta	Meta data download website(s).
ethnicity	
feature_reference	
feature_type	
id	
individual	
loaded	return: Whether DataSet was loaded into memory.
meta	
meta_fn	
ncells	
ontology_class_maps	
organ	
organism	
primary_data	
sample_source	
sex	
source	

continues on next page

Table 11 – continued from previous page

source_doi
state_exact
tech_sample
tech_sample_obs_key
title
year

sfaira.data.DatasetInteractive.additional_annotation_key

property DatasetInteractive.**additional_annotation_key:** Union[None, str]

sfaira.data.DatasetInteractive.annotated

property DatasetInteractive.**annotated:** Optional[bool]

sfaira.data.DatasetInteractive.assay_differentiation

property DatasetInteractive.**assay_differentiation:** Union[None, str]

sfaira.data.DatasetInteractive.assay_sc

property DatasetInteractive.**assay_sc:** Union[None, str]

sfaira.data.DatasetInteractive.assay_type_differentiation

property DatasetInteractive.**assay_type_differentiation:** Union[None, str]

sfaira.data.DatasetInteractive.author

property DatasetInteractive.**author:** str

sfaira.data.DatasetInteractive.bio_sample

property DatasetInteractive.**bio_sample**: Union[None, str]

sfaira.data.DatasetInteractive.bio_sample_obs_key

property DatasetInteractive.**bio_sample_obs_key**: Union[None, str]

sfaira.data.DatasetInteractive.cache_fn

property DatasetInteractive.**cache_fn**

sfaira.data.DatasetInteractive.cell_line

property DatasetInteractive.**cell_line**: Union[None, str]

sfaira.data.DatasetInteractive.cell_type

property DatasetInteractive.**cell_type**: Union[None, str]

sfaira.data.DatasetInteractive.celltypes_universe

property DatasetInteractive.**celltypes_universe**

sfaira.data.DatasetInteractive.citation

property DatasetInteractive.**citation**

Return all information necessary to cite data set.

Returns

sfaira.data.DatasetInteractive.data_dir

property DatasetInteractive.**data_dir**

sfaira.data.DatasetInteractive.default_embedding

property DatasetInteractive.**default_embedding**: Union[None, str]

sfaira.data.DatasetInteractive.development_stage

property DatasetInteractive.development_stage: Union[None, str]

sfaira.data.DatasetInteractive.directory_formatted_doi

property DatasetInteractive.directory_formatted_doi: str

sfaira.data.DatasetInteractive.disease

property DatasetInteractive.disease: Union[None, str]

sfaira.data.DatasetInteractive.doi

property DatasetInteractive.doi: List[str]

All publication DOI associated with the study which are the journal publication and the preprint. See also .doi_preprint, .doi_journal.

sfaira.data.DatasetInteractive.doi_cleaned_id

property DatasetInteractive.doi_cleaned_id

sfaira.data.DatasetInteractive.doi_journal

property DatasetInteractive.doi_journal: str

The prepring publication (secondary) DOI associated with the study. See also .doi_journal.

sfaira.data.DatasetInteractive.doi_main

property DatasetInteractive.doi_main: str

The main DOI associated with the study which is the journal publication if available, otherwise the preprint. See also .doi_preprint, .doi_journal.

sfaira.data.DatasetInteractive.doi_preprint

property DatasetInteractive.doi_preprint: str

The journal publication (main) DOI associated with the study. See also .doi_preprint.

sfaira.data.DatasetInteractive.download_url_data

property DatasetInteractive.**download_url_data**: Union[Tuple[List[str]], Tuple[List[None]]]

Data download website(s).

Save as tuple with single element, which is a list of all download websites relevant to dataset. :return:

sfaira.data.DatasetInteractive.download_url_meta

property DatasetInteractive.**download_url_meta**: Union[Tuple[List[str]], Tuple[List[None]]]

Meta data download website(s).

Save as tuple with single element, which is a list of all download websites relevant to dataset. :return:

sfaira.data.DatasetInteractive.ethnicity

property DatasetInteractive.**ethnicity**: Union[None, str]

sfaira.data.DatasetInteractive.feature_reference

property DatasetInteractive.**feature_reference**: str

sfaira.data.DatasetInteractive.feature_type

property DatasetInteractive.**feature_type**: str

sfaira.data.DatasetInteractive.id

property DatasetInteractive.**id**: str

sfaira.data.DatasetInteractive.individual

property DatasetInteractive.**individual**: Union[None, str]

sfaira.data.DatasetInteractive.loaded

property DatasetInteractive.**loaded**: bool

return: Whether DataSet was loaded into memory.

sfaira.data.DatasetInteractive.meta

property DatasetInteractive.**meta**: Union[None, pandas.core.frame.DataFrame]

sfaira.data.DatasetInteractive.meta_fn

property DatasetInteractive.**meta_fn**

sfaira.data.DatasetInteractive.ncells

property DatasetInteractive.**ncells**: Union[None, int]

sfaira.data.DatasetInteractive.ontology_class_maps

property DatasetInteractive.**ontology_class_maps**: Dict[str, pandas.core.frame.DataFrame]

sfaira.data.DatasetInteractive.organ

property DatasetInteractive.**organ**: Union[None, str]

sfaira.data.DatasetInteractive.organism

property DatasetInteractive.**organism**: Union[None, str]

sfaira.data.DatasetInteractive.primary_data

property DatasetInteractive.**primary_data**: Union[None, bool]

sfaira.data.DatasetInteractive.sample_source

property DatasetInteractive.**sample_source**: Union[None, str]

sfaira.data.DatasetInteractive.sex

property DatasetInteractive.**sex**: Union[None, str]

sfaira.data.DatasetInteractive.source

property DatasetInteractive.**source**: str

sfaira.data.DatasetInteractive.source_doi

property DatasetInteractive.**source_doi**: str

sfaira.data.DatasetInteractive.state_exact

property DatasetInteractive.**state_exact**: Union[None, str]

sfaira.data.DatasetInteractive.tech_sample

property DatasetInteractive.**tech_sample**: Union[None, str]

sfaira.data.DatasetInteractive.tech_sample_obs_key

property DatasetInteractive.**tech_sample_obs_key**: Union[None, str]

sfaira.data.DatasetInteractive.title

property DatasetInteractive.**title**

sfaira.data.DatasetInteractive.year

property DatasetInteractive.**year**: Union[None, int]

Methods

clear()	Remove loaded .adata to reduce memory footprint.
collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
get_ontology(k)	
load([load_raw, allow_caching])	**param remove_gene_version** Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
load_meta(fn)	

continues on next page

Table 12 – continued from previous page

project_free_to_ontology(attr)	Project free text cell type names to ontology based on mapping table.
read_ontology_class_maps(fns)	Load class maps of free text class labels to ontology classes.
set_dataset_id([idx])	
show_summary()	
streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance to a defined output schema.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.
write_meta([fn_meta, dir_out])	Write meta data object for data set.
write_ontology_class_maps(fn, attrs[, ...])	Load class maps of ontology-controlled field to ontology classes.

sfaira.data.DatasetInteractive.clear

DatasetInteractive.**clear**()

Remove loaded .adata to reduce memory footprint.

Returns

sfaira.data.DatasetInteractive.collapse_counts

DatasetInteractive.**collapse_counts**()

Collapse count matrix along duplicated index.

sfaira.data.DatasetInteractive.download

DatasetInteractive.**download**(***kwargs*)

sfaira.data.DatasetInteractive.get_ontology

DatasetInteractive.**get_ontology**(*k*) → Optional[*sfaira.versions.metadata.base.OntologyHierarchical*]

sfaira.data.DatasetInteractive.load

DatasetInteractive.**load**(*load_raw: bool = False*, *allow_caching: bool = True*, ***kwargs*)

Parameters

- **remove_gene_version** – Remove gene version string from ENSEMBL ID so that different versions in different data sets are superimposed.
- **match_to_reference** – Reference genomes name or False to keep original feature space.
- **load_raw** – Loads unprocessed version of data if available in data loader.
- **allow_caching** – Whether to allow method to cache adata object for faster reloading.

sfaira.data.DatasetInteractive.load_meta

DatasetInteractive.**load_meta**(*fn: Optional[Union[os.PathLike, str]]*)

sfaira.data.DatasetInteractive.project_free_to_ontology

DatasetInteractive.**project_free_to_ontology**(*attr: str*)

Project free text cell type names to ontology based on mapping table.

ToDo: add ontology ID setting here.

sfaira.data.DatasetInteractive.read_ontology_class_maps

DatasetInteractive.**read_ontology_class_maps**(*fns: List[str]*)

Load class maps of free text class labels to ontology classes.

Parameters **fns** – File names of tsv to load class maps from.

Returns

sfaira.data.DatasetInteractive.set_dataset_id

DatasetInteractive.**set_dataset_id**(*idx: int = 1*)

sfaira.data.DatasetInteractive.show_summary

DatasetInteractive.**show_summary**()

sfaira.data.DatasetInteractive.streamline_features

DatasetInteractive.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding. This also adds missing ensid or gene symbol columns if match_to_reference is not set to False and removes all adata.var columns that are not defined as gene_id_ensembl_var_key or gene_id_symbol_var_key in the dataloader.

Parameters

- **match_to_release** – Which genome annotation release to map the feature space to. Note that assemblies from ensbeml are usually named as Organism.Assembly.Release, this is the Release string. Can be:
  - str: Provide the name of the release.
  - **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.
- **remove_gene_version** – Whether to remove the version number after the colon sometimes found in ensembl gene ids.
- **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:
  - None: All genes in assembly.
  - "protein_coding": All protein coding genes in assembly.

sfaira.data.DatasetInteractive.streamline_metadata

DatasetInteractive.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance to a defined output schema.

Output format are saved in ADATA_FIELDS* classes.

Note on ontology-controlled meta data: These are defined for a given format in `ADATA_FIELDS*.ontology_constrained`. They may appear in three different formats:

- original (free text) annotation
- ontology symbol
- ontology ID

During streamlining, these ontology-controlled meta data are projected to all of these three different formats. The initially annotated column may be any of these and is defined as "{attr}_obs_col". The resulting three column per meta data item are named:

- ontology symbol: "{ADATA_FIELDS*.attr}"
- ontology ID: {ADATA_FIELDS*.attr}_{ADATA_FIELDS*.onto_id_suffix}"
- original (free text) annotation: "{ADATA_FIELDS*.attr}_{ADATA_FIELDS*.onto_original_suffix}"

Parameters

- **schema** – Export format. - “sfaira” - “cellxgene”
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.
- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.
- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.DatasetInteractive.subset_cells

DatasetInteractive.**subset_cells**(*key*, *values*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key
  - ”sample_source” points to self.sample_source_obs_key
  - ”sex” points to self.sex_obs_key
  - ”state_exact” points to self.state_exact_obs_key
- **values** – Classes to overlap to.

Returns

sfaira.data.DatasetInteractive.write_distributed_store

DatasetInteractive.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: Optional[dict] = None*, *chunks: Optional[int] = None*, *shuffle_data: bool = False*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets.

Parameters

- **dir_cache** – Directory to write cache in.
- **store_format** – Disk format for objects in cache. Recommended is "dao".
  - **"h5ad": Allows access via backed .h5ad.**

    Note on compression: .h5ad supports sparse data with is a good compression that gives fast ro
    access if the files are csr, so further compression potentially not necessary.
  - "dao": Distributed access optimised format, recommended for batched access in optimisation, for example.
- **dense** – Whether to write sparse or dense store, this will be homogenously enforced.
- **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format=="h5ad", see also anndata.AnnData.write_h5ad:
  - compression,
  - compression_opts.

  For store_format=="dao", see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:
  - compressor
  - overwrite
  - order

  and others.
- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store=="dao". The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.
- **shuffle_data** – If True -> shuffle ordering of cells in datasets before writing store

sfaira.data.DatasetInteractive.write_meta

DatasetInteractive.**write_meta**(*fn_meta: Optional[str] = None*, *dir_out: Optional[str] = None*)

Write meta data object for data set.

Does not cache data and attempts to load raw data.

Parameters

- **fn_meta** – File to write to, selects automatically based on self.meta_path and self.id otherwise.
- **dir_out** – Path to write to, file name is selected automatically based on self.id.

Returns

sfaira.data.DatasetInteractive.write_ontology_class_maps

DatasetInteractive.**write_ontology_class_maps**(*fn*, *attrs: List[str]*, *protected_writing: bool = True*, ***kwargs*)

Load class maps of ontology-controlled field to ontology classes.

TODO: deprecate and only keep DatasetGroup writing?

Parameters

- **fn** – File name of tsv to write class maps to.
- **attrs** – Attributes to create a tsv for. Must correspond to *_obs_key in yaml.
- **protected_writing** – Only write if file was not already found.

Returns

Dataset universe to interact with all data loader classes:

data.Universe([data_path, meta_path, ...])	

sfaira.data.Universe

class sfaira.data.**Universe**(*data_path: Optional[str] = None*, *meta_path: Optional[str] = None*, *cache_path: Optional[str] = None*, *exclude_databases: bool = True*)

Attributes

adata	
adata_ls	
additional_annotation_key	"
datasets	Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.

continues on next page

Table 14 – continued from previous page

ids	

sfaira.data.Universe.adata

property Universe.**adata**

sfaira.data.Universe.adata_ls

property Universe.**adata_ls**

sfaira.data.Universe.additional_annotation_key

property Universe.**additional_annotation_key: List[Dict[str, Union[None, str]]]**

" Return list (by data set group) of dictionaries of additional_annotation_key for each data set with ids as keys.

sfaira.data.Universe.datasets

property Universe.**datasets: Dict[str, *sfaira.data.dataloaders.base.dataset.DatasetBase*]**

Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.

Returns

sfaira.data.Universe.ids

property Universe.**ids**

Methods

collapse_counts()	Collapse count matrix along duplicated index.
download(**kwargs)	
extend_dataset_groups(dataset_groups)	
flatten()	Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.
get_gc([genome])	
load([annotated_only, load_raw, ...])	Loads data set homosapiens into anndata object.
load_config(fn)	Load a config file and recreates a data sub-setting.
ncells([annotated_only])	
ncells_bydataset([annotated_only])	List of list of length of all data sets by data set group.

continues on next page

Table 15 – continued from previous page

ncells_bydataset_flat([annotated_only])	Flattened list of length of all data sets.
project_celltypes_to_ontology([...])	Project free text cell type names to ontology based on mapping table.
remove_duplicates([supplier_hierarchy])	Remove duplicate data loaders from super group, e.g.
set_dataset_groups(dataset_groups)	
show_summary()	
streamline_features([match_to_release, ...])	Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.
streamline_metadata([schema, clean_obs, ...])	Streamline the adata instance in each group and each data set to output format.
subset(key, values)	Subset list of adata objects based on match to values in key property.
subset_cells(key, values)	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.
write_distributed_store(dir_cache[, ...])	Write data set into a format that allows distributed access to data set on disk.

sfaira.data.Universe.collapse_counts

Universe.**collapse_counts**()
Collapse count matrix along duplicated index.

sfaira.data.Universe.download

Universe.**download**(***kwargs*)

sfaira.data.Universe.extend_dataset_groups

Universe.**extend_dataset_groups**(*dataset_groups: Union[List[sfaira.data.dataloaders.base.dataset_group.DatasetGroup], List[sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup]]*)

sfaira.data.Universe.flatten

Universe.**flatten**() → *sfaira.data.dataloaders.base.dataset_group.DatasetGroup*
Returns DatasetGroup (rather than self = DatasetSuperGroup) containing all listed data sets.

Returns

sfaira.data.Universe.get_gc

Universe.**get_gc**(*genome: Optional[str] = None*)

sfaira.data.Universe.load

Universe.**load**(*annotated_only: bool = False*, *load_raw: bool = False*, *allow_caching: bool = True*, *processes: int = 1*, ***kwargs*)

Loads data set homosapiens into anndata object.

Parameters

- **annotated_only** –
- **load_raw** – See .load().
- **allow_caching** – See .load().
- **processes** – Processes to parallelise loading over. Uses python multiprocessing if > 1, for loop otherwise. Note: parallelises loading of each dataset group, but not across groups.

Returns

sfaira.data.Universe.load_config

Universe.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting.

Parameters **fn** – Output file.

sfaira.data.Universe.ncells

Universe.**ncells**(*annotated_only: bool = False*)

sfaira.data.Universe.ncells_bydataset

Universe.**ncells_bydataset**(*annotated_only: bool = False*)

List of list of length of all data sets by data set group. :return:

sfaira.data.Universe.ncells_bydataset_flat

Universe.**ncells_bydataset_flat**(*annotated_only: bool = False*)

Flattened list of length of all data sets. :return:

sfaira.data.Universe.project_celltypes_to_ontology

Universe.**project_celltypes_to_ontology**(*adata_fields: Optional[sfaira.consts.adata_fields.AdataIds] = None*, *copy=False*)

Project free text cell type names to ontology based on mapping table. :return:

sfaira.data.Universe.remove_duplicates

Universe.**remove_duplicates**(*supplier_hierarchy: str = 'cellxgene,sfaira'*)

Remove duplicate data loaders from super group, e.g. loaders that map to the same DOI.

Any DOI match is removed (pre-print or journal publication). Data sets without DOI are removed, too. Loaders are kept in the hierarchy indicated in supplier_hierarchy. Requires a super group with homogenous suppliers across DatasetGroups, throws an error otherwise. This is given for sfaira maintained libraries but may not be the case if custom assembled DatasetGroups are used.

Parameters **supplier_hierarchy** – Hierarchy to resolve duplications by. Comma separated string that indicates which data provider takes priority. Choose "cellxgene,sfaira" to prioritise use of data sets downloaded from cellxgene. Choose "sfaira,cellxgene" to prioritise use of raw data processing pipelines locally.

- cellxgene: cellxgene downloads
- sfaira: local raw file processing

Returns

sfaira.data.Universe.set_dataset_groups

Universe.**set_dataset_groups**(*dataset_groups: Union[sfaira.data.dataloaders.base.dataset_group.DatasetGroup, sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup, List[sfaira.data.dataloaders.base.dataset_group.DatasetGroup], List[sfaira.data.dataloaders.base.dataset_group.DatasetSuperGroup]]*)

sfaira.data.Universe.show_summary

Universe.**show_summary**()

sfaira.data.Universe.streamline_features

Universe.**streamline_features**(*match_to_release: Optional[Union[str, Dict[str, str]]] = None*, *remove_gene_version: bool = True*, *subset_genes_to_type: Union[None, str, List[str]] = None*, *schema: Optional[str] = None*)

Subset and sort genes to genes defined in an assembly or genes of a particular type, such as protein coding.

Parameters

- **match_to_release** – Which genome annotation release to map the feature space to. Note that assemblies from ensembl are usually named as Organism.Assembly.Release, this is the Release string. Can be:
  - str: Provide the name of the release.

  - **dict: Mapping of organism to name of the release (see str format). Chooses release for each** data set based on organism annotation.:param remove_gene_version: Whether to remove the version number after the colon sometimes found in ensembl gene ids.

- **subset_genes_to_type** – Type(s) to subset to. Can be a single type or a list of types or None. Types can be:
  - None: All genes in assembly.
  - "protein_coding": All protein coding genes in assembly.

sfaira.data.Universe.streamline_metadata

Universe.**streamline_metadata**(*schema: str = 'sfaira'*, *clean_obs: bool = True*, *clean_var: bool = True*, *clean_uns: bool = True*, *clean_obs_names: bool = True*, *keep_orginal_obs: bool = False*, *keep_symbol_obs: bool = True*, *keep_id_obs: bool = True*)

Streamline the adata instance in each group and each data set to output format. Output format are saved in ADATA_FIELDS* classes.

Parameters

- **schema** – Export format. - "sfaira" - "cellxgene"
- **clean_obs** – Whether to delete non-streamlined fields in .obs, .obsm and .obsp.
- **clean_var** – Whether to delete non-streamlined fields in .var, .varm and .varp.
- **clean_uns** – Whether to delete non-streamlined fields in .uns.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **clean_obs_names** – Whether to replace obs_names with a string comprised of dataset id and an increasing integer.
- **keep_orginal_obs** – For ontology-constrained .obs columns, whether to keep a column with original annotation.
- **keep_symbol_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology symbol annotation.
- **keep_id_obs** – For ontology-constrained .obs columns, whether to keep a column with ontology ID annotation.

Returns

sfaira.data.Universe.subset

Universe.**subset**(*key*, *values*)

Subset list of adata objects based on match to values in key property.

These keys are properties that are available in lazy model. Subsetting happens on .datasets.

Parameters

- **key** – Property to subset by.
- **values** – Classes to overlap to.

Returns

sfaira.data.Universe.subset_cells

Universe.**subset_cells**(*key*, *values: Union[str, List[str]]*)

Subset list of adata objects based on cell-wise properties.

These keys are properties that are not available in lazy model and require loading first because the subsetting works on the cell-level: .adata are maintained but reduced to matches.

Parameters

- **key** – Property to subset by. Options:
  - "assay_sc" points to self.assay_sc_obs_key
  - "assay_differentiation" points to self.assay_differentiation_obs_key
  - "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
  - "cell_line" points to self.cell_line
  - "cell_type" points to self.cell_type_obs_key
  - "developmental_stage" points to self.developmental_stage_obs_key
  - "ethnicity" points to self.ethnicity_obs_key
  - "organ" points to self.organ_obs_key
  - "organism" points to self.organism_obs_key
  - "sample_source" points to self.sample_source_obs_key
  - "sex" points to self.sex_obs_key
  - "state_exact" points to self.state_exact_obs_key
- **values** – Classes to overlap to.

Returns

sfaira.data.Universe.write_config

Universe.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting.

Parameters **fn** – Output file.

sfaira.data.Universe.write_distributed_store

Universe.**write_distributed_store**(*dir_cache: Union[str, os.PathLike]*, *store_format: str = 'dao'*, *dense: bool = False*, *compression_kwargs: dict = {}*, *chunks: Optional[int] = None*)

Write data set into a format that allows distributed access to data set on disk.

Stores are useful for distributed access to data sets, in many settings this requires some streamlining of the data sets that are accessed. Use .streamline_* before calling this method to streamline the data sets. This method writes a separate file for each data set in this object.

Parameters

- **dir_cache** – Directory to write cache in.

- **store_format** – Disk format for objects in cache. Recommended is "dao".
  - **"h5ad": Allows access via backed .h5ad.**

    Note on compression: .h5ad supports sparse data with is a good compression that gives fast ro
    access if the files are csr, so further compression potentially not necessary.
  - "dao": Distributed access optimised format, recommended for batched access in optimisation, for example.
- **dense** – Whether to write sparse or dense store, this will be homogenously enforced.
- **compression_kwargs** – Compression key word arguments to give to h5py or zarr For store_format=="h5ad", see also anndata.AnnData.write_h5ad:
  - compression,
  - compression_opts.

  For store_format=="dao", see also sfaira.data.write_dao which relays kwargs to zarr.hierarchy.create_dataset:
  - compressor
  - overwrite
  - order

  and others.
- **chunks** – Observation axes of chunk size of zarr array, see anndata.AnnData.write_zarr documentation. Only relevant for store=="dao". The feature dimension of the chunks is always is the full feature space. Uses zarr default chunking across both axes if None.

Stores

We distinguish stores for a single feature space, which could for example be a single organism, and those for multiple feature spaces. Critically, data from multiple feature spaces can be represented as a data array for each feature space. In `load_store` we represent a directory of datasets as a instance of a multi-feature space store and discover all feature spaces present. This store can be subsetted to a single store if only data corresponding to a single organism is desired, for example. The core API exposed to users is:

data.load_store(cache_path[, store_format, ...])	Instantiates a distributed store class.

sfaira.data.load_store

sfaira.data.**load_store**(*cache_path: Union[str, os.PathLike, List[str], List[os.PathLike]]*, *store_format: str = 'dao'*, *columns: Optional[List[str]] = None*) → *sfaira.data.store.stores.multi.StoreMultipleFeatureSpaceBase*

Instantiates a distributed store class.

Note that any store is instantiated as a DistributedStoreMultipleFeatureSpaceBase. This instances can be subsetted to the desired single feature space.

Parameters

- **cache_path** – Store directory.
- **store_format** – Format of store {"h5ad", "dao"}.

- ”h5ad”: Returns instance of DistributedStoreH5ad and keeps data in memory. See also “h5ad_backed”.
- ”dao”: Returns instance of DistributedStoreDoa (distributed access optimized).
- **”h5ad_backed”: Returns instance of DistributedStoreH5ad and keeps data as backed (out of memor** ”h5ad”.

• **columns** – Which columns to read into the obs copy in the output, see pandas.read_parquet(). Only relevant if store_format is “dao”.

Returns Instances of a distributed store class.

Store classes for a single feature space:

data.StoreSingleFeatureSpace(adata_by_key, ...)	Data set group class tailored to data access requirements common in high-performance computing (HPC).

sfaira.data.StoreSingleFeatureSpace

class sfaira.data.**StoreSingleFeatureSpace**(*adata_by_key: Dict[str, anndata._core.anndata.AnnData]*, *indices: Dict[str, numpy.ndarray]*, *obs_by_key: Union[None, Dict[str, dask.dataframe.core.DataFrame]] = None*, *data_source: str = 'X'*)

Data set group class tailored to data access requirements common in high-performance computing (HPC).

This class does not inherit from DatasetGroup because it entirely relies on the cached objects. This class is centred around .adata_by_key and .indices.

.adata_by_key is a dictionary (by id) of backed anndata instances that point to individual h5ads. This dictionary is intialised with all h5ads in the store. As the store is sub-setted, key-value pairs are deleted from this dictionary.

.indices have keys that correspond to keys in .adata_by_key and contain index vectors of observations in the anndata instances in .adata_by_key which are still kept. These index vectors are a form of lazy slicing that does not require data set loading or re-writing. As the store is sub-setted, key-value pairs are deleted from this dictionary if no observations from a given key match the sub-setting. If a subset of observations from a key matches the subsetting operation, the index set in the corresponding value is reduced. All data retrieval operations work on .indices: Generators run over these indices when retrieving observations for example.

Attributes

adata_by_key	Anndata instance for each selected data set in store, sub-setted by selected cells.
adata_memory_footprint	Memory foot-print of data set k in MB.
data_by_key	Data matrix for each selected data set in store, sub-setted by selected cells.
dataset_weights	
genome_container	
idx	Global indices.
indices	Indices of observations that are currently exposed in adata of this instance.
n_obs	Number of observations selected in store.

continues on next page

Table 18 – continued from previous page

n_vars	Number of selected features per organism in store
obs_by_key	
organism	Organism of store.
organisms_by_key	Data set-wise organism label as dictionary of data set keys.
shape	
var	
var_names	Feature names of selected genes by organism in store.
data_source	

sfaira.data.StoreSingleFeatureSpace.adata_by_key

property StoreSingleFeatureSpace.**adata_by_key**: Dict[str, anndata._core.anndata.AnnData]

Anndata instance for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoreSingleFeatureSpace.adata_memory_footprint

property StoreSingleFeatureSpace.**adata_memory_footprint**: Dict[str, float]

Memory foot-print of data set k in MB.

sfaira.data.StoreSingleFeatureSpace.data_by_key

property StoreSingleFeatureSpace.**data_by_key**

Data matrix for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoreSingleFeatureSpace.dataset_weights

property StoreSingleFeatureSpace.**dataset_weights**

sfaira.data.StoreSingleFeatureSpace.genome_container

property StoreSingleFeatureSpace.**genome_container**: Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]

sfaira.data.StoreSingleFeatureSpace.idx

property StoreSingleFeatureSpace.**idx**: numpy.ndarray
Global indices.

sfaira.data.StoreSingleFeatureSpace.indices

property StoreSingleFeatureSpace.**indices**: Dict[str, numpy.ndarray]
Indices of observations that are currently exposed in adata of this instance.

This depends on previous subsetting.

sfaira.data.StoreSingleFeatureSpace.n_obs

property StoreSingleFeatureSpace.**n_obs**: int
Number of observations selected in store.

sfaira.data.StoreSingleFeatureSpace.n_vars

property StoreSingleFeatureSpace.**n_vars**: int
Number of selected features per organism in store

sfaira.data.StoreSingleFeatureSpace.obs_by_key

property StoreSingleFeatureSpace.**obs_by_key**: Dict[str,
Union[pandas.core.frame.DataFrame, dask.dataframe.core.DataFrame]]

sfaira.data.StoreSingleFeatureSpace.organism

property StoreSingleFeatureSpace.**organism**
Organism of store.

sfaira.data.StoreSingleFeatureSpace.organisms_by_key

property StoreSingleFeatureSpace.**organisms_by_key**: Dict[str, str]
Data set-wise organism label as dictionary of data set keys.

sfaira.data.StoreSingleFeatureSpace.shape

property StoreSingleFeatureSpace.**shape**: Tuple[int, int]

sfaira.data.StoreSingleFeatureSpace.var

property StoreSingleFeatureSpace.**var**: pandas.core.frame.DataFrame

sfaira.data.StoreSingleFeatureSpace.var_names

property StoreSingleFeatureSpace.**var_names**: List[str]

Feature names of selected genes by organism in store.

sfaira.data.StoreSingleFeatureSpace.data_source

StoreSingleFeatureSpace.**data_source**: str

Methods

checkout([idx, batch_size, ...])	Yields an instance of a generator class over observations in the contained data sets.
get_subset_idx(attr_key, values, excluded_values)	Get indices of subset list of adata objects based on cell-wise properties.
load_config(fn)	Load a config file and recreates a data sub-setting.
subset(attr_key[, values, excluded_values, ...])	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.

sfaira.data.StoreSingleFeatureSpace.checkout

StoreSingleFeatureSpace.**checkout**(*idx: Optional[numpy.ndarray] = None*, *batch_size: int = 1*, *retrieval_batch_size: int = 128*, *map_fn=None*, *obs_keys: Optional[List[str]] = None*, *return_dense: bool = True*, *randomized_batch_access: bool = False*, *random_access: bool = False*, *batch_schedule: str = 'base'*, ***kwargs*) → *sfaira.data.store.carts.single.CartSingle*

Yields an instance of a generator class over observations in the contained data sets.

Multiple such instances can be emitted by a single store class and point to data stored in this store class. Effectively, these generators are heavily reduced pointers to the data in an instance of self. A common use case is the instantiation of a training data generator and a validation data generator over a data subset defined in this class.

Parameters

- **idx** – Global idx to query from store. These is an array with indices corresponding to a contiuous index along all observations in self.adata_by_key, ordered along a hypothetical concatenation along the keys of self.adata_by_key. If None, all observations are selected.
- **batch_size** – Number of observations to yield in each access (generator invocation).

- **retrieval_batch_size** – Number of observations read from disk in each batched access (data-backend generator invocation).
- **map_fn** – Map functino to apply to output tuple of raw generator. Each draw i from the generator is then: `yield map_fn(x[i, var_idx], obs[i, obs_keys])`
- **obs_keys** – .obs columns to return in the generator. These have to be a subset of the columns available in self.adata_by_key.
- **return_dense** – Whether to force return count data .X as dense batches. This allows more efficient feature indexing if the store is sparse (column indexing on csr matrices is slow).
- **randomized_batch_access** – Whether to randomize batches during reading (in generator). Lifts necessity of using a shuffle buffer on generator, however, batch composition stays unchanged over epochs unless there is overhangs in retrieval_batch_size in the raw data files, which often happens and results in modest changes in batch composition. Do not use randomized_batch_access and random_access.
- **random_access** – Whether to fully shuffle observations before batched access takes place. May slow down access compared randomized_batch_access and to no randomization. Do not use randomized_batch_access and random_access.
- **batch_schedule** – A valid batch schedule name or a class that inherits from BatchDesignBase.
  - "basic": sfaira.data.store.batch_schedule.BatchDesignBasic
  - "balanced": sfaira.data.store.batch_schedule.BatchDesignBalanced
  - "blocks": sfaira.data.store.batch_schedule.BatchDesignBlocks
  - "full": sfaira.data.store.batch_schedule.BatchDesignFull
  - class: batch_schedule needs to be a class (not instance), subclassing BatchDesignBase.
- **kwargs** – kwargs for idx_generator chosen.

Returns Generator function which yields batch_size at every invocation. The generator returns a tuple of (.X, .obs).

sfaira.data.StoreSingleFeatureSpace.get_subset_idx

StoreSingleFeatureSpace.**get_subset_idx**(*attr_key*, *values: Optional[Union[str, List[str]]]*, *excluded_values: Optional[Union[str, List[str]]]*) → dict

Get indices of subset list of adata objects based on cell-wise properties.

Parameters

- **attr_key** – Property to subset by. Options:
  - "assay_differentiation" points to self.assay_differentiation_obs_key
  - "assay_sc" points to self.assay_sc_obs_key
  - "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
  - "cell_line" points to self.cell_line
  - "cell_type" points to self.cell_type_obs_key

- ”developmental_stage” points to self.developmental_stage_obs_key
- ”ethnicity” points to self.ethnicity_obs_key
- ”organ” points to self.organ_obs_key
- ”organism” points to self.organism_obs_key
- ”sample_source” points to self.sample_source_obs_key
- ”sex” points to self.sex_obs_key
- ”state_exact” points to self.state_exact_obs_key

- **values** – Classes to overlap to. Supply either values or excluded_values.
- **excluded_values** – Classes to exclude from match list. Supply either values or excluded_values.

:return dictionary of files and observation indices by file.

sfaira.data.StoreSingleFeatureSpace.load_config

StoreSingleFeatureSpace.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoreSingleFeatureSpace.subset

StoreSingleFeatureSpace.**subset**(*attr_key*, *values: Union[None, str, List[str]] = None*, *excluded_values: Union[None, str, List[str]] = None*, *verbose: int = 1*)

Subset list of adata objects based on cell-wise properties.

Subsetting is done based on index vectors, the objects remain untouched.

Parameters

- **attr_key** – Property to subset by. Options:
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key
  - ”sample_source” points to self.sample_source_obs_key
  - ”sex” points to self.sex_obs_key
  - ”state_exact” points to self.state_exact_obs_key

- **values** – Classes to overlap to. Supply either values or excluded_values.
- **excluded_values** – Classes to exclude from match list. Supply either values or excluded_values.
- **verbose** – If >1 print warning message if store is empty after subsetting

sfaira.data.StoreSingleFeatureSpace.write_config

StoreSingleFeatureSpace.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

Store classes for a multiple feature spaces:

data.StoreMultipleFeatureSpaceBase(stores)	Umbrella class for a dictionary over multiple instances DistributedStoreSingleFeatureSpace.
data.StoresAnndata(adatas)	
data.StoresDao(cache_path[, columns])	
data.StoresH5ad(cache_path[, in_memory])	

sfaira.data.StoreMultipleFeatureSpaceBase

class sfaira.data.**StoreMultipleFeatureSpaceBase**(*stores: Dict[str, sfaira.data.store.stores.single.StoreSingleFeatureSpace]*)

Umbrella class for a dictionary over multiple instances DistributedStoreSingleFeatureSpace.

Allows for operations on data sets that are defined in different feature spaces.

Attributes

adata_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
adata_memory_footprint	Memory foot-print of data set k in MB.
data_by_key	Data matrix for each selected data set in store, sub-setted by selected cells.
genome_container	
indices	Dictionary of indices of selected observations contained in all stores.
n_obs	Dictionary of number of observations across stores.
n_obs_dict	Dictionary of number of observations by store.
n_vars	Dictionary of number of features by store.

continues on next page

Table 21 – continued from previous page

obs_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
shape	Dictionary of full selected data matrix shape by store.
stores	Only expose stores that contain observations.
var_names	Dictionary of variable names by store.

sfaira.data.StoreMultipleFeatureSpaceBase.adata_by_key

property StoreMultipleFeatureSpaceBase.**adata_by_key**: Dict[str, anndata._core.anndata.AnnData]

Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoreMultipleFeatureSpaceBase.adata_memory_footprint

property StoreMultipleFeatureSpaceBase.**adata_memory_footprint**: Dict[str, float]

Memory foot-print of data set k in MB.

sfaira.data.StoreMultipleFeatureSpaceBase.data_by_key

property StoreMultipleFeatureSpaceBase.**data_by_key**

Data matrix for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoreMultipleFeatureSpaceBase.genome_container

property StoreMultipleFeatureSpaceBase.**genome_container**: Dict[str, Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]]

sfaira.data.StoreMultipleFeatureSpaceBase.indices

property StoreMultipleFeatureSpaceBase.**indices**: Dict[str, numpy.ndarray]

Dictionary of indices of selected observations contained in all stores.

sfaira.data.StoreMultipleFeatureSpaceBase.n_obs

property StoreMultipleFeatureSpaceBase.**n_obs**: int

Dictionary of number of observations across stores.

sfaira.data.StoreMultipleFeatureSpaceBase.n_obs_dict

property StoreMultipleFeatureSpaceBase.**n_obs_dict**: Dict[str, int]
Dictionary of number of observations by store.

sfaira.data.StoreMultipleFeatureSpaceBase.n_vars

property StoreMultipleFeatureSpaceBase.**n_vars**: Dict[str, int]
Dictionary of number of features by store.

sfaira.data.StoreMultipleFeatureSpaceBase.obs_by_key

property StoreMultipleFeatureSpaceBase.**obs_by_key**: Dict[str, Union[pandas.core.frame.DataFrame, dask.dataframe.core.DataFrame]]
Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoreMultipleFeatureSpaceBase.shape

property StoreMultipleFeatureSpaceBase.**shape**: Dict[str, Tuple[int, int]]
Dictionary of full selected data matrix shape by store.

sfaira.data.StoreMultipleFeatureSpaceBase.stores

property StoreMultipleFeatureSpaceBase.**stores**: Dict[str, *sfaira.data.store.stores.single.StoreSingleFeatureSpace*]
Only expose stores that contain observations.

sfaira.data.StoreMultipleFeatureSpaceBase.var_names

property StoreMultipleFeatureSpaceBase.**var_names**: Dict[str, List[str]]
Dictionary of variable names by store.

Methods

checkout([idx, intercalated])	Carts per store.
load_config(fn)	Load a config file and recreates a data sub-setting.
subset(attr_key[, values, excluded_values, ...])	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.

sfaira.data.StoreMultipleFeatureSpaceBase.checkout

StoreMultipleFeatureSpaceBase.**checkout**(*idx: Optional[Dict[str, Optional[numpy.ndarray]]] = None*, *intercalated: bool = True*, ***kwargs*) → *sfaira.data.store.carts.multi.CartMulti*

Carts per store.

See also DistributedStore*.checkout().

Parameters

- **idx** –
- **intercalated** – Whether to do sequential or intercalated emission.
- **kwargs** – See parameters of DistributedStore*.generator().

Returns Generator function which yields batch_size at every invocation. The generator returns a tuple of (.X, .obs).

sfaira.data.StoreMultipleFeatureSpaceBase.load_config

StoreMultipleFeatureSpaceBase.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoreMultipleFeatureSpaceBase.subset

StoreMultipleFeatureSpaceBase.**subset**(*attr_key*, *values: Union[None, str, List[str]] = None*, *excluded_values: Union[None, str, List[str]] = None*, *verbose: int = 1*)

Subset list of adata objects based on cell-wise properties.

Subsetting is done based on index vectors, the objects remain untouched.

Parameters

- **attr_key** – Property to subset by. Options:
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key
  - ”sample_source” points to self.sample_source_obs_key
  - ”sex” points to self.sex_obs_key

  - "state_exact" points to self.state_exact_obs_key
- **values** – Classes to overlap to. Supply either values or excluded_values.
- **excluded_values** – Classes to exclude from match list. Supply either values or excluded_values.

sfaira.data.StoreMultipleFeatureSpaceBase.write_config

StoreMultipleFeatureSpaceBase.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresAnndata

class sfaira.data.**StoresAnndata**(*adatas: Union[anndata._core.anndata.AnnData, List[anndata._core.anndata.AnnData], Tuple[anndata._core.anndata.AnnData]]*)

Attributes

adata_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
adata_memory_footprint	Memory foot-print of data set k in MB.
data_by_key	Data matrix for each selected data set in store, sub-setted by selected cells.
genome_container	
indices	Dictionary of indices of selected observations contained in all stores.
n_obs	Dictionary of number of observations across stores.
n_obs_dict	Dictionary of number of observations by store.
n_vars	Dictionary of number of features by store.
obs_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
shape	Dictionary of full selected data matrix shape by store.
stores	Only expose stores that contain observations.
var_names	Dictionary of variable names by store.

sfaira.data.StoresAnndata.adata_by_key

property StoresAnndata.**adata_by_key**: Dict[str, anndata._core.anndata.AnnData]
Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresAnndata.adata_memory_footprint

property StoresAnndata.**adata_memory_footprint**: Dict[str, float]
Memory foot-print of data set k in MB.

sfaira.data.StoresAnndata.data_by_key

property StoresAnndata.**data_by_key**
Data matrix for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoresAnndata.genome_container

property StoresAnndata.**genome_container**: Dict[str, Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]]

sfaira.data.StoresAnndata.indices

property StoresAnndata.**indices**: Dict[str, numpy.ndarray]
Dictionary of indices of selected observations contained in all stores.

sfaira.data.StoresAnndata.n_obs

property StoresAnndata.**n_obs**: int
Dictionary of number of observations across stores.

sfaira.data.StoresAnndata.n_obs_dict

property StoresAnndata.**n_obs_dict**: Dict[str, int]
Dictionary of number of observations by store.

sfaira.data.StoresAnndata.n_vars

property StoresAnndata.**n_vars**: Dict[str, int]
Dictionary of number of features by store.

sfaira.data.StoresAnndata.obs_by_key

property StoresAnndata.**obs_by_key**: Dict[str, Union[pandas.core.frame.DataFrame, dask.dataframe.core.DataFrame]]

Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresAnndata.shape

property StoresAnndata.**shape**: Dict[str, Tuple[int, int]]

Dictionary of full selected data matrix shape by store.

sfaira.data.StoresAnndata.stores

property StoresAnndata.**stores**: Dict[str, *sfaira.data.store.stores.single.StoreSingleFeatureSpace*]

Only expose stores that contain observations.

sfaira.data.StoresAnndata.var_names

property StoresAnndata.**var_names**: Dict[str, List[str]]

Dictionary of variable names by store.

Methods

checkout([idx, intercalated])	Carts per store.
load_config(fn)	Load a config file and recreates a data sub-setting.
subset(attr_key[, values, excluded_values, ...])	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.

sfaira.data.StoresAnndata.checkout

StoresAnndata.**checkout**(*idx: Optional[Dict[str, Optional[numpy.ndarray]]] = None*, *intercalated: bool = True*, ***kwargs*) → *sfaira.data.store.carts.multi.CartMulti*

Carts per store.

See also DistributedStore*.checkout().

Parameters

- **idx** –
- **intercalated** – Whether to do sequential or intercalated emission.
- **kwargs** – See parameters of DistributedStore*.generator().

Returns Generator function which yields batch_size at every invocation. The generator returns a tuple of (.X, .obs).

sfaira.data.StoresAnndata.load_config

StoresAnndata.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresAnndata.subset

StoresAnndata.**subset**(*attr_key*, *values: Union[None, str, List[str]] = None*, *excluded_values: Union[None, str, List[str]] = None*, *verbose: int = 1*)

Subset list of adata objects based on cell-wise properties.

Subsetting is done based on index vectors, the objects remain untouched.

Parameters

- **attr_key** – Property to subset by. Options:
  - ”assay_differentiation” points to self.assay_differentiation_obs_key
  - ”assay_sc” points to self.assay_sc_obs_key
  - ”assay_type_differentiation” points to self.assay_type_differentiation_obs_key
  - ”cell_line” points to self.cell_line
  - ”cell_type” points to self.cell_type_obs_key
  - ”developmental_stage” points to self.developmental_stage_obs_key
  - ”ethnicity” points to self.ethnicity_obs_key
  - ”organ” points to self.organ_obs_key
  - ”organism” points to self.organism_obs_key
  - ”sample_source” points to self.sample_source_obs_key
  - ”sex” points to self.sex_obs_key
  - ”state_exact” points to self.state_exact_obs_key
- **values** – Classes to overlap to. Supply either values or excluded_values.
- **excluded_values** – Classes to exclude from match list. Supply either values or excluded_values.

sfaira.data.StoresAnndata.write_config

StoresAnndata.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresDao

class sfaira.data.**StoresDao**(*cache_path: Union[str, os.PathLike, List[str], List[os.PathLike]]*, *columns: Optional[List[str]] = None*)

Attributes

adata_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
adata_memory_footprint	Memory foot-print of data set k in MB.
data_by_key	Data matrix for each selected data set in store, sub-setted by selected cells.
genome_container	
indices	Dictionary of indices of selected observations contained in all stores.
n_obs	Dictionary of number of observations across stores.
n_obs_dict	Dictionary of number of observations by store.
n_vars	Dictionary of number of features by store.
obs_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
shape	Dictionary of full selected data matrix shape by store.
stores	Only expose stores that contain observations.
var_names	Dictionary of variable names by store.

sfaira.data.StoresDao.adata_by_key

property StoresDao.**adata_by_key**: Dict[str, anndata._core.anndata.AnnData]

Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresDao.adata_memory_footprint

property StoresDao.**adata_memory_footprint**: Dict[str, float]

Memory foot-print of data set k in MB.

sfaira.data.StoresDao.data_by_key

property StoresDao.**data_by_key**

Data matrix for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoresDao.genome_container

property StoresDao.**genome_container**: Dict[str, Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]]

sfaira.data.StoresDao.indices

property StoresDao.**indices**: Dict[str, numpy.ndarray]

Dictionary of indices of selected observations contained in all stores.

sfaira.data.StoresDao.n_obs

property StoresDao.**n_obs**: int

Dictionary of number of observations across stores.

sfaira.data.StoresDao.n_obs_dict

property StoresDao.**n_obs_dict**: Dict[str, int]

Dictionary of number of observations by store.

sfaira.data.StoresDao.n_vars

property StoresDao.**n_vars**: Dict[str, int]

Dictionary of number of features by store.

sfaira.data.StoresDao.obs_by_key

property StoresDao.**obs_by_key**: Dict[str, Union[pandas.core.frame.DataFrame, dask.dataframe.core.DataFrame]]

Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresDao.shape

property StoresDao.**shape**: Dict[str, Tuple[int, int]]

Dictionary of full selected data matrix shape by store.

sfaira.data.StoresDao.stores

property StoresDao.**stores**: Dict[str, *sfaira.data.store.stores.single.StoreSingleFeatureSpace*]

Only expose stores that contain observations.

sfaira.data.StoresDao.var_names

property StoresDao.**var_names**: Dict[str, List[str]]
Dictionary of variable names by store.

Methods

checkout([idx, intercalated])	Carts per store.
load_config(fn)	Load a config file and recreates a data sub-setting.
subset(attr_key[, values, excluded_values, ...])	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.

sfaira.data.StoresDao.checkout

StoresDao.**checkout**(*idx: Optional[Dict[str, Optional[numpy.ndarray]]] = None*, *intercalated: bool = True*, ***kwargs*) → *sfaira.data.store.carts.multi.CartMulti*
Carts per store.

See also DistributedStore*.checkout().

Parameters

- **idx** –
- **intercalated** – Whether to do sequential or intercalated emission.
- **kwargs** – See parameters of DistributedStore*.generator().

Returns Generator function which yields batch_size at every invocation. The generator returns a tuple of (.X, .obs).

sfaira.data.StoresDao.load_config

StoresDao.**load_config**(*fn: Union[str, os.PathLike]*)
Load a config file and recreates a data sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresDao.subset

StoresDao.**subset**(*attr_key*, *values: Union[None, str, List[str]] = None*, *excluded_values: Union[None, str, List[str]] = None*, *verbose: int = 1*)
Subset list of adata objects based on cell-wise properties.

Subsetting is done based on index vectors, the objects remain untouched.

Parameters

- **attr_key** – Property to subset by. Options:
  - "assay_differentiation" points to self.assay_differentiation_obs_key

- "assay_sc" points to self.assay_sc_obs_key
- "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
- "cell_line" points to self.cell_line
- "cell_type" points to self.cell_type_obs_key
- "developmental_stage" points to self.developmental_stage_obs_key
- "ethnicity" points to self.ethnicity_obs_key
- "organ" points to self.organ_obs_key
- "organism" points to self.organism_obs_key
- "sample_source" points to self.sample_source_obs_key
- "sex" points to self.sex_obs_key
- "state_exact" points to self.state_exact_obs_key

values – Classes to overlap to. Supply either values or excluded_values.

excluded_values – Classes to exclude from match list. Supply either values or excluded_values.

sfaira.data.StoresDao.write_config

StoresDao.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresH5ad

class sfaira.data.**StoresH5ad**(*cache_path: Union[str, os.PathLike, List[str], List[os.PathLike]]*, *in_memory: bool = False*)

Attributes

adata_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
adata_memory_footprint	Memory foot-print of data set k in MB.
data_by_key	Data matrix for each selected data set in store, sub-setted by selected cells.
genome_container	
indices	Dictionary of indices of selected observations contained in all stores.
n_obs	Dictionary of number of observations across stores.
n_obs_dict	Dictionary of number of observations by store.

continues on next page

Table 27 – continued from previous page

n_vars	Dictionary of number of features by store.
obs_by_key	Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.
shape	Dictionary of full selected data matrix shape by store.
stores	Only expose stores that contain observations.
var_names	Dictionary of variable names by store.

sfaira.data.StoresH5ad.adata_by_key

property StoresH5ad.**adata_by_key**: Dict[str, anndata._core.anndata.AnnData]
Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresH5ad.adata_memory_footprint

property StoresH5ad.**adata_memory_footprint**: Dict[str, float]
Memory foot-print of data set k in MB.

sfaira.data.StoresH5ad.data_by_key

property StoresH5ad.**data_by_key**
Data matrix for each selected data set in store, sub-setted by selected cells.

sfaira.data.StoresH5ad.genome_container

property StoresH5ad.**genome_container**: Dict[str, Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]]

sfaira.data.StoresH5ad.indices

property StoresH5ad.**indices**: Dict[str, numpy.ndarray]
Dictionary of indices of selected observations contained in all stores.

sfaira.data.StoresH5ad.n_obs

property StoresH5ad.**n_obs**: int
Dictionary of number of observations across stores.

sfaira.data.StoresH5ad.n_obs_dict

property StoresH5ad.**n_obs_dict**: Dict[str, int]
Dictionary of number of observations by store.

sfaira.data.StoresH5ad.n_vars

property StoresH5ad.**n_vars**: Dict[str, int]
Dictionary of number of features by store.

sfaira.data.StoresH5ad.obs_by_key

property StoresH5ad.**obs_by_key**: Dict[str, Union[pandas.core.frame.DataFrame, dask.dataframe.core.DataFrame]]
Dictionary of all anndata instances for each selected data set in store, sub-setted by selected cells, for each stores.

sfaira.data.StoresH5ad.shape

property StoresH5ad.**shape**: Dict[str, Tuple[int, int]]
Dictionary of full selected data matrix shape by store.

sfaira.data.StoresH5ad.stores

property StoresH5ad.**stores**: Dict[str, *sfaira.data.store.stores.single.StoreSingleFeatureSpace*]
Only expose stores that contain observations.

sfaira.data.StoresH5ad.var_names

property StoresH5ad.**var_names**: Dict[str, List[str]]
Dictionary of variable names by store.

Methods

checkout([idx, intercalated])	Carts per store.
load_config(fn)	Load a config file and recreates a data sub-setting.
subset(attr_key[, values, excluded_values, ...])	Subset list of adata objects based on cell-wise properties.
write_config(fn)	Writes a config file that describes the current data sub-setting.

sfaira.data.StoresH5ad.checkout

StoresH5ad.**checkout**(*idx: Optional[Dict[str, Optional[numpy.ndarray]]] = None*, *intercalated: bool = True*, ***kwargs*) → *sfaira.data.store.carts.multi.CartMulti*

Carts per store.

See also DistributedStore*.checkout().

Parameters

- **idx** –
- **intercalated** – Whether to do sequential or intercalated emission.
- **kwargs** – See parameters of DistributedStore*.generator().

Returns Generator function which yields batch_size at every invocation. The generator returns a tuple of (.X, .obs).

sfaira.data.StoresH5ad.load_config

StoresH5ad.**load_config**(*fn: Union[str, os.PathLike]*)

Load a config file and recreates a data sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

sfaira.data.StoresH5ad.subset

StoresH5ad.**subset**(*attr_key*, *values: Union[None, str, List[str]] = None*, *excluded_values: Union[None, str, List[str]] = None*, *verbose: int = 1*)

Subset list of adata objects based on cell-wise properties.

Subsetting is done based on index vectors, the objects remain untouched.

Parameters

- **attr_key** – Property to subset by. Options:
  - "assay_differentiation" points to self.assay_differentiation_obs_key
  - "assay_sc" points to self.assay_sc_obs_key
  - "assay_type_differentiation" points to self.assay_type_differentiation_obs_key
  - "cell_line" points to self.cell_line
  - "cell_type" points to self.cell_type_obs_key
  - "developmental_stage" points to self.developmental_stage_obs_key
  - "ethnicity" points to self.ethnicity_obs_key
  - "organ" points to self.organ_obs_key
  - "organism" points to self.organism_obs_key
  - "sample_source" points to self.sample_source_obs_key
  - "sex" points to self.sex_obs_key
  - "state_exact" points to self.state_exact_obs_key

- **values** – Classes to overlap to. Supply either values or excluded_values.
- **excluded_values** – Classes to exclude from match list. Supply either values or excluded_values.

sfaira.data.StoresH5ad.write_config

StoresH5ad.**write_config**(*fn: Union[str, os.PathLike]*)

Writes a config file that describes the current data sub-setting.

This config file can be loaded later to recreate a sub-setting. This config file contains observation-wise subsetting information.

Parameters **fn** – Output file without file type extension.

Carts

Stores represent on-disk data collection and perform operations such as subsetting. Ultimatively, they are often used to emit data objects, which are "carts". Carts are specific to the underlying store's data format and expose iterators, data matrices and adaptors to machine learning framework data pipelines, such as tensorflow and torchc data. Again, carts can cover one or multiple feature spaces.

data.store.carts.CartSingle(obs_idx, ...[, ...])	Cart for a DistributedStoreSingleFeatureSpace().
data.store.carts.CartMulti(carts[, intercalated])	Cart for a DistributedStoreMultipleFeatureSpaceBase().

sfaira.data.store.carts.CartSingle

class sfaira.data.store.carts.**CartSingle**(*obs_idx*, *obs_keys*, *var*, *var_idx=None*, *batch_schedule='base'*, *batch_size=1*, *map_fn=None*, ***kwargs*)

Cart for a DistributedStoreSingleFeatureSpace().

Attributes

adata	Assembles a slice of this cart based on .obs_idx as an anndata instance.
n_batches	
n_obs	Total number of observations in cart.
n_obs_selected	Total number of selected observations in cart.
n_var	Total number of features defined for return in cart.
obs	Selected meta data matrix (cells x meta data) that is emitted in batches by .iterator().
obs_idx	Integer observation indices to select from cart.
x	Selected data matrix (cells x features) that is emitted in batches by .iterator().
batch_size	
obs_keys	

continues on next page

Table 30 – continued from previous page

schedule
var
var_idx

sfaira.data.store.carts.CartSingle.adata

property CartSingle.**adata**: anndata._core.anndata.AnnData
Assembles a slice of this cart based on .obs_idx as an anndata instance.

sfaira.data.store.carts.CartSingle.n_batches

property CartSingle.**n_batches**

sfaira.data.store.carts.CartSingle.n_obs

property CartSingle.**n_obs**: int
Total number of observations in cart.

sfaira.data.store.carts.CartSingle.n_obs_selected

property CartSingle.**n_obs_selected**: int
Total number of selected observations in cart.

sfaira.data.store.carts.CartSingle.n_var

property CartSingle.**n_var**
Total number of features defined for return in cart.

sfaira.data.store.carts.CartSingle.obs

property CartSingle.**obs**
Selected meta data matrix (cells x meta data) that is emitted in batches by .iterator().

sfaira.data.store.carts.CartSingle.obs_idx

property CartSingle.**obs_idx**
Integer observation indices to select from cart. These will be emitted if data is queried.

sfaira.data.store.carts.CartSingle.x

property CartSingle.**x**
Selected data matrix (cells x features) that is emitted in batches by .iterator().

sfaira.data.store.carts.CartSingle.batch_size

CartSingle.**batch_size**: int

sfaira.data.store.carts.CartSingle.obs_keys

CartSingle.**obs_keys**: List[str]

sfaira.data.store.carts.CartSingle.schedule

CartSingle.**schedule**: *sfaira.data.store.batch_schedule.BatchDesignBase*

sfaira.data.store.carts.CartSingle.var

CartSingle.**var**: pandas.core.frame.DataFrame

sfaira.data.store.carts.CartSingle.var_idx

CartSingle.**var_idx**: Union[None, numpy.ndarray]

Methods

adaptor(generator_type[, dataset_kwargs, ...])	The adaptor turns a python base generator into a different iteratable object, defined by generator_type.
iterator([repeat, shuffle_buffer])	Iterator over data matrix and meta data table, yields batches of data points.
move_to_memory()	Load underlying array into memory into memory.

sfaira.data.store.carts.CartSingle.adaptor

CartSingle.**adaptor**(*generator_type: str*, *dataset_kwargs: Optional[dict] = None*, *shuffle_buffer: int = 0*, *repeat: int = 1*, ***kwargs*)
The adaptor turns a python base generator into a different iteratable object, defined by generator_type.

Parameters

- **generator_type** – Type of output iteratable. - python base generator (no change to `.generator`) - tensorflow dataset: This dataset is defined on a python iterator.

  Important: This only returns the tf.data.Dataset.from_generator(). You need to define the input pipeline (e.g. .batch(), .prefetch()) on top of this data set.

  - **pytorch: We distinguish torch.data.Dataset and torch.data.DataLoader ontop of either.** The Dataset vs DataLoader distinction is made by the "" suffix for Dataset or "-loader" suffix for + dataloader. The distinction between Dataset and IteratableDataset defines if the object is defined directly on a dask array or based on a python iterator on a dask array. Note that the python iterator can implement favorable remote access schemata but the torch.data.Dataset generally causes less trouble in out-of-the-box usage.
    * torch.data.Dataset: "torch" prefix, ie "torch" or "torch-loader"
    * torch.data.IteratableDataset: "torch-iter" prefix, ie "torch-iter" or "torch-iter-loader"

    Important: For model training in pytorch you need the "-loader" prefix. You can specify the arguments passed to torch.utils.data.DataLoader by the dataset_kwargs dictionary.

- **dataset_kwargs** – Dict Parameters to pass to the constructor of torch Dataset. Only relevant if generator_type in ['torch', 'torch-loader']
- **shuffle_buffer** – int If shuffle_buffer > 0 -> Use a shuffle buffer with size shuffle_buffer to shuffle output of self.iterator (this option is useful when using randomized_batch_access in the DaskCart)
- **repeat** – int Number of times to repeat the dataset until the underlying generator runs out of samples. If repeat <= 0 -> repeat dataset forever

Returns Modified iteratable (see generator_type).

sfaira.data.store.carts.CartSingle.iterator

CartSingle.**iterator**(*repeat: int = 1*, *shuffle_buffer: int = 0*)

Iterator over data matrix and meta data table, yields batches of data points.

sfaira.data.store.carts.CartSingle.move_to_memory

CartSingle.**move_to_memory**()

Load underlying array into memory into memory.

sfaira.data.store.carts.CartMulti

class sfaira.data.store.carts.**CartMulti**(*carts: Dict[str,* sfaira.data.store.carts.single.CartSingle*]*, *intercalated: bool = False*)

Cart for a DistributedStoreMultipleFeatureSpaceBase().

Attributes

adata	Assembles a dictionary of slices of this cart based on .obs_idx as anndata instances per organism.
n_batches	
n_obs	Total number of observations in cart.
n_obs_selected	Total number of selected observations in cart.
n_var	Total number of features defined for return in cart.
obs	Selected meta data matrix (cells x meta data) that is emitted in batches by .iterator().
obs_idx	Dictionary of integer observation indices to select from cart.
ratios	Define relative drawing frequencies from iterators for intercalation.
var	
x	Selected data matrix (cells x features) that is emitted in batches by .iterator().
carts	
intercalated	

sfaira.data.store.carts.CartMulti.adata

property CartMulti.**adata**: Dict[str, anndata._core.anndata.AnnData]
Assembles a dictionary of slices of this cart based on .obs_idx as anndata instances per organism.

sfaira.data.store.carts.CartMulti.n_batches

property CartMulti.**n_batches**: int

sfaira.data.store.carts.CartMulti.n_obs

property CartMulti.**n_obs**: int
Total number of observations in cart.

sfaira.data.store.carts.CartMulti.n_obs_selected

property CartMulti.**n_obs_selected**: int
Total number of selected observations in cart.

sfaira.data.store.carts.CartMulti.n_var

property CartMulti.**n_var**: Dict[str, int]
Total number of features defined for return in cart.

sfaira.data.store.carts.CartMulti.obs

property CartMulti.**obs**
Selected meta data matrix (cells x meta data) that is emitted in batches by .iterator().

sfaira.data.store.carts.CartMulti.obs_idx

property CartMulti.**obs_idx**
Dictionary of integer observation indices to select from cart. These will be emitted if data is queried.

sfaira.data.store.carts.CartMulti.ratios

property CartMulti.**ratios**
Define relative drawing frequencies from iterators for intercalation.

Note that these can be float and will be randomly rounded during intercalation.

sfaira.data.store.carts.CartMulti.var

property CartMulti.**var**

sfaira.data.store.carts.CartMulti.x

property CartMulti.**x**
Selected data matrix (cells x features) that is emitted in batches by .iterator().

sfaira.data.store.carts.CartMulti.carts

CartMulti.**carts**: Dict[str, *sfaira.data.store.carts.single.CartSingle*]

sfaira.data.store.carts.CartMulti.intercalated

CartMulti.**intercalated**: bool

Methods

adaptor(generator_type[, dataset_kwargs, ...])	The adaptor turns a python base generator into a different iteratable object, defined by generator_type.
iterator([repeat, shuffle_buffer])	Iterator over data matrix and meta data table, yields batches of data points.
move_to_memory()	Persist underlying arrays into memory in sparse.COO format.

sfaira.data.store.carts.CartMulti.adaptor

CartMulti.**adaptor**(*generator_type: str*, *dataset_kwargs: Optional[dict] = None*, *shuffle_buffer: int = 0*, *repeat: int = 1*, ***kwargs*)

The adaptor turns a python base generator into a different iteratable object, defined by generator_type.

Parameters

- **generator_type** – Type of output iteratable. - python base generator (no change to `.generator`) - tensorflow dataset: This dataset is defined on a python iterator.

  Important: This only returns the tf.data.Dataset.from_generator(). You need to define the input pipeline (e.g. .batch(), .prefetch()) on top of this data set.

  - **pytorch: We distinguish torch.data.Dataset and torch.data.DataLoader ontop of either.** The Dataset vs DataLoader distinction is made by the "" suffix for Dataset or "-loader" suffix for + dataloader. The distinction between Dataset and IteratableDataset defines if the object is defined directly on a dask array or based on a python iterator on a dask array. Note that the python iterator can implement favorable remote access schemata but the torch.data.Dataset generally causes less trouble in out-of-the-box usage.

    * torch.data.Dataset: "torch" prefix, ie "torch" or "torch-loader"
    * torch.data.IteratableDataset: "torch-iter" prefix, ie "torch-iter" or "torch-iter-loader"

    Important: For model training in pytorch you need the "-loader" prefix. You can specify the arguments passed to torch.utils.data.DataLoader by the dataset_kwargs dictionary.

- **dataset_kwargs** – Dict Parameters to pass to the constructor of torch Dataset. Only relevant if generator_type in ['torch', 'torch-loader']
- **shuffle_buffer** – int If shuffle_buffer > 0 -> Use a shuffle buffer with size shuffle_buffer to shuffle output of self.iterator (this option is useful when using randomized_batch_access in the DaskCart)
- **repeat** – int Number of times to repeat the dataset until the underlying generator runs out of samples. If repeat <= 0 -> repeat dataset forever

Returns Modified iteratable (see generator_type).

sfaira.data.store.carts.CartMulti.iterator

CartMulti.**iterator**(*repeat: int = 1*, *shuffle_buffer: int = 0*)
Iterator over data matrix and meta data table, yields batches of data points.

sfaira.data.store.carts.CartMulti.move_to_memory

CartMulti.**move_to_memory**()
Persist underlying arrays into memory in sparse.COO format.

The emission of data from cart iterators and adaptors is controlled by batch schedules, which direct how data is released from the underlying data matrix:

data.store.batch_schedule.BatchDesignBase(...)	Manages distribution of selected indices for a given data object over subsequent batches.
data.store.batch_schedule.BatchDesignBasic(...)	Standard batched access to data.
data.store.batch_schedule.BatchDesignBalanced(...)	Balanced batches across meta data partitions of data.
data.store.batch_schedule.BatchDesignBlocks(...)	Meta data-defined blocks of observations in each batch.
data.store.batch_schedule.BatchDesignFull(...)	Emits full dataset as a single batch in each query.

sfaira.data.store.batch_schedule.BatchDesignBase

class sfaira.data.store.batch_schedule.**BatchDesignBase**(*retrieval_batch_size: int*, *randomized_batch_access: bool*, *random_access: bool*, ***kwargs*)

Manages distribution of selected indices for a given data object over subsequent batches.

This involves randomisation and possible meta-data dependent batching. This class is centred on the property `.design` which yields a list of observation indices (arrays), where each array is the set of indices that map to a batch and the sequence of batches in the list encodes the sequence of batches during an epoch over the data.

Attributes

batchsplits	Batch size of the yieleded batches.
design	Yields index objects for one epoch of all data.
idx	Protects property from uncontrolled changing.
n_batches	

sfaira.data.store.batch_schedule.BatchDesignBase.batchsplits

property BatchDesignBase.**batchsplits: Tuple[int]**
Batch size of the yieleded batches.

sfaira.data.store.batch_schedule.BatchDesignBase.design

property BatchDesignBase.**design: List[numpy.array]**
Yields index objects for one epoch of all data.

These index objects are used by generators that have access to the data objects to build data batches. Randomization is performed anew with every call to this property.

Returns List[np.array] List of indices per batch

sfaira.data.store.batch_schedule.BatchDesignBase.idx

property BatchDesignBase.**idx**
Protects property from uncontrolled changing. Changes to _idx require changes to _batch_bounds.

sfaira.data.store.batch_schedule.BatchDesignBase.n_batches

property BatchDesignBase.**n_batches: int**

Methods

---

sfaira.data.store.batch_schedule.BatchDesignBasic

class sfaira.data.store.batch_schedule.**BatchDesignBasic**(*retrieval_batch_size: int*, *randomized_batch_access: bool*, *random_access: bool*, ***kwargs*)
Standard batched access to data.

Attributes

batchsplits	Batch size of the yieleded batches.
design	Yields index objects for one epoch of all data.
idx	Protects property from uncontrolled changing.
n_batches	

sfaira.data.store.batch_schedule.BatchDesignBasic.batchsplits

property BatchDesignBasic.**batchsplits**: Tuple[int]
Batch size of the yieleded batches.

sfaira.data.store.batch_schedule.BatchDesignBasic.design

property BatchDesignBasic.**design**: List[numpy.ndarray]
Yields index objects for one epoch of all data.

These index objects are used by generators that have access to the data objects to build data batches. Randomization is performed anew with every call to this property.

Returns List[np.array] List of indices per batch

sfaira.data.store.batch_schedule.BatchDesignBasic.idx

property BatchDesignBasic.**idx**
Protects property from uncontrolled changing. Changes to _idx require changes to _batch_bounds.

sfaira.data.store.batch_schedule.BatchDesignBasic.n_batches

property BatchDesignBasic.**n_batches**: int

Methods

---

sfaira.data.store.batch_schedule.BatchDesignBalanced

class sfaira.data.store.batch_schedule.**BatchDesignBalanced**(*grouping*, *group_weights: dict*, *randomized_batch_access: bool*, *random_access: bool*, ***kwargs*)
Balanced batches across meta data partitions of data.

Attributes

batchsplits	Batch size of the yieleded batches.
design	Yields index objects for one epoch of all data.
idx	Protects property from uncontrolled changing.
n_batches	

sfaira.data.store.batch_schedule.BatchDesignBalanced.batchsplits

property BatchDesignBalanced.**batchsplits**: Tuple[int]
Batch size of the yieleded batches.

sfaira.data.store.batch_schedule.BatchDesignBalanced.design

property BatchDesignBalanced.**design**: List[numpy.ndarray]
Yields index objects for one epoch of all data.

These index objects are used by generators that have access to the data objects to build data batches. Randomization is performed anew with every call to this property.

Returns List[np.array] List of indices per batch

sfaira.data.store.batch_schedule.BatchDesignBalanced.idx

property BatchDesignBalanced.**idx**
Protects property from uncontrolled changing. Changes to _idx require changes to _batch_bounds.

sfaira.data.store.batch_schedule.BatchDesignBalanced.n_batches

property BatchDesignBalanced.**n_batches**: int

Methods

—

sfaira.data.store.batch_schedule.BatchDesignBlocks

class sfaira.data.store.batch_schedule.**BatchDesignBlocks**(*grouping*, *random_access: bool*, ***kwargs*)
Meta data-defined blocks of observations in each batch.

Note that the batches do not necessarily contain equal numbers of observations! The yielded batch size depends solely on the number of observations in a meta data partition.

Attributes

batchsplits	Batch size of the yieleded batches.
design	Yields index objects for one epoch of all data.
grouping	
groups	
idx	Protects property from uncontrolled changing.

continues on next page

Table 41 – continued from previous page

n_batches

sfaira.data.store.batch_schedule.BatchDesignBlocks.batchsplits

property BatchDesignBlocks.**batchsplits**: Tuple[int]
Batch size of the yieleded batches.

sfaira.data.store.batch_schedule.BatchDesignBlocks.design

property BatchDesignBlocks.**design**: List[numpy.ndarray]
Yields index objects for one epoch of all data.

These index objects are used by generators that have access to the data objects to build data batches. Randomization is performed anew with every call to this property.

Returns List[np.array] List of indices per batch

sfaira.data.store.batch_schedule.BatchDesignBlocks.grouping

property BatchDesignBlocks.**grouping**

sfaira.data.store.batch_schedule.BatchDesignBlocks.groups

property BatchDesignBlocks.**groups**

sfaira.data.store.batch_schedule.BatchDesignBlocks.idx

property BatchDesignBlocks.**idx**
Protects property from uncontrolled changing. Changes to _idx require changes to batch splitting.

sfaira.data.store.batch_schedule.BatchDesignBlocks.n_batches

property BatchDesignBlocks.**n_batches**: int

Methods

—

sfaira.data.store.batch_schedule.BatchDesignFull

class sfaira.data.store.batch_schedule.**BatchDesignFull**(*retrieval_batch_size: int*, *randomized_batch_access: bool*, *random_access: bool*, ***kwargs*)

Emits full dataset as a single batch in each query.

Attributes

batchsplits	Batch size of the yieleded batches.
design	Yields index objects for one epoch of all data.
idx	Protects property from uncontrolled changing.
n_batches	

sfaira.data.store.batch_schedule.BatchDesignFull.batchsplits

property BatchDesignFull.**batchsplits**: Tuple[int]

Batch size of the yieleded batches.

sfaira.data.store.batch_schedule.BatchDesignFull.design

property BatchDesignFull.**design**: List[numpy.ndarray]

Yields index objects for one epoch of all data.

These index objects are used by generators that have access to the data objects to build data batches. Randomization is performed anew with every call to this property.

Returns List[np.array] List of indices per batch

sfaira.data.store.batch_schedule.BatchDesignFull.idx

property BatchDesignFull.**idx**

Protects property from uncontrolled changing. Changes to _idx require changes to _batch_bounds.

sfaira.data.store.batch_schedule.BatchDesignFull.n_batches

property BatchDesignFull.**n_batches**: int

Methods

—

For most purposes related to stochastic optimisation, `BatchDesignBasic` is chosen.

Estimator classes: `estimators`

Estimator classes from the sfaira model zoo API for advanced use.

estimators.EstimatorKeras()	Estimator base class for keras models.
estimators.EstimatorKerasCelltype(data, ...)	Estimator class for the cell type model.
estimators.EstimatorKerasEmbedding(data, ...)	Estimator class for the embedding model.

sfaira.estimators.EstimatorKeras

class sfaira.estimators.**EstimatorKeras**

Estimator base class for keras models.

Important: Subclass implementing abstract classes also has to inherit from EstimatorBase class.

Attributes

data
model
weights
model_dir
history
train_hyperparam
idx_train
idx_eval
idx_test
cache_path
model_id
md5

sfaira.estimators.EstimatorKeras.data

EstimatorKeras.**data:** *sfaira.data.store.stores.single.StoreSingleFeatureSpace*

sfaira.estimators.EstimatorKeras.model

EstimatorKeras.**model:** Optional[sfaira.models.base.BasicModelKeras]

sfaira.estimators.EstimatorKeras.weights

EstimatorKeras.**weights:** Optional[numpy.ndarray]

sfaira.estimators.EstimatorKeras.model_dir

EstimatorKeras.**model_dir:** Optional[str]

sfaira.estimators.EstimatorKeras.history

EstimatorKeras.**history:** Optional[dict]

sfaira.estimators.EstimatorKeras.train_hyperparam

EstimatorKeras.**train_hyperparam:** Optional[dict]

sfaira.estimators.EstimatorKeras.idx_train

EstimatorKeras.**idx_train:** Optional[numpy.ndarray]

sfaira.estimators.EstimatorKeras.idx_eval

EstimatorKeras.**idx_eval:** Optional[numpy.ndarray]

sfaira.estimators.EstimatorKeras.idx_test

EstimatorKeras.**idx_test:** Optional[numpy.ndarray]

sfaira.estimators.EstimatorKeras.cache_path

EstimatorKeras.**cache_path**: str

sfaira.estimators.EstimatorKeras.model_id

EstimatorKeras.**model_id**: Optional[str]

sfaira.estimators.EstimatorKeras.md5

EstimatorKeras.**md5**: Optional[str]

Methods

get_one_time_tf_dataset(idx, mode[, ...])	
init_model([clear_weight_cache, ...])	Instantiate the model.
load_pretrained_weights()	Loads model weights from local directory or zenodo.
load_weights_from_cache()	
save_weights_to_cache()	
split_train_val_test(val_split, test_split)	Split indices in store into train, valiation and test split.
train(optimizer, lr[, epochs, ...])	Train model.

sfaira.estimators.EstimatorKeras.get_one_time_tf_dataset

EstimatorKeras.**get_one_time_tf_dataset**(*idx*, *mode*, *batch_size=None*, *prefetch=None*)

sfaira.estimators.EstimatorKeras.init_model

EstimatorKeras.**init_model**(*clear_weight_cache=True*, *override_hyperpar=None*)
Instantiate the model. :return:

sfaira.estimators.EstimatorKeras.load_pretrained_weights

EstimatorKeras.**load_pretrained_weights**()
Loads model weights from local directory or zenodo.

sfaira.estimators.EstimatorKeras.load_weights_from_cache

EstimatorKeras.**load_weights_from_cache**()

sfaira.estimators.EstimatorKeras.save_weights_to_cache

EstimatorKeras.**save_weights_to_cache**()

sfaira.estimators.EstimatorKeras.split_train_val_test

abstract EstimatorKeras.**split_train_val_test**(*val_split: float*, *test_split: Union[float, dict]*)

Split indices in store into train, valiation and test split.

sfaira.estimators.EstimatorKeras.train

EstimatorKeras.**train**(*optimizer: str*, *lr: float*, *epochs: int = 1000*, *max_steps_per_epoch: Optional[int] = 20*, *batch_size: int = 128*, *validation_split: float = 0.1*, *test_split: Union[float, dict] = 0.0*, *validation_batch_size: int = 256*, *max_validation_steps: Optional[int] = 10*, *patience: int = 20*, *lr_schedule_min_lr: float = 1e-05*, *lr_schedule_factor: float = 0.2*, *lr_schedule_patience: int = 5*, *shuffle_buffer_size: Optional[int] = None*, *cache_full: bool = False*, *randomized_batch_access: bool = True*, *retrieval_batch_size: int = 512*, *prefetch: Optional[int] = 1*, *log_dir: Optional[str] = None*, *callbacks: Optional[list] = None*, *weighted: bool = False*, *verbose: int = 2*)

Train model.

Uses validation loss and maximum number of epochs as termination criteria.

Parameters

- **optimizer** – str corresponding to tf.keras optimizer to use for fitting.
- **lr** – Learning rate
- **epochs** – refer to tf.keras.models.Model.fit() documentation
- **max_steps_per_epoch** – Maximum steps per epoch.
- **batch_size** – refer to tf.keras.models.Model.fit() documentation
- **validation_split** – refer to tf.keras.models.Model.fit() documentation Refers to fraction of training data (1-test_split) to use for validation.
- **test_split** – Fraction of data to set apart for test model before train-validation split.
- **validation_batch_size** – Number of validation data observations to evaluate evaluation metrics on.
- **max_validation_steps** – Maximum number of validation steps to perform.
- **patience** – refer to tf.keras.models.Model.fit() documentation
- **lr_schedule_min_lr** – Minimum learning rate for learning rate reduction schedule.
- **lr_schedule_factor** – Factor to reduce learning rate by within learning rate reduction schedule when plateau is reached.

- **lr_schedule_patience** – Patience for learning rate reduction in learning rate reduction schedule.
- **shuffle_buffer_size** – tf.Dataset.shuffle(): buffer_size argument.
- **cache_full** – Whether to use tensorflow caching on full training and validation data.
- **randomized_batch_access** – Whether to randomize batches during reading (in generator). Lifts necessity of using a shuffle buffer on generator, however, batch composition stays unchanged over epochs unless there is overhangs in retrieval_batch_size in the raw data files, which often happens and results in modest changes in batch composition.
- **log_dir** – Directory to save tensorboard callback to. Disabled if None.
- **callbacks** – Add additional callbacks to the training call
- **weighted** –
- **verbose** –

Returns

sfaira.estimators.EstimatorKerasCelltype

class sfaira.estimators.**EstimatorKerasCelltype**(*data: typing.Union[anndata._core.anndata.AnnData, sfaira.data.store.stores.single.StoreSingleFeatureSpace], model_dir: typing.Optional[str], model_id: typing.Optional[str], model_topology: sfaira.versions.topologies.class_interface.TopologyContainer, weights_md5: typing.Optional[str] = None, cache_path: str = 'cache/', celltype_ontology: typing.Optional[sfaira.versions.metadata.base.OntologyObo] = None, max_class_weight: float = 1000.0, remove_unlabeled_cells: bool = True, adata_ids: sfaira.consts.adata_fields.AdataIds = <sfaira.consts.adata_fields.AdataIdsSfaira object>*)

Estimator class for the cell type model.

Attributes

model_type
ntypes
obs
obs_eval
obs_test
obs_train

continues on next page

Table 48 – continued from previous page

ontology_ids
ontology_names
organism
using_store
celltype_universe
model

sfaira.estimators.EstimatorKerasCelltype.model_type

property EstimatorKerasCelltype.**model_type**

sfaira.estimators.EstimatorKerasCelltype.ntypes

property EstimatorKerasCelltype.**ntypes**

sfaira.estimators.EstimatorKerasCelltype.obs

property EstimatorKerasCelltype.**obs**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasCelltype.obs_eval

property EstimatorKerasCelltype.**obs_eval**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasCelltype.obs_test

property EstimatorKerasCelltype.**obs_test**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasCelltype.obs_train

property EstimatorKerasCelltype.**obs_train**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasCelltype.ontology_ids

property EstimatorKerasCelltype.**ontology_ids**

sfaira.estimators.EstimatorKerasCelltype.ontology_names

property EstimatorKerasCelltype.**ontology_names**

sfaira.estimators.EstimatorKerasCelltype.organism

property EstimatorKerasCelltype.**organism**

sfaira.estimators.EstimatorKerasCelltype.using_store

property EstimatorKerasCelltype.**using_store**: bool

sfaira.estimators.EstimatorKerasCelltype.celltype_universe

EstimatorKerasCelltype.**celltype_universe**:
sfaira.versions.metadata.universe.CelltypeUniverse

sfaira.estimators.EstimatorKerasCelltype.model

EstimatorKerasCelltype.**model**:
Optional[sfaira.models.celltype.base.BasicModelKerasCelltype]

Methods

compute_gradients_input([test_data, ...])	
evaluate([batch_size, max_steps, weighted])	Evaluate the custom model on local data.
evaluate_any(idx[, batch_size, max_steps, ...])	Evaluate the custom model on any local data.
get_one_time_tf_dataset(idx, mode[, ...])	
init_model([clear_weight_cache, ...])	instantiate the model :return:
load_pretrained_weights()	Loads model weights from local directory or zenodo.
load_weights_from_cache()	
predict([batch_size, max_steps])	Return the prediction of the model
save_weights_to_cache()	
split_train_val_test(val_split, test_split)	Split indices in store into train, valiation and test split.
train(optimizer, lr[, epochs, ...])	Train model.
ytrue([batch_size, max_steps])	Return the true labels of the test set.

sfaira.estimators.EstimatorKerasCelltype.compute_gradients_input

EstimatorKerasCelltype.**compute_gradients_input**(*test_data: bool = False*, *abs_gradients: bool = True*)

sfaira.estimators.EstimatorKerasCelltype.evaluate

EstimatorKerasCelltype.**evaluate**(*batch_size: int = 128*, *max_steps: int = inf*, *weighted: bool = False*)

Evaluate the custom model on local data.

Defaults to run on full data if idx_test was not set before, ie. train() has not been called before.

Parameters

- **batch_size** – Batch size for evaluation.
- **max_steps** – Maximum steps before evaluation round is considered complete.
- **weighted** – Whether to use class weights in evaluation.

Returns Dictionary of metric names and values.

sfaira.estimators.EstimatorKerasCelltype.evaluate_any

EstimatorKerasCelltype.**evaluate_any**(*idx*, *batch_size: int = 128*, *max_steps: int = inf*, *weighted: bool = False*)

Evaluate the custom model on any local data.

Defaults to run on full data if idx is None.

Parameters

- **idx** – Indices of observations to evaluate on. Evaluates on all observations if None.
- **batch_size** – Batch size for evaluation.
- **max_steps** – Maximum steps before evaluation round is considered complete.
- **weighted** – Whether to use class weights in evaluation.

Returns Dictionary of metric names and values.

sfaira.estimators.EstimatorKerasCelltype.get_one_time_tf_dataset

EstimatorKerasCelltype.**get_one_time_tf_dataset**(*idx*, *mode*, *batch_size=None*, *prefetch=None*)

sfaira.estimators.EstimatorKerasCelltype.init_model

EstimatorKerasCelltype.**init_model**(*clear_weight_cache: bool = True*, *override_hyperpar: Optional[dict] = None*)

instantiate the model :return:

sfaira.estimators.EstimatorKerasCelltype.load_pretrained_weights

EstimatorKerasCelltype.**load_pretrained_weights**()
Loads model weights from local directory or zenodo.

sfaira.estimators.EstimatorKerasCelltype.load_weights_from_cache

EstimatorKerasCelltype.**load_weights_from_cache**()

sfaira.estimators.EstimatorKerasCelltype.predict

EstimatorKerasCelltype.**predict**(*batch_size: int = 128*, *max_steps: int = inf*)
Return the prediction of the model

Parameters

- **batch_size** – Batch size for evaluation.
- **max_steps** – Maximum steps before evaluation round is considered complete.

Returns Prediction tensor.

sfaira.estimators.EstimatorKerasCelltype.save_weights_to_cache

EstimatorKerasCelltype.**save_weights_to_cache**()

sfaira.estimators.EstimatorKerasCelltype.split_train_val_test

EstimatorKerasCelltype.**split_train_val_test**(*val_split: float*, *test_split: Union[float, dict]*)
Split indices in store into train, valiation and test split.

sfaira.estimators.EstimatorKerasCelltype.train

EstimatorKerasCelltype.**train**(*optimizer: str*, *lr: float*, *epochs: int = 1000*, *max_steps_per_epoch: Optional[int] = 20*, *batch_size: int = 128*, *validation_split: float = 0.1*, *test_split: Union[float, dict] = 0.0*, *validation_batch_size: int = 256*, *max_validation_steps: Optional[int] = 10*, *patience: int = 20*, *lr_schedule_min_lr: float = 1e-05*, *lr_schedule_factor: float = 0.2*, *lr_schedule_patience: int = 5*, *shuffle_buffer_size: Optional[int] = None*, *cache_full: bool = False*, *randomized_batch_access: bool = True*, *retrieval_batch_size: int = 512*, *prefetch: Optional[int] = 1*, *log_dir: Optional[str] = None*, *callbacks: Optional[list] = None*, *weighted: bool = False*, *verbose: int = 2*)
Train model.

Uses validation loss and maximum number of epochs as termination criteria.

Parameters

- **optimizer** – str corresponding to tf.keras optimizer to use for fitting.
- **lr** – Learning rate

- **epochs** – refer to tf.keras.models.Model.fit() documentation
- **max_steps_per_epoch** – Maximum steps per epoch.
- **batch_size** – refer to tf.keras.models.Model.fit() documentation
- **validation_split** – refer to tf.keras.models.Model.fit() documentation Refers to fraction of training data (1-test_split) to use for validation.
- **test_split** – Fraction of data to set apart for test model before train-validation split.
- **validation_batch_size** – Number of validation data observations to evaluate evaluation metrics on.
- **max_validation_steps** – Maximum number of validation steps to perform.
- **patience** – refer to tf.keras.models.Model.fit() documentation
- **lr_schedule_min_lr** – Minimum learning rate for learning rate reduction schedule.
- **lr_schedule_factor** – Factor to reduce learning rate by within learning rate reduction schedule when plateau is reached.
- **lr_schedule_patience** – Patience for learning rate reduction in learning rate reduction schedule.
- **shuffle_buffer_size** – tf.Dataset.shuffle(): buffer_size argument.
- **cache_full** – Whether to use tensorflow caching on full training and validation data.
- **randomized_batch_access** – Whether to randomize batches during reading (in generator). Lifts necessity of using a shuffle buffer on generator, however, batch composition stays unchanged over epochs unless there is overhangs in retrieval_batch_size in the raw data files, which often happens and results in modest changes in batch composition.
- **log_dir** – Directory to save tensorboard callback to. Disabled if None.
- **callbacks** – Add additional callbacks to the training call
- **weighted** –
- **verbose** –

Returns

sfaira.estimators.EstimatorKerasCelltype.ytrue

EstimatorKerasCelltype.**ytrue**(*batch_size: int = 128*, *max_steps: int = inf*)

Return the true labels of the test set.

Returns true labels

sfaira.estimators.EstimatorKerasEmbedding

class sfaira.estimators.**EstimatorKerasEmbedding**(*data: typing.Union[anndata._core.anndata.AnnData, numpy.ndarray, sfaira.data.store.stores.single.StoreSingleFeatureSpace], model_dir: typing.Optional[str], model_id: typing.Optional[str], model_topology: sfaira.versions.topologies.class_interface.TopologyContainer, weights_md5: typing.Optional[str] = None, cache_path: str = 'cache/', adata_ids: sfaira.consts.adata_fields.AdataIds = <sfaira.consts.adata_fields.AdataIdsSfaira object>*)

Estimator class for the embedding model.

Attributes

model_type
obs
obs_eval
obs_test
obs_train
organism
using_store
model

sfaira.estimators.EstimatorKerasEmbedding.model_type

property EstimatorKerasEmbedding.**model_type**

sfaira.estimators.EstimatorKerasEmbedding.obs

property EstimatorKerasEmbedding.**obs**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasEmbedding.obs_eval

property EstimatorKerasEmbedding.**obs_eval**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasEmbedding.obs_test

property EstimatorKerasEmbedding.**obs_test**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasEmbedding.obs_train

property EstimatorKerasEmbedding.**obs_train**: pandas.core.frame.DataFrame

sfaira.estimators.EstimatorKerasEmbedding.organism

property EstimatorKerasEmbedding.**organism**

sfaira.estimators.EstimatorKerasEmbedding.using_store

property EstimatorKerasEmbedding.**using_store**: bool

sfaira.estimators.EstimatorKerasEmbedding.model

EstimatorKerasEmbedding.**model**:
Optional[sfaira.models.embedding.base.BasicModelKerasEmbedding]

Methods

compute_gradients_input([batch_size, ...])	
evaluate([batch_size, max_steps])	Evaluate the custom model on test data.
evaluate_any(idx[, batch_size, max_steps])	Evaluate the custom model on any local data.
get_one_time_tf_dataset(idx, mode[, ...])	
init_model([clear_weight_cache, ...])	instantiate the model :return:
load_pretrained_weights()	Loads model weights from local directory or zenodo.
load_weights_from_cache()	
predict([batch_size])	return the prediction of the model
predict_embedding([batch_size, variational])	return the prediction in the latent space (z_mean for variational models)

continues on next page

Table 51 – continued from previous page

save_weights_to_cache()	
split_train_val_test(val_split, test_split)	Split indices in store into train, valiation and test split.
train(optimizer, lr[, epochs, ...])	Train model.

sfaira.estimators.EstimatorKerasEmbedding.compute_gradients_input

EstimatorKerasEmbedding.**compute_gradients_input**(*batch_size: int = 128*, *test_data: bool = False*, *abs_gradients: bool = True*, *per_celltype: bool = False*)

sfaira.estimators.EstimatorKerasEmbedding.evaluate

EstimatorKerasEmbedding.**evaluate**(*batch_size: int = 128*, *max_steps: int = inf*)

Evaluate the custom model on test data.

Defaults to run on full data if idx_test was not set before, ie. train() has not been called before.

Parameters

- **batch_size** – Batch size for evaluation.
- **max_steps** – Maximum steps before evaluation round is considered complete.

Returns Dictionary of metric names and values.

sfaira.estimators.EstimatorKerasEmbedding.evaluate_any

EstimatorKerasEmbedding.**evaluate_any**(*idx*, *batch_size: int = 128*, *max_steps: int = inf*)

Evaluate the custom model on any local data.

Defaults to run on full data if idx is None.

Parameters

- **idx** – Indices of observations to evaluate on. Evaluates on all observations if None.
- **batch_size** – Batch size for evaluation.
- **max_steps** – Maximum steps before evaluation round is considered complete.

Returns Dictionary of metric names and values.

sfaira.estimators.EstimatorKerasEmbedding.get_one_time_tf_dataset

EstimatorKerasEmbedding.**get_one_time_tf_dataset**(*idx*, *mode*, *batch_size=None*, *prefetch=None*)

sfaira.estimators.EstimatorKerasEmbedding.init_model

EstimatorKerasEmbedding.**init_model**(*clear_weight_cache: bool = True*, *override_hyperpar: Optional[dict] = None*)

instantiate the model :return:

sfaira.estimators.EstimatorKerasEmbedding.load_pretrained_weights

EstimatorKerasEmbedding.**load_pretrained_weights**()

Loads model weights from local directory or zenodo.

sfaira.estimators.EstimatorKerasEmbedding.load_weights_from_cache

EstimatorKerasEmbedding.**load_weights_from_cache**()

sfaira.estimators.EstimatorKerasEmbedding.predict

EstimatorKerasEmbedding.**predict**(*batch_size: int = 128*)

return the prediction of the model

Returns

prediction

sfaira.estimators.EstimatorKerasEmbedding.predict_embedding

EstimatorKerasEmbedding.**predict_embedding**(*batch_size: int = 128*, *variational: bool = False*)

return the prediction in the latent space (z_mean for variational models)

Params variational Whether toreturn the prediction of z, z_mean, z_log_var in the variational latent space.

Returns

latent space

sfaira.estimators.EstimatorKerasEmbedding.save_weights_to_cache

EstimatorKerasEmbedding.**save_weights_to_cache**()

sfaira.estimators.EstimatorKerasEmbedding.split_train_val_test

EstimatorKerasEmbedding.**split_train_val_test**(*val_split: float*, *test_split: Union[float, dict]*)

Split indices in store into train, valiation and test split.

sfaira.estimators.EstimatorKerasEmbedding.train

EstimatorKerasEmbedding.**train**(*optimizer: str*, *lr: float*, *epochs: int = 1000*, *max_steps_per_epoch: Optional[int] = 20*, *batch_size: int = 128*, *validation_split: float = 0.1*, *test_split: Union[float, dict] = 0.0*, *validation_batch_size: int = 256*, *max_validation_steps: Optional[int] = 10*, *patience: int = 20*, *lr_schedule_min_lr: float = 1e-05*, *lr_schedule_factor: float = 0.2*, *lr_schedule_patience: int = 5*, *shuffle_buffer_size: Optional[int] = None*, *cache_full: bool = False*, *randomized_batch_access: bool = True*, *retrieval_batch_size: int = 512*, *prefetch: Optional[int] = 1*, *log_dir: Optional[str] = None*, *callbacks: Optional[list] = None*, *weighted: bool = False*, *verbose: int = 2*)

Train model.

Uses validation loss and maximum number of epochs as termination criteria.

Parameters

- **optimizer** – str corresponding to tf.keras optimizer to use for fitting.
- **lr** – Learning rate
- **epochs** – refer to tf.keras.models.Model.fit() documentation
- **max_steps_per_epoch** – Maximum steps per epoch.
- **batch_size** – refer to tf.keras.models.Model.fit() documentation
- **validation_split** – refer to tf.keras.models.Model.fit() documentation Refers to fraction of training data (1-test_split) to use for validation.
- **test_split** – Fraction of data to set apart for test model before train-validation split.
- **validation_batch_size** – Number of validation data observations to evaluate evaluation metrics on.
- **max_validation_steps** – Maximum number of validation steps to perform.
- **patience** – refer to tf.keras.models.Model.fit() documentation
- **lr_schedule_min_lr** – Minimum learning rate for learning rate reduction schedule.
- **lr_schedule_factor** – Factor to reduce learning rate by within learning rate reduction schedule when plateau is reached.
- **lr_schedule_patience** – Patience for learning rate reduction in learning rate reduction schedule.
- **shuffle_buffer_size** – tf.Dataset.shuffle(): buffer_size argument.
- **cache_full** – Whether to use tensorflow caching on full training and validation data.
- **randomized_batch_access** – Whether to randomize batches during reading (in generator). Lifts necessity of using a shuffle buffer on generator, however, batch composition stays unchanged over epochs unless there is overhangs in retrieval_batch_size in the raw data files, which often happens and results in modest changes in batch composition.
- **log_dir** – Directory to save tensorboard callback to. Disabled if None.
- **callbacks** – Add additional callbacks to the training call

- **weighted** –
- **verbose** –

Returns

Model classes: `models`

Model classes from the sfaira model zoo API for advanced use.

Cell type models

Classes that wrap tensorflow cell type predictor models.

models.celltype.CellTypeMarker(in_dim, out_dim)	Marker gene-based cell type classifier: Learns whether or not each gene exceeds requires threshold and learns cell type assignment as linear combination of these marker gene presence probabilities.
models.celltype.CellTypeMarker(in_dim, out_dim)	Marker gene-based cell type classifier: Learns whether or not each gene exceeds requires threshold and learns cell type assignment as linear combination of these marker gene presence probabilities.
models.celltype.CellTypeMlp(in_dim, out_dim)	Multi-layer perceptron to predict cell type.
models.celltype.CellTypeMlpVersioned(...[, ...])	

sfaira.models.celltype.CellTypeMarker

class sfaira.models.celltype.**CellTypeMarker**(*in_dim: int*, *out_dim: int*, *l1_coef: float = 0.0*, *l2_coef: float = 0.0*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *bias_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *name='celltype_marker'*, ***kwargs*)

Marker gene-based cell type classifier: Learns whether or not each gene exceeds requires threshold and learns cell type assignment as linear combination of these marker gene presence probabilities. Activitiy and weight regularizers keep this sparse.

Attributes

version

sfaira.models.celltype.CellTypeMarker.version

property CellTypeMarker.**version**

Methods

predict(x, **kwarg)

sfaira.models.celltype.CellTypeMarker.predict

CellTypeMarker.**predict**(*x*, ***kwarg*)

sfaira.models.celltype.CellTypeMlp

class sfaira.models.celltype.**CellTypeMlp**(*in_dim: int*, *out_dim: int*, *units: List[int] = []*, *activation='relu'*, *use_bias=True*, *l1_coef: float = 0.0*, *l2_coef: float = 0.0*, *kernel_initializer='glorot_uniform'*, *bias_initializer='zeros'*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *name='celltype_mlp'*, ***kwargs*)

Multi-layer perceptron to predict cell type.

Attributes

version

sfaira.models.celltype.CellTypeMlp.version

property CellTypeMlp.**version**

Methods

predict(x, **kwarg)

sfaira.models.celltype.CellTypeMlp.predict

CellTypeMlp.**predict**(*x*, ***kwarg*)

sfaira.models.celltype.CellTypeMlpVersioned

class sfaira.models.celltype.**CellTypeMlpVersioned**(*celltypes_version:* sfaira.versions.metadata.universe.CelltypeUniverse, *topology_container:* sfaira.versions.topologies.class_interface.TopologyContainer, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.celltype.CellTypeMlpVersioned.version

property CellTypeMlpVersioned.**version**

Methods

predict(x, **kwarg)

sfaira.models.celltype.CellTypeMlpVersioned.predict

CellTypeMlpVersioned.**predict**(*x*, ***kwarg*)

Embedding models

Classes that wrap tensorflow embedding models.

models.embedding.ModelKerasAe(in_dim[, ...])	Combines the encoder and decoder into an end-to-end model for training.
models.embedding.ModelAeVersioned(...[, ...])	
models.embedding.ModelKerasVae(in_dim[, ...])	
models.embedding.ModelVaeVersioned(...[, ...])	
models.embedding.ModelKerasLinear(in_dim[, ...])	

continues on next page

Table 59 – continued from previous page

models.embedding.ModelLinearVersioned(...[, ...])	
models.embedding.ModelKerasVaeIAF(in_dim[, ...])	
models.embedding.ModelVaeIAFVersioned(...[, ...])	
models.embedding.ModelKerasVaeVamp(in_dim[, ...])	
models.embedding.ModelVaeVampVersioned(...)	

sfaira.models.embedding.ModelKerasAe

class sfaira.models.embedding.**ModelKerasAe**(*in_dim*, *latent_dim: Tuple = (64, 32, 64)*, *l2_coef: float = 0.0*, *l1_coef: float = 0.0*, *dropout_rate: float = 0.0*, *input_dropout: float = 0.0*, *batchnorm: bool = True*, *activation='relu'*, *init='glorot_uniform'*, *output_layer='nb'*)

Combines the encoder and decoder into an end-to-end model for training.

Attributes

version	

sfaira.models.embedding.ModelKerasAe.version

property ModelKerasAe.**version**

Methods

predict_embedding(x, **kwargs)	
predict_reconstructed(x, **kwargs)	

sfaira.models.embedding.ModelKerasAe.predict_embedding

ModelKerasAe.**predict_embedding**(*x*, ***kwargs*)

sfaira.models.embedding.ModelKerasAe.predict_reconstructed

ModelKerasAe.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelAeVersioned

class sfaira.models.embedding.**ModelAeVersioned**(*topology_container:* sfaira.versions.topologies.class_interface.TopologyContainer, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.embedding.ModelAeVersioned.version

property ModelAeVersioned.**version**

Methods

predict_embedding(x, **kwargs)
predict_reconstructed(x, **kwargs)

sfaira.models.embedding.ModelAeVersioned.predict_embedding

ModelAeVersioned.**predict_embedding**(*x*, ***kwargs*)

sfaira.models.embedding.ModelAeVersioned.predict_reconstructed

ModelAeVersioned.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelKerasVae

class sfaira.models.embedding.**ModelKerasVae**(*in_dim*, *latent_dim=(128, 64, 2, 64, 128)*, *dropout_rate=0.1*, *l1_coef: float = 0.0*, *l2_coef: float = 0.0*, *batchnorm: bool = False*, *activation='tanh'*, *init='glorot_uniform'*, *output_layer='nb'*)

Attributes

version

sfaira.models.embedding.ModelKerasVae.version

property ModelKerasVae.**version**

Methods

predict_embedding(x[, variational])
predict_reconstructed(x)

sfaira.models.embedding.ModelKerasVae.predict_embedding

ModelKerasVae.**predict_embedding**(*x*, *variational=False*)

sfaira.models.embedding.ModelKerasVae.predict_reconstructed

ModelKerasVae.**predict_reconstructed**(*x: numpy.ndarray*)

sfaira.models.embedding.ModelVaeVersioned

class sfaira.models.embedding.**ModelVaeVersioned**(*topology_container: sfaira.versions.topologies.class_interface.TopologyContainer*, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.embedding.ModelVaeVersioned.version

property ModelVaeVersioned.**version**

Methods

predict_embedding(x[, variational])
predict_reconstructed(x)

sfaira.models.embedding.ModelVaeVersioned.predict_embedding

ModelVaeVersioned.**predict_embedding**(*x*, *variational=False*)

sfaira.models.embedding.ModelVaeVersioned.predict_reconstructed

ModelVaeVersioned.**predict_reconstructed**(*x: numpy.ndarray*)

sfaira.models.embedding.ModelKerasLinear

class sfaira.models.embedding.**ModelKerasLinear**(*in_dim*, *latent_dim: int = 10*, *positive_components: bool = False*, *l2_coef: float = 0.0*, *l1_coef: float = 0.0*, *dropout_rate=None*, *output_layer='nb'*)

Attributes

version

sfaira.models.embedding.ModelKerasLinear.version

property ModelKerasLinear.**version**

Methods

predict_embedding(x, **kwargs)
predict_reconstructed(x, **kwargs)

sfaira.models.embedding.ModelKerasLinear.predict_embedding

ModelKerasLinear.**predict_embedding**(*x*, ***kwargs*)

sfaira.models.embedding.ModelKerasLinear.predict_reconstructed

ModelKerasLinear.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelLinearVersioned

class sfaira.models.embedding.**ModelLinearVersioned**(*topology_container:* sfaira.versions.topologies.class_interface.TopologyContainer, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.embedding.ModelLinearVersioned.version

property ModelLinearVersioned.**version**

Methods

predict_embedding(x, **kwargs)
predict_reconstructed(x, **kwargs)

sfaira.models.embedding.ModelLinearVersioned.predict_embedding

ModelLinearVersioned.**predict_embedding**(*x*, ***kwargs*)

sfaira.models.embedding.ModelLinearVersioned.predict_reconstructed

ModelLinearVersioned.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelKerasVaeIAF

class sfaira.models.embedding.**ModelKerasVaeIAF**(*in_dim*, *latent_dim=(128, 64, 2, 64, 128)*, *n_iaf=2*, *dropout_rate=0.1*, *l2_coef: float = 0.0*, *l1_coef: float = 0.0*, *mc_samples=10*, *batchnorm=False*, *activation='tanh'*, *init='glorot_uniform'*, *output_layer='nb'*)

Attributes

version

sfaira.models.embedding.ModelKerasVaeIAF.version

property ModelKerasVaeIAF.**version**

Methods

predict_embedding(x[, variational, return_z0])
predict_reconstructed(x, **kwargs)

sfaira.models.embedding.ModelKerasVaeIAF.predict_embedding

ModelKerasVaeIAF.**predict_embedding**(*x*, *variational=False*, *return_z0=False*)

sfaira.models.embedding.ModelKerasVaeIAF.predict_reconstructed

ModelKerasVaeIAF.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelVaeIAFVersioned

class sfaira.models.embedding.**ModelVaeIAFVersioned**(*topology_container: sfaira.versions.topologies.class_interface.TopologyContainer*, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.embedding.ModelVaeIAFVersioned.version

property ModelVaeIAFVersioned.**version**

Methods

predict_embedding(x[, variational, return_z0])
predict_reconstructed(x, **kwargs)

sfaira.models.embedding.ModelVaeIAFVersioned.predict_embedding

ModelVaeIAFVersioned.**predict_embedding**(*x*, *variational=False*, *return_z0=False*)

sfaira.models.embedding.ModelVaeIAFVersioned.predict_reconstructed

ModelVaeIAFVersioned.**predict_reconstructed**(*x*, ***kwargs*)

sfaira.models.embedding.ModelKerasVaeVamp

class sfaira.models.embedding.**ModelKerasVaeVamp**(*in_dim*, *latent_dim=(256, 128, (32, 32), 128, 256)*, *dropout_rate=0.1*, *l1_coef: float = 0.0*, *l2_coef: float = 0.0*, *batch_size_u: int = 500*, *batchnorm: bool = False*, *activation='tanh'*, *init='glorot_uniform'*, *output_layer='nb'*)

Attributes

version

sfaira.models.embedding.ModelKerasVaeVamp.version

property ModelKerasVaeVamp.**version**

Methods

predict_embedding(x[, variational])
predict_reconstructed(x)

sfaira.models.embedding.ModelKerasVaeVamp.predict_embedding

ModelKerasVaeVamp.**predict_embedding**(*x*, *variational=False*)

sfaira.models.embedding.ModelKerasVaeVamp.predict_reconstructed

ModelKerasVaeVamp.**predict_reconstructed**(*x: numpy.ndarray*)

sfaira.models.embedding.ModelVaeVampVersioned

class sfaira.models.embedding.**ModelVaeVampVersioned**(*topology_container:* sfaira.versions.topologies.class_interface.TopologyContainer, *override_hyperpar: Optional[dict] = None*)

Attributes

version

sfaira.models.embedding.ModelVaeVampVersioned.version

property ModelVaeVampVersioned.**version**

Methods

predict_embedding(x[, variational])	
predict_reconstructed(x)	

sfaira.models.embedding.ModelVaeVampVersioned.predict_embedding

ModelVaeVampVersioned.**predict_embedding**(*x*, *variational=False*)

sfaira.models.embedding.ModelVaeVampVersioned.predict_reconstructed

ModelVaeVampVersioned.**predict_reconstructed**(*x: numpy.ndarray*)

Train: train

The interface for training sfaira compatible models.

Trainer classes

Classes that wrap estimator classes to use in grid search training.

train.TrainModelCelltype(model_path, data, ...)	
train.TrainModelEmbedding(model_path, data)	

sfaira.train.TrainModelCelltype

class sfaira.train.**TrainModelCelltype**(*model_path: str*, *data: Union[str, anndata._core.anndata.AnnData, sfaira.data.dataloaders.super_group.Universe, sfaira.data.store.stores.single.StoreSingleFeatureSpace]*, *fn_target_universe: str*)

Attributes

topology_dict	
estimator	

sfaira.train.TrainModelCelltype.topology_dict

property TrainModelCelltype.**topology_dict**: dict

sfaira.train.TrainModelCelltype.estimator

TrainModelCelltype.**estimator**: *sfaira.estimators.keras.base.EstimatorKerasCelltype*

Methods

init_estim([override_hyperpar])	
load_into_memory()	Loads backed objects from DistributedStoreBase into single adata object in memory in .data slot.
n_counts(idx)	
save(fn[, model, specific])	Save weights and summary statistics.
save_eval(fn[, eval_weighted])	

sfaira.train.TrainModelCelltype.init_estim

TrainModelCelltype.**init_estim**(*override_hyperpar: Optional[dict] = None*)

sfaira.train.TrainModelCelltype.load_into_memory

TrainModelCelltype.**load_into_memory**()

Loads backed objects from DistributedStoreBase into single adata object in memory in .data slot. :return:

sfaira.train.TrainModelCelltype.n_counts

TrainModelCelltype.**n_counts**(*idx*)

sfaira.train.TrainModelCelltype.save

TrainModelCelltype.**save**(*fn: str*, *model: bool = True*, *specific: bool = True*)
Save weights and summary statistics.

sfaira.train.TrainModelCelltype.save_eval

TrainModelCelltype.**save_eval**(*fn: str*, *eval_weighted: bool = False*, ***kwargs*)

sfaira.train.TrainModelEmbedding

class sfaira.train.**TrainModelEmbedding**(*model_path: str*, *data: Union[str, anndata._core.anndata.AnnData, sfaira.data.dataloaders.super_group.Universe, sfaira.data.store.stores.single.StoreSingleFeatureSpace]*)

Attributes

topology_dict
estimator

sfaira.train.TrainModelEmbedding.topology_dict

property TrainModelEmbedding.**topology_dict**: **dict**

sfaira.train.TrainModelEmbedding.estimator

TrainModelEmbedding.**estimator**: ***sfaira.estimators.keras.base.EstimatorKerasEmbedding***

Methods

init_estim([override_hyperpar])	
load_into_memory()	Loads backed objects from DistributedStoreBase into single adata object in memory in .data slot.
n_counts(idx)	
save(fn[, model, specific])	Save weights and summary statistics.
save_eval(fn, **kwargs)	

sfaira.train.TrainModelEmbedding.init_estim

TrainModelEmbedding.**init_estim**(*override_hyperpar: Optional[dict] = None*)

sfaira.train.TrainModelEmbedding.load_into_memory

TrainModelEmbedding.**load_into_memory**()
Loads backed objects from DistributedStoreBase into single adata object in memory in .data slot. :return:

sfaira.train.TrainModelEmbedding.n_counts

TrainModelEmbedding.**n_counts**(*idx*)

sfaira.train.TrainModelEmbedding.save

TrainModelEmbedding.**save**(*fn: str*, *model: bool = True*, *specific: bool = True*)
Save weights and summary statistics.

sfaira.train.TrainModelEmbedding.save_eval

TrainModelEmbedding.**save_eval**(*fn: str*, ***kwargs*)

Grid search summaries

Classes to pool evaluation metrics across fits in a grid search.

train.GridsearchContainer(source_path, cv)	
train.SummarizeGridsearchCelltype(...[, ...])	
train.SummarizeGridsearchEmbedding(...[, ...])	

sfaira.train.GridsearchContainer

class sfaira.train.**GridsearchContainer**(*source_path: dict*, *cv: bool*)

Attributes

cv_keys	Returns keys of cross-validation used in dictionaries in this class.
histories	
evals	
run_ids	
gs_keys	
summary_tab	
cv	
source_path	
model_id_len	

sfaira.train.GridsearchContainer.cv_keys

property GridsearchContainer.**cv_keys**: List[str]

Returns keys of cross-validation used in dictionaries in this class.

Returns list of string keys

sfaira.train.GridsearchContainer.histories

GridsearchContainer.**histories**: Union[None, dict]

sfaira.train.GridsearchContainer.evals

GridsearchContainer.**evals**: Union[None, dict]

sfaira.train.GridsearchContainer.run_ids

GridsearchContainer.**run_ids**: Union[None, list]

sfaira.train.GridsearchContainer.gs_keys

GridsearchContainer.**gs_keys**: Union[None, dict]

sfaira.train.GridsearchContainer.summary_tab

GridsearchContainer.**summary_tab**: Union[None, pandas.core.frame.DataFrame]

sfaira.train.GridsearchContainer.cv

GridsearchContainer.**cv**: bool

sfaira.train.GridsearchContainer.source_path

GridsearchContainer.**source_path**: dict

sfaira.train.GridsearchContainer.model_id_len

GridsearchContainer.**model_id_len**: Union[None, int]

Methods

best_model_by_partition(partition_select, ...)	**param partition_select**
get_best_model_ids(tab, metric_select, ...)	**param tab**
load_gs(gs_ids)	Loads all relevant data of a grid search.
load_y(hat_or_true, run_id)	
plot_best_model_by_hyperparam(metric_select)	Produces boxplots for all hyperparameters choices by organ.
plot_completions([groupby, height_fig, ...])	Plot number of completed grid search points by category.
plot_training_history(metric_select, metric_show)	Plot train and validation loss during training and learning rate reduction for each organ
save_best_weight(path[, partition, metric, ...])	Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.
write_best_hyparam(write_path[, subset, ...])	

sfaira.train.GridsearchContainer.best_model_by_partition

GridsearchContainer.**best_model_by_partition**(*partition_select: str*, *metric_select: str*, *cv_mode: str = 'mean'*, *subset: dict = {}*, *return_run_only: bool = False*, *grouping: list = ['organ', 'model_type']*)

Parameters

- **partition_select** –
- **metric_select** –
- **cv_mode** –
- **subset** –
- **return_run_only** –
- **grouping** –

Returns

sfaira.train.GridsearchContainer.get_best_model_ids

GridsearchContainer.**get_best_model_ids**(*tab*, *metric_select: str*, *partition_select: str*, *subset: dict = {}*, *cv_mode: str = 'mean'*)

Parameters

- **tab** –
- **metric_select** –
- **partition_select** –
- **subset** –
- **cv_mode** –

Returns

sfaira.train.GridsearchContainer.load_gs

GridsearchContainer.**load_gs**(*gs_ids: List[str]*)

Loads all relevant data of a grid search.

Parameters **gs_ids** –

Returns

sfaira.train.GridsearchContainer.load_y

GridsearchContainer.**load_y**(*hat_or_true: str*, *run_id: str*)

sfaira.train.GridsearchContainer.plot_best_model_by_hyperparam

GridsearchContainer.**plot_best_model_by_hyperparam**(*metric_select: str*, *param_hue='lr'*, *partition_select: str = 'val'*, *partition_show: str = 'test'*, *subset: dict = {}*, *param_x: Union[tuple, list] = ('lr', 'depth', 'width', 'dropout', 'l1', 'l2')*, *show_swarm: bool = False*, *panel_width: float = 4.0*, *panel_height: float = 2.0*)

Produces boxplots for all hyperparameters choices by organ.

Parameters

- **partition** – "train" or "eval" or "test" partition of data.
- **metric_select** – Metric to plot.
- **param_x** – Hyper-parameter for x-axis partition.
- **param_hue** – Hyper-parameter for hue-axis partition.
- **panel_width** –
- **panel_height** –

Returns

sfaira.train.GridsearchContainer.plot_completions

GridsearchContainer.**plot_completions**(*groupby=['depth', 'width', 'lr', 'dropout', 'l1', 'l2']*, *height_fig=7*, *width_fig=7*)

Plot number of completed grid search points by category.

Parameters

- **groupby** –
- **height_fig** –
- **width_fig** –

Returns

sfaira.train.GridsearchContainer.plot_training_history

GridsearchContainer.**plot_training_history**(*metric_select: str*, *metric_show: str*, *partition_select: str = 'val'*, *subset: dict = {}*, *cv_key: Optional[str] = None*, *log_loss: bool = False*)

Plot train and validation loss during training and learning rate reduction for each organ

The partition that is shown in train+val by default because these are the only ones recorded during training.

Parameters

- **metric_select** – metric to select best model by

- **metric_show** – metric to show as function of training progress, together with loss and learing rate.
- **partition_select** – "train" or "eval" or "test" partition of data to select fit by.
- **metric_select** – Metric to select fit by.
- **cv_key** – Index of cross-validation to plot training history for.
- **log_loss** –

Returns

sfaira.train.GridsearchContainer.save_best_weight

GridsearchContainer.**save_best_weight**(*path: str*, *partition: str = 'val'*, *metric: str = 'loss'*, *subset: dict = {}*)

Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.

Parameters

- **path** – Target file to save to. This is intended to be the zoo directory ready for upload.
- **partition** –
- **metric** –
- **subset** –

Returns

sfaira.train.GridsearchContainer.write_best_hyparam

GridsearchContainer.**write_best_hyparam**(*write_path*, *subset: dict = {}*, *partition: str = 'test'*, *metric: str = 'custom_negll'*, *cvs: Union[None, List[int]] = None*)

sfaira.train.SummarizeGridsearchCelltype

class sfaira.train.**SummarizeGridsearchCelltype**(*source_path: dict*, *cv: bool*, *model_id_len: int = 3*)

Attributes

cv_keys	Returns keys of cross-validation used in dictionaries in this class.
loss_idx	
acc_idx	

sfaira.train.SummarizeGridsearchCelltype.cv_keys

property SummarizeGridsearchCelltype.**cv_keys: List[str]**

Returns keys of cross-validation used in dictionaries in this class.

Returns list of string keys

sfaira.train.SummarizeGridsearchCelltype.loss_idx

SummarizeGridsearchCelltype.**loss_idx: int**

sfaira.train.SummarizeGridsearchCelltype.acc_idx

SummarizeGridsearchCelltype.**acc_idx: int**

Methods

best_model_by_partition(partition_select, ...)	**param partition_select**
best_model_celltype([subset, partition, ...])	
create_summary_tab()	
get_best_model_ids(tab, metric_select, ...)	**param tab**
load_gs(gs_ids)	Loads all relevant data of a grid search.
load_ontology_names(run_id)	Loads ontology ids from a specific model of a previously loaded grid search.
load_y(hat_or_true, run_id)	
plot_best([rename_levels, partition_select, ...])	Plot accuracy or other metric heatmap by organ and model type.
plot_best_classwise_heatmap(organ, organism, ...)	Plot evaluation metric heatmap for specified organ by cell classes and model types.
plot_best_classwise_scatter(organ, organism, ...)	Plot evaluation metric scatterplot for specified organ by cell classes and model types.
plot_best_model_by_hyperparam(metric_select)	Produces boxplots for all hyperparameters choices by organ.
plot_completions([groupby, height_fig, ...])	Plot number of completed grid search points by category.
plot_training_history(metric_select, metric_show)	Plot train and validation loss during training and learning rate reduction for each organ
save_best_weight(path[, partition, metric, ...])	Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.
write_best_hyparam(write_path[, subset, ...])	

sfaira.train.SummarizeGridsearchCelltype.best_model_by_partition

SummarizeGridsearchCelltype.**best_model_by_partition**(*partition_select: str*, *metric_select: str*, *cv_mode: str = 'mean'*, *subset: dict = {}*, *return_run_only: bool = False*, *grouping: list = ['organ', 'model_type']*)

Parameters

- **partition_select** –
- **metric_select** –
- **cv_mode** –
- **subset** –
- **return_run_only** –
- **grouping** –

Returns

sfaira.train.SummarizeGridsearchCelltype.best_model_celltype

SummarizeGridsearchCelltype.**best_model_celltype**(*subset: dict = {}*, *partition: str = 'val'*, *metric: str = 'loss'*, *cvs: Union[None, List[int]] = None*)

sfaira.train.SummarizeGridsearchCelltype.create_summary_tab

SummarizeGridsearchCelltype.**create_summary_tab**()

sfaira.train.SummarizeGridsearchCelltype.get_best_model_ids

SummarizeGridsearchCelltype.**get_best_model_ids**(*tab*, *metric_select: str*, *partition_select: str*, *subset: dict = {}*, *cv_mode: str = 'mean'*)

Parameters

- **tab** –
- **metric_select** –
- **partition_select** –
- **subset** –
- **cv_mode** –

Returns

sfaira.train.SummarizeGridsearchCelltype.load_gs

SummarizeGridsearchCelltype.**load_gs**(*gs_ids: List[str]*)

Loads all relevant data of a grid search.

Parameters **gs_ids** –

Returns

sfaira.train.SummarizeGridsearchCelltype.load_ontology_names

SummarizeGridsearchCelltype.**load_ontology_names**(*run_id: str*)

Loads ontology ids from a specific model of a previously loaded grid search.

Parameters **run_id** –

Returns

sfaira.train.SummarizeGridsearchCelltype.load_y

SummarizeGridsearchCelltype.**load_y**(*hat_or_true: str*, *run_id: str*)

sfaira.train.SummarizeGridsearchCelltype.plot_best

SummarizeGridsearchCelltype.**plot_best**(*rename_levels=[]*, *partition_select: str = 'val'*, *partition_show: str = 'test'*, *metric_select: str = 'acc'*, *metric_show: str = 'acc'*, *collapse_cv: str = 'max'*, *vmin=None*, *vmax=None*, *height_fig=7*, *width_fig=7*)

Plot accuracy or other metric heatmap by organ and model type.

Parameters

- **rename_levels** –
- **partition_select** –
- **partition_show** –
- **metric_select** –
- **metric_show** –
- **collapse_cv** –
- **vmin** –
- **vmax** –
- **height_fig** –
- **width_fig** –

Returns

sfaira.train.SummarizeGridsearchCelltype.plot_best_classwise_heatmap

SummarizeGridsearchCelltype.**plot_best_classwise_heatmap**(*organ: str*, *organism: str*, *datapath: str*, *store_format: str*, *targetpath: str*, *configpath: str*, *partition_select: str = 'val'*, *metric_select: str = 'custom_cce_agg'*, *metric_show: str = 'f1'*, *collapse_cv: str = 'mean'*, *min_cells: int = 10*, *height_fig: int = 7*, *width_fig: int = 7*)

Plot evaluation metric heatmap for specified organ by cell classes and model types.

Parameters

- **organ** – Organ to plot in heatmap.
- **organism** – Species that the gridsearch was run on
- **datapath** – Path to the local sfaira data repository
- **store_format** –
- **targetpath** –
- **configpath** –
- **partition_select** – Based on which partition to select the best model - train - val - test - all
- **metric_select** – Based on which metric to select the best model - loss - accuracy - custom_cce_agg - acc_agg - f1 - tpr - fpr
- **metric_show** – Which classwise metric to plot. - accuracy - f1
- **collapse_cv** – How to collapse over the single cv runs.
- **min_cells** – Minimum number of cells of a type must be present in the whole dataset for that class to be included in the plot.
- **height_fig** – Figure height.
- **width_fig** – Figure width.

Returns fig, axs, sns_data_heatmap

sfaira.train.SummarizeGridsearchCelltype.plot_best_classwise_scatter

SummarizeGridsearchCelltype.**plot_best_classwise_scatter**(*organ: str*, *organism: str*, *datapath: str*, *store_format: str*, *targetpath: str*, *configpath: str*, *partition_select: str = 'val'*, *metric_select: str = 'custom_cce_agg'*, *metric_show: str = 'f1'*, *collapse_cv: str = 'mean'*, *min_cells: int = 10*, *height_fig: int = 7*, *width_fig: int = 7*, *annotate_thres_ncells: int = 1000*, *annotate_thres_f1: float = 0.5*)

Plot evaluation metric scatterplot for specified organ by cell classes and model types.

Parameters

- **organ** – Organ to plot in heatmap.
- **organism** – Organism that the gridsearch was run on
- **datapath** – Path to the local sfaira data repository
- **store_format** –
- **targetpath** –
- **configpath** –
- **partition_select** – Based on which partition to select the best model - train - val - test - all
- **metric_select** – Based on which metric to select the best model - loss - accuracy - custom_cce_agg - acc_agg - f1 - tpr - fpr
- **metric_show** – Which classwise metric to plot. - accuracy - f1
- **collapse_cv** – How to collapse over the single cv runs.
- **min_cells** – Minimum number of cells of a type must be present in the whole dataset for that class to be included in the plot.
- **height_fig** – Figure height.
- **width_fig** – Figure width.
- **annotate_thres_ncells** –
- **annotate_thres_f1** –

Returns fig, axs, sns_data_scatter

sfaira.train.SummarizeGridsearchCelltype.plot_best_model_by_hyperparam

SummarizeGridsearchCelltype.**plot_best_model_by_hyperparam**(*metric_select: str*, *param_hue='lr'*, *partition_select: str = 'val'*, *partition_show: str = 'test'*, *subset: dict = {}*, *param_x: Union[tuple, list] = ('lr', 'depth', 'width', 'dropout', 'l1', 'l2')*, *show_swarm: bool = False*, *panel_width: float = 4.0*, *panel_height: float = 2.0*)

Produces boxplots for all hyperparameters choices by organ.

Parameters

- **partition** – “train” or “eval” or “test” partition of data.
- **metric_select** – Metric to plot.
- **param_x** – Hyper-parameter for x-axis partition.
- **param_hue** – Hyper-parameter for hue-axis partition.
- **panel_width** –
- **panel_height** –

Returns

sfaira.train.SummarizeGridsearchCelltype.plot_completions

SummarizeGridsearchCelltype.**plot_completions**(*groupby=['depth', 'width', 'lr', 'dropout', 'l1', 'l2']*, *height_fig=7*, *width_fig=7*)

Plot number of completed grid search points by category.

Parameters

- **groupby** –
- **height_fig** –
- **width_fig** –

Returns

sfaira.train.SummarizeGridsearchCelltype.plot_training_history

SummarizeGridsearchCelltype.**plot_training_history**(*metric_select: str*, *metric_show: str*, *partition_select: str = 'val'*, *subset: dict = {}*, *cv_key: Optional[str] = None*, *log_loss: bool = False*)

Plot train and validation loss during training and learning rate reduction for each organ

The partition that is shown in train+val by default because these are the only ones recorded during training.

Parameters

- **metric_select** – metric to select best model by
- **metric_show** – metric to show as function of training progress, together with loss and learing rate.
- **partition_select** – “train” or “eval” or “test” partition of data to select fit by.
- **metric_select** – Metric to select fit by.
- **cv_key** – Index of cross-validation to plot training history for.
- **log_loss** –

Returns

sfaira.train.SummarizeGridsearchCelltype.save_best_weight

SummarizeGridsearchCelltype.**save_best_weight**(*path: str*, *partition: str = 'val'*, *metric: str = 'loss'*, *subset: dict = {}*)

Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.

Parameters

- **path** – Target file to save to. This is intended to be the zoo directory ready for upload.
- **partition** –
- **metric** –
- **subset** –

Returns

sfaira.train.SummarizeGridsearchCelltype.write_best_hyparam

SummarizeGridsearchCelltype.**write_best_hyparam**(*write_path*, *subset: dict = {}*, *partition: str = 'test'*, *metric: str = 'custom_negll'*, *cvs: Union[None, List[int]] = None*)

sfaira.train.SummarizeGridsearchEmbedding

class sfaira.train.**SummarizeGridsearchEmbedding**(*source_path: dict*, *cv: bool*, *loss_idx: int = 0*, *mse_idx: int = 1*, *model_id_len: int = 3*)

Attributes

List	alias of List
Union	
cv_keys	Returns keys of cross-validation used in dictionaries in this class.
loss_idx	
mse_idx	

sfaira.train.SummarizeGridsearchEmbedding.List

SummarizeGridsearchEmbedding.**List**
alias of List

alias of List .. autoattribute:: SummarizeGridsearchEmbedding.List

sfaira.train.SummarizeGridsearchEmbedding.Union

SummarizeGridsearchEmbedding.**Union = typing.Union**

sfaira.train.SummarizeGridsearchEmbedding.cv_keys

property SummarizeGridsearchEmbedding.**cv_keys: List[str]**
Returns keys of cross-validation used in dictionaries in this class.

Returns list of string keys

sfaira.train.SummarizeGridsearchEmbedding.loss_idx

SummarizeGridsearchEmbedding.**loss_idx**: int

sfaira.train.SummarizeGridsearchEmbedding.mse_idx

SummarizeGridsearchEmbedding.**mse_idx**: int

Methods

best_model_by_partition(partition_select, ...)	**param partition_select**
best_model_embedding([subset, partition, ...])	
create_summary_tab()	
get_best_model_ids(tab, metric_select, ...)	**param tab**
get_gradients_by_celltype(model_organ, ...)	Compute gradients across latent units with respect to input features for each cell type.
load_gs(gs_ids)	Loads all relevant data of a grid search.
load_y(hat_or_true, run_id)	
plot_active_latent_units(organ, topology_version)	Plots latent unit activity measured by empirical variance of the expected latent space.
plot_best([rename_levels, partition_select, ...])	**param rename_levels**
plot_best_model_by_hyperparam(metric_select)	Produces boxplots for all hyperparameters choices by organ.
plot_completions([groupby, height_fig, ...])	Plot number of completed grid search points by category.
plot_gradient_cor(model_organ, data_organ, ...)	Plot correlation heatmap of gradient vectors accumulated on input features between cell types or models.
plot_gradient_distr(model_organ, data_organ, ...)	
plot_npc(organ, topology_version[, cvs])	Plots the explained variance ration that accumulates explained variation of the latent space's ordered principal components.
plot_training_history(metric_select, metric_show)	Plot train and validation loss during training and learning rate reduction for each organ
save_best_weight(path[, partition, metric, ...])	Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.
write_best_hyparam(write_path[, subset, ...])	

sfaira.train.SummarizeGridsearchEmbedding.best_model_by_partition

SummarizeGridsearchEmbedding.**best_model_by_partition**(*partition_select: str*, *metric_select: str*, *cv_mode: str = 'mean'*, *subset: dict = {}*, *return_run_only: bool = False*, *grouping: list = ['organ', 'model_type']*)

Parameters

- **partition_select** –
- **metric_select** –
- **cv_mode** –
- **subset** –
- **return_run_only** –
- **grouping** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.best_model_embedding

SummarizeGridsearchEmbedding.**best_model_embedding**(*subset: dict = {}*, *partition: str = 'val'*, *metric: str = 'loss'*, *cvs: Union[None, List[int]] = None*)

sfaira.train.SummarizeGridsearchEmbedding.create_summary_tab

SummarizeGridsearchEmbedding.**create_summary_tab**()

sfaira.train.SummarizeGridsearchEmbedding.get_best_model_ids

SummarizeGridsearchEmbedding.**get_best_model_ids**(*tab*, *metric_select: str*, *partition_select: str*, *subset: dict = {}*, *cv_mode: str = 'mean'*)

Parameters

- **tab** –
- **metric_select** –
- **partition_select** –
- **subset** –
- **cv_mode** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.get_gradients_by_celltype

SummarizeGridsearchEmbedding.**get_gradients_by_celltype**(*model_organ: str*, *data_organ: str*, *organism: Optional[str]*, *genome: Union[str, None, dict]*, *model_type: Union[str, List[str]]*, *metric_select: str*, *data_source: str*, *datapath*, *gene_type: str = 'protein_coding'*, *configpath: Optional[str] = None*, *store_format: Optional[str] = None*, *test_data=True*, *partition_select: str = 'val'*, *ignore_cache=False*, *min_cells=10*)

Compute gradients across latent units with respect to input features for each cell type.

Parameters

- **model_organ** –
- **data_organ** –
- **organism** –
- **model_type** –
- **metric_select** –
- **datapath** –
- **test_data** –
- **partition_select** –
- **ignore_cache** –
- **min_cells** –

Returns (cell types, input features) cumulative gradients

sfaira.train.SummarizeGridsearchEmbedding.load_gs

SummarizeGridsearchEmbedding.**load_gs**(*gs_ids: List[str]*)

Loads all relevant data of a grid search.

Parameters **gs_ids** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.load_y

SummarizeGridsearchEmbedding.**load_y**(*hat_or_true: str*, *run_id: str*)

sfaira.train.SummarizeGridsearchEmbedding.plot_active_latent_units

SummarizeGridsearchEmbedding.**plot_active_latent_units**(*organ*, *topology_version*, *cvs=None*)

Plots latent unit activity measured by empirical variance of the expected latent space. See: https://arxiv.org/abs/1509.00519 If an embedding file is found that contains z, z_mean, z_var (eg. output from predict_variational() function) the model will use z, and not z_mean.

sfaira.train.SummarizeGridsearchEmbedding.plot_best

SummarizeGridsearchEmbedding.**plot_best**(*rename_levels=[]*, *partition_select: str = 'val'*, *partition_show: str = 'test'*, *metric_select: str = 'll'*, *metric_show: str = 'll'*, *collapse_cv: str = 'min'*, *vmin=None*, *vmax=None*, *height_fig=7*, *width_fig=7*)

Parameters

- **rename_levels** –
- **partition_select** –
- **partition_show** –
- **metric_select** –
- **metric_show** –
- **collapse_cv** –
- **vmin** –
- **vmax** –
- **height_fig** –
- **width_fig** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.plot_best_model_by_hyperparam

SummarizeGridsearchEmbedding.**plot_best_model_by_hyperparam**(*metric_select: str*, *param_hue='lr'*, *partition_select: str = 'val'*, *partition_show: str = 'test'*, *subset: dict = {}*, *param_x: Union[tuple, list] = ('lr', 'depth', 'width', 'dropout', 'l1', 'l2')*, *show_swarm: bool = False*, *panel_width: float = 4.0*, *panel_height: float = 2.0*)

Produces boxplots for all hyperparameters choices by organ.

Parameters

- **partition** – “train” or “eval” or “test” partition of data.
- **metric_select** – Metric to plot.
- **param_x** – Hyper-parameter for x-axis partition.
- **param_hue** – Hyper-parameter for hue-axis partition.

- **panel_width** –
- **panel_height** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.plot_completions

SummarizeGridsearchEmbedding.**plot_completions**(*groupby=['depth', 'width', 'lr', 'dropout', 'l1', 'l2']*, *height_fig=7*, *width_fig=7*)

Plot number of completed grid search points by category.

Parameters

- **groupby** –
- **height_fig** –
- **width_fig** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.plot_gradient_cor

SummarizeGridsearchEmbedding.**plot_gradient_cor**(*model_organ: str*, *data_organ: str*, *model_type: Union[str, List[str]]*, *metric_select: str*, *datapath: str*, *data_source: str*, *organism: Optional[str] = None*, *genome: Optional[str] = None*, *configpath: Optional[str] = None*, *store_format: Optional[str] = None*, *test_data=True*, *gene_type: str = 'protein_coding'*, *partition_select: str = 'val'*, *height_fig=7*, *width_fig=7*, *ignore_cache=False*, *min_cells=10*, *by_type=True*, *vmin=0.0*, *vmax=1.0*, *save=None*)

Plot correlation heatmap of gradient vectors accumulated on input features between cell types or models.

Parameters

- **model_organ** –
- **data_organ** –
- **model_type** –
- **metric_select** –
- **datapath** –
- **configpath** –
- **store_format** –
- **test_data** –
- **partition_select** –
- **height_fig** –
- **width_fig** –
- **ignore_cache** –

- **min_cells** –
- **by_type** –
- **vmin** –
- **vmax** –
- **save** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.plot_gradient_distr

SummarizeGridsearchEmbedding.**plot_gradient_distr**(*model_organ: str*, *data_organ: str*, *model_type: Union[str, List[str]]*, *metric_select: str*, *datapath: str*, *data_source: str*, *organism: Optional[str] = None*, *genome: Optional[str] = None*, *configpath: Optional[str] = None*, *store_format: Optional[str] = None*, *test_data=True*, *gene_type: str = 'protein_coding'*, *partition_select: str = 'val'*, *normalize=True*, *remove_inactive=True*, *min_cells=10*, *bw=0.02*, *xlim=None*, *by_type=True*, *height_fig=7*, *width_fig=7*, *hist=False*, *ignore_cache=False*, *save=None*)

sfaira.train.SummarizeGridsearchEmbedding.plot_npc

SummarizeGridsearchEmbedding.**plot_npc**(*organ*, *topology_version*, *cvs=None*)

Plots the explained variance ration that accumulates explained variation of the latent space's ordered principal components. If an embedding file is found that contains z, z_mean, z_var (eg. output from predict_variational() function) the model will use z, and not z_mean.

sfaira.train.SummarizeGridsearchEmbedding.plot_training_history

SummarizeGridsearchEmbedding.**plot_training_history**(*metric_select: str*, *metric_show: str*, *partition_select: str = 'val'*, *subset: dict = {}*, *cv_key: Optional[str] = None*, *log_loss: bool = False*)

Plot train and validation loss during training and learning rate reduction for each organ

The partition that is shown in train+val by default because these are the only ones recorded during training.

Parameters

- **metric_select** – metric to select best model by
- **metric_show** – metric to show as function of training progress, together with loss and learing rate.
- **partition_select** – "train" or "eval" or "test" partition of data to select fit by.
- **metric_select** – Metric to select fit by.
- **cv_key** – Index of cross-validation to plot training history for.

- **log_loss** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.save_best_weight

SummarizeGridsearchEmbedding.**save_best_weight**(*path: str*, *partition: str = 'val'*, *metric: str = 'loss'*, *subset: dict = {}*)

Copies weight file from best hyperparameter setting from grid search directory to zoo directory with cleaned file name.

Parameters

- **path** – Target file to save to. This is intended to be the zoo directory ready for upload.
- **partition** –
- **metric** –
- **subset** –

Returns

sfaira.train.SummarizeGridsearchEmbedding.write_best_hyparam

SummarizeGridsearchEmbedding.**write_best_hyparam**(*write_path*, *subset: dict = {}*, *partition: str = 'test'*, *metric: str = 'custom_negll'*, *cvs: Union[None, List[int]] = None*)

Versions: versions

The interface for sfaira metadata management.

Genomes

Genome management.

versions.genomes.GenomeContainer([organism, ...])	Container class for a genome annotation for a specific release.

sfaira.versions.genomes.GenomeContainer

class sfaira.versions.genomes.**GenomeContainer**(*organism: Optional[str] = None*, *release: Optional[str] = None*)

Container class for a genome annotation for a specific release.

This class can be used to translate between symbols and ENSEMBL IDs for a specific assembly, to store specific gene subsets of an assembly, and to subselect genes by biotypes in an assembly.

Attributes

biotype	List of biotypes of genes in genome container.
ensembl	List of ENSEMBL IDs of genes in genome container.
id_to_symbols_dict	Dictionary-formatted map of ENSEMBL IDs to gene symbols.
n_var	Number of genes in genome container.
strippednames_to_id_dict	
symbol_to_id_dict	Dictionary-formatted map of gene symbols to ENSEMBL IDs.
symbols	List of symbols of genes in genome container.
genome_tab	
release	

sfaira.versions.genomes.GenomeContainer.biotype

property GenomeContainer.**biotype: List[str]**
List of biotypes of genes in genome container.

sfaira.versions.genomes.GenomeContainer.ensembl

property GenomeContainer.**ensembl: List[str]**
List of ENSEMBL IDs of genes in genome container.

sfaira.versions.genomes.GenomeContainer.id_to_symbols_dict

property GenomeContainer.**id_to_symbols_dict**
Dictionary-formatted map of ENSEMBL IDs to gene symbols.

sfaira.versions.genomes.GenomeContainer.n_var

property GenomeContainer.**n_var: int**
Number of genes in genome container.

sfaira.versions.genomes.GenomeContainer.strippednames_to_id_dict

property GenomeContainer.**strippednames_to_id_dict**

sfaira.versions.genomes.GenomeContainer.symbol_to_id_dict

property GenomeContainer.**symbol_to_id_dict**
Dictionary-formatted map of gene symbols to ENSEMBL IDs.

sfaira.versions.genomes.GenomeContainer.symbols

property GenomeContainer.**symbols: List[str]**
List of symbols of genes in genome container.

sfaira.versions.genomes.GenomeContainer.genome_tab

GenomeContainer.**genome_tab: pandas.core.frame.DataFrame**

sfaira.versions.genomes.GenomeContainer.release

GenomeContainer.**release: str**

Methods

load_genome()

organism()

set([biotype, symbols, ensg])	Subset by gene biotype or to gene list defined by identifiers (symbol or ensemble ID).
translate_id_to_symbols(x)	Translate ENSEMBL IDs to gene symbols.
translate_symbols_to_id(x)	Translate gene symbols to ENSEMBL IDs.

sfaira.versions.genomes.GenomeContainer.load_genome

GenomeContainer.**load_genome**()

sfaira.versions.genomes.GenomeContainer.organism

abstract GenomeContainer.**organism**()

sfaira.versions.genomes.GenomeContainer.set

GenomeContainer.**set**(*biotype: Union[None, str, List[str]] = None*, *symbols: Union[None, str, List[str]] = None*, *ensg: Union[None, str, List[str]] = None*)

Subset by gene biotype or to gene list defined by identifiers (symbol or ensemble ID).

Will subset by multiple factors if more than one parameter is not None.

Parameters

- **biotype** – Gene biotype(s) of gene(s) to subset genome to. Elements have to appear in genome. Separate in string via "," if choosing multiple or supply as list of string.
- **symbols** – Gene symbol(s) of gene(s) to subset genome to. Elements have to appear in genome. Separate in string via "," if choosing multiple or supply as list of string.
- **ensg** – Ensemble gene ID(s) of gene(s) to subset genome to. Elements have to appear in genome. Separate in string via "," if choosing multiple or supply as list of string.

sfaira.versions.genomes.GenomeContainer.translate_id_to_symbols

GenomeContainer.**translate_id_to_symbols**(*x: Union[str, Iterable[str]]*) → Union[str, List[str]]

Translate ENSEMBL IDs to gene symbols.

Parameters **x** – ENSEMBL ID(s) to translate.

Returns Gene symbols.

sfaira.versions.genomes.GenomeContainer.translate_symbols_to_id

GenomeContainer.**translate_symbols_to_id**(*x: Union[str, Iterable[str]]*) → Union[str, List[str]]

Translate gene symbols to ENSEMBL IDs.

Parameters **x** – Symbol(s) to translate.

Returns ENSEMBL IDs

Metadata

Dataset metadata management. Base classes to manage ontology files:

versions.metadata.Ontology()	
versions.metadata.OntologyList(terms, **kwargs)	Basic unordered ontology container.
versions.metadata.OntologyHierarchical()	Basic ordered ontology container
versions.metadata.OntologyObo(obo, **kwargs)	

continues on next page

Table 95 – continued from previous page

versions.metadata.OntologyOboCustom(obo, ...)	

sfaira.versions.metadata.Ontology

class sfaira.versions.metadata.**Ontology**

Attributes

leaves	

sfaira.versions.metadata.Ontology.leaves

Ontology.**leaves: List[str]**

Methods

is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
node_names()	
validate_node(x)	

sfaira.versions.metadata.Ontology.is_node

Ontology.**is_node**(*x: str*)

sfaira.versions.metadata.Ontology.map_node_suggestion

abstract Ontology.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.Ontology.node_names

abstract Ontology.**node_names**()

sfaira.versions.metadata.Ontology.validate_node

Ontology.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyList

class sfaira.versions.metadata.**OntologyList**(*terms: List[Union[str, bool, int]]*, ***kwargs*)

Basic unordered ontology container.

Node IDs and names are the same.

Attributes

leaves
n_leaves
node_ids
node_names
nodes

sfaira.versions.metadata.OntologyList.leaves

property OntologyList.**leaves**: List[str]

sfaira.versions.metadata.OntologyList.n_leaves

property OntologyList.**n_leaves**: int

sfaira.versions.metadata.OntologyList.node_ids

property OntologyList.**node_ids**: List[str]

sfaira.versions.metadata.OntologyList.node_names

property OntologyList.**node_names**: List[str]

sfaira.versions.metadata.OntologyList.nodes

OntologyList.**nodes**: list

Methods

convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
is_a(query, reference)	Checks if query node is reference node.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
synonym_node_properties()	
validate_node(x)	

sfaira.versions.metadata.OntologyList.convert_to_id

static OntologyList.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyList.convert_to_name

static OntologyList.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyList.get_ancestors

OntologyList.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyList.is_a

OntologyList.**is_a**(*query: str*, *reference: str*) → bool

Checks if query node is reference node.

Note that there is no notion of ancestors for list ontologies.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyList.is_a_node_id

OntologyList.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyList.is_a_node_name

OntologyList.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyList.is_node

OntologyList.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyList.map_node_suggestion

OntologyList.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.
- **n_suggest** – number of suggestions returned

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyList.prepare_maps_to_leaves

OntologyList.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Note that for a list ontology, this maps each node to itself.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyList.synonym_node_properties

OntologyList.**synonym_node_properties**() → List[str]

sfaira.versions.metadata.OntologyList.validate_node

OntologyList.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyHierarchical

class sfaira.versions.metadata.**OntologyHierarchical**

Basic ordered ontology container

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict

sfaira.versions.metadata.OntologyHierarchical.graph

property OntologyHierarchical.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyHierarchical.leaves

property OntologyHierarchical.**leaves**: List[str]

sfaira.versions.metadata.OntologyHierarchical.n_leaves

property OntologyHierarchical.**n_leaves**: int

sfaira.versions.metadata.OntologyHierarchical.node_ids

property OntologyHierarchical.**node_ids**: List[str]

sfaira.versions.metadata.OntologyHierarchical.node_names

property OntologyHierarchical.**node_names**: List[str]

sfaira.versions.metadata.OntologyHierarchical.nodes

property OntologyHierarchical.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyHierarchical.nodes_dict

property OntologyHierarchical.**nodes_dict**: dict

Methods

convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	

continues on next page

Table 101 – continued from previous page

is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
synonym_node_properties()	
validate_node(x)	

sfaira.versions.metadata.OntologyHierarchical.convert_to_id

OntologyHierarchical.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyHierarchical.convert_to_name

OntologyHierarchical.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyHierarchical.get_ancestors

OntologyHierarchical.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyHierarchical.get_descendants

OntologyHierarchical.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyHierarchical.get_effective_leaves

OntologyHierarchical.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyHierarchical.is_a

OntologyHierarchical.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyHierarchical.is_a_node_id

OntologyHierarchical.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyHierarchical.is_a_node_name

OntologyHierarchical.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyHierarchical.is_node

OntologyHierarchical.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyHierarchical.map_node_suggestion

abstract OntologyHierarchical.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyHierarchical.map_to_leaves

OntologyHierarchical.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – “ids”: IDs of mapped leave nodes “idx”: indicies in leave note list of mapped leave nodes

- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyHierarchical.prepare_maps_to_leaves

OntologyHierarchical.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyHierarchical.reset_root

OntologyHierarchical.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyHierarchical.synonym_node_properties

abstract OntologyHierarchical.**synonym_node_properties**() → List[str]

sfaira.versions.metadata.OntologyHierarchical.validate_node

OntologyHierarchical.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyObo

class sfaira.versions.metadata.**OntologyObo**(*obo: str*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict

sfaira.versions.metadata.OntologyObo.graph

property OntologyObo.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyObo.leaves

property OntologyObo.**leaves**: List[str]

sfaira.versions.metadata.OntologyObo.n_leaves

property OntologyObo.**n_leaves**: int

sfaira.versions.metadata.OntologyObo.node_ids

property OntologyObo.**node_ids**: List[str]

sfaira.versions.metadata.OntologyObo.node_names

property OntologyObo.**node_names**: List[str]

sfaira.versions.metadata.OntologyObo.nodes

property OntologyObo.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyObo.nodes_dict

property OntologyObo.**nodes_dict**: dict

Methods

convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	

continues on next page

Table 103 – continued from previous page

is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
synonym_node_properties()	
validate_node(x)	

sfaira.versions.metadata.OntologyObo.convert_to_id

OntologyObo.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyObo.convert_to_name

OntologyObo.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyObo.get_ancestors

OntologyObo.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyObo.get_descendants

OntologyObo.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyObo.get_effective_leaves

OntologyObo.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyObo.is_a

OntologyObo.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyObo.is_a_node_id

OntologyObo.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyObo.is_a_node_name

OntologyObo.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyObo.is_node

OntologyObo.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyObo.map_node_suggestion

OntologyObo.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyObo.map_to_leaves

OntologyObo.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – “ids”: IDs of mapped leave nodes “idx”: indicies in leave note list of mapped leave nodes

- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyObo.prepare_maps_to_leaves

OntologyObo.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyObo.reset_root

OntologyObo.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyObo.synonym_node_properties

abstract OntologyObo.**synonym_node_properties**() → List[str]

sfaira.versions.metadata.OntologyObo.validate_node

OntologyObo.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyOboCustom

class sfaira.versions.metadata.**OntologyOboCustom**(*obo: str*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyOboCustom.graph

property OntologyOboCustom.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyOboCustom.leaves

property OntologyOboCustom.**leaves**: List[str]

sfaira.versions.metadata.OntologyOboCustom.n_leaves

property OntologyOboCustom.**n_leaves**: int

sfaira.versions.metadata.OntologyOboCustom.node_ids

property OntologyOboCustom.**node_ids**: List[str]

sfaira.versions.metadata.OntologyOboCustom.node_names

property OntologyOboCustom.**node_names**: List[str]

sfaira.versions.metadata.OntologyOboCustom.nodes

property OntologyOboCustom.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyOboCustom.nodes_dict

property OntologyOboCustom.**nodes_dict**: dict

sfaira.versions.metadata.OntologyOboCustom.synonym_node_properties

property OntologyOboCustom.**synonym_node_properties**: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	

continues on next page

Table 105 – continued from previous page

get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyOboCustom.add_extension

OntologyOboCustom.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyOboCustom.convert_to_id

OntologyOboCustom.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyOboCustom.convert_to_name

OntologyOboCustom.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyOboCustom.get_ancestors

OntologyOboCustom.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyOboCustom.get_descendants

OntologyOboCustom.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyOboCustom.get_effective_leaves

OntologyOboCustom.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyOboCustom.is_a

OntologyOboCustom.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyOboCustom.is_a_node_id

OntologyOboCustom.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyOboCustom.is_a_node_name

OntologyOboCustom.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyOboCustom.is_node

OntologyOboCustom.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyOboCustom.map_node_suggestion

OntologyOboCustom.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyOboCustom.map_to_leaves

OntologyOboCustom.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyOboCustom.prepare_maps_to_leaves

OntologyOboCustom.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyOboCustom.reset_root

OntologyOboCustom.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyOboCustom.validate_node

OntologyOboCustom.**validate_node**(*x: str*)

Onotology-specific classes:

versions.metadata.OntologyCellosaurus([recache])	
versions.metadata.OntologyCl(branch[, ...])	
versions.metadata.OntologyHsapdv(branch[, ...])	
versions.metadata.OntologyMondo(branch[, ...])	
versions.metadata.OntologyMmusdv(branch[, ...])	
versions.metadata.OntologyUberon(branch[, ...])	

sfaira.versions.metadata.OntologyCellosaurus

class sfaira.versions.metadata.**OntologyCellosaurus**(*recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyCellosaurus.graph

property OntologyCellosaurus.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyCellosaurus.leaves

property OntologyCellosaurus.**leaves**: List[str]

sfaira.versions.metadata.OntologyCellosaurus.n_leaves

property OntologyCellosaurus.**n_leaves**: int

sfaira.versions.metadata.OntologyCellosaurus.node_ids

property OntologyCellosaurus.**node_ids**: List[str]

sfaira.versions.metadata.OntologyCellosaurus.node_names

property OntologyCellosaurus.**node_names**: List[str]

sfaira.versions.metadata.OntologyCellosaurus.nodes

property OntologyCellosaurus.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyCellosaurus.nodes_dict

property OntologyCellosaurus.**nodes_dict**: dict

sfaira.versions.metadata.OntologyCellosaurus.synonym_node_properties

property OntologyCellosaurus.**synonym_node_properties**: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	

continues on next page

Table 108 – continued from previous page

get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyCellosaurus.add_extension

OntologyCellosaurus.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyCellosaurus.convert_to_id

OntologyCellosaurus.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyCellosaurus.convert_to_name

OntologyCellosaurus.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyCellosaurus.get_ancestors

OntologyCellosaurus.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyCellosaurus.get_descendants

OntologyCellosaurus.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyCellosaurus.get_effective_leaves

OntologyCellosaurus.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyCellosaurus.is_a

OntologyCellosaurus.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyCellosaurus.is_a_node_id

OntologyCellosaurus.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyCellosaurus.is_a_node_name

OntologyCellosaurus.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyCellosaurus.is_node

OntologyCellosaurus.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyCellosaurus.map_node_suggestion

OntologyCellosaurus.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyCellosaurus.map_to_leaves

OntologyCellosaurus.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyCellosaurus.prepare_maps_to_leaves

OntologyCellosaurus.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyCellosaurus.reset_root

OntologyCellosaurus.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyCellosaurus.validate_node

OntologyCellosaurus.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyCl

class sfaira.versions.metadata.**OntologyCl**(*branch: str*, *use_developmental_relationships: bool = False*, *recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyCl.graph

property OntologyCl.**graph: networkx.classes.multidigraph.MultiDiGraph**

sfaira.versions.metadata.OntologyCl.leaves

property OntologyCl.leaves: List[str]

sfaira.versions.metadata.OntologyCl.n_leaves

property OntologyCl.n_leaves: int

sfaira.versions.metadata.OntologyCl.node_ids

property OntologyCl.node_ids: List[str]

sfaira.versions.metadata.OntologyCl.node_names

property OntologyCl.node_names: List[str]

sfaira.versions.metadata.OntologyCl.nodes

property OntologyCl.nodes: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyCl.nodes_dict

property OntologyCl.nodes_dict: dict

sfaira.versions.metadata.OntologyCl.synonym_node_properties

property OntologyCl.synonym_node_properties: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.

continues on next page

Table 110 – continued from previous page

is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyCl.add_extension

OntologyCl.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyCl.convert_to_id

OntologyCl.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyCl.convert_to_name

OntologyCl.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyCl.get_ancestors

OntologyCl.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyCl.get_descendants

OntologyCl.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyCl.get_effective_leaves

OntologyCl.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyCl.is_a

OntologyCl.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyCl.is_a_node_id

OntologyCl.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyCl.is_a_node_name

OntologyCl.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyCl.is_node

OntologyCl.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyCl.map_node_suggestion

OntologyCl.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyCl.map_to_leaves

OntologyCl.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyCl.prepare_maps_to_leaves

OntologyCl.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyCl.reset_root

OntologyCl.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyCl.validate_node

OntologyCl.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyHsapdv

class sfaira.versions.metadata.**OntologyHsapdv**(*branch: str*, *recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyHsapdv.graph

property OntologyHsapdv.**graph: networkx.classes.multidigraph.MultiDiGraph**

sfaira.versions.metadata.OntologyHsapdv.leaves

property OntologyHsapdv.**leaves: List[str]**

sfaira.versions.metadata.OntologyHsapdv.n_leaves

property OntologyHsapdv.**n_leaves: int**

sfaira.versions.metadata.OntologyHsapdv.node_ids

property OntologyHsapdv.**node_ids: List[str]**

sfaira.versions.metadata.OntologyHsapdv.node_names

property OntologyHsapdv.node_names: List[str]

sfaira.versions.metadata.OntologyHsapdv.nodes

property OntologyHsapdv.nodes: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyHsapdv.nodes_dict

property OntologyHsapdv.nodes_dict: dict

sfaira.versions.metadata.OntologyHsapdv.synonym_node_properties

property OntologyHsapdv.synonym_node_properties: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyHsapdv.add_extension

OntologyHsapdv.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyHsapdv.convert_to_id

OntologyHsapdv.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyHsapdv.convert_to_name

OntologyHsapdv.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyHsapdv.get_ancestors

OntologyHsapdv.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyHsapdv.get_descendants

OntologyHsapdv.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyHsapdv.get_effective_leaves

OntologyHsapdv.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyHsapdv.is_a

OntologyHsapdv.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyHsapdv.is_a_node_id

OntologyHsapdv.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyHsapdv.is_a_node_name

OntologyHsapdv.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyHsapdv.is_node

OntologyHsapdv.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyHsapdv.map_node_suggestion

OntologyHsapdv.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyHsapdv.map_to_leaves

OntologyHsapdv.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – “ids”: IDs of mapped leave nodes “idx”: indicies in leave note list of mapped leave nodes

- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyHsapdv.prepare_maps_to_leaves

OntologyHsapdv.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]
Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyHsapdv.reset_root

OntologyHsapdv.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyHsapdv.validate_node

OntologyHsapdv.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyMondo

class sfaira.versions.metadata.**OntologyMondo**(*branch: str*, *recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyMondo.graph

property OntologyMondo.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyMondo.leaves

property OntologyMondo.**leaves**: List[str]

sfaira.versions.metadata.OntologyMondo.n_leaves

property OntologyMondo.**n_leaves**: int

sfaira.versions.metadata.OntologyMondo.node_ids

property OntologyMondo.**node_ids**: List[str]

sfaira.versions.metadata.OntologyMondo.node_names

property OntologyMondo.**node_names**: List[str]

sfaira.versions.metadata.OntologyMondo.nodes

property OntologyMondo.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyMondo.nodes_dict

property OntologyMondo.**nodes_dict**: dict

sfaira.versions.metadata.OntologyMondo.synonym_node_properties

property OntologyMondo.**synonym_node_properties**: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	

continues on next page

Table 114 – continued from previous page

get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyMondo.add_extension

OntologyMondo.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyMondo.convert_to_id

OntologyMondo.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyMondo.convert_to_name

OntologyMondo.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyMondo.get_ancestors

OntologyMondo.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyMondo.get_descendants

OntologyMondo.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyMondo.get_effective_leaves

OntologyMondo.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyMondo.is_a

OntologyMondo.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyMondo.is_a_node_id

OntologyMondo.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyMondo.is_a_node_name

OntologyMondo.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyMondo.is_node

OntologyMondo.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyMondo.map_node_suggestion

OntologyMondo.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyMondo.map_to_leaves

OntologyMondo.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyMondo.prepare_maps_to_leaves

OntologyMondo.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyMondo.reset_root

OntologyMondo.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyMondo.validate_node

OntologyMondo.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyMmusdv

class sfaira.versions.metadata.**OntologyMmusdv**(*branch: str*, *recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names
nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyMmusdv.graph

property OntologyMmusdv.**graph: networkx.classes.multidigraph.MultiDiGraph**

sfaira.versions.metadata.OntologyMmusdv.leaves

property OntologyMmusdv.**leaves: List[str]**

sfaira.versions.metadata.OntologyMmusdv.n_leaves

property OntologyMmusdv.**n_leaves**: int

sfaira.versions.metadata.OntologyMmusdv.node_ids

property OntologyMmusdv.**node_ids**: List[str]

sfaira.versions.metadata.OntologyMmusdv.node_names

property OntologyMmusdv.**node_names**: List[str]

sfaira.versions.metadata.OntologyMmusdv.nodes

property OntologyMmusdv.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyMmusdv.nodes_dict

property OntologyMmusdv.**nodes_dict**: dict

sfaira.versions.metadata.OntologyMmusdv.synonym_node_properties

property OntologyMmusdv.**synonym_node_properties**: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	

continues on next page

Table 116 – continued from previous page

map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyMmusdv.add_extension

OntologyMmusdv.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyMmusdv.convert_to_id

OntologyMmusdv.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyMmusdv.convert_to_name

OntologyMmusdv.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyMmusdv.get_ancestors

OntologyMmusdv.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyMmusdv.get_descendants

OntologyMmusdv.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyMmusdv.get_effective_leaves

OntologyMmusdv.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyMmusdv.is_a

OntologyMmusdv.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyMmusdv.is_a_node_id

OntologyMmusdv.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyMmusdv.is_a_node_name

OntologyMmusdv.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyMmusdv.is_node

OntologyMmusdv.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyMmusdv.map_node_suggestion

OntologyMmusdv.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyMmusdv.map_to_leaves

OntologyMmusdv.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyMmusdv.prepare_maps_to_leaves

OntologyMmusdv.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyMmusdv.reset_root

OntologyMmusdv.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyMmusdv.validate_node

OntologyMmusdv.**validate_node**(*x: str*)

sfaira.versions.metadata.OntologyUberon

class sfaira.versions.metadata.**OntologyUberon**(*branch: str*, *recache: bool = False*, ***kwargs*)

Attributes

graph
leaves
n_leaves
node_ids
node_names

continues on next page

Table 117 – continued from previous page

nodes
nodes_dict
synonym_node_properties

sfaira.versions.metadata.OntologyUberon.graph

property OntologyUberon.**graph**: networkx.classes.multidigraph.MultiDiGraph

sfaira.versions.metadata.OntologyUberon.leaves

property OntologyUberon.**leaves**: List[str]

sfaira.versions.metadata.OntologyUberon.n_leaves

property OntologyUberon.**n_leaves**: int

sfaira.versions.metadata.OntologyUberon.node_ids

property OntologyUberon.**node_ids**: List[str]

sfaira.versions.metadata.OntologyUberon.node_names

property OntologyUberon.**node_names**: List[str]

sfaira.versions.metadata.OntologyUberon.nodes

property OntologyUberon.**nodes**: List[Tuple[str, dict]]

sfaira.versions.metadata.OntologyUberon.nodes_dict

property OntologyUberon.**nodes_dict**: dict

sfaira.versions.metadata.OntologyUberon.synonym_node_properties

property OntologyUberon.**synonym_node_properties**: List[str]

Methods

add_extension(dict_ontology)	Extend ontology by additional edges and nodes defined in a dictionary.
convert_to_id(x)	
convert_to_name(x)	
get_ancestors(node)	
get_descendants(node)	
get_effective_leaves(x)	Get effective leaves in ontology given set of observed nodes.
is_a(query, reference[, convert_to_id])	Checks if query node is reference node or an ancestor thereof.
is_a_node_id(x)	
is_a_node_name(x)	
is_node(x)	
map_node_suggestion(x[, include_synonyms, ...])	Map free text node name to ontology node names via fuzzy string matching.
map_to_leaves(node[, return_type, include_self])	Map a given node to leave nodes.
prepare_maps_to_leaves([include_self])	Precomputes all maps of nodes to their leave nodes.
reset_root(root)	
validate_node(x)	

sfaira.versions.metadata.OntologyUberon.add_extension

OntologyUberon.**add_extension**(*dict_ontology: Dict[str, List[Dict[str, dict]]]*)

Extend ontology by additional edges and nodes defined in a dictionary.

Checks that DAG is not broken after graph assembly.

Parameters **dict_ontology** – Dictionary of nodes and edges to add to ontology. Parsing:

- keys: parent nodes (which must be in ontology)
- **values: children nodes (which can be in ontology), must be given as a dictionary in which keys are** ontology IDs and values are node values.. If these are in the ontology, an edge is added, otherwise, an edge and the node are added.

Returns

sfaira.versions.metadata.OntologyUberon.convert_to_id

OntologyUberon.**convert_to_id**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyUberon.convert_to_name

OntologyUberon.**convert_to_name**(*x: Union[str, List[str]]*) → Union[str, List[str]]

sfaira.versions.metadata.OntologyUberon.get_ancestors

OntologyUberon.**get_ancestors**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyUberon.get_descendants

OntologyUberon.**get_descendants**(*node: str*) → List[str]

sfaira.versions.metadata.OntologyUberon.get_effective_leaves

OntologyUberon.**get_effective_leaves**(*x: List[str]*) → List[str]

Get effective leaves in ontology given set of observed nodes.

The effective leaves are the minimal set of nodes such that all nodes in x are ancestors of this set, ie the observed nodes which represent leaves of a sub-DAG of the ontology DAG, which captures all observed nodes.

Parameters **x** – Observed node IDs.

Returns Effective leaves.

sfaira.versions.metadata.OntologyUberon.is_a

OntologyUberon.**is_a**(*query: str*, *reference: str*, *convert_to_id: bool = True*) → bool

Checks if query node is reference node or an ancestor thereof.

Parameters

- **query** – Query node name. Node ID or name.
- **reference** – Reference node name. Node ID or name.
- **convert_to_id** – Whether to call self.convert_to_id on `query` and `reference` input arguments

Returns If query node is reference node or an ancestor thereof.

sfaira.versions.metadata.OntologyUberon.is_a_node_id

OntologyUberon.**is_a_node_id**(*x: str*) → bool

sfaira.versions.metadata.OntologyUberon.is_a_node_name

OntologyUberon.**is_a_node_name**(*x: str*) → bool

sfaira.versions.metadata.OntologyUberon.is_node

OntologyUberon.**is_node**(*x: str*)

sfaira.versions.metadata.OntologyUberon.map_node_suggestion

OntologyUberon.**map_node_suggestion**(*x: str*, *include_synonyms: bool = True*, *n_suggest: int = 10*)

Map free text node name to ontology node names via fuzzy string matching.

Parameters

- **x** – Free text node label which is to be matched to ontology nodes.
- **include_synonyms** – Whether to search for meaches in synonyms field of node instances, too.

:return List of proposed matches in ontology.

sfaira.versions.metadata.OntologyUberon.map_to_leaves

OntologyUberon.**map_to_leaves**(*node: str*, *return_type: str = 'ids'*, *include_self: bool = True*) → Union[List[str], numpy.ndarray]

Map a given node to leave nodes.

Parameters

- **node** – Node(s) to map as symbol(s) or ID(s).
- **return_type** – "ids": IDs of mapped leave nodes "idx": indicies in leave note list of mapped leave nodes
- **include_self** – DEPRECEATED.

Returns

sfaira.versions.metadata.OntologyUberon.prepare_maps_to_leaves

OntologyUberon.**prepare_maps_to_leaves**(*include_self: bool = True*) → Dict[str, numpy.ndarray]

Precomputes all maps of nodes to their leave nodes.

Parameters **include_self** – whether to include node itself

Returns Dictionary of index vectors of leave node matches for each node (key).

sfaira.versions.metadata.OntologyUberon.reset_root

OntologyUberon.**reset_root**(*root: str*)

sfaira.versions.metadata.OntologyUberon.validate_node

OntologyUberon.**validate_node**(*x: str*)

Class wrapping cell type ontology for predictor models:

versions.metadata.CelltypeUniverse(cl, ...)	Cell type universe (list) and ontology (hierarchy) container class.

sfaira.versions.metadata.CelltypeUniverse

class sfaira.versions.metadata.**CelltypeUniverse**(*cl: Union[*sfaira.versions.metadata.base.OntologyCl, sfaira.versions.metadata.base.OntologyList*]*, *uberon:* sfaira.versions.metadata.base.OntologyUberon, ***kwargs*)

Cell type universe (list) and ontology (hierarchy) container class.

Basic checks on the organ specific instance are performed in the constructor.

Attributes

onto_cl
onto_uberon

sfaira.versions.metadata.CelltypeUniverse.onto_cl

CelltypeUniverse.**onto_cl**: *sfaira.versions.metadata.base.OntologyCl*

sfaira.versions.metadata.CelltypeUniverse.onto_uberon

CelltypeUniverse.**onto_uberon:** *sfaira.versions.metadata.base.OntologyUberon*

Methods

load_target_universe(fn)	**param fn** .csv file containing target universe.
write_target_universe(fn, x)	**param fn** .csv file containing target universe.

sfaira.versions.metadata.CelltypeUniverse.load_target_universe

CelltypeUniverse.**load_target_universe**(*fn*)

Parameters **fn** – .csv file containing target universe.

Returns

sfaira.versions.metadata.CelltypeUniverse.write_target_universe

CelltypeUniverse.**write_target_universe**(*fn*, *x: List[str]*)

Parameters

- **fn** – .csv file containing target universe.
- **x** – Nodes that make up target universe.

Returns

Topologies

Model topology management.

versions.topologies.TopologyContainer(...[, ...])	Class interface for a YAML-style defined model topology that loads a genome container tailored to the model.

sfaira.versions.topologies.TopologyContainer

class sfaira.versions.topologies.**TopologyContainer**(*topology: dict*, *topology_id: str*, *custom_genome_constainer: Optional[*sfaira.versions.genomes.genomes.GenomeContainer*]* = *None*)

Class interface for a YAML-style defined model topology that loads a genome container tailored to the model.

Attributes

model_type
n_var
organism
output

sfaira.versions.topologies.TopologyContainer.model_type

property TopologyContainer.**model_type**

sfaira.versions.topologies.TopologyContainer.n_var

property TopologyContainer.**n_var**

sfaira.versions.topologies.TopologyContainer.organism

property TopologyContainer.**organism**

sfaira.versions.topologies.TopologyContainer.output

property TopologyContainer.**output**

Methods

User interface: ui

This sub-module gives users access to the model zoo, including model query from remote servers. This API is designed to be used in analysis workflows and does not require any understanding of the way models are defined and stored.

ui.UserInterface([custom_repo, sfaira_repo, ...])	This class performs data set handling and coordinates estimators for the different model types. Example code to obtain a UMAP embedding plot of the embedding created from your data with cell-type labels::.

sfaira.ui.UserInterface

class sfaira.ui.**UserInterface**(*custom_repo: Optional[Union[list, str]] = None*, *sfaira_repo: bool = False*, *cache_path: str = 'cache/'*)

This class performs data set handling and coordinates estimators for the different model types. Example code to obtain a UMAP embedding plot of the embedding created from your data with cell-type labels:

```
import sfaira
import anndata
import scanpy

# initialise your sfaira instance with a model lookuptable.
ui = sfaira.ui.UserInterface(custom_repo="/path/to/local/repo/folder/or/zenodo/repo/
↪URL", sfaira_repo=False)
ui.zoo_embedding.model_id = 'embedding_human-blood-ae-0.2-0.1_theislab'  # pick␣
↪desired model here
ui.zoo_celltype.model_id = 'celltype_human-blood-mlp-0.1.3-0.1_theislab'  # pick␣
↪desired model here
ui.load_data(anndata.read("/path/to/file.h5ad"), gene_symbol_col='index', gene_ens_
↪col='gene_ids')
ui.load_model_embedding()
ui.load_model_celltype()
ui.predict_all()
adata = ui.data.adata
scanpy.pp.neighbors(adata, use_rep="X_sfaira")
scanpy.tl.umap(adata)
scanpy.pl.umap(adata, color="celltypes_sfaira", show=True, save="UMAP_sfaira.png")
```

Attributes

estimator_embedding
estimator_celltype
zoo_embedding
zoo_celltype
data

continues on next page

Table 126 – continued from previous page

model_lookuptable	
adata_ids	

sfaira.ui.UserInterface.estimator_embedding

UserInterface.**estimator_embedding:**
Optional[*sfaira.estimators.keras.base.EstimatorKerasEmbedding*]

sfaira.ui.UserInterface.estimator_celltype

UserInterface.**estimator_celltype:**
Optional[*sfaira.estimators.keras.base.EstimatorKerasCelltype*]

sfaira.ui.UserInterface.zoo_embedding

UserInterface.**zoo_embedding: Optional[sfaira.ui.model_zoo.ModelZoo]**

sfaira.ui.UserInterface.zoo_celltype

UserInterface.**zoo_celltype: Optional[sfaira.ui.model_zoo.ModelZoo]**

sfaira.ui.UserInterface.data

UserInterface.**data: Optional[*sfaira.data.interactive.loader.DatasetInteractive*]**

sfaira.ui.UserInterface.model_lookuptable

UserInterface.**model_lookuptable: Optional[pandas.core.frame.DataFrame]**

sfaira.ui.UserInterface.adata_ids

UserInterface.**adata_ids: sfaira.consts.adata_fields.AdataIds**

Methods

celltype_summary()	Return type with frequencies of predicted cell types.
compute_denoised_expression()	Run local embedding prediction model and add denoised expression to new adata layer.
deposit_zenodo(zenodo_access_token, title, ...)	Deposit all models in model lookup table on Zenodo.
load_data(data[, gene_symbol_col, ...])	Loads the provided AnnData object into sfaira.

continues on next page

Table 127 – continued from previous page

load_model_celltype()	Initialise cell type model and load parameters from public parameter repository.
load_model_embedding()	Initialise embedding model and load parameters from public parameter repository.
predict_all()	Run local cell type prediction and embedding models and add results of both to adata.
predict_celltypes()	Run local cell type prediction model and add predictions to adata.obs.
predict_embedding()	Run local embedding prediction model and add embedding to adata.obsm.
write_lookuptable(repo_path)	**param repo_path**

sfaira.ui.UserInterface.celltype_summary

UserInterface.**celltype_summary**()

Return type with frequencies of predicted cell types.

Returns

sfaira.ui.UserInterface.compute_denoised_expression

UserInterface.**compute_denoised_expression**()

Run local embedding prediction model and add denoised expression to new adata layer.

Returns

sfaira.ui.UserInterface.deposit_zenodo

UserInterface.**deposit_zenodo**(*zenodo_access_token: str*, *title: str*, *authors: list*, *description: str*, *metadata: dict = {}*, *update_existing_deposition: Optional[str] = None*, *publish: bool = False*, *sandbox: bool = False*, *deposit_topologies: bool = True*)

Deposit all models in model lookup table on Zenodo. If publish is set to false, files will be uploaded to a deposition draft, which can be further edited (additional metadata, files etc.). Returns the DOI link if publish=True or a link to the deposition draft if publish=False.

Parameters

- **zenodo_access_token** – Your personal Zenodo API access token. Create one here: https://zenodo.org/account/settings/applications/tokens/new/
- **title** – Title of the Zenodo deposition
- **authors** – List of dicts, where each dict defines one author (dict keys: name: Name of creator in the format "Family name, Given names", affiliation: Affiliation of creator (optional), orcid: ORCID identifier of creator (optional), gnd: GND identifier of creator (optional)
- **description** – Description of the Zenodo deposition.
- **metadata** – Dictionary with further metadata attributes of the deposit. See the Zenodo API refenrece for accepted keys: https://developers.zenodo.org/#representation

- **update_existing_deposition** – If None, a new deposition will be created.

If an existing deposition ID is provided as a sting, than this deposition will be updated with a new version. :param publish: Set this to True to directly publish the weights on Zenodo.

When set to False a draft will be created, which can be edited in the browser before publishing.

Parameters **sandbox** – If True, use the Zenodo testing platform at https://sandbox.zenodo.org for your deposition. We recommend testing your upload with sandbox first as depositions cannot be deleted from the main Zenodo platform once created. :param deposit_topologies: If true, an associated topology file for every weights file will be uploaded to zenodo. The naming format for the topology files is <model_id>_topology.pickle

sfaira.ui.UserInterface.load_data

UserInterface.**load_data**(*data: anndata._core.anndata.AnnData*, *gene_symbol_col: Optional[str] = None*, *gene_ens_col: Optional[str] = None*, *obs_key_celltypes: Optional[str] = None*)

Loads the provided AnnData object into sfaira.

If genes in the provided AnnData object are annotated as gene symbols, please provide the name of the corresponding var column (or 'index') through the gene_symbol_col argument.

If genes in the provided AnnData object are annotated as ensembl ids, please provide the name of the corresponding var column (or 'index') through the gene_ens_col argument.

You need to provide at least one of the two. :param data: AnnData object to load :param gene_symbol_col: Var column name (or 'index') which contains gene symbols :param gene_ens_col: ar column name (or 'index') which contains ensembl ids :param obs_key_celltypes: .obs column name which contains cell type labels.

sfaira.ui.UserInterface.load_model_celltype

UserInterface.**load_model_celltype**()

Initialise cell type model and load parameters from public parameter repository.

Loads model defined in self.model_id.

Returns Model ID loaded.

sfaira.ui.UserInterface.load_model_embedding

UserInterface.**load_model_embedding**()

Initialise embedding model and load parameters from public parameter repository.

Loads model defined in self.model_id.

Returns Model ID loaded.

sfaira.ui.UserInterface.predict_all

UserInterface.**predict_all**()
Run local cell type prediction and embedding models and add results of both to adata.

Returns

sfaira.ui.UserInterface.predict_celltypes

UserInterface.**predict_celltypes**()
Run local cell type prediction model and add predictions to adata.obs.

Returns

sfaira.ui.UserInterface.predict_embedding

UserInterface.**predict_embedding**()
Run local embedding prediction model and add embedding to adata.obsm.

Returns

sfaira.ui.UserInterface.write_lookuptable

UserInterface.**write_lookuptable**(*repo_path: str*)

Parameters **repo_path** –

Returns

2.2.3 Commandline interface

sfaira

Create and manage sfaira dataloaders.

```
sfaira [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

-v, --verbose
Enable verbose output (print debug statements).

-l, --log-file <log_file>
Save a verbose log to a file.

annotate-dataloader

Annotates a dataloader.

```
sfaira annotate-dataloader [OPTIONS]
```

Options

--doi <doi>
Required The doi of the paper that the data loader refers to.

--path-data <path_data>
Absolute path of the location of the raw data directory.

--path-loader <path_loader>
Relative path from the current directory to the location of the data loader.

--schema <schema>
The curation schema to check meta data availability for.

cache-clear

Clears sfaira cache, including ontology and genome cache.

```
sfaira cache-clear [OPTIONS]
```

cache-reload

Downloads new ontology versions into cache.

```
sfaira cache-reload [OPTIONS]
```

create-dataloader

Interactively create a new sfaira dataloader.

```
sfaira create-dataloader [OPTIONS]
```

Options

--path-data <path_data>
Absolute path of the desired location of the raw data directory.

--path-loader <path_loader>
Relative path from the current directory to the desired location of the data loader.

export-h5ad

Creates a collection of streamlined h5ad object for a given DOI.

```
sfaira export-h5ad [OPTIONS]
```

Options

--doi <doi>
Required The doi of the paper that the data loader refers to.

--schema <schema>
Schema to streamline to, e.g. 'cellxgene'

--path-out <path_out>
Absolute path of the location of the streamlined output h5ads.

--path-data <path_data>
Absolute path of the location of the raw data directory.

--path-loader <path_loader>
Relative path from the current directory to the location of the data loader.

--path-cache <path_cache>
The optional absolute path to cached data library maintained by sfaira. Using such a cache speeds up loading in sequential runs but is not necessary.

finalize-dataloader

Formats .tsvs and runs a full data loader test.

```
sfaira finalize-dataloader [OPTIONS]
```

Options

--doi <doi>
Required The doi of the paper that the data loader refers to.

--path-data <path_data>
Absolute path of the location of the raw data directory.

--path-loader <path_loader>
Relative path from the current directory to the location of the data loader.

--schema <schema>
The curation schema to check meta data availability for.

publish-dataloader

Interactively create a GitHub pull request for a newly created data loader. This only works when called in the sfaira CLI docker container. Runs a full data loader test before starting the pull request.

```
sfaira publish-dataloader [OPTIONS]
```

test-dataloader

Runs a full data loader test.

```
sfaira test-dataloader [OPTIONS]
```

Options

--doi <doi>
Required The doi of the paper that the data loader refers to.

--path-data <path_data>
Absolute path of the location of the raw data directory.

--path-loader <path_loader>
Relative path from the current directory to the location of the data loader.

--schema <schema>
The curation schema to check meta data availability for.

validate-dataloader

Verifies the dataloader against sfaira's requirements.

```
sfaira validate-dataloader [OPTIONS]
```

Options

--doi <doi>
Required The doi of the paper that the data loader refers to.

--path-loader <path_loader>
Relative path from the current directory to the desired location of the data loader.

--schema <schema>
The curation schema to check meta data availability for.

validate-h5ad

Runs a component test on a streamlined h5ad object.

h5ad is the absolute path of the .h5ad file to test. schema is the schema type ("cellxgene",) to test.

```
sfaira validate-h5ad [OPTIONS]
```

Options

--h5ad <h5ad>

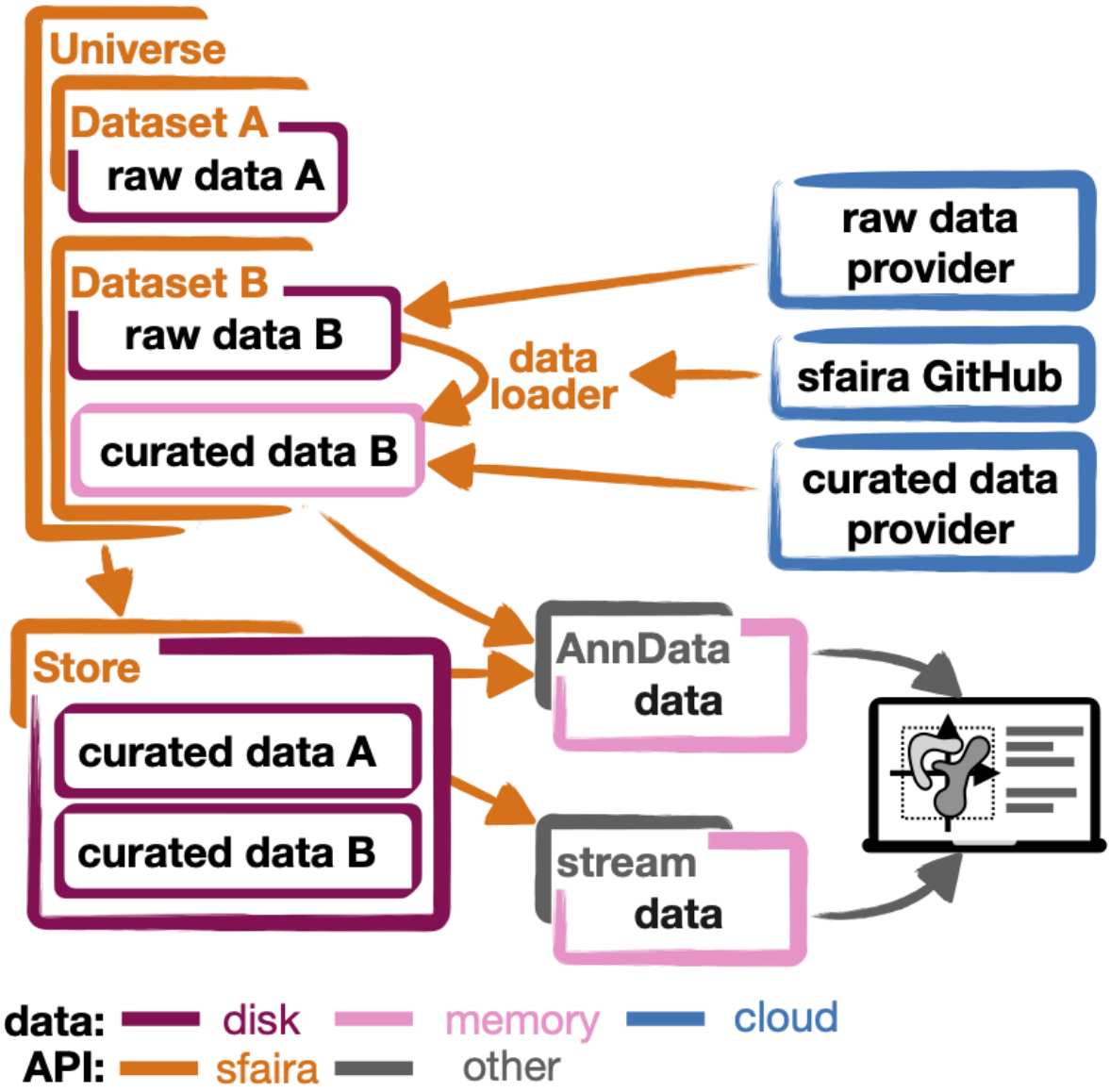
--schema <schema>

2.2.4 Tutorials

We provide multiple tutorials in separate repository.

- A tutorial for interacting with the data loaders via the `Universe` class (universe).
- A tutorial for general usage of the user interface (user_interface).
- A tutorial for zero-shot analysis with the user interface (pbmc3k).
- A tutorial for creating meta data-based data zoo overview figure (meta_data)

2.2.5 The data life cycle



The life cycle of a single-cell count matrix often looks as follows:

1. **Generation** from primary read data in a read alignment pipeline.
2. **Annotation** with cell types and sample meta data.
3. **Publication** of annotated data, often together with a manuscript.
4. **Curation** of this public data set for the purpose of a meta study. In a python workflow, this curation step could be a scanpy script based on data from step 3, for example.
5. **Usage** of data curated specifically for the use case at hand, for example for a targeted analysis or a training of a machine learning model.

where step 1-3 is often only performed once by the original authors of the data set, while step 4 and 5 are repeated multiple times in the community for different meta studies. Sfaira offers the following functionality groups that accelerate

steps along this pipeline:

Sfaira tools across life cycle

I) Data loaders

We maintain streamlined data loader code that improve **Curation** (step 4) and make this step sharable and iteratively improvable. Read more in our guide to data contribution *Writing data loaders*.

II) Dataset, DatasetGroup, DatasetSuperGroup

Using the data loaders from (I), we built an interface that can flexibly download, subset and curate data sets from the sfaira data zoo, thus improving **Usage** (step 5). This interface can yield adata instances to be used in a scanpy pipeline, for example. Read more in our guide to data consumption *Using data loaders*.

III) Stores

Using the streamlined data set collections from (II), we built a computationally efficient data interface for machine learning on such large distributed data set collection, thus improving **Usage** (step 5): Specifically, this interface is optimised for out-of-core observation-centric indexing in scenarios that are typical to machine learning on single-cell data. Read more in our guide to data stores *Data stores*.

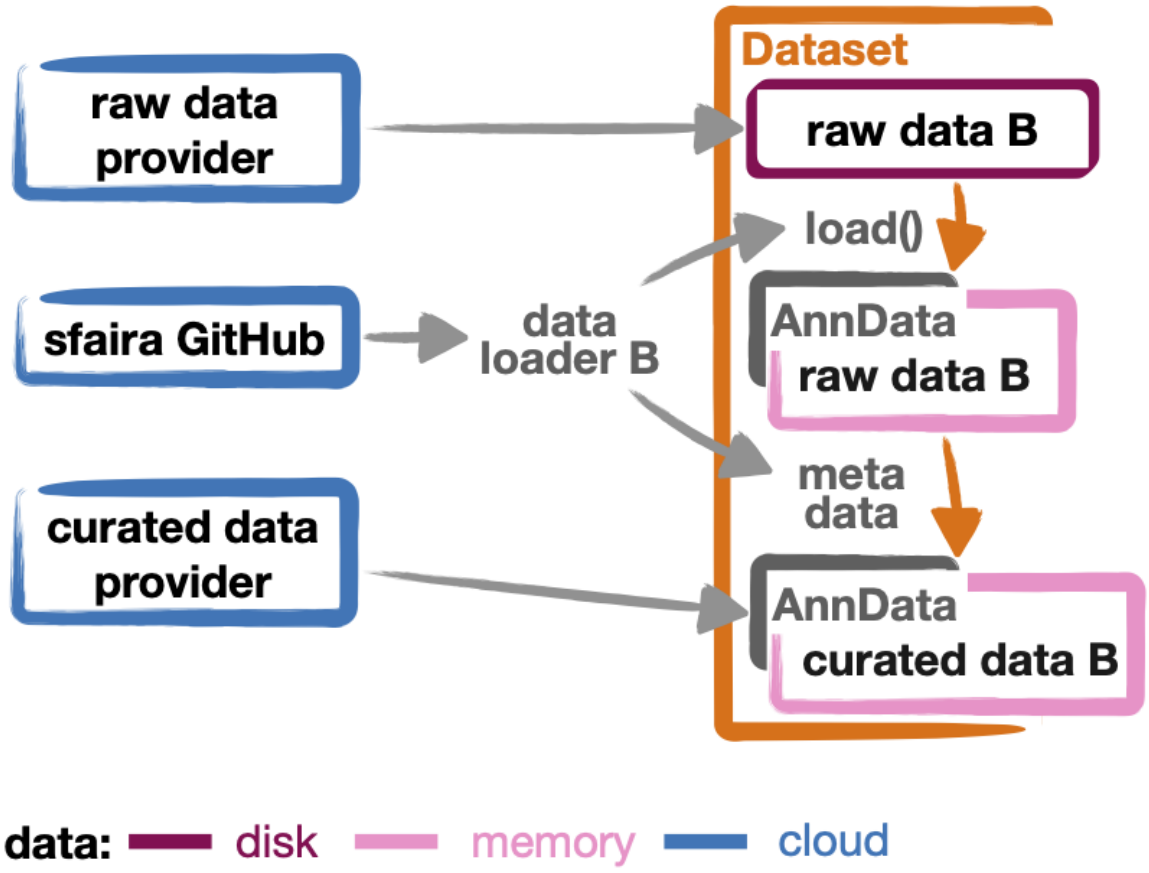
FAIR data

FAIR data is a set of data management guidelines that are designed to improve data reuse and automated access (see also the original publication of FAIR for more details). The key data management topics addressed by FAIR are findability, accessibility, interoperability and reusability. Single-cell data sets are usually public and also adhere to varying degrees to FAIR principles. We designed sfaira so that it improves FAIR attributes of published data sets beyond their state at publication. Specifically, sfaira:

- improves **findability** of data sets by serving data sets through complex meta data query.
- improves **accessibility** of data sets by serving streamlined data sets.
- improves **interoperability** of data sets by streamlining data using versioned meta data ontologies.
- improves **reusability** of data sets by allowing for iterative improvements of meta data annotation and by shipping usage critical meta data.

2.2.6 Writing data loaders

For a high-level overview of data management in sfaira, read *The data life cycle* first. In brief, a data loader is a set of instructions that allows for streamlining of raw count matrices and meta data into objects of a target format. Here, streamlining means that gene names are controlled based on a genome assembly, metadata items are constrained to follow ontologies, and key study metadata are described. This streamlining increases accessibility and visibility of a dataset and to makes it available to a large audience. In sfaira, data loaders are grouped by scientific study (DOI of a preprint or DOI of a publication). A data loader for a study is a directory named after the DOI of the study that contains code and text files. This directory is part of the sfaira python package and, thus, maintained on GitHub. This allows for data loaders to be maintained via GitHub workflows: contribution and fixes via pull requests and deployment via repository cloning and package installation.



A dataloader consists of four file components within a single directory:

1. `__init__.py` file which is has same content in all loaders,
2. `ID.py` file that contains a `load()` functions with based instructions of loading raw data on disk,
3. `ID.yaml` file that describes most meta data,
4. `ID*.tsv` files with ontology-wise maps of free-text metadata items to constrained vocabulary.

All dataset-specific components receive an `ID` that is set during the curation process. Below, we describe how multiple datasets within a study can be handled with the same dataloder. In cases where this is not efficient, one can go through the data loader creation process once for each dataset and then group the resulting loaders (file groups 1-4) in a single directory named after the study's DOI.

An experienced curator can directly write such a data loader. However, first-time contributors often struggle with the interplay of individual files, metadata maps from free-text annotation are notoriously buggy and comprehensive testing is important also for contributions by experienced curators. Therefore, we broke the process of writing a loader down into phases and built a CLI to guide users through this process. Each phase corresponds to one command (one execution of a shell command) in the CLI. In addition, the CLI guides the user through manual steps that are necessary in each phase. We structured the process of curation into four phases, a preparatory phase P precedes CLI execution and is described in this documentation.

0. Phase P (`prepare`): data and python environment setup for curation.
1. Phase 1 (`create`): a `load()` function (in a `.py`) and a YAML are written.
2. Phase 2 (`annotate`): ontology-specific maps of free-text metadata to contrained vocabulary (in `*.tsv`) are written.

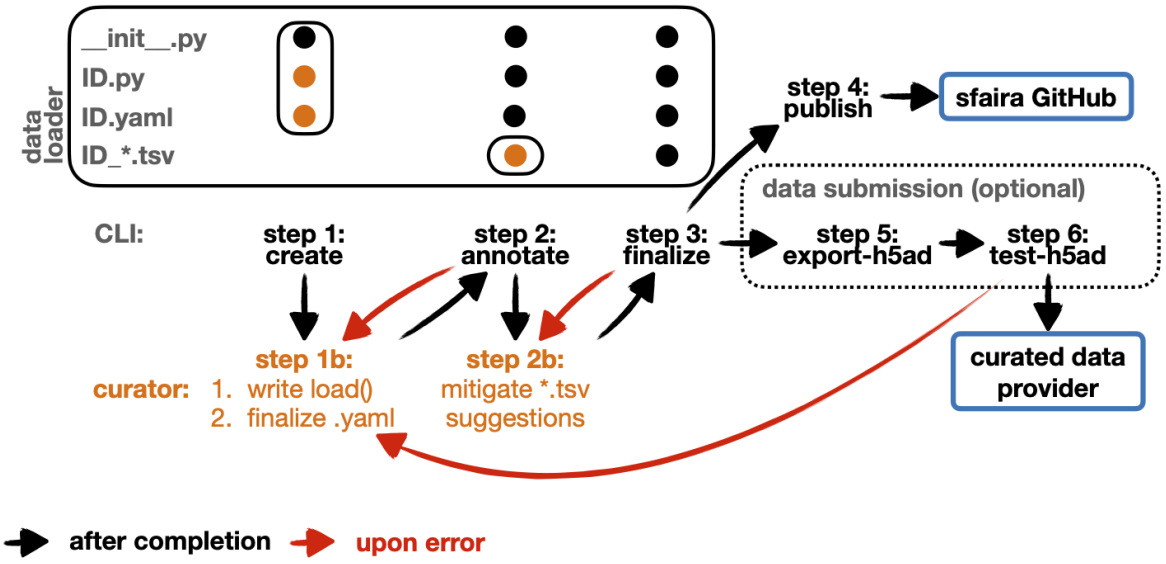
3. Phase 3 (`finalize`): the data loader is tested and metadata are cleaned up.
4. Phase 4 (`publish`): the data loader is uploaded to the sfaira GitHub repository.

An experienced curator could skip using the CLI for phase 1 and write the `__init__.py`, `ID.py` and `ID.yaml` by hand. In this case, we still highly recommend using the CLI for phase 2 and 3. Note that phase 2 is only necessary if you have free-text metadata that needs to be mapped, the CLI will point this out accordingly in phase 1. This 4-phase cycle completes initial curation and results in data loader code that can be pushed to the sfaira GitHub repository. You have the choice between using a docker image or a sfaira installation (e.g. in conda) for phase P-4. The workflow is more restricted but safer in conda, we recommend docker if you are inexperienced with software development with conda, git and GitHub. Where appropriate, separate instruction options are given for workflows in conda and docker below. Overall, the workflow looks the same in both frameworks, though. This cycle can be complemented by an optional workflow to cache curated `.h5ad` objects (e.g. on the cellxgene website):

5. Phase 5 (`export-h5ad`): the data loader is used to create a streamlined `.h5ad` of a particular format.
6. Phase 6 (`validate-h5ad`): the `.h5ad` from phase 4 is checked for compliance with a particular (e.g. the cellxgene format).

The resulting `.h5ad` can be shared with collaborators or uploaded to data submission servers.

Create a new data loader



Phase P: Preparation

Before you start writing the data loader, we recommend completing this checks and preparation measures. Phase P is sub-structured into sub-phases:

Pa. Name the data loader. We will decide for a name of the dataloader based on its DOI. Prefix the DOI with "d" and replace the special characters in the DOI with "_" here to prevent copy mistakes, e.g. the DOI `10.1000/j.journal.2021.01.001` becomes `d10_1000_j_journal_2021_01_001` Remember to replace this DOI with the DOI of the study you want to contribute, choose a publication (journal) DOI if available, otherwise a preprint DOI. If neither DOI is available, because this is unpublished data, for example, use an identifier that makes sense to you, that is prefixed with `dno_doi` and contains a name of an author of the dataset, e.g. `dno_doi_einstein_brain_atlas`. We will refer to this name as `DOI-name` and it will be used to label the contributed code and the stored data.

Pb. Check that the data loader was not already implemented. We will open issues for all planned data loaders, so you can search both the code base and our GitHub issues for matching data loaders before you start writing one. You can also search for GEO IDs if our code base as they are included in the data URL that is annotated in the data loader. The core data loader identified is the directory compatible doi, which is the doi with all special characters replaced by "_" and a "d" prefix is used: "10.1016/j.cell.2019.06.029" becomes "d10_1016_j_cell_2019_06_029". Searching for this string should yield a match if it is already implemented, take care to look for both preprint and publication DOIs if both are available. We will also mention publication names in issues, you will however not find these in the code.

Pc. Prepare an installation of sfaira to use for data loader writing. Instead of working in your own sfaira installation, you can download the sfaira data curation docker container instead of going through any of the steps here.

Pc-docker.

1. Install docker (and start Docker Desktop if you're on Mac or Windows).
2. **Pull the latest version of the sfaira cli container.**

```
sudo docker pull leanderd/sfaira-cli:latest
```

3. Run the sfaira CLI within the docker image. Please replace <path_data> and <path_loader> with paths to two empty directories on your machine. The sfaira CLI will use these to read your datafiles from and write the dataloaders to respectively.

```
PATH_DATA=<path_data>
PATH_LOADER=<path_loader>
sudo docker run --rm -it -v ${PATH_DATA}:/root/sfaira_data -v $
↪{PATH_LOADER}:/root/sfaira_loader leanderd/sfaira-cli:latest
```

Pc-conda. Jump to step 4 if you do not require explanations of specific parts of the shell script.

1. **Install sfaira.** Clone sfaira into a local repository `DIR_SFAIRA`.

```
cd DIR_SFAIRA
git clone https://github.com/theislab/sfaira.git
cd sfaira
git checkout dev
```

2. **Prepare a local branch of sfaira dedicated to your loader.** You can name this branch after the `DOI-name`, prefix this branch with `data/` as the code change suggested is a data addition.

```
cd DIR_SFAIRA
cd sfaira
git checkout dev
git pull
git checkout -b data/DOI-name
```

3. **Install sfaira into a conda environment.** You can for example use pip inside of a conda environment dedicated to data curation.

```
cd DIR_SFAIRA
cd sfaira
git checkout -b data/DOI-name
conda create -n sfaira_loader
conda install -n sfaira_loader python=3.8
conda activate sfaira_loader
pip install -e .
```

4. **Summary of step 1-3.** Pc1-3 are all covered by the following code block. Remember to name the git branch after your DOI:

```
cd DIR_SFAIRA
git clone https://github.com/theislab/sfaira.git
cd sfaira
git checkout dev
git pull
git checkout -b data/DOI-name
conda create -n sfaira_loader
conda install -n sfaira_loader python=3.8
conda activate sfaira_loader
pip install -e .
```

Pd. Download the raw data into a local directory. You will need to set a path in which the data files can be accessed by sfaira, in the following referred to as `<path_data>/<DOI-name>/`. Identify the raw data files and copy them into the datafolder `<path_data>/<DOI-name>/`. Note that this should be the exact files that are downloadable from the download URL you provided in the dataloader: Do not decompress these files if these files are archives such as zip, tar or gz. In some cases, multiple processing forms of the raw data are available, some times even on different websites. Follow these rules to disambiguate the data source for the data loader:

- **Rule 1: Prefer unprocessed gene expression count data over normalised data.** Often it makes sense to provide author-normalised data in a curated object in addition to count data.
- **Rule 2: Prefer dedicated data archives over websites that may be temporary** Examples of archives include EGA, GEO, zenodo, potentially temporary websites may be institute websites, cloud files linked to a person's account.

Note that it may in exception cases make sense to collect count data and cell-wise meta data from different locations, or similar, collect normalised and count matrices from different locations. You can supply multiple data URLs below, so collect all relevant files in this phase.

Pe. Get an overview of the published data. Data curation is much easier if you have an idea of what the data that you are curating looks like before you start. Especially, you will notice a difference in your ability to fully leverage phase 1a if you prepare here. We recommend you load the cell-wise and gene-wise meta in a python session and explore the type of meta data provided there. You will receive further guidance throughout the curation process here, but we recommend that you try locate the following meta data items now already if they are annotated in the data set and if they are shared across the dataset or specific to a feature or observation, where the latter usually corresponds to a column in `.obs` or `.var` of a published `.h5ad`, or to a corresponding column in a tabular file:

- single-cell assay
- cell type
- developmental stage
- disease state
- ethnicity (only relevant for human samples)
- organ / tissue
- organism
- sex

Note that these are also the key ontology-restricted and required meta data in the cellxgene curation schema. Next, we recommend you briefly consider the available features: Are count matrices, processed matrices or spliced/unspliced RNA published? Which gene identifiers are used (symbols or ENSEMBL IDs)? Which non-RNA modalities are present in the data?

Phase 1: create

This phase creates a skeleton for a data loader: `__init__.py`, `.py` and `.yaml` files. Phase 1 is sub-structured into 2 sub-phases:

- 1a: Create template files (`sfaira create-dataloader`).
- 1b: Completion of created files (manual).

1a. Create template files. When creating a dataloader with `sfaira create-dataloader` dataloader specific attributes such as organ, organism and many more are prompted for. We provide a description of all meta data items at the bottom of this page, note that these metadata underly specific formattig and ontology constraints described below. If the requested information is not available simply hit enter to skip the entry. Note that some meta data items are always defined per data set, e.g. a DOI, whereas other meta data items may or may not be the same for all cells in a data set. For example, an entire organ may belong to one disease condition or one organ, or may consist of a pool of multiple samples that cover multiple values of the given metadata item. The questionaire and YAML are set up to guide you through finding the best fit. Note that annotating dataset-wide is preferable where possible as it results in briefer curation code. The CLI decides on an `ID` of this dataset within the loader that you are writing, this will be used to label all files associated with the current dataset. The CLI tells you how to continue from here, phase 1b) is always necessary, phase 2) is case-dependent and mistakes in naming the data folder in phase Pd) are flagged here. As indicated at appropriate places by the CLI, some meta data are ontology constrained. You should input symbols, ie. readable words and not IDs in these places. For example, the `.yaml` entry `organ` could be "lung", which is a symbol in the UBERON ontology, whereas `organ_obs_key` could be any string pointing to a column in the `.obs` in the `anndata` instance that is output by `load()`, where the elements of the column are then mapped to UBERON terms in phase 2.

1a-docker. You can run the `create-dataloader` command directly.

```
sfaira create-dataloader
```

1a-conda. In the following command, replace `DATA_DIR` with the path `<path_data>/` you used above. You can optionally supply `--path-loader` to `create-dataloader` to change the location of the created data loader to an arbitrary directory other than the internal collection of sfaira in `./sfaira/data/dataloaders/loaders/`. Note: Use the default location if you want to commit and push changes from this sfaira clone.

```
sfaira create-dataloader --path-data DATA_DIR
```

1b. Manual completion of created files (manual).

1. **Correct the `.yaml` file.** Correct errors in `<path_loader>/<DOI-name>/ID.yaml` file and add further attributes you may have forgotten in step 2. See sec-multiple-files for short-cuts if you have multiple data sets. This step is can be skipped if there are the `.yaml` is complete after phase 1a). Note on lists and dictionaries in the yaml file format: Some times, you need to write a list in yaml, e.g. because you have multiple data URLs. A list looks as follows:

```
# Single URL:
download_url_data: "URL1"
# Two URLs:
download_url_data:
    - "URL1"
    - "URL2"
```

As suggested in this example, do not use lists of length 1. In contrast, you may need to map a specific `sample_fns` to a meta data in multi file loaders:

```
sample_fns:
    - "FN1"
    - "FN2"
[...]
assay_sc:
    FN1: 10x 3' v2
    FN2: 10x 3' v3
```

Take particular care with the usage of quotes and ":" when using maps as outlined in this example.

2. **Complete the load function.** Complete the `load()` function in `<path_loader>/<DOI-name>/ID.py`. If you need to read compressed files directly from python, consider our guide reading-compressed-files. If you need to read R files directly from python, consider our guide reading-r-files.

Phase 2: annotate

This phase creates annotation map files: `.tsv`. The metadata items that require annotation maps all non-empty entries that end on `*obs_key` under `dataset_or_observation_wise` in the `.yaml` which are subject to an ontology field-descriptions:. One file is created per such metadata `ITEM`, the corresponding file is `<path_loader>/<DOI-name>/<ID>_<ITEM>.tsv` This means that a variable number of such files is created and dependending on the scenario, even no such files may be necessary: Phase 2 can be entirely skipped if no annotation maps are necessary, this is indicated by the CLI at the end of phase 1a. Phase 2 is sub-structured into 2 sub-phases:

- 2a: Create metadata annotation files (`sfaira annotate-dataloader`).
- 2b: Completion of annotation (manual).

2a. Create metadata annotation files (`sfaira annotate-dataloader`). This creates `<path_loader>/<DOI-name>/ID*.tsv` files with meta data map suggestions for each meta data item that requires such maps. Note: You can identify the loader via `--doi` with the main DOI (ie. journal > preprint if both are defined) or with the DOI-based data loader name defined by sfaira, ie. `<DOI-name>` in `<path_loader>/<DOI-name>`, which is either `d10_*` or `dno_doi_*`.

2a-docker. In the following command, replace `DOI` with the DOI of your data loader.

```
sfaira annotate-dataloader --doi DOI
```

2a-conda. In the following command, replace `DATA_DIR` with the path `<path_data>/` you used above and replace `DOI` with the DOI of your data loader. You can optionally supply `--path-loader` to `create-dataloader` if the data loader is not in the internal collection of sfaira in `./sfaira/data/dataloaders/loaders/`.

```
sfaira annotate-dataloader --doi DOI --path-data DATA_DIR
```

2b. Completion of annotation (manual). Each `<path_loader>/<DOI-name>/ID*.tsv` files contains two columns with one row for each unique free-text meta data item, e.g. each cell type label. One file is created for each `*_obs_key` that requires mapping to an ontology, which are: `assay_sc_obs_key`, `cell_line_sc_obs_key`, `cell_type_sc_obs_key`, `development_stage_sc_obs_key`, `disease_sc_obs_key`, `ethnicity_sc_obs_key`, `organ_sc_obs_key`, `organism_sc_obs_key`, `sex_sc_obs_key`. Depending on the number of such `*_obs_key` items that are set in the `.yaml`, you will have between 0 amd 9 `.tsv` files.

- **"source":** The first column is labeled "source" and contains free-text identifiers.
- **"target":** The second column is labeled "target" and contains suggestions for matching the symbols from the corresponding ontology.

The suggestions are based on multiple search criteria, mostly on similarity of the free-text token to tokes in the ontology. Suggested tokens are separated by ":" in the target column, for each token, the same number of suggestions is supplied. We use different search strategies on each token and separate the output by strategy by ":||:". You might notice that one strategy works well for a particular `ID*.tsv` and focus your attention on that group. It is now up to you to manually mitigate the suggestions in the "target" column of each `.tsv` file, for example in a text editor. Depending on the ontology and on the accuracy of the free-text annotation, these suggestions may be more or less helpful. The worst case is that you need to go to search engine of the ontology at hand for each entry to check for matches. The best case is that you know the ontology well enough to choose from the suggestions, assuming that the best match is in the suggestions. Reality lies somewhere in the middle of the two, do not be too conservative with looking items up online. We suggest to use the ontology search engine on the OLS web-interface for your manual queries. For each meta data item, the correspond ontology is listed in the detailed meta data description field-descriptions. Make sure to read our notes on cell type curation celltype-annotation.

Note 1: If you compare these `ID*.tsv` to `tsv` files from published data loaders, you will notice that published ones contain a third column. This column is automatically added in phase 3 if the second column was correctly filled here.

Note 2: The two columns in the `ID*.tsv` are separated by a tab-separator ("\t"), make sure to not accidentally delete this token. If you accidentally replace it with `" "`, you will receive errors in phase 3, so do a visual check after finishing your work on each `ID*.tsv` file.

Note 3: Perfect matches are filled wihtout further suggestions, you can often directly leave these rows as they are after a brief sanity check.

Phase 3: finalize

3a. Clean and test data loader. This command will test data loading and will format the metadata maps in `ID*.tsv` files from phase 2b). If this command passes without further change requests, the data loader is finished and ready for phase 4. Note: You can identify the loader via `--doi` with the main DOI (ie. journal > preprint if both are defined) or with the DOI-based data loader name defined by sfaira, ie. `<DOI-name>` in `<path_loader>/<DOI-name>`, which is either `d10_*` or `dno_doi_*`.

3a-docker. In the following command, replace `DOI` with the DOI of your data loader.

```
sfaira finalize-dataloader --doi DOI
```

3a-conda. In the following command, replace `DATA_DIR` with the path `<path_data>/` you used above and replace `DOI` with the DOI of your data loader. You can optionally supply `--path-loader` to `create-dataloader` if the data loader is not in the internal collection of sfaira in `./sfaira/data/dataloaders/loaders/`. Once this command passes, it will give you a message you can use in phase 4 to document this test on the pull request.

```
sfaira finalize-dataloader --doi DOI --path-data DATA_DIR
```

Phase 4: publish

You will need to authenticate with GitHub during this phase. You can push the code from with the sfaira docker with a single command or you can use `git` directly:

4a. Push data loader to the public sfaira repository. You will test the loader one last time, this test will not throw errors if you have not introduced changes since phase 3. Note: You can identify the loader via `--doi` with the main DOI (ie. journal > preprint if both are defined) or with the DOI-based data loader name defined by sfaira, ie. <DOI-name> in <path_loader>/<DOI-name>, which is either `d10_*` or `dno_doi_*`.

4a-docker. If you are writing a data loader from within the sfaira data curation docker, you can run phase 4 with a single command. In the following command, replace `DOI` with the DOI of your data loader.

```
sfaira test-dataloader --doi DOI
sfaira publish-dataloader
```

You will be prompted to paste your github token in order to authenticate with github. If you do not have a token you can leave the field blank and you will be interactively guided to authenticating your github account using your browser. (You will have to manually copy a url into the browser at some point.) In certain cases you might be prompted to enter you github username and password again during the process. Please note that this requires you to enter you username and github token as before, not the password you use to log into github.com in your browser.

You will also be prompted the following by the CLI: `Where should we push the xxx branch?` You generally want to select the second option here ("Create a fork of theislab/sfaira") unless you are a member of the theislab organisation or otherwise have previously obtained write access to the sfaira repository. In this case you can select the first option ("theislab/sfaira").

4a-git. You can contribute the data loader to public sfaira as code through a pull request. Note that you can also just keep the data loader in your local installation if you do not want to make it public. In the following command, replace `DATA_DIR` with the path <path_data>/ you used above and replace `DOI` with the DOI of your data loader. If you have not modified any aspects of the data loader since phase 3, you can skip `sfaira test-dataloader` below. In order to create a pullrequest you first need to fork the sfaira repository on GitHub. Once forked, you can use the code shown below to submit your new dataloader. Note: the CLI will ask you to copy a data loader testing summary into the pull request at the end of the output generated by `finalize-dataloader`.

```
sfaira test-dataloader --doi DOI --path-data DATA_DIR
cd DIR_SFAIRA
cd sfaira
git remote set-url origin https://github.com/<user>/sfaira.git  # Replace
↪<user> with your github username.
git checkout dev
git add *
git commit -m "Completed data loader."
git push
```

After successfully pushing the new dataloader to your fork, you can go to github.com and create a pull-request from your fork to the dev branch of the original sfaira repo. Please include the doi of your added dataset in the PR title

Phase 5: export-h5ad

Phase 5 and 6 are optional, see also introduction paragraphs on this documentation page.

5a. Export `.h5ads`'s. Write streamlined dataset(s) corresponding to data loader into (an) `.h5ad` file(s) according to a specific set of rules (a schema, e.g. "cellxgene"). Note: You can identify the loader via `--doi` with the main DOI (ie. journal > preprint if both are defined) or with the DOI-based data loader name defined by sfaira, ie. <DOI-name> in <path_loader>/<DOI-name>, which is either `d10_*` or `dno_doi_*`.

5a-docker. In the following command, replace `DOI` with the DOI of your data loader, replace `SCHEMA` with the target data schema. You can find the resulting h5ad file in the `sfaira_data` directory you specified when starting the container. .. code-block:

```
sfaira export-h5ad --doi DOI --schema SCHEMA --path-out /root/sfaira_data/
```

5a-conda. In the following command, replace `DATA_DIR` with the path <path_data>/ you used above, replace `DOI` with the DOI of your data loader, replace `SCHEMA` with the target data schema, and replace `OUT_DIR` with the directory to which the objects are written to. You can optionally supply `--path-loader` to `create-dataloader` if the data loader is not in the internal collection of sfaira in `./sfaira/data/dataloaders/loaders/`. .. code-block:

```
sfaira export-h5ad --doi DOI --path-data DATA_DIR --schema SCHEMA --path-out␣
↪OUT_DIR
```

Phase 6: validate-h5ad

Phase 5 and 6 are optional, see also introduction paragraphs on this documentation page.

6a. Validate format of `.h5ad` The streamlined `.h5ad` files from phase 5 are validated according to a specific set of rules (a schema).

6a-docker. In the following command, replace FN with the file name of the `.h5ad` file to test (just the filename, not the full path here), and replace `SCHEMA` with the target data schema. The h5ad file must be placed in the `sfaira_data` directory you specified when starting the container.

```
sfaira validate-h5ad --h5ad /root/sfaira_data/FN --schema SCHEMA
```

6a-conda. In the following command, replace FN with ful path of the `.h5ad` file to test, and replace `SCHEMA` with the target data schema. .. code-block:

```
sfaira validate-h5ad --h5ad FN --schema SCHEMA
```

Advanced topics

Loading multiple files of similar structure

Only one loader has to be written for each set of files that are similarly structured which belong to one DOI. `sample_fns` in `dataset_structure` in the `.yaml` indicates the presence of these files. The identifiers listed there do not have to be the full file names. They are received by `load()` as the argument `sample_fn` and can then be used in custom code in `load()` to load the correct file. This allows sharing code across these files in `load()`. If these files share all meta data in the `.yaml`, you do not have to change anything else here. If a some meta data items are file specific, you can further subdefine them under the keys in this `.yaml` via their identifiers stated here. In the following example, we show how this formalism can be used to identify one file declared as "A" as a healthy lung sample and another file "B" as a healthy pancreas sample.

```
dataset_structure:
    dataset_index: 1
    sample_fns:
        - "A"
        - "B"
dataset_wise:
    # ... part of yaml omitted ...
dataset_or_observation_wise:
    # ... part of yaml omitted
    healthy: True
    healthy_obs_key:
    individual:
    individual_obs_key:
    organ:
        A: "lung"
        B: "pancreas"
    organ_obs_key:
    # part of yaml omitted ...
```

Note that not all meta data items have to subdefined into "A" and "B" but only the ones with differing values! The corresponding `load` function would be:

```
def load(data_dir, sample_fn, fn=None) -> anndata.AnnData:
    # The following reads either my_file_A.h5ad or my_file_B.h5ad which correspond to A␣
↪and B in the yaml.
    fn = os.path.join(data_dir, f"my_file_{sample_fn}.h5ad")
    adata = anndata.read(fn)
    return adata
```

Loaders for meta studies or atlases

Meta studies are studies on published gene expression data. Often, multiple previous studies are combined or meta data annotation is changed. Data sets from such meta studies can be added to sfaira just as primary data can be added, we ask for theses studies to be identified through the meta data attribute `primary_data` to allow sfaira users to avoid duplicate cells in data universe partitions.

Let's consider an example case: Study `A` published 2 data sets `A1` and `A2`. Study `B` published 1 data set `B1`. Data loaders for `A` and `B` can label as `primary_data: True`. Now, study `C` published 1 data set `C1` that consists of `A2` and `B1`. We can write a data loaders for `C` and label it as `primary_data: False`. Moreover, when conducting the study `C`, we could even base our analyses directly on the data loaders of `A2` and `B1` to make the data analysis pipeline more reproducible.

Curating cell type annotation

Common challenges in cell type curation include the following:

1. **An free-text label is used that is not well captured by the automated search.** Often, these are abbreviations are synonyms that can be mapped to the ontology after looking these terms up online or in the manuscript corresponding to the data loader. Indeed, it is good practice to manually verify non-trivial cell type label maps with a quick contextualization in manuscript figures or text. As for all other ontology-constrained meta data, EBI OLS maintains a great interface to the ontology under CL.

2. **The free-text labels contain nested annotation.** For example, a low-resolution cluster may be annotated as "T cell" in one data set, while other data sets within the same study have more specific T cell labels. Simply map each of these labels to their best fit ontology name, you do not need to mitigate differential granularity.
3. **The free-text labels contain cellular phenotypes that map badly to the ontology.** A common example would be "cycling cells". In some tissues, these phenotypes can be related to specific cell types through knowledge on the phenotypes of the cell types that occur in that tissue. If this is not possible or you do not know the tissue well enough, you can leave the cell type as "UNKNOWN" and future curators may improve this annotaiton. In cases such as "cycling T cell", you may just resort to the parent label "T cell" unless you have reason to believe that "cycling" identifies a specific T cell subset here.
4. **The free-text labels are more fine-grained than the ontology.** A common example would be the addition of marker gene expression to cell cluster labels that are grouped under the same ontology identifier. Some times, these marker genes can be mapped to a child node of the ontology identifier. However, often these indicate cell state variation or other, not fully attributed, variation and do not need to be accounted for in this cell type curation step. These are often among the hardest cell type curation problems, keep in mind that you want to find a reasonable translation of the existing curation, you may be limited by the ontology or by the data reported by the authors, so keep an eye on the overall effort that you spend on optimizing these label maps.
5. **A new cell type in annotated in free-text but is not available in the ontology yet.** This is most likely only a problem for a limited period of time in which the ontology works on adding this element. Chose the best match from the ontology and leave an issue on the sfaira GitHub describing the missing cell type. We can then later update this data loader once the ontology is updated.

Multi-modal data

Multi-modal can be represented in the sfaira curation schema, here we briefly outline what modalities are supported and how they are accounted for. You can use any combination of orthogonal meta data, e.g. organ and disease annotation, with multi-modal measurements.

- **RNA:** RNA is the standard modality in sfaira, unless otherwise specified, all information in this document is centered around RNA data.
- **ATAC:** We support scATAC-seq and joint scRNA+ATAC-seq (multiome) data. In both cases, the ATAC data is commonly represented as a UMI count matrix of the dimensions `(observations x peaks)`. Here, peaks are defined by a peak calling algorithm as part of the read processing pipeline upstream of sfaira. Peak counts can be deposited in the core data matrices managed in sfaira. The corresponding feature meta data can be set such that they allow differentiation of RNA and peak features. These features are documented dataset-or-feature-wise and feature-wise.
- **protein quantification through antibody quantification:** We support CITE-seq and spatial molecular profiling assays with protein quantification read-outs. In these cases, the protein data can be represented as a gene expression matrix of the dimensions `(observations x proteins)`. In the case of oligo-nucleotide-tagged antibody quantification, e.g. in CITE-seq, this can also be an UMI matrix. The corresponding feature meta data can be set such that they allow differentiation of RNA and protein features. These features are documented dataset-or-feature-wise and feature-wise.
- **spatial:** A couple of single-cell and spot-based assays have spatial coordinates associated with molecular profiles. We use relative coordinates of observations in a batch as `(x, y, z)` tuples to characterize the spatial information. Note that spatial proximity graphs and similar spatial analyses are down-stream analyses on these coordinates. This features are documented feature-wise.
- **spliced, unspliced transcript and velocities:** We support gene expression matrices on the level of spliced and unspliced transcript and the common processed format of a RNA velocity matrix. Note that the velocity matrix depends on the inference procedure. These matrices share `.var` annotation with the core RNA data

matrix and can, therefore, be supplemented as further layeres in the `AnnData` object without further effort. This features is documented data-matrices.

- **V(D)J in TCR and BCR reconstructions:** V(D)J data is collected in parallel to RNA data in a couple of single-cell assays. We use key meta data defined by the AIRR consortium to characterize the reconstructed V(D)J genes, which are all direct outputs of V(D)J alignment pipelines and are are stored in `.obs`. This features are documented feature-wise.

Reading compressed files

This is a collection of code snippets that can be used in tha `load()` function to read compressed download files. See also the anndata and scanpy IO documentation.

- **Read a .gz compressed .mtx (.mtx.gz):** Note that this often occurs in cellranger output for which their is a scanpy load function that applies to data of the following structure `./PREFIX_matrix.mtx.gz`, `./PREFIX_barcodes.tsv.gz`, and `./PREFIX_features.mtx.gz`. This can be read as:

```
import scanpy
adata = scanpy.read_10x_mtx("./", prefix="PREFIX_")
```

- **Read from within a .gz archive (.gz):** Note: this requires temporary files, so avoid if read_function can read directly from .gz.

```
import gzip
from tempfile import TemporaryDirectory
import shutil
# Insert the file type as a string here so that read_function recognizes the␣
↪decompressed file:
uncompressed_file_type = ""
with TemporaryDirectory() as tmpdir:
    tmppth = tmpdir + f"/decompressed.{uncompressed_file_type}"
    with gzip.open(fn, "rb") as input_f, open(tmppth, "wb") as output_f:
        shutil.copyfileobj(input_f, output_f)
    x = read_function(tmppth)
```

- **Read from within a .tar archive (.tar.gz):** It is often useful to decompress the tar archive once manually to understand its internal directory structure. Let's assume you are interested in a file `fn_target` within a tar archive `fn_tar`, i.e. after decompressing the tar the director is <fn_tar>/<fn_target>.

```
import pandas
import tarfile
with tarfile.open(fn_tar) as tar:
    # Access files in archive with tar.extractfile(fn_target), e.g.
    tab = pandas.read_csv(tar.extractfile(sample_fn))
```

Reading R files

Some studies deposit single-cell data in R language files, e.g. `.rdata`, `.Rds` or Seurat objects. These objects can be read with python functions in sfaira using anndata2ri and rpy2. These modules allow you to run R code from within this python code:

```
def load(data_dir, **kwargs):
    import anndata2ri
    from rpy2.robjects import r
    anndata2ri.activate()

    fn = os.path.join(data_dir, "SOME_FILE.rdata")
    seurat_object_name = "tissue"
    adata = r(
        f"library(Seurat)\n"
        f"load('{fn}')\n"
        f"new_obj = CreateSeuratObject(counts = {seurat_object_name}@raw.data)\n"
        f"new_obj@meta.data = {seurat_object_name}@meta.data\n"
        f"as.SingleCellExperiment(new_obj)\n"
    )
    return adata
```

Loading third party annotation

In some cases, the data set in question is already in the sfaira zoo but there is alternative (third party), cell-wise annotation of the data. This could be different cell type annotation for example. The underlying data (count matrix and variable names) stay the same in these cases, and often, even some cell-wise meta data are kept and only some are added or replaced. Therefore, these cases do not require an additional `load()` function. Instead, you can contribute `load_annotation_*()` functions into the `.py` file of the corresponding study. You can chose an arbitrary suffix for the function but ideally one that identifies the source of this additional annotation in a human readable manner at least to someone who is familiar with this data set. Second you need to add this function into the dictionary `LOAD_ANNOTATION` in the `.py` file, with the suffix as a key. If this dictionary does not exist yet, you need to add it into the `.py` file with this function as its sole entry. Here an example of a `.py` file with additional annotation:

```
def load(data_dir, sample_fn, **kwargs):
    pass

def load_annotation_meta_study_x(data_dir, sample_fn, **kwargs):
    # Read a tabular file indexed with the observation names used in the adata used in␣
↪load().
    pass

def load_annotation_meta_study_y(data_dir, sample_fn, **kwargs):
    # Read a tabular file indexed with the observation names used in the adata used in␣
↪load().
    pass

LOAD_ANNOTATION = {
    "meta_study_x": load_annotation_meta_study_x,
    "meta_study_y": load_annotation_meta_study_y,
}
```

The table returned by `load_annotation_meta_study_x` needs to be indexed with the observation names used in

`.adata`, the object generated in `load()`. If `load_annotation_meta_study_x` contains a subset of the observations defined in `load()`, and this alternative annotation is chosen, `.adata` is subsetted to these observations during loading.

You can also add functions in the `.py` file in the same DOI-based module in sfaira_extensions if you want to keep this additional annotation private. For this to work with a public data loader, you need nothing more than the `.py` file with this `load_annotation_*()` function and the `LOAD_ANNOTATION` of these private functions in sfaira_extensions.

To access additional annotation during loading, use the setter functions `additional_annotation_key` on an instance of either `Dataset`, `DatasetGroup` or `DatasetSuperGroup` to define data sets for which you want to load additional annotation and which additional you want to load for these. See also the docstrings of these functions for further details on how these can be set.

Required metadata

The CLI will flag any required meta data that is missing. Note that you can use the CLI under a specific schema, e.g. the more lenient sfaira schema (default) or the stricter cellxgene schema, by giving the arguent `--schema cellxgene` to `finalize-dataloader` or `test-dataloader`. Moreover, `.h5ad` files from phase 5 can be checked for match to a particular schema in phase 6. In brief, the following meta data are required:

- **dataset_structure:**
  - `dataset_index`
  - `sample_fns` is required in multi-dataset loaders to define the number and identity of datasets.
- **dataset_wise:**
  - `author`
  - one DOI (i.e. either `doi_journal` or `doi_preprint`)
  - `download_url_data`
  - `primary_data`
  - `year`
- **layers:**
  - `layer_counts` or `layer_processed`
- **dataset_or_feature_wise:**
  - `feature_type` or `feature_type_var_key`
- **dataset_or_observation_wise:** Either the dataset-wide item or the corresponding `_obs_key` are required to submit a data loader to sfaira:
  - `assay_sc`
  - `organism`

  The following are encouraged in sfaira and required in the cellxgene schema:
  - `assay_sc`
  - `cell_type`
  - `developmental_stage`
  - `disease`
  - `ethnicity`
  - `organ`

        - organism
        - sex

- **feature_wise:** None is required.
- **feature_wise:**
        - feature_id_var_key or feature_symbol_var_key
- **meta:**
        - version

Field descriptions

We constrain meta data by ontologies where possible. Meta data can either be dataset-wise, observation-wise or feature-wise.

Dataset structure

Dataset structure meta data are in the section `dataset_structure` in the `.yaml` file.

- **dataset_index [int]** Numeric identifier of the first loader defined by this python file. Only relevant if multiple python files for one DOI generate loaders of the same name. In these cases, this numeric index can be used to distinguish them.
- **sample_fns [list of strings]** If there are multiple data files which can be covered by one `load()` function and `.yaml` file because they are structured similarly, these can identified here. See also section `Loading multiple files of similar structure`. You can simply hardcode a file name in the `load()` function and skip defining it here if you are writing a single file loader. Note: A sample is a an object similar to a count matrix or a `.h5ad`, the definition of biological or technical batches or samples within one count matrix does not affect this entry.

Dataset-wise

Dataset-wise meta data are in the section `dataset_wise` in the `.yaml` file.

- **author [list of strings]** List of author names of dataset (not of loader).
- **doi [list of strings]** DOIs associated with dataset. These can be preprints and journal publication DOIs.
- **download_url_data [list of strings]** Download links for data. Full URLs of all data files such as count matrices. Note that distinct observation-wise annotation files can be supplied in download_url_meta.
- **download_url_meta [list of strings]** Download links for observation-wise data. Full URLs of all observation-wise meta data files such as count matrices. This attribute is optional and not necessary ff observation-wise meta data is already in the files defined in `download_url_data`, e.g. often the case for `.h5ad`.
- **primary_data: If this is the first publication to report this gene expression data {True, False}.** This is False if the study is a meta study that uses data that was previously published. This usually implies that one can also write a data loader for the data from the primary study. Usually, the data here contains new meta data or is combined with other data sets (e.g. in an "atlas"), Therefore, this data loader is different from a data laoder for the primary data. In sfaira, we maintain data loaders both for the corresponding primary and such meta data publications. See also the section on meta studies meta-studies.
- **year: Year in which sample was first described [integer]** Pre-print publication year.

Data matrices

A curated AnnData object may contain multiple data matrices: raw and processed gene expression counts, or spliced and unspliced count data and velocity estimates, for example. Minimally, you need to supply either of the matrices "counts" or "processed". In the following, "*counts" refers to the INTEGER count of alignment events (e.g. transcripts for RNA). In the following, "*processed" refers to any processing that modifies these counts, for example: normalization, batch correction, ambient RNA correction.

- layer_counts: The total event counts per feature, e.g. UMIs that align to a gene. {'X', 'raw', or a .layers key}
- layer_processed: Processed complement of 'layer_counts'. {'X', 'raw', or a .layers key}
- layer_spliced_counts: The total spliced RNA counts per gene. {a .layers key}
- layer_spliced_processed: Processed complement of 'layer_spliced_counts'. {a .layers key}
- layer_unspliced_counts: The total unspliced RNA counts per gene. {a .layers key}
- layer_unspliced_processed: Processed complement of 'layer_unspliced_counts'. {a .layers key}
- layer_velocity: The RNA velocity estimates per gene. {a .layers key}

Dataset- or feature-wise

These meta data may be defined across the entire dataset or per feature and are in the section `dataset_or_feature_wise` in the `.yaml` file: They can all be supplied as `NAME` or as `NAME_var_key`: The former indicates that the entire data set has the value stated in the yaml. The latter, `NAME_var_key`, indicates that there is a column in `adata.var` emitted by the `load()` function of the name `NAME_var_key` which contains the annotation per feature for this meta data item. Note that in both cases the value, or the column values, have to fulfill constraints imposed on the meta data item as

- **feature_reference and feature_reference_var_key [string]** The genome annotation release that was used to quantify the features presented here, e.g. "Homo_sapiens.GRCh38.105". You can find all ENSEMBL gtf files on the ensembl ftp server. Here, you ll find a summary of the gtf files by release, e.g. for 105. You will find a list across organisms for this release, the target release name is the name of the gtf files that ends on `.RELEASE.gtf.gz` under the corresponding organism. For homo_sapiens and release 105, this yields the following reference name "Homo_sapiens.GRCh38.105".
- **feature_type and feature_type_var_key {"rna", "protein", "peak"}** The type of a feature:
  - **"rna": gene expression quantification on the level of RNA** e.g. from scRNA-seq or spatial RNA capture experiments
  - **"protein": gene expression quantification on the level of proteins** e.g. via antibody counts in CITE-seq or spatial protocols
  - **"peak": chromatin accessibility by peak** e.g. from scATAC-seq

Dataset- or observation-wise

These meta data may be defined across the entire dataset or per observation and are in the section `dataset_or_observation_wise` in the `.yaml` file: They can all be supplied as `NAME` or as `NAME_obs_key`: The former indicates that the entire data set has the value stated in the yaml. The latter, `NAME_obs_key`, indicates that there is a column in `adata.obs` emitted by the `load()` function of the name `NAME_obs_key` which contains the annotation per observation for this meta data item. Note that in both cases the value, or the column values, have to fulfill constraints imposed on the meta data item as outlined below.

- **assay_sc and assay_sc_obs_key [ontology term]** The EFO label corresponding to single-cell assay of the sample. The corresponding subset of EFO_SUBSET is the set of child nodes of "single cell library construction" (EFO:0010183).
- **assay_differentiation and assay_differentiation_obs_key [string]** Try to provide a base differentiation protocol (eg. "Lancaster, 2014") as well as any amendments to the original protocol.
- **assay_type_differentiation and assay_type_differentiation_obs_key {"guided", "unguided"}** For cell-culture samples: Whether a guided (patterned) differentiation protocol was used in the experiment.
- **bio_sample and bio_sample_obs_key [string]** Column name in `adata.obs` emitted by the `load()` function which reflects biologically distinct samples, either different in condition or biological replicates, as a categorical variable. The values of this column are not constrained and can be arbitrary identifiers of observation groups. You can concatenate multiple columns to build more fine grained observation groupings by concatenating the column keys with * in this string, e.g. `patient*treatment` to get one `bio_sample` for each patient and treatment. Note that the notion of biologically distinct sample is slightly subjective, we allow this element to allow researchers to distinguish technical and biological replicates within one study for example. See also the meta data items `individual` and `tech_sample`.
- **cell_line and cell_line_obs_key [ontology term]** Cell line name from the cellosaurus cell line database.
- **cell_type and cell_type_obs_key [ontology term]** Cell type name from the Cell Ontology CL database. Note that sometimes, original (free-text) cell type annotation is provided at different granularities. We recommend choosing the most fine-grained annotation here so that future re-annotation of the cell types in this loader is easier. You may choose to compromise the potential for re-annotation of the data loader with the size of the mapping `.tsv` that is generated during annotation: This file has one row for free text label and may be undesirably large in some cases, which reduces accessibilty of the data loader code for future curators, thus presenting a trade-off. See also the section on cell type annotation celltype-annotation.
- **developmental_stage and developmental_stage_obs_key [ontology term]** Developmental stage (age) of individual sampled. Choose from HSAPDV for human or from MMUSDEV for mouse.
- **disease and disease_obs_key [ontology term]** Choose from MONDO.
- **ethnicity and ethnicity_obs_key [ontology term]** Choose from HANCESTRO.
- **gm and gm_obs_key [string]** Genetic modification. E.g. identify gene knock-outs or over-expression as a boolean indicator per cell or as guide RNA counts in approaches like CROP-seq or PERTURB-seq.
- **individual and individual_obs_key [string]** Column name in `adata.obs` emitted by the `load()` function which reflects the indvidual sampled as a categorical variable. The values of this column are not constrained and can be arbitrary identifiers of observation groups. You can concatenate multiple columns to build more fine grained observation groupings by concatenating the column keys with * in this string, e.g. `group1*group2` to get one `individual` for each group1 and group2 entry. Note that the notion of individuals is slightly mal-defined in some cases, we allow this element to allow researchers to distinguish sample groups that originate from biological material with distinct genotypes. See also the meta data items `individual` and `tech_sample`.
- **organ and organ_obs_key [ontology term]** The UBERON label of the sample. This meta data item ontology is for tissue or organ identifiers from UBERON.

- **organism and organism_obs_key. [ontology term]** The NCBItaxon label of the main organism sampled here. For a data matrix of an infection sample aligned against a human and virus joint reference genome, this would "Homo sapiens" as it is the "main organism" in this case. For example, "Homo sapiens" or "Mus musculus". See also the documentation of feature_reference to see which orgainsms are supported.
- **primary_data [bool]** Whether contains cells that were measured in this study (ie this is not a meta study on published data).
- **sample_source and sample_source_obs_key. {"primary_tissue", "2d_culture", "3d_culture", "tumor"}** Which cellular system the sample was derived from.
- **sex and sex_obs_key. Sex of individual sampled. [ontology term]** The PATO label corresponding to sex of the sample. The corresponding subset of PATO_SUBSET is the set of child nodes of "phenotypic sex" (PATO:0001894).
- **source_doi and source_doi_obs_key [string]** If this dataset is not primary data, you can supply the source of the analyzed data as a DOI per dataset or per cell in this meta data item. The value of this metadata item (or the entries in the corresponding `.obs` column) needs to be a DOI
- **state_exact and state_exact_obs_key [string]** Free text description of condition. If you give treatment concentrations, intervals or similar measurements use square brackets around the quantity and use units: `[1g]`
- **tech_sample and tech_sample_obs_key [string]** Column name in `adata.obs` emitted by the `load()` function which reflects technically distinct samples, either different in condition or technical replicates, as a categorical variable. Any data batch is a `tech_sample`. The values of this column are not constrained and can be arbitrary identifiers of observation groups. You can concatenate multiple columns to build more fine grained observation groupings by concatenating the column keys with `*` in this string, e.g. `patient*treatment*protocol` to get one `tech_sample` for each patient, treatment and measurement protocol. See also the meta data items `individual` and `tech_sample`.
- **treatment and treatment_obs_key [string]** Treatment of sample, e.g. compound names in stimulation experiments.

Feature-wise

These meta data are always defined per feature and are in the section `feature_wise` in the `.yaml` file:

- **feature_id_var_key [string]** Name of the column in `adata.var` emitted by the `load()` which contains ENSEMBL gene IDs. This can also be "index" if the ENSEMBL gene names are in the index of the `adata.var` data frame. Note that you do not have to map IDs to a specific annotation release but can keep them in their original form. If available, IDs are preferred over symbols.
- **feature_symbol_var_key [string]** Name of the column in `adata.var` emitted by the `load()` which contains gene symbol: HGNC for human and MGI for mouse. This can also be "index" if the gene symbol are in the index of the `adata.var` data frame. Note that you do not have to map symbols to a specific annotation release but can keep them in their original form.

Observation-wise

These meta data are always defined per observation and are in the section `observation_wise` in the `.yaml` file:

The following items are only relevant for spatially resolved data, e.g. spot transcriptomics or MERFISH:

- **spatial_x_coord, spatial_y_coord, spatial_z_coord [string]** Spatial coordinates (numeric) of observations. Most commonly, the centre of a segment or of a spot is indicated here. For 2D data, a z-coordinate is not relevant and can be skipped.

The following items are only relevant for V(D)J reconstruction data, e.g. TCR or BCR sequencing in single cells. These meta data items are described in the AIRR project, search the this link for the element in question without the prefixed "vdj_". These 10 meta data items describe chains (or loci). In accordance with the corresponding scirpy defaults, we allow for up to two loci per cell. In T cells, this correspond to two VJ loci (TRA) and two VDJ loci (TRB). You can set the prefix of the column of each of the four loci below. In total, these 10+4 meta data queries in sfaira describe 4*10 columns in `.obs` after `.load()`. Note that for this to work, you need to stick to the naming convention `PREFIX_SUFFIX`. We recommend that you use `scirpy.io` functions for reading the VDJ data in your `load()` to use the default meta data keys suggested by the CLI and to guarantee that this naming convention is obeyed.

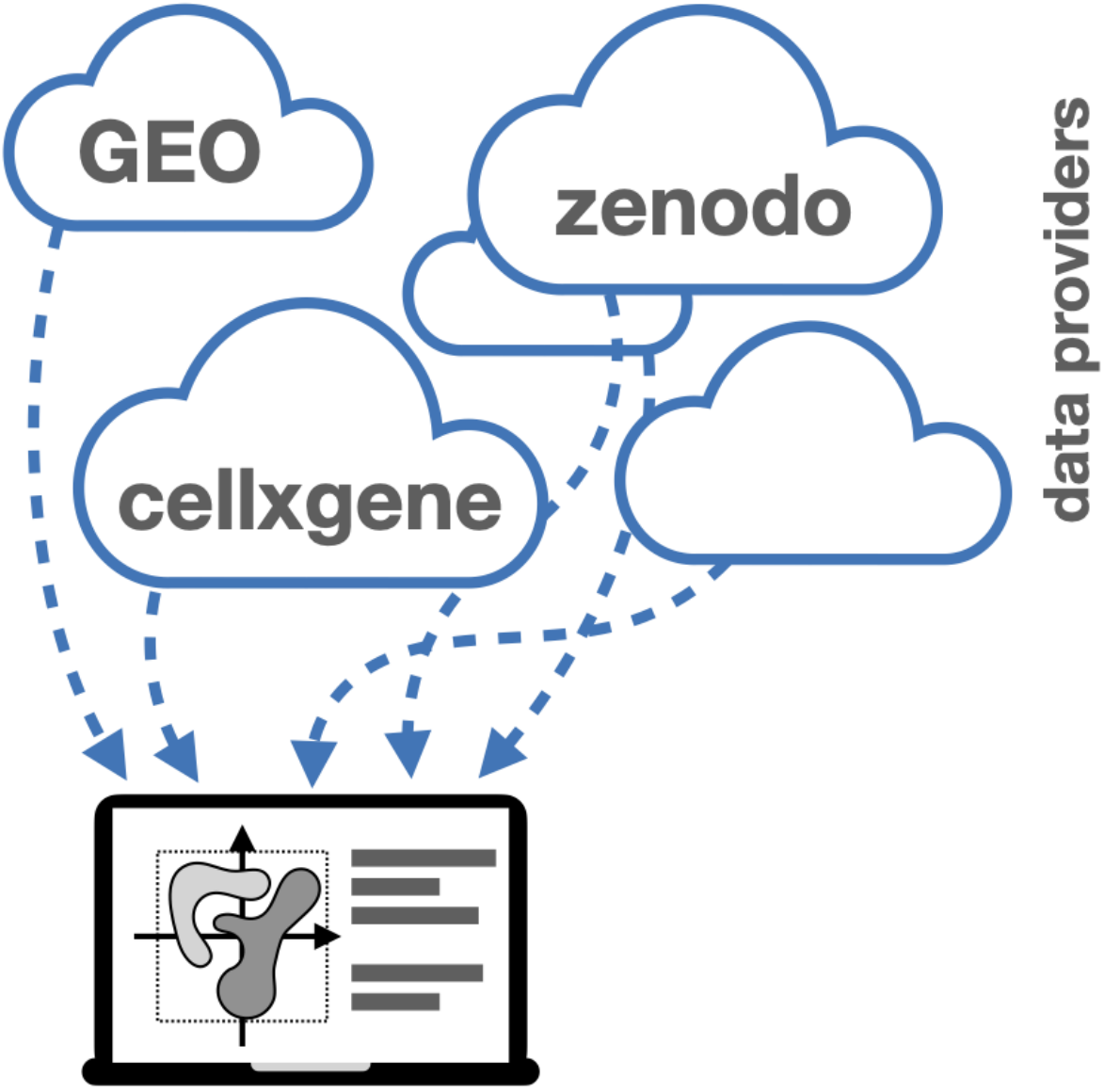
- **vdj_vj_1_obs_key_prefix** Prefix of key of columns corresponding to first VJ gene.
- **vdj_vj_2_obs_key_prefix** Prefix of key of columns corresponding to second VJ gene.
- **vdj_vdj_1_obs_key_prefix** Prefix of key of columns corresponding to first VDJ gene.
- **vdj_vdj_2_obs_key_prefix** Prefix of key of columns corresponding to second VDJ gene.
- **vdj_c_call_obs_key_suffix** Suffix of key of columns corresponding to C gene.
- **vdj_consensus_count_obs_key_suffix** Suffix of key of columns corresponding to number of reads contributing to consensus.
- **vdj_d_call_obs_key_suffix** Suffix of key of columns corresponding to D gene.
- **vdj_duplicate_count_obs_key_suffix** Suffix of key of columns corresponding to number of duplicate UMIs.
- **vdj_j_call_obs_key_suffix** Suffix of key of columns corresponding to J gene.
- **vdj_junction_obs_key_suffix** Suffix of key of columns corresponding to junction nt sequence.
- **vdj_junction_aa_obs_key_suffix** Suffix of key of columns corresponding to junction aa sequence.
- **vdj_locus_obs_key_suffix** Suffix of key of columns corresponding to gene locus, i.e IGH, IGK, or IGL for BCR data and TRA, TRB, TRD, or TRG for TCR data.
- **vdj_productive_obs_key_suffix** Suffix of key of columns corresponding to locus productivity: whether the V(D)J gene is productive.
- **vdj_v_call_obs_key_suffix** Suffix of key of columns corresponding to V gene.

Meta

These meta data contain information about the curation process and schema:

- **version: [string]** Version identifier of meta data scheme.

2.2.7 Using data loaders



For a high-level overview of data management in sfaira, read *The data life cycle* first.

Build data repository locally

Build a repository structure

1. Choose a directory to dedicate to the data base, called root in the following.
2. Run the sfaira download script (sfaira.data.utils.download_all). Alternatively, you can manually set up a data base by making subfolders for each study.

Note that the automated download is a feature of sfaira but not the core purpose of the package: Sfaira allows you efficiently interact with such a local data repository. Some data sets cannot be automatically downloaded and need you

manual intervention, which we report in the download script output.

Use 3rd party repositories

Some organization provide streamlined data objects that can be directly consumed by data zoos such as sfaira. One example for such an organization is the cellxgene data portal. Through these repositories, one can easily build or extend a collection of data sets that can be easily interfaced with sfaira. Data loaders for cellxgene structured data objects will be available soon! Contact us for support of any other repositories.

2.2.8 Data stores

For a high-level overview of data management in sfaira, read *The data life cycle* first. Sfaira supports usage of distributed data for model training and execution. The tools are summarized under `sfaira.data.store`. In contrast to using an instance of AnnData in memory, these tools can be used to use data sets that are saved in different files (because they come from different studies) flexibly and out-of-core, which means without loading them into memory. A general use case is the training of a model on a large set of data sets, subsetted by particular cell-wise meta data, without creating a merged AnnData instance in memory first.

Build a distributed data repository

You can use the sfaira dataset API to write streamlined groups of adata instances to a particular disk locaiton that then is the store directory. Some of the array backends used for loading stores can read arrays from cloud servers, such as dask. Therefore, these store directories can also be on cloud servers in some cases.

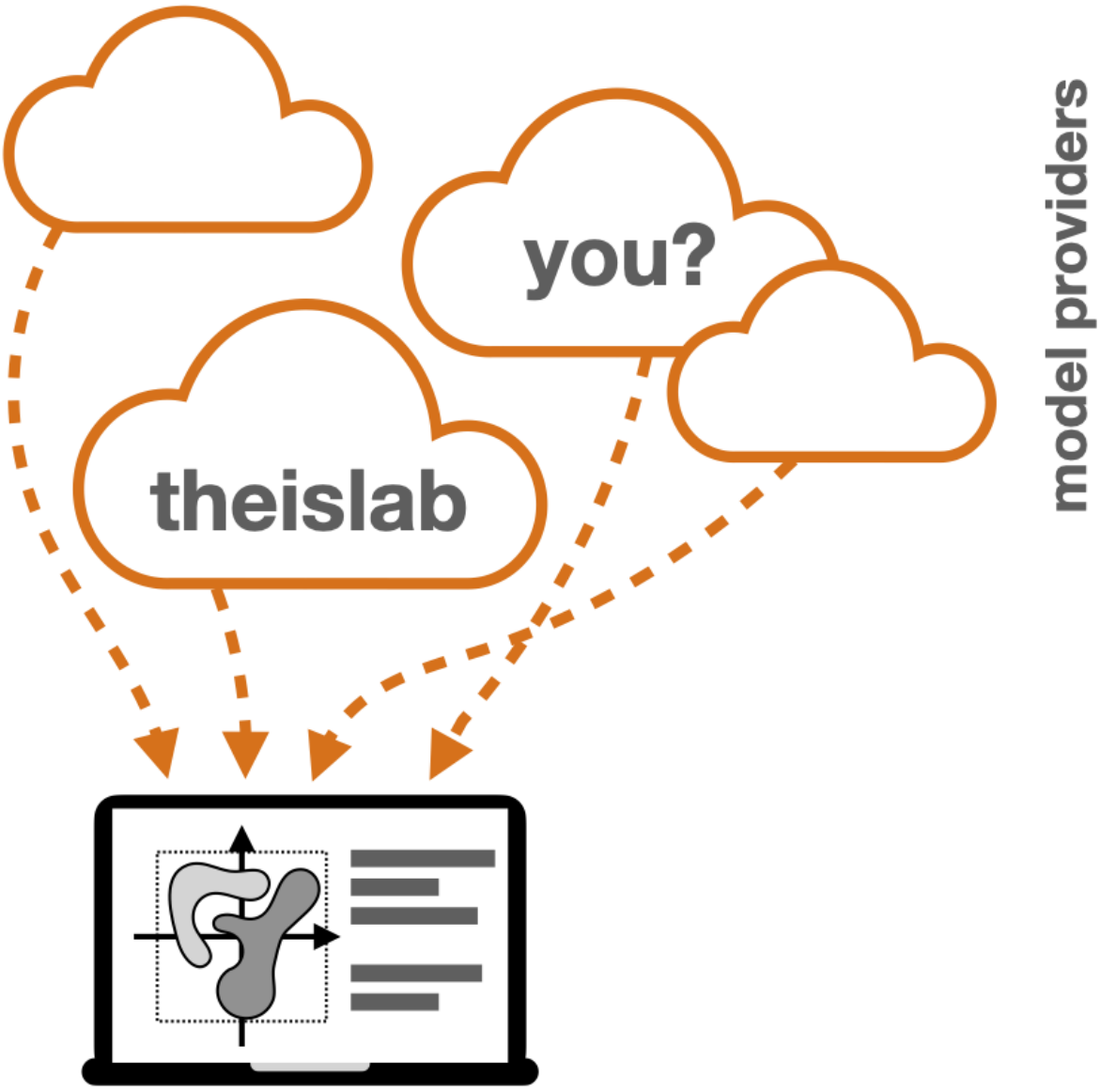
Reading from a distributed data repository

The core use-case is the consumption of data in batches from a python iterator (a "generator"). In contrast to using the full data matrix, this allows for workflows that never require the full data matrix in memory. This generators can for example directly be used in tensorflow or pytorch stochastic mini-batch learning pipelines. The core interface is `sfaira.data.load_store()` which can be used to initialise a store instance that exposes a generator, for example. An important concept in store reading is that the data sets are already streamlined on disk, which means that they have the same feature space for example.

Distributed access optimised (DAO) store

The DAO store format is a on-disk representation of single-cell data which is optimised for generator-based access and distributed access. In brief, DAO stores optimize memory consumption and data batch access speed. Right now, we are using zarr and parquet, this may change in the future, we will continue to work on this format using the project name "dao". Note that data sets represented as DAO on disk can still be read into AnnData instances in memory if you wish!

2.2.9 Models



User interface

The user interface allows users to query model code and parameter estimates to run on local data. It takes care of downloading model parameters from the relevant cloud storage, loading parameters into a model instance locally and performing the forward pass. With the user interface, users only have to worry about which model they want to execute, but now how this is facilitated.

Model management

A sfaira model is a class that inherits from BasicModel which defines a tf.keras.models.Model in self.training_model. This training_model describes the full forward pass. Additionally, embedding models also have an attribute X, a tf.keras.models.Model that describes the partial forward pass into the embedding layer.

Such a model class, e.g. ModelX, is wrapped by an inheriting class ModelXVersioned, that handles properties of the model architecture. In particular, ModelXVersioned

- has access to the cell ontology container (a daughter class of CelltypeVersionsBase) that corresponds to this model if applicable
- has access to a map of a version ID to an architectural hyperparameter setting (Topologies), allowing this class to set depth, width, etc of the model directly based on the name of the yielded model.
- has access to the feature space of the model, including its gene names, which are defined by the model topology in Topologies

Contribute models

Models can be contributed and used in two ways

- Full model code in sfaira repo
- sfaira compatible model code in external package (to come)

Training

Estimator classes

We define estimator classes that have model instances as an attribute, that orchestrate all major aspects of model fitting, such as a data loading, data streaming and model evaluation.

2.2.10 Ecosystem

scanpy

scanpy provides an environment of tools that can be used to analysis single-cell data in python. sfaira allows users to easily query third party data sets and models to complement these analysis workflows.

Data zoo

Data providers which streamline data

Some organization provide streamlined data objects that can be directly consumed by data zoos such as sfaira. Examples for such data providers are:

- Human Cell Atlas data portal (HCA DCP)
- cellxgene data portal
- Broad institute single cell data portal
- EBI single cell expression atlas

Through these repositories, one can easily build or extend a collection of data sets that can be easily interfaced with sfaira. Data loaders for cellxgene structured data objects will be available soon, we are working on interfacing more such organisations! Contact us for support of any other repositories.

Study-centric data set servers

Many authors of data sets provide their data sets on servers:

- GEO
- cloud storage servers
- manuscript supplements

Our data zoo interface is able to represent these data sets such that they can be queried in a streamlined fashion, together with many other data sets.

Single-cell study look-up tables

Svensson et al. published a single-cell database in the form of a table in which each row contains a description of a study which published single-cell RNA-seq data. Some of these data sets are already included in sfaira, consider also our interactive website for a graphical user interface to our complete data zoo. Note that this website can be used as a look-up table but sfaira also allows you to directly load and interact with these data sets.

2.2.11 Roadmap

Cell ontologies

We are currently migrating our ontology to use the Cell Ontology as a backbone. For details, read through this milestone.

Interface online data repositories

We are preparing to interface online data repositories which provide streamlined data. This allows users to build local data set collections more easily because these providers usually have a clear download interface, consider the cellxgene data portal for example. We aim to represent both these data set portals and data sets that have not been streamlined in such a fashion to provide a comprehensive collection of as many data sets as possible.

2.2.12 FAQ

Data zoo

How is load() function used in data loading?

`load()` contains all processing steps that load raw data files into a ready to use adata object. This adata object can be cached as an h5ad file named after the dataset ID for faster reloading (if allow_caching=True), which exactly skips the code in `load()`. `load()` can be triggered to reload from scratch even if cached data is available (if use_cached=False).

How is the feature space (gene names) manipulated during data loading?

Sfaira provides both gene names and ENSEMBL IDs. Missing IDs will automatically be inferred from the gene names and vice versa. Version tags on ENSEMBL gene IDs will be removed if specified (if remove_gene_version=True); in this case, counts are aggregated across these features. Sfaira makes sure that gene IDs in a dataset match IDs of chosen reference genomes.

Datasets, DatasetGroups, DatasetSuperGroups - what are they?

Dataset: Custom class that loads a specific dataset. DatasetGroup: A dataset group manages collection of data loaders (multiple instances of Dataset). This is useful to group for example all data loaders corresponding to a certain study or a certain tissue. DatasetSuperGroups: A group of DatasetGroups that allow easy addition of multiple instances of DatasetGroup.

Basics of sfaira lazy loading via split into constructor and load() function.

The constructor of a dataset defines all metadata associated with this data set. The loading of the actual data happens in the `load()` function and not in the constructor. This is useful as it allows initialising the datasets and accessing dataset metadata without loading the actual count data. DatasetGroups can contain initialised Datasets and can be sub-setted based on metadata before loading is triggered across the entire group.

2.2.13 Changelog

This project adheres to Semantic Versioning.

0.2.1 2020-09-7

Added

- A commandline interface with Click, Rich and Questionary
- upgrade command, which checks whether the latest version of sfaira is installed on every sfaria startup and upgrades it if not.
- create-dataloader command which allows for the interactive creation of a sfaira dataloader script
- clean-dataloader command which cleans a with sfaira create-dataloader created dataloader script
- lint-dataloader command which runs static checks on the style and completeness of a dataloader script
- test-dataloader command which runs a unittest on a provided dataloader

Fixed

Dependencies

Deprecated

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z