scArches

Mohammad Lotfollahi, Sergei Rybakov, Mohsen Naghipourfar

Jun 27, 2023

MAIN

1 scArches (PyTorch) - single-cell architecture surgery 3

2 Installation 5

3 API 7

4 A few tips on training models 43

5 Unsupervised surgery pipeline with SCVI 45

6 Semi-supervised surgery pipeline with SCANVI 51

7 Multi-Modal Surgery Pipeline with TOTALVI 61

8 Unsupervised surgery pipeline with TRVAE 71

9 Build reference atlas from scratch 77

10 Reference maping using scGen 85

11 Basic tutorial for query to reference maping using expiMap 91

12 Advanced tutorial for query to reference mapping using expiMap with de novo learned gene programs 101

13 treeArches: learning and updating a cell-type hierarchy (basic tutorial) 113

14 treeArches: identifying new cell types (advanced tutorial) 137

15 Spatial reconstruction of the mouse embryo with SageNet 151

16 Tutorial for mvTCR 165

17 Integration and reference mapping with multigrate 173

18 Integration, label transfer and multi-scale analysis with scPoli 189

19 Mapping data to the Human Lung Cell Atlas for joint analysis and cell type label transfer 201

Python Module Index 223

Index 225

scArches allows your single-cell query data to be analyzed by integrating it into a reference atlas. To map your data, you need an integrated atlas using one of the reference-building methods for different applications that are supported by scArches which are, including:

- **Annotating a single-cell dataset using a reference atlas**: You can check following models/tutorials using scPoli (De Donno et al., 2022) or scANVI (Xu et al., 2019).
- **Identify novel cell states present in your data by mapping to an atlas**: If you want to detect cell-states affected by disease or novel subpopulations see treeArches (Michielsen*, Lotfollahi* et al., 2022) and also similar use case by mapping to Human Lung cell atlas.
- **Multimodal single-cell atlases**: You can check the tutorial for Multigrate (Litinetskaya*, Lotfollahi* et al., 2022) to work with CITE-seq + Multiome (ATAC+ RNA). Additionally, you can check mvTCR (Drost et al., 2022) for joint analysis of T-cell Receptor (TCR) and scRNAseq data. To impute missing surface proteins for your query single-cell RNAseq data using a CITE-seq reference, see totalVI (Gayoso et al., 2019).
- **Data integration/batch correction**: For integration of multiple scRNAseq datasets see scVI (Lopez et al, 2018) or trVAE (Lotfollahi et al, 2020). In case of strong batch effect and access to cell-type labels, consider using scGen (Lotfollahi et al., 2019).
- **Spatial transcriptomics**: To map scRNAseq data to a spatial reference and infer spatial locations check SageNet (Heidari et al., 2022).
- **Querying gene programs in single-cell atlases**: Using gene programs (GPs), you can embed your datasets into known subspaces (e.g., interferon signaling) and see the activity of your query dataset within desired GPs. You can use available GP databases (e.g, GO pathways) or your curated GPs, see expiMap (Lotfollahi*, Rybakov* et al., 2023). One can also learn novel GPs as shown here.

Links to the papers can be found here.

CHAPTER

ONE

SCARCHES (PYTORCH) - SINGLE-CELL ARCHITECTURE SURGERY

scArches is a package to integrate newly produced single-cell datasets into integrated reference atlases. Our method can facilitate large collaborative projects with decentralized training and integration of multiple datasets by different groups. scArches is compatible with scanpy. and hosts efficient implementations of all conditional generative models for single-cell data.

Note: expiMap has been added to scArches code base. It allows interpretable representation learning from scRNA-seq data and also reference mapping. Try it in the tutorial section.

1.1 What can you do with scArches?

- Construct single or multi-modal (CITE-seq) reference atlases and share the trained model and the data (if possible).
- Download a pre-trained model for your atlas of interest, update it with new datasets and share with your collaborators.
- Project and integrate query datasets on the top of a reference and use latent representation for downstream tasks, e.g.:diff testing, clustering, classification

1.2 What are the different models?

scArches is itself an algorithm to map to project query on the top of reference datasets and applies to different models. Here we provide a short explanation and hints on when to use which model. Our models are:

- **scVI** (Lopez et al., 2018): Requires access to raw counts values for data integration and assumes count distribution on the data (NB, ZINB, Poisson).
- **trVAE** (Lotfollahi et al.,2020): It supports both normalized log-transformed or count data as input and applies additional MMD loss to have better merging in the latent space.
- **scANVI** (Xu et al., 2019): It needs cell type labels for reference data. Your query data can be either unlabeled or labeled. In the case of unlabeled query data, you can use this method also to classify your query cells using reference labels.
- **scGen** (Lotfollahi et al., 2019): This method requires cell-type labels for both reference building and Mapping. The reference mapping for this method solely relies on the integrated reference and requires no fine-tuning.
- **expiMap** (Lotfollahi*, Rybakov* et al., 2023): This method takes prior knowledge from gene sets databases or users allowing to analyze your query data in the context of known gene programs.

- **totalVI** (Gayoso al., 2019): This model can be used to build multi-modal CITE-seq reference atalses.
- **treeArches** (Michielsen*, Lotfollahi* et al., 2022): This model builds a hierarchical tree for cell-types in the reference atlas and when mapping the query data can annotate and also identify novel cell-states and populations present in the query data.
- **SageNet** (Heidari et al., 2022): This model allows constrcution of a spatial atlas by mapping query dissociated single cells/spots (e.g., from scRNAseq or visium datasets) into a common coordinate framework using one or more spatially resolved reference datasets.
- **mvTCR** (Drost et al., 2022): Using this model you will be able to integrate T-cell receptor (TCR, treated as a sequence) and scRNA-seq dataset across multiple donors into a joint representation capturing information from both modalities.
- **scPoli** (De Donno et al., 2022): This model allows data integration of scRNA-seq dataset, prototype-based label transfer and reference mapping. scPoli learns both sample embeddings and integrated cell embeddings, thus providing the user with a multi-scale view of the data, especially useful in the case of many samples to integrate.

1.3 Where to start?

To get a sense of how the model works please go through this tutorial. To find out how to construct and share or use pre-trained models example sections.

1.4 Reference

If scArches is useful in your research, please consider citing the paper.

CHAPTER

TWO

INSTALLATION

scArches requires Python 3.7 or 3.8. We recommend to use Miniconda.

2.1 PyPI

The easiest way to get scArches is through pip using the following command:

```
sudo pip install -U scarches
```

2.2 Conda Environment

You can also use our environment file. This will create the conda environment 'scarches' with the required dependencies:

```
git clone https://github.com/theislab/scarches
cd scarches
conda env create -f envs/scarches_linux.yaml
conda activate scarches
```

2.3 Development

You can also get the latest development version of scArches from Github using the following steps: First, clone scArches using `git`:

```
git clone https://github.com/theislab/scarches
```

Then, `cd` to the scArches folder and run the install command:

```
cd scarches
python3 setup.py install
```

On Windows machines you may need to download a C++ compiler if you wish to build from source yourself.

2.4 Dependencies

The list of dependencies for scArches can be found in the requirements.txt file in the repository.

If you run into issues, do not hesitate to approach us or raise a GitHub issue.

CHAPTER

THREE

API

The API reference contains detailed descriptions of the different end-user classes, functions, methods, etc.

Note: This API reference only contains end-user documentation. If you are looking to hack away at scArches' internals, you will find more detailed comments in the source code.

Import scarches as:

```
import scarches as sca
```

After reading the data (`sca.data.read`), you can you can instantiate one of the implemented models from `sca.models` module (currently we support `trVAE`, `scVI`, `scANVI`, and `TotalVI`) and train it on your dataset.

3.1 Data Processing

`scarches.dataset.`**`label_encoder`**(*adata*, *encoder*, *condition_key=None*)

Encode labels of Annotated *adata* matrix. :param adata: Annotated data matrix. :type adata: : *~anndata.AnnData* :param encoder: dictionary of encoded labels. :type encoder: Dict :param condition_key: column name of conditions in *adata.obs* data frame. :type condition_key: String

Returns

- **labels** (*~numpy.ndarray*) – Array of encoded labels
- **label_encoder** (*Dict*) – dictionary with labels and encoded labels as key, value pairs.

`scarches.dataset.`**`remove_sparsity`**(*adata*)

If `adata.X` is a sparse matrix, this will convert it in to normal matrix. :param adata: Annotated data matrix. :type adata: `AnnData`

Returns
adata – Annotated dataset.

Return type
`AnnData`

`scarches.dataset.`**`trVAEDataset`**

alias of `AnnotatedDataset`

3.2 Models

- *trVAE*
- *expiMap*
- *scPoli*
- *scVI*
- *scANVI*
- *TotalVI*

3.2.1 trVAE

class scarches.models.**TRVAE**(*adata: AnnData, condition_key: str | None = None, conditions: list | None = None, hidden_layer_sizes: list = [256, 64], latent_dim: int = 10, dr_rate: float = 0.05, use_mmd: bool = True, mmd_on: str = 'z', mmd_boundary: int | None = None, recon_loss: str | None = 'nb', beta: float = 1, use_bn: bool = False, use_ln: bool = True*)

Bases: `BaseMixin`, `SurgeryMixin`, `CVAELatentsMixin`

Model for scArches class. This class contains the implementation of Conditional Variational Auto-encoder.

Parameters

- **adata** (: *~anndata.AnnData*) – Annotated data matrix. Has to be count data for 'nb' and 'zinb' loss and normalized log transformed data for 'mse' loss.
- **condition_key** (*String*) – column name of conditions in *adata.obs* data frame.
- **conditions** (*List*) – List of Condition names that the used data will contain to get the right encoding when used after reloading.
- **hidden_layer_sizes** (*List*) – A list of hidden layer sizes for encoder network. Decoder network will be the reversed order.
- **latent_dim** (*Integer*) – Bottleneck layer (z) size.
- **dr_rate** (*Float*) – Dropput rate applied to all layers, if `dr_rate`==0 no dropout will be applied.
- **use_mmd** (*Boolean*) – If 'True' an additional MMD loss will be calculated on the latent dim. 'z' or the first decoder layer 'y'.
- **mmd_on** (*String*) – Choose on which layer MMD loss will be calculated on if 'use_mmd=True': 'z' for latent dim or 'y' for first decoder layer.
- **mmd_boundary** (*Integer or None*) – Choose on how many conditions the MMD loss should be calculated on. If 'None' MMD will be calculated on all conditions.
- **recon_loss** (*String*) – Definition of Reconstruction-Loss-Method, 'mse', 'nb' or 'zinb'.
- **beta** (*Float*) – Scaling Factor for MMD loss
- **use_bn** (*Boolean*) – If *True* batch normalization will be applied to layers.
- **use_ln** (*Boolean*) – If *True* layer normalization will be applied to layers.

Methods

`get_latent`([x, c, mean, mean_var])	Map *x* in to the latent space. This function will feed data in encoder and return z for each sample in data. :param x: Numpy nd-array to be mapped to latent space. *x* has to be in shape [n_obs, input_dim]. If None, then *self.adata.X* is used. :param c: *numpy nd-array* of original (unencoded) desired labels for each sample. :param mean: return mean instead of random sample from the latent space :param mean_var: return mean and variance instead of random sample from the latent space if *mean=False*.
`get_y`([x, c])	Map *x* in to the latent space.
`load`(dir_path[, adata, map_location])	Instantiate a model from the saved output. :param dir_path: Path to saved outputs. :param adata: AnnData object. If None, will check for and load anndata saved with the model. :param map_location: a function, torch.device, string or a dict specifying how to remap storage locations.
`load_query_data`(adata, reference_model[, ...])	Transfer Learning function for new data.
`save`(dir_path[, overwrite, save_anndata])	Save the state of the model. Neither the trainer optimizer state nor the trainer history are saved. :param dir_path: Path to a directory. :param overwrite: Overwrite existing data or not. If *False* and directory already exists at *dir_path*, error will be raised. :param save_anndata: If True, also saves the anndata :param anndata_write_kwargs: Kwargs for anndata write function.
`train`([n_epochs, lr, eps])	Train the model.

train(*n_epochs: int = 400, lr: float = 0.001, eps: float = 0.01, **kwargs*)

Train the model.

Parameters

- **n_epochs** – Number of epochs for training the model.
- **lr** – Learning rate for training the model.
- **eps** – torch.optim.Adam eps parameter
- **kwargs** – kwargs for the TrVAE trainer.

3.2.2 expiMap

class scarches.models.**EXPIMAP**(*adata: AnnData, condition_key: str | None = None, conditions: list | None = None, hidden_layer_sizes: list = [256, 256], dr_rate: float = 0.05, recon_loss: str = 'nb', use_l_encoder: bool = False, use_bn: bool = False, use_ln: bool = True, mask: ndarray | list | None = None, mask_key: str = 'I', decoder_last_layer: str | None = None, soft_mask: bool = False, n_ext: int = 0, n_ext_m: int = 0, use_hsic: bool = False, hsic_one_vs_all: bool = False, ext_mask: ndarray | list | None = None, soft_ext_mask: bool = False*)

Bases: `BaseMixin`, `SurgeryMixin`, `CVAELatentsMixin`

Model for scArches class. This class contains the implementation of Conditional Variational Auto-encoder.

Parameters

- **adata** (: *~anndata.AnnData*) – Annotated data matrix. Has to be count data for 'nb' and 'zinb' loss and normalized log transformed data for 'mse' loss.
- **condition_key** (*String*) – column name of conditions in *adata.obs* data frame.
- **conditions** (*List*) – List of Condition names that the used data will contain to get the right encoding when used after reloading.
- **hidden_layer_sizes** (*List*) – A list of hidden layer sizes for encoder network. Decoder network will be the reversed order.
- **latent_dim** (*Integer*) – Bottleneck layer (z) size.
- **dr_rate** (*Float*) – Dropput rate applied to all layers, if `dr_rate`==0 no dropout will be applied.
- **recon_loss** (*String*) – Definition of Reconstruction-Loss-Method, 'mse' or 'nb'.
- **use_l_encoder** (*Boolean*) – If True and `decoder_last_layer`='softmax', libary size encoder is used.
- **use_bn** (*Boolean*) – If *True* batch normalization will be applied to layers.
- **use_ln** (*Boolean*) – If *True* layer normalization will be applied to layers.
- **mask** (*Array or List*) – if not None, an array of 0s and 1s from utils.add_annotations to create VAE with a masked linear decoder.
- **mask_key** (*String*) – A key in *adata.varm* for the mask if the mask is not provided.
- **decoder_last_layer** (*String or None*) – The last layer of the decoder. Must be 'softmax' (default for 'nb' loss), identity(default for 'mse' loss), 'softplus', 'exp' or 'relu'.
- **soft_mask** (*Boolean*) – Use soft mask option. If True, the model will enforce mask with L1 regularization instead of multipling weight of the linear decoder by the binary mask.
- **n_ext** (*Integer*) – Number of unconstarined extension terms. Used for query mapping.
- **n_ext_m** (*Integer*) – Number of constrained extension terms. Used for query mapping.
- **use_hsic** (*Boolean*) – If True, add HSIC regularization for unconstarined extension terms. Used for query mapping.
- **hsic_one_vs_all** (*Boolean*) – If True, calculates the sum of HSIC losses for each unconstarined term vs the other terms. If False, calculates HSIC for all unconstarined terms vs the other terms. Used for query mapping.
- **ext_mask** (*Array or List*) – Mask (similar to the mask argument) for unconstarined extension terms. Used for query mapping.
- **soft_ext_mask** (*Boolean*) – Use the soft mask mode for training with the constarined extension terms. Used for query mapping.

Methods

get_latent([x, c, only_active, mean, mean_var])	Map *x* in to the latent space.
get_y([x, c])	Map *x* in to the latent space.
latent_directions([method, get_confidence, ...])	Get directions of upregulation for each latent dimension.
latent_enrich(groups[, comparison, ...])	Gene set enrichment test for the latent space.
load(dir_path[, adata, map_location])	Instantiate a model from the saved output. :param dir_path: Path to saved outputs. :param adata: AnnData object. If None, will check for and load anndata saved with the model. :param map_location: a function, torch.device, string or a dict specifying how to remap storage locations.
load_query_data(adata, reference_model[, ...])	Transfer Learning function for new data.
mask_genes([terms])	Return lists of genes belonging to the terms in the mask.
nonzero_terms()	Return indices of active terms.
save(dir_path[, overwrite, save_anndata])	Save the state of the model. Neither the trainer optimizer state nor the trainer history are saved. :param dir_path: Path to a directory. :param overwrite: Overwrite existing data or not. If *False* and directory already exists at *dir_path*, error will be raised. :param save_anndata: If True, also saves the anndata :param anndata_write_kwargs: Kwargs for anndata write function.
term_genes(term[, terms])	Return the dataframe with genes belonging to the term after training sorted by absolute weights in the decoder.
train([n_epochs, lr, eps, alpha, omega])	Train the model.
update_terms([terms, adata])	Add extension terms' names to the terms.

get_latent(*x: ndarray | None = None, c: ndarray | None = None, only_active: bool = False, mean: bool = False, mean_var: bool = False*)

Map *x* in to the latent space. This function will feed data in encoder and return z for each sample in data.

Parameters

- **x** – Numpy nd-array to be mapped to latent space. *x* has to be in shape [n_obs, input_dim]. If None, then *self.adata.X* is used.
- **c** – *numpy nd-array* of original (unencoded) desired labels for each sample.
- **only_active** – Return only the latent variables which correspond to active terms, i.e terms that were not deactivated by the group lasso regularization.
- **mean** – return mean instead of random sample from the latent space
- **mean_var** – return mean and variance instead of random sample from the latent space if *mean=False*.

Return type

Returns array containing latent space encoding of 'x'.

latent_directions(*method='sum', get_confidence=False, adata=None, key_added='directions'*)

Get directions of upregulation for each latent dimension. Multipling this by raw latent scores ensures positive latent scores correspond to upregulation.

Parameters

- **method** (*String*) – Method of calculation, it should be 'sum' or 'counts'.
- **get_confidence** (*Boolean*) – Only for method='counts'. If 'True', also calculate confidence of the directions.
- **adata** (*AnnData*) – An AnnData object to store dimensions. If 'None', self.adata is used.
- **key_added** (*String*) – key of adata.uns where to put the dimensions.

latent_enrich(*groups*, *comparison='rest'*, *n_sample=5000*, *use_directions=False*, *directions_key='directions'*, *select_terms=None*, *adata=None*, *exact=True*, *key_added='bf_scores'*)

Gene set enrichment test for the latent space. Test the hypothesis that latent scores for each term in one group (z_1) is bigger than in the other group (z_2).

Puts results to *adata.uns[key_added]*. Results are a dictionary with *p_h0* - probability that z_1 > z_2, *p_h1 = 1-p_h0* and *bf* - bayes factors equal to *log(p_h0/p_h1)*.

Parameters

- **groups** (*String or Dict*) – A string with the key in *adata.obs* to look for categories or a dictionary with categories as keys and lists of cell names as values.
- **comparison** (*String*) – The category name to compare against. If 'rest', then compares each category against all others.
- **n_sample** (*Integer*) – Number of random samples to draw for each category.
- **use_directions** (*Boolean*) – If 'True', multiplies the latent scores by directions in *adata*.
- **directions_key** (*String*) – The key in *adata.uns* for directions.
- **select_terms** (*Array*) – If not 'None', then an index of terms to select for the test. Only does the test for these terms.
- **adata** (*AnnData*) – An AnnData object to use. If 'None', uses *self.adata*.
- **exact** (*Boolean*) – Use exact probabilities for comparisons.
- **key_added** (*String*) – key of adata.uns where to put the results of the test.

classmethod load_query_data(*adata: AnnData*, *reference_model: str* | TRVAE, *freeze: bool = True*, *freeze_expression: bool = True*, *unfreeze_ext: bool = True*, *remove_dropout: bool = True*, *new_n_ext: int* | *None = None*, *new_n_ext_m: int* | *None = None*, *new_ext_mask: ndarray* | *list* | *None = None*, *new_soft_ext_mask: bool = False*, ***kwargs*)

Transfer Learning function for new data. Uses old trained model and expands it for new conditions.

Parameters

- **adata** – Query anndata object.
- **reference_model** – A model to expand or a path to a model folder.
- **freeze** (*Boolean*) – If 'True' freezes every part of the network except the first layers of encoder/decoder.
- **freeze_expression** (*Boolean*) – If 'True' freeze every weight in first layers except the condition weights.
- **remove_dropout** (*Boolean*) – If 'True' remove Dropout for Transfer Learning.

- **unfreeze_ext** (*Boolean*) – If 'True' do not freeze weights for new constrained and unconstrained extension terms.
- **new_n_ext** (*Integer*) – Number of new unconstarined extension terms to add to the reference model. Used for query mapping.
- **new_n_ext_m** (*Integer*) – Number of new constrained extension terms to add to the reference model. Used for query mapping.
- **new_ext_mask** (*Array or List*) – Mask (similar to the mask argument) for new unconstarined extension terms.
- **new_soft_ext_mask** (*Boolean*) – Use the soft mask mode for training with the constarined extension terms.
- **kwargs** – kwargs for the initialization of the EXPIMAP class for the query model.

Returns
New (query) model to train on query data.

Return type
new_model

mask_genes(*terms: str | list = 'terms'*)

Return lists of genes belonging to the terms in the mask.

nonzero_terms()

Return indices of active terms. Active terms are the terms which were not deactivated by the group lasso regularization.

term_genes(*term: str | int, terms: str | list = 'terms'*)

Return the dataframe with genes belonging to the term after training sorted by absolute weights in the decoder.

train(*n_epochs: int = 400, lr: float = 0.001, eps: float = 0.01, alpha: float | None = None, omega: Tensor | None = None, **kwargs*)

Train the model.

Parameters

- **n_epochs** (*Integer*) – Number of epochs for training the model.
- **lr** (*Float*) – Learning rate for training the model.
- **eps** (*Float*) – torch.optim.Adam eps parameter
- **alpha_kl** (*Float*) – Multiplies the KL divergence part of the loss. Set to 0.35 by default.
- **alpha_epoch_anneal** (*Integer or None*) – If not 'None', the KL Loss scaling factor (alpha_kl) will be annealed from 0 to 1 every epoch until the input integer is reached. By default is set to 130 epochs or to n_epochs if n_epochs < 130.
- **alpha** (*Float*) – Group Lasso regularization coefficient
- **omega** (*Tensor or None*) – If not 'None', vector of coefficients for each group
- **alpha_l1** (*Float*) – L1 regularization coefficient for the soft mask of reference (old) and new constrained terms. Specifies the strength for deactivating the genes which are not in the corresponding annotations groups in the mask.
- **alpha_l1_epoch_anneal** (*Integer*) – If not 'None', the alpha_l1 scaling factor will be annealed from 0 to 1 every 'alpha_l1_anneal_each' epochs until the input integer is reached.

- **alpha_l1_anneal_each** (*Integer*) – Anneal alpha_l1 every alpha_l1_anneal_each'th epoch, i.e. for 5 (default) do annealing every 5th epoch.
- **gamma_ext** (*Float*) – L1 regularization coefficient for the new unconstrained terms. Specifies the strength of sparcity enforcement.
- **gamma_epoch_anneal** (*Integer*) – If not 'None', the gamma_ext scaling factor will be annealed from 0 to 1 every 'gamma_anneal_each' epochs until the input integer is reached.
- **gamma_anneal_each** (*Integer*) – Anneal gamma_ext every gamma_anneal_each'th epoch, i.e. for 5 (default) do annealing every 5th epoch.
- **beta** (*Float*) – HSIC regularization coefficient for the unconstrained terms. Multiplies the HSIC loss terms if not 'None'.
- **kwargs** – kwargs for the expiMap trainer.

update_terms(*terms: str | list = 'terms', adata=None*)

Add extension terms' names to the terms.

3.2.3 scPoli

class scarches.models.**scPoli**(*adata: AnnData, share_metadata: bool | None = True, obs_metadata: DataFrame | None = None, condition_keys: list | str | None = None, conditions: list | None = None, conditions_combined: list | None = None, inject_condition: list | None = ['encoder', 'decoder'], cell_type_keys: list | str | None = None, cell_types: dict | None = None, unknown_ct_names: list | None = None, labeled_indices: list | None = None, prototypes_labeled: dict | None = None, prototypes_unlabeled: dict | None = None, hidden_layer_sizes: list | None = None, latent_dim: int = 10, embedding_dims: list | int = 10, embedding_max_norm: float = 1.0, dr_rate: float = 0.05, use_mmd: bool = False, mmd_on: str = 'z', mmd_boundary: int | None = None, recon_loss: str | None = 'nb', beta: float = 1, use_bn: bool = False, use_ln: bool = True*)

Bases: BaseMixin

Model for scPoli class. This class contains the methods and functionalities for label transfer and prototype training.

Parameters

- **adata** (: *~anndata.AnnData*) – Annotated data matrix.
- **share_metadata** (*Bool*) – Whether or not to share metadata associated with samples. The metadata is aggregated using the condition_keys. First element is taken. Consider manually adding an .obs_metadata attribute if you need more flexibility.
- **condition_keys** (*String*) – column name of conditions in *adata.obs* data frame.
- **conditions** (*List*) – List of Condition names that the used data will contain to get the right encoding when used after reloading.
- **cell_type_keys** (*List or str*) – List or string of obs columns to use as cell type annotation for prototypes.
- **cell_types** (*Dictionary*) – Dictionary of cell types. Keys are cell types and values are cell_type_keys. Needed for surgery.
- **unknown_ct_names** (*List*) – List of strings with the names of cell clusters to be ignored for prototypes computation.

- **labeled_indices** (*List*) – List of integers with the indices of the labeled cells.
- **prototypes_labeled** (*Dictionary*) – Dictionary with keys mean, cov and the respective mean or covariate matrices for prototypes.
- **prototypes_unlabeled** (*Dictionary*) – Dictionary with keys mean and the respective mean for unlabeled prototypes.
- **hidden_layer_sizes** (*List*) – A list of hidden layer sizes for encoder network. Decoder network will be the reversed order.
- **latent_dim** (*Integer*) – Bottleneck layer (z) size.
- **embedding_dim** (*Integer*) – Conditional embedding size.
- **embedding_max_norm** – Max norm allowed for conditional embeddings.
- **dr_rate** (*Float*) – Dropput rate applied to all layers, if `dr_rate`==0 no dropout will be applied.
- **use_mmd** (*Boolean*) – If 'True' an additional MMD loss will be calculated on the latent dim. 'z' or the first decoder layer 'y'.
- **mmd_on** (*String*) – Choose on which layer MMD loss will be calculated on if 'use_mmd=True': 'z' for latent dim or 'y' for first decoder layer.
- **mmd_boundary** (*Integer or None*) – Choose on how many conditions the MMD loss should be calculated on. If 'None' MMD will be calculated on all conditions.
- **recon_loss** (*String*) – Definition of Reconstruction-Loss-Method, 'mse', 'nb' or 'zinb'.
- **beta** (*Float*) – Scaling Factor for MMD loss
- **use_bn** (*Boolean*) – If *True* batch normalization will be applied to layers.
- **use_ln** (*Boolean*) – If *True* layer normalization will be applied to layers.

Methods

add_new_cell_type(cell_type_name, obs_key, ...)	Function used to add new annotation for a novel cell type.
classify(adata[, prototype, p, get_prob, ...])	Classifies unlabeled cells using the prototypes obtained during training.
get_conditional_embeddings()	Returns anndata object of the conditional embeddings
get_latent(adata[, mean])	Map *x* in to the latent space.
get_prototypes_info([prototype_set])	Generates anndata file with prototype features and annotations.
load(dir_path[, adata, map_location])	Instantiate a model from the saved output. :param dir_path: Path to saved outputs. :param adata: AnnData object. If None, will check for and load anndata saved with the model. :param map_location: a function, torch.device, string or a dict specifying how to remap storage locations.
load_query_data(adata, reference_model[, ...])	Transfer Learning function for new data.
save(dir_path[, overwrite, save_anndata])	Save the state of the model. Neither the trainer optimizer state nor the trainer history are saved. :param dir_path: Path to a directory. :param overwrite: Overwrite existing data or not. If *False* and directory already exists at *dir_path*, error will be raised. :param save_anndata: If True, also saves the anndata :param anndata_write_kwargs: Kwargs for anndata write function.
train([n_epochs, pretraining_epochs, eta, ...])	Train the model.

add_new_cell_type(*cell_type_name*, *obs_key*, *prototypes*, *x=None*, *c=None*)

Function used to add new annotation for a novel cell type.

Parameters

- **cell_type_name** (*str*) – Name of the new cell type
- **obs_key** (*str*) – Obs column key to define the hierarchy level of celltype annotation.
- **prototypes** (*list*) – List of indices of the unlabeled prototypes that correspond to the new cell type
- **x** (*np.ndarray*) – Features to be classified. If None the stored model's adata is used.
- **c** (*np.ndarray*) – Condition vector. If None the stored model's condition vector is used.

classify(*adata*, *prototype=False*, *p=2*, *get_prob=False*, *log_distance=True*, *scale_uncertainties=False*)

Classifies unlabeled cells using the prototypes obtained during training. Data handling before call to model's classify method.

x: np.ndarray
Features to be classified. If None the stored model's adata is used.

c: Dict or np.ndarray
Condition vector, or dictionary when the model is conditioned on multiple batch covariates.

prototype:
Boolean whether to classify the gene features or prototypes stored stored in the model.

get_conditional_embeddings()

Returns anndata object of the conditional embeddings

get_latent(*adata*, *mean: bool = False*)

Map *x* in to the latent space. This function will feed data in encoder and return z for each sample in data.

Parameters

x

Numpy nd-array to be mapped to latent space. *x* has to be in shape [n_obs, input_dim].

c

numpy nd-array of original (unencoded) desired labels for each sample.

mean

return mean instead of random sample from the latent space

:rtype: Returns array containing latent space encoding of 'x'.

get_prototypes_info(*prototype_set='labeled'*)

Generates anndata file with prototype features and annotations.

Parameters

- **cell_type_name** (*str*) – Name of the new cell type
- **prototypes** (*list*) – List of indices of the unlabeled prototypes that correspond to the new cell type

classmethod load_query_data(*adata: AnnData*, *reference_model: str | SCPOLI*, *labeled_indices: list | None = None*, *unknown_ct_names: list | None = None*, *freeze: bool = True*, *freeze_expression: bool = True*, *remove_dropout: bool = True*)

Transfer Learning function for new data. Uses old trained model and expands it for new conditions.

Parameters

- **adata** – Query anndata object.
- **reference_model** – SCPOLI model to expand or a path to SCPOLI model folder.
- **freeze** (*Boolean*) – If 'True' freezes every part of the network except the first layers of encoder/decoder.
- **freeze_expression** (*Boolean*) – If 'True' freeze every weight in first layers except the condition weights.
- **remove_dropout** (*Boolean*) – If 'True' remove Dropout for Transfer Learning.

Returns

new_model – New SCPOLI model to train on query data.

Return type

scPoli

train(*n_epochs: int = 100*, *pretraining_epochs=None*, *eta: float = 1*, *lr: float = 0.001*, *eps: float = 0.01*, *alpha_epoch_anneal=100.0*, *reload_best: bool = False*, *prototype_training: bool | None = True*, *unlabeled_prototype_training: bool | None = True*, ***kwargs*)

Train the model.

Parameters

- **n_epochs** – Number of epochs for training the model.

- **lr** – Learning rate for training the model.
- **eps** – torch.optim.Adam eps parameter
- **kwargs** – kwargs for the scPoli trainer.

3.2.4 scVI

class `scarches.models.SCVI`(*adata: AnnData, n_hidden: int = 128, n_latent: int = 10, n_layers: int = 1, dropout_rate: float = 0.1, dispersion: Literal['gene', 'gene-batch', 'gene-label', 'gene-cell'] = 'gene', gene_likelihood: Literal['zinb', 'nb', 'poisson'] = 'zinb', latent_distribution: Literal['normal', 'ln'] = 'normal', **model_kwargs*)

Bases: `RNASeqMixin`, `VAEMixin`, `ArchesMixin`, `UnsupervisedTrainingMixin`, `BaseMinifiedModeModelClass`

single-cell Variational Inference **:cite:p:`Lopez18`**.

Parameters

- **adata** – AnnData object that has been registered via `setup_anndata()`.
- **n_hidden** – Number of nodes per hidden layer.
- **n_latent** – Dimensionality of the latent space.
- **n_layers** – Number of hidden layers used for encoder and decoder NNs.
- **dropout_rate** – Dropout rate for neural networks.
- **dispersion** – One of the following:
  - `'gene'` - dispersion parameter of NB is constant per gene across cells
  - `'gene-batch'` - dispersion can differ between different batches
  - `'gene-label'` - dispersion can differ between different labels
  - `'gene-cell'` - dispersion can differ for every gene in every cell
- **gene_likelihood** – One of:
  - `'nb'` - Negative binomial distribution
  - `'zinb'` - Zero-inflated negative binomial distribution
  - `'poisson'` - Poisson distribution
- **latent_distribution** – One of:
  - `'normal'` - Normal distribution
  - `'ln'` - Logistic normal distribution (Normal(0, I) transformed by softmax)
- ****model_kwargs** – Keyword args for `VAE`

Examples

```
>>> adata = anndata.read_h5ad(path_to_anndata)
>>> scvi.model.SCVI.setup_anndata(adata, batch_key="batch")
>>> vae = scvi.model.SCVI(adata)
>>> vae.train()
>>> adata.obsm["X_scVI"] = vae.get_latent_representation()
>>> adata.obsm["X_normalized_scVI"] = vae.get_normalized_expression()
```

Notes

See further usage examples in the following tutorials:

1. /tutorials/notebooks/api_overview
2. /tutorials/notebooks/harmonization
3. /tutorials/notebooks/scarches_scvi_tools
4. /tutorials/notebooks/scvi_in_R

Attributes

`adata`
Data attached to model instance.

`adata_manager`
Manager instance associated with self.adata.

`device`
The current device that the module's params are on.

`history`
Returns computed metrics during training.

`is_trained`
Whether the model has been trained.

`minified_data_type`
The type of minified data associated with this model, if applicable.

`test_indices`
Observations that are in test set.

`train_indices`
Observations that are in train set.

`validation_indices`
Observations that are in validation set.

Methods

`convert_legacy_save`(dir_path, output_dir_path)	Converts a legacy saved model (<v0.15.0) to the updated save format.
`differential_expression`([adata, groupby, ...])	.
`get_anndata_manager`(adata[, required])	Retrieves the `AnnDataManager` for a given AnnData object specific to this model instance.
`get_elbo`([adata, indices, batch_size])	Return the ELBO for the data.
`get_feature_correlation_matrix`([adata, ...])	Generate gene-gene correlation matrix using scvi uncertainty and expression.
`get_from_registry`(adata, registry_key)	Returns the object in AnnData associated with the key in the data registry.
`get_latent_library_size`([adata, indices, ...])	Returns the latent library size for each cell.
`get_latent_representation`([adata, indices, ...])	Return the latent representation for each cell.
`get_likelihood_parameters`([adata, indices, ...])	Estimates for the parameters of the likelihood $p(x \mid z)$.
`get_marginal_ll`([adata, indices, ...])	Return the marginal LL for the data.
`get_normalized_expression`([adata, indices, ...])	Returns the normalized (decoded) gene expression.
`get_reconstruction_error`([adata, indices, ...])	Return the reconstruction error for the data.
`load`(dir_path[, adata, use_gpu, prefix, ...])	Instantiate a model from the saved output.
`load_query_data`(adata, reference_model[, ...])	Online update of a reference model with scArches algorithm **:cite:p:`Lotfollahi21`**.
`load_registry`(dir_path[, prefix])	Return the full registry saved with the model.
`minify_adata`([minified_data_type, ...])	Minifies the model's adata.
`posterior_predictive_sample`([adata, ...])	Generate observation samples from the posterior predictive distribution.
`prepare_query_anndata`(adata, reference_model)	Prepare data for query integration.
`register_manager`(adata_manager)	Registers an `AnnDataManager` instance with this model class.
`save`(dir_path[, prefix, overwrite, save_anndata])	Save the state of the model.
`setup_anndata`(adata[, layer, batch_key, ...])	Sets up the `AnnData` object for this model.
`to_device`(device)	Move model to device.
`train`([max_epochs, use_gpu, train_size, ...])	Train the model.
`view_anndata_setup`([adata, ...])	Print summary of the setup for the initial AnnData or a given AnnData object.
`view_setup_args`(dir_path[, prefix])	Print args used to setup a saved model.

`minify_adata`(*minified_data_type: Literal['latent_posterior_parameters'] = 'latent_posterior_parameters', use_latent_qzm_key: str = 'X_latent_qzm', use_latent_qzv_key: str = 'X_latent_qzv'*) → None

Minifies the model's adata.

Minifies the adata, and registers new anndata fields: latent qzm, latent qzv, adata uns containing minified-adata type, and library size. This also sets the appropriate property on the module to indicate that the adata is minified.

Parameters

- **`minified_data_type`** – How to minify the data. Currently only supports *latent_posterior_parameters*. If minified_data_type == *latent_posterior_parameters*:
  - the original count data is removed (*adata.X*, adata.raw, and any layers)
  - the parameters of the latent representation of the original data is stored
  - everything else is left untouched

- **use_latent_qzm_key** – Key to use in *adata.obsm* where the latent qzm params are stored
- **use_latent_qzv_key** – Key to use in *adata.obsm* where the latent qzv params are stored

Notes

The modification is not done inplace – instead the model is assigned a new (minified) version of the adata.

classmethod setup_anndata(*adata: AnnData, layer: str | None = None, batch_key: str | None = None, labels_key: str | None = None, size_factor_key: str | None = None, categorical_covariate_keys: List[str] | None = None, continuous_covariate_keys: List[str] | None = None, **kwargs*)

Sets up the `AnnData` object for this model.

A mapping will be created between data fields used by this model to their respective locations in adata. None of the data in adata are modified. Only adds fields to adata.

Parameters

- **adata** – AnnData object. Rows represent cells, columns represent features.
- **layer** – if not *None*, uses this as the key in *adata.layers* for raw count data.
- **batch_key** – key in *adata.obs* for batch information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_batch']*. If *None*, assigns the same batch to all the data.
- **labels_key** – key in *adata.obs* for label information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_labels']*. If *None*, assigns the same label to all the data.
- **size_factor_key** – key in *adata.obs* for size factor information. Instead of using library size as a size factor, the provided size factor column will be used as offset in the mean of the likelihood. Assumed to be on linear scale.
- **categorical_covariate_keys** – keys in *adata.obs* that correspond to categorical data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.
- **continuous_covariate_keys** – keys in *adata.obs* that correspond to continuous data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.

3.2.5 scANVI

class scarches.models.**SCANVI**(*adata: AnnData, n_hidden: int = 128, n_latent: int = 10, n_layers: int = 1, dropout_rate: float = 0.1, dispersion: Literal['gene', 'gene-batch', 'gene-label', 'gene-cell'] = 'gene', gene_likelihood: Literal['zinb', 'nb', 'poisson'] = 'zinb', **model_kwargs*)

Bases: `RNASeqMixin`, `VAEMixin`, `ArchesMixin`, `BaseMinifiedModeModelClass`

Single-cell annotation using variational inference **:cite:p:`Xu21`**.

Inspired from M1 + M2 model, as described in (https://arxiv.org/pdf/1406.5298.pdf).

Parameters

- **adata** – AnnData object that has been registered via `setup_anndata()`.
- **n_hidden** – Number of nodes per hidden layer.
- **n_latent** – Dimensionality of the latent space.
- **n_layers** – Number of hidden layers used for encoder and decoder NNs.
- **dropout_rate** – Dropout rate for neural networks.
- **dispersion** – One of the following:
  - `'gene'` - dispersion parameter of NB is constant per gene across cells
  - `'gene-batch'` - dispersion can differ between different batches
  - `'gene-label'` - dispersion can differ between different labels
  - `'gene-cell'` - dispersion can differ for every gene in every cell
- **gene_likelihood** – One of:
  - `'nb'` - Negative binomial distribution
  - `'zinb'` - Zero-inflated negative binomial distribution
  - `'poisson'` - Poisson distribution
- ****model_kwargs** – Keyword args for `SCANVAE`

Examples

```
>>> adata = anndata.read_h5ad(path_to_anndata)
>>> scvi.model.SCANVI.setup_anndata(adata, batch_key="batch", labels_key="labels")
>>> vae = scvi.model.SCANVI(adata, "Unknown")
>>> vae.train()
>>> adata.obsm["X_scVI"] = vae.get_latent_representation()
>>> adata.obs["pred_label"] = vae.predict()
```

Notes

See further usage examples in the following tutorials:

1. /tutorials/notebooks/harmonization
2. /tutorials/notebooks/scarches_scvi_tools
3. /tutorials/notebooks/seed_labeling

Attributes

adata
Data attached to model instance.

adata_manager
Manager instance associated with self.adata.

device
The current device that the module's params are on.

history
Returns computed metrics during training.

is_trained
Whether the model has been trained.

minified_data_type
The type of minified data associated with this model, if applicable.

test_indices
Observations that are in test set.

train_indices
Observations that are in train set.

validation_indices
Observations that are in validation set.

Methods

`convert_legacy_save`(dir_path, output_dir_path)	Converts a legacy saved model (<v0.15.0) to the updated save format.
`differential_expression`([adata, groupby, ...])	.
`from_scvi_model`(scvi_model, unlabeled_category)	Initialize scanVI model with weights from pretrained `SCVI` model.
`get_anndata_manager`(adata[, required])	Retrieves the `AnnDataManager` for a given AnnData object specific to this model instance.
`get_elbo`([adata, indices, batch_size])	Return the ELBO for the data.
`get_feature_correlation_matrix`([adata, ...])	Generate gene-gene correlation matrix using scvi uncertainty and expression.
`get_from_registry`(adata, registry_key)	Returns the object in AnnData associated with the key in the data registry.
`get_latent_library_size`([adata, indices, ...])	Returns the latent library size for each cell.
`get_latent_representation`([adata, indices, ...])	Return the latent representation for each cell.
`get_likelihood_parameters`([adata, indices, ...])	Estimates for the parameters of the likelihood $p(x \mid z)$.
`get_marginal_ll`([adata, indices, ...])	Return the marginal LL for the data.
`get_normalized_expression`([adata, indices, ...])	Returns the normalized (decoded) gene expression.
`get_reconstruction_error`([adata, indices, ...])	Return the reconstruction error for the data.
`load`(dir_path[, adata, use_gpu, prefix, ...])	Instantiate a model from the saved output.
`load_query_data`(adata, reference_model[, ...])	Online update of a reference model with scArches algorithm **:cite:p:`Lotfollahi21`**.
`load_registry`(dir_path[, prefix])	Return the full registry saved with the model.
`minify_adata`([minified_data_type, ...])	Minifies the model's adata.
`posterior_predictive_sample`([adata, ...])	Generate observation samples from the posterior predictive distribution.
`predict`([adata, indices, soft, batch_size])	Return cell label predictions.
`prepare_query_anndata`(adata, reference_model)	Prepare data for query integration.
`register_manager`(adata_manager)	Registers an `AnnDataManager` instance with this model class.
`save`(dir_path[, prefix, overwrite, save_anndata])	Save the state of the model.
`setup_anndata`(adata, labels_key, ...[, ...])	Sets up the `AnnData` object for this model.
`to_device`(device)	Move model to device.
`train`([max_epochs, n_samples_per_label, ...])	Train the model.
`view_anndata_setup`([adata, ...])	Print summary of the setup for the initial AnnData or a given AnnData object.
`view_setup_args`(dir_path[, prefix])	Print args used to setup a saved model.

classmethod from_scvi_model(*scvi_model: SCVI, unlabeled_category: str, labels_key: str | None = None, adata: AnnData | None = None, **scanvi_kwargs*)

Initialize scanVI model with weights from pretrained `SCVI` model.

Parameters

- **scvi_model** – Pretrained scvi model
- **labels_key** – key in *adata.obs* for label information. Label categories can not be different if labels_key was used to setup the SCVI model. If None, uses the *labels_key* used to setup the SCVI model. If that was None, and error is raised.
- **unlabeled_category** – Value used for unlabeled cells in *labels_key* used to setup AnnData with scvi.

- **adata** – AnnData object that has been registered via `setup_anndata()`.
- **scanvi_kwargs** – kwargs for scANVI model

minify_adata(*minified_data_type: Literal['latent_posterior_parameters'] = 'latent_posterior_parameters', use_latent_qzm_key: str = 'X_latent_qzm', use_latent_qzv_key: str = 'X_latent_qzv'*)

Minifies the model's adata.

Minifies the adata, and registers new anndata fields: latent qzm, latent qzv, adata uns containing minified-adata type, and library size. This also sets the appropriate property on the module to indicate that the adata is minified.

Parameters

- **minified_data_type** – How to minify the data. Currently only supports *latent_posterior_parameters*. If minified_data_type == *latent_posterior_parameters*:
  - the original count data is removed (*adata.X*, adata.raw, and any layers)
  - the parameters of the latent representation of the original data is stored
  - everything else is left untouched
- **use_latent_qzm_key** – Key to use in *adata.obsm* where the latent qzm params are stored
- **use_latent_qzv_key** – Key to use in *adata.obsm* where the latent qzv params are stored

Notes

The modification is not done inplace – instead the model is assigned a new (minified) version of the adata.

predict(*adata: AnnData | None = None, indices: Sequence[int] | None = None, soft: bool = False, batch_size: int | None = None*) → ndarray | DataFrame

Return cell label predictions.

Parameters

- **adata** – AnnData object that has been registered via `setup_anndata()`.
- **indices** – Return probabilities for each class label.
- **soft** – If True, returns per class probabilities
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.

classmethod setup_anndata(*adata: AnnData, labels_key: str, unlabeled_category: str | int | float, layer: str | None = None, batch_key: str | None = None, size_factor_key: str | None = None, categorical_covariate_keys: List[str] | None = None, continuous_covariate_keys: List[str] | None = None, **kwargs*)

Sets up the `AnnData` object for this model.

A mapping will be created between data fields used by this model to their respective locations in adata. None of the data in adata are modified. Only adds fields to adata.

Parameters

- **adata** – AnnData object. Rows represent cells, columns represent features.
- **layer** – if not *None*, uses this as the key in *adata.layers* for raw count data.

- **batch_key** – key in *adata.obs* for batch information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_batch']*. If *None*, assigns the same batch to all the data.
- **labels_key** – key in *adata.obs* for label information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_labels']*. If *None*, assigns the same label to all the data.
- **size_factor_key** – key in *adata.obs* for size factor information. Instead of using library size as a size factor, the provided size factor column will be used as offset in the mean of the likelihood. Assumed to be on linear scale.
- **categorical_covariate_keys** – keys in *adata.obs* that correspond to categorical data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.
- **continuous_covariate_keys** – keys in *adata.obs* that correspond to continuous data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.

train(*max_epochs: int | None = None, n_samples_per_label: float | None = None, check_val_every_n_epoch: int | None = None, train_size: float = 0.9, validation_size: float | None = None, batch_size: int = 128, use_gpu: str | int | bool | None = None, plan_kwargs: dict | None = None, **trainer_kwargs*)

Train the model.

Parameters

- **max_epochs** – Number of passes through the dataset for semisupervised training.
- **n_samples_per_label** – Number of subsamples for each label class to sample per epoch. By default, there is no label subsampling.
- **check_val_every_n_epoch** – Frequency with which metrics are computed on the data for validation set for both the unsupervised and semisupervised trainers. If you'd like a different frequency for the semisupervised trainer, set check_val_every_n_epoch in semisupervised_train_kwargs.
- **train_size** – Size of training set in the range [0.0, 1.0].
- **validation_size** – Size of the test set. If *None*, defaults to 1 - *train_size*. If *train_size + validation_size < 1*, the remaining cells belong to a test set.
- **batch_size** – Minibatch size to use during training.
- **use_gpu** – Use default GPU if available (if None or True), or index of GPU to use (if int), or name of GPU (if str, e.g., *'cuda:0'*), or use CPU (if False).
- **plan_kwargs** – Keyword args for `SemiSupervisedTrainingPlan`. Keyword arguments passed to *train()* will overwrite values present in *plan_kwargs*, when appropriate.
- ****trainer_kwargs** – Other keyword args for `Trainer`.

3.2.6 TotalVI

class scarches.models.**TOTALVI**(*adata: AnnData, n_latent: int = 20, gene_dispersion: Literal['gene', 'gene-batch', 'gene-label', 'gene-cell'] = 'gene', protein_dispersion: Literal['protein', 'protein-batch', 'protein-label'] = 'protein', gene_likelihood: Literal['zinb', 'nb'] = 'nb', latent_distribution: Literal['normal', 'ln'] = 'normal', empirical_protein_background_prior: bool | None = None, override_missing_proteins: bool = False, **model_kwargs*)

Bases: `RNASeqMixin`, `VAEMixin`, `ArchesMixin`, `BaseModelClass`

total Variational Inference **:cite:p:`GayosoSteier21`**.

Parameters

- **adata** – AnnData object that has been registered via `setup_anndata()`.
- **n_latent** – Dimensionality of the latent space.
- **gene_dispersion** – One of the following:
  - `'gene'` - genes_dispersion parameter of NB is constant per gene across cells
  - `'gene-batch'` - genes_dispersion can differ between different batches
  - `'gene-label'` - genes_dispersion can differ between different labels
- **protein_dispersion** – One of the following:
  - `'protein'` - protein_dispersion parameter is constant per protein across cells
  - `'protein-batch'` - protein_dispersion can differ between different batches NOT TESTED
  - `'protein-label'` - protein_dispersion can differ between different labels NOT TESTED
- **gene_likelihood** – One of:
  - `'nb'` - Negative binomial distribution
  - `'zinb'` - Zero-inflated negative binomial distribution
- **latent_distribution** – One of:
  - `'normal'` - Normal distribution
  - `'ln'` - Logistic normal distribution (Normal(0, I) transformed by softmax)
- **empirical_protein_background_prior** – Set the initialization of protein background prior empirically. This option fits a GMM for each of 100 cells per batch and averages the distributions. Note that even with this option set to *True*, this only initializes a parameter that is learned during inference. If *False*, randomly initializes. The default (*None*), sets this to *True* if greater than 10 proteins are used.
- **override_missing_proteins** – If *True*, will not treat proteins with all 0 expression in a particular batch as missing.
- ****model_kwargs** – Keyword args for `TOTALVAE`

Examples

```
>>> adata = anndata.read_h5ad(path_to_anndata)
>>> scvi.model.TOTALVI.setup_anndata(adata, batch_key="batch", protein_expression_
↪obsm_key="protein_expression")
>>> vae = scvi.model.TOTALVI(adata)
>>> vae.train()
>>> adata.obsm["X_totalVI"] = vae.get_latent_representation()
```

Notes

See further usage examples in the following tutorials:

1. /tutorials/notebooks/totalVI
2. /tutorials/notebooks/cite_scrna_integration_w_totalVI
3. /tutorials/notebooks/scarches_scvi_tools

Attributes

`adata`
Data attached to model instance.

`adata_manager`
Manager instance associated with self.adata.

`device`
The current device that the module's params are on.

`history`
Returns computed metrics during training.

`is_trained`
Whether the model has been trained.

`test_indices`
Observations that are in test set.

`train_indices`
Observations that are in train set.

`validation_indices`
Observations that are in validation set.

Methods

`convert_legacy_save`(dir_path, output_dir_path)	Converts a legacy saved model (<v0.15.0) to the updated save format.
differential_expression([adata, groupby, ...])	.
`get_anndata_manager`(adata[, required])	Retrieves the `AnnDataManager` for a given AnnData object specific to this model instance.
`get_elbo`([adata, indices, batch_size])	Return the ELBO for the data.
get_feature_correlation_matrix([adata, ...])	Generate gene-gene correlation matrix using scvi uncertainty and expression.
`get_from_registry`(adata, registry_key)	Returns the object in AnnData associated with the key in the data registry.
get_latent_library_size([adata, indices, ...])	Returns the latent library size for each cell.
`get_latent_representation`([adata, indices, ...])	Return the latent representation for each cell.
get_likelihood_parameters([adata, indices, ...])	Estimates for the parameters of the likelihood $p(x, y \mid z)$.
`get_marginal_ll`([adata, indices, ...])	Return the marginal LL for the data.
get_normalized_expression([adata, indices, ...])	Returns the normalized gene expression and protein expression.
get_protein_background_mean(adata, indices, ...)	Get protein background mean.
get_protein_foreground_probability([adata, ...])	Returns the foreground probability for proteins.
`get_reconstruction_error`([adata, indices, ...])	Return the reconstruction error for the data.
`load`(dir_path[, adata, use_gpu, prefix, ...])	Instantiate a model from the saved output.
`load_query_data`(adata, reference_model[, ...])	Online update of a reference model with scArches algorithm **:cite:p:`Lotfollahi21`**.
`load_registry`(dir_path[, prefix])	Return the full registry saved with the model.
posterior_predictive_sample([adata, ...])	Generate observation samples from the posterior predictive distribution.
`prepare_query_anndata`(adata, reference_model)	Prepare data for query integration.
`register_manager`(adata_manager)	Registers an `AnnDataManager` instance with this model class.
`save`(dir_path[, prefix, overwrite, save_anndata])	Save the state of the model.
setup_anndata(adata, protein_expression_obsm_key)	Sets up the `AnnData` object for this model.
setup_mudata(mdata[, rna_layer, ...])	Sets up the `MuData` object for this model.
`to_device`(device)	Move model to device.
train([max_epochs, lr, use_gpu, train_size, ...])	Trains the model using amortized variational inference.
`view_anndata_setup`([adata, ...])	Print summary of the setup for the initial AnnData or a given AnnData object.
`view_setup_args`(dir_path[, prefix])	Print args used to setup a saved model.

differential_expression(*adata: AnnData | None = None, groupby: str | None = None, group1: Iterable[str] | None = None, group2: str | None = None, idx1: Sequence[int] | Sequence[bool] | str | None = None, idx2: Sequence[int] | Sequence[bool] | str | None = None, mode: Literal['vanilla', 'change'] = 'change', delta: float = 0.25, batch_size: int | None = None, all_stats: bool = True, batch_correction: bool = False, batchid1: Iterable[str] | None = None, batchid2: Iterable[str] | None = None, fdr_target: float = 0.05, silent: bool = False, protein_prior_count: float = 0.1, scale_protein: bool = False, sample_protein_mixing: bool = False, include_protein_background: bool = False, **kwargs*) → DataFrame

.

A unified method for differential expression analysis.

Implements *"vanilla"* DE **:cite:p:`Lopez18`**. and *"change"* mode DE **:cite:p:`Boyeau19`**.

Parameters

adata
AnnData object with equivalent structure to initial AnnData. If None, defaults to the AnnData object used to initialize the model.

groupby
The key of the observations grouping to consider.

group1
Subset of groups, e.g. [*'g1'*, *'g2'*, *'g3'*], to which comparison shall be restricted, or all groups in *groupby* (default).

group2
If *None*, compare each group in *group1* to the union of the rest of the groups in *groupby*. If a group identifier, compare with respect to this group.

idx1
idx1 and *idx2* can be used as an alternative to the AnnData keys. Custom identifier for *group1* that can be of three sorts: (1) a boolean mask, (2) indices, or (3) a string. If it is a string, then it will query indices that verifies conditions on *adata.obs*, as described in `pandas.DataFrame.query()` If *idx1* is not *None*, this option overrides *group1* and *group2*.

idx2
Custom identifier for *group2* that has the same properties as *idx1*. By default, includes all cells not specified in *idx1*.

mode
Method for differential expression. See user guide for full explanation.

delta
specific case of region inducing differential expression. In this case, we suppose that $R \setminus [-\delta, \delta]$ does not induce differential expression (change model default case).

batch_size
Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.

all_stats
Concatenate count statistics (e.g., mean expression group 1) to DE results.

batch_correction
Whether to correct for batch effects in DE inference.

batchid1
Subset of categories from *batch_key* registered in `setup_anndata`, e.g. [*'batch1'*, *'batch2'*, *'batch3'*], for *group1*. Only used if *batch_correction* is *True*, and by default all categories are used.

batchid2
Same as *batchid1* for group2. *batchid2* must either have null intersection with *batchid1*, or be exactly equal to *batchid1*. When the two sets are exactly equal, cells are compared by decoding on the same batch. When sets have null intersection, cells from *group1* and *group2* are decoded on each group in *group1* and *group2*, respectively.

fdr_target
Tag features as DE based on posterior expected false discovery rate.

silent
If True, disables the progress bar. Default: False.

protein_prior_count
Prior count added to protein expression before LFC computation

scale_protein
Force protein values to sum to one in every single cell (post-hoc normalization)

sample_protein_mixing
Sample the protein mixture component, i.e., use the parameter to sample a Bernoulli that determines if expression is from foreground/background.

include_protein_background
Include the protein background component as part of the protein expression

****kwargs**
Keyword args for `scvi.model.base.DifferentialComputation.get_bayes_factors()`

Returns

Differential expression DataFrame.

`get_feature_correlation_matrix`(*adata=None*, *indices=None*, *n_samples: int = 10*, *batch_size: int = 64*, *rna_size_factor: int = 1000*, *transform_batch: Sequence[int | float | str] | None = None*, *correlation_type: Literal['spearman', 'pearson'] = 'spearman'*, *log_transform: bool = False*) → DataFrame

Generate gene-gene correlation matrix using scvi uncertainty and expression.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **n_samples** – Number of posterior samples to use for estimation.
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.
- **rna_size_factor** – size factor for RNA prior to sampling gamma distribution
- **transform_batch** – Batches to condition on. If transform_batch is:
  - None, then real observed batch is used
  - int, then batch transform_batch is used
  - list of int, then values are averaged over provided batches.
- **correlation_type** – One of "pearson", "spearman".
- **log_transform** – Whether to log transform denoised values prior to correlation calculation.

Return type
Gene-protein-gene-protein correlation matrix

get_latent_library_size(*adata: AnnData | None = None, indices: Sequence[int] | None = None, give_mean: bool = True, batch_size: int | None = None*) → ndarray

Returns the latent library size for each cell.

This is denoted as ℓ_n in the totalVI paper.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **give_mean** – Return the mean or a sample from the posterior distribution.
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.

get_likelihood_parameters(*adata: AnnData | None = None, indices: Sequence[int] | None = None, n_samples: int | None = 1, give_mean: bool | None = False, batch_size: int | None = None*) → Dict[str, ndarray]

Estimates for the parameters of the likelihood $p(x, y \mid z)$.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **n_samples** – Number of posterior samples to use for estimation.
- **give_mean** – Return expected value of parameters or a samples
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.

get_normalized_expression(*adata=None, indices=None, n_samples_overall: int | None = None, transform_batch: Sequence[int | float | str] | None = None, gene_list: Sequence[str] | None = None, protein_list: Sequence[str] | None = None, library_size: float | Literal['latent'] | None = 1, n_samples: int = 1, sample_protein_mixing: bool = False, scale_protein: bool = False, include_protein_background: bool = False, batch_size: int | None = None, return_mean: bool = True, return_numpy: bool | None = None*) → Tuple[ndarray | DataFrame, ndarray | DataFrame]

Returns the normalized gene expression and protein expression.

This is denoted as ρ_n in the totalVI paper for genes, and TODO for proteins, $(1 - \pi_{nt})\alpha_{nt}\beta_{nt}$.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **n_samples_overall** – Number of samples to use in total
- **transform_batch** – Batch to condition on. If transform_batch is:
  - None, then real observed batch is used
  - int, then batch transform_batch is used

- List[int], then average over batches in list

- **gene_list** – Return frequencies of expression for a subset of genes. This can save memory when working with large datasets and few genes are of interest.
- **protein_list** – Return protein expression for a subset of genes. This can save memory when working with large datasets and few genes are of interest.
- **library_size** – Scale the expression frequencies to a common library size. This allows gene expression levels to be interpreted on a common scale of relevant magnitude.
- **n_samples** – Get sample scale from multiple samples.
- **sample_protein_mixing** – Sample mixing bernoulli, setting background to zero
- **scale_protein** – Make protein expression sum to 1
- **include_protein_background** – Include background component for protein expression
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.
- **return_mean** – Whether to return the mean of the samples.
- **return_numpy** – Return a *np.ndarray* instead of a *pd.DataFrame*. Includes gene names as columns. If either n_samples=1 or return_mean=True, defaults to False. Otherwise, it defaults to True.

Returns

- - **gene_normalized_expression* - normalized expression for RNA*
- - **protein_normalized_expression* - normalized expression for proteins*
- If `n_samples` > 1 and `return_mean` is False, then the shape is `(samples, cells, genes)`.
- Otherwise, shape is `(cells, genes)`. Return type is `pd.DataFrame` unless `return_numpy` is True.

get_protein_background_mean(*adata*, *indices*, *batch_size*)

Get protein background mean.

get_protein_foreground_probability(*adata: AnnData | None = None*, *indices: Sequence[int] | None = None*, *transform_batch: Sequence[int | float | str] | None = None*, *protein_list: Sequence[str] | None = None*, *n_samples: int = 1*, *batch_size: int | None = None*, *return_mean: bool = True*, *return_numpy: bool | None = None*)

Returns the foreground probability for proteins.

This is denoted as $(1 - \pi_{nt})$ in the totalVI paper.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **transform_batch** – Batch to condition on. If transform_batch is:
  - None, then real observed batch is used

- int, then batch transform_batch is used
- List[int], then average over batches in list

- **protein_list** – Return protein expression for a subset of genes. This can save memory when working with large datasets and few genes are of interest.
- **n_samples** – Number of posterior samples to use for estimation.
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.
- **return_mean** – Whether to return the mean of the samples.
- **return_numpy** – Return a `ndarray` instead of a `DataFrame`. DataFrame includes gene names as columns. If either *n_samples=1* or *return_mean=True*, defaults to *False*. Otherwise, it defaults to *True*.

Returns

- *- **foreground_probability** - probability foreground for each protein**
- If *n_samples* > 1 and *return_mean* is False, then the shape is *(samples, cells, genes)*.
- Otherwise, shape is *(cells, genes)*. In this case, return type is `DataFrame` unless *return_numpy* is True.

posterior_predictive_sample(*adata: AnnData | None = None, indices: Sequence[int] | None = None, n_samples: int = 1, batch_size: int | None = None, gene_list: Sequence[str] | None = None, protein_list: Sequence[str] | None = None*) → ndarray

Generate observation samples from the posterior predictive distribution.

The posterior predictive distribution is written as $p(\hat{x}, \hat{y} \mid x, y)$.

Parameters

- **adata** – AnnData object with equivalent structure to initial AnnData. If *None*, defaults to the AnnData object used to initialize the model.
- **indices** – Indices of cells in adata to use. If *None*, all cells are used.
- **n_samples** – Number of required samples for each cell
- **batch_size** – Minibatch size for data loading into model. Defaults to *scvi.settings.batch_size*.
- **gene_list** – Names of genes of interest
- **protein_list** – Names of proteins of interest

Returns

x_new – tensor with shape (n_cells, n_genes, n_samples)

Return type

`ndarray`

classmethod setup_anndata(*adata: AnnData, protein_expression_obsm_key: str, protein_names_uns_key: str | None = None, batch_key: str | None = None, layer: str | None = None, size_factor_key: str | None = None, categorical_covariate_keys: List[str] | None = None, continuous_covariate_keys: List[str] | None = None, **kwargs*)

Sets up the `AnnData` object for this model.

A mapping will be created between data fields used by this model to their respective locations in adata. None of the data in adata are modified. Only adds fields to adata.

Parameters

- **adata** – AnnData object. Rows represent cells, columns represent features.
- **protein_expression_obsm_key** – key in *adata.obsm* for protein expression data.
- **protein_names_uns_key** – key in *adata.uns* for protein names. If None, will use the column names of *adata.obsm[protein_expression_obsm_key]* if it is a DataFrame, else will assign sequential names to proteins.
- **batch_key** – key in *adata.obs* for batch information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_batch']*. If *None*, assigns the same batch to all the data.
- **layer** – if not *None*, uses this as the key in *adata.layers* for raw count data.
- **size_factor_key** – key in *adata.obs* for size factor information. Instead of using library size as a size factor, the provided size factor column will be used as offset in the mean of the likelihood. Assumed to be on linear scale.
- **categorical_covariate_keys** – keys in *adata.obs* that correspond to categorical data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.
- **continuous_covariate_keys** – keys in *adata.obs* that correspond to continuous data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.

Returns

- *None. Adds the following fields*
- *.uns['_scvi']* – *scvi* setup dictionary
- *.obs['_scvi_labels']* – labels encoded as integers
- *.obs['_scvi_batch']* – batch encoded as integers

classmethod setup_mudata(*mdata: MuData, rna_layer: str | None = None, protein_layer: str | None = None, batch_key: str | None = None, size_factor_key: str | None = None, categorical_covariate_keys: List[str] | None = None, continuous_covariate_keys: List[str] | None = None, modalities: Dict[str, str] | None = None, **kwargs*)

Sets up the `MuData` object for this model.

A mapping will be created between data fields used by this model to their respective locations in adata. None of the data in adata are modified. Only adds fields to adata.

Parameters

- **mdata** – MuData object. Rows represent cells, columns represent features.
- **rna_layer** – RNA layer key. If *None*, will use *.X* of specified modality key.
- **protein_layer** – Protein layer key. If *None*, will use *.X* of specified modality key.

- **batch_key** – key in *adata.obs* for batch information. Categories will automatically be converted into integer categories and saved to *adata.obs['_scvi_batch']*. If *None*, assigns the same batch to all the data.
- **size_factor_key** – key in *adata.obs* for size factor information. Instead of using library size as a size factor, the provided size factor column will be used as offset in the mean of the likelihood. Assumed to be on linear scale.
- **categorical_covariate_keys** – keys in *adata.obs* that correspond to categorical data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.
- **continuous_covariate_keys** – keys in *adata.obs* that correspond to continuous data. These covariates can be added in addition to the batch covariate and are also treated as nuisance factors (i.e., the model tries to minimize their effects on the latent space). Thus, these should not be used for biologically-relevant factors that you do _not_ want to correct for.
- **modalities** – Dictionary mapping parameters to modalities.

Examples

```
>>> mdata = muon.read_10x_h5("pbmc_10k_protein_v3_filtered_feature_bc_matrix.h5
↪")
>>> scvi.model.TOTALVI.setup_mudata(mdata, modalities={"rna_layer": "rna":
↪"protein_layer": "prot"})
>>> vae = scvi.model.TOTALVI(mdata)
```

train(*max_epochs: int | None = None, lr: float = 0.004, use_gpu: str | int | bool | None = None, train_size: float = 0.9, validation_size: float | None = None, batch_size: int = 256, early_stopping: bool = True, check_val_every_n_epoch: int | None = None, reduce_lr_on_plateau: bool = True, n_steps_kl_warmup: int | None = None, n_epochs_kl_warmup: int | None = None, adversarial_classifier: bool | None = None, plan_kwargs: dict | None = None, **kwargs*)

Trains the model using amortized variational inference.

Parameters

- **max_epochs** – Number of passes through the dataset.
- **lr** – Learning rate for optimization.
- **use_gpu** – Use default GPU if available (if None or True), or index of GPU to use (if int), or name of GPU (if str, e.g., *'cuda:0'*), or use CPU (if False).
- **train_size** – Size of training set in the range [0.0, 1.0].
- **validation_size** – Size of the test set. If *None*, defaults to 1 - *train_size*. If *train_size + validation_size < 1*, the remaining cells belong to a test set.
- **batch_size** – Minibatch size to use during training.
- **early_stopping** – Whether to perform early stopping with respect to the validation set.
- **check_val_every_n_epoch** – Check val every n train epochs. By default, val is not checked, unless *early_stopping* is *True* or *reduce_lr_on_plateau* is *True*. If either of the latter conditions are met, val is checked every epoch.

- **reduce_lr_on_plateau** – Reduce learning rate on plateau of validation metric (default is ELBO).
- **n_steps_kl_warmup** – Number of training steps (minibatches) to scale weight on KL divergences from 0 to 1. Only activated when *n_epochs_kl_warmup* is set to None. If *None*, defaults to *floor(0.75 * adata.n_obs)*.
- **n_epochs_kl_warmup** – Number of epochs to scale weight on KL divergences from 0 to 1. Overrides *n_steps_kl_warmup* when both are not *None*.
- **adversarial_classifier** – Whether to use adversarial classifier in the latent space. This helps mixing when there are missing proteins in any of the batches. Defaults to *True* is missing proteins are detected.
- **plan_kwargs** – Keyword args for `AdversarialTrainingPlan`. Keyword arguments passed to *train()* will overwrite values present in *plan_kwargs*, when appropriate.
- ****kwargs** – Other keyword args for `Trainer`.

3.3 Plotting

class `scarches.plotting.`**SCVI_EVAL**(*model:* SCVI | SCANVI | TOTALVI, *adata:* *AnnData*, *trainer:* *Trainer* | *None* = *None*, *cell_type_key:* *str* = *None*, *batch_key:* *str* = *None*)

Bases: `object`

Methods

plot_latent([show, save, dir_path, ...])	if save:

get_asw
get_classification_accuracy
get_ebm
get_f1_score
get_knn_purity
get_latent_score
get_model_arch
get_nmi
latent_as_anndata
plot_history

get_asw()

get_classification_accuracy()

get_ebm(*n_neighbors=50*, *n_pools=50*, *n_samples_per_pool=100*, *verbose=True*)

get_f1_score()

get_knn_purity(*n_neighbors=50*, *verbose=True*)

get_latent_score()

get_model_arch()

get_nmi()

latent_as_anndata()

plot_history(*show=True*, *save=False*, *dir_path=None*)

plot_latent(*show=True*, *save=False*, *dir_path=None*, *n_neighbors=8*, *predictions=False*, *in_one=False*, *colors=None*)

if save:

if dir_path is None:
name = 'scanvi_latent.png'

else:
name = f'{dir_path}.png'

else:
name = False

class scarches.plotting.**TRVAE_EVAL**(*model: trVAE* | TRVAE, *adata: AnnData*, *trainer: trVAETrainer* | *None* = *None*, *condition_key: str* | *None* = *None*, *cell_type_key: str* | *None* = *None*)

Bases: object

Methods

get_asw
get_ebm
get_knn_purity
get_latent_score
get_model_arch
get_nmi
latent_as_anndata
plot_history
plot_latent

get_asw()

get_ebm(*n_neighbors=50*, *n_pools=50*, *n_samples_per_pool=100*, *verbose=True*)

get_knn_purity(*n_neighbors=50*, *verbose=True*)

get_latent_score()

get_model_arch()

get_nmi()

latent_as_anndata()

plot_history(*show=True*, *save=False*, *dir_path=None*)

plot_latent(*show=True*, *save=False*, *dir_path=None*, *n_neighbors=8*)

scarches.plotting.**plot_abs_bfs**(*adata*, *scores_key='bf_scores'*, *terms: str | list = 'terms'*, *keys=None*, *n_cols=3*, ***kwargs*)

Plot the absolute bayes scores rankings.

scarches.plotting.**sankey_diagram**(*data*, *save_path=None*, *show=False*, ***kwargs*)

Draws Sankey diagram for the given `data`. :param data: array with 2 columns. One for predictions and another for true values. :type data: `ndarray` :param save_path: Path to save the drawn Sankey diagram. if `None`, the diagram will not be saved. :type save_path: str :param show: if `True` will show the diagram. :type show: bool :param kwargs: additional arguments for diagram configuration. See `_alluvial.plot` function.

3.4 Utils

scarches.utils.**add_annotations**(*adata*, *files*, *min_genes=0*, *max_genes=None*, *varm_key='I'*, *uns_key='terms'*, *clean=True*, *genes_use_upper=True*)

Add annotations to an AnnData object from files.

Parameters

- **adata** – Annotated data matrix.
- **files** – Paths to text files with annotations. The function considers rows to be gene sets with name of a gene set in the first column followed by names of genes.
- **min_genes** – Only include gene sets which have the total number of genes in adata greater than this value.
- **max_genes** – Only include gene sets which have the total number of genes in adata less than this value.
- **varm_key** – Store the binary array I of size n_vars x number of annotated terms in files in *adata.varm[varm_key]*. if I[i,j]=1 then the gene i is present in the annotation j.
- **uns_key** – Sore gene sets' names in *adata.uns[uns_key]*.
- **clean** – If 'True', removes the word before the first underscore for each term name (like '**REACTOME_**') and cuts the name to the first thirty symbols.
- **genes_use_upper** – if 'True', converts genes' names from files and adata to uppercase for comparison.

scarches.utils.**weighted_knn_trainer**(*train_adata*, *train_adata_emb*, *n_neighbors=50*)

Trains a weighted KNN classifier on `train_adata`. :param train_adata: Annotated dataset to be used to train KNN classifier with `label_key` as the target variable. :type train_adata: `AnnData` :param train_adata_emb: Name of the obsm layer to be used for calculation of neighbors. If set to "X", anndata.X will be

used

Parameters

n_neighbors (*int*) – Number of nearest neighbors in KNN classifier.

scarches.utils.**weighted_knn_transfer**(*query_adata*, *query_adata_emb*, *ref_adata_obs*, *label_keys*, *knn_model*, *threshold=1*, *pred_unknown=False*, *mode='package'*)

Annotates `query_adata` cells with an input trained weighted KNN classifier. :param query_adata: Annotated dataset to be used to queryate KNN classifier. Embedding to be used :type query_adata: `AnnData` :param query_adata_emb: Name of the obsm layer to be used for label transfer. If set to "X",

query_adata.X will be used

Parameters

- **ref_adata_obs** (pd.DataFrame) – obs of ref Anndata
- **label_keys** (*str*) – Names of the columns to be used as target variables (e.g. cell_type) in query_adata.
- **knn_model** (KNeighborsTransformer) – knn model trained on reference adata with weighted_knn_trainer function
- **threshold** (*float*) – Threshold of uncertainty used to annotating cells as "Unknown". cells with uncertainties higher than this value will be annotated as "Unknown". Set to 1 to keep all predictions. This enables one to later on play with thresholds.
- **pred_unknown** (*bool*) – False by default. Whether to annotate any cell as "unknown" or not. If *False*, threshold will not be used and each cell will be annotated with the label which is the most common in its n_neighbors nearest cells.
- **mode** (*str*) – Has to be one of "paper" or "package". If mode is set to "package", uncertainties will be 1 - P(pred_label), otherwise it will be 1 - P(true_label).

3.5 Zenodo

- *Deposition helpers*
- *File helpers*

scarches.zenodo.**download_model**(*download_link: str, save_path: str = './', make_dir: bool = False*)

Downloads the zip file of the model in the link and saves it in save_path and extracts.

Parameters

- **link** (*str*) – Direct downloadable link.
- **save_path** (*str*) – Directory path for downloaded file
- **make_dir** (*bool*) – Whether to make the save_path if it does not exist in the system.

Returns

extract_dir – Full path to the folder of the model.

Return type

str

scarches.zenodo.**upload_model**(*model: TRVAE | SCVI | SCANVI | TOTALVI | str, deposition_id: str, access_token: str, model_name: str | None = None*)

Uploads trained model to Zenodo.

Parameters

- **model** (*TRVAE*, *SCVI*, *SCANVI*, *TOTALVI*, str) – An instance of one of classes defined in scarches.models module or a path to a saved model.
- **deposition_id** (*str*) – ID of a deposition in your Zenodo account.
- **access_token** (*str*) – Your Zenodo access token.
- **model_name** (*str*) – An optional name of the model to upload

Returns
download_link – Generated direct download link for the uploaded model in the deposition. Please **Note** that the link is usable **after** your published your deposition.

Return type
str

3.5.1 Deposition helpers

`scarches.zenodo.deposition.create_deposition(access_token: str, upload_type: str, title: str, description: str, **kwargs)`

Creates a deposition in your Zenodo account.

Parameters

- **access_token** (*str*) – Your Zenodo access token.
- **upload_type** (*str*) –
- **title** (*str*) –
- **description** (*str*) –
- **kwargs** –

Returns
deposition_id – ID of the created deposition.

Return type
str

`scarches.zenodo.deposition.delete_deposition(deposition_id: str, access_token: str)`

Deletes the existing deposition with `deposition_id` in your Zenodo account.

Parameters

- **deposition_id** (*str*) – ID of a deposition in your Zenodo account.
- **access_token** (*str*) – Your Zenodo Access token.

`scarches.zenodo.deposition.get_all_deposition_ids(access_token: str)`

Gets list of all of deposition IDs existed in your Zenodo account.

Parameters
access_token (*str*) – Your Zenodo access token.

Returns
deposition_ids – List of deposition IDs.

Return type
list

`scarches.zenodo.deposition.publish_deposition(deposition_id: str, access_token: str)`

Publishes the existing deposition with `deposition_id` in your Zenodo account.

Parameters

- **deposition_id** (*str*) – ID of a deposition in your Zenodo account.
- **access_token** (*str*) – Your Zenodo access token.

Returns

download_link – Generated direct download link for the uploaded model in the deposition. Please **Note** that the link is usable **after** your published your deposition.

Return type

str

`scarches.zenodo.deposition.update_deposition(deposition_id: str, access_token: str, metadata: dict)`

Updates the existing deposition with `deposition_id` in your Zenodo account.

Parameters

- **deposition_id** (*str*) – ID of a deposition in your Zenodo account.
- **access_token** (*str*) – Your Zenodo access token.
- **metadata** (*dict*) –

3.5.2 File Helpers

`scarches.zenodo.file.download_file(link: str, save_path: str | None = None, make_dir: bool = False)`

Downloads the file in the `link` and saves it in `save_path`.

Parameters

- **link** (*str*) – Direct downloadable link.
- **save_path** (*str*) – Path with the name and extension of downloaded file.
- **make_dir** (*bool*) – Whether to make the `save_path` if it does not exist in the system.

Returns

- **file_path** (*str*) – Full path with name and extension of downloaded file.
- **http_response** (`HTTPMessage`) – `HttpMessage` object containing status code and information about the http request.

`scarches.zenodo.file.upload_file(file_path: str, deposition_id: str, access_token: str)`

Downloads the file in the `link` and saves it in `save_path`.

Parameters

- **file_path** (*str*) – Full path with the name and extension of the file you want to upload.
- **deposition_id** (*str*) – ID of a deposition in your Zenodo account.
- **access_token** (*str*) – Your Zenodo Access token.

Returns

- **file_path** (*str*) – Full path with name and extension of downloaded file.
- **http_response** (`HTTPMessage`) – `HttpMessage` object containing status code and information about the http request.

CHAPTER

FOUR

A FEW TIPS ON TRAINING MODELS

trVAE

- We recommend you to set *recon_loss = nb* or *zinb*. These loss functions require access to count data. You need to have raw count data in *adata.raw.X*.
- If you don't have access to count data and have normalized log-transformed data then set *recon_loss* to *mse*.
- trVAE relies on an extra MMD term to force further integration of data sets. There is a parameter called *beta* (default=1) which regulates MMD effect in training. Higher values of *beta* will force extra mixing (might remove biological variation if too big!) while smaller values might result in less mixing (still batch effect). If you set *beta = 0* the model reduces to a Vanilla CVAE, but it is better to set 'use_mmd' to 'False' when MMD should not be used.
- It is important to use highly variable genes for training. We recommend to use at least 2000 HVGs and if you have more complicated datasets, conditions then try to increase it to 5000 or so to include enough information for the model.
- Regarding *architecture* always try with the default one ([128,128], *z_dimension `=10) and check the results. If you have more complicated data sets with many datasets and conditions and etc then you can increase the depth ([128,128,128] or [128,128,128,128]). According to our experiments, small values of `z_dimension* between 10 (default) and 20 are good.

scVI

- scVI require access to raw count data.
- scVI already has a default good parameter the only thing you might change is *n_layers* which we suggest increasing to 2 (min) and max 4-5 for more

complicated datasets.

scANVI

- It requires access to raw count data.
- If you have query data the query data should be treated as unlabelled (Unknown) or have the same set of cell-types labels as reference. If you have a new cell-type label that is in the query data but not in reference and you want to use this in the training query you will get an error! We will fix this in future releases.

expiMap

- The main hyperparameter that affects the quality of integration for the reference training is alpha_kl, the value of which is multiplied by the kl divergence term in the total loss.
- If the visualized latent space looks like a single blob after the reference training, we recommend to decrease the value of alpha_kl. If the visualized latent space shows bad integration quality, we recommend to increase the value of alpha_kl. The good default value in most cases is alpha_kl=0.5.

- The required strength of group lasso regularization (alpha) depends on the number of used GPs and the size of the dataset. For 300–500 GPs, we recommend to use alpha=0.7 and increase for larger numbers of GPs.
- If soft mask in the reference training is used (*soft_ext_mask=True* in the model initialization), it is better to start with *alpha_l1=0.5* (higher value means more constraints on how many genes are added to the gene sets) and use *print_stats=True* in the training for monitoring to check the reported "Share of deactivated inactive genes: __" is around 95% (0.95) at the end and stays so at the final 10 epochs of training. If it is much smaller, *alpha_l1* should be increased by a small value (around 0.05), and if it is 100% (1.) then alpha_l1 should be decreased.
- Using new terms (*n_ext*) in the reference training is not recommended.

CHAPTER

FIVE

UNSUPERVISED SURGERY PIPELINE WITH SCVI

[1]:
```
import os
os.chdir('../')
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
```

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

5.1 Set relevant anndata.obs labels and training length

Here we use the CelSeq2 and SS2 studies as query data and the other 3 studies as reference atlas.

[4]:
```
condition_key = 'study'
cell_type_key = 'cell_type'
target_conditions = ['Pancreas CelSeq2', 'Pancreas SS2']
```

5.2 Download Dataset and split into reference dataset and query dataset

[5]:
```
url = 'https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd'
output = 'pancreas.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd
To: /home/marco/Documents/git_repos/scarches/pancreas.h5ad
126MB [00:03, 32.0MB/s]
```

[5]:
```
'pancreas.h5ad'
```

[6]:
```
adata_all = sc.read('pancreas.h5ad')
```

This line makes sure that count data is in the adata.X. Remember that count data in adata.X is necessary when using "nb" or "zinb" loss.

[7]:
```
adata = adata_all.raw.to_adata()
adata = remove_sparsity(adata)
source_adata = adata[~adata.obs[condition_key].isin(target_conditions)].copy()
target_adata = adata[adata.obs[condition_key].isin(target_conditions)].copy()
```

[8]:
```
source_adata
```

[8]:
```
AnnData object with n_obs × n_vars = 10294 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

[9]:
```
target_adata
```

[9]:
```
AnnData object with n_obs × n_vars = 5387 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

5.3 Create SCVI model and train it on reference dataset

Preprocess reference dataset. Remember that the adata file has to have count data in adata.X for SCVI/SCANVI if not further specified

[10]:
```
sca.models.SCVI.setup_anndata(source_adata, batch_key=condition_key)
```

```
INFO     Using batches from adata.obs["study"]
INFO     No label_key inputted, assuming all cells have same label
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Successfully registered anndata object containing 10294 cells, 1000 vars, 3␣
↪batches,
         1 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
INFO     Please do not further modify adata until model is trained.
```

Create the SCVI model instance with ZINB loss as default. Insert "gene_likelihood='nb'," to change the reconstruction loss to NB loss.

[11]:
```
vae = sca.models.SCVI(
    source_adata,
    n_layers=2,
    encode_covariates=True,
```

(continues on next page)

(continued from previous page)

```
    deeply_inject_covariates=False,
    use_layer_norm="both",
    use_batch_norm="none",
)
```

[12]:
```
vae.train()
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

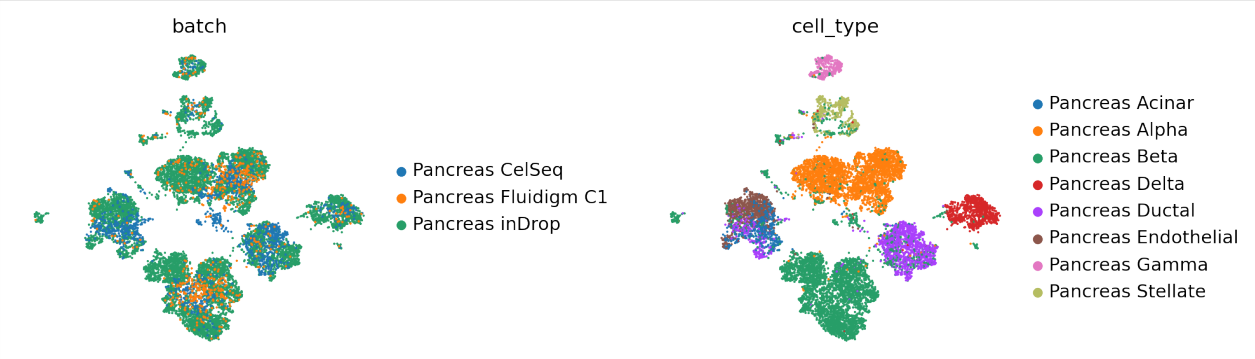
```
Epoch 400/400: 100%|| 400/400 [03:31<00:00,  1.89it/s, loss=502, v_num=1]
```

5.4 Create anndata file of latent representation and compute UMAP

[13]:
```
reference_latent = sc.AnnData(vae.get_latent_representation())
reference_latent.obs["cell_type"] = source_adata.obs[cell_type_key].tolist()
reference_latent.obs["batch"] = source_adata.obs[condition_key].tolist()
```

[14]:
```
sc.pp.neighbors(reference_latent, n_neighbors=8)
sc.tl.leiden(reference_latent)
sc.tl.umap(reference_latent)
sc.pl.umap(reference_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```



After pretraining the model can be saved for later use

[15]:
```
ref_path = 'ref_model/'
vae.save(ref_path, overwrite=True)
```

5.5 Perform surgery on reference model and train on query dataset

[16]:
```
model = sca.models.SCVI.load_query_data(
    target_adata,
    ref_path,
    freeze_dropout = True,
)
```

```
INFO     .obs[_scvi_labels] not found in target, assuming every cell is same category
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels']
INFO     Successfully registered anndata object containing 5387 cells, 1000 vars, 5␣
↪batches,
         1 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
```

[17]:
```
model.train(max_epochs=200, plan_kwargs=dict(weight_decay=0.0))
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

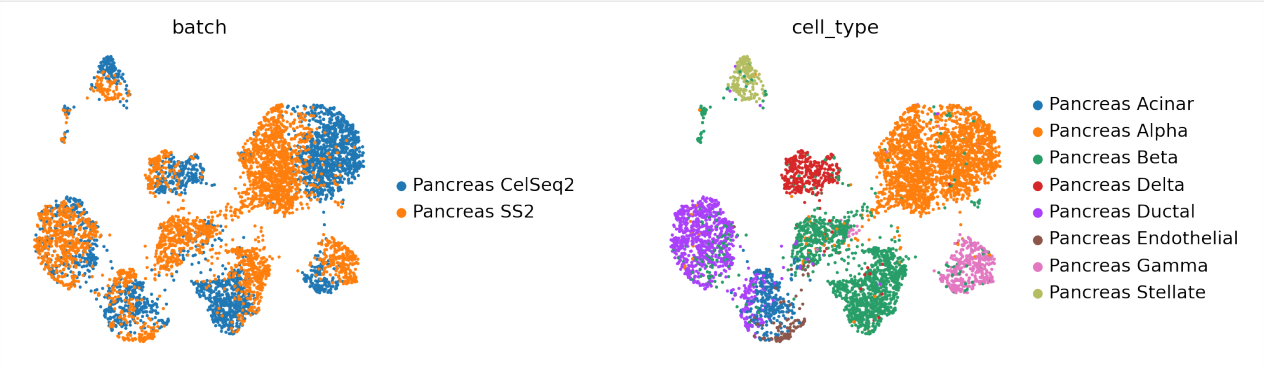
```
Epoch 200/200: 100%|| 200/200 [00:45<00:00,  4.37it/s, loss=1.16e+03, v_num=1]
```

[18]:
```
query_latent = sc.AnnData(model.get_latent_representation())
query_latent.obs['cell_type'] = target_adata.obs[cell_type_key].tolist()
query_latent.obs['batch'] = target_adata.obs[condition_key].tolist()
```

[19]:
```
sc.pp.neighbors(query_latent)
sc.tl.leiden(query_latent)
sc.tl.umap(query_latent)
plt.figure()
sc.pl.umap(
    query_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```

```
<Figure size 320x320 with 0 Axes>
```



[20]:
```
surgery_path = 'surgery_model'
model.save(surgery_path, overwrite=True)
```

5.6 Get latent representation of reference + query dataset and compute UMAP

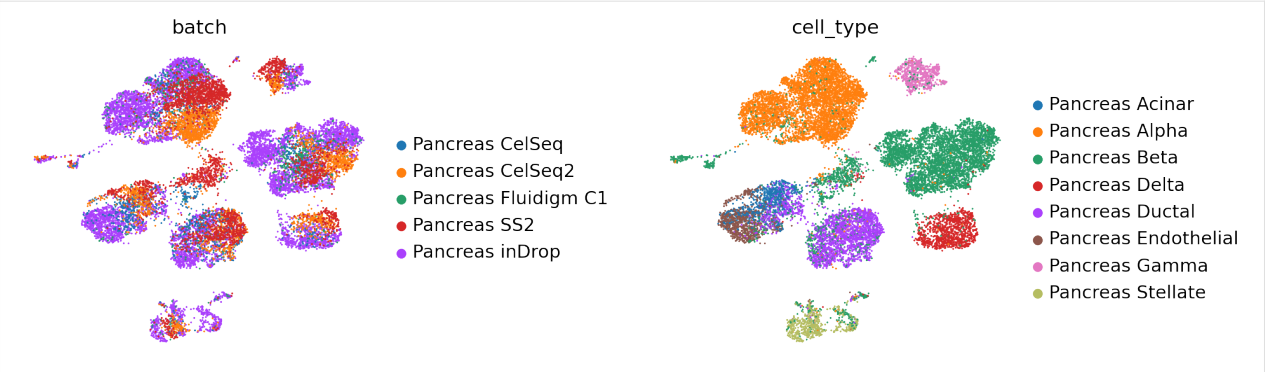
[21]:
```
adata_full = source_adata.concatenate(target_adata)
full_latent = sc.AnnData(model.get_latent_representation(adata=adata_full))
full_latent.obs['cell_type'] = adata_full.obs[cell_type_key].tolist()
full_latent.obs['batch'] = adata_full.obs[condition_key].tolist()
```

```
INFO     Input adata not setup with scvi. attempting to transfer anndata setup
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels']
INFO     Successfully registered anndata object containing 15681 cells, 1000 vars, 5␣
↪batches,
         1 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
```

[22]:
```
sc.pp.neighbors(full_latent)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
plt.figure()
sc.pl.umap(
    full_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```

```
<Figure size 320x320 with 0 Axes>
```


batch
Pancreas CelSeq
Pancreas CelSeq2
Pancreas Fluidigm C1
Pancreas SS2
Pancreas inDrop
cell_type
Pancreas Acinar
Pancreas Alpha
Pancreas Beta
Pancreas Delta
Pancreas Ductal
Pancreas Endothelial
Pancreas Gamma
Pancreas Stellate

[ ]:

CHAPTER

SIX

SEMI-SUPERVISED SURGERY PIPELINE WITH SCANVI

[1]:
```
try:
    from nbproject import header
    header()
except ModuleNotFoundError:
    print("If you want to see the header with dependencies, please install nbproject -␣
↪pip install nbproject")
```

```
<IPython.core.display.HTML object>
```

[2]:
```
import os
os.chdir('../')
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[3]:
```
import scanpy as sc
import torch
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
```

```
Global seed set to 0
```

[4]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

6.1 Set relevant anndata.obs labels and training length

Here we use the CelSeq2 and SS2 studies as query data and the other 3 studies as reference atlas.

[5]:
```
condition_key = 'study'
cell_type_key = 'cell_type'
target_conditions = ['Pancreas CelSeq2', 'Pancreas SS2']
```

6.2 Download Dataset and split into reference dataset and query dataset

[6]:
```
url = 'https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd'
output = 'pancreas.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd
To: C:\Users\sergei.rybakov\Projects\scarches\pancreas.h5ad
100%|| 126M/126M [00:20<00:00, 6.32MB/s]
```

[6]:
```
'pancreas.h5ad'
```

[7]:
```
adata_all = sc.read('pancreas.h5ad')
```

This line makes sure that count data is in the adata.X. Remember that count data in adata.X is necessary when using "nb" or "zinb" loss.

[8]:
```
adata = adata_all.raw.to_adata()
adata = remove_sparsity(adata)
source_adata = adata[~adata.obs[condition_key].isin(target_conditions)].copy()
target_adata = adata[adata.obs[condition_key].isin(target_conditions)].copy()
```

[9]:
```
source_adata
```

[9]:
```
AnnData object with n_obs × n_vars = 10294 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

[10]:
```
target_adata
```

[10]:
```
AnnData object with n_obs × n_vars = 5387 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

6.3 Create SCANVI model and train it on fully labelled reference dataset

Preprocess reference dataset. Remember that the adata file has to have count data in adata.X for SCVI/SCANVI if not further specified

[11]:
```
sca.models.SCVI.setup_anndata(source_adata, batch_key=condition_key, labels_key=cell_
↪type_key)
```

```
INFO     Using batches from adata.obs["study"]
INFO     Using labels from adata.obs["cell_type"]
INFO     Using data from adata.X
INFO     Successfully registered anndata object containing 10294 cells, 1000 vars, 3␣
↪batches,
         8 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
INFO     Please do not further modify adata until model is trained.
```

[12]:
```
vae = sca.models.SCVI(
    source_adata,
    n_layers=2,
    encode_covariates=True,
    deeply_inject_covariates=False,
    use_layer_norm="both",
    use_batch_norm="none",
)
```

[13]:
```
vae.train()
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 400/400: 100%|| 400/400 [14:41<00:00,  2.20s/it, loss=502, v_num=1]
```

Create the SCANVI model instance with ZINB loss as default. Insert "gene_likelihood='nb'," to change the reconstruction loss to NB loss.

[14]:
```
scanvae = sca.models.SCANVI.from_scvi_model(vae, unlabeled_category = "Unknown")
```

[15]:
```
print("Labelled Indices: ", len(scanvae._labeled_indices))
print("Unlabelled Indices: ", len(scanvae._unlabeled_indices))
```

```
Labelled Indices:  10294
Unlabelled Indices:  0
```

[16]:
```
scanvae.train(max_epochs=20)
```

```
INFO     Training for 20 epochs.
```

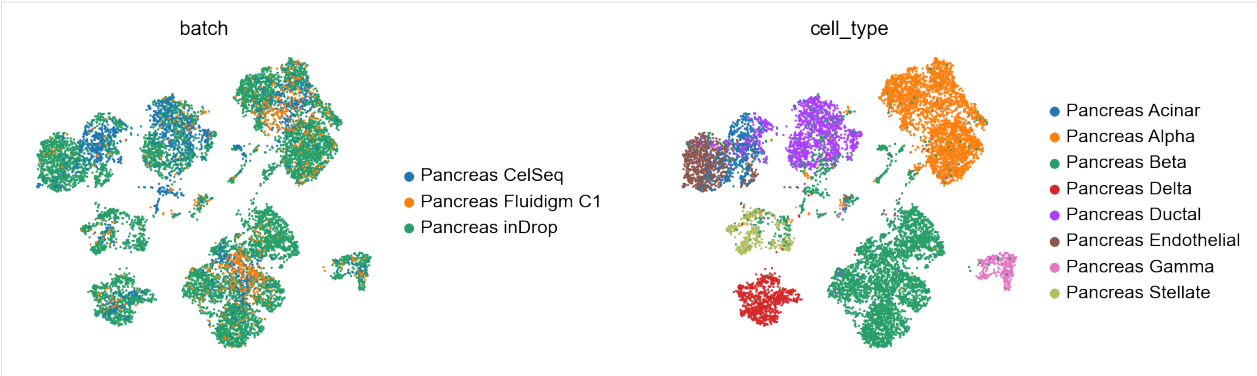
```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 20/20: 100%|| 20/20 [01:38<00:00,  4.90s/it, loss=533, v_num=1]
```

6.4 Create anndata file of latent representation and compute UMAP

[17]:
```
reference_latent = sc.AnnData(scanvae.get_latent_representation())
reference_latent.obs["cell_type"] = source_adata.obs[cell_type_key].tolist()
reference_latent.obs["batch"] = source_adata.obs[condition_key].tolist()
```

[18]:
```
sc.pp.neighbors(reference_latent, n_neighbors=8)
sc.tl.leiden(reference_latent)
sc.tl.umap(reference_latent)
sc.pl.umap(reference_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```



One can also compute the accuracy of the learned classifier

[19]:
```
reference_latent.obs['predictions'] = scanvae.predict()
print("Acc: {}".format(np.mean(reference_latent.obs.predictions == reference_latent.obs.
↪cell_type)))
```

```
Acc: 0.9435593549640567
```

After pretraining the model can be saved for later use

[20]:
```
ref_path = 'ref_model/'
scanvae.save(ref_path, overwrite=True)
```

6.5 Perform surgery on reference model and train on query dataset without cell type labels

If the cell types in 'target_adata' are equal to or a subset of the reference data cell types, one can just pass the adata without further preprocessing. It is also possible then to do semi-supervised training with scArches.

However if there are new cell types in 'target_adata' or if there is no '.obs' in the anndata for cell type labels (e.g. the data is unlabeled), one can only use scANVI in an unsupervised manner during surgery due to the nature of the classifier.

In addition one has to preprocess 'target_adata' in the following way:

If there are new celltypes in there, save the original labels in other column and replace all labels with unlabeled category:

target$_a data.obs['orig_c ell_t ypes'] = target_a data.obs[cell_t ype_k ey].copy()target_a data.obs[cell_t ype_k ey] = scanvae.unlabeled_c ategory$

If there is no '.obs' column for cell types:

target$_a data.obs[cell_t ype_k ey] = scanvae.unlabeled_c ategory$

If 'target_adata' is in the right format, one can proceed with the surgery pipeline. Here we do the surgery unsupervised, but due to the overlapping cell types in query and reference data, one could also do supervised or semi-supervised surgery by setting the indices accordingly.

[21]:
```
model = sca.models.SCANVI.load_query_data(
    target_adata,
    ref_path,
    freeze_dropout = True,
```

(continues on next page)

(continued from previous page)

```
)
model._unlabeled_indices = np.arange(target_adata.n_obs)
model._labeled_indices = []
print("Labelled Indices: ", len(model._labeled_indices))
print("Unlabelled Indices: ", len(model._unlabeled_indices))
```

```
INFO     Using data from adata.X
INFO     Registered keys:['X', 'batch_indices', 'labels']
INFO     Successfully registered anndata object containing 5387 cells, 1000 vars, 5␣
↪batches,
         8 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
Labelled Indices:  0
Unlabelled Indices:  5387
```

[22]:
```
model.train(
    max_epochs=100,
    plan_kwargs=dict(weight_decay=0.0),
    check_val_every_n_epoch=10,
)
```

```
INFO     Training for 100 epochs.
```

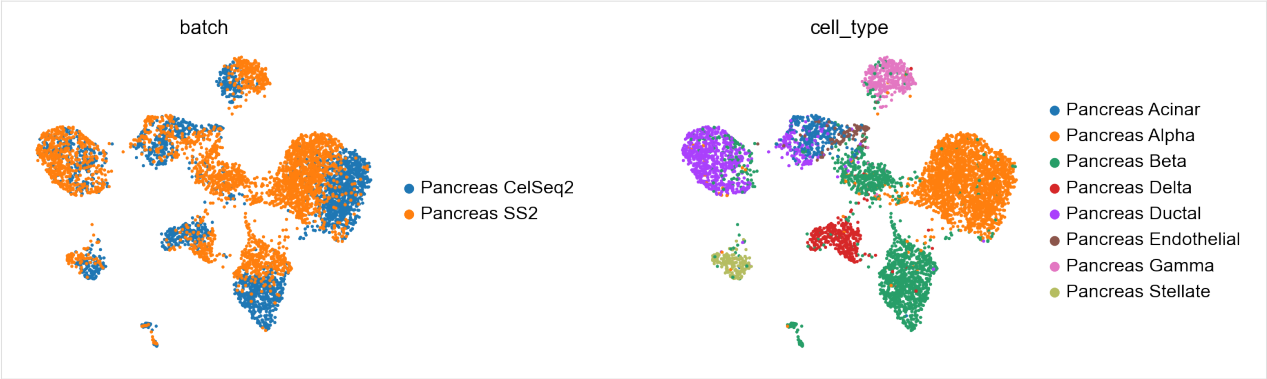
```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 100/100: 100%|| 100/100 [04:23<00:00,  2.63s/it, loss=1.24e+03, v_num=1]
```

[23]:
```
query_latent = sc.AnnData(model.get_latent_representation())
query_latent.obs['cell_type'] = target_adata.obs[cell_type_key].tolist()
query_latent.obs['batch'] = target_adata.obs[condition_key].tolist()
```

[24]:
```
sc.pp.neighbors(query_latent)
sc.tl.leiden(query_latent)
sc.tl.umap(query_latent)
plt.figure()
sc.pl.umap(
    query_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
<Figure size 320x320 with 0 Axes>
```



[25]:
```
surgery_path = 'surgery_model'
model.save(surgery_path, overwrite=True)
```

6.6 Compute Accuracy of model classifier for query dataset and compare predicted and observed cell types

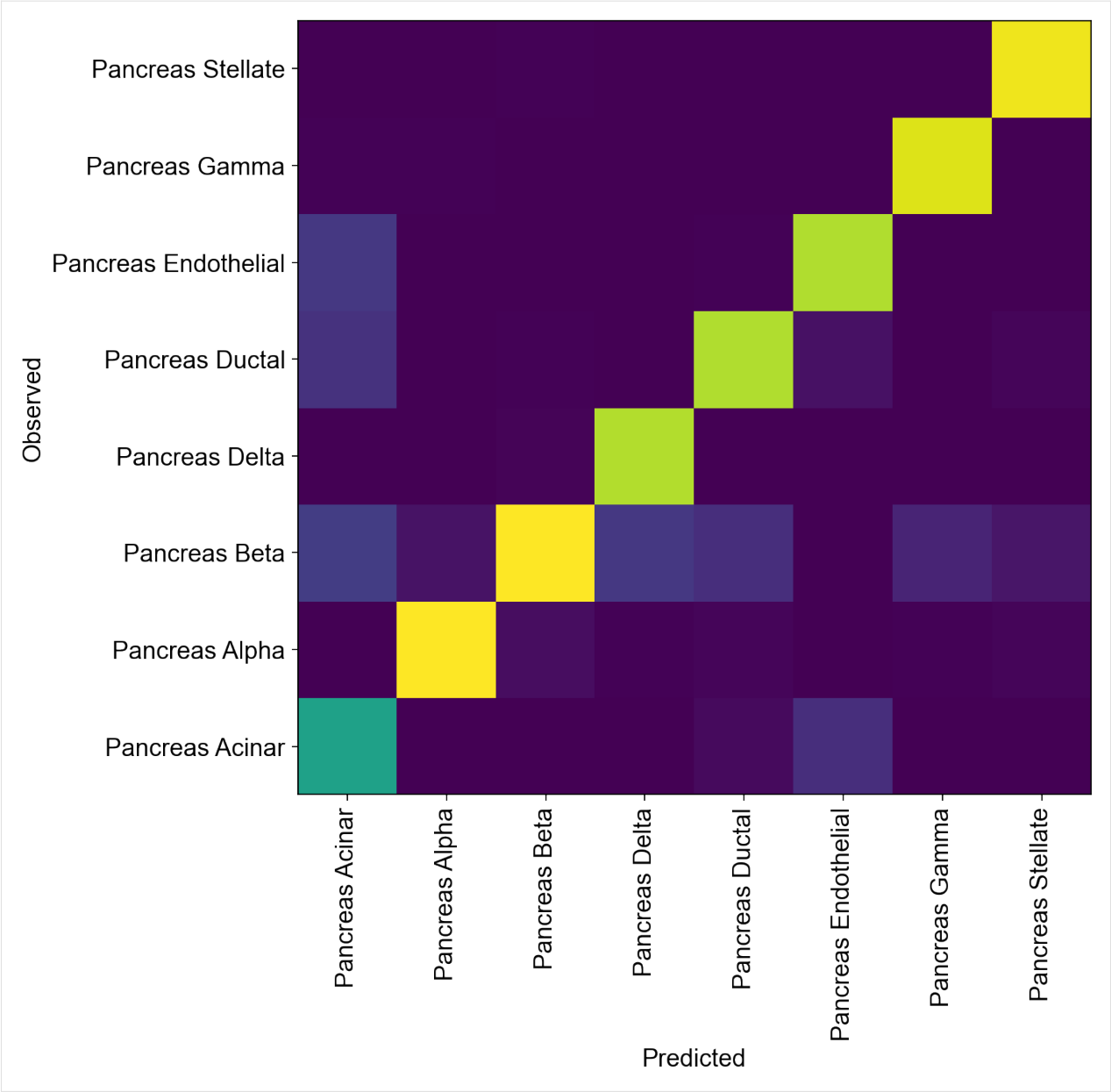
[26]:
```
query_latent.obs['predictions'] = model.predict()
print("Acc: {}".format(np.mean(query_latent.obs.predictions == query_latent.obs.cell_
↪type)))
```

```
Acc: 0.8815667347317616
```

[27]:
```
df = query_latent.obs.groupby(["cell_type", "predictions"]).size().unstack(fill_value=0)
norm_df = df / df.sum(axis=0)

plt.figure(figsize=(8, 8))
_ = plt.pcolor(norm_df)
_ = plt.xticks(np.arange(0.5, len(df.columns), 1), df.columns, rotation=90)
_ = plt.yticks(np.arange(0.5, len(df.index), 1), df.index)
plt.xlabel("Predicted")
plt.ylabel("Observed")
```

```
<ipython-input-27-218c373f617f>:5: MatplotlibDeprecationWarning: Auto-removal of grids␣
↪by pcolor() and pcolormesh() is deprecated since 3.5 and will be removed two minor␣
↪releases later; please call grid(False) first.
  _ = plt.pcolor(norm_df)
```

[27]:
```
Text(0, 0.5, 'Observed')
```



6.7 Get latent representation of reference + query dataset and compute UMAP

[28]:
```
adata_full = source_adata.concatenate(target_adata)
full_latent = sc.AnnData(model.get_latent_representation(adata=adata_full))
full_latent.obs['cell_type'] = adata_full.obs[cell_type_key].tolist()
full_latent.obs['batch'] = adata_full.obs[condition_key].tolist()
```

```
INFO     Input adata not setup with scvi. attempting to transfer anndata setup
INFO     Using data from adata.X
INFO     Registered keys:['X', 'batch_indices', 'labels']
```

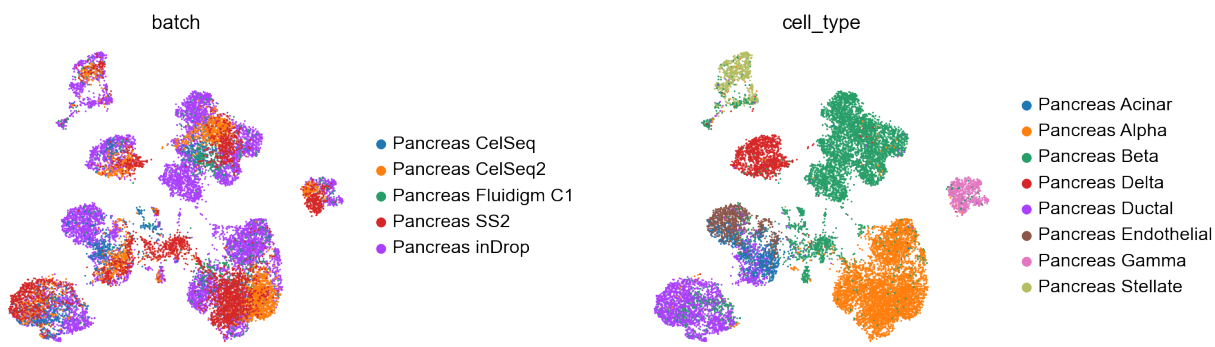
(continues on next page)

(continued from previous page)

```
INFO     Successfully registered anndata object containing 15681 cells, 1000 vars, 5␣
↪batches,
         8 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
```

[29]:
```
sc.pp.neighbors(full_latent)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
plt.figure()
sc.pl.umap(
    full_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
<Figure size 320x320 with 0 Axes>
```



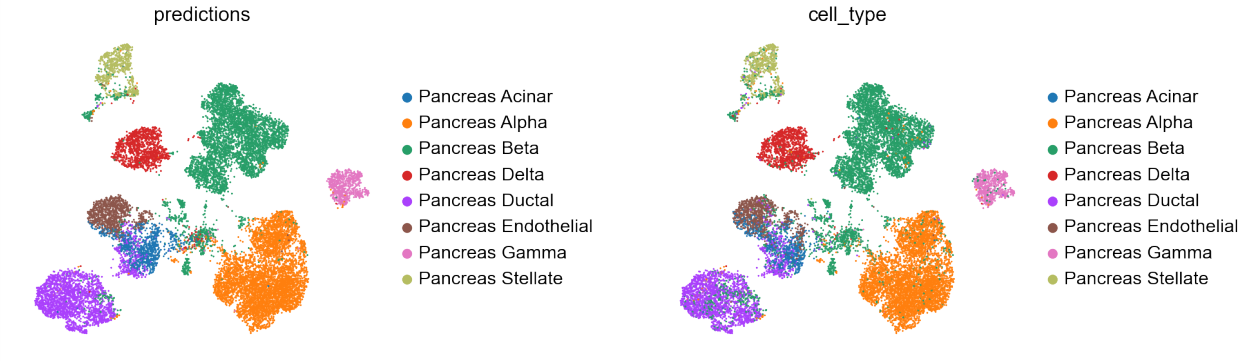
6.8 Comparison of observed and predicted celltypes for reference + query dataset

[30]:
```
full_latent.obs['predictions'] = model.predict(adata=adata_full)
print("Acc: {}".format(np.mean(full_latent.obs.predictions == full_latent.obs.cell_
↪type)))
```

```
Acc: 0.9222626108028825
```

[31]:
```
sc.pp.neighbors(full_latent)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
plt.figure()
sc.pl.umap(
    full_latent,
    color=["predictions", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
<Figure size 320x320 with 0 Axes>
```



CHAPTER

SEVEN

MULTI-MODAL SURGERY PIPELINE WITH TOTALVI

[1]:
```
import os
os.chdir('../')
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import anndata
import torch
import scarches as sca
import matplotlib.pyplot as plt
import numpy as np
import scvi as scv
import pandas as pd
```

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

7.1 Data loading and preprocessing

For totalVI, we will treat two CITE-seq PBMC datasets from 10X Genomics as the reference. These datasets were already filtered for outliers like doublets, as described in the totalVI manuscript. There are 14 proteins in the reference.

[4]:
```
adata_ref = scv.data.pbmcs_10x_cite_seq(run_setup_anndata=False)
```

```
INFO     Downloading file at data/pbmc_10k_protein_v3.h5ad
Downloading...: 24938it [00:00, 31923.71it/s]
INFO     Downloading file at data/pbmc_5k_protein_v3.h5ad
Downloading...: 100%|| 18295/18295.0 [00:05<00:00, 3198.56it/s]
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

[5]:
```
adata_query = scv.data.dataset_10x("pbmc_10k_v3")
adata_query.obs["batch"] = "PBMC 10k (RNA only)"
# put matrix of zeros for protein expression (considered missing)
pro_exp = adata_ref.obsm["protein_expression"]
```

(continues on next page)

(continued from previous page)

```
data = np.zeros((adata_query.n_obs, pro_exp.shape[1]))
adata_query.obsm["protein_expression"] = pd.DataFrame(columns=pro_exp.columns,␣
↪index=adata_query.obs_names, data = data)
```

```
INFO     Downloading file at data/10X/pbmc_10k_v3/filtered_feature_bc_matrix.h5
Downloading...: 37492it [00:02, 13500.06it/s]
```

```
Variable names are not unique. To make them unique, call `.var_names_make_unique`.
Variable names are not unique. To make them unique, call `.var_names_make_unique`.
```

Now to concatenate the objects, which intersects the genes properly.

[6]:
```
adata_full = anndata.concat([adata_ref, adata_query])
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

And split them back up into reference and query (but now genes are properly aligned between objects).

[7]:
```
adata_ref = adata_full[np.logical_or(adata_full.obs.batch == "PBMC5k", adata_full.obs.
↪batch == "PBMC10k")].copy()
adata_query = adata_full[adata_full.obs.batch == "PBMC 10k (RNA only)"].copy()
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

We run gene selection on the reference, because that's all that will be avaialble to us at first.

[8]:
```
sc.pp.highly_variable_genes(
    adata_ref,
    n_top_genes=4000,
    flavor="seurat_v3",
    batch_key="batch",
    subset=True,
)
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

Finally, we use these selected genes for the query dataset as well.

[9]:
```
adata_query = adata_query[:, adata_ref.var_names].copy()
```

7.2 Create TOTALVI model and train it on CITE-seq reference dataset

[10]:
```
sca.models.TOTALVI.setup_anndata(
    adata_ref,
    batch_key="batch",
    protein_expression_obsm_key="protein_expression"
)
```

```
INFO     Using batches from adata.obs["batch"]
INFO     No label_key inputted, assuming all cells have same label
INFO     Using data from adata.X
INFO     Computing library size prior per batch
```

(continues on next page)

(continued from previous page)

```
INFO     Using protein expression from adata.obsm['protein_expression']
INFO     Using protein names from columns of adata.obsm['protein_expression']
INFO     Successfully registered anndata object containing 10849 cells, 4000 vars, 2␣
↪batches,
         1 labels, and 14 proteins. Also registered 0 extra categorical covariates and 0
         extra continuous covariates.
INFO     Please do not further modify adata until model is trained.
```

[11]:
```
arches_params = dict(
    use_layer_norm="both",
    use_batch_norm="none",
)
vae_ref = sca.models.TOTALVI(
    adata_ref,
    **arches_params
)
```

[12]:
```
vae_ref.train()
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

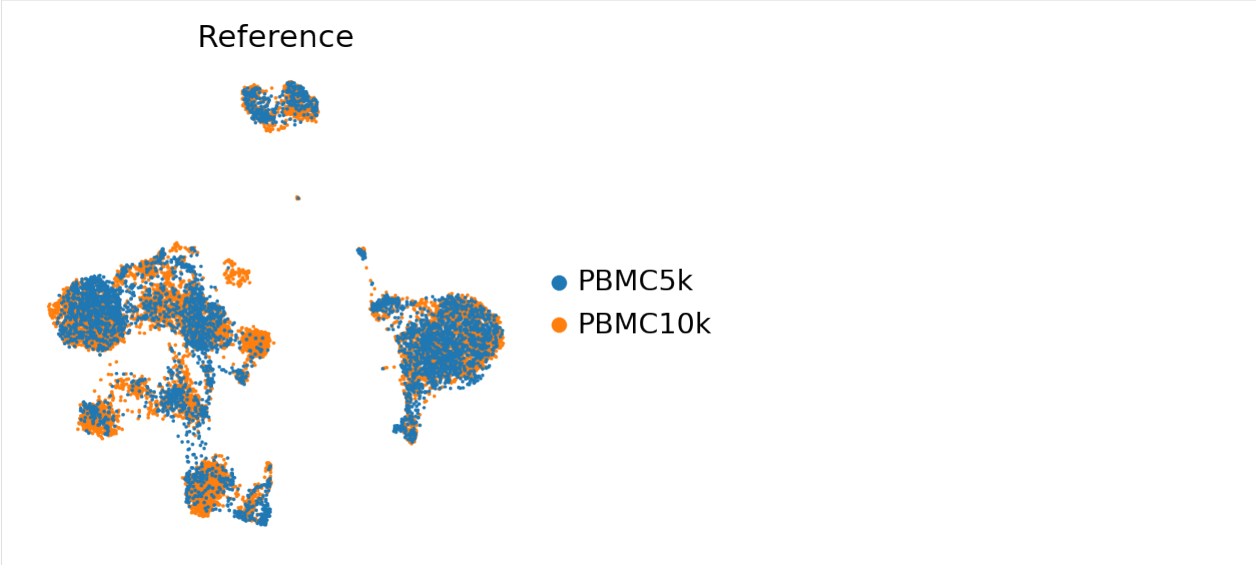
```
Epoch 323/400:  81%|| 323/400 [03:08<00:43,  1.76it/s, loss=1.23e+03, v_num=1]Epoch ␣
↪323: reducing learning rate of group 0 to 2.4000e-03.
Epoch 358/400:  90%|| 358/400 [03:28<00:24,  1.74it/s, loss=1.23e+03, v_num=1]Epoch ␣
↪358: reducing learning rate of group 0 to 1.4400e-03.
Epoch 395/400:  99%|| 395/400 [03:50<00:02,  1.71it/s, loss=1.22e+03, v_num=1]Epoch ␣
↪395: reducing learning rate of group 0 to 8.6400e-04.
Epoch 400/400: 100%|| 400/400 [03:53<00:00,  1.71it/s, loss=1.22e+03, v_num=1]
```

7.3 Save Latent representation and visualize RNA data

[13]:
```
adata_ref.obsm["X_totalVI"] = vae_ref.get_latent_representation()
sc.pp.neighbors(adata_ref, use_rep="X_totalVI")
sc.tl.umap(adata_ref, min_dist=0.4)
```

[14]:
```
sc.pl.umap(
    adata_ref,
    color=["batch"],
    frameon=False,
    ncols=1,
    title="Reference"
)
```

```
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/anndata/_core/anndata.
↪py:1220: FutureWarning: The `inplace` parameter in pandas.Categorical.reorder_
↪categories is deprecated and will be removed in a future version. Removing unused␣
↪categories will always return a new Categorical object.
  c.reorder_categories(natsorted(c.categories), inplace=True)
... storing 'batch' as categorical
```



7.4 Save trained reference model

[15]:
```
dir_path = "saved_model/"
vae_ref.save(dir_path, overwrite=True)
```

7.5 Perform surgery on reference model and train on query dataset without protein data

[16]:
```
vae_q = sca.models.TOTALVI.load_query_data(
    adata_query,
    dir_path,
    freeze_expression=True
)
```

```
INFO     .obs[_scvi_labels] not found in target, assuming every cell is same category
INFO     Found batches with missing protein expression
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels',
         'protein_expression']
INFO     Successfully registered anndata object containing 11769 cells, 4000 vars, 3␣
↪batches,
         1 labels, and 14 proteins. Also registered 0 extra categorical covariates and 0
         extra continuous covariates.
```

```
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/scvi/model/base/_
↪archesmixin.py:96: UserWarning: Query integration should be performed using models␣
↪trained with version >= 0.8
  "Query integration should be performed using models trained with version >= 0.8"
```

[17]:
```
vae_q.train(200, plan_kwargs=dict(weight_decay=0.0))
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 200/200: 100%|| 200/200 [02:56<00:00,  1.13it/s, loss=744, v_num=1]
```

[18]:
```
adata_query.obsm["X_totalVI"] = vae_q.get_latent_representation()
sc.pp.neighbors(adata_query, use_rep="X_totalVI")
sc.tl.umap(adata_query, min_dist=0.4)
```

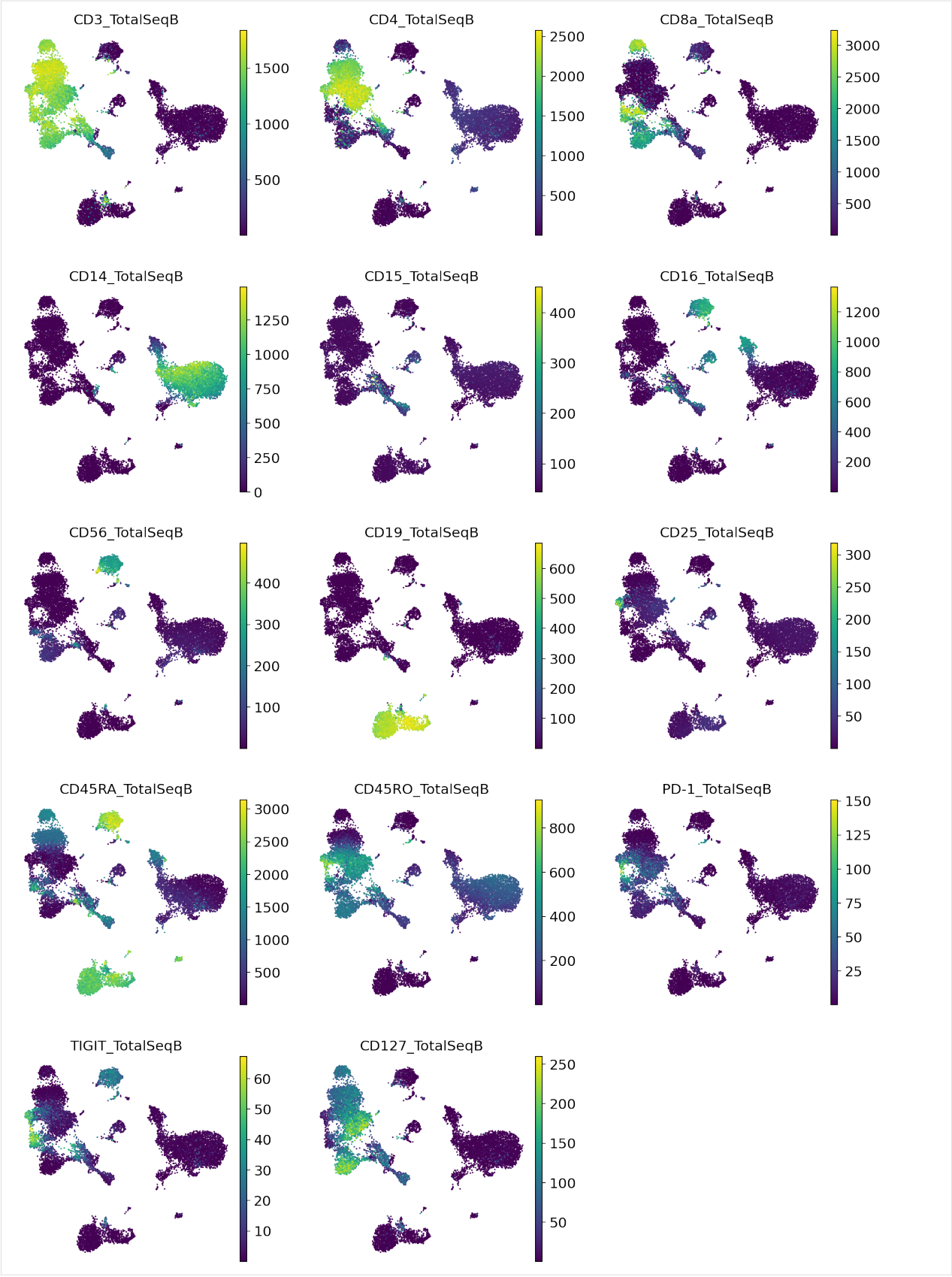
7.6 Impute protein data for the query dataset and visualize

Impute the proteins that were observed in the reference, using the `transform_batch` parameter.

[19]:
```
_, imputed_proteins = vae_q.get_normalized_expression(
    adata_query,
    n_samples=25,
    return_mean=True,
    transform_batch=["PBMC10k", "PBMC5k"],
)
```

[20]:
```
adata_query.obs = pd.concat([adata_query.obs, imputed_proteins], axis=1)

sc.pl.umap(
    adata_query,
    color=imputed_proteins.columns,
    frameon=False,
    ncols=3,
)
```

```
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/anndata/_core/anndata.
↪py:1220: FutureWarning: The `inplace` parameter in pandas.Categorical.reorder_
↪categories is deprecated and will be removed in a future version. Removing unused␣
↪categories will always return a new Categorical object.
  c.reorder_categories(natsorted(c.categories), inplace=True)
... storing 'batch' as categorical
```


CD3_TotalSeqB
1500
1000
500
CD4_TotalSeqB
2500
2000
1500
1000
500
CD8a_TotalSeqB
3000
2500
2000
1500
1000
500
CD14_TotalSeqB
1250
1000
750
500
250
0
CD15_TotalSeqB
400
300
200
100
CD16_TotalSeqB
1200
1000
800
600
400
200
CD56_TotalSeqB
400
300
200
100
CD19_TotalSeqB
600
500
400
300
200
100
CD25_TotalSeqB
300
250
200
150
100
50
CD45RA_TotalSeqB
3000
2500
2000
1500
1000
500
CD45RO_TotalSeqB
800
600
400
200
PD-1_TotalSeqB
150
125
100
75
50
25
TIGIT_TotalSeqB
60
50
40
30
20
10
CD127_TotalSeqB
250
200
150
100
50

7.7 Get latent representation of reference + query dataset and compute UMAP

[21]:
```
adata_full_new = adata_query.concatenate(adata_ref, batch_key="none")
```

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

[22]:
```
adata_full_new.obsm["X_totalVI"] = vae_q.get_latent_representation(adata_full_new)
sc.pp.neighbors(adata_full_new, use_rep="X_totalVI")
sc.tl.umap(adata_full_new, min_dist=0.3)
```

```
INFO     Input adata not setup with scvi. attempting to transfer anndata setup
INFO     Found batches with missing protein expression
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels',
         'protein_expression']
INFO     Successfully registered anndata object containing 22618 cells, 4000 vars, 3␣
↪batches,
         1 labels, and 14 proteins. Also registered 0 extra categorical covariates and 0
         extra continuous covariates.
```

[23]:
```
_, imputed_proteins_all = vae_q.get_normalized_expression(
    adata_full_new,
    n_samples=25,
    return_mean=True,
    transform_batch=["PBMC10k", "PBMC5k"],
)

for i, p in enumerate(imputed_proteins_all.columns):
    adata_full_new.obs[p] = imputed_proteins_all[p].to_numpy().copy()
```

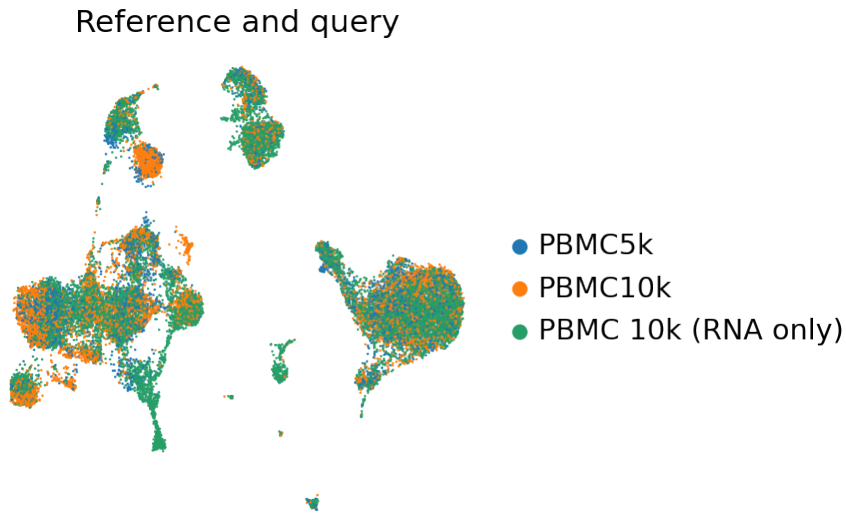
[24]:
```
perm_inds = np.random.permutation(np.arange(adata_full_new.n_obs))
sc.pl.umap(
    adata_full_new[perm_inds],
    color=["batch"],
    frameon=False,
    ncols=1,
    title="Reference and query"
)
```

```
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/anndata/_core/anndata.
↪py:1220: FutureWarning: The `inplace` parameter in pandas.Categorical.reorder_
↪categories is deprecated and will be removed in a future version. Removing unused␣
↪categories will always return a new Categorical object.
  c.reorder_categories(natsorted(c.categories), inplace=True)
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/anndata/_core/anndata.
↪py:1229: ImplicitModificationWarning: Initializing view as actual.
  "Initializing view as actual.", ImplicitModificationWarning
Trying to set attribute `.obs` of view, copying.
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
```

(continues on next page)

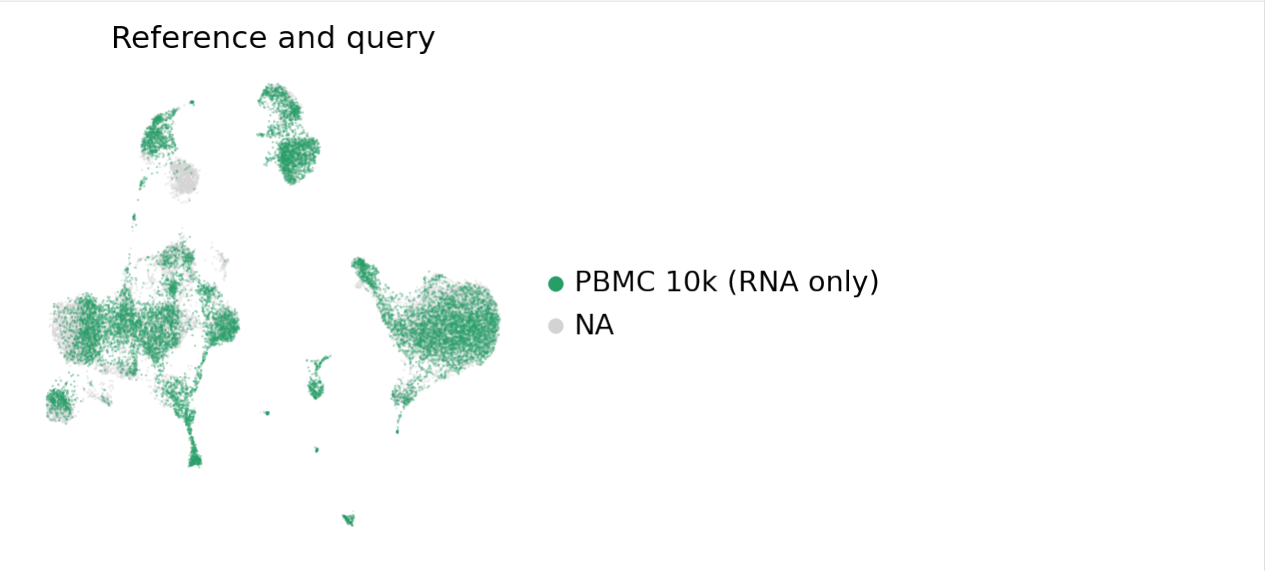
(continued from previous page)

```
Observation names are not unique. To make them unique, call `.obs_names_make_unique`.
... storing 'batch' as categorical
```

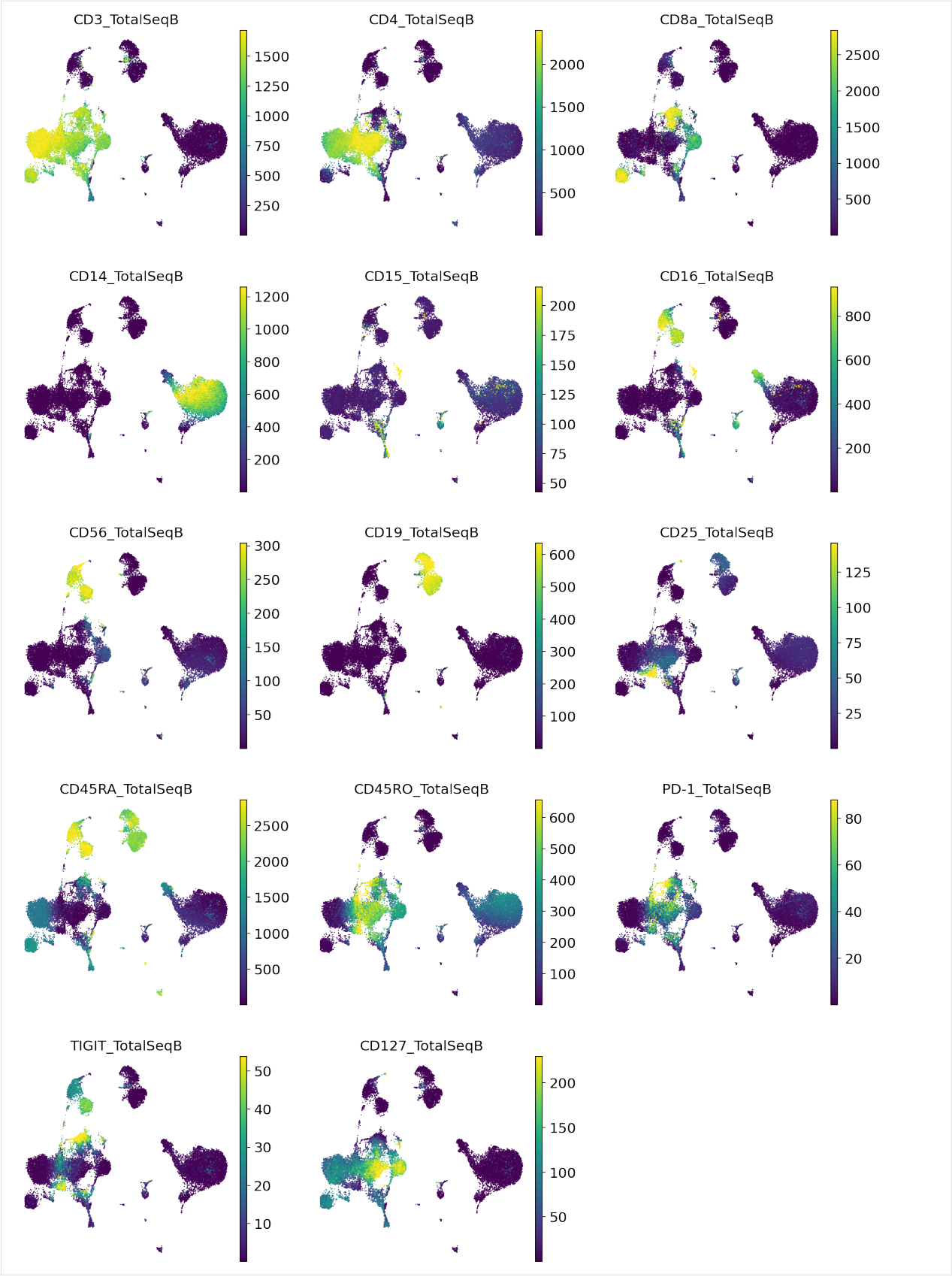


[25]:
```
ax = sc.pl.umap(
    adata_full_new,
    color="batch",
    groups=["PBMC 10k (RNA only)"],
    frameon=False,
    ncols=1,
    title="Reference and query",
    alpha=0.4
)
```

```
/home/marco/.pyenv/versions/scarches/lib/python3.7/site-packages/anndata/_core/anndata.
↪py:1220: FutureWarning: The `inplace` parameter in pandas.Categorical.reorder_
↪categories is deprecated and will be removed in a future version. Removing unused␣
↪categories will always return a new Categorical object.
  c.reorder_categories(natsorted(c.categories), inplace=True)
... storing 'batch' as categorical
```



[26]:
```
sc.pl.umap(
    adata_full_new,
    color=imputed_proteins_all.columns,
    frameon=False,
    ncols=3,
    vmax="p99"
)
```


CD3_TotalSeqB
1500
1250
1000
750
500
250
CD4_TotalSeqB
2000
1500
1000
500
CD8a_TotalSeqB
2500
2000
1500
1000
500
CD14_TotalSeqB
1200
1000
800
600
400
200
CD15_TotalSeqB
200
175
150
125
100
75
50
CD16_TotalSeqB
800
600
400
200
CD56_TotalSeqB
300
250
200
150
100
50
CD19_TotalSeqB
600
500
400
300
200
100
CD25_TotalSeqB
125
100
75
50
25
CD45RA_TotalSeqB
2500
2000
1500
1000
500
CD45RO_TotalSeqB
600
500
400
300
200
100
PD-1_TotalSeqB
80
60
40
20
TIGIT_TotalSeqB
50
40
30
20
10
CD127_TotalSeqB
200
150
100
50

CHAPTER

EIGHT

UNSUPERVISED SURGERY PIPELINE WITH TRVAE

[1]:
```
import os
os.chdir('../')
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
```

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

8.1 Set relevant anndata.obs labels and training length

Here we use the CelSeq2 and SS2 studies as query data and the other 3 studies as reference atlas. We strongly suggest to use earlystopping to avoid over-fitting. The best earlystopping criteria is the 'val_unweighted_loss' for TRVAE.

[5]:
```
condition_key = 'study'
cell_type_key = 'cell_type'
target_conditions = ['Pancreas CelSeq2', 'Pancreas SS2']


trvae_epochs = 500
surgery_epochs = 500

early_stopping_kwargs = {
    "early_stopping_metric": "val_unweighted_loss",
    "threshold": 0,
    "patience": 20,
    "reduce_lr": True,
```

(continues on next page)

(continued from previous page)

```
    "lr_patience": 13,
    "lr_factor": 0.1,
}
```

8.2 Download Dataset and split into reference dataset and query dataset

[6]:
```
url = 'https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd'
output = 'pancreas.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd
To: C:\Users\sergei.rybakov\projects\notebooks\pancreas.h5ad
126MB [00:35, 3.52MB/s]
```

[6]:
```
'pancreas.h5ad'
```

[7]:
```
adata_all = sc.read('pancreas.h5ad')
```

This line makes sure that count data is in the adata.X. Remember that count data in adata.X is necessary when using "nb" or "zinb" loss. However, when using trVAE with MSE loss normalized data is necessary in adata.X

[8]:
```
adata = adata_all.raw.to_adata()
adata = remove_sparsity(adata)
source_adata = adata[~adata.obs[condition_key].isin(target_conditions)]
target_adata = adata[adata.obs[condition_key].isin(target_conditions)]
source_conditions = source_adata.obs[condition_key].unique().tolist()
```

[9]:
```
source_adata
```

[9]:
```
View of AnnData object with n_obs × n_vars = 10294 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

[10]:
```
target_adata
```

[10]:
```
View of AnnData object with n_obs × n_vars = 5387 × 1000
    obs: 'batch', 'study', 'cell_type', 'size_factors'
```

8.3 Create TRVAE model and train it on reference dataset

Create the trVAE model instance with NB loss as default. Insert "recon_loss='mse'," or "recon_loss='zinb'," to change the reconstruction loss.

[11]:
```
trvae = sca.models.TRVAE(
    adata=source_adata,
    condition_key=condition_key,
    conditions=source_conditions,
```

(continues on next page)

(continued from previous page)

```
    hidden_layer_sizes=[128, 128],
)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1000 128 3
        Hidden Layer 1 in/out: 128 128
        Mean/Var Layer in/out: 128 10
Decoder Architecture:
        First Layer in, out and cond:  10 128 3
        Hidden Layer 1 in/out: 128 128
        Output Layer in/out:  128 1000
```

[12]:
```
trvae.train(
    n_epochs=trvae_epochs,
    alpha_epoch_anneal=200,
    early_stopping_kwargs=early_stopping_kwargs
)
```

```
Trying to set attribute `.obs` of view, copying.
Trying to set attribute `.obs` of view, copying.
```

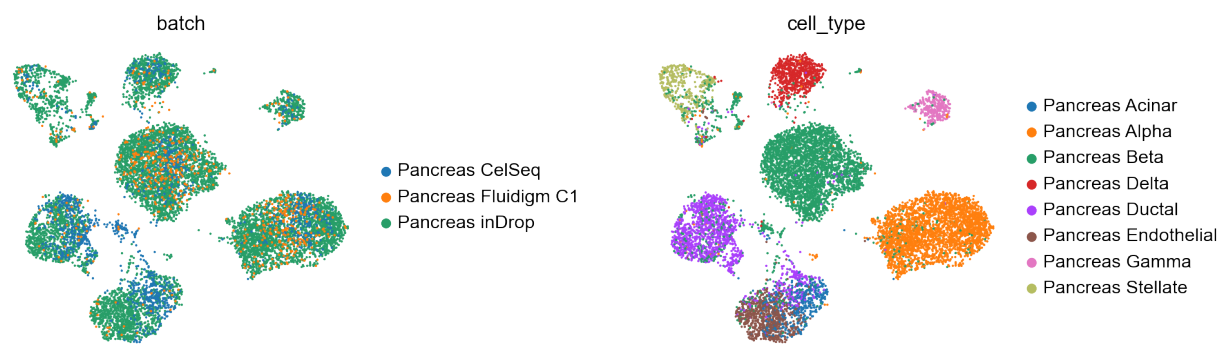
```
Valid_data 1029
Condition: 0 Counts in TrainData: 821
Condition: 1 Counts in TrainData: 147
Condition: 2 Counts in TrainData: 61
 |------------| 40.6%  - epoch_loss:    2387 - epoch_unweighted_loss:    2387 - epoch_
↪recon_loss:    2367 - epoch_kl_loss:      18 - epoch_mmd_loss:       2 - val_loss:   ␣
↪1267 - val_unweighted_loss:    1267 - val_recon_loss:    1248 - val_kl_loss:      13 -␣
↪val_mmd_loss:       5
ADJUSTED LR
 |-----------| 47.0%  - epoch_loss:    2357 - epoch_unweighted_loss:    2357 - epoch_
↪recon_loss:    2338 - epoch_kl_loss:      17 - epoch_mmd_loss:       2 - val_loss:   ␣
↪1346 - val_unweighted_loss:    1346 - val_recon_loss:    1327 - val_kl_loss:      14 -␣
↪val_mmd_loss:       5
ADJUSTED LR
 |-----------| 48.4%  - epoch_loss:    2371 - epoch_unweighted_loss:    2371 - epoch_
↪recon_loss:    2352 - epoch_kl_loss:      18 - epoch_mmd_loss:       2 - val_loss:   ␣
↪1302 - val_unweighted_loss:    1302 - val_recon_loss:    1284 - val_kl_loss:      14 -␣
↪val_mmd_loss:       5
Stopping early: no improvement of more than 0 nats in 20 epochs
If the early stopping criterion is too strong, please instantiate it with different␣
↪parameters in the train method.
Saving best state of network...
Best State was in Epoch 220
```

8.4 Create anndata file of latent representation and compute UMAP

[13]:
```
adata_latent = sc.AnnData(trvae.get_latent())
adata_latent.obs['cell_type'] = source_adata.obs[cell_type_key].tolist()
adata_latent.obs['batch'] = source_adata.obs[condition_key].tolist()
```

[14]:
```
sc.pp.neighbors(adata_latent, n_neighbors=8)
sc.tl.leiden(adata_latent)
sc.tl.umap(adata_latent)
sc.pl.umap(adata_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```



After pretraining the model can be saved for later use

[15]:
```
ref_path = 'reference_model/'
trvae.save(ref_path, overwrite=True)
```

8.5 Perform surgery on reference model and train on query dataset

[16]:
```
new_trvae = sca.models.TRVAE.load_query_data(adata=target_adata, reference_model=ref_
↪path)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1000 128 5
        Hidden Layer 1 in/out: 128 128
        Mean/Var Layer in/out: 128 10
Decoder Architecture:
        First Layer in, out and cond:  10 128 5
        Hidden Layer 1 in/out: 128 128
        Output Layer in/out:  128 1000
```

[17]:
```
new_trvae.train(
    n_epochs=surgery_epochs,
    alpha_epoch_anneal=200,
    early_stopping_kwargs=early_stopping_kwargs,
    weight_decay=0
)
```

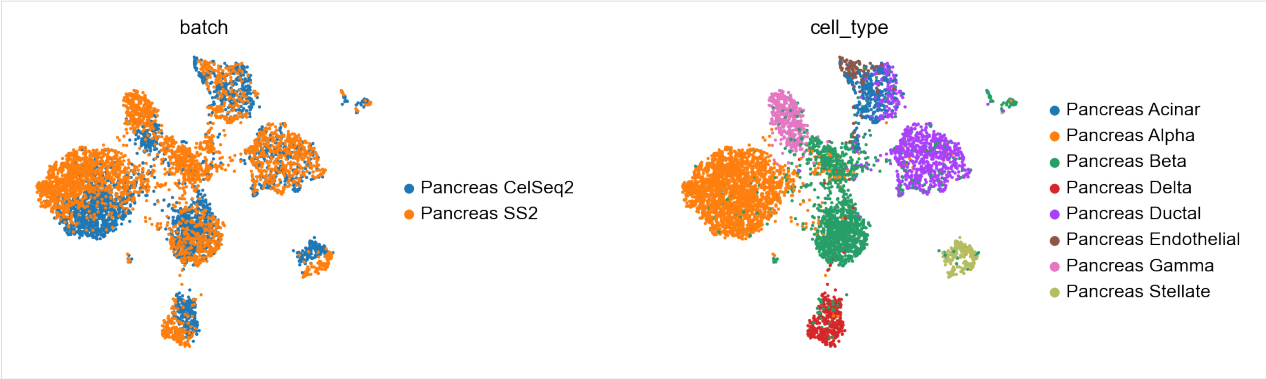
```
Trying to set attribute `.obs` of view, copying.
Trying to set attribute `.obs` of view, copying.
```

```
Valid_data 538
Condition: 0 Counts in TrainData: 0
Condition: 1 Counts in TrainData: 0
 |--------| 64.0%  - epoch_loss:    2664 - epoch_unweighted_loss:    2664 - epoch_recon_
↪loss:    2647 - epoch_kl_loss:      16 - epoch_mmd_loss:       0 - val_loss:    2606 -␣
↪val_unweighted_loss:    2606 - val_recon_loss:    2589 - val_kl_loss:      16 - val_
↪mmd_loss:       1
ADJUSTED LR
 |-------| 68.8%  - epoch_loss:    2576 - epoch_unweighted_loss:    2576 - epoch_recon_
↪loss:    2559 - epoch_kl_loss:      16 - epoch_mmd_loss:       0 - val_loss:    2493 -␣
↪val_unweighted_loss:    2493 - val_recon_loss:    2477 - val_kl_loss:      16 - val_
↪mmd_loss:       1
ADJUSTED LR
 |------| 70.2%  - epoch_loss:    2528 - epoch_unweighted_loss:    2528 - epoch_recon_
↪loss:    2512 - epoch_kl_loss:      16 - epoch_mmd_loss:       0 - val_loss:    2495 -␣
↪val_unweighted_loss:    2495 - val_recon_loss:    2478 - val_kl_loss:      16 - val_
↪mmd_loss:       1
Stopping early: no improvement of more than 0 nats in 20 epochs
If the early stopping criterion is too strong, please instantiate it with different␣
↪parameters in the train method.
Saving best state of network...
Best State was in Epoch 329
```

[18]:
```
adata_latent = sc.AnnData(new_trvae.get_latent())
adata_latent.obs['cell_type'] = target_adata.obs[cell_type_key].tolist()
adata_latent.obs['batch'] = target_adata.obs[condition_key].tolist()
```

[19]:
```
sc.pp.neighbors(adata_latent, n_neighbors=8)
sc.tl.leiden(adata_latent)
sc.tl.umap(adata_latent)
sc.pl.umap(adata_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```



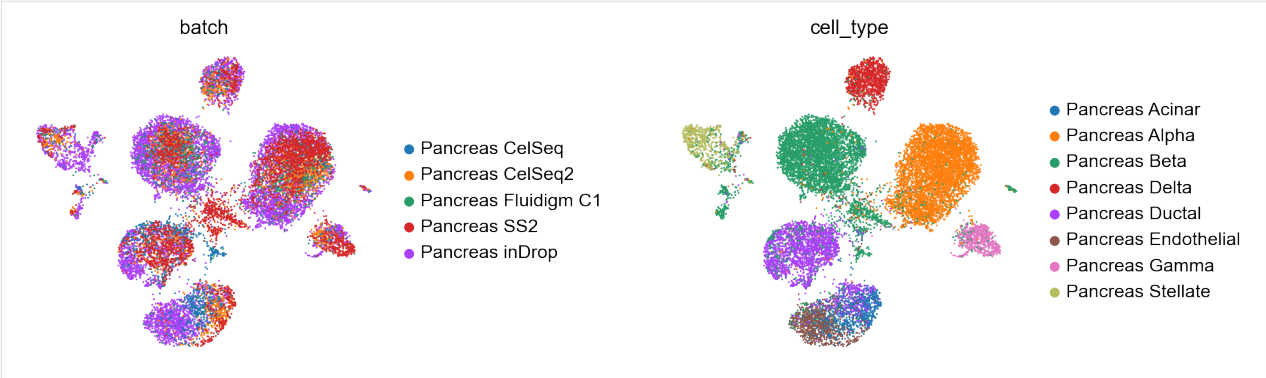
[20]:
```
surg_path = 'surgery_model'
new_trvae.save(surg_path, overwrite=True)
```

8.6 Get latent representation of reference + query dataset and compute UMAP

[21]:
```
full_latent = sc.AnnData(new_trvae.get_latent(adata.X, adata.obs[condition_key]))
full_latent.obs['cell_type'] = adata.obs[cell_type_key].tolist()
full_latent.obs['batch'] = adata.obs[condition_key].tolist()
```

[22]:
```
sc.pp.neighbors(full_latent, n_neighbors=8)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
sc.pl.umap(full_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```



CHAPTER

NINE

BUILD REFERENCE ATLAS FROM SCRATCH

[1]:
```
import os
os.chdir('../')
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
```

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

9.1 Download raw Dataset

[4]:
```
url = 'https://drive.google.com/uc?id=1Vh6RpYkusbGIZQC8GMFe3OKVDk5PWEpC'
output = 'pbmc.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1Vh6RpYkusbGIZQC8GMFe3OKVDk5PWEpC
To: /home/marco/Documents/git_repos/scarches/pbmc.h5ad
2.06GB [00:18, 113MB/s]
```

[4]:
```
'pbmc.h5ad'
```

[5]:
```
adata = sc.read('pbmc.h5ad')
```

[6]:
```
adata.X = adata.layers["counts"].copy()
```

We now split the data into reference and query dataset to simulate the building process. Here we use the '10X' batch as query data.

[7]:
```
target_conditions = ["10X"]
source_adata = adata[~adata.obs.study.isin(target_conditions)].copy()
target_adata = adata[adata.obs.study.isin(target_conditions)].copy()
print(source_adata)
print(target_adata)
```

```
AnnData object with n_obs × n_vars = 22779 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
AnnData object with n_obs × n_vars = 10727 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

For a better model performance it is necessary to select HVGs. We are doing this by applying the scanpy.pp function highly_variable_genes(). The n_top_genes is set to 2000 here. However, if you have more complicated datasets you might have to increase number of genes to capture more diversity in the data.

[8]:
```
source_adata.raw = source_adata
```

[9]:
```
source_adata
```

[9]:
```
AnnData object with n_obs × n_vars = 22779 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

[10]:
```
sc.pp.normalize_total(source_adata)
```

[11]:
```
sc.pp.log1p(source_adata)
```

[12]:
```
sc.pp.highly_variable_genes(
    source_adata,
    n_top_genes=2000,
    batch_key="batch",
    subset=True)
```

For consistency we set adata.X to be raw counts. In other datasets that may be already the case

[13]:
```
source_adata.X = source_adata.raw[:, source_adata.var_names].X
```

[14]:
```
source_adata
```

[14]:
```
AnnData object with n_obs × n_vars = 22779 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm', 'highly_variable_
↪nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'
    layers: 'counts'
```

9.2 Create SCVI model and train it on reference dataset

Remember that the adata file has to have count data in adata.X for SCVI/SCANVI if not further specified.

[15]:
```
sca.models.SCVI.setup_anndata(source_adata, batch_key="batch")
```

```
INFO     Using batches from adata.obs["batch"]
INFO     No label_key inputted, assuming all cells have same label
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Successfully registered anndata object containing 22779 cells, 2000 vars, 9␣
↪batches,
         1 labels, and 0 proteins. Also registered 0 extra categorical covariates and 0␣
↪extra
         continuous covariates.
INFO     Please do not further modify adata until model is trained.
```

Create the SCVI model instance with ZINB loss as default. Insert "gene_likelihood='nb'," to change the reconstruction loss to NB loss.

[16]:
```
vae = sca.models.SCVI(
    source_adata,
    n_layers=2,
    encode_covariates=True,
    deeply_inject_covariates=False,
    use_layer_norm="both",
    use_batch_norm="none",
)
```

[17]:
```
early_stopping_kwargs = {
    "early_stopping_metric": "elbo",
    "save_best_state_metric": "elbo",
    "patience": 10,
    "threshold": 0,
    "reduce_lr_on_plateau": True,
    "lr_patience": 8,
    "lr_factor": 0.1,
}
vae.train(n_epochs=500, frequency=1, early_stopping_kwargs=early_stopping_kwargs)
```

```
INFO     Training for 500 epochs
INFO     KL warmup for 400 epochs
Training...:  19%|        | 93/500 [02:04<09:08,  1.35s/it]INFO     Reducing LR on epoch␣
↪93.
Training...:  27%|        | 133/500 [02:58<08:23,  1.37s/it]INFO     Reducing LR on epoch␣
↪133.
Training...:  38%|       | 188/500 [04:13<06:59,  1.34s/it]INFO     Reducing LR on epoch␣
↪188.
Training...:  40%|       | 198/500 [04:26<06:41,  1.33s/it]INFO     Reducing LR on epoch␣
↪198.
Training...:  42%|      | 208/500 [04:39<06:31,  1.34s/it]INFO     Reducing LR on epoch␣
↪208.
Training...:  42%|      | 210/500 [04:42<06:25,  1.33s/it]INFO
         Stopping early: no improvement of more than 0 nats in 10 epochs
```

(continues on next page)

(continued from previous page)

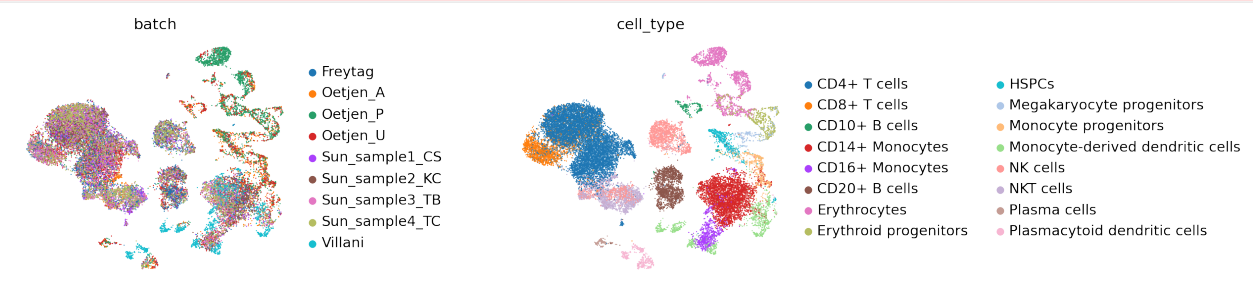
```
INFO     If the early stopping criterion is too strong, please instantiate it with␣
↪different
         parameters in the train method.
Training...:  42%|     | 210/500 [04:43<06:32,  1.35s/it]
INFO     Training is still in warming up phase. If your applications rely on the␣
↪posterior
         quality, consider training for more epochs or reducing the kl warmup.
INFO     Training time:  201 s. / 500 epochs
```

The resulting latent representation of the data can then be visualized with UMAP

[18]:
```
reference_latent = sc.AnnData(vae.get_latent_representation())
reference_latent.obs["cell_type"] = source_adata.obs["final_annotation"].tolist()
reference_latent.obs["batch"] = source_adata.obs["batch"].tolist()
```

[19]:
```
sc.pp.neighbors(reference_latent, n_neighbors=8)
sc.tl.leiden(reference_latent)
sc.tl.umap(reference_latent)
sc.pl.umap(reference_latent,
           color=['batch', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```



After pretraining the model can be saved for later use or also be uploaded for other researchers with via Zenodo. For the second option please also have a look at the Zenodo notebook.

[20]:
```
ref_path = 'ref_model/'
vae.save(ref_path, overwrite=True)
```

9.3 Use pretrained reference model and apply surgery with a new query dataset to get a bigger reference atlas

[21]:
```
target_adata
```

[21]:
```
AnnData object with n_obs × n_vars = 10727 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
```

(continues on next page)

(continued from previous page)

```
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

Since the model requires the datasets to have the same genes we also filter the query dataset to have the same genes as the reference dataset.

[22]:
```
target_adata = target_adata[:, source_adata.var_names]
target_adata
```

[22]:
```
View of AnnData object with n_obs × n_vars = 10727 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

We then can apply the model surgery with the new query dataset:

[23]:
```
model = sca.models.SCVI.load_query_data(
    target_adata,
    ref_path,
    freeze_dropout = True,
)
```

```
Trying to set attribute `.uns` of view, copying.
```

```
INFO     .obs[_scvi_labels] not found in target, assuming every cell is same category
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels']
INFO     Successfully registered anndata object containing 10727 cells, 2000 vars, 10
         batches, 1 labels, and 0 proteins. Also registered 0 extra categorical␣
↪covariates
         and 0 extra continuous covariates.
```

[24]:
```
model.train(n_epochs=500, frequency=1, early_stopping_kwargs=early_stopping_kwargs,␣
↪weight_decay=0)
```

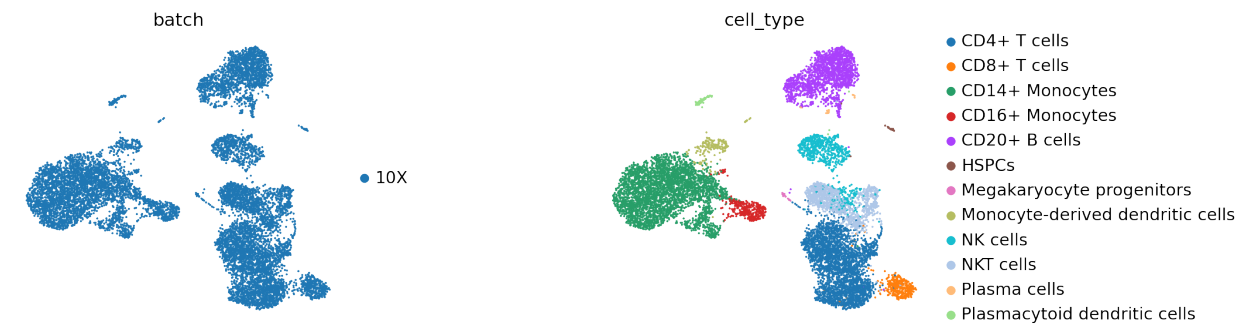
```
INFO     Training for 500 epochs
INFO     KL warmup for 400 epochs
Training...:  12%|        | 61/500 [00:33<04:03,  1.80it/s]INFO     Reducing LR on epoch␣
↪61.
Training...:  14%|        | 72/500 [00:39<03:52,  1.84it/s]INFO     Reducing LR on epoch␣
↪72.
Training...:  15%|        | 74/500 [00:40<03:52,  1.83it/s]INFO
         Stopping early: no improvement of more than 0 nats in 10 epochs
INFO     If the early stopping criterion is too strong, please instantiate it with␣
↪different
         parameters in the train method.
Training...:  15%|        | 74/500 [00:41<03:58,  1.79it/s]
INFO     Training is still in warming up phase. If your applications rely on the␣
↪posterior
         quality, consider training for more epochs or reducing the kl warmup.
INFO     Training time:  26 s. / 500 epochs
```

[25]:
```
query_latent = sc.AnnData(model.get_latent_representation())
query_latent.obs['cell_type'] = target_adata.obs["final_annotation"].tolist()
query_latent.obs['batch'] = target_adata.obs["batch"].tolist()
```

[26]:
```
sc.pp.neighbors(query_latent)
sc.tl.leiden(query_latent)
sc.tl.umap(query_latent)
plt.figure()
sc.pl.umap(
    query_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```

```
<Figure size 320x320 with 0 Axes>
```



And again we can save or upload the retrained model for later use or additional extensions.

[27]:
```
surgery_path = 'surgery_model'
model.save(surgery_path, overwrite=True)
```

9.4 Get latent representation of reference + query dataset and compute UMAP

[28]:
```
adata_full = source_adata.concatenate(target_adata, batch_key="ref_query")
adata_full
```

[28]:
```
AnnData object with n_obs × n_vars = 33506 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue
↪', '_scvi_batch', '_scvi_labels', '_scvi_local_l_mean', '_scvi_local_l_var', 'ref_query
↪'
    var: 'highly_variable-0', 'means-0', 'dispersions-0', 'dispersions_norm-0', 'highly_
↪variable_nbatches-0', 'highly_variable_intersection-0'
    layers: 'counts'
```

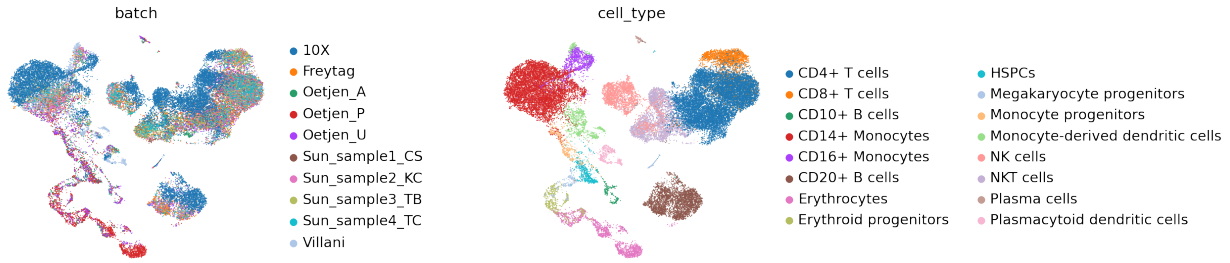
[29]:
```
full_latent = sc.AnnData(model.get_latent_representation(adata=adata_full))
full_latent.obs['cell_type'] = adata_full.obs["final_annotation"].tolist()
full_latent.obs['batch'] = adata_full.obs["batch"].tolist()
```

```
INFO     Input adata not setup with scvi. attempting to transfer anndata setup
INFO     Using data from adata.X
INFO     Computing library size prior per batch
INFO     Registered keys:['X', 'batch_indices', 'local_l_mean', 'local_l_var', 'labels']
INFO     Successfully registered anndata object containing 33506 cells, 2000 vars, 10
         batches, 1 labels, and 0 proteins. Also registered 0 extra categorical␣
↪covariates
         and 0 extra continuous covariates.
```

[30]:
```
sc.pp.neighbors(full_latent)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
plt.figure()
sc.pl.umap(
    full_latent,
    color=["batch", "cell_type"],
    frameon=False,
    wspace=0.6,
)
```

```
... storing 'cell_type' as categorical
... storing 'batch' as categorical
```

```
<Figure size 320x320 with 0 Axes>
```



[ ]:

CHAPTER

TEN

REFERENCE MAPING USING SCGEN

In this tutorial, we are going to build a reference atlas using scGen and also map two new query datasets on the top of the reference atlas.

Note: scGen requires cell-type labels for data integration. The method outputs both **corrected gene expression** and also latent space.

[1]:
```
import os
import sys
sys.path.insert(0, "../")

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
```

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
```

10.1 set relevant anndata.obs labels and training hyperparameters

[4]:
```
condition_key = 'study'
cell_type_key = 'cell_type'
target_conditions = ['Pancreas CelSeq2', 'Pancreas SS2']


epoch = 50

early_stopping_kwargs = {
    "early_stopping_metric": "val_loss",
    "patience": 20,
    "threshold": 0,
```

(continues on next page)

(continued from previous page)

```
    "reduce_lr": True,
    "lr_patience": 13,
    "lr_factor": 0.1,
}
```

10.2 Download Dataset and split into reference dataset and query dataset

[5]:
```
url = 'https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd'
output = 'pancreas.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ehxgfHTsMZXy6YzlFKGJOsBKQ5rrvMnd
To: /home/mo/projects/scarches/notebooks/pancreas.h5ad
126MB [00:01, 102MB/s]
```

[5]:
```
'pancreas.h5ad'
```

Imortant note : scGen requires normalized and log-transformed data in ``adata.X``

[6]:
```
adata = sc.read('pancreas.h5ad')
```

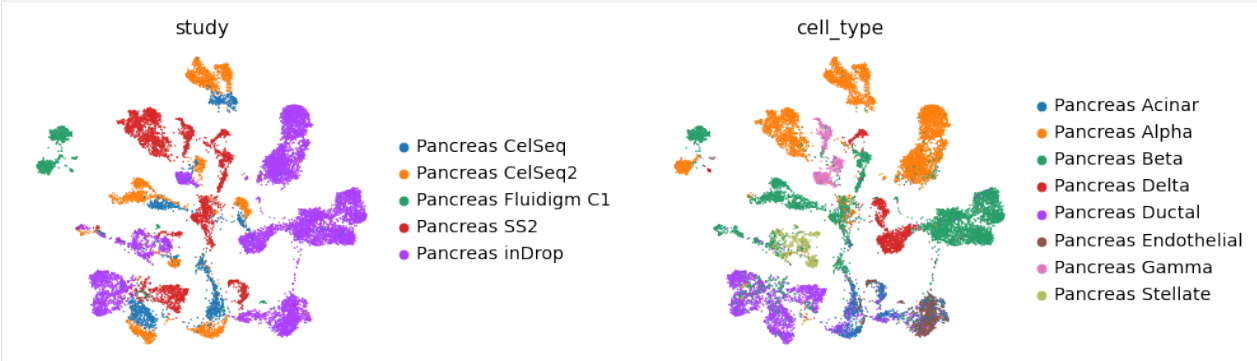
10.3 original uncorrected data UMAP

[7]:
```
sc.pp.neighbors(adata)
```

```
WARNING: You're trying to run this on 1000 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
```

[8]:
```
sc.tl.umap(adata)
```

[9]:
```
sc.pl.umap(adata, color=['study', 'cell_type'],
           frameon=False, wspace=0.6)
```



As observed above these Pancreas studies are seperated accroding to source of study

Here we use the CelSeq2 and SS2 studies as query data and the other 3 studies to build as reference atlas.

[10]:
```
adata = remove_sparsity(adata) # remove sparsity
source_adata = adata[~adata.obs[condition_key].isin(target_conditions)].copy()
target_adata = adata[adata.obs[condition_key].isin(target_conditions)].copy()
```

10.4 Create scGen model and train it on reference dataset

Create the scgen model instance

[11]:
```
network = sca.models.scgen(adata = source_adata,
                           hidden_layer_sizes=[256,128])
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out: 1000 256
        Hidden Layer 1 in/out: 256 128
        Mean/Var Layer in/out: 128 10
Decoder Architecture:
        First Layer in, out 10 128
        Hidden Layer 1 in/out: 128 256
        Output Layer in/out:  256 1000
```

[12]:
```
network.train(n_epochs=epoch, early_stopping_kwargs = early_stopping_kwargs)
```

```
 || 100.0%  - epoch_loss: 1.9351395184 - val_loss: 1.8963928728
Saving best state of network...
Best State was in Epoch 87
```

10.4.1 Correct batches in reference data

This function returns corrected gene expression in `adata.X`, raw uncorrected data in `adata.obsm["original_data"]`. Also it returns uncorrected data in `adata.layers["original_data"]`.

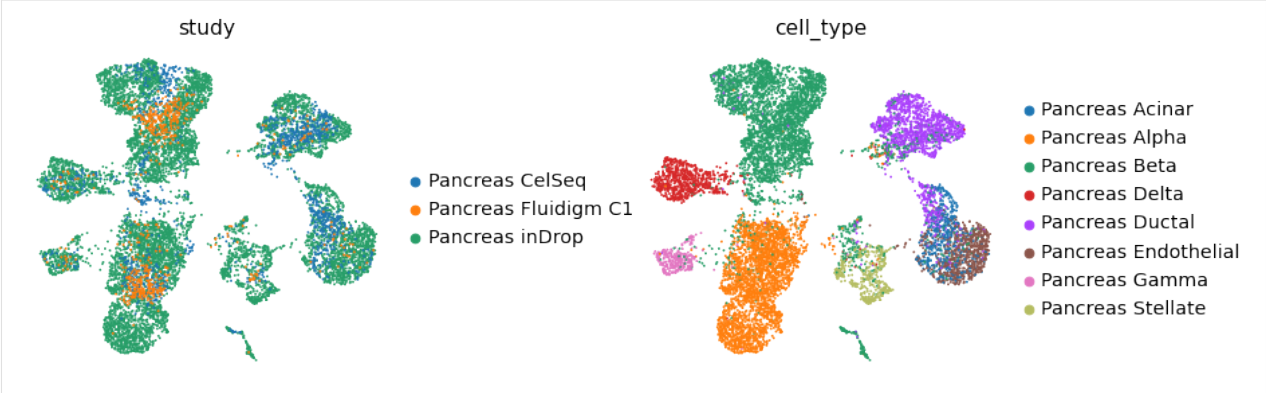
The low-dimensional corrected latent space in `adata.obsm["latent_corrected"]`

[13]:
```
corrected_reference_adata = network.batch_removal(source_adata, batch_key="study", cell_
↪label_key="cell_type",return_latent=True)
```

Corrected gene expression

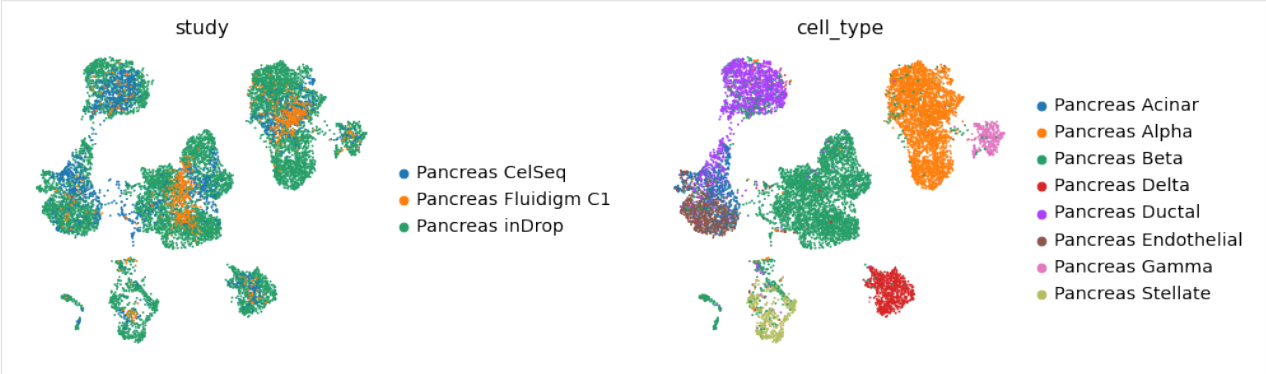
[14]:
```
sc.pp.neighbors(corrected_reference_adata)
sc.tl.umap(corrected_reference_adata)
sc.pl.umap(corrected_reference_adata, color=["study", "cell_type"], wspace=.5,␣
↪frameon=False)
```

```
WARNING: You're trying to run this on 1000 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
... storing 'cell_type' as categorical
```



We an also use low-dim corrected reference data

[15]:
```
sc.pp.neighbors(corrected_reference_adata, use_rep="latent_corrected")
sc.tl.umap(corrected_reference_adata)
sc.pl.umap(corrected_reference_adata,
           color=['study', 'cell_type'],
           frameon=False,
           wspace=0.6,
           )
```



After training, the model can be saved for later use and projection of new query studies

[16]:
```
ref_path = './ref_model/'
network.save(ref_path, overwrite=True)
```

[17]:
```
os.getcwd()
```

[17]:
```
'/home/mo/projects/scarches/notebooks'
```

10.5 Project query on top of the reference

query data needs to be preprocessed same way as reference data with same genes

This function need pretrained reference model, corrected gene expression from reference data and incorrected query data

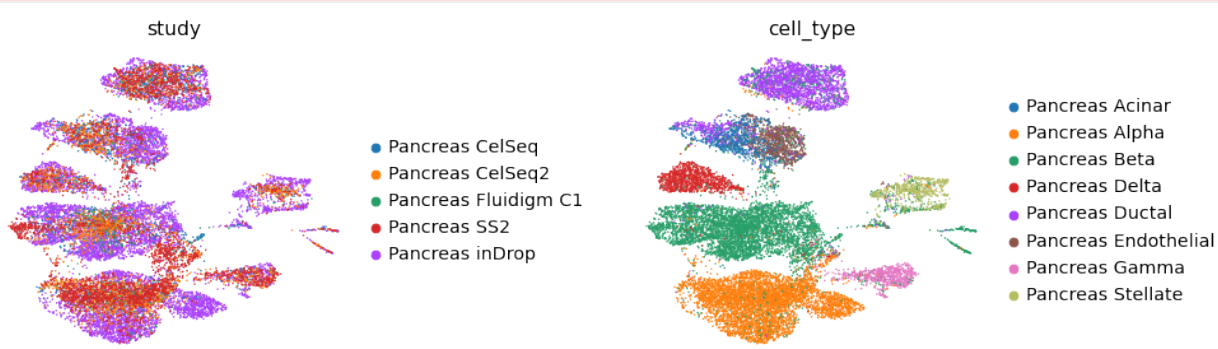
[18]:
```
# here we pass the saved model from a file to the map query
integrated_query = sca.models.scgen.map_query_data(reference_model = network,
                                                   corrected_reference = corrected_
↪reference_adata,
                                                   query = target_adata,
                                                   batch_key = 'study',
                                                   return_latent=True)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out: 1000 256
        Hidden Layer 1 in/out: 256 128
        Mean/Var Layer in/out: 128 10
Decoder Architecture:
        First Layer in, out 10 128
        Hidden Layer 1 in/out: 128 256
        Output Layer in/out:  256 1000
```

10.6 Plot the latent space of integrated query and reference

[19]:
```
sc.pp.neighbors(integrated_query, use_rep="latent_corrected")
sc.tl.umap(integrated_query)
sc.pl.umap(
    integrated_query,
    color=["study", "cell_type"],
    frameon=False,
    wspace=0.6)
```

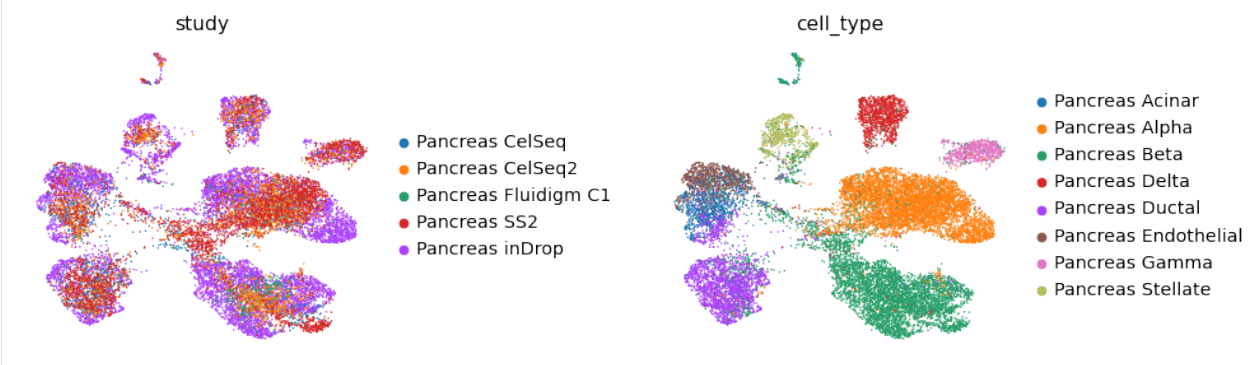
```
... storing 'batch' as categorical
... storing 'study' as categorical
... storing 'cell_type' as categorical
... storing 'original_batch' as categorical
```



10.7 Plot corrected gene expression space of integrated query and reference

[20]:
```
sc.pp.neighbors(integrated_query)
sc.tl.umap(integrated_query)
sc.pl.umap(
    integrated_query,
    color=["study", "cell_type"],
    frameon=False,
    wspace=0.6)
```

```
WARNING: You're trying to run this on 1000 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
        Falling back to preprocessing with `sc.pp.pca` and default params.
```



CHAPTER

ELEVEN

BASIC TUTORIAL FOR QUERY TO REFERENCE MAPING USING EXPIMAP

Also see the advanced tutorial to learn about adding constrained and unconstrained extension nodes to the query to capture new sources of variation, that is new and de novo gene programs, not in the reference dataset.

[1]:
```
import warnings
warnings.simplefilter(action='ignore')
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
import numpy as np
import gdown
```

```
Global seed set to 0
```

[3]:
```
sc.set_figure_params(frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

11.1 Download reference and do preprocessing

[4]:
```
url = 'https://drive.google.com/uc?id=1Rnm-XKEqPLdOq3lpa3ka2aV4bOXVCLP0'
output = 'pbmc_tutorial.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1Rnm-XKEqPLdOq3lpa3ka2aV4bOXVCLP0
To: C:\Users\sergei.rybakov\projects\notebooks\pbmc_tutorial.h5ad
100%|| 231M/231M [00:42<00:00, 5.39MB/s]
```

[4]:
```
'pbmc_tutorial.h5ad'
```

[6]:
```
adata = sc.read('pbmc_tutorial.h5ad')
```

`.X` should contain raw counts.

[5]:
```
adata.X = adata.layers["counts"].copy()
```

Read the Reactome annotations, make a binary matrix where rows represent gene symbols and columns represent the terms, and add the annotations matrix to the reference dataset. The binary matrix of annotations is stored in `adata.varm['I']`. Note that only terms with minimum of 12 genes in the reference dataset are retained.

[ ]:
```
url = 'https://drive.google.com/uc?id=1136LntaVr92G1MphGeMVcmpE0AqcqM6c'
output = 'reactome.gmt'
gdown.download(url, output, quiet=False)
```

[6]:
```
sca.utils.add_annotations(adata, 'reactome.gmt', min_genes=12, clean=True)
```

Remove all genes which are not present in the Reactome annotations.

[7]:
```
adata._inplace_subset_var(adata.varm['I'].sum(1)>0)
```

For a better model performance it is necessary to select HVGs. We are doing this by applying the `scanpy.pp` function `highly_variable_genes()`. The `n_top_genes` is set to 2000 here. However, for more complicated datasets you might have to increase number of genes to capture more diversity in the data.

[8]:
```
sc.pp.normalize_total(adata)
```

[9]:
```
sc.pp.log1p(adata)
```

[10]:
```
sc.pp.highly_variable_genes(
    adata,
    n_top_genes=2000,
    batch_key="batch",
    subset=True)
```

Filter out all annotations (terms) with less than 12 genes.

[11]:
```
select_terms = adata.varm['I'].sum(0)>12
```

[12]:
```
adata.uns['terms'] = np.array(adata.uns['terms'])[select_terms].tolist()
```

[13]:
```
adata.varm['I'] = adata.varm['I'][:, select_terms]
```

Filter out genes not present in any of the terms after selection of HVGs.

[14]:
```
adata._inplace_subset_var(adata.varm['I'].sum(1)>0)
```

Put the counts data back to `adata.X`.

[15]:
```
adata.X = adata.layers["counts"].copy()
```

11.2 Create expiMap model and train it on reference dataset

[16]:
```
intr_cvae = sca.models.EXPIMAP(
    adata=adata,
    condition_key='study',
    hidden_layer_sizes=[256, 256, 256],
    recon_loss='nb'
)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1972 256 4
        Hidden Layer 1 in/out: 256 256
        Hidden Layer 2 in/out: 256 256
        Mean/Var Layer in/out: 256 282
Decoder Architecture:
        Masked linear layer in, ext_m, ext, cond, out:  282 0 0 4 1972
        with hard mask.
Last Decoder layer: softmax
```

Set the `alpha` hyperparameter. This regulates the strength of group lasso regularization of annotations (terms). Higher value means that a larger number of latent variables corresponding to annotations will be deactivated during training depending on their contribution to the reconstruction loss.

See also https://docs.scarches.org/en/latest/training_tips.html for the recommendation on hyperparameter choice.

[1]:
```
ALPHA = 0.7
```

[18]:
```
early_stopping_kwargs = {
    "early_stopping_metric": "val_unweighted_loss", # val_unweighted_loss
    "threshold": 0,
    "patience": 50,
    "reduce_lr": True,
    "lr_patience": 13,
    "lr_factor": 0.1,
}
intr_cvae.train(
    n_epochs=400,
    alpha_epoch_anneal=100,
    alpha=ALPHA,
    alpha_kl=0.5,
    weight_decay=0.,
    early_stopping_kwargs=early_stopping_kwargs,
    use_early_stopping=True,
    monitor_only_val=False,
    seed=2020,
)
```

```
Init the group lasso proximal operator for the main terms.
 |------------| 41.8%  - epoch_loss: 875.0875513335 - epoch_recon_loss: 849.8969689248 -␣
↪epoch_kl_loss: 50.3811637679 - val_loss: 938.5590139536 - val_recon_loss: 912.
↪9722806490 - val_kl_loss: 51.17346543531427418
ADJUSTED LR
```

(continues on next page)

(continued from previous page)

```
 |-----------| 46.0%  - epoch_loss: 867.2534920638 - epoch_recon_loss: 842.2286955321 -␣
↪epoch_kl_loss: 50.0495922638 - val_loss: 939.3953810472 - val_recon_loss: 913.
↪6503906250 - val_kl_loss: 51.4899893541
ADJUSTED LR
 |----------| 50.5%  - epoch_loss: 868.1691942506 - epoch_recon_loss: 843.1531918455 -␣
↪epoch_kl_loss: 50.0320032611 - val_loss: 937.6233802209 - val_recon_loss: 911.
↪8207420936 - val_kl_loss: 51.6052835905
ADJUSTED LR
 |--------| 60.2%  - epoch_loss: 874.0884924476 - epoch_recon_loss: 848.9184943453 -␣
↪epoch_kl_loss: 50.3399958715 - val_loss: 938.2763272799 - val_recon_loss: 912.
↪5495535044 - val_kl_loss: 51.4535569411
ADJUSTED LR
 |--------| 63.5%  - epoch_loss: 871.6074394659 - epoch_recon_loss: 846.5056099038 -␣
↪epoch_kl_loss: 50.2036596073 - val_loss: 938.8283644456 - val_recon_loss: 913.
↪1071777344 - val_kl_loss: 51.4423753298
ADJUSTED LR
 |-------| 66.8%  - epoch_loss: 872.8487410233 - epoch_recon_loss: 847.7790292798 -␣
↪epoch_kl_loss: 50.1394200054 - val_loss: 938.3776691143 - val_recon_loss: 912.
↪6276245117 - val_kl_loss: 51.5000856840
ADJUSTED LR
 |-------| 69.5%  - epoch_loss: 872.2917062018 - epoch_recon_loss: 847.1418017258 -␣
↪epoch_kl_loss: 50.2998066532 - val_loss: 943.0606759878 - val_recon_loss: 917.
↪2311988244 - val_kl_loss: 51.6589726668
Stopping early: no improvement of more than 0 nats in 50 epochs
If the early stopping criterion is too strong, please instantiate it with different␣
↪parameters in the train method.
Saving best state of network...
Best State was in Epoch 226
```

[16]:
```
MEAN = False
```

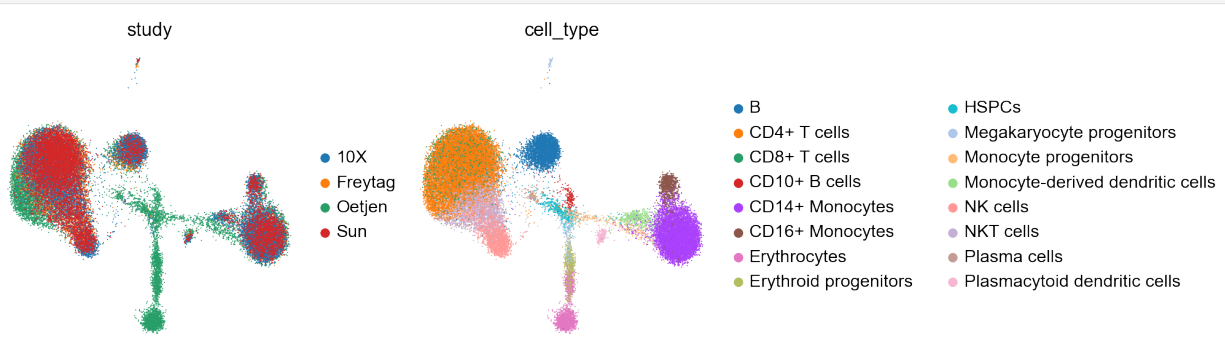
Plot the latent space of the reference.

[20]:
```
adata.obsm['X_cvae'] = intr_cvae.get_latent(mean=MEAN, only_active=True)
```

[21]:
```
sc.pp.neighbors(adata, use_rep='X_cvae')
```

[22]:
```
sc.tl.umap(adata)
```

[23]:
```
sc.pl.umap(adata, color=['study', 'cell_type'], frameon=False)
```



11.3 Downaload the query dataset for reference mapping

The Kang dataset contains control and IFN-beta stimulated cells. We use this as the query dataset.

[ ]:
```
url = 'https://drive.google.com/uc?id=1t3oMuUfueUz_caLm5jmaEYjBxVNSsfxG'
output = 'kang_tutorial.h5ad'
gdown.download(url, output, quiet=False)
```

[17]:
```
kang = sc.read('kang_tutorial.h5ad')[:, adata.var_names].copy()
```

[18]:
```
kang.obs['study'] = 'Kang'
```

[19]:
```
kang.uns['terms'] = adata.uns['terms']
```

11.4 Initlizling the model for query training

[27]:
```
q_intr_cvae = sca.models.EXPIMAP.load_query_data(kang, intr_cvae)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1972 256 5
        Hidden Layer 1 in/out: 256 256
        Hidden Layer 2 in/out: 256 256
        Mean/Var Layer in/out: 256 282
Decoder Architecture:
        Masked linear layer in, ext_m, ext, cond, out:  282 0 0 5 1972
        with hard mask.
Last Decoder layer: softmax
```

[28]:
```
q_intr_cvae.train(n_epochs=400, alpha_epoch_anneal=100, weight_decay=0., alpha_kl=0.1,␣
↪seed=2020, use_early_stopping=True)
```

```
 |------------| 41.2%  - val_loss: 519.4205793901 - val_recon_loss: 512.4798778187 - val_
↪kl_loss: 69.40706010300
ADJUSTED LR
 |------------| 43.0%  - val_loss: 520.2541309703 - val_recon_loss: 513.2979486639 - val_
↪kl_loss: 69.5618133545
Stopping early: no improvement of more than 0 nats in 20 epochs
If the early stopping criterion is too strong, please instantiate it with different␣
↪parameters in the train method.
Saving best state of network...
Best State was in Epoch 150
```

Save your model.

[ ]:
```
q_intr_cvae.save('query_kang_tutorial')
```

11.5 Get latent representation of reference + query dataset

[33]:
```
kang_pbmc = sc.AnnData.concatenate(adata, kang, batch_key='batch_join', uns_merge='same')
```

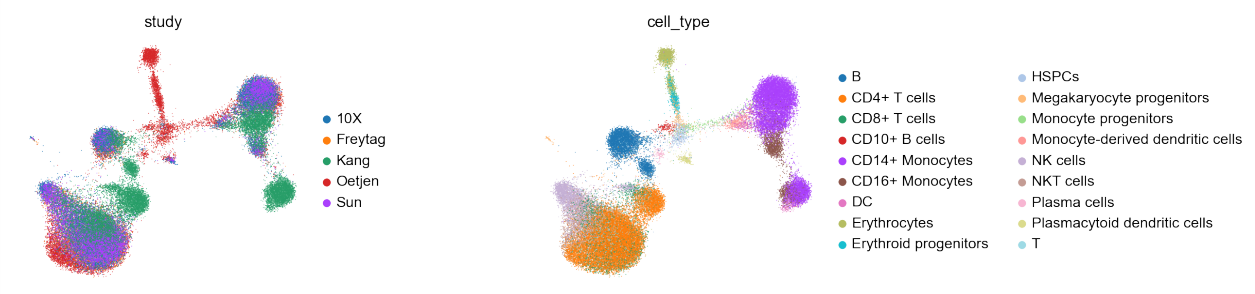
[34]:
```
kang_pbmc.obsm['X_cvae'] = q_intr_cvae.get_latent(kang_pbmc.X, kang_pbmc.obs['study'],␣
↪mean=MEAN, only_active=True)
```

[35]:
```
sc.pp.neighbors(kang_pbmc, use_rep='X_cvae')
sc.tl.umap(kang_pbmc)
```

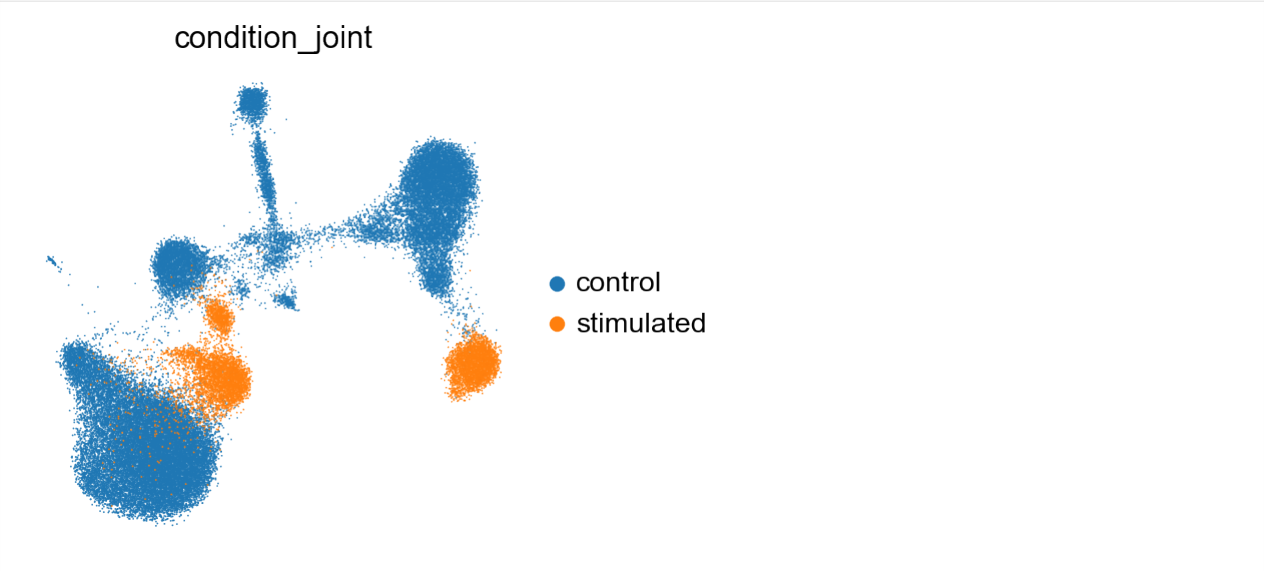
[36]:
```
kang_pbmc.obs['condition_joint'] = kang_pbmc.obs.condition.astype(str)
kang_pbmc.obs['condition_joint'][kang_pbmc.obs['condition_joint'].astype(str)=='nan']=
↪'control'
```

[37]:
```
sc.pl.umap(kang_pbmc, color=['study', 'cell_type'], frameon=False, wspace=0.6)
```

```
... storing 'batch' as categorical
... storing 'chemistry' as categorical
... storing 'data_type' as categorical
... storing 'final_annotation' as categorical
... storing 'sample_ID' as categorical
... storing 'species' as categorical
... storing 'study' as categorical
... storing 'tissue' as categorical
... storing 'cell_type' as categorical
... storing 'orig.ident' as categorical
... storing 'stim' as categorical
... storing 'seurat_annotations' as categorical
... storing 'condition' as categorical
... storing 'condition_joint' as categorical
```



[38]:
```
sc.pl.umap(kang_pbmc, color='condition_joint', frameon=False, wspace=0.6)
```



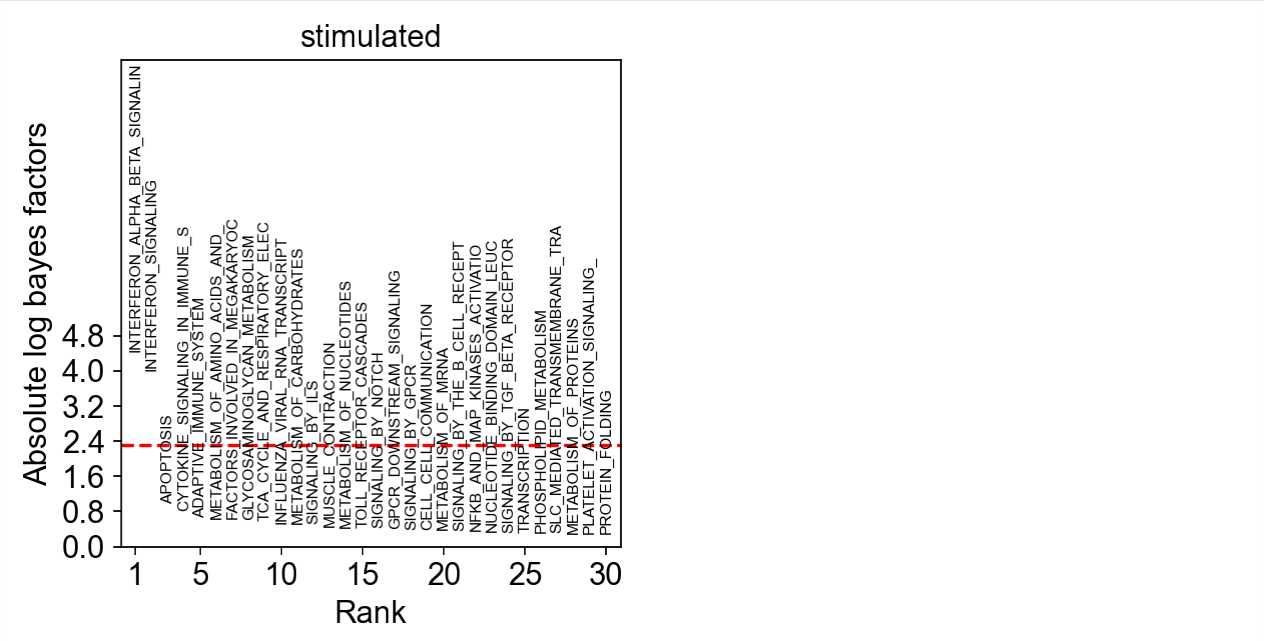
Calculate directions of upregulation for each latent score and put them to `kang_pbmc.uns['directions']`.

[39]:
```
q_intr_cvae.latent_directions(adata=kang_pbmc)
```

Do gene set enrichment test for condition in reference + query using Bayes Factors.

[40]:
```
q_intr_cvae.latent_enrich(groups='condition_joint', comparison='control', use_
↪directions=True, adata=kang_pbmc)
```

[41]:
```
fig = sca.plotting.plot_abs_bfs(kang_pbmc, yt_step=0.8, scale_y=2.5, fontsize=7)
```



As expected, `INTERFERON_ALPHA_BETA_SIGNALING` is the top differential program in stimulated compared to control cells.

Plot the latent variables for query + reference corresponding to the annotations 'INTERFERON_SIGNALING', 'SIG-

NALING_BY_THE_B_CELL_RECEPTOR', 'INTERFERON_ALPHA_BETA_SIGNALING'.

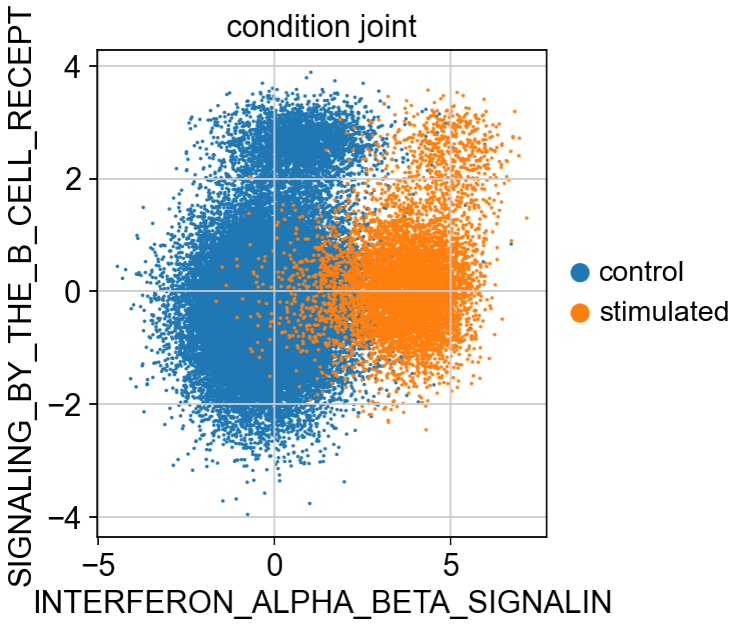
[42]:
```
terms = kang_pbmc.uns['terms']
select_terms = ['INTERFERON_SIGNALING', 'INTERFERON_ALPHA_BETA_SIGNALIN', 'SIGNALING_BY_
↪THE_B_CELL_RECEPT']
idx = [terms.index(term) for term in select_terms]
```

Multiplying the latent varibales by the calculated directons to ensure positive latent scores corresponf to upregulation.

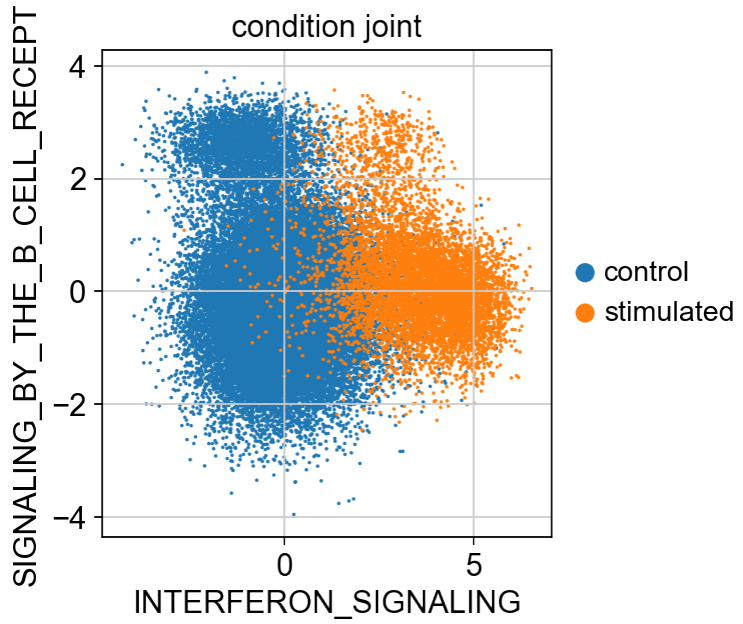
[43]:
```
latents = (q_intr_cvae.get_latent(kang_pbmc.X, kang_pbmc.obs['study'], mean=MEAN) * kang_
↪pbmc.uns['directions'])[:, idx]
```

[44]:
```
kang_pbmc.obs['INTERFERON_SIGNALING'] = latents[:, 0]
kang_pbmc.obs['INTERFERON_ALPHA_BETA_SIGNALIN'] = latents[:, 1]

kang_pbmc.obs['SIGNALING_BY_THE_B_CELL_RECEPT'] = latents[:, 2]
```

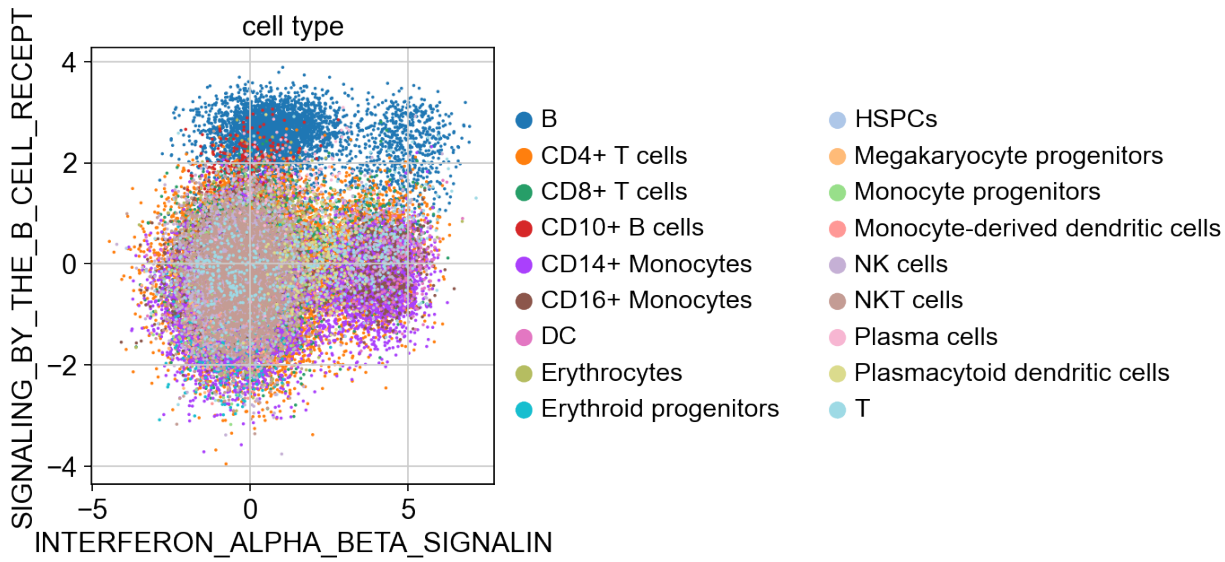
[45]:
```
sc.pl.scatter(kang_pbmc, x='INTERFERON_ALPHA_BETA_SIGNALIN', y='SIGNALING_BY_THE_B_CELL_
↪RECEPT', color='condition_joint', size=10)
```



[46]:
```
sc.pl.scatter(kang_pbmc, x='INTERFERON_SIGNALING', y='SIGNALING_BY_THE_B_CELL_RECEPT',␣
↪color='condition_joint', size=10)
```



[47]:
```
sc.pl.scatter(kang_pbmc, x='INTERFERON_ALPHA_BETA_SIGNALIN', y='SIGNALING_BY_THE_B_CELL_
↪RECEPT', color='cell_type', size=10)
```



CHAPTER

TWELVE

ADVANCED TUTORIAL FOR QUERY TO REFERENCE MAPPING USING EXPIMAP WITH DE NOVO LEARNED GENE PROGRAMS

[1]:
```
import warnings
warnings.simplefilter(action='ignore')
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
import numpy as np
import gdown
```

```
Global seed set to 0
```

[3]:
```
sc.set_figure_params(frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)
```

12.1 Download reference and do preprocessing

[4]:
```
url = 'https://drive.google.com/uc?id=1Rnm-XKEqPLdOq3lpa3ka2aV4bOXVCLP0'
output = 'pbmc_tutorial.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1Rnm-XKEqPLdOq3lpa3ka2aV4bOXVCLP0
To: C:\Users\sergei.rybakov\projects\notebooks\pbmc_tutorial.h5ad
100%|| 231M/231M [00:42<00:00, 5.39MB/s]
```

[4]:
```
'pbmc_tutorial.h5ad'
```

[4]:
```
adata = sc.read('pbmc_tutorial.h5ad')
```

`.X` should contain raw counts.

[5]:
```
adata.X = adata.layers["counts"].copy()
```

Read the Reactome annotations, make a binary matrix where rows represent gene symbols and columns represent the terms, and add the annotations matrix to the reference dataset. The binary matrix of annotations is stored in `adata.varm['I']`. Note that only terms with minimum of 12 genes in the reference dataset are retained.

[ ]:
```
url = 'https://drive.google.com/uc?id=1136LntaVr92G1MphGeMVcmpE0AqcqM6c'
output = 'reactome.gmt'
gdown.download(url, output, quiet=False)
```

[6]:
```
sca.utils.add_annotations(adata, 'reactome.gmt', min_genes=12, clean=True)
```

Remove all genes which are not present in the Reactome annotations.

[7]:
```
adata._inplace_subset_var(adata.varm['I'].sum(1)>0)
```

For a better model performance it is necessary to select HVGs. We are doing this by applying the `scanpy.pp` function `highly_variable_genes()`. The `n_top_genes` is set to 2000 here. However, for more complicated datasets you might have to increase number of genes to capture more diversity in the data.

[8]:
```
sc.pp.normalize_total(adata)
```

[9]:
```
sc.pp.log1p(adata)
```

[10]:
```
sc.pp.highly_variable_genes(
    adata,
    n_top_genes=2000,
    batch_key="batch",
    subset=True)
```

Filter out any annotations (terms) with less than 12 genes.

[11]:
```
select_terms = adata.varm['I'].sum(0)>12
```

[12]:
```
adata.uns['terms'] = np.array(adata.uns['terms'])[select_terms].tolist()
```

[13]:
```
adata.varm['I'] = adata.varm['I'][:, select_terms]
```

Filter out genes not present in any retained terms after selection of HVGs.

[14]:
```
adata._inplace_subset_var(adata.varm['I'].sum(1)>0)
```

Put the count data back to `adata.X`.

[15]:
```
adata.X = adata.layers["counts"].copy()
```

12.2 Example with constrained and unconstrained extension nodes

Later, we will use a query dataset that contains IFN-beta stimulated and unstimulated PBMC cells.

Here, we remove some Interferon beta and B cell specific signals from the reference by dropping some related terms from the annotation matrix. The signals corresponding to these terms will be recovered later with the extension nodes added in the query at the surgery step.

Select the interferon beta annotations from the loaded Reactome pathway database for removal.

[16]:
```
rm_terms = ['INTERFERON_SIGNALING', 'INTERFERON_ALPHA_BETA_SIGNALIN',
            'CYTOKINE_SIGNALING_IN_IMMUNE_S', 'ANTIVIRAL_MECHANISM_BY_IFN_STI']
```

Select the annotations related to B cells for removal.

[17]:
```
rm_terms += ['SIGNALING_BY_THE_B_CELL_RECEPT', 'MHC_CLASS_II_ANTIGEN_PRESENTAT']
```

[18]:
```
ix_f = []
for t in rm_terms:
    ix_f.append(adata.uns['terms'].index(t))
```

Store the 'SIGNALING_BY_THE_B_CELL_RECEPT' annotation separately.

[19]:
```
query_mask = adata.varm['I'][:, ix_f[4]][:, None].copy()
```

Remove the selected annotations.

[20]:
```
adata.varm['I'] = np.delete(adata.varm['I'], ix_f, axis=1)
```

[21]:
```
adata.uns['terms'] = [term for term in adata.uns['terms'] if term not in rm_terms]
```

Remove B cells from the reference.

[22]:
```
rm_b = ["B", "CD10+ B cells"]
```

[23]:
```
adata = adata[~adata.obs['cell_type'].isin(rm_b)].copy()
```

12.3 Train the reference.

[24]:
```
intr_cvae = sca.models.EXPIMAP(
    adata=adata,
    condition_key='study',
    hidden_layer_sizes=[300, 300, 300],
    recon_loss='nb'
)
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1972 300 4
        Hidden Layer 1 in/out: 300 300
        Hidden Layer 2 in/out: 300 300
        Mean/Var Layer in/out: 300 276
Decoder Architecture:
        Masked linear layer in, ext_m, ext, cond, out:  276 0 0 4 1972
        with hard mask.
Last Decoder layer: softmax
```

See https://docs.scarches.org/en/latest/training_tips.html for the recommendation on hyperparameter choice.

[25]:
```
ALPHA = 0.7
```

[26]:
```
early_stopping_kwargs = {
    "early_stopping_metric": "val_unweighted_loss",
```

(continues on next page)

(continued from previous page)

```
    "threshold": 0,
    "patience": 50,
    "reduce_lr": True,
    "lr_patience": 13,
    "lr_factor": 0.1,
}
intr_cvae.train(
    n_epochs=400,
    alpha_epoch_anneal=100,
    alpha=ALPHA,
    alpha_kl=0.5,
    weight_decay=0.,
    early_stopping_kwargs=early_stopping_kwargs,
    use_early_stopping=True,
    seed=2020
)
```

```
Init the group lasso proximal operator for the main terms.
 |------| 72.2%  - val_loss: 935.2109799592 - val_recon_loss: 909.6967879586 - val_kl_
↪loss: 51.0283828404208
ADJUSTED LR
 |-----| 76.8%  - val_loss: 934.3165150518 - val_recon_loss: 908.9699786642 - val_kl_
↪loss: 50.6930810680
ADJUSTED LR
 |----| 80.2%  - val_loss: 934.5560806938 - val_recon_loss: 909.1601987092 - val_kl_loss:
↪ 50.7917618130
ADJUSTED LR
 |----| 83.5%  - val_loss: 934.7631597104 - val_recon_loss: 909.4011548913 - val_kl_loss:
↪ 50.7240132871
ADJUSTED LR
 |---| 86.8%  - val_loss: 934.5707238239 - val_recon_loss: 909.2114868164 - val_kl_loss:␣
↪50.7184793224
ADJUSTED LR
 |---| 89.5%  - val_loss: 934.3915139903 - val_recon_loss: 909.0323433254 - val_kl_loss:␣
↪50.7183265686
Stopping early: no improvement of more than 0 nats in 50 epochs
If the early stopping criterion is too strong, please instantiate it with different␣
↪parameters in the train method.
Saving best state of network...
Best State was in Epoch 306
```

12.4 Referece mapping while learning new varation from query data with extension nodes

The Kang dataset contains control and IFN-beta stimulated cells. We use this as the query dataset.

[ ]:
```
url = 'https://drive.google.com/uc?id=1t3oMuUfueUz_caLm5jmaEYjBxVNSsfxG'
output = 'kang_tutorial.h5ad'
gdown.download(url, output, quiet=False)
```

[27]:
```
kang = sc.read('kang_tutorial.h5ad')[:, adata.var_names].copy()
```

[28]:
```
kang.obs['study'] = 'Kang'
```

[29]:
```
kang.uns['terms'] = adata.uns['terms']
```

Add 3 unconstrained (to capture de novo programs) and one constrained nodes, where the constrain represents the 'SIGNALING_BY_THE_B_CELL_RECEPT' term. Note that this term was dropped when the reference model was learned. Also use HSIC regularization for the unconstrained nodes to encourge independence of learned de novo gene programs.

[30]:
```
q_intr_cvae = sca.models.EXPIMAP.load_query_data(kang, intr_cvae,
                                                 unfreeze_ext=True,
                                                 new_n_ext=3,
                                                 new_n_ext_m=1,
                                                 new_ext_mask=query_mask.T,
                                                 new_soft_ext_mask=True,
                                                 use_hsic=True,
                                                 hsic_one_vs_all=True
                                                )
```

```
INITIALIZING NEW NETWORK...
Encoder Architecture:
        Input Layer in, out and cond: 1972 300 5
        Hidden Layer 1 in/out: 300 300
        Hidden Layer 2 in/out: 300 300
        Mean/Var Layer in/out: 300 276
        Expanded Mean/Var Layer in/out: 300 4
Decoder Architecture:
        Masked linear layer in, ext_m, ext, cond, out:  276 1 3 5 1972
        with hard mask.
Last Decoder layer: softmax
```

Train with hypeparameters:

`gamma_ext` - L1 regularization coefficient for the new unconstrained nodes. Specifies the strength of sparcity enforcement for these nodes.

`gamma_epoch_anneal` - number of epochs for `gamma_ext` annealing.

`alpha_l1` - L1 regularization coefficient for the soft mask of the new constrained node.

`beta` - HSIC regularization coefficient for the unconstrained nodes, enforces their independence from the old reference nodes and from each other if `hsic_one_vs_all=True`.

[31]:
```
q_intr_cvae.train(
    n_epochs=250,
    alpha_epoch_anneal=120,
    alpha_kl=0.22,
    weight_decay=0.,
    alpha_l1=0.96,
    gamma_ext=0.7,
    gamma_epoch_anneal=50,
    beta=3.,
```

(continues on next page)

(continued from previous page)

```
    seed=2020,
    use_early_stopping=False
)
```

```
Init the L1 proximal operator for the unannotated extension.
Init the soft mask proximal operator for the annotated extension.
 || 100.0%  - val_hsic_loss: 0.2133808624 - val_loss: 517.5771179199 - val_recon_loss:␣
↪501.4962740811 - val_kl_loss: 70.1850114302
```

12.5 Analysis of the extension nodes for reference + query dataset

[32]:
```
kang_pbmc = sc.AnnData.concatenate(adata, kang, batch_key='batch_join', uns_merge='same')
```

[33]:
```
kang_pbmc.obs['condition_joint'] = kang_pbmc.obs.condition.astype(str)
kang_pbmc.obs['condition_joint'][kang_pbmc.obs['condition_joint'].astype(str)=='nan']=
↪'control'
```

This adds extension nodes' names to `kang_pbmc.uns['terms']`.

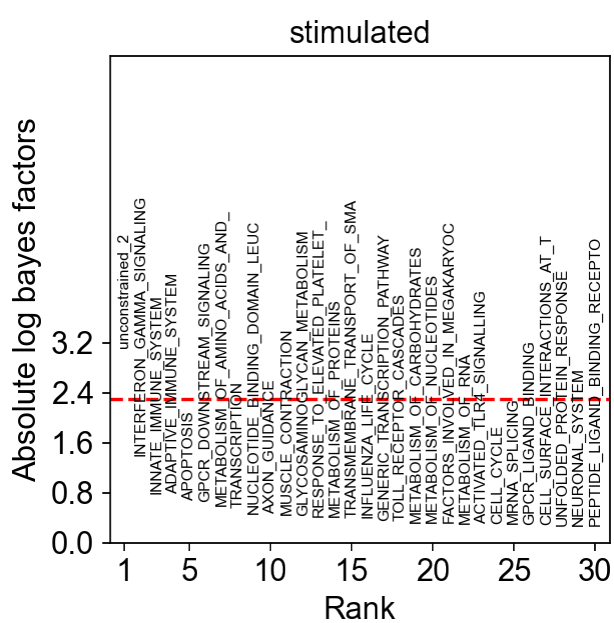
[34]:
```
q_intr_cvae.update_terms(adata=kang_pbmc)
```

[35]:
```
q_intr_cvae.latent_directions(adata=kang_pbmc)
```

Do gene set enrichment test for **condition** in reference + query using Bayes Factors.

[36]:
```
q_intr_cvae.latent_enrich(groups='condition_joint', comparison='control', use_
↪directions=True, adata=kang_pbmc)
```

[37]:
```
fig = sca.plotting.plot_abs_bfs(kang_pbmc, yt_step=0.8, scale_y=2.5, fontsize=7)
```



Do gene set enrichment test for **cell types** in reference + query control using Bayes Factors.

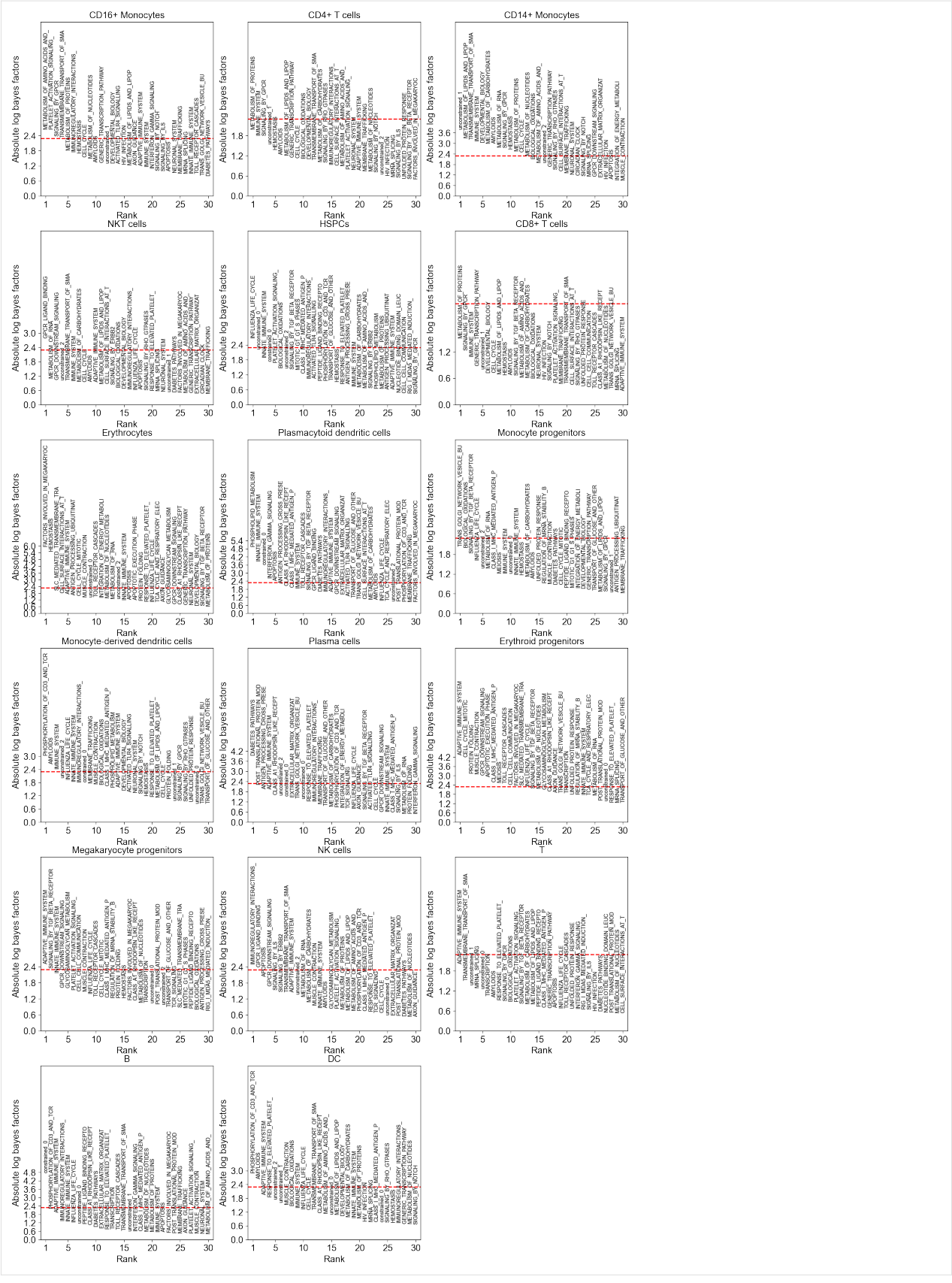
[38]:
```
kang_pbmc_control = kang_pbmc[kang_pbmc.obs['condition_joint']=='control'].copy()

q_intr_cvae.latent_enrich(groups='cell_type', use_directions=True, adata=kang_pbmc_
↪control, n_sample=10000)
```

[ ]:
```
fig = sca.plotting.plot_abs_bfs(kang_pbmc_control, n_cols=3, scale_y=2.6, yt_step=0.6)
```

[40]:
```
fig.set_size_inches(16, 34)
```

[41]:
```
fig
```

[41]:


CD16+ Monocytes
CD4+ T cells
CD14+ Monocytes
NKT cells
HSPCs
CD8+ T cells
Erythrocytes
Plasmacytoid dendritic cells
Monocyte progenitors
Monocyte-derived dendritic cells
Plasma cells
Erythroid progenitors
Megakaryocyte progenitors
NK cells
T
B
DC
Absolute log bayes factors
Rank

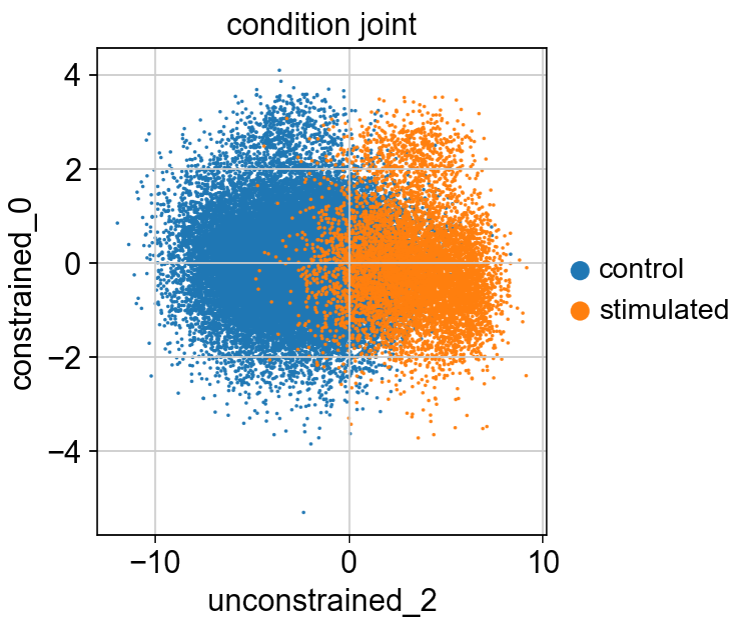
Plot the latent variables for query + reference corresponding to the constrained and unconstrained extension nodes.

[42]:
```
terms = kang_pbmc.uns['terms']
select_terms = ['constrained_0', 'unconstrained_0', 'unconstrained_1', 'unconstrained_2']
idx = [terms.index(term) for term in select_terms]
```

[43]:
```
latents = (q_intr_cvae.get_latent(kang_pbmc.X, kang_pbmc.obs['study'], mean=False) *␣
↪kang_pbmc.uns['directions'])[:, idx]
```

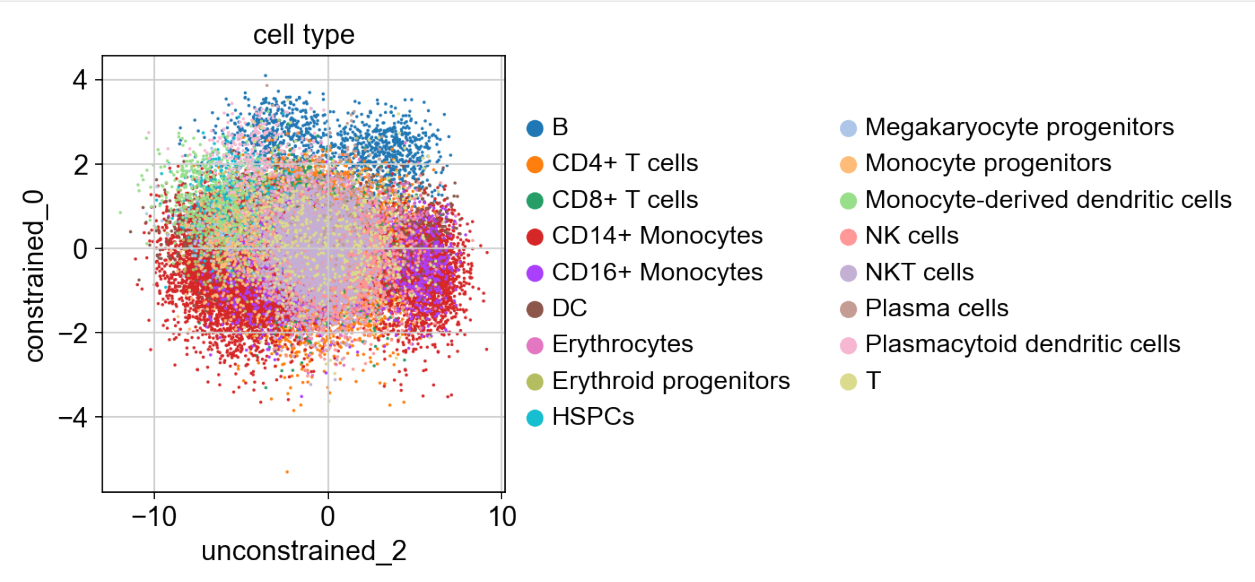
[44]:
```
kang_pbmc.obs['constrained_0'] = latents[:, 0]

kang_pbmc.obs['unconstrained_0'] = latents[:, 1]
kang_pbmc.obs['unconstrained_1'] = latents[:, 2]
kang_pbmc.obs['unconstrained_2'] = latents[:, 3]
```

[45]:
```
sc.pl.scatter(kang_pbmc, x='unconstrained_2', y='constrained_0', color='condition_joint',
↪ size=10)
```

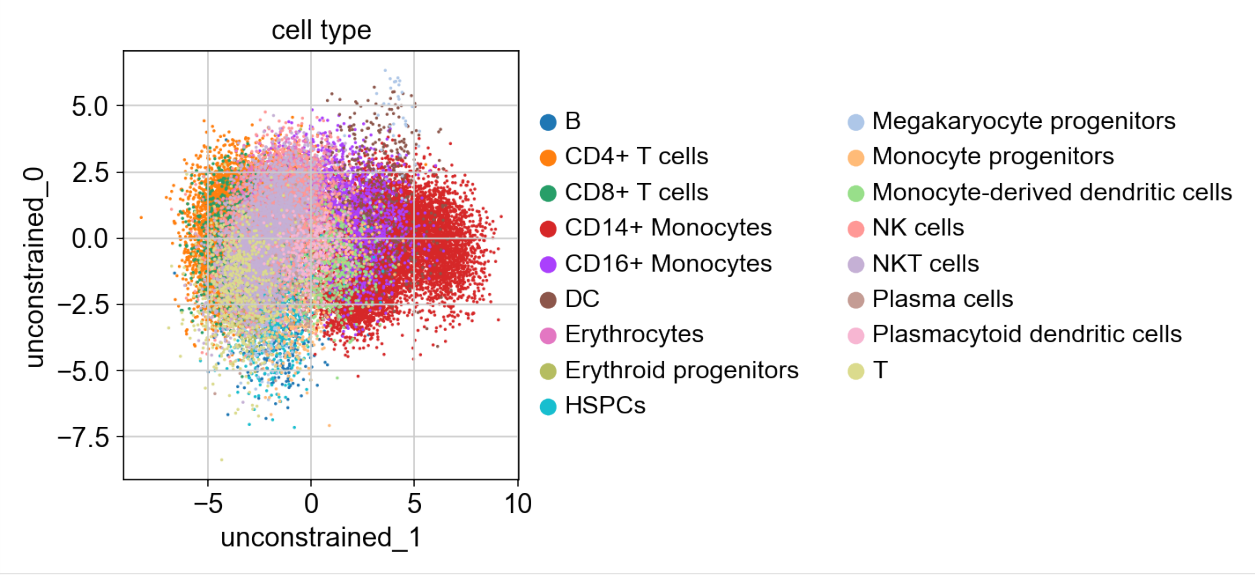


Note that the signal associated with the program learned by `unconstrained_2` was enriched in stimulated condition compared to control. Here, the cells are separated by their latent scores for `unconstrained_2`, which suggests that this node is indeed capturing the variation induced by IFN-beta stimulation.

[46]:
```
sc.pl.scatter(kang_pbmc, x='unconstrained_2', y='constrained_0', color='cell_type',␣
↪size=10)
```



[47]:
```
sc.pl.scatter(kang_pbmc, x='unconstrained_1', y='unconstrained_0', color='cell_type',␣
↪size=10)
```



Recall that `unconstrained_1` was differentially enriched in CD14+ Monocytes. Therefore, this node is capturing the CD14+ Monocyte cell type variation.

Get genes from extension nodes sorted by their absolute weights in the decoder. Higher absolute value of the weight means that this gene is affected more by the gene program.

[48]:
```
q_intr_cvae.term_genes('constrained_0', terms=kang_pbmc.uns['terms'])
```

[48]:

	genes	weights	in_mask
0	CD79A	-1.686799	True
1	BLK	-1.411135	True
2	CD19	-1.384097	True
3	BLNK	-1.166159	True
4	CD79B	-1.164143	True

(continues on next page)

(continued from previous page)

```
..       ...       ...      ...
66     STIM1 -0.000940     True
67     CERS4 -0.000227    False
68  HLA-DPB1 -0.000045    False
69     HLA-A  0.000036    False
70     KCNG1 -0.000012    False

[71 rows x 3 columns]
```

[51]:
```
q_intr_cvae.term_genes('unconstrained_1', terms=kang_pbmc.uns['terms'])
```

[51]:
```
        genes       weights  in_mask
0        RGS2  3.689368e-01    False
1        CCL2 -3.177418e-01    False
2       TIMP1 -3.089722e-01    False
3      PLA2G7 -2.966527e-01    False
4        GMPR -2.102084e-01    False
..        ...           ...      ...
486    PNPLA8  1.008977e-05    False
487    HS2ST1  8.462230e-06    False
488  SLC25A13 -7.957511e-06    False
489    SLC4A2 -3.113702e-06    False
490      EBF1  3.527966e-07    False

[491 rows x 3 columns]
```

[50]:
```
q_intr_cvae.term_genes('unconstrained_2', terms=kang_pbmc.uns['terms'])
```

[50]:
```
       genes   weights  in_mask
0      IFIT3 -0.341521    False
1      IFIT1 -0.338443    False
2      IFIT2 -0.335060    False
3     CXCL10 -0.334820    False
4      ISG15 -0.332265    False
..       ...       ...      ...
391   ZNF235 -0.000018    False
392      ILK -0.000017    False
393     RGS2  0.000017    False
394  SLC44A1  0.000006    False
395    FURIN  0.000003    False

[396 rows x 3 columns]
```

Note that `unconstrained_2` was capturing the variation induced by IFN-beta stimulation. Here, the genes from the Interferon Induced Protein gene family have the largest absolute weights in the program captured by this unconstrained node, certifying that the learned program is indeed capturing variations in gene expression due to activity of the interferon signalling, which was induced by IFN-beta stimulation.

CHAPTER

THIRTEEN

TREEARCHES: LEARNING AND UPDATING A CELL-TYPE HIERARCHY (BASIC TUTORIAL)

In this tutorial, we explain the different functionalities of treeArches. We show how to:

- *Step 1*: Integrate reference datasets using scVI
- *Step 2*: Match the cell-types in the reference datasets to learn the cell-type hierarchy of the reference datasets using scHPL
- *Step 3*: Apply architural surgery to extend the reference dataset using scArches
- *Step 4a*: Update the learned hierarchy with the cell-types from the query dataset using scHPL (useful when the query dataset is labeled)
- *Step 4b*: Predict the labels of the cells in the query dataset using scHPL (useful when the query dataset is unlabeled)

[1]:
```
import os
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
```

[2]:
```
import scanpy as sc
import torch
import scarches as sca
from scarches.dataset.trvae.data_handling import remove_sparsity
import matplotlib.pyplot as plt
import numpy as np
import gdown
import copy as cp
import seaborn as sns
```

```
Global seed set to 0
```

[3]:
```
sc.settings.set_figure_params(dpi=1000, frameon=False)
sc.set_figure_params(dpi=1000)
sc.set_figure_params(figsize=(7,7))
torch.set_printoptions(precision=3, sci_mode=False, edgeitems=7)

import matplotlib
matplotlib.rcParams['pdf.fonttype'] = 42
```

13.1 Download raw Dataset

[4]:
```
url = 'https://drive.google.com/uc?id=1Vh6RpYkusbGIZQC8GMFe3OKVDk5PWEpC'
output = 'pbmc.h5ad'
gdown.download(url, output, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1Vh6RpYkusbGIZQC8GMFe3OKVDk5PWEpC
To: /exports/humgen/lmichielsen/scArches-scHPL/PBMC/pbmc.h5ad
100%|| 2.06G/2.06G [01:37<00:00, 21.1MB/s]
```

[4]:
```
'pbmc.h5ad'
```

[5]:
```
adata = sc.read('pbmc.h5ad')
```

[6]:
```
adata.X = adata.layers["counts"].copy()
```

[7]:
```
adata = adata[adata.obs.study != "Villani"]
```

We now split the data into reference and query dataset to simulate the building process. Here we use the '10X' batch as query data.

[8]:
```
target_conditions = ["10X"]
source_adata = adata[~adata.obs.study.isin(target_conditions)].copy()
target_adata = adata[adata.obs.study.isin(target_conditions)].copy()
print(source_adata)
print(target_adata)
```

```
AnnData object with n_obs × n_vars = 21757 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
AnnData object with n_obs × n_vars = 10727 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

For a better model performance it is necessary to select HVGs. We are doing this by applying the function `scanpy.pp.highly_variable_genes()`. The parameter `n_top_genes` is set to 2000 here. However, for more complicated datasets you might have to increase number of genes to capture more diversity in the data.

[9]:
```
source_adata.raw = source_adata
```

[10]:
```
source_adata
```

[10]:
```
AnnData object with n_obs × n_vars = 21757 × 12303
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

[11]:
```
sc.pp.normalize_total(source_adata)
```

[12]:
```
sc.pp.log1p(source_adata)
```

[13]:
```
sc.pp.highly_variable_genes(
    source_adata,
    n_top_genes=2000,
    batch_key="study",
    subset=True)
```

For consistency we set adata.X to be raw counts. In other datasets that may be already the case

[14]:
```
source_adata.X = source_adata.raw[:, source_adata.var_names].X
```

13.2 Create scVI model and train it on reference dataset

Remember: The adata object has to have count data in adata.X for scVI/scANVI if not further specified.

[15]:
```
sca.models.SCVI.setup_anndata(source_adata, batch_key="batch")
```

The scVI model uses the zero-inflated negative binomial (ZINB) loss by default. Insert `gene_likelihood='nb'` to change the reconstruction loss to negative binomial (NB) loss.

[16]:
```
vae = sca.models.SCVI(
    source_adata,
    n_layers=2,
    encode_covariates=True,
    deeply_inject_covariates=True,
    use_layer_norm="both",
    use_batch_norm="none",
)
```

[17]:
```
vae.train(max_epochs=80)
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 80/80: 100%|| 80/80 [01:34<00:00,  1.18s/it, loss=565, v_num=1]
```

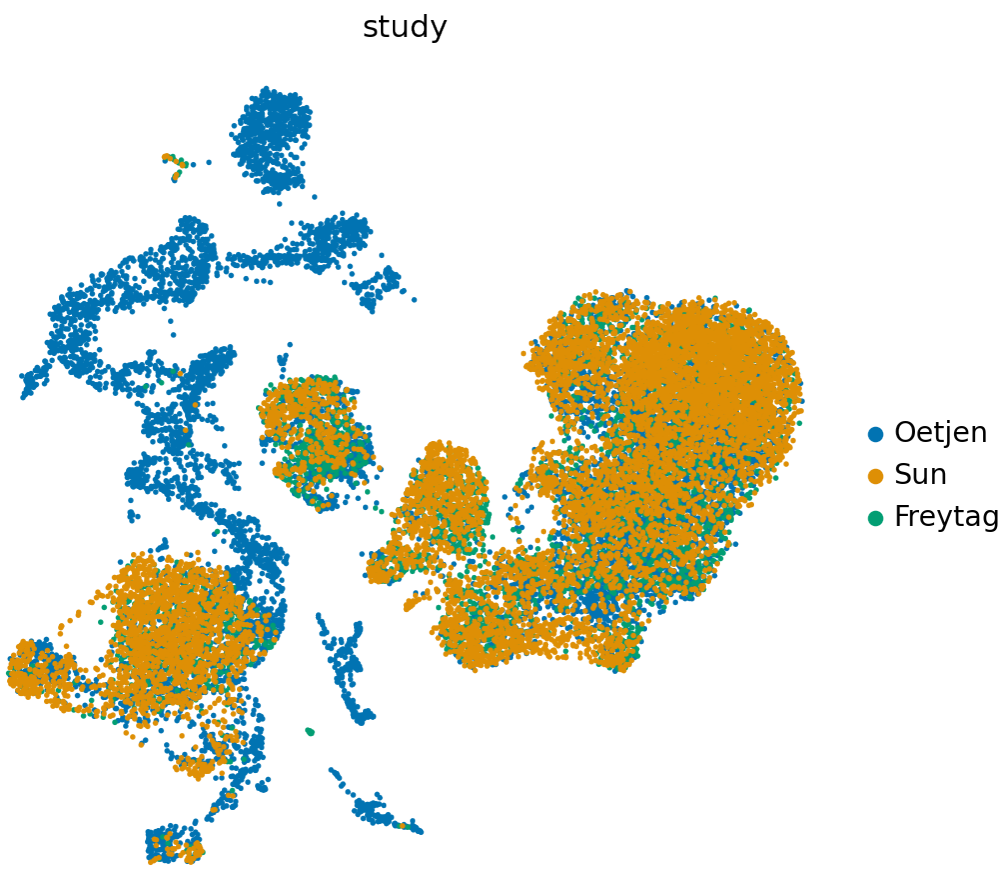
The resulting latent representation of the data can then be visualized with UMAP

[18]:
```
reference_latent = sc.AnnData(vae.get_latent_representation())
reference_latent.obs["cell_type"] = source_adata.obs["final_annotation"].tolist()
reference_latent.obs["batch"] = source_adata.obs["batch"].tolist()
reference_latent.obs["study"] = source_adata.obs["study"].tolist()
```

[19]:
```
sc.pp.neighbors(reference_latent, n_neighbors=8)
sc.tl.leiden(reference_latent)
sc.tl.umap(reference_latent)
```

[20]:
```
reference_latent.obs['study'] = reference_latent.obs['study'].astype('category')

# Reorder categories, so smallest dataset is plotted on top
reference_latent.obs['study'].cat.reorder_categories(['Oetjen', 'Sun', 'Freytag'],␣
↪inplace=True)
```

[21]:
```
sc.pl.umap(reference_latent,
           color=['study'],
           frameon=False,
           wspace=0.6, s=25,
           palette=sns.color_palette('colorblind', as_cmap=True)[:3]
           )
```



The colorblind color map only contains 10 different colors. To visualize the different cell-types, we rename some cells to a lower resolution.

[22]:
```
reference_latent.obs['ct_low'] = 0

idx = ((reference_latent.obs.cell_type == 'CD4+ T cells') |
       (reference_latent.obs.cell_type == 'CD8+ T cells'))
reference_latent.obs['ct_low'][idx] = 'T cells'

idx = ((reference_latent.obs.cell_type == 'CD10+ B cells') |
```

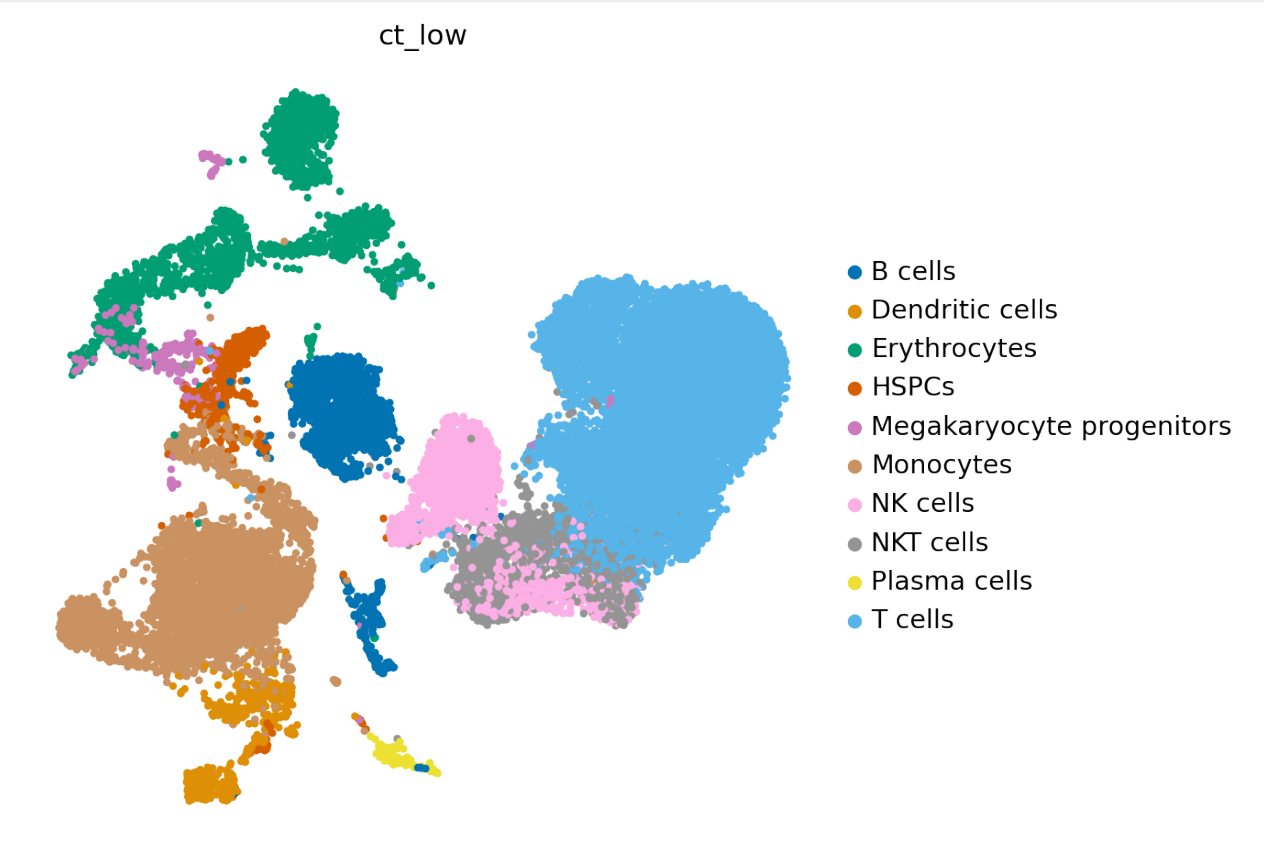
(continues on next page)

(continued from previous page)

```
       (reference_latent.obs.cell_type == 'CD20+ B cells'))
reference_latent.obs['ct_low'][idx] = 'B cells'

idx = ((reference_latent.obs.cell_type == 'CD14+ Monocytes') |
       (reference_latent.obs.cell_type == 'CD16+ Monocytes') |
       (reference_latent.obs.cell_type == 'Monocyte progenitors'))
reference_latent.obs['ct_low'][idx] = 'Monocytes'

idx = ((reference_latent.obs.cell_type == 'Erythrocytes') |
       (reference_latent.obs.cell_type == 'Erythroid progenitors'))
reference_latent.obs['ct_low'][idx] = 'Erythrocytes'

idx = ((reference_latent.obs.cell_type == 'Monocyte-derived dendritic cells') |
       (reference_latent.obs.cell_type == 'Plasmacytoid dendritic cells'))
reference_latent.obs['ct_low'][idx] = 'Dendritic cells'

idx = reference_latent.obs.cell_type == 'HSPCs'
reference_latent.obs['ct_low'][idx] = 'HSPCs'

idx = reference_latent.obs.cell_type == 'Megakaryocyte progenitors'
reference_latent.obs['ct_low'][idx] = 'Megakaryocyte progenitors'

idx = reference_latent.obs.cell_type == 'NK cells'
reference_latent.obs['ct_low'][idx] = 'NK cells'

idx = reference_latent.obs.cell_type == 'NKT cells'
reference_latent.obs['ct_low'][idx] = 'NKT cells'

idx = reference_latent.obs.cell_type == 'Plasma cells'
reference_latent.obs['ct_low'][idx] = 'Plasma cells'
```

```
/tmp/ipykernel_1063462/4260835584.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  reference_latent.obs['ct_low'][idx] = 'T cells'
```

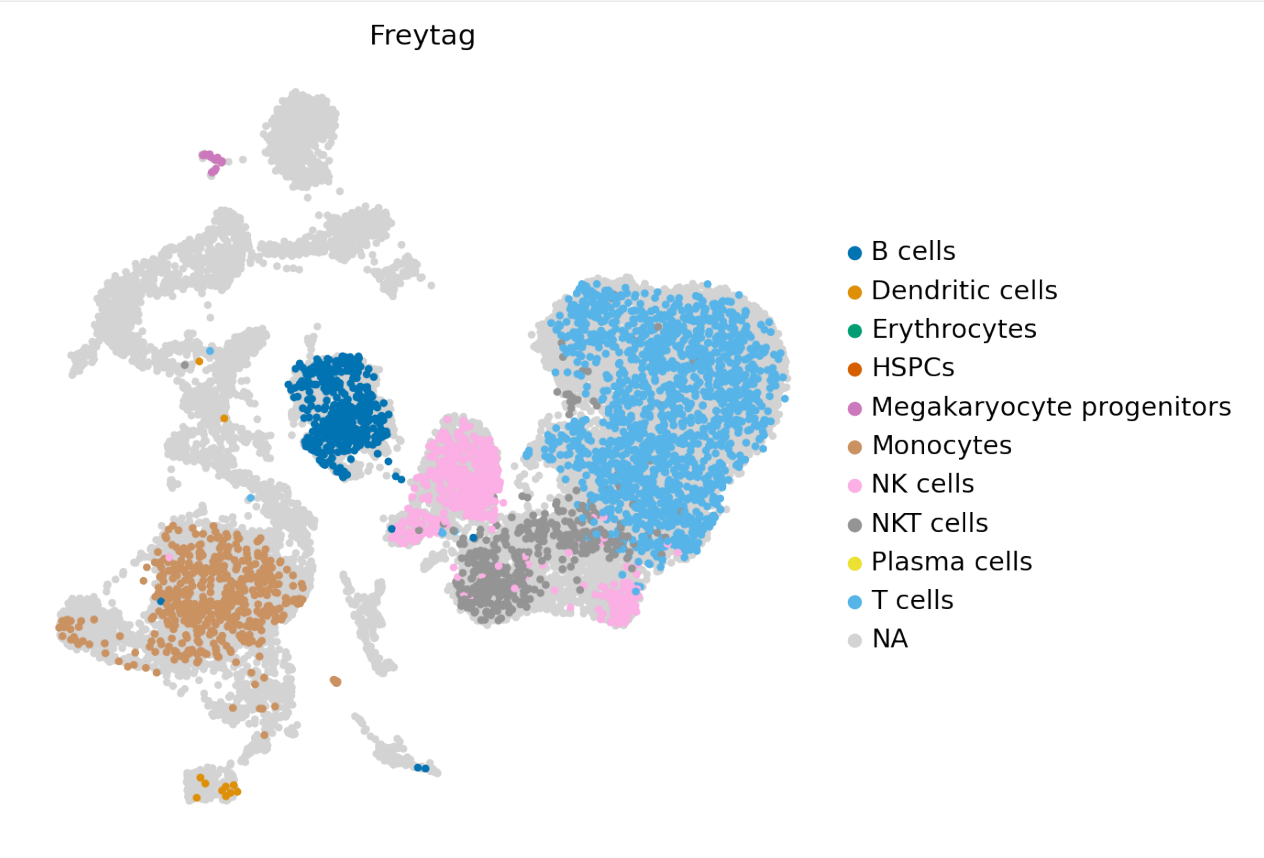
[23]:
```
sc.pl.umap(reference_latent,
           color=['ct_low'],
           frameon=False,
           wspace=0.6, s=60,
           palette=sns.color_palette('colorblind', as_cmap=True)
           )
```



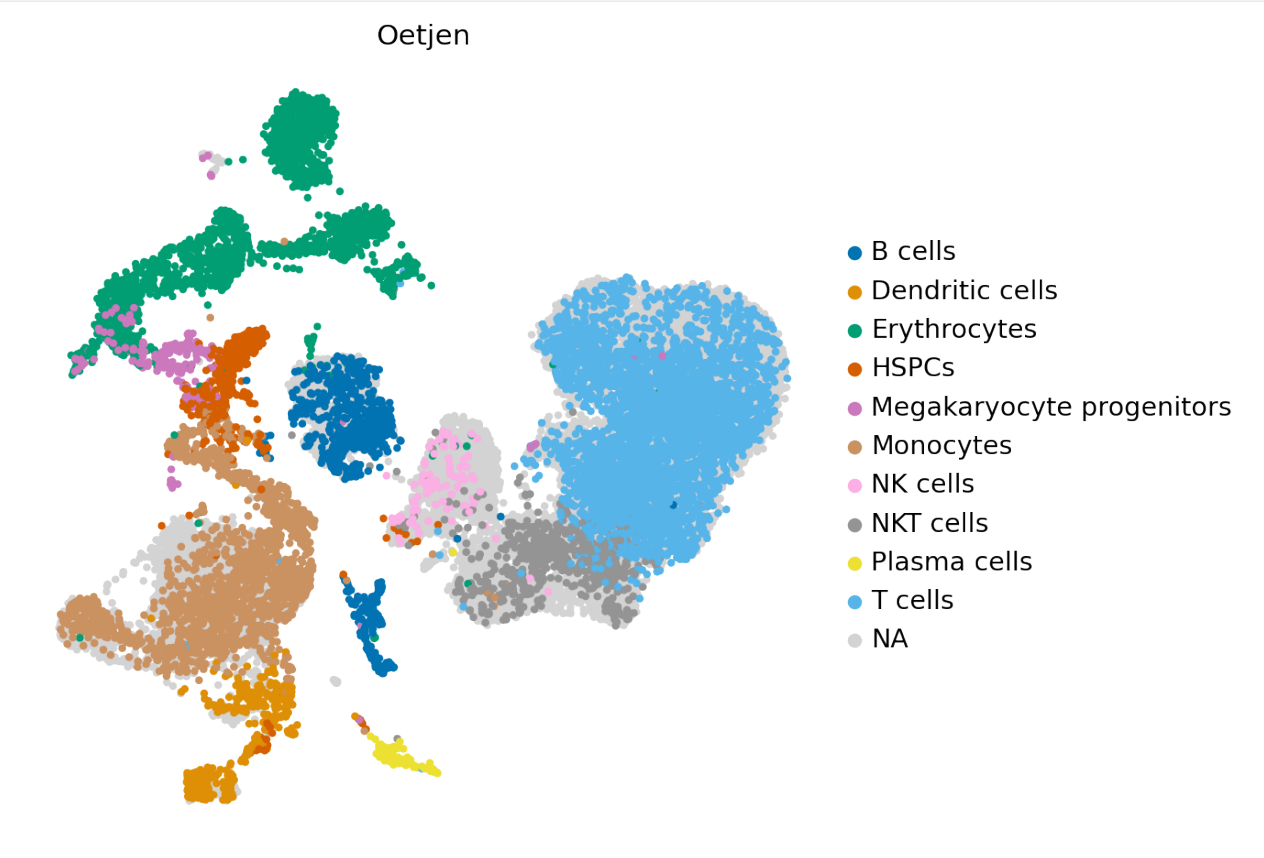
We can also visualize the cell-types per dataset.

[24]:
```
for s in np.unique(reference_latent.obs.study):
    ref_s = cp.deepcopy(reference_latent)
    ref_s.obs.ct_low[reference_latent.obs.study != s] = np.nan

    sc.pl.umap(ref_s,
               color=['ct_low'],
               frameon=False,
               wspace=0.6, s=60,
               palette=sns.color_palette('colorblind', as_cmap=True), title=s,
               save=s+'.pdf'
               )
```

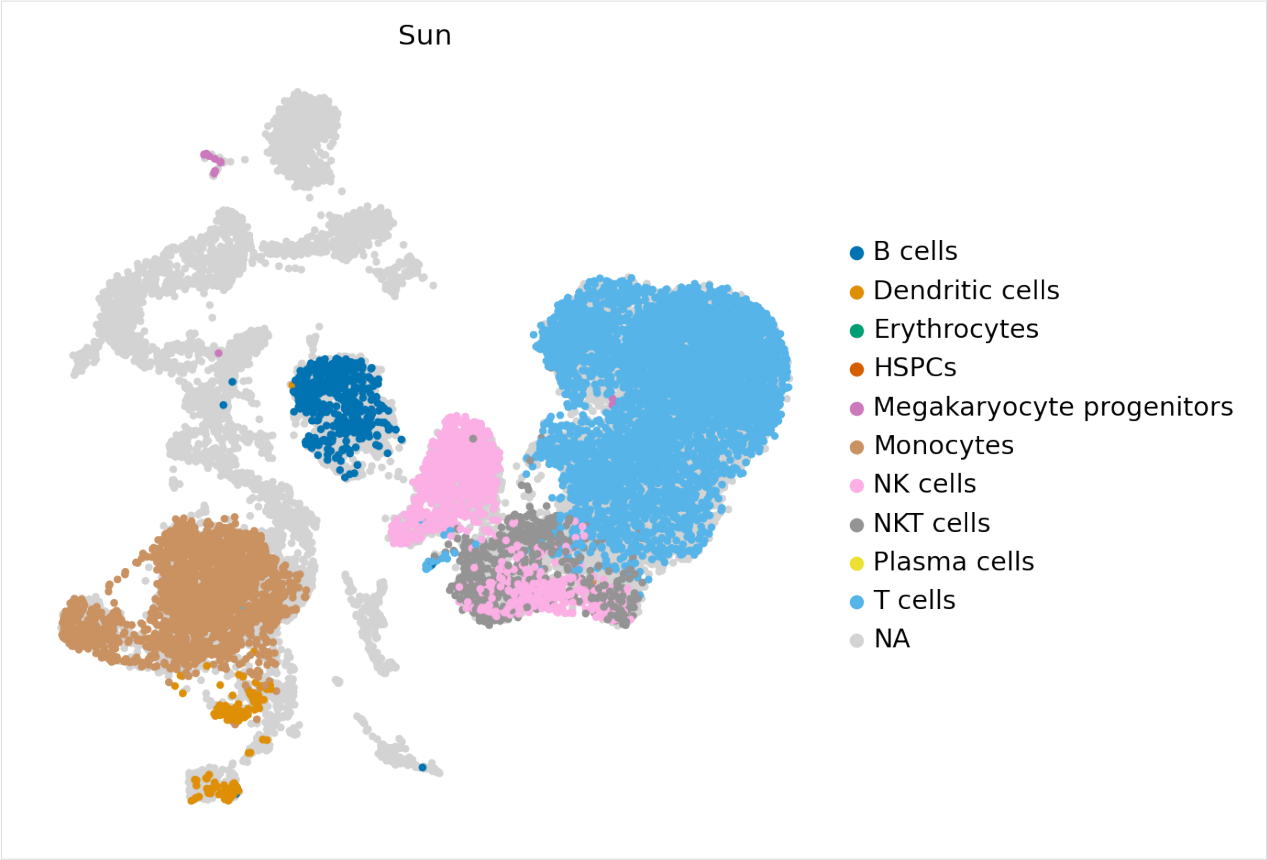
```
WARNING: saving figure to file figures/umapFreytag.pdf
```




```
WARNING: saving figure to file figures/umapOetjen.pdf
```




```
WARNING: saving figure to file figures/umapSun.pdf
```



After pretraining the model can be saved for later use or also be uploaded for other researchers with via Zenodo. For the second option please also have a look at the Zenodo notebook.

[25]:
```
ref_path = 'ref_model/'
vae.save(ref_path, overwrite=True)
reference_latent.write(ref_path + 'ref_latent.h5ad')
```

13.3 Construct hierarchy for the reference using scHPL

First, we concatenate all cell type labels with the study labels. This way, we ensure that the cell types of the different studies are seen as unique.

> Warning: Always ensure that the cell type labels of each dataset are unique!

[26]:
```
reference_latent.obs['celltype_batch'] = np.char.add(np.char.add(np.array(reference_
↪latent.obs['cell_type'], dtype= str), '-'),
                                                    np.array(reference_latent.obs['study'],␣
↪dtype=str))
```

Now, we are ready to learn the cell-type hierarchy. In this example we use the `classifier='knn'`, this can be changed to either a linear SVM (`'svm'`) or a one-class SVM (`'svm_occ'`). We recommend to use the kNN classifier when the dimensionality is low since the cell-types are not linearly separable anymore.

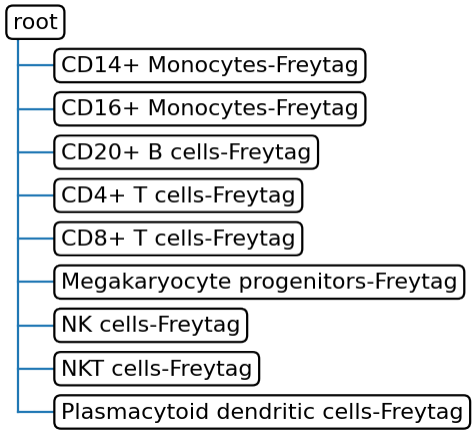
The option `dynamic_neighbors=True` implies that the number of neighbors changes depending on the number of cells in the dataset. If a cell-type is small, the number of neighbors used will also be lower. The number of neighbors can also be set manually using `n_neighbors`.

During each step of scHPL, a classifier is trained on the datasets we want to match and the labels are cross-predicted. If you're interested in the confusion matrices used for the matching, set `print_conf=True`. The confusion matrices are also saved to .csv files then.

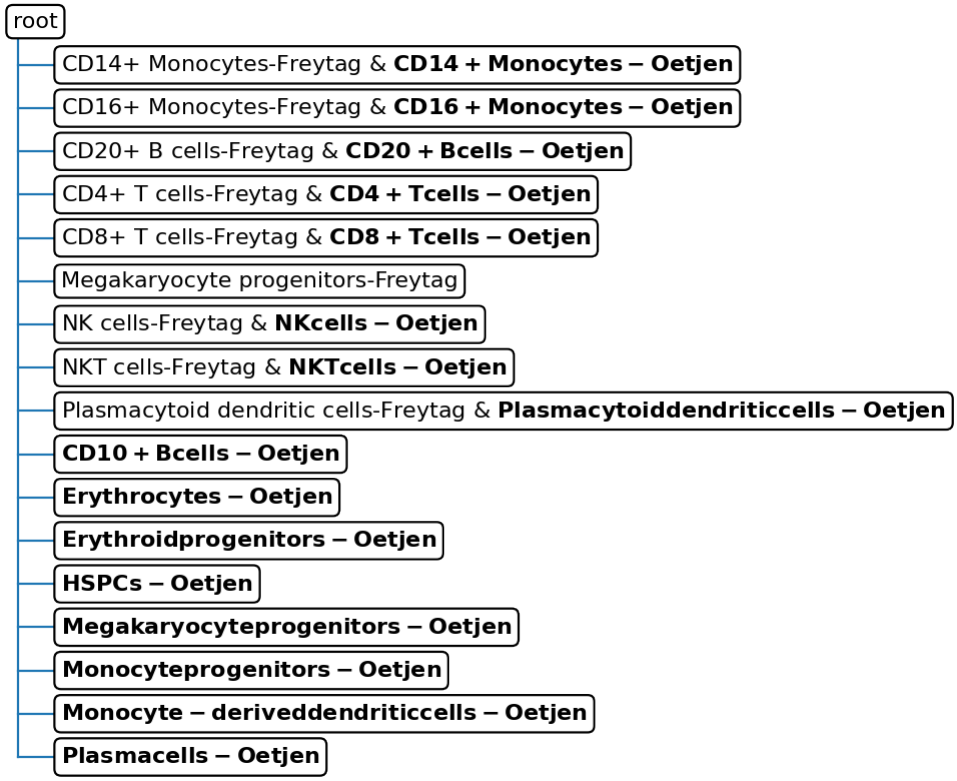
For more details about other parameters, take a look at the scHPL GitHub

[27]:
```
tree_ref, mp_ref = sca.classifiers.scHPL.learn_tree(data = reference_latent,
                batch_key = 'study',
                batch_order = ['Freytag', 'Oetjen', 'Sun'],
                cell_type_key='celltype_batch',
                classifier = 'knn', dynamic_neighbors=True,
                dimred = False, print_conf= False)
```

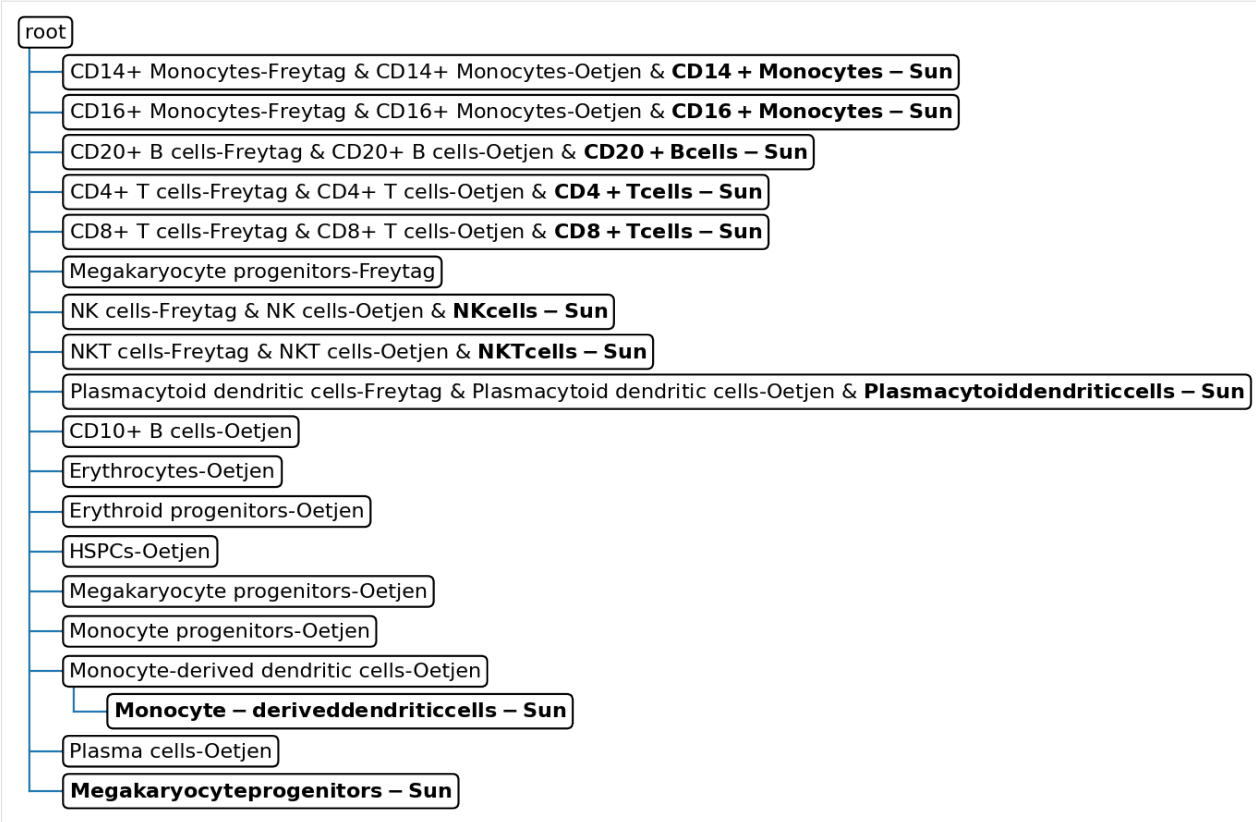
```
Starting tree:
```




```
Adding dataset Oetjen to the tree

Updated tree:
```




```
Adding dataset Sun to the tree

Updated tree:
```



13.4 Use pretrained reference model and apply surgery with a new query dataset to get a bigger reference atlas

Since the model requires the datasets to have the same genes we also filter the query dataset to have the same genes as the reference dataset.

[28]:
```
target_adata = target_adata[:, source_adata.var_names]
target_adata
```

[28]:
```
View of AnnData object with n_obs × n_vars = 10727 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue'
    layers: 'counts'
```

We then can apply the model surgery with the new query dataset:

[29]:
```
target_adata = target_adata.copy()
```

[30]:
```
model = sca.models.SCVI.load_query_data(
    target_adata,
    ref_path,
    freeze_dropout = True,
)
```

```
INFO     File ref_model/model.pt already downloaded
```

[31]:
```
model.train(max_epochs=50)
```

```
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 50/50: 100%|| 50/50 [00:26<00:00,  1.89it/s, loss=975, v_num=1]
```

[32]:
```
query_latent = sc.AnnData(model.get_latent_representation())
query_latent.obs['cell_type'] = target_adata.obs["final_annotation"].tolist()
query_latent.obs['batch'] = target_adata.obs["batch"].tolist()
```

And again we can save or upload the retrained model for later use or additional extensions.

[33]:
```
surgery_path = 'surgery_model'
model.save(surgery_path, overwrite=True)
query_latent.write('query_latent.h5ad')
```

Get latent representation of reference + query dataset and compute UMAP

[34]:
```
target_adata.obs.study = "10X"
```

[35]:
```
target_adata
```

[35]:
```
AnnData object with n_obs × n_vars = 10727 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue
↪', '_scvi_batch', '_scvi_labels'
    uns: '_scvi_uuid', '_scvi_manager_uuid'
    layers: 'counts'
```

[36]:
```
adata_full = source_adata.concatenate(target_adata, batch_key="ref_query")
adata_full
```

[36]:
```
AnnData object with n_obs × n_vars = 32484 × 2000
    obs: 'batch', 'chemistry', 'data_type', 'dpt_pseudotime', 'final_annotation', 'mt_
↪frac', 'n_counts', 'n_genes', 'sample_ID', 'size_factors', 'species', 'study', 'tissue
↪', '_scvi_batch', '_scvi_labels', 'ref_query'
    var: 'highly_variable-0', 'means-0', 'dispersions-0', 'dispersions_norm-0', 'highly_
↪variable_nbatches-0', 'highly_variable_intersection-0'
    layers: 'counts'
```

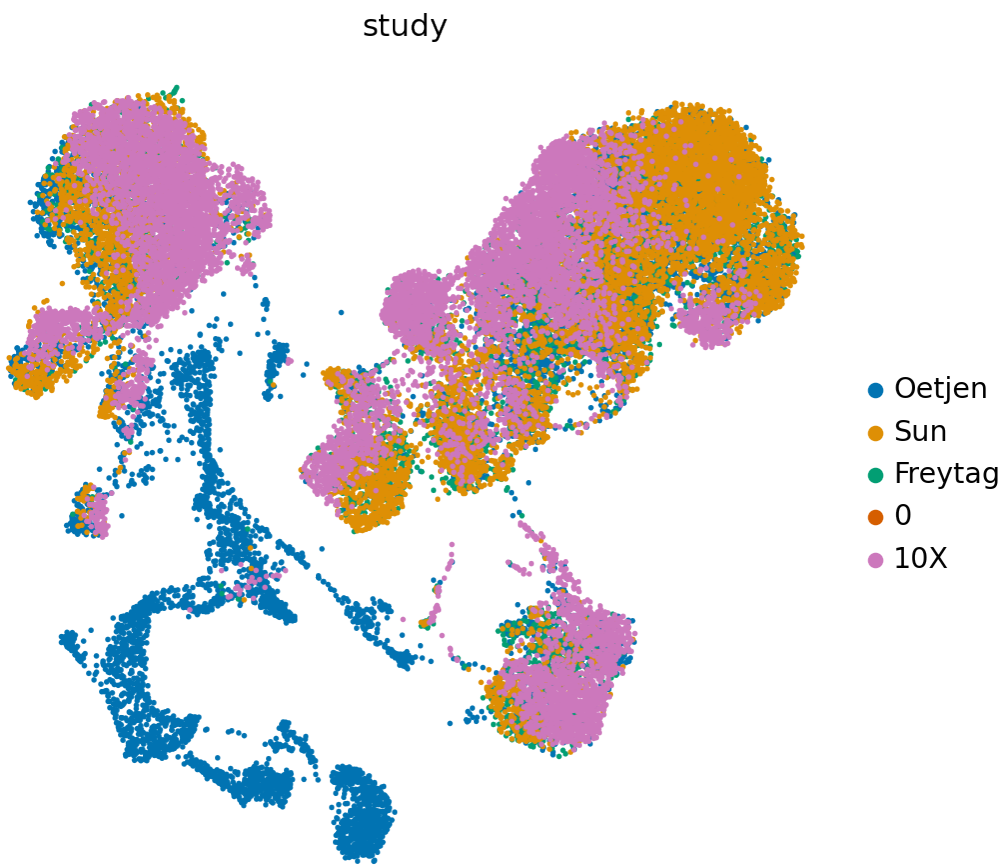
[37]:
```
full_latent = sc.AnnData(model.get_latent_representation(adata=adata_full))
full_latent.obs['cell_type'] = adata_full.obs["final_annotation"].tolist()
full_latent.obs['batch'] = adata_full.obs["batch"].tolist()
full_latent.obs['study'] = adata_full.obs["study"].tolist()
```

```
INFO     Input AnnData not setup with scvi-tools. attempting to transfer AnnData setup
```

[38]:
```
sc.pp.neighbors(full_latent)
sc.tl.leiden(full_latent)
sc.tl.umap(full_latent)
```

[39]:
```
full_latent.obs['study'] = full_latent.obs['study'].astype('category')
full_latent.obs['study'].cat.add_categories(['0'], inplace=True)
full_latent.obs['study'].cat.reorder_categories(['Oetjen', 'Sun', 'Freytag', '0', '10X'],
↪ inplace=True)

sc.pl.umap(full_latent,
           color=['study'],
           frameon=False,
           wspace=0.6, s=25,
           palette=sns.color_palette('colorblind', as_cmap=True)[:5],
           save='study_query.pdf'
           )
```

```
WARNING: saving figure to file figures/umapstudy_query.pdf
```



[40]:
```
full_latent.obs['ct_low'] = 0

idx = ((full_latent.obs.cell_type == 'CD4+ T cells') |
       (full_latent.obs.cell_type == 'CD8+ T cells'))
```

(continues on next page)

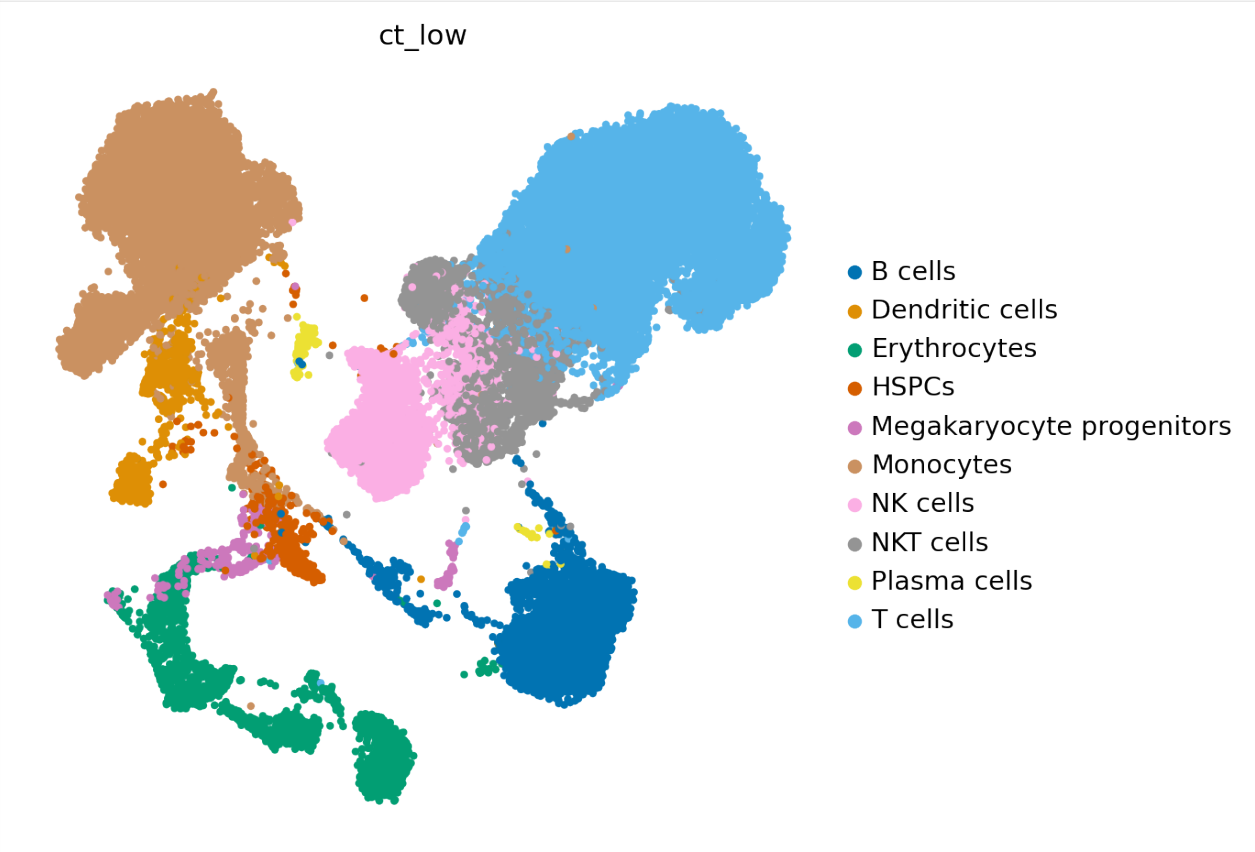
(continued from previous page)

```
full_latent.obs['ct_low'][idx] = 'T cells'

idx = ((full_latent.obs.cell_type == 'CD10+ B cells') |
       (full_latent.obs.cell_type == 'CD20+ B cells'))
full_latent.obs['ct_low'][idx] = 'B cells'

idx = ((full_latent.obs.cell_type == 'CD14+ Monocytes') |
       (full_latent.obs.cell_type == 'CD16+ Monocytes') |
       (full_latent.obs.cell_type == 'Monocyte progenitors'))
full_latent.obs['ct_low'][idx] = 'Monocytes'

idx = ((full_latent.obs.cell_type == 'Erythrocytes') |
       (full_latent.obs.cell_type == 'Erythroid progenitors'))
full_latent.obs['ct_low'][idx] = 'Erythrocytes'

idx = ((full_latent.obs.cell_type == 'Monocyte-derived dendritic cells') |
       (full_latent.obs.cell_type == 'Plasmacytoid dendritic cells'))
full_latent.obs['ct_low'][idx] = 'Dendritic cells'

idx = full_latent.obs.cell_type == 'HSPCs'
full_latent.obs['ct_low'][idx] = 'HSPCs'

idx = full_latent.obs.cell_type == 'Megakaryocyte progenitors'
full_latent.obs['ct_low'][idx] = 'Megakaryocyte progenitors'

idx = full_latent.obs.cell_type == 'NK cells'
full_latent.obs['ct_low'][idx] = 'NK cells'

idx = full_latent.obs.cell_type == 'NKT cells'
full_latent.obs['ct_low'][idx] = 'NKT cells'

idx = full_latent.obs.cell_type == 'Plasma cells'
full_latent.obs['ct_low'][idx] = 'Plasma cells'
```

```
/tmp/ipykernel_1063462/1546695568.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  full_latent.obs['ct_low'][idx] = 'T cells'
```

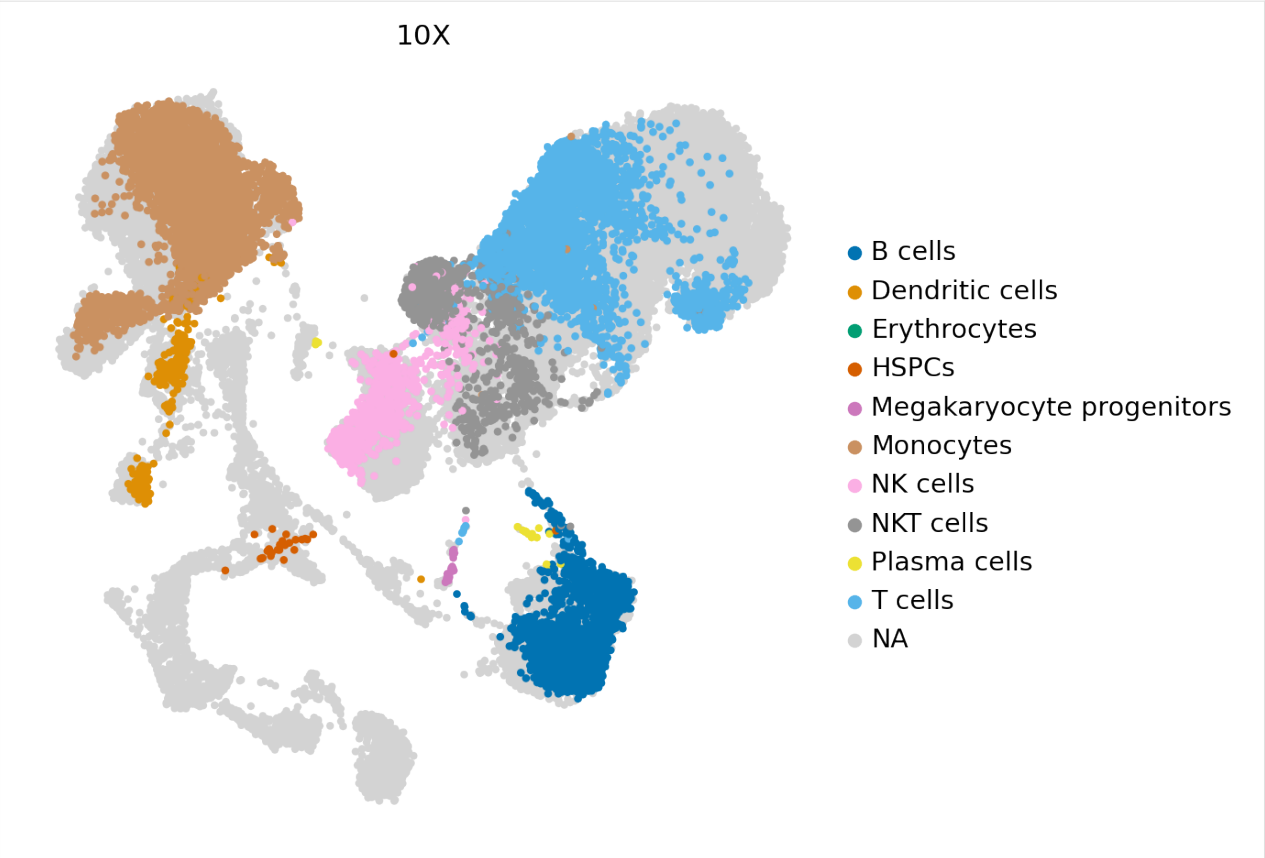
[41]:
```
sc.pl.umap(full_latent,
           color=['ct_low'],
           frameon=False,
           wspace=0.6, s=60,
           palette=sns.color_palette('colorblind', as_cmap=True),
           save='cp_query.pdf'
           )
```

```
WARNING: saving figure to file figures/umapcp_query.pdf
```

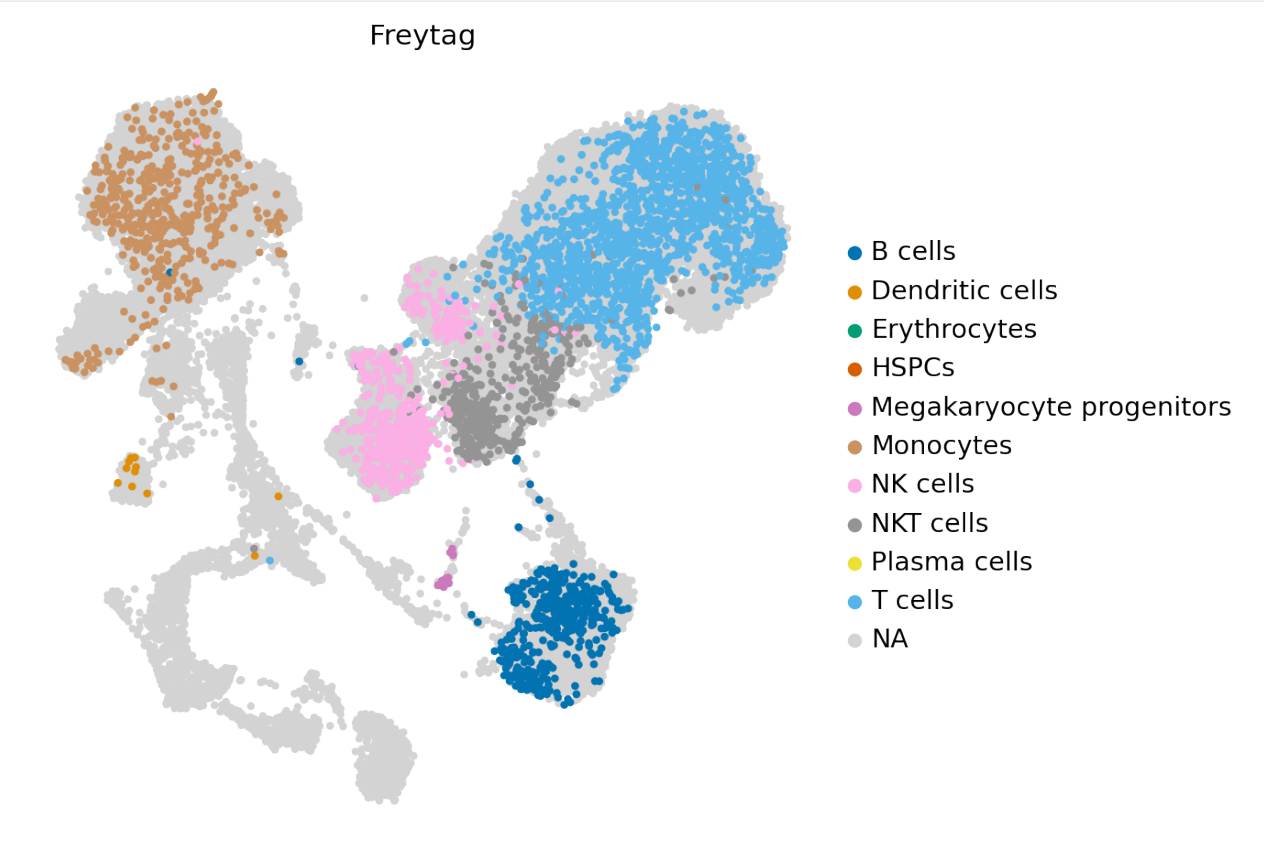


[42]:
```
for s in np.unique(full_latent.obs.study):
    ref_s = cp.deepcopy(full_latent)
    ref_s.obs.ct_low[full_latent.obs.study != s] = np.nan

    sc.pl.umap(ref_s,
               color=['ct_low'],
               frameon=False,
               wspace=0.6, s=60,
               palette=sns.color_palette('colorblind', as_cmap=True), title=s,
               save=s+'_query.pdf'
               )
```

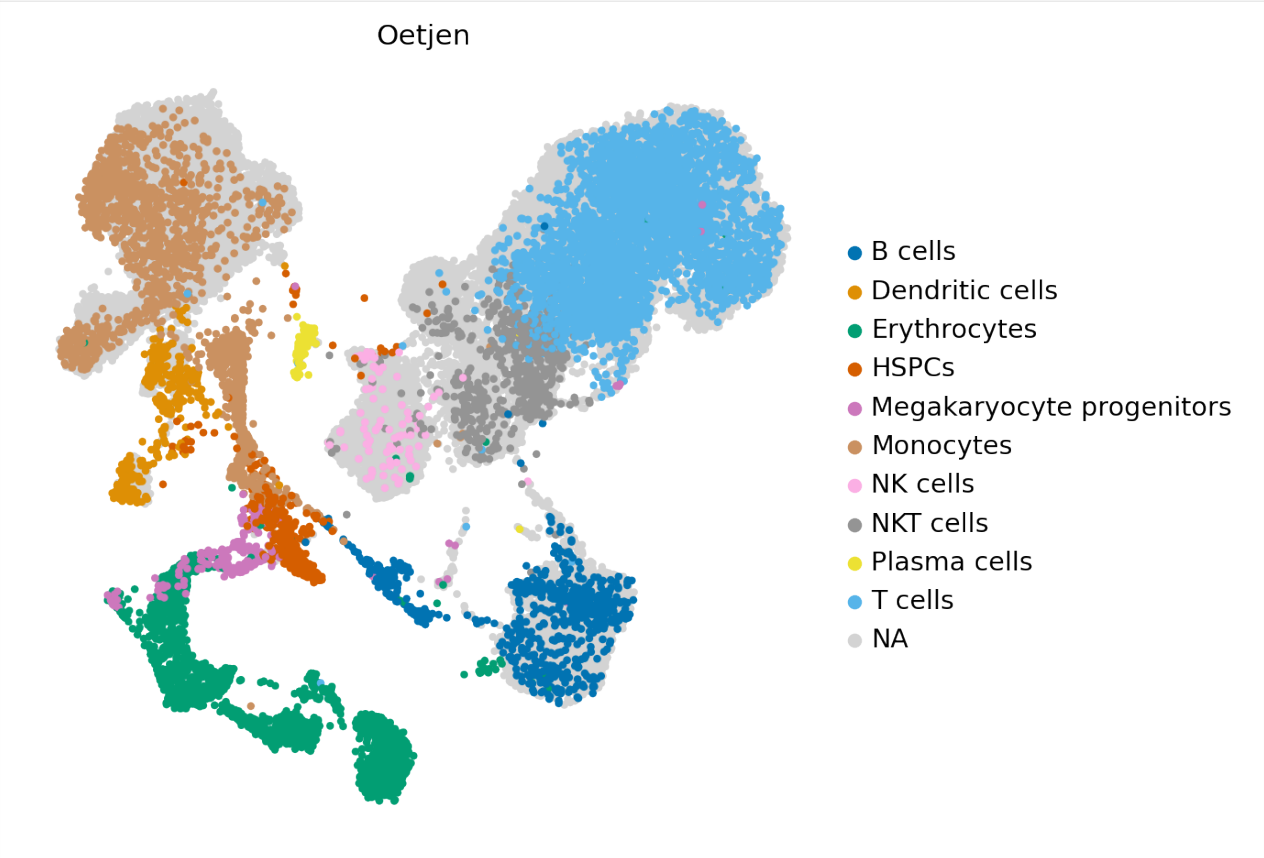
```
WARNING: saving figure to file figures/umap10X_query.pdf
```



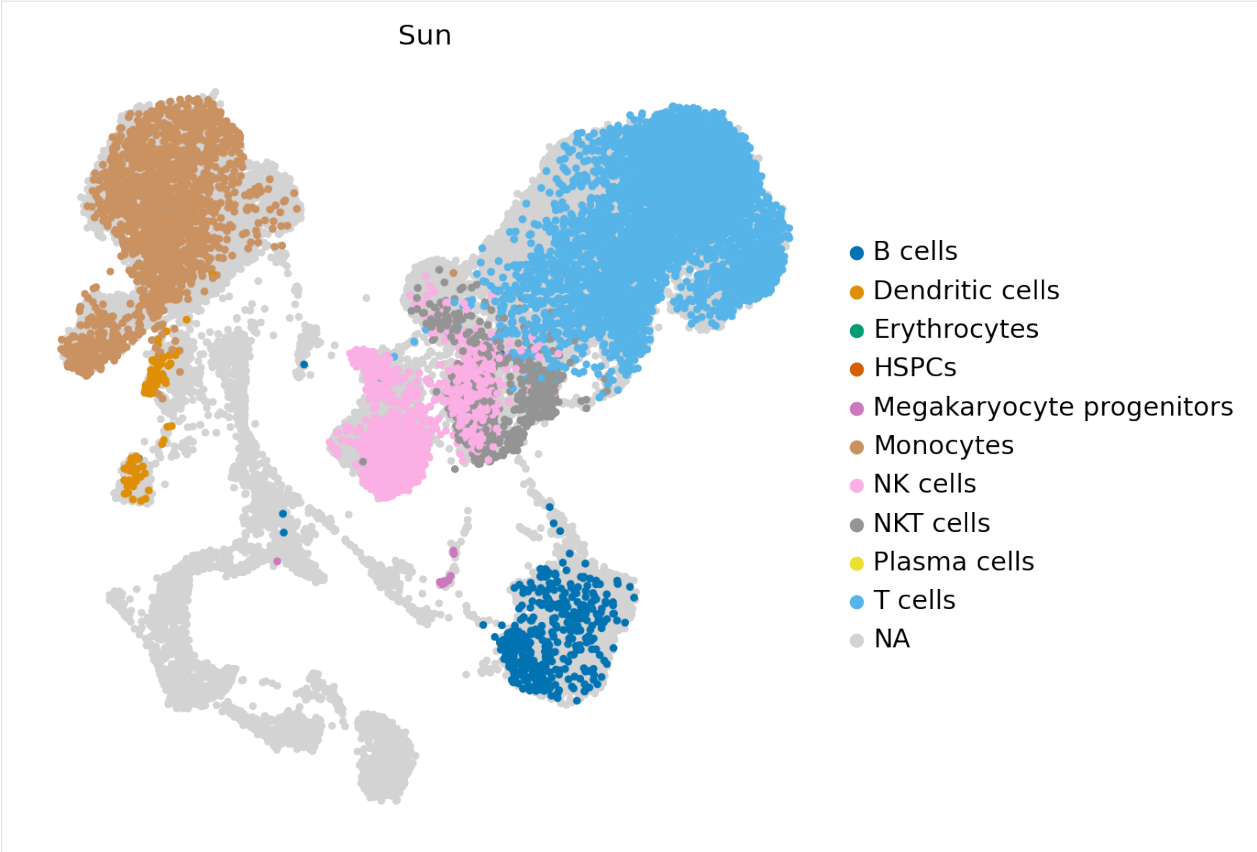

```
WARNING: saving figure to file figures/umapFreytag_query.pdf
```




```
WARNING: saving figure to file figures/umapOetjen_query.pdf
```




```
WARNING: saving figure to file figures/umapSun_query.pdf
```



13.5 Updating the hierarchy using scHPL

If the cells in the query dataset are labeled, we can update the hierarchy using scHPL. If the cells are unlabeled, we can predict their label (see section below).

Again, we first have to ensure that the labels of the cell-types are unique

[43]:
```
full_latent.obs['celltype_batch'] = np.char.add(np.char.add(np.array(full_latent.obs[
↪'cell_type'], dtype= str), '-'),
                                                np.array(full_latent.obs['study'],␣
↪dtype=str))
```

Now, we are ready to update the cell-type hierarchy. It is important to use the same classifier settings here as used before. Furthermore, it is important to indicate which batches are already in the tree (`batch_added`) and which you want to add to the tree (`batch_order`).

[44]:
```
# First make a deep copy of the original classifier to ensure we do not overwrite it
tree_rq = cp.deepcopy(tree_ref)

tree_rq, mp_rq = sca.classifiers.scHPL.learn_tree(data = full_latent, batch_key = 'study
↪',
                  batch_order = ['10X'],
                  batch_added = ['Oetjen', 'Freytag', 'Sun'],
                  cell_type_key='celltype_batch',
```

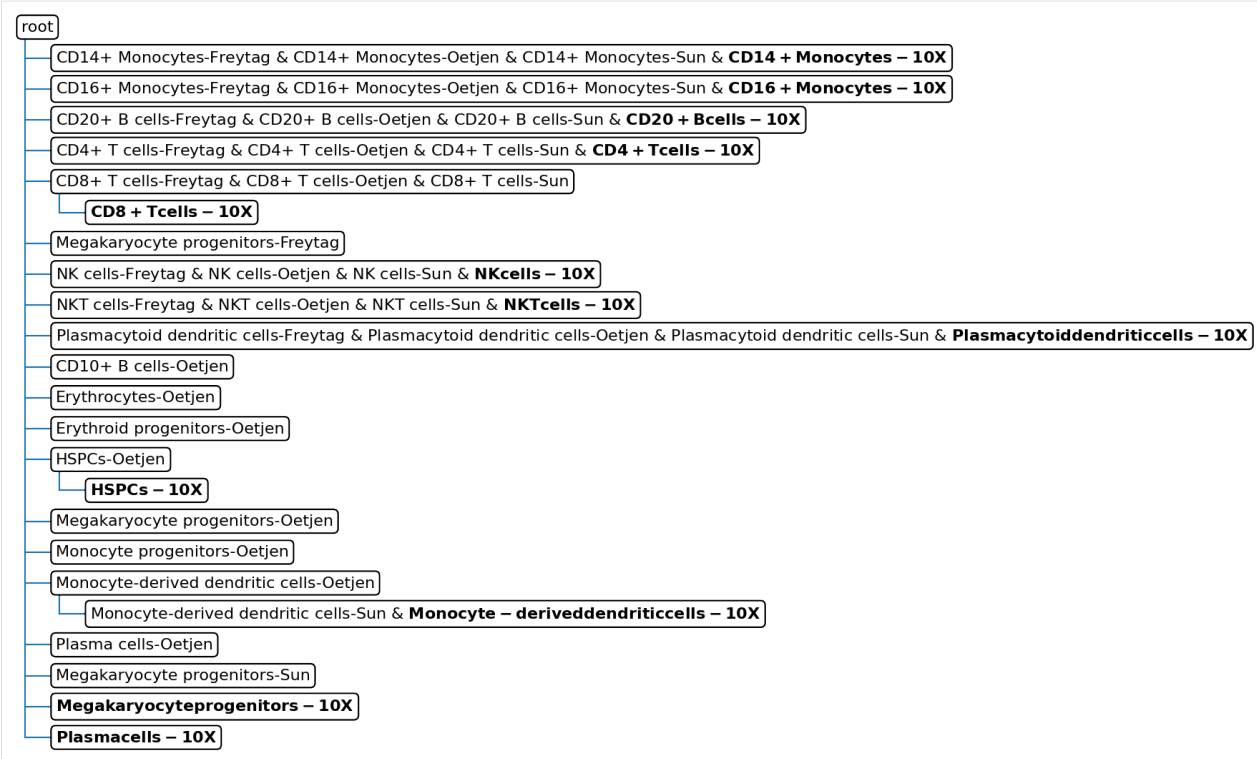
(continues on next page)

(continued from previous page)

```
                 tree = tree_rq, retrain = False,
                classifier = 'knn',
                dimred = False)
```

```
Starting tree:
```

```
Adding dataset 10X to the tree

Updated tree:
```



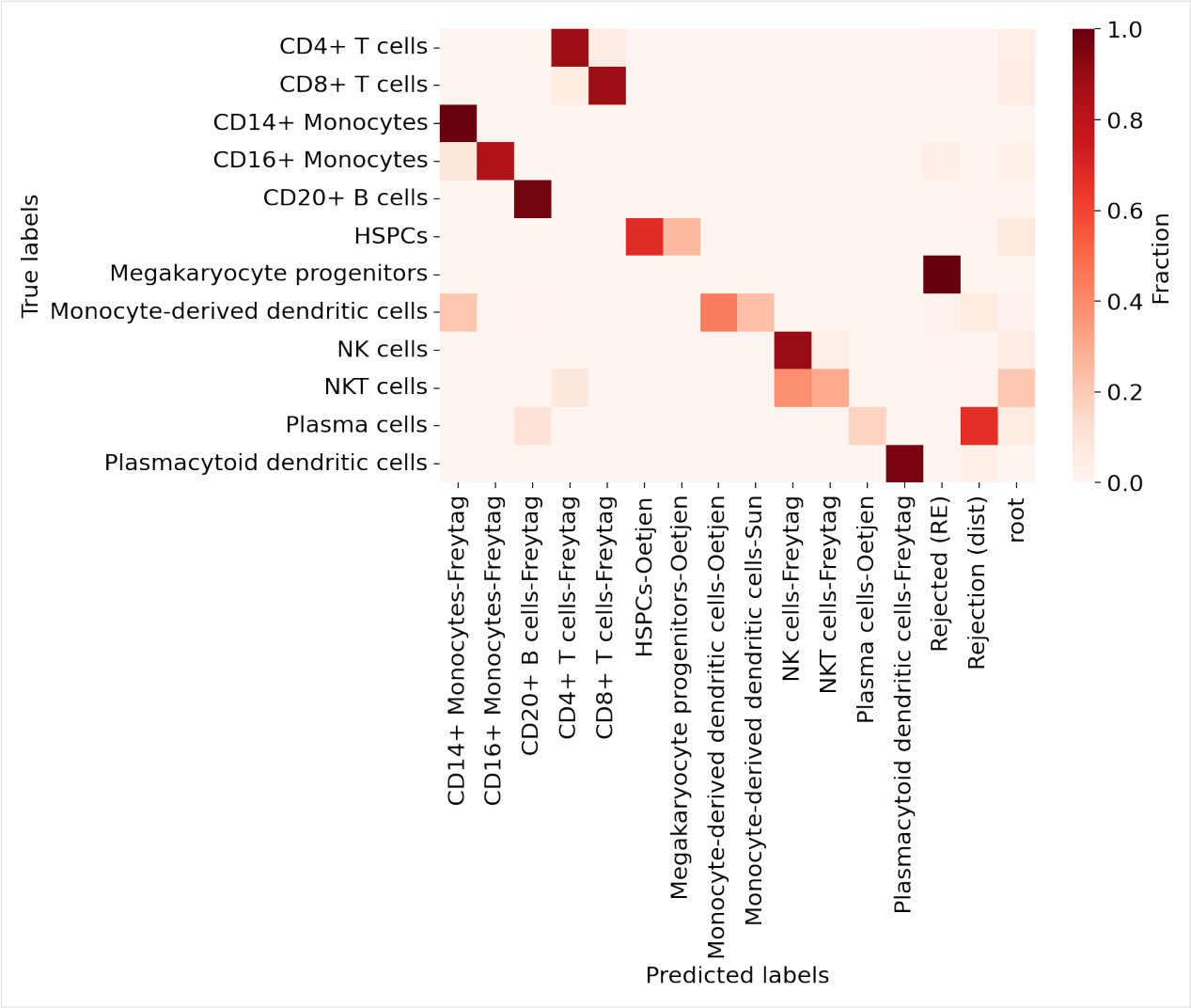
13.6 Predicting cell-type labels using scHPL

If the cells in the query dataset are unlabeled or if you're interested in comparing the transferred labels to your own annotations without updating the hierarchy, you can predict the labels with scHPL.

[45]:
```
query_pred = sca.classifiers.scHPL.predict_labels(query_latent.X, tree=tree_ref)
```

Using the `evaluate.heatmap()` function, the predictions can be compared to other annotations

[46]:
```
sca.classifiers.scHPL.evaluate.heatmap(query_latent.obs['cell_type'], query_pred,␣
↪shape=[8,5])
```


CD4+ T cells
CD8+ T cells
CD14+ Monocytes
CD16+ Monocytes
CD20+ B cells
HSPCs
Megakaryocyte progenitors
Monocyte-derived dendritic cells
NK cells
NKT cells
Plasma cells
Plasmacytoid dendritic cells
True labels
CD14+ Monocytes-Freytag
CD16+ Monocytes-Freytag
CD20+ B cells-Freytag
CD4+ T cells-Freytag
CD8+ T cells-Freytag
HSPCs-Oetjen
Megakaryocyte progenitors-Oetjen
Monocyte-derived dendritic cells-Oetjen
Monocyte-derived dendritic cells-Sun
NK cells-Freytag
NKT cells-Freytag
Plasma cells-Oetjen
Plasmacytoid dendritic cells-Freytag
Rejected (RE)
Rejection (dist)
root
Predicted labels
1.0
0.8
0.6
0.4
0.2
0.0
Fraction

CHAPTER

FOURTEEN

TREEARCHES: IDENTIFYING NEW CELL TYPES (ADVANCED TUTORIAL)

In this more advanced tutorial, we will show how to use a reference atlas and corresponding cell type hierarchy to detect new cell types in your query dataset. Here, we assume that the query dataset is labeled. If the query dataset is unlabeled, you can just predict the labels of the cells (see previous basic tutorial) and check which cells are rejected. Here, we'll focus more on complete cell types or clusters instead of individual cells.

In this tutorial we'll show two ways to detect new (sub)types: - Option 1: detecting a complete new cell type. We will update the reference cell type hierarchy with the query labels. A new cell type is detected if a cell type from the query is not matched to a cell type in the reference hierarchy. - Option 2: detecting a new subtype. Sometimes a query cell type matches a cell type in the hierarchy, but still a lot of cells are rejected. This could indicate that part of that query cell type is a different subtype that is not detected yet. We will show that you can detect this by comparing the updated hierarchy to the predictions made.

[1]:
```
import scanpy as sc
import scHPL
import numpy as np
import pickle
import time as tm
import copy as cp
import pandas as pd
import matplotlib
import seaborn as sns
```

[2]:
```
sc.settings.set_figure_params(dpi=1000, frameon=False)
sc.set_figure_params(dpi=1000)
sc.set_figure_params(figsize=(7,7))

matplotlib.rcParams['pdf.fonttype'] = 42
```

14.1 Load reference and cell type hierarchy

In this tutorial, we will use the human lung cell atlas (HLCA) as reference. The embeddings for the reference and IPF (query) data we use, can be downloaded here. These embeddings were created with scArches. The trained classifier can be downloaded from Zenodo. There are two classifier here: one trained with the FAISS library and one without. The FAISS library makes the model faster but only works on Linux and with a gpu. More information about installation can be found here.

If you want to train the classifier for the cell type hierarchy yourself, you can find a tutorial here.

[3]:
```
LCA = sc.read('HLCA_emb_and_metadata.h5ad')
file_to_read = open("tree_HCLA_FAISS_withRE.pickle", "rb")
HLCA_tree = pickle.load(file_to_read)
file_to_read.close()
```

14.2 Load query embedding and annotations

We will use a dataset consisting of healthy and IPF (Idiopathic Pulmonary Fibrosis) cells as query dataset. The query embeddings can be downloaded here. The data with annotations can be downloaded here.

[4]:
```
emb_ipf = sc.read('HLCA_extended_models_and_embs/surgery_output_embeddings/Sheppard_2020_
↪emb_LCAv2.h5ad')
```

[5]:
```
data_IPF = sc.read('Sheppard_2020_noSC_finalAnno.h5ad')
```

14.3 Updating the cell type hierarchy

Before we can update the cell-type hierarchy. We have to preprocess the reference and query embeddings a bit. First, we concatenate the cell type labels with the condition labels. This way, we ensure that we can differentiate between the healthy and IPF cells.

[6]:
```
data_IPF.obs['ct-batch'] = np.char.add(np.char.add(np.array(data_IPF.obs['anno_final'],␣
↪dtype=str), '-'), np.array(data_IPF.obs['condition']))
```

Prepare query embeddings

[35]:
```
emb_ipf = emb_ipf[data_IPF.obs_names]
emb_ipf.obs['ct-batch'] = data_IPF.obs['ct-batch']
emb_ipf.obs['batch'] = 'Query'
emb_ipf.obs['batch2'] = data_IPF.obs['condition']
```

```
/tmp/ipykernel_2427513/1399860873.py:2: ImplicitModificationWarning: Trying to modify␣
↪attribute `.obs` of view, initializing view as actual.
  emb_ipf.obs['ct-batch'] = data_IPF.obs['ct-batch']
```

Prepare reference embeddings

[36]:
```
LCA.obs['ct-batch'] = LCA.obs['ann_finest_level']
LCA.obs['batch'] = 'Reference'
LCA.obs['batch2'] = 'Reference'
```

Concatenate the reference and query data

[37]:
```
LCA_IPF = sc.concat([LCA, emb_ipf])
LCA_IPF.obs['ct-batch'] = LCA_IPF.obs['ct-batch'].str.replace('_',' ')
```

Remove cell types that are smaller than 10 cells from the data.

[38]:
```
xx = LCA_IPF.obs.groupby(['ct-batch']).count()
cp_toremove = xx[xx['batch'] < 10].index
idx_tokeep = np.isin(LCA_IPF.obs['ct-batch'], cp_toremove) == False
LCA_IPF = LCA_IPF[idx_tokeep]
LCA_IPF
```

[38]:
```
View of AnnData object with n_obs × n_vars = 646445 × 30
    obs: 'ct-batch', 'batch', 'batch2'
```

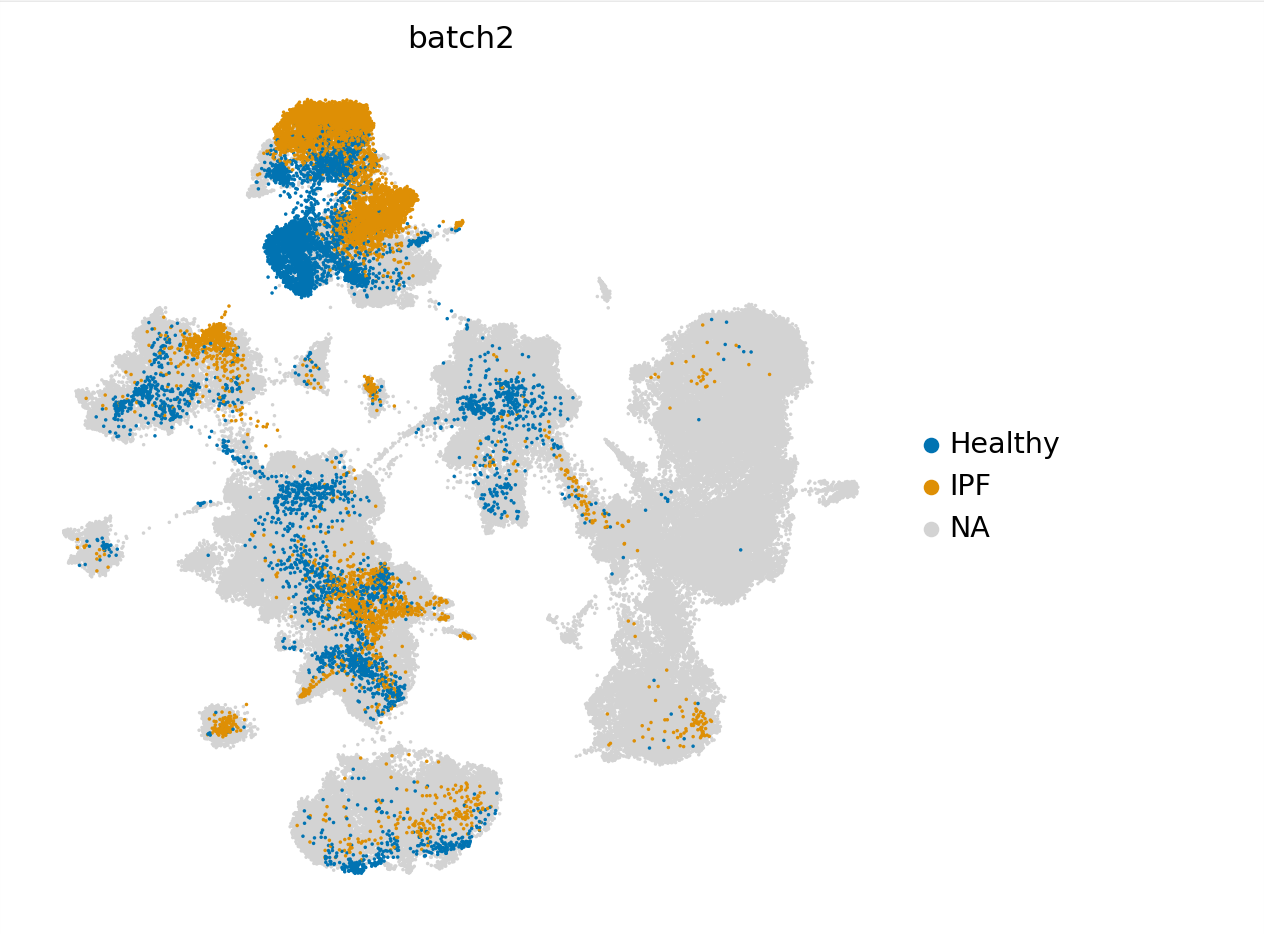
Before updating the hierarchy, we subsample the data (otherwise it takes long to run) and make a UMAP.

[39]:
```
LCA_IPF_subs = sc.pp.subsample(LCA_IPF, fraction=0.25, copy=True)

sc.pp.neighbors(LCA_IPF_subs)
sc.tl.leiden(LCA_IPF_subs)
sc.tl.umap(LCA_IPF_subs)
```

In the UMAP, we can for instance see where the healthy and IPF cells are with respect to the reference

[40]:
```
sc.pl.umap(LCA_IPF_subs,
           color=['batch2'], groups = ['Healthy', 'IPF'],
           frameon=False,
           wspace=0.6, s=10, palette=sns.color_palette('colorblind', as_cmap=True)
           )
```

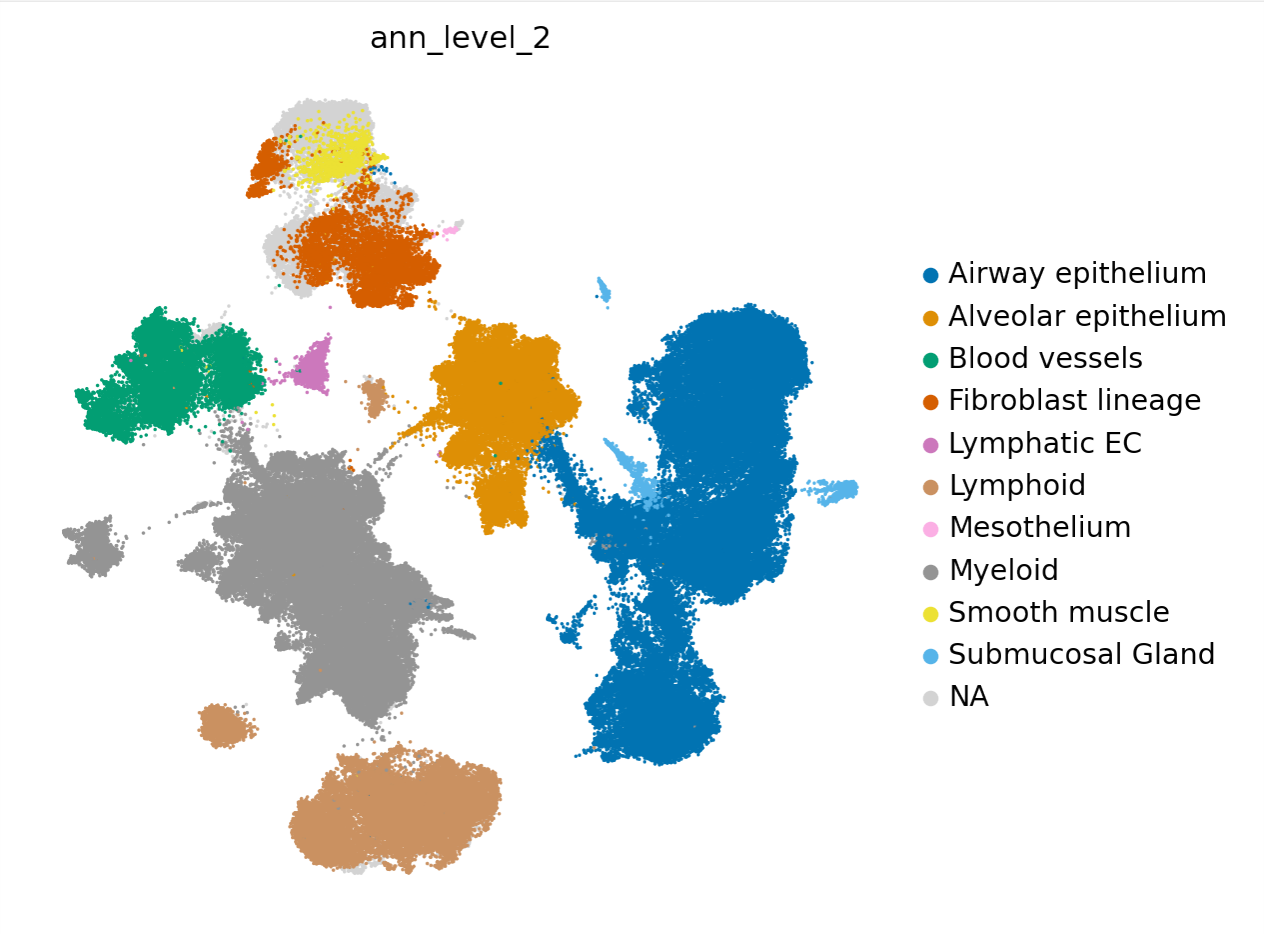
```
/exports/humgen/lmichielsen/miniconda3_v2/envs/scarches2/lib/python3.8/site-packages/
↪scanpy/plotting/_tools/scatterplots.py:1171: FutureWarning: Categorical.replace is␣
↪deprecated and will be removed in a future version. Use Series.replace directly␣
↪instead.
  values = values.replace(values.categories.difference(groups), np.nan)
```



The reference data contains many cell types. Therefore, we visualize the second annotation level here instead of the most detailed level.

[43]:
```
LCA_IPF_subs.obs['ann_level_2'] = LCA.obs.ann_level_2

sc.pl.umap(LCA_IPF_subs, color=['ann_level_2'],
           groups = ['Airway epithelium', 'Alveolar epithelium',
                     'Blood vessels', 'Fibroblast lineage', 'Lymphatic EC',
                     'Lymphoid', 'Mesothelium', 'Myeloid', 'Smooth muscle',
                     'Submucosal Gland'],
           frameon=False,
           wspace=0.6, s=10, palette=sns.color_palette('colorblind', as_cmap=True)
           )
```

```
/exports/humgen/lmichielsen/miniconda3_v2/envs/scarches2/lib/python3.8/site-packages/
↪scanpy/plotting/_tools/scatterplots.py:1171: FutureWarning: Categorical.replace is␣
↪deprecated and will be removed in a future version. Use Series.replace directly␣
↪instead.
  values = values.replace(values.categories.difference(groups), np.nan)
```



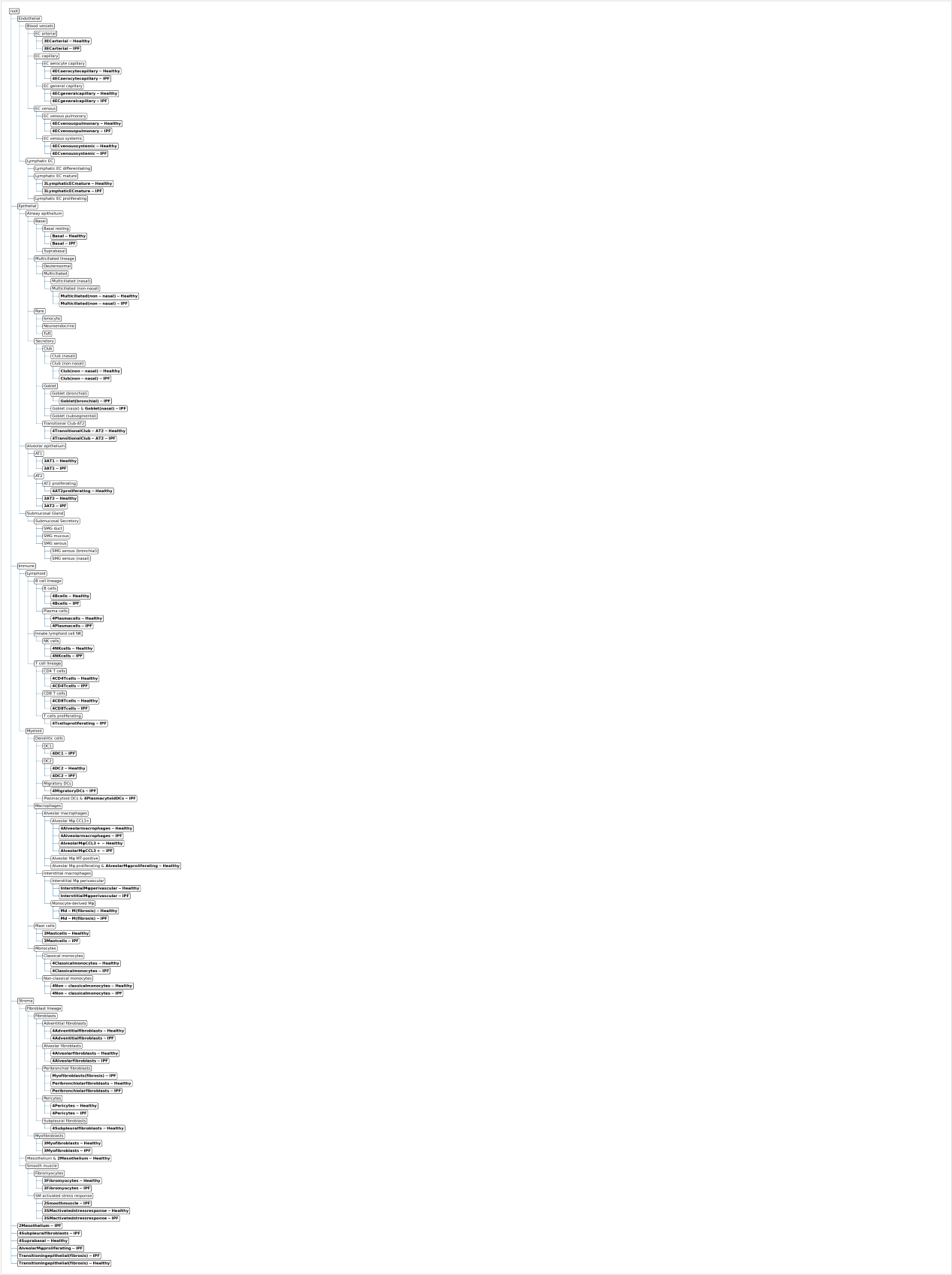
Update the cell type hierarchy with the query cell types. In the updated hierarchy, we see that for instance the 'Transitioning epithelial cells' are added as a new cell type to the tree. This cell type was indeed not in the reference and thus correctly discovered as new.

[11]:
```
## Since the data contains >600.000 cells, this step can take a while to run (~1 hour)
HLCA_tree = scHPL.learn.learn_tree(LCA_IPF,
                                   batch_key = 'batch',
                                   batch_order = ['Query'],
                                   cell_type_key = 'ct-batch',
                                   tree = HLCA_tree,
                                   retrain = False, useRE=True,
                                   batch_added = ['Reference']
                                  )
```

```
Starting tree:
```

root
Endothelial
Blood vessels
EC arterial
EC capillary
EC aerocyte capillary
EC general capillary
EC venous
EC venous pulmonary
EC venous systemic
Lymphatic EC
Lymphatic EC differentiating
Lymphatic EC mature
Lymphatic EC proliferating
Epithelial
Airway epithelium
Basal
Basal resting
Suprabasal
Multiciliated lineage
Deuterosomal
Multiciliated
Multiciliated (nasal)
Multiciliated (non-nasal)
Rare
Ionocyte
Neuroendocrine
Tuft
Secretory
Club
Club (nasal)
Club (non-nasal)
Goblet
Goblet (bronchial)
Goblet (nasal)
Goblet (subsegmental)
Transitional Club-AT2
Alveolar epithelium
AT1
AT2
AT2 proliferating
Submucosal Gland
Submucosal Secretory
SMG duct
SMG mucous
SMG serous
SMG serous (bronchial)
SMG serous (nasal)
Immune
Lymphoid
B cell lineage
B cells
Plasma cells
Innate lymphoid cell NK
NK cells
T cell lineage
CD4 T cells
CD8 T cells
T cells proliferating
Myeloid
Dendritic cells
DC1
DC2
Migratory DCs
Plasmacytoid DCs
Macrophages
Alveolar macrophages
Alveolar Mφ CCL3+
Alveolar Mφ MT-positive
Alveolar Mφ proliferating
Interstitial macrophages
Interstitial Mφ perivascular
Monocyte-derived Mφ
Mast cells
Monocytes
Classical monocytes
Non-classical monocytes
Stroma
Fibroblast lineage
Fibroblasts
Adventitial fibroblasts
Alveolar fibroblasts
Peribronchial fibroblasts
Pericytes
Subpleural fibroblasts
Myofibroblasts
Mesothelium
Smooth muscle
Fibromyocytes
SM activated stress response

```
Adding dataset Query to the tree
These populations are missing from the tree:
['2 Smooth muscle-Healthy']

Updated tree:
```


root
Endothelial
Blood vessels
EC arterial
3ECarterial – Healthy
3ECarterial – IPF
EC capillary
EC aerocyte capillary
4ECaerocytecapillary – Healthy
4ECaerocytecapillary – IPF
EC general capillary
4ECgeneralcapillary – Healthy
4ECgeneralcapillary – IPF
EC venous
EC venous pulmonary
4ECvenouspulmonary – Healthy
4ECvenouspulmonary – IPF
EC venous systemic
4ECvenousystemic – Healthy
4ECvenousystemic – IPF
Lymphatic EC
Lymphatic EC differentiating
Lymphatic EC mature
3LymphaticECmature – Healthy
3LymphaticECmature – IPF
Lymphatic EC proliferating
Epithelial
Airway epithelium
Basal
Basal resting
Basal – Healthy
Basal – IPF
Suprabasal
Multiciliated lineage
Deuterosomal
Multiciliated
Multiciliated (nasal)
Multiciliated (non-nasal)
Multiciliated(non – nasal) – Healthy
Multiciliated(non – nasal) – IPF
Rare
Ionocyte
Neuroendocrine
Tuft
Secretory
Club
Club (nasal)
Club (non-nasal)
Club(non – nasal) – Healthy
Club(non – nasal) – IPF
Goblet
Goblet (bronchial)
Goblet(bronchial) – IPF
Goblet (nasal) & Goblet(nasal) – IPF
Goblet (subsegmental)
Transitional Club-AT2
4TransitionalClub – AT2 – Healthy
4TransitionalClub – AT2 – IPF
Alveolar epithelium
AT1
3AT1 – Healthy
3AT1 – IPF
AT2
AT2 proliferating
4AT2proliferating – Healthy
3AT2 – Healthy
3AT2 – IPF
Submucosal Gland
Submucosal Secretory
SMG duct
SMG mucous
SMG serous
SMG serous (bronchial)
SMG serous (nasal)
Immune
Lymphoid
B cell lineage
B cells
4Bcells – Healthy
4Bcells – IPF
Plasma cells
4Plasmacells – Healthy
4Plasmacells – IPF
Innate lymphoid cell NK
NK cells
4NKcells – Healthy
4NKcells – IPF
T cell lineage
CD4 T cells
4CD4Tcells – Healthy
4CD4Tcells – IPF
CD8 T cells
4CD8Tcells – Healthy
4CD8Tcells – IPF
T cells proliferating
4Tcellsproliferating – IPF
Myeloid
Dendritic cells
DC1
4DC1 – IPF
DC2
4DC2 – Healthy
4DC2 – IPF
Migratory DCs
4MigratoryDCs – IPF
Plasmacytoid DCs & 4PlasmacytoidDCs – IPF
Macrophages
Alveolar macrophages
Alveolar Mφ CCL3+
4Alveolarmacrophages – Healthy
4Alveolarmacrophages – IPF
AlveolarMφCCL3 + – Healthy
AlveolarMφCCL3 + – IPF
Alveolar Mφ MT-positive
Alveolar Mφ proliferating & AlveolarMφproliferating – Healthy
Interstitial macrophages
Interstitial Mφ perivascular
InterstitialMφperivascular – Healthy
InterstitialMφperivascular – IPF
Monocyte-derived Mφ
Mφ – M(fibrosis) – Healthy
Mφ – M(fibrosis) – IPF
Mast cells
3Mastcells – Healthy
3Mastcells – IPF
Monocytes
Classical monocytes
4Classicalmonocytes – Healthy
4Classicalmonocytes – IPF
Non-classical monocytes
4Non – classicalmonocytes – Healthy
4Non – classicalmonocytes – IPF
Stroma
Fibroblast lineage
Fibroblasts
Adventitial fibroblasts
4Adventitialfibroblasts – Healthy
4Adventitialfibroblasts – IPF
Alveolar fibroblasts
4Alveolarfibroblasts – Healthy
4Alveolarfibroblasts – IPF
Peribronchial fibroblasts
Myofibroblasts(fibrosis) – IPF
Peribronchialfibroblasts – Healthy
Peribronchialfibroblasts – IPF
Pericytes
4Pericytes – Healthy
4Pericytes – IPF
Subpleural fibroblasts
4Subpleuralfibroblasts – Healthy
Myofibroblasts
3Myofibroblasts – Healthy
3Myofibroblasts – IPF
Mesothelium & 2Mesothelium – Healthy
Smooth muscle
Fibromyocytes
3Fibromyocytes – Healthy
3Fibromyocytes – IPF
SM activated stress response
2Smoothmuscle – IPF
3SMactivatedstressresponse – Healthy
3SMactivatedstressresponse – IPF
2Mesothelium – IPF
4Subpleuralfibroblasts – IPF
4Suprabasal – Healthy
AlveolarMφproliferating – IPF
Transitioningepithelial(fibrosis) – IPF
Transitioningepithelial(fibrosis) – Healthy

14.4 Predict cell type labels

In this part, we will show how to detect a subtype. First, we use the original tree to predict the labels of the query data.

[12]:
```
file_to_read = open("tree_HCLA_FAISS_withRE.pickle", "rb")
HLCA_ref = pickle.load(file_to_read)
file_to_read.close()

y_pred = scHPL.predict.predict_labels(emb_ipf.X,
                tree = HLCA_ref,
                threshold = 0.5)

emb_ipf.obs['scHPL_pred'] = y_pred
```

Here, we will zoom in on the macrophages to see how they are predicted. Since, we're also interested in marker genes, we need the count data. The count data for the reference can be downloaded here.

[13]:
```
data_LCA = sc.read_h5ad('local.h5ad')
data_LCA.var_names = np.asarray(data_LCA.var['feature_name'], dtype=str)
```

We normalize the IPF data

[14]:
```
sc.pp.normalize_total(data_IPF)
sc.pp.log1p(data_IPF)
```

Concatenate the data

[15]:
```
data_all = sc.concat([data_LCA, data_IPF])
data_all
```

```
/exports/humgen/lmichielsen/miniconda3_v2/envs/scarches2/lib/python3.8/site-packages/
↪anndata/_core/merge.py:942: UserWarning: Only some AnnData objects have `.raw`␣
↪attribute, not concatenating `.raw` attributes.
  warn(
```

[15]:
```
AnnData object with n_obs × n_vars = 646487 × 1897
    obs: 'sample', 'study', 'subject_ID', 'smoking_status', 'BMI', 'condition', 'sample_
↪type', 'dataset', 'age', 'original_ann_level_1', 'original_ann_level_2', 'original_ann_
↪level_3', 'original_ann_level_4', 'original_ann_level_5', 'original_ann_nonharmonized',
↪ 'sex', 'ethnicity'
    obsm: 'X_scanvi_emb'
```

Prepare the metadata

[23]:
```
data_all.obs['ann_level_3'] = data_LCA.obs.ann_level_3
data_all.obs['ann_finest_level'] = data_LCA.obs.ann_finest_level
data_all.obs['anno_final'] = data_IPF.obs.anno_final
data_all.obs['scHPL_pred'] = data_IPF.obs.scHPL_pred
data_all.obs['condition'] = data_IPF.obs.condition
```

Select the macrophages

[24]:
```
idx_macro = ((data_all.obs['ann_level_3'] == 'Macrophages') |
             np.isin(data_all.obs.anno_final, ['4_Alveolar macrophages', 'Alveolar M␣
```

(continues on next page)

(continued from previous page)

```
↪CCL3+',
                                                 'Alveolar M proliferating',
                                                  'Interstitial M perivascular', 'Md-M␣
↪(fibrosis)']))
data_macro = data_all[idx_macro]
data_macro
```

[24]:
```
View of AnnData object with n_obs × n_vars = 117425 × 1897
    obs: 'sample', 'study', 'subject_ID', 'smoking_status', 'BMI', 'condition', 'sample_
↪type', 'dataset', 'age', 'original_ann_level_1', 'original_ann_level_2', 'original_ann_
↪level_3', 'original_ann_level_4', 'original_ann_level_5', 'original_ann_nonharmonized',
↪ 'sex', 'ethnicity', 'ann_level_3', 'anno_final', 'scHPL_pred', 'ann_finest_level'
    obsm: 'X_scanvi_emb'
```

treeArches contains three types of rejection options (based on the distance, posterior probability and the reconstruction error). Here, we will rename these different types all to 'Rejected'.

[25]:
```
idx_rej = ((data_macro.obs['scHPL_pred'] == 'Rejection (dist)') | (data_macro.obs['scHPL_
↪pred'] == 'Rejected (RE)'))
data_macro.obs['scHPL_pred'] = data_macro.obs['scHPL_pred'].cat.add_categories('Rejected
↪')
data_macro.obs['scHPL_pred'].values[idx_rej] = 'Rejected'
```

```
/tmp/ipykernel_2427513/617128616.py:2: ImplicitModificationWarning: Trying to modify␣
↪attribute `.obs` of view, initializing view as actual.
  data_macro.obs['scHPL_pred'] = data_macro.obs['scHPL_pred'].cat.add_categories(
↪'Rejected')
```

Here, we visualize the predictions for the IPF and healthy data separately using a Sankey diagram. The python function can be found here. An alternative would be to visualize the predictions using `scHPL.evaluate.heatmap()` (see the treeArches basic tutorial).

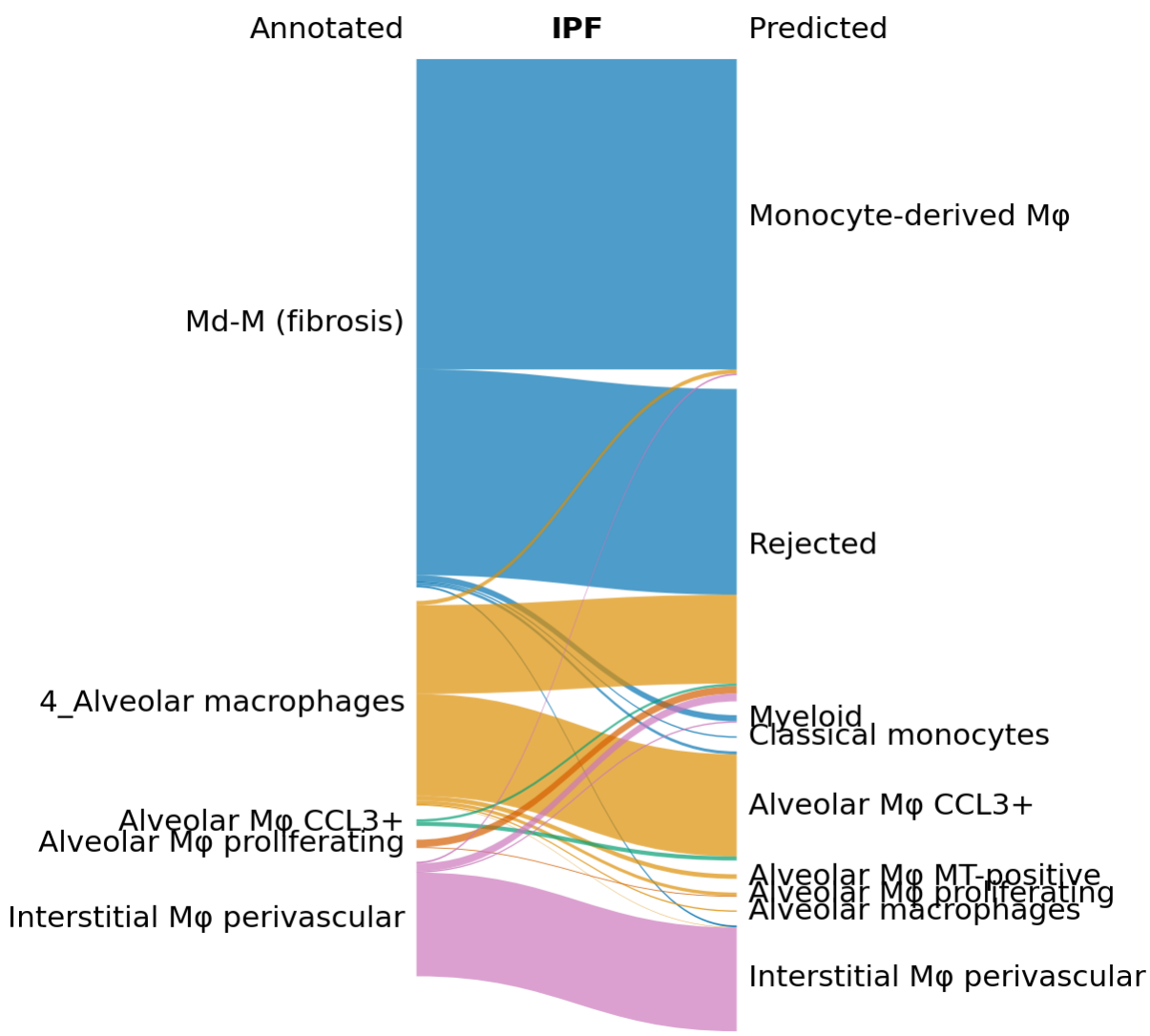
In the Sankey plot, we notice that many of the Md-M fibrosis IPF cells are rejected, but that this is not the case for the healthy cells.

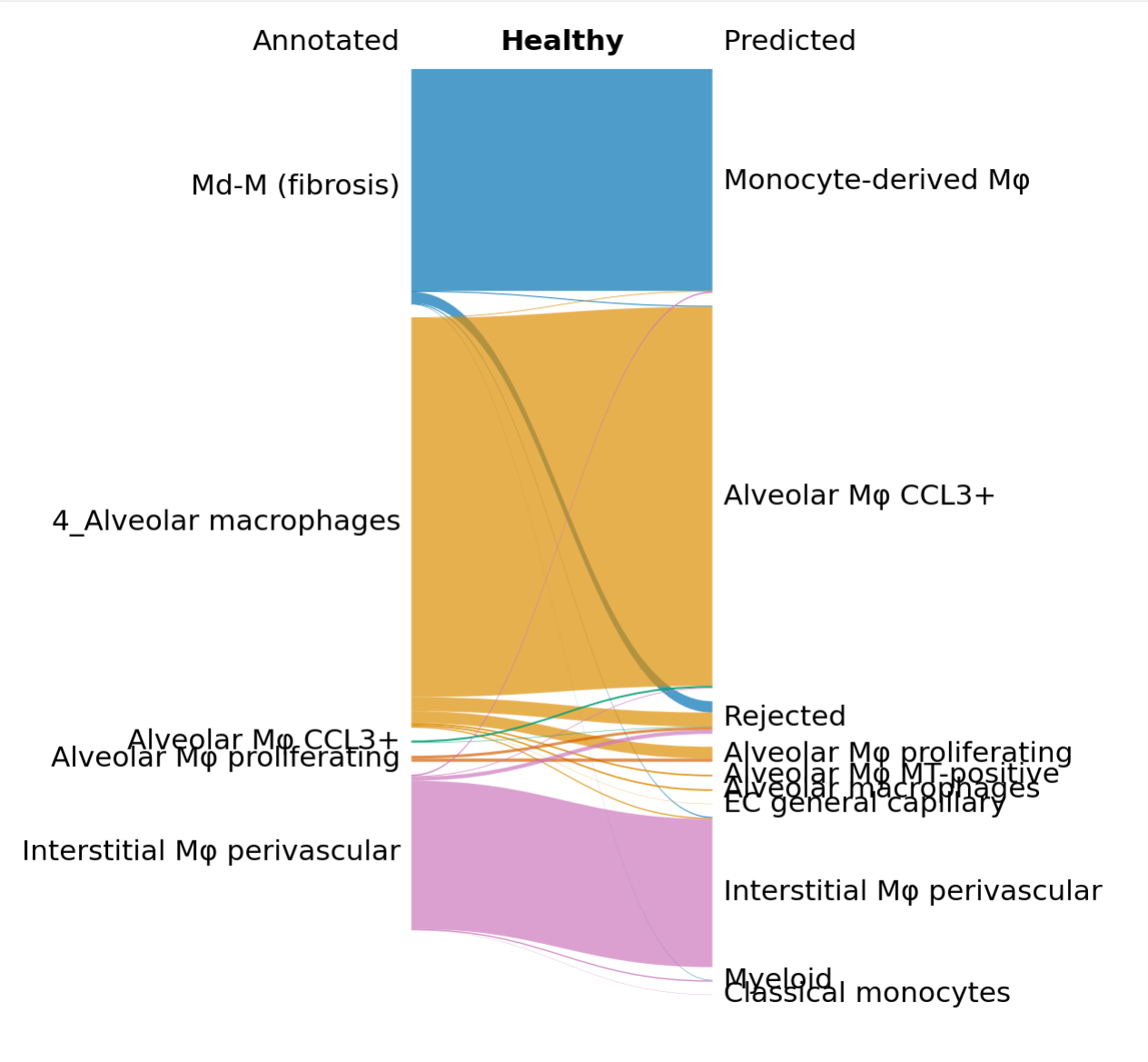
[26]:
```
import sankey

idx1 = (data_macro.obs.study == 'Sheppard_2020') & (data_macro.obs.condition == 'IPF')
idx2 = (data_macro.obs.study == 'Sheppard_2020') & (data_macro.obs.condition == 'Healthy
↪')

x = sankey.sankey( data_macro.obs['anno_final'][idx1],
                   data_macro.obs['scHPL_pred'][idx1], save=True,
                   name_file='sankey_IPF', title="IPF", title_left="Annotated",
                   title_right="Predicted", alpha=0.7,
                   left_order=['Md-M (fibrosis)',
                               '4_Alveolar macrophages',
                                'Alveolar M CCL3+',
                                'Alveolar M MT-positive',
                                'Alveolar M proliferating',
                                'Interstitial M perivascular'
                               ], fontsize='medium')

x = sankey.sankey( data_macro.obs['anno_final'][idx2],
```

(continues on next page)

(continued from previous page)

```
                data_macro.obs['scHPL_pred'][idx2], save=True,
                name_file='sankey_NML', title="Healthy", title_left="Annotated",
                title_right="Predicted", alpha=0.7,
                left_order=['Md-M (fibrosis)',
                           '4_Alveolar macrophages',
                            'Alveolar M CCL3+',
                            'Alveolar M MT-positive',
                            'Alveolar M proliferating',
                            'Interstitial M perivascular'
                           ], fontsize='medium')
```





Next, we show how to verify the results by doing some downstream analysis. We split the Md-M fibrosis IPF cells in two groups: the rejected and not rejected cells, and we find differentially expression genes between the two.

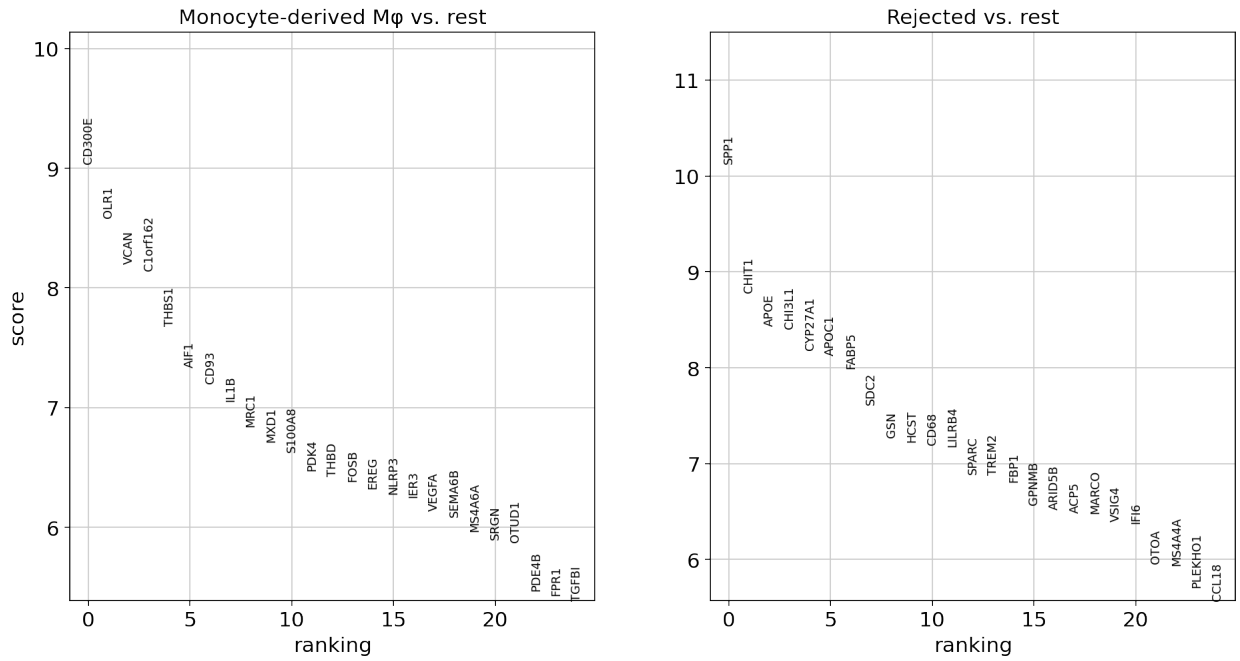
[27]:
```
### Do DE
# Group 1: anno_final = Md-M (fibrosis), scHPL_pred = Monocyte-derived macro,␣
↪condition=IPF
# Group 2: anno_final = Md-M (fibrosis), scHPL_pred = Rejected, condition=IPF
MdM = (data_macro.obs.condition == 'IPF') & (data_macro.obs.anno_final == 'Md-M␣
↪(fibrosis)') & ((data_macro.obs.scHPL_pred == 'Rejected') | (data_macro.obs.scHPL_pred␣
↪== 'Monocyte-derived M'))
data_MdM = data_macro[MdM]

sc.pp.normalize_total(data_MdM)
sc.pp.log1p(data_MdM)

sc.tl.rank_genes_groups(data_MdM, 'scHPL_pred', method='t-test')
```

(continues on next page)

(continued from previous page)

```
sc.pl.rank_genes_groups(data_MdM, n_genes=25, sharey=False)
```

```
/exports/humgen/lmichielsen/miniconda3_v2/envs/scarches2/lib/python3.8/site-packages/
↪scanpy/preprocessing/_normalization.py:170: UserWarning: Received a view of an AnnData.
↪ Making a copy.
  view_to_actual(adata)
```



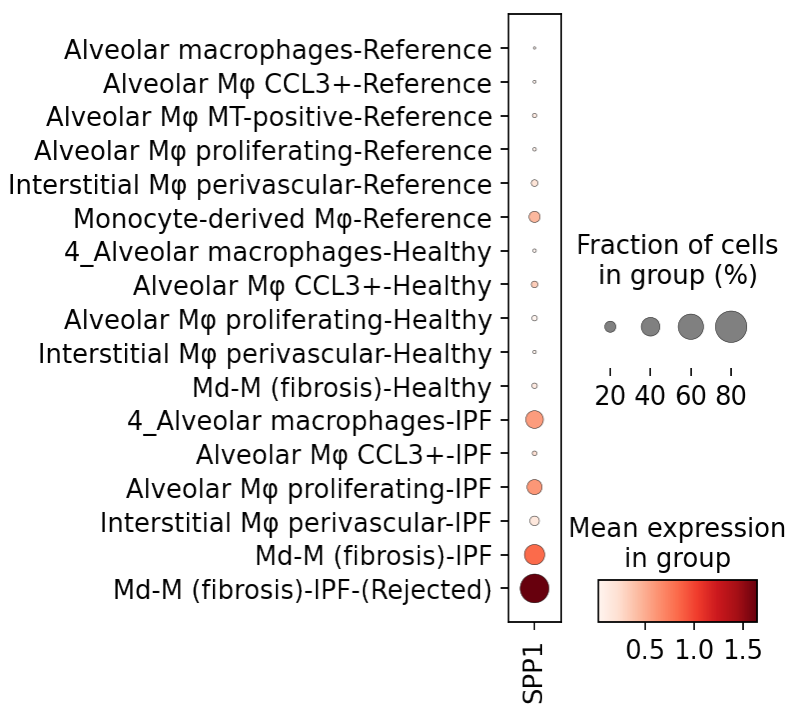
[28]:
```
data_macro.obs['ann_toplot'] = np.char.add(np.array(data_macro.obs.ann_finest_level,␣
↪dtype=str),'-Reference')
data_macro.obs.ann_toplot[data_macro.obs.study == 'Sheppard_2020'] = np.char.add(np.char.
↪add(np.array(data_macro.obs.anno_final, dtype=str), '-'), np.array(data_macro.obs.
↪condition, dtype=str))
idx = (data_macro.obs.ann_toplot == 'Md-M (fibrosis)-IPF') & (data_macro.obs.scHPL_pred␣
↪== 'Rejected')
data_macro.obs['ann_toplot'][idx] = 'Md-M (fibrosis)-IPF-(Rejected)'
```

```
/tmp/ipykernel_2427513/3849917173.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  data_macro.obs.ann_toplot[data_macro.obs.study == 'Sheppard_2020'] = np.char.add(np.
↪char.add(np.array(data_macro.obs.anno_final, dtype=str), '-'), np.array(data_macro.obs.
↪condition, dtype=str))
/tmp/ipykernel_2427513/3849917173.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  data_macro.obs['ann_toplot'][idx] = 'Md-M (fibrosis)-IPF-(Rejected)'
```

When we visualize this marker gene, we see that it is only expressed in the Md-M IPF rejected cells. According to literature, SPP1 is known to be a hallmark for IPF pathogenesis 1, 2.

[29]:
```
sc.pl.dotplot(data_macro, ['SPP1'], groupby='ann_toplot',
              categories_order=[
                  'Alveolar macrophages-Reference',
                  'Alveolar M CCL3+-Reference',
                  'Alveolar M MT-positive-Reference',
                  'Alveolar M proliferating-Reference',
                  'Interstitial M perivascular-Reference',
                  'Monocyte-derived M-Reference',

                  '4_Alveolar macrophages-Healthy',
                  'Alveolar M CCL3+-Healthy',
                  'Alveolar M proliferating-Healthy',
                  'Interstitial M perivascular-Healthy',
                  'Md-M (fibrosis)-Healthy',

                  '4_Alveolar macrophages-IPF',
                  'Alveolar M CCL3+-IPF',
                  'Alveolar M proliferating-IPF',
                  'Interstitial M perivascular-IPF',
                  'Md-M (fibrosis)-IPF',
                  'Md-M (fibrosis)-IPF-(Rejected)'
              ], figsize=(2,5), save='_IPF_SPP1.pdf')
```

```
WARNING: saving figure to file figures/dotplot__IPF_SPP1.pdf
```



[ ]:

CHAPTER

FIFTEEN

SPATIAL RECONSTRUCTION OF THE MOUSE EMBRYO WITH SAGENET

In this notebook present spatial reconstruction of the mouse embryo with SageNet. We use the seqFISH dataset collected by Lohoff et al. (2022) as the spatial reference. This Spatial Mouse Atlas dataset contains barcoded gene expression measurements for 351 genes in three distinct mouse embryo sagittal sections. We also use combination of a subset of this dataset and the dissociated scRNAseq mouse gastrulation atlas by Pijuan-Sala et al. (2019) as the query dataset. For both spatial and scRNAseq datasets, we focus on embryonic day (E)8.5.

We specifically show how to aggeregate multiple spatial references and build an ensemble SageNet model.

15.1 Setup

15.1.1 Install scArches

SageNet is now implemented as a model in the scarches code base. In order to get the latest developments, we recommend installing the package via github.

[ ]:
```
!git clone https://github.com/theislab/scarches
%cd scarches
!pip install .
```

15.1.2 Install pytorch geometric

SageNet employs pytroch geometric (PyG) to implement graph neural networks (GGNs). We install pytorch geometric and its requirments as instructed here.

PyG's requirements to be installed depend on the torch and cuda versions:

[ ]:
```
import torch; print(torch.__version__)
import torch; print(torch.version.cuda)
```

```
1.12.1+cu113
11.3
```

[3]:
```
!pip install -q torch-scatter -f https://data.pyg.org/whl/torch-1.12.1+cu113.html
!pip install -q torch-sparse -f https://data.pyg.org/whl/torch-1.12.1+cu113.html
!pip install -q git+https://github.com/pyg-team/pytorch_geometric.git
```

```
     || 7.9 MB 1.1 MB/s
     || 3.5 MB 1.2 MB/s
  Building wheel for torch-geometric (setup.py) ... done
```

15.1.3 Other auxilary packages

We use squidpy for preprocessing spatial data and captum for interpreting our GGNs.

[ ]:
```
!pip install squidpy
!pip install captum
!pip install patchworklib
```

15.1.4 Import libraries

[ ]:
```
import scarches as sca
import scanpy as sc # for plotting
import anndata as ad # for handling the spatial and single-cell datasets
import random # for setting a random seed
import numpy as np
import copy
import squidpy as sq
import pandas as pd
from scarches.models.sagenet.utils import glasso
from matplotlib import *
import patchworklib as pw
import functools
```

15.1.5 Cell type colours

[3]:
```
celltype_colours = {
  "Epiblast" : "#635547",
  "Primitive Streak" : "#DABE99",
  "Caudal epiblast" : "#9e6762",
  "PGC" : "#FACB12",
  "Anterior Primitive Streak" : "#c19f70",
  "Notochord" : "#0F4A9C",
  "Def. endoderm" : "#F397C0",
  "Definitive endoderm" : "#F397C0",
  "Gut" : "#EF5A9D",
  "Gut tube" : "#EF5A9D",
  "Nascent mesoderm" : "#C594BF",
  "Mixed mesoderm" : "#DFCDE4",
  "Intermediate mesoderm" : "#139992",
  "Caudal Mesoderm" : "#3F84AA",
  "Paraxial mesoderm" : "#8DB5CE",
  "Somitic mesoderm" : "#005579",
  "Pharyngeal mesoderm" : "#C9EBFB",
  "Splanchnic mesoderm" : "#C9EBFB",
```

(continues on next page)

(continued from previous page)

```
  "Cardiomyocytes" : "#B51D8D",
  "Allantois" : "#532C8A",
  "ExE mesoderm" : "#8870ad",
  "Lateral plate mesoderm" : "#8870ad",
  "Mesenchyme" : "#cc7818",
  "Mixed mesenchymal mesoderm" : "#cc7818",
  "Haematoendothelial progenitors" : "#FBBE92",
  "Endothelium" : "#ff891c",
  "Blood progenitors 1" : "#f9decf",
  "Blood progenitors 2" : "#c9a997",
  "Erythroid1" : "#C72228",
  "Erythroid2" : "#f79083",
  "Erythroid3" : "#EF4E22",
  "Erythroid" : "#f79083",
  "Blood progenitors" : "#f9decf",
  "NMP" : "#8EC792",
  "Rostral neurectoderm" : "#65A83E",
  "Caudal neurectoderm" : "#354E23",
  "Neural crest" : "#C3C388",
  "Forebrain/Midbrain/Hindbrain" : "#647a4f",
  "Spinal cord" : "#CDE088",
  "Surface ectoderm" : "#f7f79e",
  "Visceral endoderm" : "#F6BFCB",
  "ExE endoderm" : "#7F6874",
  "ExE ectoderm" : "#989898",
  "Parietal endoderm" : "#1A1A1A",
  "Unknown" : "#FFFFFF",
  "Low quality" : "#e6e6e6",
  # somitic and paraxial types
  # colour from T chimera paper Guibentif et al Developmental Cell 2021
  "Cranial mesoderm" : "#77441B",
  "Anterior somitic tissues" : "#F90026",
  "Sclerotome" : "#A10037",
  "Dermomyotome" : "#DA5921",
  "Posterior somitic tissues" : "#E1C239",
  "Presomitic mesoderm" : "#9DD84A"
}
```

Setting the torch device

We set the torch device to cuda if it's available.

[4]:
```
import torch
if torch.cuda.is_available():
  dev = "cuda:0"
else:
  dev = "cpu"
device = torch.device(dev)
print(device)
```

```
cuda:0
```

15.2 Training and Mapping

15.3 The spatial and scRNAseq mouse gastrulation datasets

Lohoff et al. (2022) carried out a SeqFISH experiment on sagittal sections from three mouse embryos corresponding to embryonic day (E)8.5–8.75 to quantify spatial gene expression at single cell resolution of a pre-selected set of 387 genes. For each embryo section, they captured two 2D planes, 12um apart, yielding a total of 6 spatially-resolved sections. The authors performed cell segmentation, quantified gene expression log-counts, and assigned cell type identities to each cell using a large-scale single cell study of mouse gastrulation Pijuan-Sala et al. (2019) as a reference. These two datasets are known as spatial and single-cell mouse gastrulation atlases (MGAs).

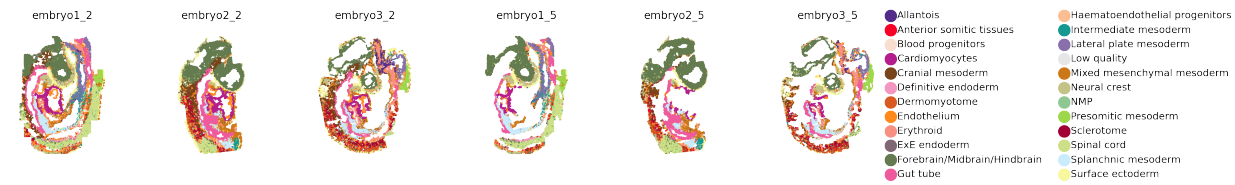
This dataset is ideal for evaluating SageNet's performance, since we have access to ground truth spatial coordinates for individual cells over multiple biological replicates, and the tissue structure observed across mouse embryos are varied and complex. We downloaded the gene expression matrix and cell type and spatial location metadata from https://content.cruk.cam.ac.uk/jmlab/SpatialMouseAtlas2020/.

Prior to analysis, we removed cells that were annotated as "Low quality" by the authors. And we subseted the scRNAseq so that all datasets have the same set of genes. No further preprocessing was performed on the datasets. The filtered seqFISH datsets could be simply downloaded and loaded in the environment by calling `scarches.dataset.MGA_data.seqFISH{embryo}_{layer}()`. The single-cell RNAseq dataset is loaded using `scarches.dataset.MGA_data.scRNAseq()`

[5]:
```
adata_seqFISH1_1 = sca.dataset.MGA_data.seqFISH1_1()
adata_seqFISH2_1 = sca.dataset.MGA_data.seqFISH2_1()
adata_seqFISH3_1 = sca.dataset.MGA_data.seqFISH3_1()
adata_seqFISH1_2 = sca.dataset.MGA_data.seqFISH1_2()
adata_seqFISH2_2 = sca.dataset.MGA_data.seqFISH2_2()
adata_seqFISH3_2 = sca.dataset.MGA_data.seqFISH3_2()
adata_scRNAseq   = sca.dataset.scRNAseq()
```

[6]:
```
sc.set_figure_params(figsize=(2, 2), fontsize=6)
import functools
adata_seqFISH_list = [adata_seqFISH1_1, adata_seqFISH2_1, adata_seqFISH3_1, adata_
↪seqFISH1_2, adata_seqFISH2_2, adata_seqFISH3_2]
axs = []
for i in range(len(adata_seqFISH_list)):
  axs.append(pw.Brick())
  adata = adata_seqFISH_list[i]
  adata.obsm['spatial'] = np.array(adata.obs[['x','y']])
  sq.gr.spatial_neighbors(adata, coord_type="generic")
  # with rc_context({'figure.figsize': (2, 2)}):
  if(i != len(adata_seqFISH_list)-1):
    sc.pl.spatial(adata, color='cell_type', palette=celltype_colours, frameon=False,␣
↪spot_size=.1, title=pd.unique(adata.obs['embryo']), ax=axs[i], legend_loc=None)
  else:
    sc.pl.spatial(adata, color='cell_type', palette=celltype_colours, frameon=False,␣
↪spot_size=.1, title=pd.unique(adata.obs['embryo']), ax=axs[i])
plots = functools.reduce(lambda a, b: a+b, axs)
plots.savefig()
#
```

[6]:



15.3.1 Training with one reference dataset

As the basic functionality of SageNet, we start with training the model on only one spatial reference. We take embryo 1 layer 1 (`seqFISH1_1`) as the spatial reference.

Building the gene interaction network

SageNet model uses a gene interaction network (across all cells in the reference dataset) to train the GNN (see the preprint). We use a utility fuction impelemented in the package for the graphical LASSO (GLASSO) algorithm to estimate the gene interaction network. The funciton takes a grid of regulatrization parameters and does a cross validation to find the optimal parameter (see this).

[7]:
```
glasso(adata_seqFISH1_1, [0.25, 0.5])
adata_seqFISH1_1
```

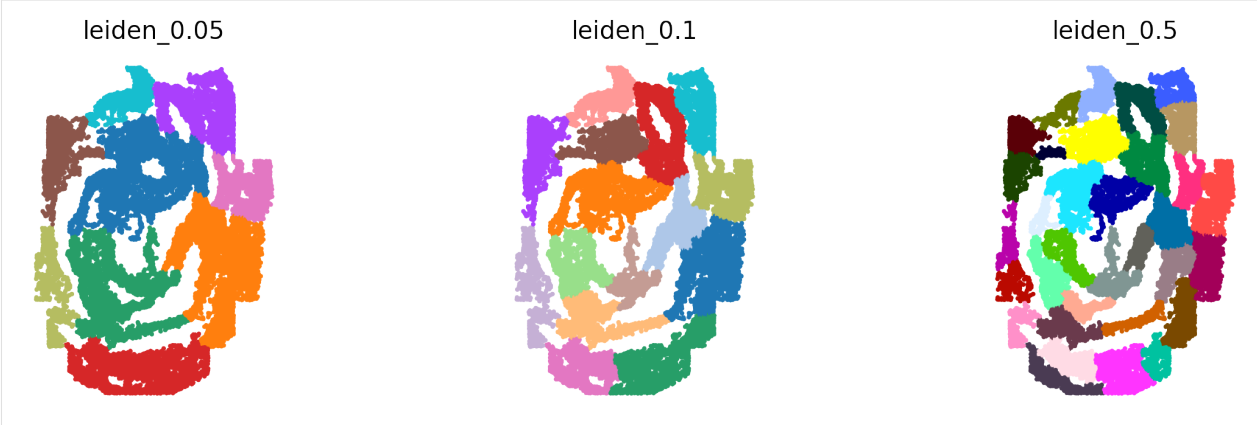
[7]:
```
AnnData object with n_obs × n_vars = 10045 × 350
    obs: 'cell_id', 'embryo', 'x', 'y', 'UMAP1', 'UMAP2', 'cell_type', 'res_0.05',
↪'class_'
    uns: 'X_name', 'celltype_colours', 'spatial_neighbors', 'cell_type_colors'
    obsm: 'spatial'
    varm: 'adj'
    obsp: 'spatial_connectivities', 'spatial_distances'
```

The adjacency matrix of the built graph has added under the name `adj` to the `varm` section of the annData object.

Spatial partitioning of the reference

The spatial reference should be partinitioned into distinct spatial neighborhood. The GGNs learn to map the dissociated query cells to those partitions. We use the `leiden <https://www.nature.com/articles/s41598-019-41695-z>`__ algorithm on the spatial graph created by `squidpy <https://squidpy.readthedocs.io/en/stable/>`__. We run the partitioning at 3 different resolutions. The idea is to capture different granularities in space.

[8]:
```
sc.set_figure_params(figsize=(4, 4), fontsize=15)
sc.tl.leiden(adata_seqFISH1_1, resolution=.05, random_state=0, key_added='leiden_0.05',␣
↪adjacency=adata_seqFISH1_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH1_1, resolution=.1, random_state=0, key_added='leiden_0.1',␣
↪adjacency=adata_seqFISH1_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH1_1, resolution=.5, random_state=0, key_added='leiden_0.5',␣
↪adjacency=adata_seqFISH1_1.obsp["spatial_connectivities"])
# with rc_context({'figure.figsize': (2, 2)}):
sc.pl.spatial(adata_seqFISH1_1, color=['leiden_0.05', 'leiden_0.1', 'leiden_0.5'],␣
↪frameon=False, ncols=3, spot_size=.1, title=['leiden_0.05', 'leiden_0.1', 'leiden_0.5
↪'],  legend_loc=None)
adata_seqFISH1_1
```



[8]:
```
AnnData object with n_obs × n_vars = 10045 × 350
    obs: 'cell_id', 'embryo', 'x', 'y', 'UMAP1', 'UMAP2', 'cell_type', 'res_0.05',
↪'class_', 'leiden_0.05', 'leiden_0.1', 'leiden_0.5'
    uns: 'X_name', 'celltype_colours', 'spatial_neighbors', 'cell_type_colors', 'leiden',
↪ 'leiden_0.05_colors', 'leiden_0.1_colors', 'leiden_0.5_colors'
    obsm: 'spatial'
    varm: 'adj'
    obsp: 'spatial_connectivities', 'spatial_distances'
```

The partitionings have been added to the `obs` section.

Training the model

We now define the SageNet model object:

[9]:
```
sg_obj = sca.models.sagenet(device=device)
```

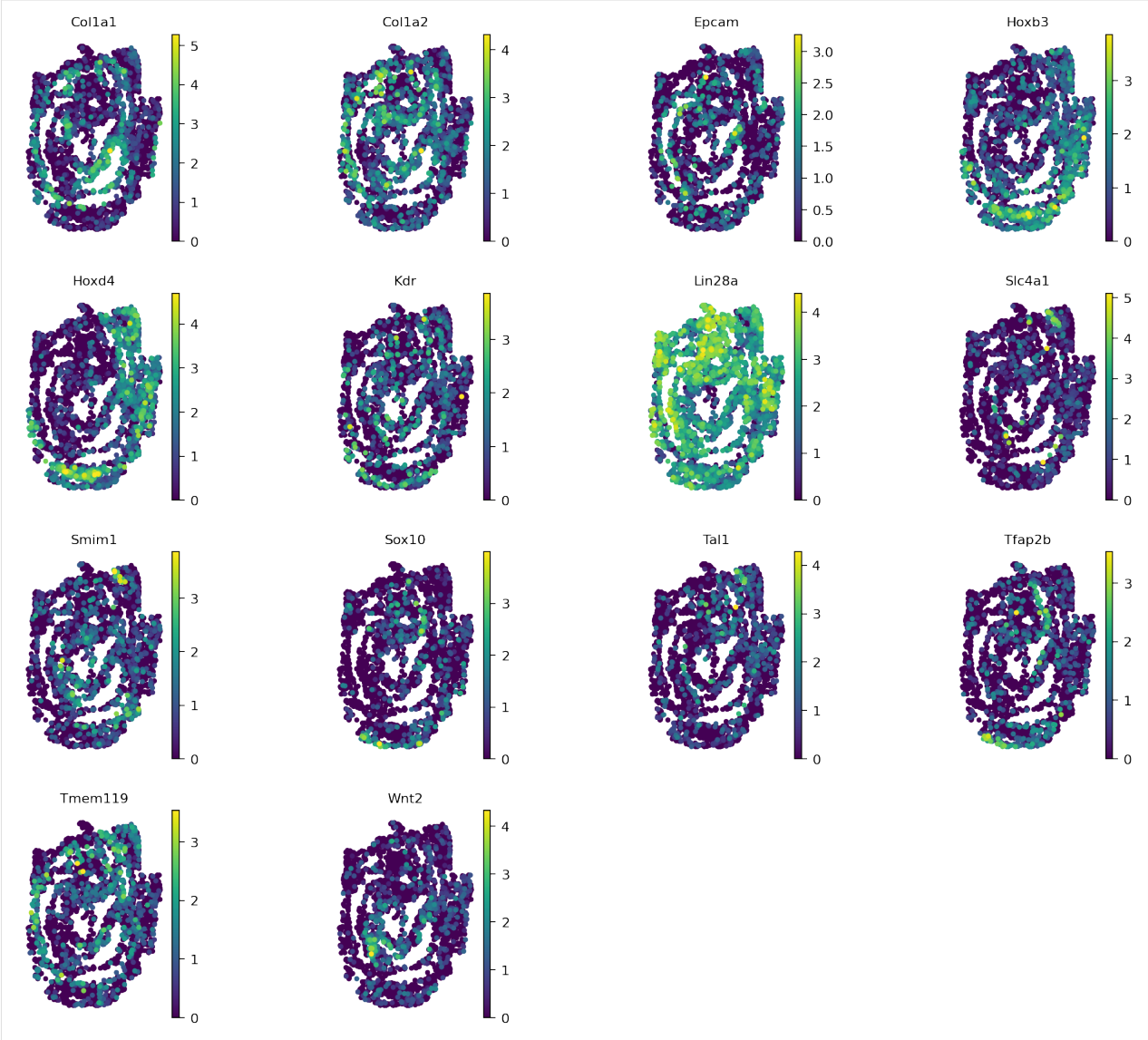
Finally, we train the model by feeding in the reference mode and specefying the `obs` columns containing the partitionings. The model trains one GNN for each partitioning:

[10]:
```
import torch_geometric.data as geo_dt
```

[11]:
```
sg_obj.train(adata_seqFISH1_1, comm_columns=['leiden_0.05', 'leiden_0.1', 'leiden_0.5'],␣
↪tag='seqFISH_ref1', epochs=15, verbose = False, importance=True)
```

By setting `importance = True` we compute the feature (here gene) importances in training. A column specifying these importance values has been added to the adata. This gives us the *Spatially Informative Genes (SIGs)*:

[12]:
```
sc.set_figure_params(figsize=(3, 3), fontsize=10)
from copy import copy
ind   = np.where(adata_seqFISH1_1.var['seqFISH_ref1_importance'] == 0)[0]
SIGs1 = list(adata_seqFISH1_1.var_names[ind])
adata_r = copy(adata_seqFISH1_1)
sc.pp.subsample(adata_r, fraction=0.25)
# with rc_context({'figure.figsize': (2, 2)}):
sc.pl.spatial(adata_r, color=SIGs1, ncols=4, spot_size=0.2, legend_loc=None,␣
↪frameon=False)
```



Saving the trained model

We can save the trained model as a folder:

[13]:
```
!mkdir models
!mkdir models/seqFISH_ref
sg_obj.save_as_folder('models/seqFISH_ref')
%ls -l models/seqFISH_ref
```

```
mkdir: cannot create directory ‘models’: File exists
mkdir: cannot create directory ‘models/seqFISH_ref’: File exists
total 2228
-rw-r--r-- 1 root root 533016 Sep 14 14:22 seqFISH_ref1_leiden_0.05.h5ad
-rw-r--r-- 1 root root 108999 Sep 14 14:22 seqFISH_ref1_leiden_0.05.pickle
-rw-r--r-- 1 root root 533016 Sep 14 14:22 seqFISH_ref1_leiden_0.1.h5ad
```

(continues on next page)

(continued from previous page)

```
-rw-r--r-- 1 root root 176199 Sep 14 14:22 seqFISH_ref1_leiden_0.1.pickle
-rw-r--r-- 1 root root 533016 Sep 14 14:22 seqFISH_ref1_leiden_0.5.h5ad
-rw-r--r-- 1 root root 377927 Sep 14 14:22 seqFISH_ref1_leiden_0.5.pickle
```

The function above saves the `torch` neural networks as well as the adjancency matrices used for training the model in a folder. For reusing the model, one can load the model into another SageNet object:

[14]:
```
sg_obj_load = sca.models.sagenet(device=device)
sg_obj_load.load_from_folder('models/seqFISH_ref')
```

15.3.2 Mapping the query dataset

We can now feed the query dataset into the trained model to get the predicted cell-cell spatial distances. Here, we take the single-cell dataset as the query dataset:

[15]:
```
sg_obj_load.load_query_data(adata_scRNAseq)
```

[16]:
```
adata_scRNAseq
```

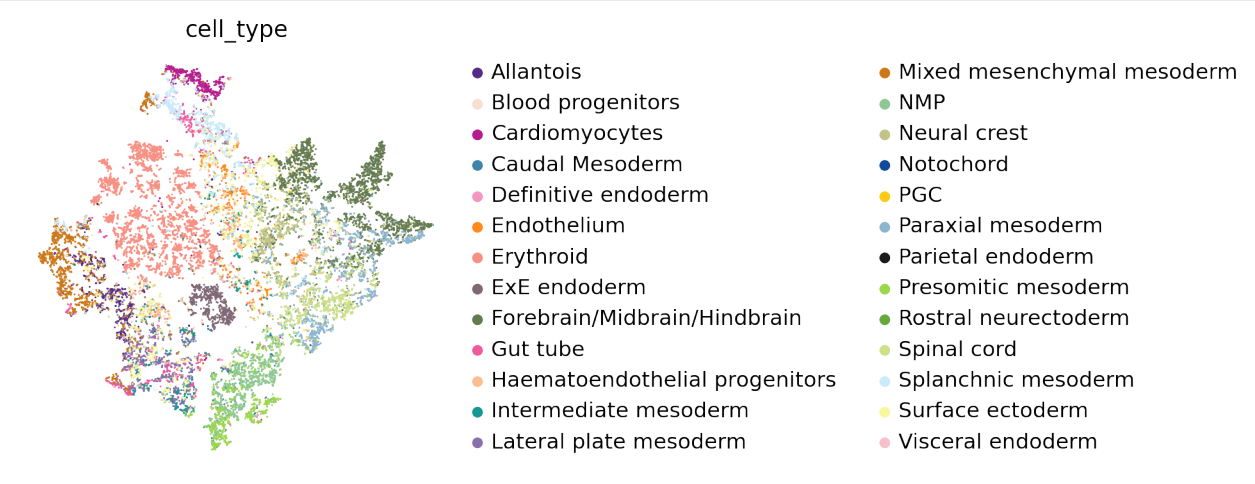
[16]:
```
AnnData object with n_obs × n_vars = 16909 × 350
    obs: 'cell_id', 'barcode', 'sample', 'pool', 'stage', 'sequencing.batch', 'theiler',
↪'doub.density', 'doublet', 'cluster', 'cluster.sub', 'cluster.stage', 'cluster.theiler
↪', 'stripped', 'celltype', 'colour', 'sizeFactor', 'cell_type', 'class_', 'pred_
↪seqFISH_ref1_leiden_0.5', 'ent_seqFISH_ref1_leiden_0.5', 'pred_seqFISH_ref1_leiden_0.1
↪', 'ent_seqFISH_ref1_leiden_0.1', 'pred_seqFISH_ref1_leiden_0.05', 'ent_seqFISH_ref1_
↪leiden_0.05'
    uns: 'X_name', 'cell_types', 'celltype_colours'
    obsm: 'dist_map'
```

The predicted cell-cell distance matrix has been added to the query adata in the `obsm` section under the name `dist_map`. We use t-SNE to map the cells based on this distance matrix in 2 dimensions:

[17]:
```
dist_adata = ad.AnnData(adata_scRNAseq.obsm['dist_map'], obs = adata_scRNAseq.obs)
knn_indices, knn_dists, forest = sc.neighbors.compute_neighbors_umap(dist_adata.X, n_
↪neighbors=50, metric='precomputed')
dist_adata.obsp['distances'], dist_adata.obsp['connectivities'] = sc.neighbors._compute_
↪connectivities_umap(
    knn_indices,
    knn_dists,
    dist_adata.shape[0],
    50 # change to neighbors you plan to use
)
sc.pp.neighbors(dist_adata, metric='precomputed', use_rep='X')
sc.tl.tsne(dist_adata)
```

```
WARNING: You're trying to run this on 16909 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
```

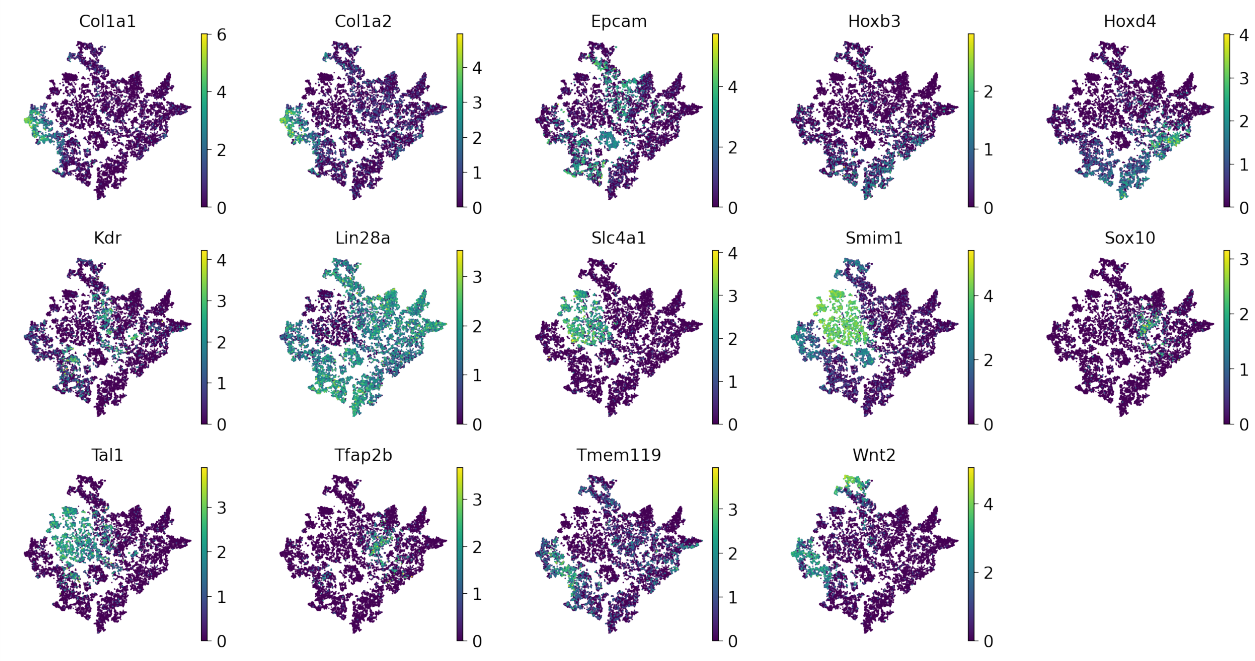
[18]:
```
sc.set_figure_params(figsize=(5, 5), fontsize=15)
sc.pl.tsne(dist_adata, color=['cell_type'], palette=celltype_colours, frameon=False)
```



[19]:
```
cluster_counts = adata_scRNAseq.obs['cell_type'].value_counts()
adata_scRNAseq = adata_scRNAseq[adata_scRNAseq.obs['cell_type'].isin( cluster_
↪counts[cluster_counts>50].index)]
dist_adata = dist_adata[dist_adata.obs['cell_type'].isin( cluster_counts[cluster_counts>
↪50].index)]
```

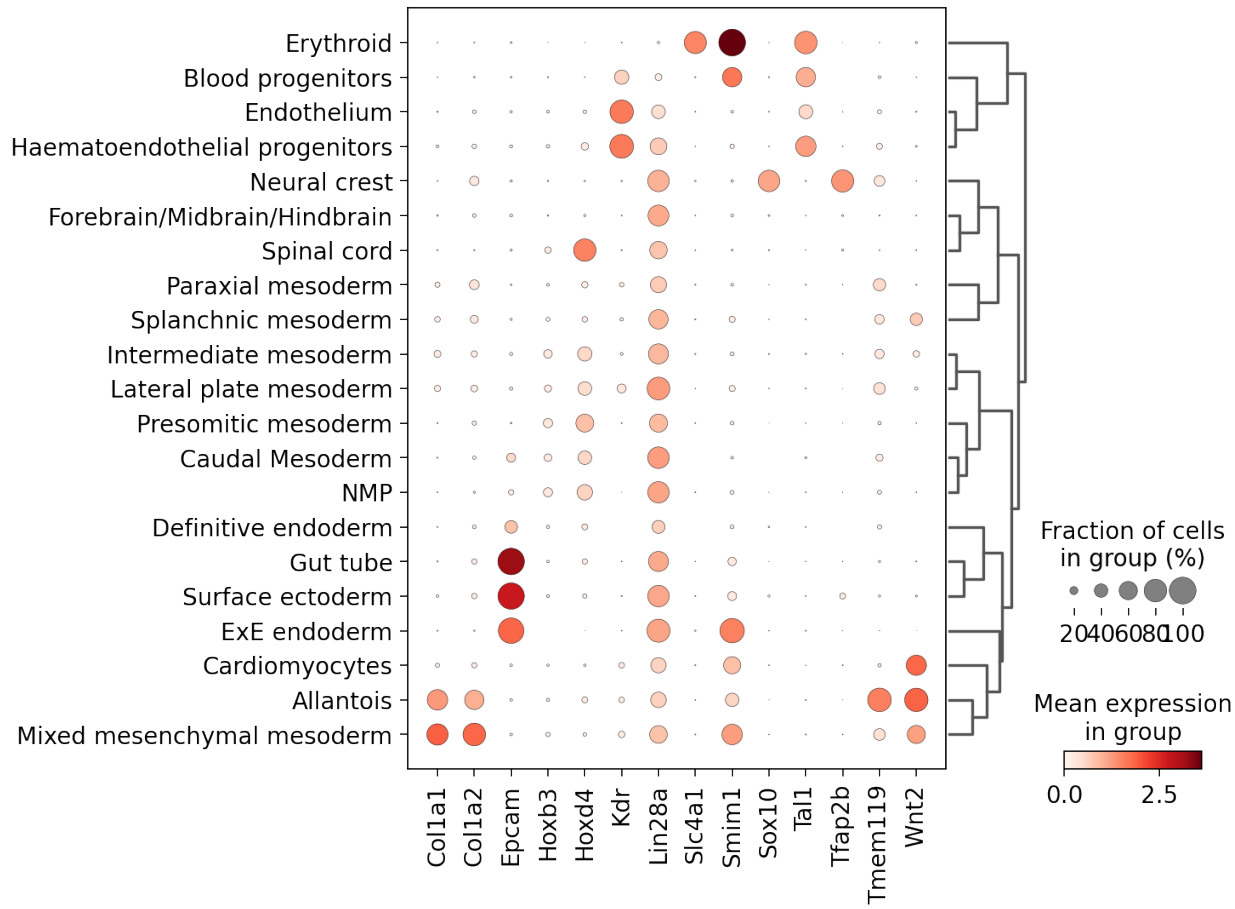
Now we can look at the expression of the SIGs (introduced above) at the 2D reconstructed space:

[20]:
```
adata_scRNAseq.obsm['tsne'] = dist_adata.obsm['X_tsne']
sc.set_figure_params(figsize=(3, 3), fontsize=15)
sc.pl.tsne(adata_scRNAseq, color=SIGs1, ncols=5, legend_loc=None, frameon=False)
```



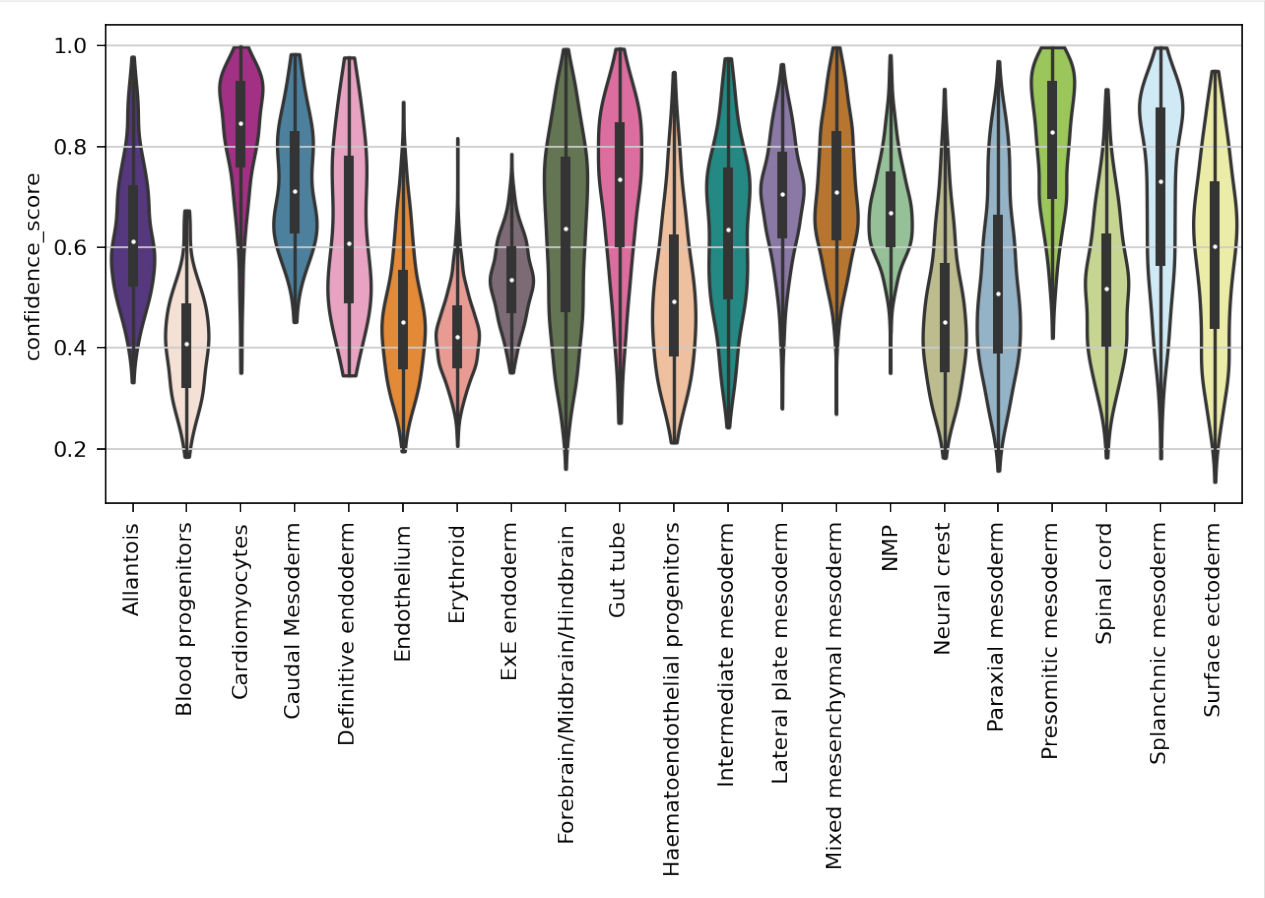
[21]:
```
sc.set_figure_params(figsize=(5, 5), fontsize=15)
sc.pl.dotplot(adata_scRNAseq, SIGs1, groupby='cell_type', dendrogram=True)
```

```
WARNING: dendrogram data not found (using key=dendrogram_cell_type). Running `sc.tl.
↪dendrogram` with default parameters. For fine tuning it is recommended to run `sc.tl.
↪dendrogram` independently.
WARNING: You're trying to run this on 350 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
```



The model outputs a confidence score for each of the cells per GGN (namely per partitioning). This score is between 0 and 1, and higher the less confident is the model to map the corresponding query cell. Therfore, one can summarize the confidence scores from all trained GGNs:

[22]:
```
sc.set_figure_params(figsize=(8, 4), fontsize=10)
adata_scRNAseq.obs['confidence_score'] = (3 - (adata_scRNAseq.obs['ent_seqFISH_ref1_
↪leiden_0.05'] + adata_scRNAseq.obs['ent_seqFISH_ref1_leiden_0.1'] + adata_scRNAseq.obs[
↪'ent_seqFISH_ref1_leiden_0.5']))/3
# with rc_context({'figure.figsize': (4, 2)}):
sc.pl.violin(adata_scRNAseq, ['confidence_score'], groupby='cell_type', palette=celltype_
↪colours, stripplot=False, inner='box', rotation=90)
```



15.3.3 Training with multiple spatial references

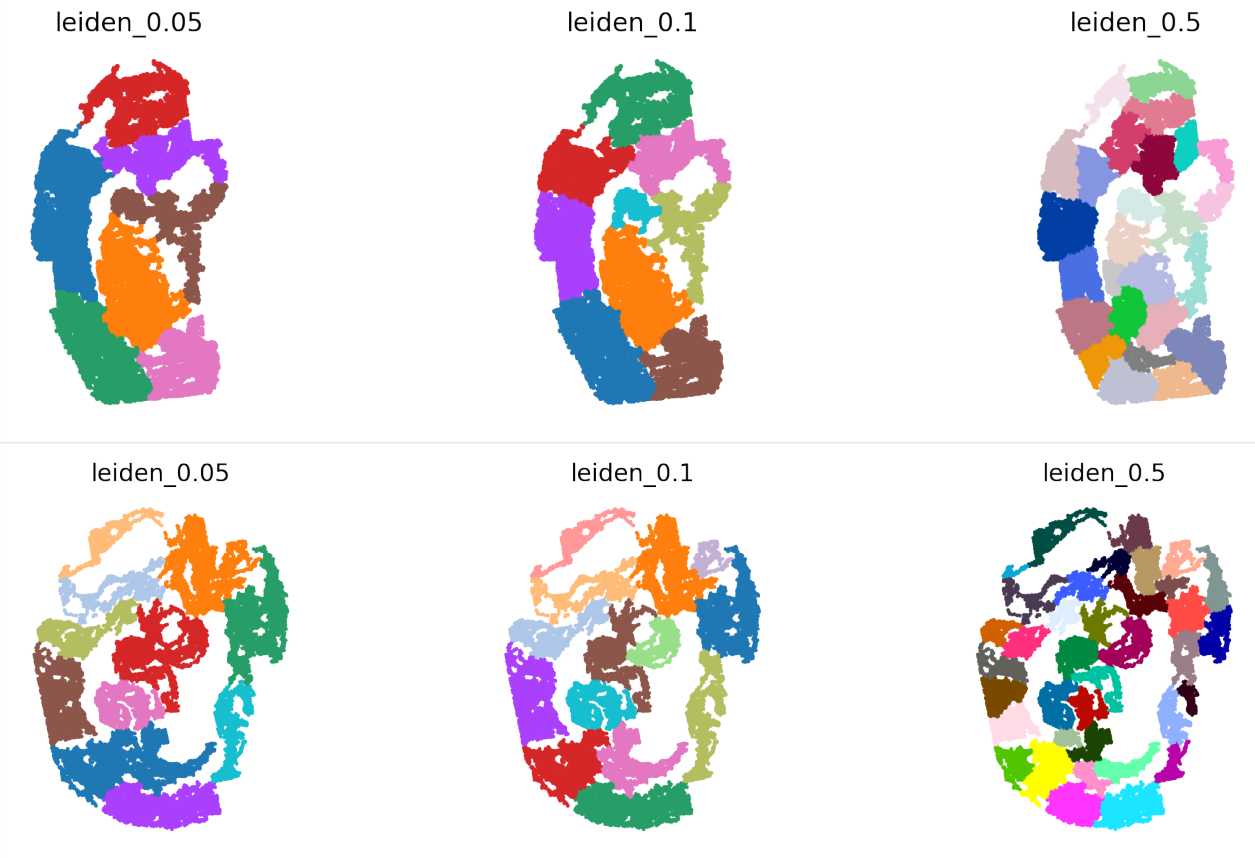
Now we add 2 other spatial references to the model. These are seqFISH datasets from other mouse embryos, `seqFISH2_1` and `seqFISH3_1`. We first perform the same preprocessing steps:

[23]:
```
glasso(adata_seqFISH2_1, [0.25, 0.5])
sc.tl.leiden(adata_seqFISH2_1, resolution=.05, random_state=0, key_added='leiden_0.05',␣
↪adjacency=adata_seqFISH2_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH2_1, resolution=.1, random_state=0, key_added='leiden_0.1',␣
↪adjacency=adata_seqFISH2_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH2_1, resolution=.5, random_state=0, key_added='leiden_0.5',␣
↪adjacency=adata_seqFISH2_1.obsp["spatial_connectivities"])

glasso(adata_seqFISH3_1, [0.25, 0.5])
sc.tl.leiden(adata_seqFISH3_1, resolution=.05, random_state=0, key_added='leiden_0.05',␣
↪adjacency=adata_seqFISH3_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH3_1, resolution=.1, random_state=0, key_added='leiden_0.1',␣
↪adjacency=adata_seqFISH3_1.obsp["spatial_connectivities"])
sc.tl.leiden(adata_seqFISH3_1, resolution=.5, random_state=0, key_added='leiden_0.5',␣
↪adjacency=adata_seqFISH3_1.obsp["spatial_connectivities"])
sc.set_figure_params(figsize=(4, 4), fontsize=15)
sc.pl.spatial(adata_seqFISH2_1, color=['leiden_0.05', 'leiden_0.1', 'leiden_0.5'],␣
↪frameon=False, ncols=3, spot_size=.1, title=['leiden_0.05', 'leiden_0.1', 'leiden_0.5
```

(continues on next page)

(continued from previous page)

```
↪'],  legend_loc=None)
sc.pl.spatial(adata_seqFISH3_1, color=['leiden_0.05', 'leiden_0.1', 'leiden_0.5'],␣
↪frameon=False, ncols=3, spot_size=.1, title=['leiden_0.05', 'leiden_0.1', 'leiden_0.5
↪'],  legend_loc=None)
```



[24]:
```
# !mkdir models
# !mkdir models/seqFISH_ref
sg_obj.save_as_folder('models/seqFISH_ref')
%ls -l models/seqFISH_ref
```

```
total 2228
-rw-r--r-- 1 root root 533016 Sep 14 14:28 seqFISH_ref1_leiden_0.05.h5ad
-rw-r--r-- 1 root root 108999 Sep 14 14:28 seqFISH_ref1_leiden_0.05.pickle
-rw-r--r-- 1 root root 533016 Sep 14 14:28 seqFISH_ref1_leiden_0.1.h5ad
-rw-r--r-- 1 root root 176199 Sep 14 14:28 seqFISH_ref1_leiden_0.1.pickle
-rw-r--r-- 1 root root 533016 Sep 14 14:28 seqFISH_ref1_leiden_0.5.h5ad
-rw-r--r-- 1 root root 377927 Sep 14 14:28 seqFISH_ref1_leiden_0.5.pickle
```

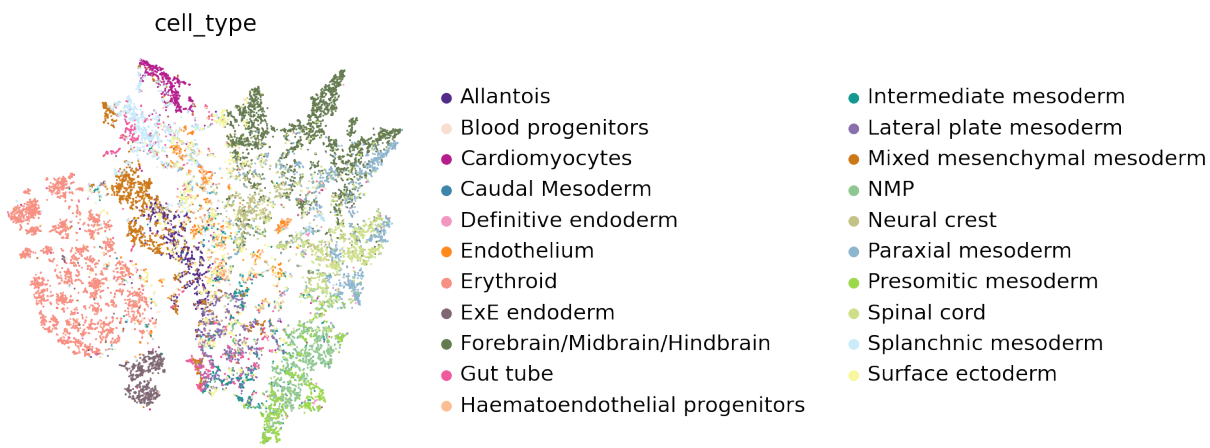
The sagenet model object now includes more models added corresponding to the new spatial references.

Mapping the query dataset

Now let's map the query dataset once again with the new ensemble model:

[25]:
```
sc.set_figure_params(figsize=(5, 5), fontsize=15)
sg_obj_load.load_query_data(adata_scRNAseq)

dist_adata = ad.AnnData(adata_scRNAseq.obsm['dist_map'], obs = adata_scRNAseq.obs)
knn_indices, knn_dists, forest = sc.neighbors.compute_neighbors_umap(dist_adata.X, n_
↪neighbors=50, metric='precomputed')
dist_adata.obsp['distances'], dist_adata.obsp['connectivities'] = sc.neighbors._compute_
↪connectivities_umap(
    knn_indices,
    knn_dists,
    dist_adata.shape[0],
    50 # change to neighbors you plan to use
)
sc.pp.neighbors(dist_adata, metric='precomputed', use_rep='X')
sc.tl.tsne(dist_adata)

sc.pl.tsne(dist_adata, color=['cell_type'], palette=celltype_colours, frameon=False)
```

```
WARNING: You're trying to run this on 16861 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
```



CHAPTER

SIXTEEN

TUTORIAL FOR MVTCR

In this tutorial, we create an atlas for tumor-infiltrating lymphocytes (TIL). The dataset is collected by Borcherding and contains ~750,000 samples from 11 cancer types and 6 tissue sources. In this example we use a subsampled version of Lung cancer specific studies. We use mvTCR to build the atlas and later integrate a heldout query into it. After the data integration, we use it to infer the tissue origin from the heldout dataset using a simple kNN trained on the learned latent representation.

mvTCR is a multi-modal generative integration method for transcriptome and T-cell receptor (TCR) data. For more information on mvTCR please refer to Drost 2022

16.1 Download dataset

First we download the mvTCR package and the preprocessed Tumor-infiltrating Lymphocyte (TIL) dataset. The unpreprocessed dataset can be downloaded at https://github.com/ncborcherding/utility

[1]:
```
%%capture
!pip install mvtcr
```

[2]:
```
import gdown
import os
url = 'https://drive.google.com/uc?id=1Lw9hytk3BiZ8aOucvnhRr9qyeMeNTs4v'
output = 'borcherding_subsampled.h5ad'
if not os.path.exists(output):
    gdown.download(url, output, quiet=False)
```

16.2 Import libraries

[3]:
```
import torch
import scanpy as sc
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score
```

[4]:
```
%%capture
import scarches as sca
```

[5]:
```
%load_ext autoreload
%autoreload 2
```

[6]:
```
random_seed = 42
sca.models.mvTCR.utils_training.fix_seeds(random_seed)
```

16.3 Set hyperparameters

Here the hyperparameters can be set

[7]:
```
# We holdout two Cohorts out of 13
holdout_cohorts = ['GSE162500']

# maps each batch to an index, important is that the holdout cohorts are at the last␣
↪positions
mapper = {'GSE176021': 0, 'GSE139555': 1, 'GSE154826': 2, 'GSE162500': 3}
```

[8]:
```
# Parameters for model and training
params_architecture = {'batch_size': 512,
                       'learning_rate': 0.0006664047426647477,
                       'loss_weights': [1.0, 0.016182440457269676, 1.0110670042409596e-
↪10],
                       'joint': {'activation': 'leakyrelu',
                                 'batch_norm': True,
                                 'dropout': 0.05,
                                 'hdim': 100,
                                 'losses': ['MSE', 'CE'],
                                 'num_layers': 2,
                                 'shared_hidden': [100, 100],
                                 'zdim': 50,
                                 'c_embedding_dim': 20,
                                 'use_embedding_for_cond': True,
                                 'num_conditional_labels': 11,
                                 'cond_dim': 20,
                                 'cond_input': True},
                       'rna': {'activation': 'leakyrelu',
                               'batch_norm': True,
                               'dropout': 0.05,
                               'gene_hidden': [500, 500, 500],
                               'num_layers': 3,
                               'output_activation': 'linear',
                               'xdim': 5000},
                       'tcr': {'embedding_size': 64,
                               'num_heads': 4,
                               'forward_expansion': 4,
                               'encoding_layers': 1,
                               'decoding_layers': 1,
                               'dropout': 0.25,
                               'max_tcr_length': 30,
                               'num_seq_labels': 24}
                      }

params_experiment = {
```

(continues on next page)

(continued from previous page)

```
    'model_name': 'moe',
    'n_epochs': 200,
    'early_stop': 5,
    'balanced_sampling': 'clonotype',
    'kl_annealing_epochs': None,
    'metadata': ['clonotype', 'Sample', 'Type', 'Tissue', 'Tissue+Type', 'functional.
↪cluster'],
    'save_path': 'saved_models',
    'conditional': 'Cohort'
}

params_optimization = {
    'name': 'pseudo_metric',
    'prediction_labels':
        {'clonotype': 1,
         'Tissue+Type': 10}
}
```

16.4 Load Data

We load the data using scanpy

[9]:
```
adata = sc.read_h5ad('borcherding_subsampled.h5ad')
adata = adata[adata.obs['Cohort'].isin(list(mapper.keys()))]
adata = adata[adata.obs['Tissue']=='Lung']
adata.obs['Cohort_id'] = adata.obs['Cohort'].map(mapper)
adata.obs['Tissue+Type'] = [f'{tissue}.{type_}' for tissue, type_ in zip(adata.obs[
↪'Tissue'], adata.obs['Type'])]
metadata = ['Tissue', 'Type', 'Tissue+Type', 'functional.cluster', 'set', 'Cohort']
```

16.5 Train model on atlas dataset

The model is initialized and trained on the atlas dataset until convergence, for early stopping we subsample 20% from the training dataset as validation dataset

[10]:
```
# Split data into training and hold-out dataset
adata.obs['set'] = 'train'
adata.obs['set'][adata.obs['Cohort'].isin(holdout_cohorts)] = 'hold_out'
adata.obsm['Cohort'] = torch.nn.functional.one_hot(torch.tensor(adata.obs['Cohort_id'])).
↪numpy()

# Stratified splitting of training into train and val. The val set is used for early␣
↪stopping
adata_train = adata[~adata.obs['Cohort'].isin(holdout_cohorts)].copy()
adata_train.obsm['Cohort'] = torch.nn.functional.one_hot(torch.tensor(adata_train.obs[
↪'Cohort_id'])).numpy()
train, val = sca.models.mvTCR.utils_preprocessing.group_shuffle_split(adata_train, group_
```

(continues on next page)

(continued from previous page)

```
↪col='clonotype', val_split=0.2, random_seed=random_seed)
adata_train.obs.loc[val.obs.index, 'set'] = 'val'
```

[11]:
```
model = sca.models.mvTCR.models.mixture_modules.moe.MoEModel(adata_train, params_
↪architecture, params_experiment['balanced_sampling'], params_experiment['metadata'],
                                                                  params_experiment[
↪'conditional'], params_optimization)
```

[12]:
```
model.train(params_experiment['n_epochs'], params_architecture['batch_size'], params_
↪architecture['learning_rate'],
            params_architecture['loss_weights'], params_experiment['kl_annealing_epochs
↪'],
            params_experiment['early_stop'], params_experiment['save_path'])
```

```
 46%|                                  | 93/200 [19:18<22:12, 12.45s/it]
```

```
Early stopped
```

16.6 Finetune model on query

To finetune the model, we load the pretrained model with best pseudo metric performance from the previous step. Then we initizalize additional embedding vectors for the query datasets and freeze all weights except the embedding layer, before further training on the query dataset

[13]:
```
# Separate holdout data and create a validation set (20%) for early stopping
adata_hold_out = adata[adata.obs['Cohort'].isin(holdout_cohorts)].copy()
adata_hold_out.obsm['Cohort'] = torch.nn.functional.one_hot(torch.tensor(adata_hold_out.
↪obs['Cohort_id'])).numpy()
train, val = sca.models.mvTCR.utils_preprocessing.group_shuffle_split(adata_hold_out,␣
↪group_col='clonotype', val_split=0.2, random_seed=random_seed)
adata_hold_out.obs['set'] = 'train'
adata_hold_out.obs.loc[val.obs.index, 'set'] = 'val'
```

[14]:
```
# Load pretrained model
model = sca.models.mvTCR.utils_training.load_model(adata_train, f'saved_models/best_
↪model_by_metric.pt', base_path='.')
model.add_new_embeddings(len(holdout_cohorts))  # add new cond embeddings
model.freeze_all_weights_except_cond_embeddings()
model.change_adata(adata_hold_out)  # change the adata to finetune on the holdout data
```

[15]:
```
# Finetune model
model.train(n_epochs=200, batch_size=params_architecture['batch_size'], learning_
↪rate=params_architecture['learning_rate'],
            loss_weights=params_architecture['loss_weights'], kl_annealing_epochs=None,␣
↪early_stop=5,
            save_path=f'saved_models/finetuning/', comet=None)
```

```
  3%|                                                          | 6/200 [00:17<09:
↪25,  2.91s/it]
```

```
Early stopped
```

16.7 Get latent representation and visualize using UMAP

After finetuning, we take a first qualitative look on the latent representation by using UMAP to visualize them

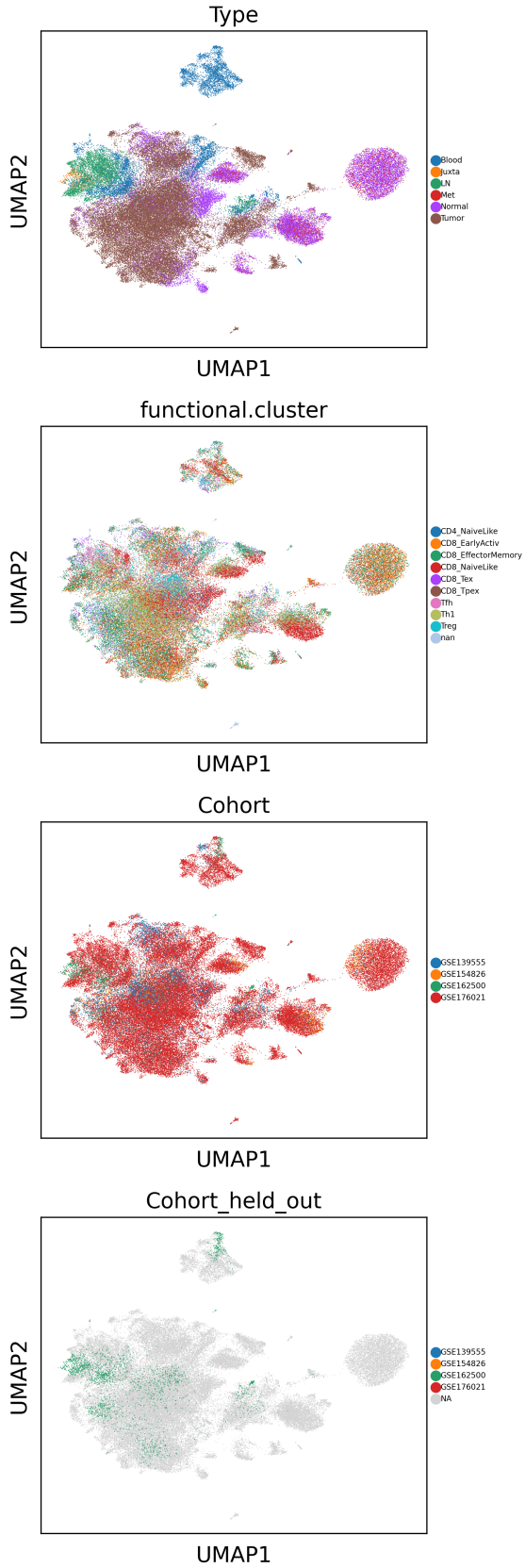
[16]:
```
model = sca.models.mvTCR.utils_training.load_model(adata_hold_out, f'saved_models/
↪finetuning/best_model_by_metric.pt', base_path='.')
```

[17]:
```
latent_adata = model.get_latent(adata, metadata=metadata, return_mean=True)
```

[18]:
```
# For visualization purpose
latent_adata.obs['Cohort_held_out'] = latent_adata.obs['Cohort'].copy()
latent_adata.obs.loc[~latent_adata.obs['Cohort'].isin(holdout_cohorts), 'Cohort_held_out
↪'] = None
```

[19]:
```
sc.pp.neighbors(latent_adata)
sc.tl.umap(latent_adata)
```

[20]:
```
from matplotlib import rcParams
rcParams['figure.figsize'] = (4, 4)
sc.pl.umap(latent_adata, color=['Type', 'functional.cluster', 'Cohort', 'Cohort_held_out
↪'], size=1, ncols=1, legend_fontsize=5)
```


Type
UMAP2
UMAP1
Blood
Juxta
LN
Met
Normal
Tumor
functional.cluster
UMAP2
UMAP1
CD4_NaiveLike
CD8_EarlyActiv
CD8_EffectorMemory
CD8_NaiveLike
CD8_Tex
CD8_Tpex
Tfh
Th1
Treg
nan
Cohort
UMAP2
UMAP1
GSE139555
GSE154826
GSE162500
GSE176021
Cohort_held_out
UMAP2
UMAP1
GSE139555
GSE154826
GSE162500
GSE176021
NA

16.8 Use latent representation for downstream task

We can now use the latent representation for downstream tasks, such as clustering, analysis or imputation. In this example we show how to use a simple kNN classifier to predict the cancer type of the holdout data.

[21]:
```
X_train = latent_adata[latent_adata.obs['set'] == 'train'].X
y_train = latent_adata[latent_adata.obs['set'] == 'train'].obs['Type']
X_test = latent_adata[latent_adata.obs['set'] == 'hold_out'].X
y_test = latent_adata[latent_adata.obs['set'] == 'hold_out'].obs['Type']
```

[22]:
```
classifier = KNeighborsClassifier(n_neighbors=5, weights='distance', n_jobs=-1)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

[23]:
```
print(f'F1-score for predicting Sample Origin: {f1_score(y_test, y_pred, average=
↪"weighted"):.3}')
```

```
F1-score for predicting Sample Origin: 0.744
```

CHAPTER

SEVENTEEN

INTEGRATION AND REFERENCE MAPPING WITH MULTIGRATE

In this notebook, we demonstrate how to use Multigrate with scArches: we build a trimodal reference atlas with Multigrate by integrating CITE-seq and multiome data, and map unimodal as well as multimodal queries onto the reference. We use publically available datasets from NeurIPS 2021 workshop https://openproblems.bio/neurips_2021/.

[1]:
```
import scarches as sca
import scanpy as sc
import anndata as ad
import numpy as np
import muon
import gdown
import json

import warnings
warnings.filterwarnings("ignore")

sc.set_figure_params(figsize=(4, 4), fontsize=8)
```

```
WARNING:root:In order to use the mouse gastrulation seqFISH datsets, please install␣
↪squidpy (see https://github.com/scverse/squidpy).
WARNING:root:In order to use sagenet models, please install pytorch geometric (see https:
↪//pytorch-geometric.readthedocs.io) and
 captum (see https://github.com/pytorch/captum).
INFO:pytorch_lightning.utilities.seed:Global seed set to 0
/lustre/groups/ml01/workspace/anastasia.litinetskaya/miniconda3/envs/scarches/lib/
↪python3.9/site-packages/pytorch_lightning/utilities/warnings.py:53:␣
↪LightningDeprecationWarning: pytorch_lightning.utilities.warnings.rank_zero_
↪deprecation has been deprecated in v1.6 and will be removed in v1.8. Use the␣
↪equivalent function from the pytorch_lightning.utilities.rank_zero module instead.
  new_rank_zero_deprecation(
/lustre/groups/ml01/workspace/anastasia.litinetskaya/miniconda3/envs/scarches/lib/
↪python3.9/site-packages/pytorch_lightning/utilities/warnings.py:58:␣
↪LightningDeprecationWarning: The `pytorch_lightning.loggers.base.rank_zero_experiment`␣
↪is deprecated in v1.7 and will be removed in v1.9. Please use `pytorch_lightning.
↪loggers.logger.rank_zero_experiment` instead.
  return new_rank_zero_deprecation(*args, **kwargs)
WARNING:root:mvTCR is not installed. To use mvTCR models, please install it first using
↪"pip install mvtcr"
```

17.1 Data preprocessing

First, we download the datasets and split them into AnnData objects corresponding to individual modalities: gene expression (RNA) and protein abundance (ADT) for CITE-seq, and gene expression (RNA) and chromatin opennes (ATAC) for multiome.

[2]:
```
# download
!wget 'ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE194nnn/GSE194122/suppl/GSE194122_
↪openproblems_neurips2021_cite_BMMC_processed.h5ad.gz'
!wget 'ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE194nnn/GSE194122/suppl/GSE194122_
↪openproblems_neurips2021_multiome_BMMC_processed.h5ad.gz'
# unzip
!gzip -d GSE194122_openproblems_neurips2021_cite_BMMC_processed.h5ad.gz
!gzip -d GSE194122_openproblems_neurips2021_multiome_BMMC_processed.h5ad.gz
```

```
--2022-11-24 18:09:53--  ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE194nnn/GSE194122/suppl/
↪GSE194122_openproblems_neurips2021_cite_BMMC_processed.h5ad.gz
           => 'GSE194122_openproblems_neurips2021_cite_BMMC_processed.h5ad.gz'
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 165.112.9.228, 165.112.9.229,␣
↪2607:f220:41e:250::11, ...
Connecting to ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)|165.112.9.228|:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD (1) /geo/series/GSE194nnn/GSE194122/suppl ... done.
==> SIZE GSE194122_openproblems_neurips2021_cite_BMMC_processed.h5ad.gz ... 615842052
==> PASV ... done.    ==> RETR GSE194122_openproblems_neurips2021_cite_BMMC_processed.
↪h5ad.gz ... done.
Length: 615842052 (587M) (unauthoritative)

100%[===================================>] 615,842,052 10.2MB/s   in 75s

2022-11-24 18:11:10 (7.81 MB/s) - 'GSE194122_openproblems_neurips2021_cite_BMMC_
↪processed.h5ad.gz' saved [615842052]

--2022-11-24 18:11:11--  ftp://ftp.ncbi.nlm.nih.gov/geo/series/GSE194nnn/GSE194122/suppl/
↪GSE194122_openproblems_neurips2021_multiome_BMMC_processed.h5ad.gz
           => 'GSE194122_openproblems_neurips2021_multiome_BMMC_processed.h5ad.gz'
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 165.112.9.228, 165.112.9.229,␣
↪2607:f220:41e:250::11, ...
Connecting to ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)|165.112.9.228|:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.    ==> PWD ... done.
==> TYPE I ... done.  ==> CWD (1) /geo/series/GSE194nnn/GSE194122/suppl ... done.
==> SIZE GSE194122_openproblems_neurips2021_multiome_BMMC_processed.h5ad.gz ...␣
↪2917117242
==> PASV ... done.    ==> RETR GSE194122_openproblems_neurips2021_multiome_BMMC_
↪processed.h5ad.gz ... done.
Length: 2917117242 (2.7G) (unauthoritative)

100%[=====================================>] 2,917,117,242 7.98MB/s   in 5m 52s

2022-11-24 18:17:04 (7.91 MB/s) - 'GSE194122_openproblems_neurips2021_multiome_BMMC_
↪processed.h5ad.gz' saved [2917117242]
```

[3]:
```
cite = sc.read('GSE194122_openproblems_neurips2021_cite_BMMC_processed.h5ad')
cite
```

[3]:
```
AnnData object with n_obs × n_vars = 90261 × 14087
    obs: 'GEX_n_genes_by_counts', 'GEX_pct_counts_mt', 'GEX_size_factors', 'GEX_phase',
↪'ADT_n_antibodies_by_counts', 'ADT_total_counts', 'ADT_iso_count', 'cell_type', 'batch
↪', 'ADT_pseudotime_order', 'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber',
↪ 'Modality', 'VendorLot', 'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType',
↪'DonorRace', 'Ethnicity', 'DonorGender', 'QCMeds', 'DonorSmoker', 'is_train'
    var: 'feature_types', 'gene_id'
    uns: 'dataset_id', 'genome', 'organism'
    obsm: 'ADT_X_pca', 'ADT_X_umap', 'ADT_isotype_controls', 'GEX_X_pca', 'GEX_X_umap'
    layers: 'counts'
```

[4]:
```
rna_cite = cite[:, cite.var['feature_types'] == 'GEX'].copy()
adt = cite[:, cite.var['feature_types'] == 'ADT'].copy()
rna_cite.shape, adt.shape
```

[4]:
```
((90261, 13953), (90261, 134))
```

[5]:
```
multiome = sc.read('GSE194122_openproblems_neurips2021_multiome_BMMC_processed.h5ad')
multiome
```

[5]:
```
AnnData object with n_obs × n_vars = 69249 × 129921
    obs: 'GEX_pct_counts_mt', 'GEX_n_counts', 'GEX_n_genes', 'GEX_size_factors', 'GEX_
↪phase', 'ATAC_nCount_peaks', 'ATAC_atac_fragments', 'ATAC_reads_in_peaks_frac', 'ATAC_
↪blacklist_fraction', 'ATAC_nucleosome_signal', 'cell_type', 'batch', 'ATAC_pseudotime_
↪order', 'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber', 'Modality',
↪'VendorLot', 'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType', 'DonorRace',
↪'Ethnicity', 'DonorGender', 'QCMeds', 'DonorSmoker'
    var: 'feature_types', 'gene_id'
    uns: 'ATAC_gene_activity_var_names', 'dataset_id', 'genome', 'organism'
    obsm: 'ATAC_gene_activity', 'ATAC_lsi_full', 'ATAC_lsi_red', 'ATAC_umap', 'GEX_X_pca
↪', 'GEX_X_umap'
    layers: 'counts'
```

[6]:
```
rna_multiome = multiome[:, multiome.var['feature_types'] == 'GEX'].copy()
atac = multiome[:, multiome.var['feature_types'] == 'ATAC'].copy()
rna_multiome.shape, atac.shape
```

[6]:
```
((69249, 13431), (69249, 116490))
```

17.1.1 RNA preprocessing

Multigrate model can work with raw counts modelled with negative binomial (NB loss) distribution or normalized counts modelled with Gaussian (MSE loss). We also subset the genes to the top 4000 highly variable genes to speed up calculations. Hence, we need concatenate both RNA objects, normalize the counts per cell, subset the genes to the batch-aware highly variable genes and split the object back into two, correpsonding to the two experiments. Note that in this notebook we will work with raw counts for RNA-seq data but we need normalized counts to select highly variable genes.

[7]:
```
# concat
rna = ad.concat([rna_cite, rna_multiome])
```

(continues on next page)

(continued from previous page)

```
# normalize
rna.X = rna.layers['counts'].copy()
sc.pp.normalize_total(rna, target_sum=1e4)
sc.pp.log1p(rna)
# subset to hvg
sc.pp.highly_variable_genes(rna, n_top_genes=4000, batch_key='Samplename')
rna = rna[:, rna.var.highly_variable].copy()
# split again
rna_cite = rna[rna.obs['Modality'] == 'cite'].copy()
rna_multiome = rna[rna.obs['Modality'] == 'multiome'].copy()
rna_multiome.shape, rna_cite.shape
```

[7]:
```
((69249, 4000), (90261, 4000))
```

17.1.2 ADT preprocessing

For ADT modality, Multigrate requires normalized counts, so we normalize the raw counts using the CLR transformation.

[8]:
```
adt.X = adt.layers['counts'].copy()
muon.prot.pp.clr(adt)
adt.layers['clr'] = adt.X.copy()
adt
```

[8]:
```
AnnData object with n_obs × n_vars = 90261 × 134
    obs: 'GEX_n_genes_by_counts', 'GEX_pct_counts_mt', 'GEX_size_factors', 'GEX_phase',
↪'ADT_n_antibodies_by_counts', 'ADT_total_counts', 'ADT_iso_count', 'cell_type', 'batch
↪', 'ADT_pseudotime_order', 'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber',
↪ 'Modality', 'VendorLot', 'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType',
↪'DonorRace', 'Ethnicity', 'DonorGender', 'QCMeds', 'DonorSmoker', 'is_train'
    var: 'feature_types', 'gene_id'
    uns: 'dataset_id', 'genome', 'organism'
    obsm: 'ADT_X_pca', 'ADT_X_umap', 'ADT_isotype_controls', 'GEX_X_pca', 'GEX_X_umap'
    layers: 'counts', 'clr'
```

17.1.3 ATAC preprocessing

We recommend log-normalized or tf-idf transformed ATAC counts with Multigrate. Similarly to RNA-seq, we subset the features to the top 20,000 highly variable features to speed up integration.

[9]:
```
atac.X = atac.layers['counts'].copy()
sc.pp.normalize_total(atac, target_sum=1e4)
sc.pp.log1p(atac)
atac.layers['log-norm'] = atac.X.copy()
atac
```

[9]:
```
AnnData object with n_obs × n_vars = 69249 × 116490
    obs: 'GEX_pct_counts_mt', 'GEX_n_counts', 'GEX_n_genes', 'GEX_size_factors', 'GEX_
↪phase', 'ATAC_nCount_peaks', 'ATAC_atac_fragments', 'ATAC_reads_in_peaks_frac', 'ATAC_
↪blacklist_fraction', 'ATAC_nucleosome_signal', 'cell_type', 'batch', 'ATAC_pseudotime_
↪order', 'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber', 'Modality',
```

(continues on next page)

(continued from previous page)

```
↪'VendorLot', 'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType', 'DonorRace',
↪'Ethnicity', 'DonorGender', 'QCMeds', 'DonorSmoker'
    var: 'feature_types', 'gene_id'
    uns: 'ATAC_gene_activity_var_names', 'dataset_id', 'genome', 'organism', 'log1p'
    obsm: 'ATAC_gene_activity', 'ATAC_lsi_full', 'ATAC_lsi_red', 'ATAC_umap', 'GEX_X_pca
↪', 'GEX_X_umap'
    layers: 'counts', 'log-norm'
```

[10]:
```
sc.pp.highly_variable_genes(atac, n_top_genes=20000, batch_key='batch')
atac = atac[:, atac.var.highly_variable].copy()
atac
```

[10]:
```
AnnData object with n_obs × n_vars = 69249 × 20000
    obs: 'GEX_pct_counts_mt', 'GEX_n_counts', 'GEX_n_genes', 'GEX_size_factors', 'GEX_
↪phase', 'ATAC_nCount_peaks', 'ATAC_atac_fragments', 'ATAC_reads_in_peaks_frac', 'ATAC_
↪blacklist_fraction', 'ATAC_nucleosome_signal', 'cell_type', 'batch', 'ATAC_pseudotime_
↪order', 'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber', 'Modality',
↪'VendorLot', 'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType', 'DonorRace',
↪'Ethnicity', 'DonorGender', 'QCMeds', 'DonorSmoker'
    var: 'feature_types', 'gene_id', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm', 'highly_variable_nbatches', 'highly_variable_intersection'
    uns: 'ATAC_gene_activity_var_names', 'dataset_id', 'genome', 'organism', 'log1p',
↪'hvg'
    obsm: 'ATAC_gene_activity', 'ATAC_lsi_full', 'ATAC_lsi_red', 'ATAC_umap', 'GEX_X_pca
↪', 'GEX_X_umap'
    layers: 'counts', 'log-norm'
```

17.1.4 Add harmonized cell type labels

Since the cell type annotations are not harmonized in the original data, we rename some of the cell types so they align between CITE-seq and multiome datasets.

[11]:
```
gdown.download("https://drive.google.com/u/1/uc?id=1D54P3jURwkdA3goPqYuby0qx6RzKqcP2")
```

```
Downloading...
From: https://drive.google.com/u/1/uc?id=1D54P3jURwkdA3goPqYuby0qx6RzKqcP2
To: /lustre/groups/ml01/workspace/anastasia.litinetskaya/code/scarches/notebooks/
↪celltype_harmonize.json
100%|| 4.77k/4.77k [00:00<00:00, 3.71MB/s]
```

[11]:
```
'celltype_harmonize.json'
```

[12]:
```
with open('celltype_harmonize.json', 'r') as f:
    harmonized_celltypes = json.load(f)
harmonized_celltypes.keys()
```

[12]:
```
dict_keys(['cite_ct_l1_map', 'cite_ct_l2_map', 'multi_ct_l1_map', 'multi_ct_l2_map'])
```

[13]:
```
rna_multiome.obs['l1_cell_type'] = rna_multiome.obs['cell_type'].map(harmonized_
↪celltypes['multi_ct_l1_map'])
rna_multiome.obs['l2_cell_type'] = rna_multiome.obs['cell_type'].map(harmonized_
↪celltypes['multi_ct_l2_map'])
```

(continues on next page)

(continued from previous page)

```
atac.obs['l1_cell_type'] = atac.obs['cell_type'].map(harmonized_celltypes['multi_ct_l1_
↪map'])
atac.obs['l2_cell_type'] = atac.obs['cell_type'].map(harmonized_celltypes['multi_ct_l2_
↪map'])

rna_cite.obs['l1_cell_type'] = rna_cite.obs['cell_type'].map(harmonized_celltypes['cite_
↪ct_l1_map'])
rna_cite.obs['l2_cell_type'] = rna_cite.obs['cell_type'].map(harmonized_celltypes['cite_
↪ct_l2_map'])

adt.obs['l1_cell_type'] = adt.obs['cell_type'].map(harmonized_celltypes['cite_ct_l1_map
↪'])
adt.obs['l2_cell_type'] = adt.obs['cell_type'].map(harmonized_celltypes['cite_ct_l2_map
↪'])
```

17.1.5 Subset to reference and query

We split the dataset into a reference (3 batches) and a query (2 batches).

[14]:
```
# define the reference and the query batches
cite_reference_batches = ['s1d1', 's1d2', 's1d3']
multiome_reference_batches = ['s1d1', 's1d2', 's1d3']
cite_query_batches = ['s2d1', 's2d4']
multiome_query_batches = ['s2d1', 's2d4']
# query
rna_multiome_query = rna_multiome[rna_multiome.obs['batch'].isin(multiome_query_
↪batches)].copy()
atac_query = atac[atac.obs['batch'].isin(multiome_query_batches)].copy()
rna_cite_query = rna_cite[rna_cite.obs['batch'].isin(cite_query_batches)].copy()
adt_query = adt[adt.obs['batch'].isin(cite_query_batches)].copy()
# reference
rna_multiome = rna_multiome[rna_multiome.obs['batch'].isin(multiome_reference_batches)].
↪copy()
atac = atac[atac.obs['batch'].isin(multiome_reference_batches)].copy()
rna_cite = rna_cite[rna_cite.obs['batch'].isin(cite_reference_batches)].copy()
adt = adt[adt.obs['batch'].isin(cite_reference_batches)].copy()
```

17.2 Prep the input AnnData object

First, we need to organize AnnData objects correspoding to different datasets and modalities into 1 AnnData object. In this example we have 1 CITE-seq dataset and 1 multiome dataset, hence we input 4 anndata objects: 2 (RNA and ADT) for CITE-seq and 2 (RNA and ATAC) for multiome.

Notes: - paired datasets have to have the same .obs_names, i.e. index; - each sublist in `adatas`, `layers` parameters corresponds to one modality. If you have multiple objects per modality, append them to the corresponding list; - objects in each sublist have to have the same set of features: if you want to integrate multiple RNA objects, we recommend first concatenating full objects and then subsetting to 2000-4000 highly variable genes; for ADT modality, we take the intersection of available proteins (double check the naming conventions, as that can vary a lot from one dataset to another, so `.var_names` can have almost no intersection but if you align the protein names, then there is an overlap);

- `layers` parameter specifies which layer the model should take the counts from. If `None`, then defaults to `.X`. The distribution of the input data should be the same within a modality, e.g. here we use raw counts for RNA-seq modality.

[15]:
```
adata = sca.models.organize_multiome_anndatas(
    adatas = [[rna_cite, rna_multiome], [None, atac], [adt, None]],    # a list of␣
↪anndata objects per modality, RNA-seq always goes first
    layers = [['counts', 'counts'], [None, 'log-norm'], ['clr', None]], # if need to use␣
↪data from .layers, if None use .X
)
adata
```

[15]:
```
AnnData object with n_obs × n_vars = 33554 × 24134
    obs: 'GEX_pct_counts_mt', 'GEX_size_factors', 'GEX_phase', 'cell_type', 'batch',
↪'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber', 'Modality', 'VendorLot',
↪'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType', 'DonorRace', 'Ethnicity',
↪'DonorGender', 'QCMeds', 'DonorSmoker', 'l1_cell_type', 'l2_cell_type', 'group', 'ADT_
↪iso_count', 'ADT_n_antibodies_by_counts', 'ADT_pseudotime_order', 'ADT_total_counts',
↪'GEX_n_genes_by_counts', 'is_train', 'ATAC_atac_fragments', 'ATAC_blacklist_fraction',
↪'ATAC_nCount_peaks', 'ATAC_nucleosome_signal', 'ATAC_pseudotime_order', 'ATAC_reads_in_
↪peaks_frac', 'GEX_n_counts', 'GEX_n_genes'
    var: 'modality'
    uns: 'modality_lengths'
    layers: 'counts'
```

From now on, we work with one concatenated anndata object `adata`.

If using raw counts for scRNA, we recommend using NB loss (or ZINB), thus we need to calculate `size_factors` by specifying the `rna_indices_end` parameter. If using normalized counts and MSE for scRNA, `rna_indices_end` does not need to be specified. Here we also want to correct for batch effects, so we specify `Samplename` as a categorical covariate. Since we want to integrate 2 different modalities, we need to register `Modality` as an additional covariate.

[16]:
```
sca.models.MultiVAE.setup_anndata(
    adata,
    categorical_covariate_keys=['Modality', 'Samplename'],
    rna_indices_end=4000,
)
```

```
WARNING:jax._src.lib.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_
↪LOG_LEVEL=0 and rerun for more info.)
```

17.3 Initialize the model

Next, we initialize the model. If using raw counts for RNA-seq, use NB loss, if normalized counts, use MSE. For ADT we use CLR-normalized counts and MSE loss. We need to specify `mmd='marginal'` and set the coeficient to the integration loss if we want to later map unimodal data onto this reference.

[17]:
```
model = sca.models.MultiVAE(
    adata,
    losses=['nb', 'mse', 'mse'],
    loss_coefs={'kl': 1e-1,
                'integ': 3000,
                },
    integrate_on='Modality',
```

(continues on next page)

(continued from previous page)

```
    mmd='marginal',
)
```

17.4 Train the model

You can specify the number of epochs by setting `max_epochs` parameter, default is 200. The default batch size is set to `batch_size = 256`, adjust if needed.

[18]:
```
model.train()
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 200/200: 100%|| 200/200 [27:27<00:00,  8.04s/it, loss=1.88e+03, v_num=1]
```

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=200`␣
↪reached.
```

```
Epoch 200/200: 100%|| 200/200 [27:28<00:00,  8.24s/it, loss=1.88e+03, v_num=1]
```

17.5 Inference

Next, we get the latent representation for all the cells and save them in `.obsm['latent_ref']` as `.obsm['latent']` will be later overwritten when we fine-tune the model on the query.

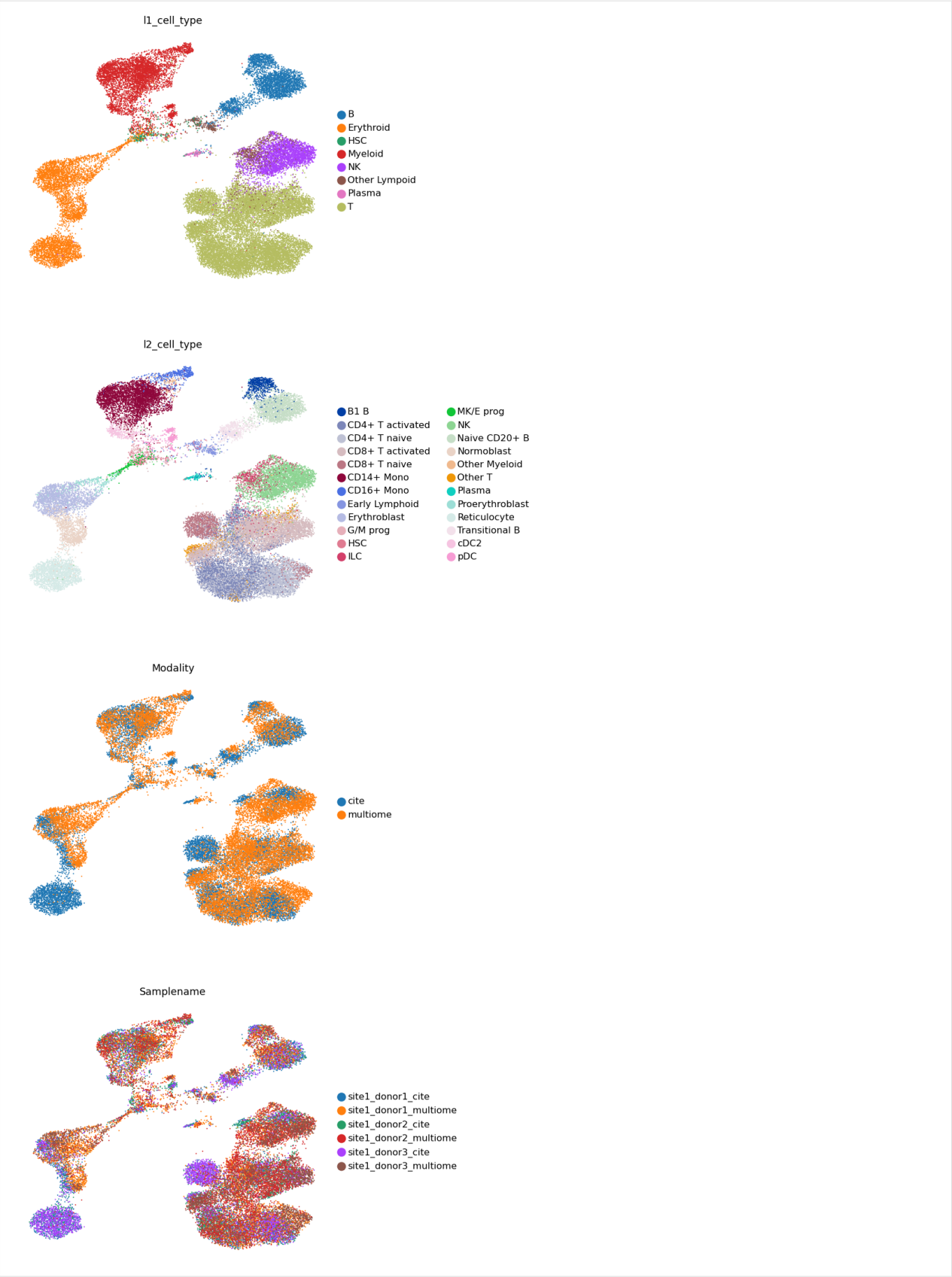
[19]:
```
model.get_latent_representation()
adata.obsm['latent_ref'] = adata.obsm['latent'].copy()
adata
```

[19]:
```
AnnData object with n_obs × n_vars = 33554 × 24134
    obs: 'GEX_pct_counts_mt', 'GEX_size_factors', 'GEX_phase', 'cell_type', 'batch',
↪'GEX_pseudotime_order', 'Samplename', 'Site', 'DonorNumber', 'Modality', 'VendorLot',
↪'DonorID', 'DonorAge', 'DonorBMI', 'DonorBloodType', 'DonorRace', 'Ethnicity',
↪'DonorGender', 'QCMeds', 'DonorSmoker', 'l1_cell_type', 'l2_cell_type', 'group', 'ADT_
↪iso_count', 'ADT_n_antibodies_by_counts', 'ADT_pseudotime_order', 'ADT_total_counts',
↪'GEX_n_genes_by_counts', 'is_train', 'ATAC_atac_fragments', 'ATAC_blacklist_fraction',
↪'ATAC_nCount_peaks', 'ATAC_nucleosome_signal', 'ATAC_pseudotime_order', 'ATAC_reads_in_
↪peaks_frac', 'GEX_n_counts', 'GEX_n_genes', 'size_factors', '_scvi_batch'
    var: 'modality'
    uns: 'modality_lengths', '_scvi_uuid', '_scvi_manager_uuid'
    obsm: '_scvi_extra_categorical_covs', 'latent', 'latent_ref'
    layers: 'counts'
```

Visualize the integrated latent embedding.

[20]:
```
sc.pp.neighbors(adata, use_rep='latent')
sc.tl.umap(adata)
```

[21]:
```
sc.pl.umap(adata, color=['l1_cell_type', 'l2_cell_type', 'Modality', 'Samplename'],␣
↪frameon=False, ncols=1)
```


l1_cell_type
B
Erythroid
HSC
Myeloid
NK
Other Lympoid
Plasma
T
l2_cell_type
B1 B
CD4+ T activated
CD4+ T naive
CD8+ T activated
CD8+ T naive
CD14+ Mono
CD16+ Mono
Early Lymphoid
Erythroblast
G/M prog
HSC
ILC
MK/E prog
NK
Naive CD20+ B
Normoblast
Other Myeloid
Other T
Plasma
Proerythroblast
Reticulocyte
Transitional B
cDC2
pDC
Modality
cite
multiome
Samplename
site1_donor1_cite
site1_donor1_multiome
site1_donor2_cite
site1_donor2_multiome
site1_donor3_cite
site1_donor3_multiome

17.6 Query mapping

We repeat the same steps for the setting up the query as for the reference before.

[22]:
```
query = sca.models.organize_multiome_anndatas(
    adatas = [[rna_cite_query, rna_multiome_query], [None, atac_query], [adt_query,␣
↪None]],
    layers = [['counts', 'counts'], [None, 'log-norm'], ['clr', None]],
)
```

[23]:
```
sca.models.MultiVAE.setup_anndata(
    query,
    categorical_covariate_keys=['Modality', 'Samplename'],
    rna_indices_end=4000,
)
```

We imitate a unimodal RNA-seq query by masking with zeros the ADT part of one CITE-seq batch and RNA part of one multiome batch.

[24]:
```
idx_atac_query = query.obs['Samplename'] == 'site2_donor4_multiome'
idx_scrna_query = query.obs['Samplename'] == 'site2_donor1_cite'

idx_mutiome_query = query.obs['Samplename'] == 'site2_donor1_multiome'
idx_cite_query = query.obs['Samplename'] == 'site2_donor4_cite'

np.sum(idx_atac_query), np.sum(idx_scrna_query), np.sum(idx_mutiome_query), np.sum(idx_
↪cite_query)
```

[24]:
```
(6111, 10465, 4220, 5584)
```

[25]:
```
query[idx_atac_query, :4000].X = 0
query[idx_scrna_query, 4000:].X = 0
```

We update the model by adding new weights to the new batches in the query and fine-tune those weights.

[26]:
```
q_model = sca.models.MultiVAE.load_query_data(query, model)
```

[27]:
```
q_model.train(weight_decay=0)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 200/200: 100%|| 200/200 [20:35<00:00,  6.21s/it, loss=1.38e+03, v_num=1]
```

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=200`␣
↪reached.
```

```
Epoch 200/200: 100%|| 200/200 [20:35<00:00,  6.18s/it, loss=1.38e+03, v_num=1]
```

We obtain the latent representation for the query and the reference from the updated model. Note that the representation of the reference is the same as before up to sampling noise.

[28]:
```
q_model.get_latent_representation(adata=query)
q_model.get_latent_representation(adata=adata)
```

```
INFO     Input AnnData not setup with scvi-tools. attempting to transfer AnnData setup
```

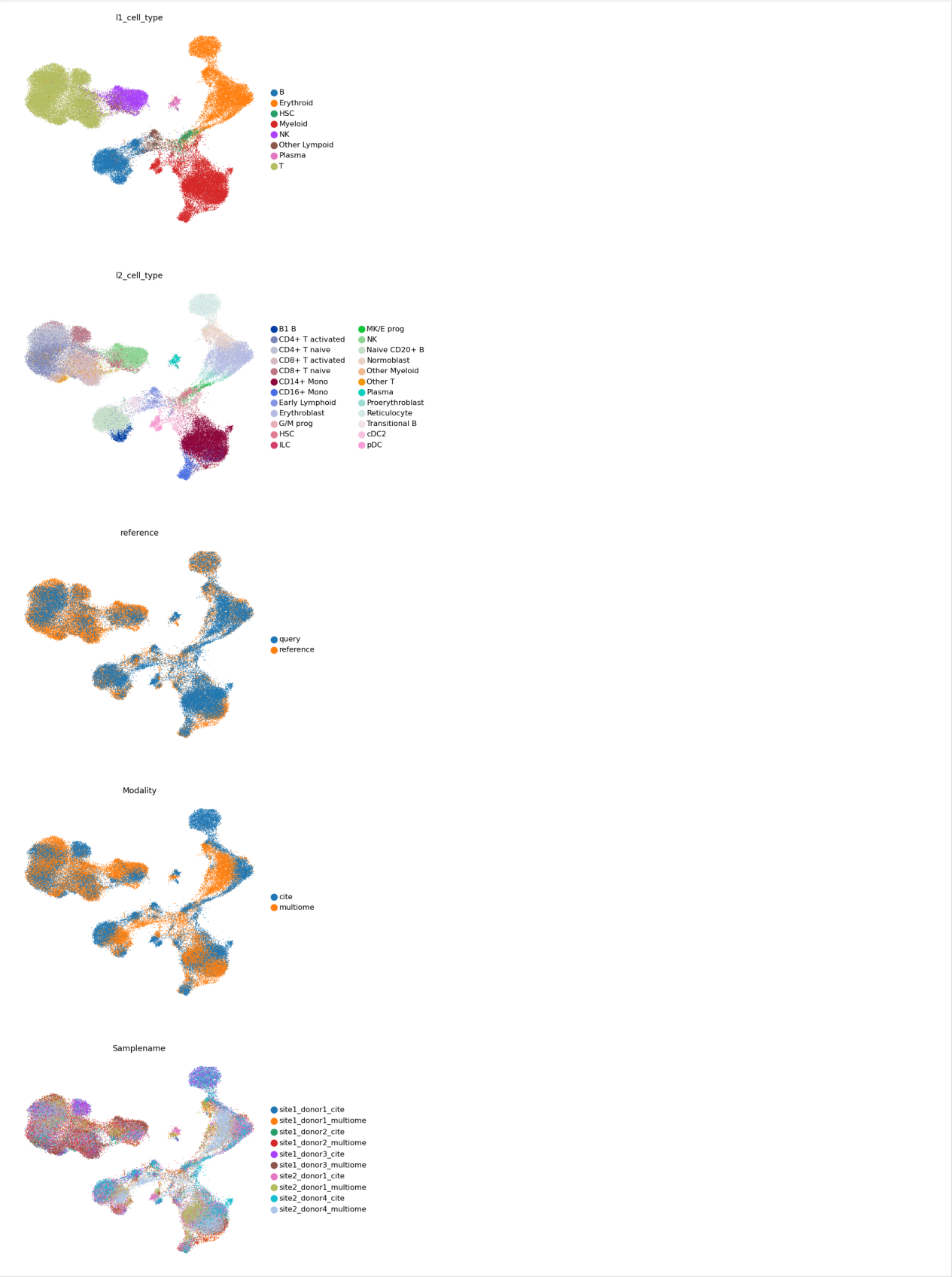
[29]:
```
adata.obs['reference'] = 'reference'
query.obs['reference'] = 'query'

adata.obs['type_of_query'] = 'reference'
query.obs.loc[idx_atac_query, 'type_of_query'] = 'ATAC query'
query.obs.loc[idx_scrna_query, 'type_of_query'] = 'scRNA query'
query.obs.loc[idx_mutiome_query, 'type_of_query'] = 'multiome query'
query.obs.loc[idx_cite_query, 'type_of_query'] = 'CITE-seq query'
```

Finally, we concatentae the reference and the query, and visualize both on a UMAP.

[30]:
```
adata_both = ad.concat([adata, query])
```

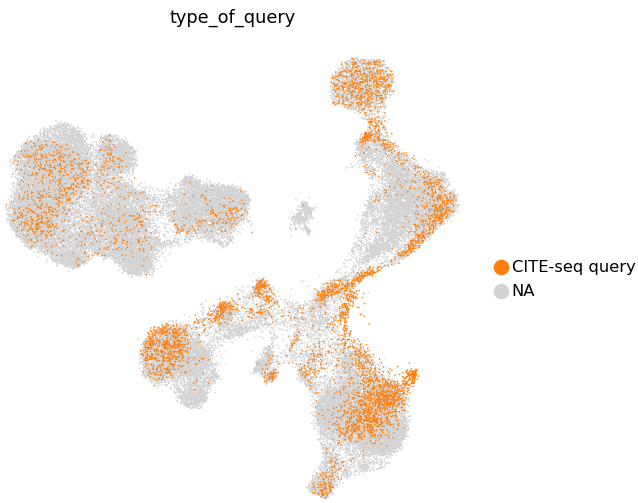
[31]:
```
sc.pp.neighbors(adata_both, use_rep='latent')
sc.tl.umap(adata_both)
```

[32]:
```
sc.pl.umap(adata_both, color=['l1_cell_type', 'l2_cell_type', 'reference', 'Modality',
↪'Samplename'], ncols=1, frameon=False)
```

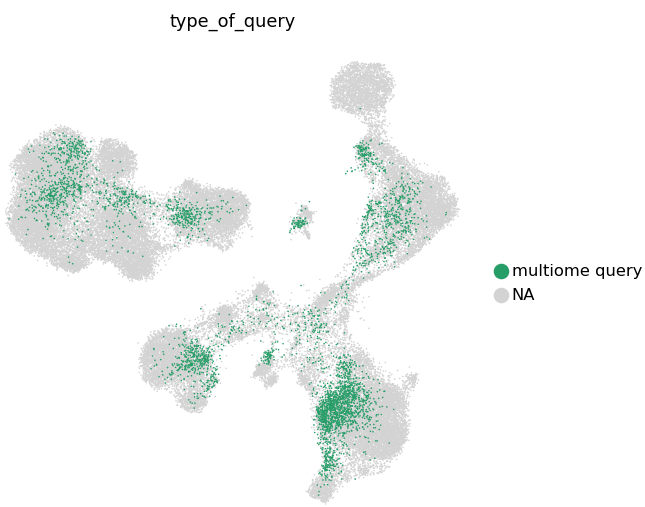

l1_cell_type
B
Erythroid
HSC
Myeloid
NK
Other Lympoid
Plasma
T
l2_cell_type
B1 B
CD4+ T activated
CD4+ T naive
CD8+ T activated
CD8+ T naive
CD14+ Mono
CD16+ Mono
Early Lymphoid
Erythroblast
G/M prog
HSC
ILC
MK/E prog
NK
Naive CD20+ B
Normoblast
Other Myeloid
Other T
Plasma
Proerythroblast
Reticulocyte
Transitional B
cDC2
pDC
reference
query
reference
Modality
cite
multiome
Samplename
site1_donor1_cite
site1_donor1_multiome
site1_donor2_cite
site1_donor2_multiome
site1_donor3_cite
site1_donor3_multiome
site2_donor1_cite
site2_donor1_multiome
site2_donor4_cite
site2_donor4_multiome

We also can take a look at separate multimodal and unimodal queries.

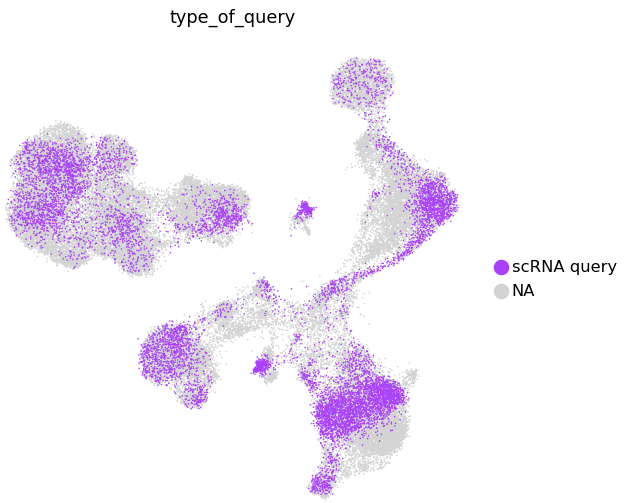
[33]:
```
sc.pl.umap(
    adata_both,
    color='type_of_query',
    ncols=1,
    frameon=False,
    groups=['CITE-seq query']
)
```



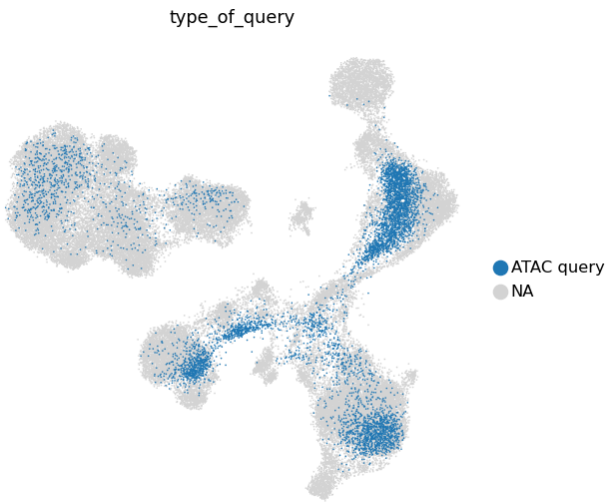
[34]:
```
sc.pl.umap(
    adata_both,
    color='type_of_query',
    ncols=1,
    frameon=False,
    groups=['multiome query']
)
```



[35]:
```
sc.pl.umap(
    adata_both,
    color='type_of_query',
    ncols=1,
    frameon=False,
    groups=['scRNA query']
)
```



[36]:
```
sc.pl.umap(
    adata_both,
    color='type_of_query',
    ncols=1,
    frameon=False,
    groups=['ATAC query']
)
```



CHAPTER

EIGHTEEN

INTEGRATION, LABEL TRANSFER AND MULTI-SCALE ANALYSIS WITH SCPOLI

In this notebook we demonstrate an example workflow of data integration, reference mapping, label transfer and multi-scale analysis of sample and cell embeddings using scPoli. We integrate pancreas data obtained from the scArches reproducibility repository. The data can be downloaded from figshare.

[1]:
```
import os
import numpy as np
import scanpy as sc
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report
from scarches.models.scpoli import scPoli

import warnings
warnings.filterwarnings('ignore')
%load_ext autoreload
%autoreload 2
```

```
WARNING:root:In order to use the mouse gastrulation seqFISH datsets, please install␣
↪squidpy (see https://github.com/scverse/squidpy).
INFO:pytorch_lightning.utilities.seed:Global seed set to 0
/home/icb/carlo.dedonno/anaconda3/envs/scarches/lib/python3.10/site-packages/pytorch_
↪lightning/utilities/warnings.py:53: LightningDeprecationWarning: pytorch_lightning.
↪utilities.warnings.rank_zero_deprecation has been deprecated in v1.6 and will be␣
↪removed in v1.8. Use the equivalent function from the pytorch_lightning.utilities.rank_
↪zero module instead.
  new_rank_zero_deprecation(
/home/icb/carlo.dedonno/anaconda3/envs/scarches/lib/python3.10/site-packages/pytorch_
↪lightning/utilities/warnings.py:58: LightningDeprecationWarning: The `pytorch_
↪lightning.loggers.base.rank_zero_experiment` is deprecated in v1.7 and will be removed␣
↪in v1.9. Please use `pytorch_lightning.loggers.logger.rank_zero_experiment` instead.
  return new_rank_zero_deprecation(*args, **kwargs)
WARNING:root:In order to use sagenet models, please install pytorch geometric (see https:
↪//pytorch-geometric.readthedocs.io) and
 captum (see https://github.com/pytorch/captum).
WARNING:root:mvTCR is not installed. To use mvTCR models, please install it first using
↪"pip install mvtcr"
WARNING:root:multigrate is not installed. To use multigrate models, please install it␣
↪first using "pip install multigrate".
```

[2]:
```
sc.settings.set_figure_params(dpi=100, frameon=False)
sc.set_figure_params(dpi=100)
sc.set_figure_params(figsize=(3, 3))
plt.rcParams['figure.dpi'] = 100
plt.rcParams['figure.figsize'] = (3, 3)
```

[3]:
```
adata = sc.read('../../../lataq_reproduce/data/pancreas.h5ad')
adata
```

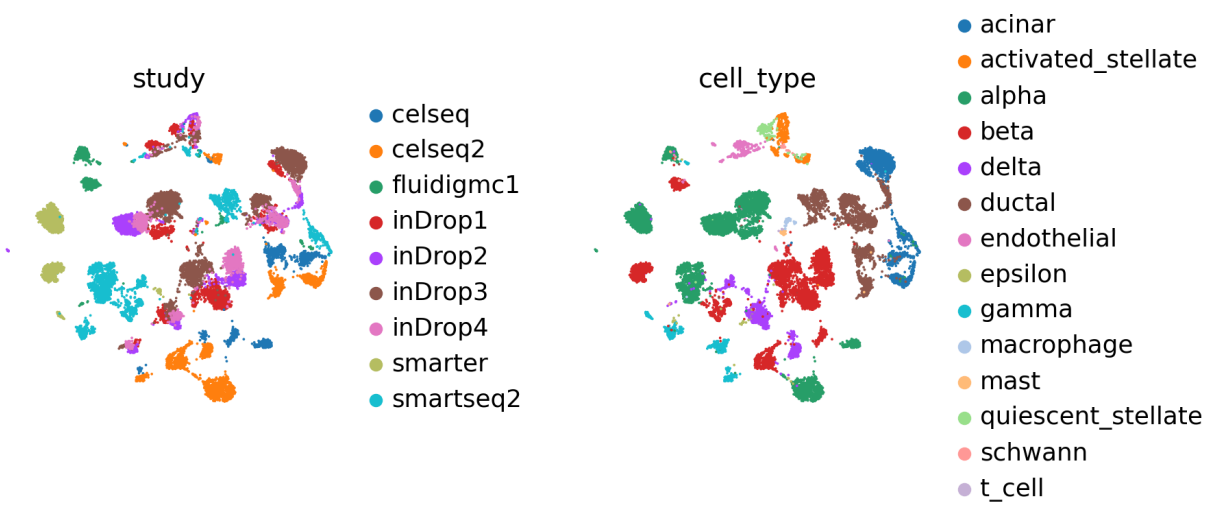
[3]:
```
AnnData object with n_obs × n_vars = 16382 × 4000
    obs: 'study', 'cell_type', 'pred_label', 'pred_score'
    obsm: 'X_seurat', 'X_symphony'
```

[4]:
```
sc.pp.neighbors(adata)
sc.tl.umap(adata)
```

```
WARNING: You're trying to run this on 4000 dimensions of `.X`, if you really want this,␣
↪set `use_rep='X'`.
         Falling back to preprocessing with `sc.pp.pca` and default params.
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels␣
↪instead.
```

[5]:
```
sc.pl.umap(adata, color=['study', 'cell_type'], wspace=0.5, frameon=False)
```



[6]:
```
early_stopping_kwargs = {
    "early_stopping_metric": "val_prototype_loss",
    "mode": "min",
    "threshold": 0,
    "patience": 20,
    "reduce_lr": True,
    "lr_patience": 13,
    "lr_factor": 0.1,
}

condition_key = 'study'
```

(continues on next page)

(continued from previous page)

```
cell_type_key = 'cell_type'
reference = [
    'inDrop1',
    'inDrop2',
    'inDrop3',
    'inDrop4',
    'fluidigmc1',
    'smartseq2',
    'smarter'
]
query = ['celseq', 'celseq2']
```

18.1 Reference - query split

We split our data in a group of reference datasets to be used for reference building, and a group of query datasets that we will map.

In order to simulate an unknown cell type scenario, we manually remove beta cells from the reference.

[7]:
```
adata.obs['query'] = adata.obs[condition_key].isin(query)
adata.obs['query'] = adata.obs['query'].astype('category')
source_adata = adata[adata.obs.study.isin(reference)].copy()
source_adata = source_adata[~source_adata.obs.cell_type.str.contains('alpha')].copy()
target_adata = adata[adata.obs.study.isin(query)].copy()
```

[8]:
```
source_adata, target_adata
```

[8]:
```
(AnnData object with n_obs × n_vars = 8634 × 4000
     obs: 'study', 'cell_type', 'pred_label', 'pred_score', 'query'
     uns: 'neighbors', 'umap', 'study_colors', 'cell_type_colors'
     obsm: 'X_seurat', 'X_symphony', 'X_pca', 'X_umap'
     obsp: 'distances', 'connectivities',
 AnnData object with n_obs × n_vars = 3289 × 4000
     obs: 'study', 'cell_type', 'pred_label', 'pred_score', 'query'
     uns: 'neighbors', 'umap', 'study_colors', 'cell_type_colors'
     obsm: 'X_seurat', 'X_symphony', 'X_pca', 'X_umap'
     obsp: 'distances', 'connectivities')
```

18.2 Train reference scPoli model on fully labeled reference data

Explanation of scPoli parameters:

Model: - **condition_keys**: obs column names of the covariate(s) you want to use for integration, if a list of names is passed, the model will use independent embeddings for each covariate - **cell_type_keys**: obs column names of the cell type annotation(s) to use for prototype learning, if a list is passed, the model will compute the prototype loss in parallel for each set of annotations passed - **embedding_dims**: embedding dimensionality, if an integer is passed, the model will use embeddings of the same dimensionality for each covariate, if the user wishes to define different dimensionalities for each covariate, a list needs to be provided

Training: - **pretraining_epochs**: number of epochs for which the model is trained in an unsupervised fashion - **n_epochs**: total number of training epochs, finetuning epochs therefore will be (n_epochs - pretraining_epochs) - **eta**: weight of the prototype loss - **prototype_training**: flag that can be used to turn off prototype training - **unlabeled_prototype_training**: flag that can be used to skip unlabeled prototype computation. This step involves Louvain clustering and can be time-consuming on big datasets. Unlabeled prototypes can be useful for downstream analyses but are not used at training time.

[9]:
```
scpoli_model = scPoli(
    adata=source_adata,
    condition_keys=condition_key,
    cell_type_keys=cell_type_key,
    embedding_dims=5,
    recon_loss='nb',
)
scpoli_model.train(
    n_epochs=50,
    pretraining_epochs=40,
    early_stopping_kwargs=early_stopping_kwargs,
    eta=5,
)
```

```
Embedding dictionary:
        Num conditions: [7]
        Embedding dim: [5]
Encoder Architecture:
        Input Layer in, out and cond: 4000 64 5
        Mean/Var Layer in/out: 64 10
Decoder Architecture:
        First Layer in, out and cond:  10 64 5
        Output Layer in/out:  64 4000

Initializing dataloaders
Starting training
 || 100.0%  - val_loss: 1091.81 - val_cvae_loss: 1079.83 - val_prototype_loss:   11.98 -␣
↪val_labeled_loss:    2.40
```

We recommend using a **pretraining/training epoch ratio** of approximately 80 or 90%. If you train for more total epochs you should use a higher ratio, whereas if you're training for only a few epochs, this ratio can be smaller. If the model is trained withthe prototype loss for too many epochs it can lead to very concentrated clusters in latent space.

18.3 Reference mapping of unlabeled query datasets

[10]:
```
scpoli_query = scPoli.load_query_data(
    adata=target_adata,
    reference_model=scpoli_model,
    labeled_indices=[],
)
```

```
Embedding dictionary:
        Num conditions: [9]
        Embedding dim: [5]
Encoder Architecture:
        Input Layer in, out and cond: 4000 64 5
```

(continues on next page)

(continued from previous page)

```
        Mean/Var Layer in/out: 64 10
Decoder Architecture:
        First Layer in, out and cond:  10 64 5
        Output Layer in/out:  64 4000
```

[11]:
```
scpoli_query.train(
    n_epochs=50,
    pretraining_epochs=40,
    eta=10
)
```

```
Warning: Labels in adata.obs[cell_type] is not a subset of label-encoder!
The missing labels are: {'alpha'}
Therefore integer value of those labels is set to -1
Warning: Labels in adata.obs[cell_type] is not a subset of label-encoder!
The missing labels are: {'alpha'}
Therefore integer value of those labels is set to -1
Initializing dataloaders
Starting training
 |----| 80.0%  - val_loss: 1766.09 - val_cvae_loss: 1766.09
Initializing unlabeled prototypes with Leiden with an unknown number of  clusters.
Clustering succesful. Found 21 clusters.
 || 100.0%  - val_loss: 1760.48 - val_cvae_loss: 1760.48 - val_prototype_loss:    0.00 -␣
↪val_unlabeled_loss:    0.57
```

18.4 Label transfer from reference to query

The uncertainties returned by the model consist of the distance between the cell in the latent space and the labeled prototypes closest to it. This distance does not have an upper bound, and the scale of the distance can provide information on the heterogeneity of the dataset. We also offer the option to scale the uncertainties between 0 and 1.

[14]:
```
results_dict = scpoli_query.classify(target_adata, scale_uncertainties=True)
```

Let's check the label transfer performance we achieved.

[18]:
```
for i in range(len(cell_type_key)):
    preds = results_dict[cell_type_key]["preds"]
    results_dict[cell_type_key]["uncert"]
    classification_df = pd.DataFrame(
        classification_report(
            y_true=target_adata.obs[cell_type_key],
            y_pred=preds,
            output_dict=True,
        )
    ).transpose()
print(classification_df)
```

```
                    precision    recall  f1-score      support
acinar               0.964497  0.974104  0.969277   502.000000
activated_stellate   0.900826  1.000000  0.947826   109.000000
```

(continues on next page)

(continued from previous page)

```
alpha                 0.000000  0.000000  0.000000  1034.000000
beta                  0.966019  0.985149  0.975490   606.000000
delta                 0.547253  0.984190  0.703390   253.000000
ductal                0.969178  0.967521  0.968349   585.000000
endothelial           1.000000  1.000000  1.000000    26.000000
epsilon               0.067568  1.000000  0.126582     5.000000
gamma                 0.147465  1.000000  0.257028   128.000000
macrophage            1.000000  0.937500  0.967742    16.000000
mast                  1.000000  0.714286  0.833333     7.000000
quiescent_stellate    1.000000  0.615385  0.761905    13.000000
schwann               1.000000  1.000000  1.000000     5.000000
t_cell                0.000000  0.000000  0.000000     0.000000
accuracy              0.669504  0.669504  0.669504     0.669504
macro avg             0.683058  0.798438  0.679352  3289.000000
weighted avg          0.595747  0.669504  0.614543  3289.000000
```

[19]:
```
#get latent representation of reference data
scpoli_query.model.eval()
data_latent_source = scpoli_query.get_latent(
    source_adata,
    mean=True
)

adata_latent_source = sc.AnnData(data_latent_source)
adata_latent_source.obs = source_adata.obs.copy()

#get latent representation of query data
data_latent= scpoli_query.get_latent(
    target_adata,
    mean=True
)

adata_latent = sc.AnnData(data_latent)
adata_latent.obs = target_adata.obs.copy()

#get label annotations
adata_latent.obs['cell_type_pred'] = results_dict['cell_type']['preds'].tolist()
adata_latent.obs['cell_type_uncert'] = results_dict['cell_type']['uncert'].tolist()
adata_latent.obs['classifier_outcome'] = (
    adata_latent.obs['cell_type_pred'] == adata_latent.obs['cell_type']
)

#get prototypes
labeled_prototypes = scpoli_query.get_prototypes_info()
labeled_prototypes.obs['study'] = 'labeled prototype'
unlabeled_prototypes = scpoli_query.get_prototypes_info(prototype_set='unlabeled')
unlabeled_prototypes.obs['study'] = 'unlabeled prototype'

#join adatas
adata_latent_full = adata_latent_source.concatenate(
    [adata_latent, labeled_prototypes, unlabeled_prototypes],
    batch_key='query'
```

(continues on next page)

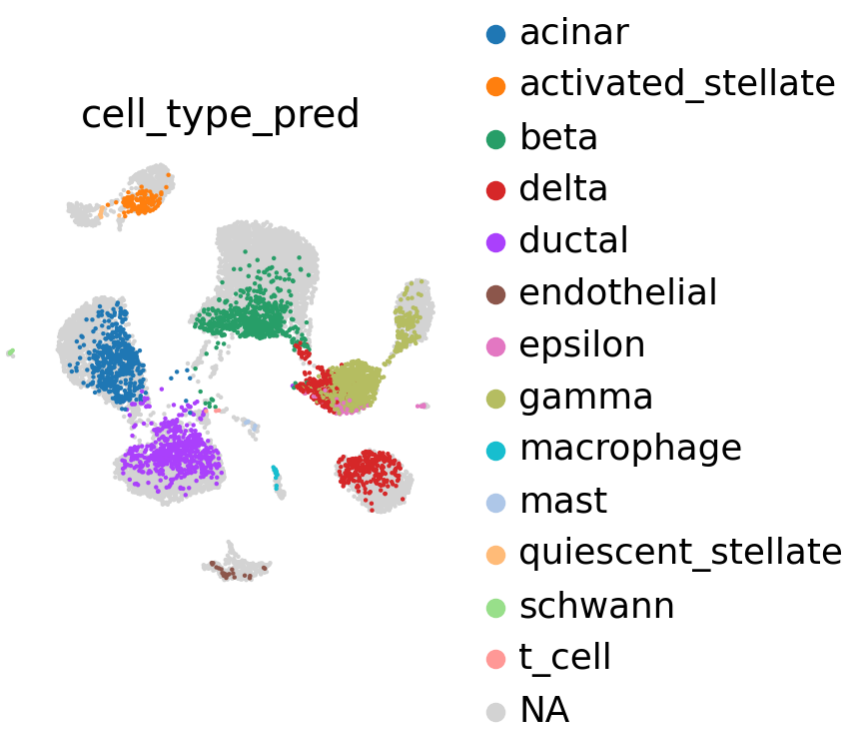
(continued from previous page)

```
)
adata_latent_full.obs['cell_type_pred'][adata_latent_full.obs['query'].isin(['0'])] = np.
↪nan
sc.pp.neighbors(adata_latent_full, n_neighbors=15)
sc.tl.umap(adata_latent_full)
```

[20]:
```
#get adata without prototypes
adata_no_prototypes = adata_latent_full[adata_latent_full.obs['query'].isin(['0', '1'])]
```

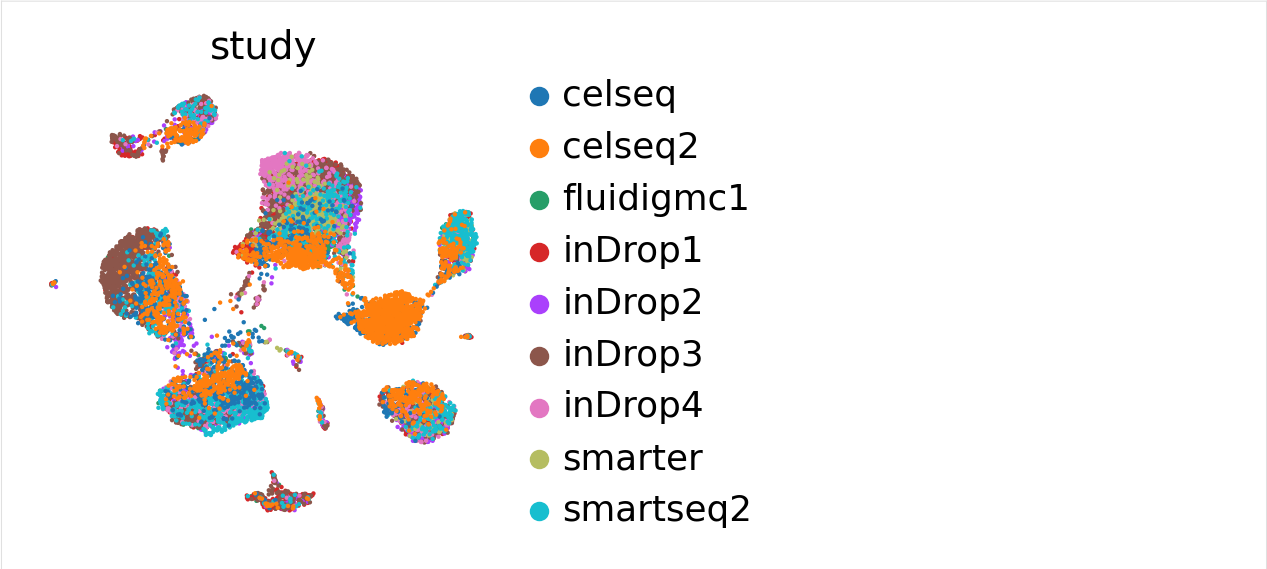
[21]:
```
sc.pl.umap(
    adata_no_prototypes,
    color='cell_type_pred',
    show=False,
    frameon=False,
)
```

[21]:
```
<AxesSubplot: title={'center': 'cell_type_pred'}, xlabel='UMAP1', ylabel='UMAP2'>
```



[22]:
```
sc.pl.umap(
    adata_no_prototypes,
    color='study',
    show=False,
    frameon=False,
)
```

[22]:
```
<AxesSubplot: title={'center': 'study'}, xlabel='UMAP1', ylabel='UMAP2'>
```

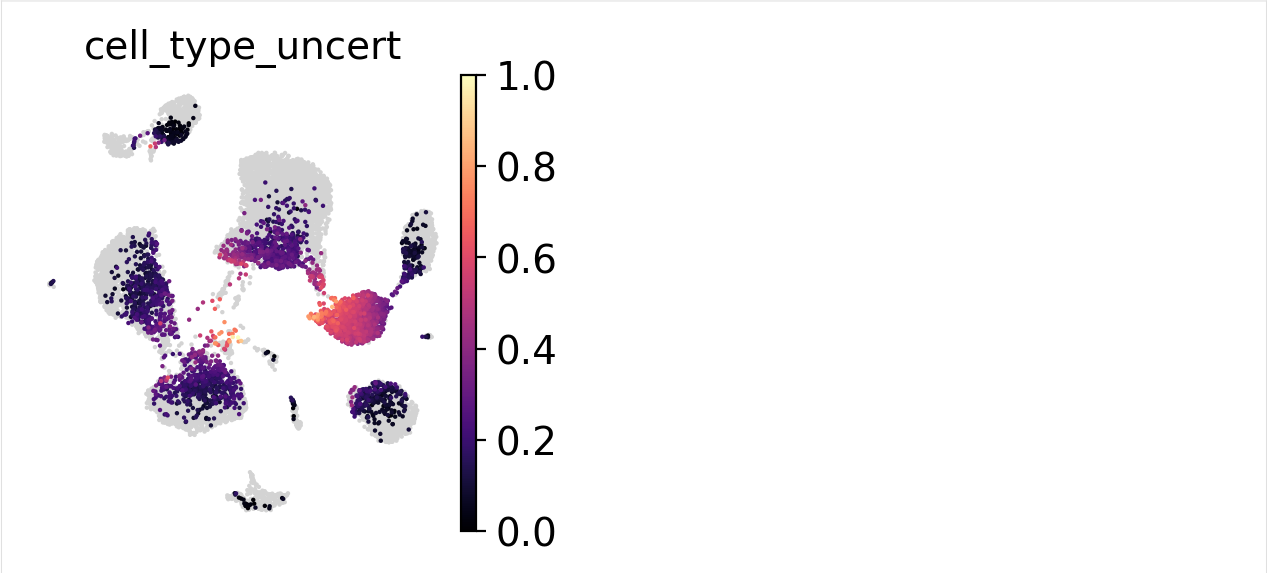


18.5 Inspect uncertainty

We can look at the uncertainty of each prediction and either select a threshold after visual inspection or by looking at the percentiles of the uncertainties distribution.

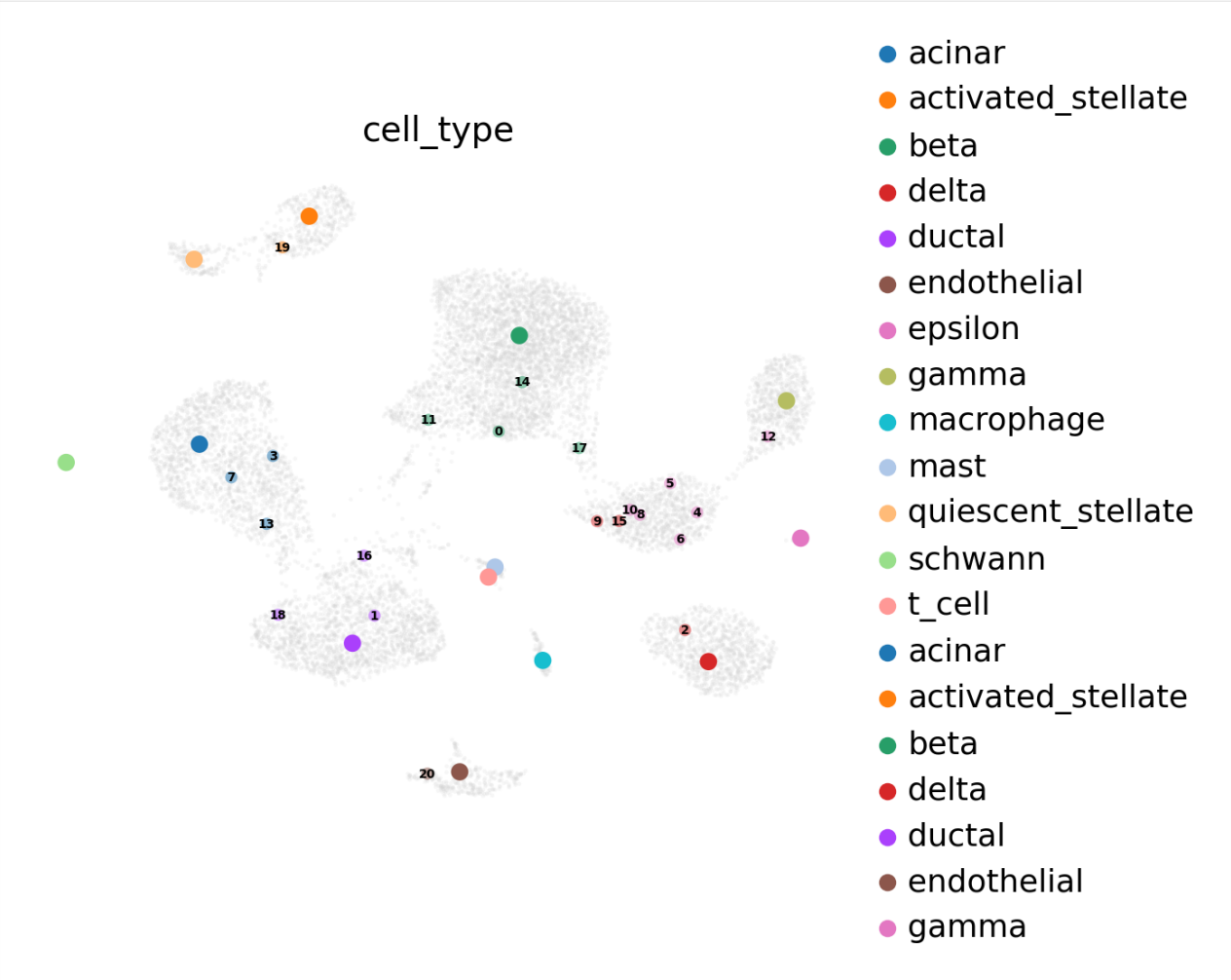
[23]:
```
sc.pl.umap(
    adata_no_prototypes,
    color='cell_type_uncert',
    show=False,
    frameon=False,
    cmap='magma',
    vmax=1
)
```

[23]:
```
<AxesSubplot: title={'center': 'cell_type_uncert'}, xlabel='UMAP1', ylabel='UMAP2'>
```



18.6 Inspect prototypes

[25]:
```
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
adata_labeled_prototypes = adata_latent_full[adata_latent_full.obs['query'].isin(['2'])]
adata_unlabeled_prototypes = adata_latent_full[adata_latent_full.obs['query'].isin(['3
↪'])]
adata_labeled_prototypes.obs['cell_type_pred'] = adata_labeled_prototypes.obs['cell_type_
↪pred'].astype('category')
adata_unlabeled_prototypes.obs['cell_type_pred'] = adata_unlabeled_prototypes.obs['cell_
↪type_pred'].astype('category')
adata_unlabeled_prototypes.obs['cell_type'] = adata_unlabeled_prototypes.obs['cell_type
↪'].astype('category')

sc.pl.umap(
    adata_no_prototypes,
    alpha=0.2,
    show=False,
    ax=ax
)
ax.legend([])
# plot labeled prototypes
sc.pl.umap(
    adata_labeled_prototypes,
    size=200,
    color=f'{cell_type_key}_pred',
    ax=ax,
    show=False,
    frameon=False,
)
# plot labeled prototypes
sc.pl.umap(
    adata_unlabeled_prototypes,
    size=100,
    color=[cell_type_key + '_pred'],
    palette=adata_labeled_prototypes.uns['cell_type_pred_colors'],
    ax=ax,
    show=False,
    frameon=False,
    alpha=0.5,
)
sc.pl.umap(
    adata_unlabeled_prototypes,
    size=0,
    color=cell_type_key,
    frameon=False,
    ax=ax,
    legend_loc='on data',
    legend_fontsize=5,
)
ax.set_title('Landmarks')
fig.tight_layout()
```



After inspecting the prototypes we can observe that unlabeled prototype 4, 5, 7, 8, 11 and 13 fall into the region of high uncertainty. With this knowledge, we can add a new labeled prototype.

[26]:
```
scpoli_query.add_new_cell_type(
    "alpha",
    cell_type_key,
    [4, 5, 6, 8, 9, 10, 15]
)
```

[27]:
```
results_dict = scpoli_query.classify(target_adata)
```

[28]:
```
#get latent representation of reference data
scpoli_query.model.eval()
data_latent_source = scpoli_query.get_latent(
    source_adata,
    mean=True
)

adata_latent_source = sc.AnnData(data_latent_source)
adata_latent_source.obs = source_adata.obs.copy()
```

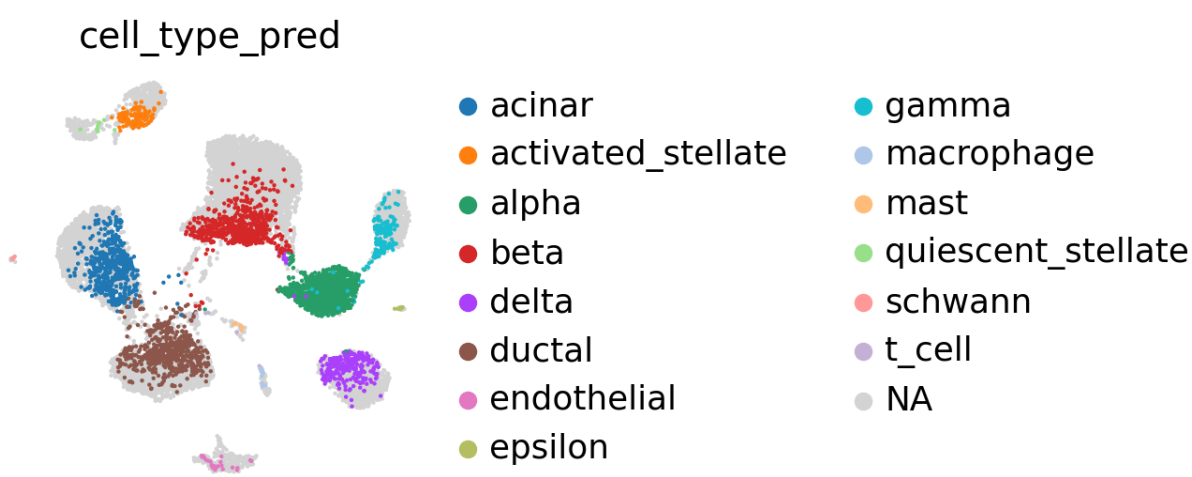
(continues on next page)

(continued from previous page)

```
#get latent representation of query data
data_latent= scpoli_query.get_latent(
    target_adata,
    mean=True
)

adata_latent = sc.AnnData(data_latent)
adata_latent.obs = target_adata.obs.copy()

#get label annotations
adata_latent.obs['cell_type_pred'] = results_dict['cell_type']['preds'].tolist()
adata_latent.obs['cell_type_uncert'] = results_dict['cell_type']['uncert'].tolist()
adata_latent.obs['classifier_outcome'] = (
    adata_latent.obs['cell_type_pred'] == adata_latent.obs['cell_type']
)

#join adatas
adata_latent_full = adata_latent_source.concatenate(
    [adata_latent, labeled_prototypes, unlabeled_prototypes],
    batch_key='query'
)
adata_latent_full.obs['cell_type_pred'][adata_latent_full.obs['query'].isin(['0'])] = np.
↪nan
sc.pp.neighbors(adata_latent_full, n_neighbors=15)
sc.tl.umap(adata_latent_full)
```

[29]:
```
sc.pl.umap(
    adata_latent_full,
    color='cell_type_pred',
    show=False,
    frameon=False,
)
```

[29]:
```
<AxesSubplot: title={'center': 'cell_type_pred'}, xlabel='UMAP1', ylabel='UMAP2'>
```



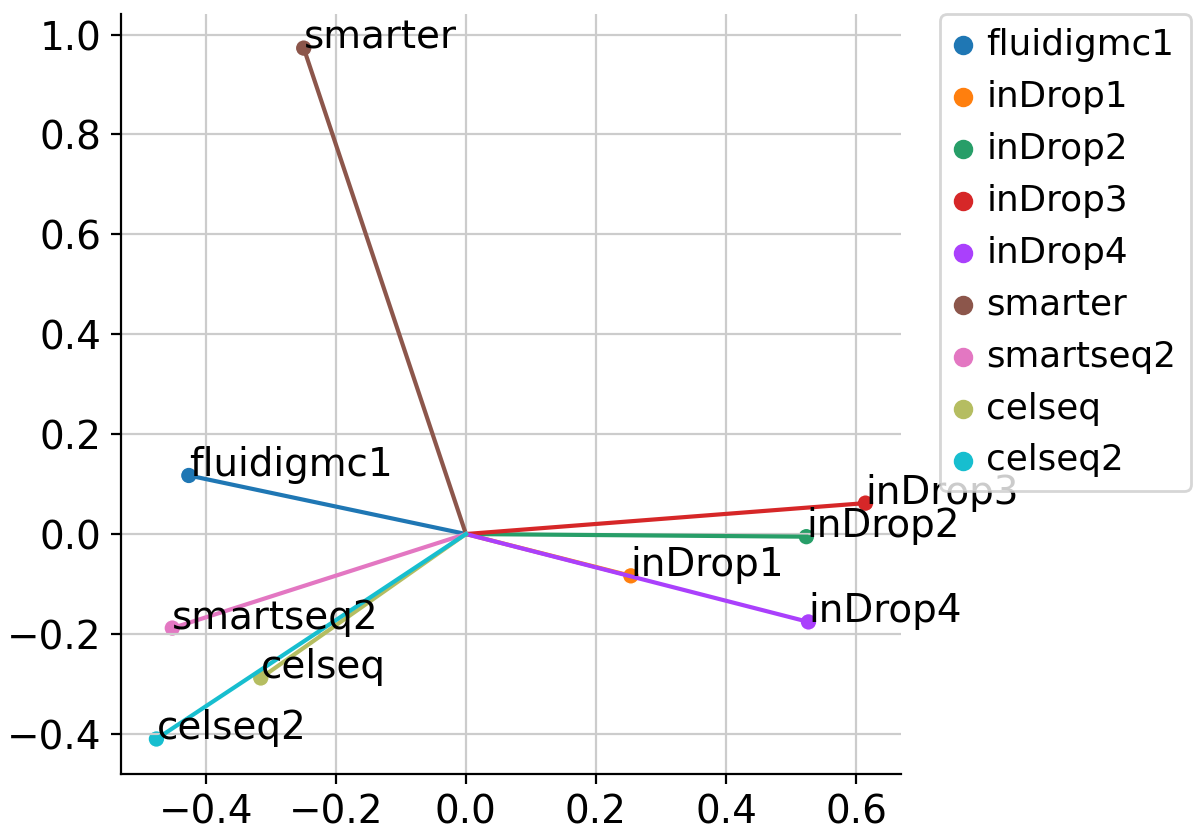
We can now see that the alpha cell cluster is correctly classified.

18.7 Sample embeddings

We can extract the conditional embeddings learnt by scPoli and analyse them.

[30]:
```
adata_emb = scpoli_query.get_conditional_embeddings()
```

[31]:
```
from sklearn.decomposition import KernelPCA
pca = KernelPCA(n_components=2, kernel='linear')
emb_pca = pca.fit_transform(adata_emb.X)
conditions = scpoli_query.conditions_['study']
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
sns.scatterplot(x=emb_pca[:, 0], y=emb_pca[:, 1], hue=conditions, ax=ax)
ax.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
for i, c in enumerate(conditions):
    ax.plot([0, emb_pca[i, 0]], [0, emb_pca[i, 1]])
    ax.text(emb_pca[i, 0], emb_pca[i, 1], c)
sns.despine()
```



CHAPTER

NINETEEN

MAPPING DATA TO THE HUMAN LUNG CELL ATLAS FOR JOINT ANALYSIS AND CELL TYPE LABEL TRANSFER

In this tutorial we will show how to map single-cell data to a reference. Such mapping yields a joint embedding of the reference and the mapped ("query") data, which can be used for downstream analysis including clustering and trajectory analysis. Moreover, once the data is mapped, it is possible to perform label transfer from the reference to the query, thus providing a first annotation of all cells in the query data with minimial effort. We will use the Human Lung Cell Atlas as an example reference. We will furthermore use the scANVI model previously generated to integrate the datasets in the HLCA as our base reference model, on which we will perform scArches surgery to enable mapping and batch correction of the query dataset. Finally, we use a KNN classifier from scArches to perform cell type label transfer from the reference to the query.

As the HLCA was annotated at five different levels (coarse to fine annotations), we will also display five different levels of cell annotation and their uncertainties for our mapping.

This notebook was compiled by Lisa Sikkema.

Note: the label transfer section of this notebook shows a bug in some environments with older scvi-tools and scanpy. We haven't figured out yet what exactly causes this. If you observe random label assignment during label transfer, make sure to update your packages and re-run. The bug has not been observed in environments with `scanpy>=1.9.3`, `scarches>=0.5.8` and `scvi-tools>=0.20.3`.

19.1 Setup

19.1.1 Import libraries and set figure parameters and paths

[1]:
```
import os

import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=DeprecationWarning)
warnings.simplefilter(action="ignore", category=UserWarning)
```

[2]:
```
import sys
import scanpy as sc
import numpy as np
import pandas as pd
import scarches as sca
import anndata as ad
from scipy import sparse
```

(continues on next page)

(continued from previous page)

```
import gdown
import gzip
import shutil
import urllib.request
```

```
WARNING:root:In order to use the mouse gastrulation seqFISH datsets, please install␣
↪squidpy (see https://github.com/scverse/squidpy).
WARNING:root:In order to use sagenet models, please install pytorch geometric (see https:
↪//pytorch-geometric.readthedocs.io) and
 captum (see https://github.com/pytorch/captum).
INFO:lightning_fabric.utilities.seed:[rank: 0] Global seed set to 0
/home/icb/lisa.sikkema/miniconda3/envs/HLCA_mapping_env_new_upgrade_scanpy/lib/python3.8/
↪site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter␣
↪and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
WARNING:root:mvTCR is not installed. To use mvTCR models, please install it first using
↪"pip install mvtcr"
WARNING:root:multigrate is not installed. To use multigrate models, please install it␣
↪first using "pip install multigrate".
```

Set scanpy figure parameters:

[3]:
```
sc.settings.set_figure_params(dpi=200, frameon=False)
sc.set_figure_params(dpi=200)
sc.set_figure_params(figsize=(4, 4))
```

Set paths:

[4]:
```
ref_model_dir_prefix = "."  # directory in which to store the reference model directory
surgery_model_dir_prefix = (
    "."  # directory in which to store the surgery model directory
)
path_reference_emb = (
    "./HLCA_emb_and_metadata.h5ad"  # path to reference embedding to be created
)
path_query_data = "./HLCA_query.h5ad"  # input test query data
# don't change the following paths:
ref_model_dir = os.path.join(
    ref_model_dir_prefix, "HLCA_reference_model"
)  # don't change this
surgery_model_dir = os.path.join(
    surgery_model_dir_prefix, "surgery_model"
)  # don't change this
```

19.1.2 Download reference data and reference model:

We will start with downloading the needed model and data. First, let's download the reference model, on which we will perform surgery. The HLCA reference model can be found on Zenodo, and we'll download it below:

[6]:
```
url = "https://zenodo.org/record/7599104/files/HLCA_reference_model.zip"
output = "HLCA_reference_model.zip"
gdown.download(url, output, quiet=False)
shutil.unpack_archive("HLCA_reference_model.zip", extract_dir=ref_model_dir_prefix)
os.remove(output)
```

We will furthermore download the reference itself. As scArches allows the mapping of query data onto an existing reference *embedding*, we will only need to download the embedding of the HLCA reference. That saves a lot of time and memory compared to downloading the full count matrix. It might take a while to download this embedding, it's 2.3Gb in size.

[7]:
```
url = "https://zenodo.org/record/7599104/files/HLCA_full_v1.1_emb.h5ad"
output = path_reference_emb
gdown.download(url, output, quiet=False)
```

Let's load our downloaded reference embedding:

[8]:
```
adata_ref = sc.read_h5ad(path_reference_emb)
```

This embedding includes both the HLCA core (healthy reference, integrated with scANVI) and the HLCA extension (core + lung disease datasets mapped with scArches). We will subset to the HLCA core here, as this is what the reference model was trained on.

[9]:
```
# subset
adata_ref = adata_ref = adata_ref[adata_ref.obs.core_or_extension == "core", :].copy()
# remove all obs variables that have no entries anymore (i.e. obs columns that were only␣
↪relevant for the HLCA extension)
cols_to_drop = [
    col for col in adata_ref.obs.columns if adata_ref.obs[col].isnull().all()
]
adata_ref.obs.drop(columns=cols_to_drop, inplace=True)
```

19.1.3 Load query data and match feature naming with reference model:

Finally, we need to have query data which we want to map to the reference. For this tutorial, we will use an example dataset from the Delorey et al. publication as query dataset to map to the HLCA reference, but you can use your own data for this as well. For our example, we only use one sample (the fresh single-cell sample) from the Delorey dataset. We already prepared an .h5ad file for that sample in this repository, so no need to download this file, we can just load it:

[10]:
```
adata_query_unprep = sc.read_h5ad(path_query_data)
```

Now we prepare our query data, so that it includes the right genes (this depends on the genes used for the reference model, missing genes are padded with zeros) and has those genes in the right order. Preparing your query data ensures data corectness and smooth running of the scArches reference mapping.

First, make sure that your counts matrix is sparse for memory efficiency:

[11]:
```
adata_query_unprep.X = sparse.csr_matrix(adata_query_unprep.X)
```

Remove obsm and varm matrices to prevent errors downstream:

[12]:
```
del adata_query_unprep.obsm
del adata_query_unprep.varm
```

Note that the data should have raw counts and not normalized counts in adata.X. Let's do a quick check to see if we have integer data:

[13]:
```
adata_query_unprep.X[:10, :30].toarray()
```

[13]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,  0.,  0.,  0.,  0.,
         6., 46.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         1.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  1.,  0.,  0.]], dtype=float32)
```

Looks like we do. Now let's check if our reference model uses gene names or gene ids as input features. We will need to match our query data with the reference model.

[14]:
```
ref_model_features = pd.read_csv(
    os.path.join(ref_model_dir, "var_names.csv"), header=None
)
```

[15]:
```
ref_model_features.head(5)
```

[15]:
```
                 0
0  ENSG00000000938
```

(continues on next page)

(continued from previous page)

```
1  ENSG00000000971
2  ENSG00000002587
3  ENSG00000002933
4  ENSG00000003436
```

As you can see, the HLCA reference model requires ensemble IDs. Therefore, if your data includes ensembl IDs, we can proceed and use the standard scArches function to subset and pad our query AnnData. Make sure your `adata_query_unprep.var.index` contains the gene ids. If you instead only have gene names and no IDs for your query data, we will have to prepare your data manually (see below).

The test data already has ensembl ids as index:

[16]:
```
adata_query_unprep.var.head(5)
```

[16]:
```
                       gene_ids    feature_types  genome   gene_names
ENSG00000243485  ENSG00000243485  Gene Expression  GRCh38  MIR1302-2HG
ENSG00000237613  ENSG00000237613  Gene Expression  GRCh38      FAM138A
ENSG00000186092  ENSG00000186092  Gene Expression  GRCh38        OR4F5
ENSG00000238009  ENSG00000238009  Gene Expression  GRCh38   AL627309.1
ENSG00000239945  ENSG00000239945  Gene Expression  GRCh38   AL627309.3
```

If your query feature naming (ensembl IDs or gene symbols) does not match the reference model feature naming:

If your query feature naming does not match your reference model feature naming, you will need to add the right feature names. For the HLCA reference, the mapping of the 2000 input gene IDs to their gene names is stored on the HLCA Zenodo page, so you can add gene ids using that mapping. Alternatively, you can map your gene IDs to gene names (or the reverse) using BioMart mapping tables. In most cases your raw data includes both gene IDs and names, in which case mapping is not necessary.

Let's download the HLCA-specific gene mapping:

[17]:
```
# path_gene_mapping_df = os.path.join(ref_model_dir, "HLCA_reference_model_gene_order_
↪ids_and_symbols.csv")
```

[18]:
```
# # Download gene information from HLCA github:
# url = "https://zenodo.org/record/7599104/files/HLCA_reference_model_gene_order_ids_and_
↪symbols.csv"
# gdown.download(url, path_gene_mapping_df, quiet=False)
```

Load the mapping table:

[19]:
```
# gene_id_to_gene_name_df = pd.read_csv(path_gene_mapping_df, index_col=0)
```

Store your gene names in an adata.var.column if they are currently the index:

[20]:
```
# adata_query_unprep.var.head(2)
```

[21]:
```
## if gene names are in .var.index:
# adata_query_unprep.var["gene_names"] = adata_query_unprep.var.index
```

and then specify the gene name column name:

[22]:
```
# gene_name_column_name = "gene_names"
```

Map gene names to gene ids for all of the 2000 reference model genes that we can find in our data:

Check number of detected genes:

[23]:
```
# n_overlap = (
#     adata_query_unprep.var[gene_name_column_name]
#     .isin(gene_id_to_gene_name_df.gene_symbol)
#     .sum()
# )
# n_genes_model = gene_id_to_gene_name_df.shape[0]
# print(
#     f"Number of model input genes detected: {n_overlap} out of {n_genes_model} (
↪{round(n_overlap/n_genes_model*100)}%)"
# )
```

Subset query data to only the genes that are part of the modeling input, then map gene names to gene ids using the table above. Store the resulting ids both in the .var.index (for scArches) and in a .var[gene_ids] (for merging duplicate genes).

[24]:
```
# adata_query_unprep = adata_query_unprep[
#     :,
#     adata_query_unprep.var[gene_name_column_name].isin(
#         gene_id_to_gene_name_df.gene_symbol
#     ),
# ].copy()  # subset your data to genes used in the reference model
# adata_query_unprep.var.index = adata_query_unprep.var[gene_name_column_name].map(
#     dict(zip(gene_id_to_gene_name_df.gene_symbol, gene_id_to_gene_name_df.index))
# )  # add gene ids for the gene names, and store in .var.index
# # remove index name to prevent bugs later on
# adata_query_unprep.var.index.name = None
# adata_query_unprep.var["gene_ids"] = adata_query_unprep.var.index
```

Check that we now have ensembl IDs in .var.index:

[25]:
```
# adata_query_unprep.var.head(3)
```

Sum any columns with identical gene IDs that have resulted from the mapping. Here we define a short function to do that easily.

[26]:
```
# def sum_by(adata: ad.AnnData, col: str) -> ad.AnnData:
#     adata.strings_to_categoricals()
#     assert pd.api.types.is_categorical_dtype(adata.obs[col])

#     cat = adata.obs[col].values
#     indicator = sparse.coo_matrix(
#         (np.broadcast_to(True, adata.n_obs), (cat.codes, np.arange(adata.n_obs))),
#         shape=(len(cat.categories), adata.n_obs),
#     )

#     return ad.AnnData(
#         indicator @ adata.X, var=adata.var, obs=pd.DataFrame(index=cat.categories)
#     )
```

shape before merging:

[27]:
```
# adata_query_unprep.shape
```

Now merge. Note that all var columns will be dropped after merging (as we don't specify how to merge). As the merging is done based on .obs indices in the function above, we transpose our anndata object and re-transpose it after merging.

[28]:
```
# adata_query_unprep = sum_by(adata_query_unprep.transpose(), col="gene_ids").transpose()
```

Check the final shape of your query data:

[29]:
```
# adata_query_unprep.shape
```

add back gene ids:

[30]:
```
# gene_id_to_gene_name_df
```

[31]:
```
# adata_query_unprep.var = adata_query_unprep.var.join(gene_id_to_gene_name_df).
↪rename(columns={"gene_symbol":"gene_names"})
```

19.1.4 Prepare query data for scArches:

Now that we have the right feature naming in our query anndata, we can finalize preparation of our query AnnData object for mapping. This includes padding of missing genes (setting them to 0).

[32]:
```
adata_query = sca.models.SCANVI.prepare_query_anndata(
    adata=adata_query_unprep, reference_model=ref_model_dir, inplace=False
)
```

```
INFO     File ./HLCA_reference_model/model.pt already downloaded
INFO     Found 99.65% reference vars in query data.
```

Your query adata will now have the same number of genes as the number of model input features:

[33]:
```
adata_query
```

[33]:
```
AnnData object with n_obs × n_vars = 1786 × 2000
    obs: 'dataset'
    var: 'gene_ids', 'feature_types', 'genome', 'gene_names'
```

You'll see that scarches printed the percentage of model input features ("reference vars") that it could find in the query data. If this number is too low, it will affect the quality of your mapping. For example, we noticed low-quality mapping in a dataset for which we had only 1300 genes out of 2000 (65%).

19.1.5 Load reference model and set relevant query variables:

Let's load our reference model, on which we will perform surgery:

[34]:
```
surgery_model = sca.models.SCANVI.load_query_data(
    adata_query,
    ref_model_dir,
    freeze_dropout=True,
)
```

```
INFO     File ./HLCA_reference_model/model.pt already downloaded
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_
↪LEVEL=0 and rerun for more info.)
```

The reference model will have a number of variables that you will need to set as well. Let's check the variables for our reference model:

[35]:
```
surgery_model.registry_["setup_args"]
```

[35]:
```
{'labels_key': 'scanvi_label',
 'unlabeled_category': 'unlabeled',
 'layer': None,
 'batch_key': 'dataset',
 'size_factor_key': None,
 'categorical_covariate_keys': None,
 'continuous_covariate_keys': None}
```

There are three setup arguments that were used for building the reference model, and that should be used for preparing scArches surgery as well: 1. `batch_key`: this key is used to specify from which batch your query dataset comes. The HLCA reference model was set up to retain variation between individuals, and so rather than treating each sample or individual as a separate batch, each dataset was considered one batch. We therefore recommend using the same logic for an HLCA query, and set an entire dataset to a single batch. If your data has further splits that could result in specific batch effects, split your data into separate batches accordingly (e.g. if part of your data was generated with 10X 3', and the rest with 10X 5'). 2. `labels_key`: as the HLCA has a scANVI reference model, it used cell type labels as input for the training. These cell type labels were stored in a column named 'scanvi_label'. We recommend not using cell type labels for surgery, and so advise to set this column to 'unlabeled' (see below). 3. `unlabeled_category`: this variable specifies how cells without label were named for this specific model. As you can see, they were in this case set to the string 'unlabeled'.

Our test data will be considered only a single batch, and so we set our batch key to a single value:

[36]:
```
adata_query.obs["dataset"] = "Delorey_batch_1"
```

We will furthermore set the cell type key to the unlabeled_category for all our cells, and recommend doing the same for any dataset mapped to the HLCA:

[37]:
```
adata_query.obs["scanvi_label"] = "unlabeled"
```

Re-load the surgery model, now with the variables for adata_query set:

[38]:
```
surgery_model = sca.models.SCANVI.load_query_data(
    adata_query,
    ref_model_dir,
    freeze_dropout=True,
)
```

```
INFO     File ./HLCA_reference_model/model.pt already downloaded
```

19.1.6 Set relevant training/surgery length and kwargs

TO DO: add explanations/tuning options with these parameters

[39]:
```
surgery_epochs = 500
early_stopping_kwargs_surgery = {
    "early_stopping_monitor": "elbo_train",
    "early_stopping_patience": 10,
    "early_stopping_min_delta": 0.001,
    "plan_kwargs": {"weight_decay": 0.0},
}
```

19.2 Perform surgery on reference model by training on the query dataset

We will now update the reference model by performing scArches surgery. During surgery, only those parts of the model are trained that affect how your query is embedded; the reference embedding cannot change. In that way, the embedding of your query data is partly based on pre-learned patterns in the reference, and partly based on the query data itself.

[40]:
```
surgery_model.train(max_epochs=surgery_epochs, **early_stopping_kwargs_surgery)
```

```
INFO     Training for 500 epochs.
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Epoch 500/500: 100%|| 500/500 [01:15<00:00,  6.91it/s, loss=510, v_num=1]
```

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=500`␣
↪reached.
```

```
Epoch 500/500: 100%|| 500/500 [01:15<00:00,  6.61it/s, loss=510, v_num=1]
```

Let's save the model:

[41]:
```
surgery_model.save(surgery_model_dir, overwrite=True)
```

And load if starting from here:

[42]:
```
surgery_model = sca.models.SCANVI.load(
    surgery_model_dir, adata_query
)  # if already trained
```

```
INFO     File ./surgery_model/model.pt already downloaded
```

19.3 Obtain query latent embedding

Now that we have the updated model, we can calculate the low-dimensional representation or "embedding" of our query data. Importantly, this embedding is in the same space as the HLCA core/reference embedding that you loaded in the beginning of the script. Hence, we can combine the two embeddings afterwards (HLCA + your new data), and do joint clustering, UMAP embedding, label transfer etc.! The latent embedding will be stored in a new anndata under .X with the following command:

[43]:
```
adata_query_latent = sc.AnnData(surgery_model.get_latent_representation(adata_query))
```

Copy over .obs metadata from our query data:

[44]:
```
adata_query_latent.obs = adata_query.obs.loc[adata_query.obs.index, :]
```

19.4 Combine reference and query embedding into one joint embedding

Now that we have our query embedding, we can combine it with the pre-existing reference embedding that we downloaded at the top of this notebook. Once we have that joint embedding, we can do all kinds of analyses on the combined reference and query, including clustering, visualization, and label transfer (see below).

Before joining the reference and the query, let's specify for the cells from each whether they came from the reference or the query:

[45]:
```
adata_query_latent.obs["ref_or_query"] = "query"
adata_ref.obs["ref_or_query"] = "ref"
```

We will now combine the two embeddings to enable joint clustering etc. If you expect non-unique barcodes (.obs index), set `index_unique` to e.g. "_" (this will add a suffix to your barcodes to ensure we can keep apart reference and query barcodes) and `batch_key` to the obs column that you want to use as barcode suffix (e.g. "ref_or_query").

[46]:
```
combined_emb = sc.concat(
    (adata_ref, adata_query_latent), index_unique=None, join="outer"
)  # index_unique="_", batch_key="ref_or_query")
```

Save the combined embedding if wanted. As the data concatenation might have resulted in some mixed dtype obs columns, we will convert those to strings here to prevent writing errors below.

[47]:
```
for cat in combined_emb.obs.columns:
    if isinstance(combined_emb.obs[cat].values, pd.Categorical):
        pass
    elif pd.api.types.is_float_dtype(combined_emb.obs[cat]):
        pass
    else:
        print(
            f"Setting obs column {cat} (not categorical neither float) to strings to␣
↪prevent writing error."
        )
        combined_emb.obs[cat] = combined_emb.obs[cat].astype(str)
```

```
Setting obs column is_primary_data (not categorical neither float) to strings to prevent␣
↪writing error.
```

(continues on next page)

(continued from previous page)

```
Setting obs column dataset (not categorical neither float) to strings to prevent writing␣
↪error.
Setting obs column scanvi_label (not categorical neither float) to strings to prevent␣
↪writing error.
Setting obs column ref_or_query (not categorical neither float) to strings to prevent␣
↪writing error.
```

[48]:
```
combined_emb.write_h5ad("combined_embedding.h5ad")
```

Read if starting from here.

[49]:
```
combined_emb = sc.read_h5ad("combined_embedding.h5ad")
```

Note that it is possible to not only combine the embeddings of the refence and query, but to add the actual gene counts to this embedding. This enables querying of gene expression across the query and reference. The full HLCA, including normalized counts is publicly available on cellxgene. For now, we will just work with the embedding, since that is all we need to perform joint visualization and label transfer.

19.5 Label transfer

Next, we use a knn classifier to transfer the lables from the reference to the query. As the HLCA includes 5 levels of annotations (from coarse to fine), we will do the label transfer for every level of annotation. Note that some cell types don't have annotations for higher levels, e.g. mast cells do not have level 4 or 5 annotations. For those cell types, we will "propagate" to the higher levels, i.e. you will see "3_Mast cells" in level 4 and 5 annotations. (Most cell types don't have a level 5 annotation!) Therefore, all highest level annotations can be found under level 5.

[50]:
```
path_celltypes = os.path.join(ref_model_dir, "HLCA_celltypes_ordered.csv")
```

[51]:
```
url = "https://github.com/LungCellAtlas/HLCA_reproducibility/raw/main/supporting_files/
↪celltype_structure_and_colors/manual_anns_and_leveled_anns_ordered.csv" # "https://
↪github.com/LungCellAtlas/mapping_data_to_the_HLCA/raw/main/supporting_files/HLCA_
↪celltypes_ordered.csv"
gdown.download(url, path_celltypes, quiet=False)
```

Import the set of finest cell type labels, and their matching lower-level annotations (cell types are also ordered in a biologically sensible order in this table, you can use this order for downstream plotting etc. if wanted):

[52]:
```
cts_ordered = pd.read_csv(path_celltypes, index_col=0).rename(
    columns={f"Level_{lev}": f"labtransf_ann_level_{lev}" for lev in range(1, 6)}
)
```

[53]:
```
cts_ordered.head(5)
```

[53]:
```
                      labtransf_ann_level_1 labtransf_ann_level_2
Basal resting                     Epithelial     Airway epithelium  \
Suprabasal                        Epithelial     Airway epithelium
Hillock-like                      Epithelial     Airway epithelium
Deuterosomal                      Epithelial     Airway epithelium
Multiciliated (nasal)             Epithelial     Airway epithelium
```

(continues on next page)

(continued from previous page)

```
                        labtransf_ann_level_3 labtransf_ann_level_4
Basal resting                           Basal         Basal resting  \
Suprabasal                              Basal            Suprabasal
Hillock-like                            Basal          Hillock-like
Deuterosomal            Multiciliated lineage          Deuterosomal
Multiciliated (nasal)   Multiciliated lineage         Multiciliated

                        labtransf_ann_level_5  ordering   colors
Basal resting                 4_Basal resting         3  #FFFF00
Suprabasal                       4_Suprabasal         4  #1CE6FF
Hillock-like                   4_Hillock-like        11  #FF34FF
Deuterosomal                   4_Deuterosomal        13  #FF4A46
Multiciliated (nasal)   Multiciliated (nasal)        15  #008941
```

Let's add annotations for all five levels including forward-propagated labels (see above) to our adata_ref. They will be stored in adata_ref.obs under `labtransf_ann_level_[1-5]`.

[54]:
```
adata_ref.obs = adata_ref.obs.join(cts_ordered, on="ann_finest_level")
```

[55]:
```
adata_ref
```

[55]:
```
AnnData object with n_obs × n_vars = 584944 × 30
    obs: 'suspension_type', 'donor_id', 'is_primary_data', 'assay_ontology_term_id',
↪'cell_type_ontology_term_id', 'development_stage_ontology_term_id', 'disease_ontology_
↪term_id', 'self_reported_ethnicity_ontology_term_id', 'tissue_ontology_term_id',
↪'organism_ontology_term_id', 'sex_ontology_term_id', "3'_or_5'", 'BMI', 'age_or_mean_
↪of_age_range', 'age_range', 'anatomical_region_ccf_score', 'ann_coarse_for_GWAS_and_
↪modeling', 'ann_finest_level', 'ann_level_1', 'ann_level_2', 'ann_level_3', 'ann_level_
↪4', 'ann_level_5', 'cause_of_death', 'core_or_extension', 'dataset', 'fresh_or_frozen',
↪ 'log10_total_counts', 'lung_condition', 'mixed_ancestry', 'original_ann_level_1',
↪'original_ann_level_2', 'original_ann_level_3', 'original_ann_level_4', 'original_ann_
↪level_5', 'original_ann_nonharmonized', 'reannotation_type', 'sample', 'scanvi_label',
↪'sequencing_platform', 'smoking_status', 'study', 'subject_type', 'tissue_coarse_
↪unharmonized', 'tissue_detailed_unharmonized', 'tissue_dissociation_protocol', 'tissue_
↪level_2', 'tissue_level_3', 'tissue_sampling_method', 'total_counts', 'ref_or_query',
↪'labtransf_ann_level_1', 'labtransf_ann_level_2', 'labtransf_ann_level_3', 'labtransf_
↪ann_level_4', 'labtransf_ann_level_5', 'ordering', 'colors'
    uns: 'schema_version'
    obsm: 'X_umap'
    obsp: 'connectivities', 'distances'
```

check subset of copy results:

[56]:
```
adata_ref.obs.loc[:, ["ann_finest_level", "labtransf_ann_level_4"]].head(5)
```

[56]:
```
                                        ann_finest_level
CGATGTAAGTTACGGG_SC10                  Alveolar macrophages  \
ATTCTACCAAGGTTCT_HD68                 EC aerocyte capillary
P1_2_TGCTGCTAGCTCCTCT                  Alveolar macrophages
CTGATAGTCTTAGAGC_F01367                          EC arterial
D344_Brus_Dis1_CATTCGCGTGCCTGCA-1-14       Club (non-nasal)

                                      labtransf_ann_level_4
```

(continues on next page)

(continued from previous page)

```
CGATGTAAGTTACGGG_SC10                    Alveolar macrophages
ATTCTACCAAGGTTCT_HD68                   EC aerocyte capillary
P1_2_TGCTGCTAGCTCCTCT                    Alveolar macrophages
CTGATAGTCTTAGAGC_F01367                         3_EC arterial
D344_Brus_Dis1_CATTCGCGTGCCTGCA-1-14                     Club
```

Now run the label transfer commands. Note that this might take quite a while if you have a large query dataset! For our small test dataset, it should not take long.

Let's prepare our label transfer:

[57]:
```
knn_transformer = sca.utils.knn.weighted_knn_trainer(
    train_adata=adata_ref,
    train_adata_emb="X",  # location of our joint embedding
    n_neighbors=50,
)
```

```
Weighted KNN with n_neighbors = 50 ...
```

Now let's peform label transfer for the 5 levels of labels in the reference ("ann_level_1" to "ann_level_5").

[58]:
```
labels, uncert = sca.utils.knn.weighted_knn_transfer(
    query_adata=adata_query_latent,
    query_adata_emb="X",  # location of our embedding, query_adata.X in this case
    label_keys="labtransf_ann_level_",  # (start of) obs column name(s) for which to␣
↪transfer labels
    knn_model=knn_transformer,
    ref_adata_obs=adata_ref.obs,
)
```

```
finished!
```

With the commands above, we labeled every cell from the query (`labels` dataframe). Moreover, for each query cell we get an uncertainty score that tells you how confidently the label was assigned to the cell (`uncert` dataframe). This uncertainty score is based on how consistent the reference labels were among the nearest neighbors of the query cell. High label transfer uncertainty can indicate a number of things: 1. The cell lies in between two cellular phenotypes, e.g. in the case of a continuous transition of one cell type into another. 2. The cell is of a cell type or subtype not present in the reference. For example, the HLCA does not include erythrocytes. Erythrocytes in a query dataset will therefore likely be labeled with high uncertainty. Similarly, disease samples might include disease-affected cell types that look different from the cells in a healthy reference. These also likely have high label transfer uncertainty. 3. The mapping did not successfully remove batch-effects in the query data from the embedding. Query cells do not mix with the reference in the joint embedding, complicating confident label transfer. To distinguish low-uncertainty from high-uncertainty transferred labels, we will set our high-uncertainty labels to "unknown" instead of giving them a cell type label. Cells with high uncertainty should be looked into in downstream analysis.

We set the uncertainty threshold to 0.2, limiting the false positive rate to <0.5 (as per Sikkema et al., bioRxiv 2022). If you are dealing with data that you expect to look very different from your reference (e.g. mouse data or cell line data), you could consider setting this threshold higher.

[59]:
```
uncertainty_threshold = 0.2
```

Let's clean up the column names and add the transferred labels and matching uncertainties to our combined embedding (including both the query and the reference).

[60]:
```
labels.rename(
    columns={
        f"labtransf_ann_level_{lev}": f"ann_level_{lev}_transferred_label_unfiltered"
        for lev in range(1, 6)
    },
    inplace=True,
)
uncert.rename(
    columns={
        f"labtransf_ann_level_{lev}": f"ann_level_{lev}_transfer_uncert"
        for lev in range(1, 6)
    },
    inplace=True,
)
```

[61]:
```
combined_emb.obs = combined_emb.obs.join(labels)
combined_emb.obs = combined_emb.obs.join(uncert)
```

Now let's generate a filtered label column for each label, setting labels transferred with uncertainty >0.2 to "Unknown":

[62]:
```
for lev in range(1, 6):
    combined_emb.obs[f"ann_level_{lev}_transferred_label"] = combined_emb.obs[
        f"ann_level_{lev}_transferred_label_unfiltered"
    ].mask(
        combined_emb.obs[f"ann_level_{lev}_transfer_uncert"] > uncertainty_threshold,
        "Unknown",
    )
```

Let's take a look at the percentage of cells set to "unknown" after our filtering:

[63]:
```
print(
    f"Percentage of unknown per level, with uncertainty_threshold={uncertainty_threshold}
↪:"
)
for level in range(1, 6):
    print(
        f"Level {level}: {np.round(sum(combined_emb.obs[f'ann_level_{level}_transferred_
↪label'] =='Unknown')/adata_query.n_obs*100,2)}%"
    )
```

```
Percentage of unknown per level, with uncertainty_threshold=0.2:
Level 1: 0.56%
Level 2: 1.12%
Level 3: 10.64%
Level 4: 51.12%
Level 5: 51.23%
```

Important note! In some environments with older versions of scanpy/scvi-tools/scarches, there is a bug in the above code that we have not been able to properly pinpoint and fix. If you observe percentages of (close to) 100% of unknown above, you likely have the same bug and should update your packages. The transfered labels will then also be shuffled/random. (See also note at the top of this notebook).

19.6 Visualization of the joint reference and query embedding

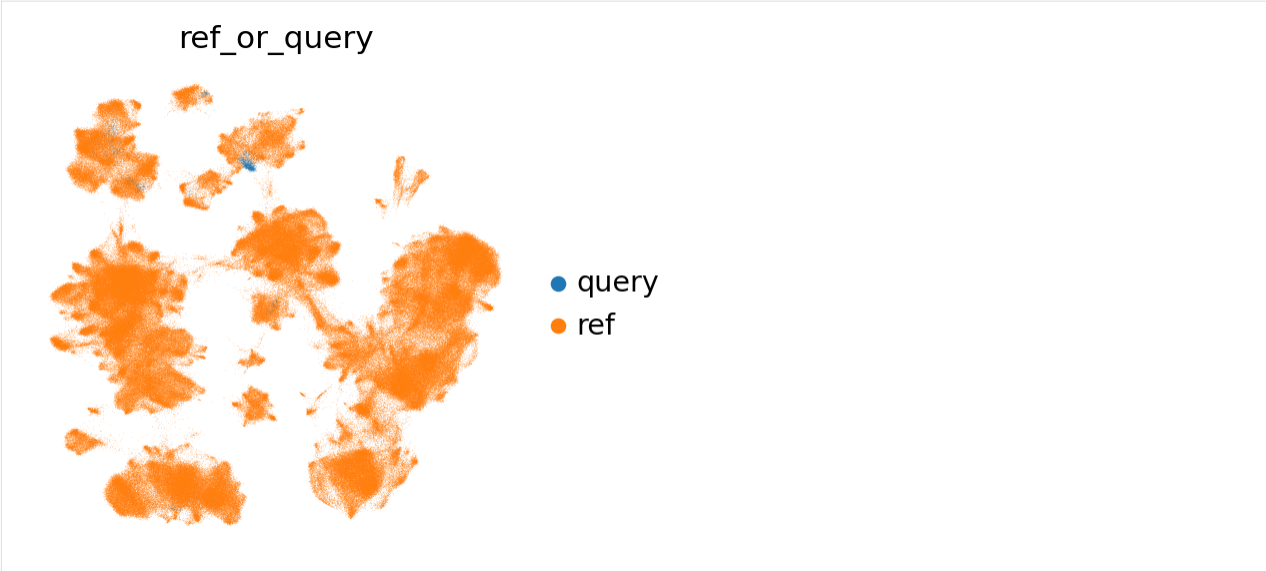
We will use a UMAP plot of our data to visually inspect the results of the mapping and label transfer. Calculating this will take a while on the HLCA (>.5M cells) + query.

[64]:
```
sc.pp.neighbors(combined_emb, n_neighbors=30)
sc.tl.umap(combined_emb)
```

```
/home/icb/lisa.sikkema/miniconda3/envs/HLCA_mapping_env_new_upgrade_scanpy/lib/python3.8/
↪site-packages/umap/distances.py:1063: NumbaDeprecationWarning: The 'nopython' keyword␣
↪argument was not supplied to the 'numba.jit' decorator. The implicit default value for␣
↪this argument is currently False, but it will be changed to True in Numba 0.59.0. See␣
↪https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-of-
↪object-mode-fall-back-behaviour-when-using-jit for details.
  @numba.jit()
/home/icb/lisa.sikkema/miniconda3/envs/HLCA_mapping_env_new_upgrade_scanpy/lib/python3.8/
↪site-packages/umap/distances.py:1071: NumbaDeprecationWarning: The 'nopython' keyword␣
↪argument was not supplied to the 'numba.jit' decorator. The implicit default value for␣
↪this argument is currently False, but it will be changed to True in Numba 0.59.0. See␣
↪https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-of-
↪object-mode-fall-back-behaviour-when-using-jit for details.
  @numba.jit()
/home/icb/lisa.sikkema/miniconda3/envs/HLCA_mapping_env_new_upgrade_scanpy/lib/python3.8/
↪site-packages/umap/distances.py:1086: NumbaDeprecationWarning: The 'nopython' keyword␣
↪argument was not supplied to the 'numba.jit' decorator. The implicit default value for␣
↪this argument is currently False, but it will be changed to True in Numba 0.59.0. See␣
↪https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-of-
↪object-mode-fall-back-behaviour-when-using-jit for details.
  @numba.jit()
/home/icb/lisa.sikkema/miniconda3/envs/HLCA_mapping_env_new_upgrade_scanpy/lib/python3.8/
↪site-packages/umap/umap_.py:660: NumbaDeprecationWarning: The 'nopython' keyword␣
↪argument was not supplied to the 'numba.jit' decorator. The implicit default value for␣
↪this argument is currently False, but it will be changed to True in Numba 0.59.0. See␣
↪https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-of-
↪object-mode-fall-back-behaviour-when-using-jit for details.
  @numba.jit()
```

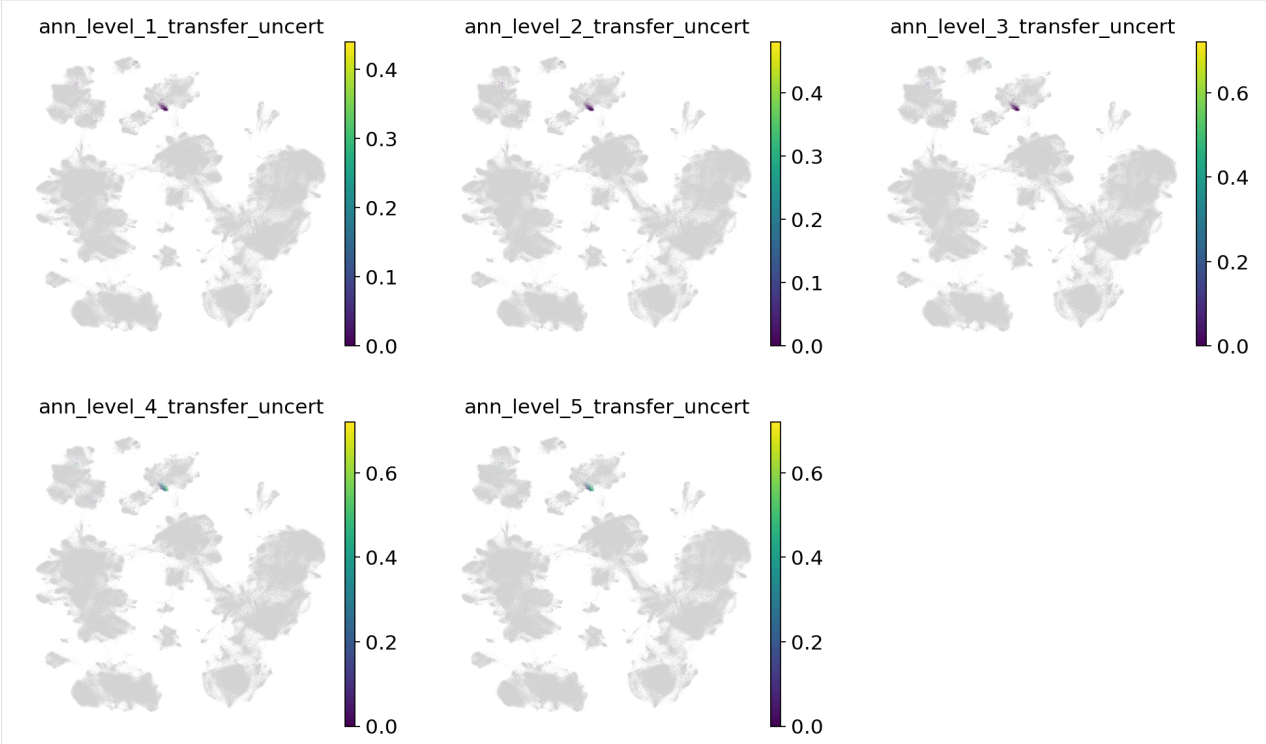
Let's first take a look at where our query cells are located in the umap. If they are completely separate from the reference, this could be a sign that something went wrong in the mapping. In our case, the query cells are largely mixing with or close to the reference cells in the UMAP.

[65]:
```
sc.pl.umap(combined_emb, color="ref_or_query", frameon=False, wspace=0.6)
```



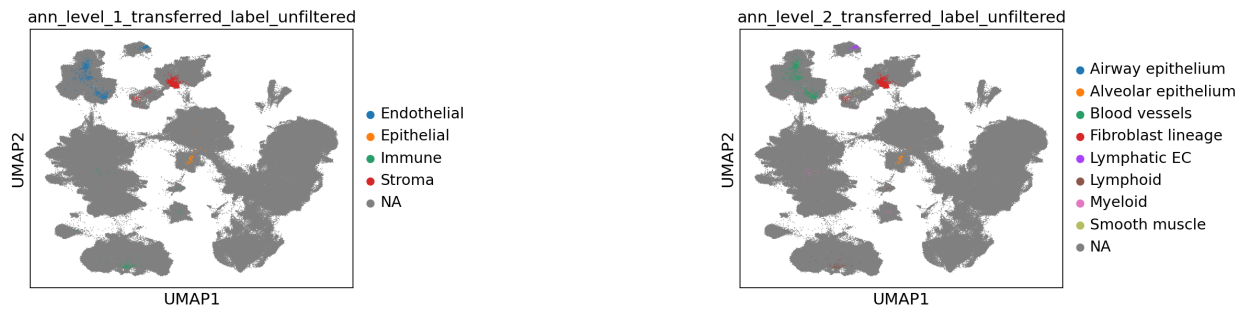
Now let's take a look at the label transfer uncertainties per level. Regions with high uncertainty can highlight interesting cell types/states, not present in the reference. Note that uncertainties will get higher, the more detailed we go. Note that as we only used very few cells in the query here, they are more difficult to see in the joint embedding.

[66]:
```
sc.pl.umap(
    combined_emb,
    color=[f"ann_level_{lev}_transfer_uncert" for lev in range(1, 6)],
    ncols=3,
    frameon=False,
)
```

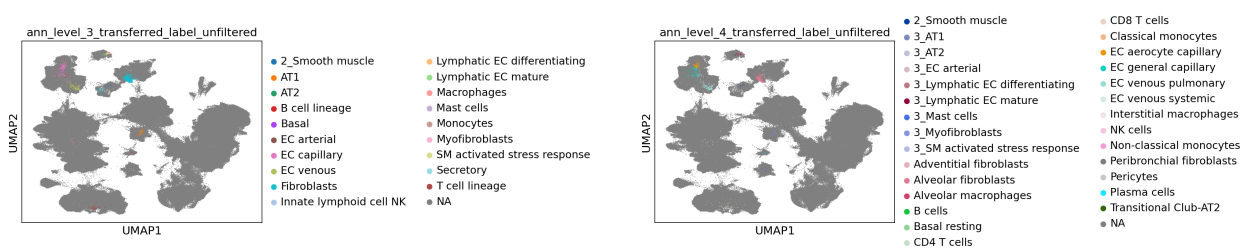


Now let's take a look at the transferred labels, at every level. Note that the color for "Unknown" switches per plot, and that all cells from the reference are set to NA.

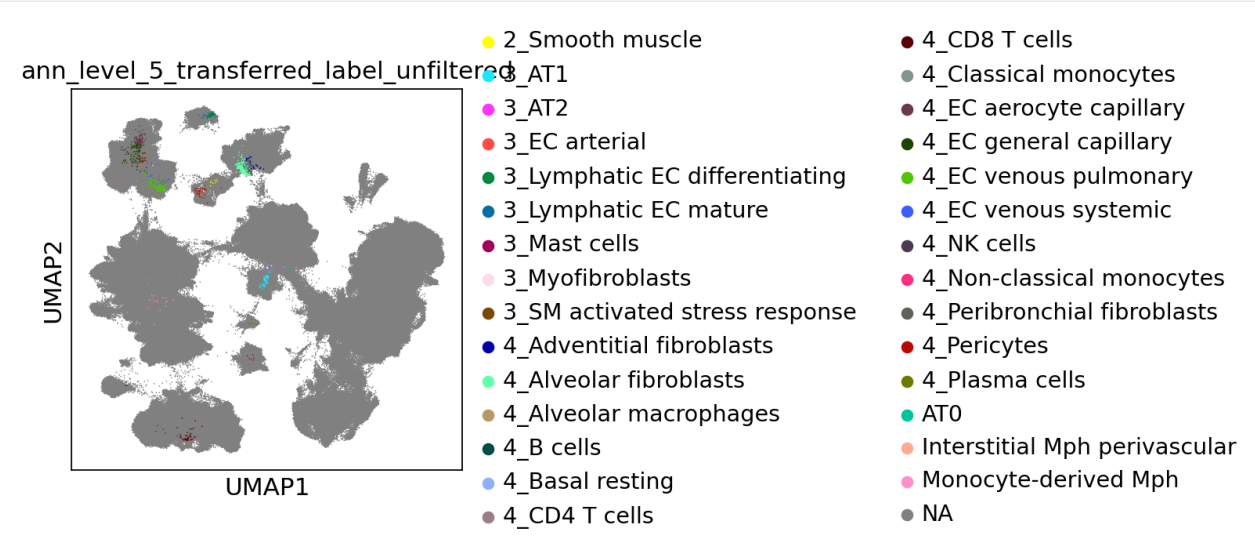
[67]:
```
sc.pl.umap(
    combined_emb,
    color=[f"ann_level_{lev}_transferred_label_unfiltered" for lev in range(1, 3)],
    na_color="grey",
    ncols=2,
    size=2,
    wspace=1,
)
```



[68]:
```
sc.pl.umap(
    combined_emb,
    color=[f"ann_level_{lev}_transferred_label_unfiltered" for lev in range(3, 5)],
    na_color="grey",
    ncols=2,
    size=2,
    wspace=1.4,
)
```

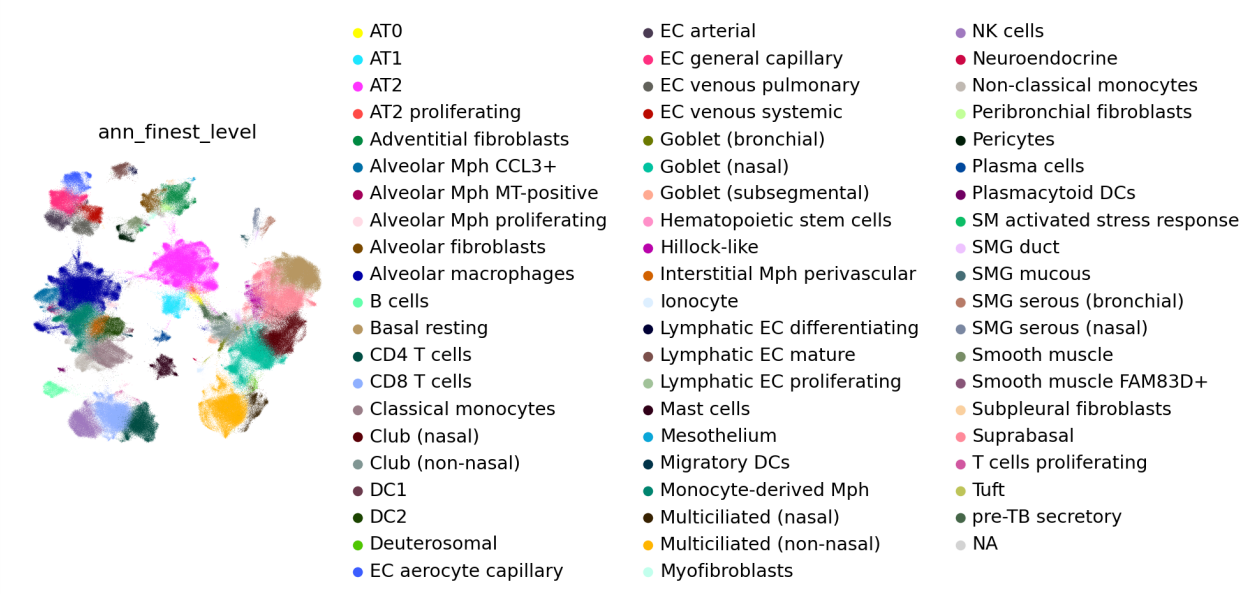


[69]:
```
sc.pl.umap(
    combined_emb,
    color=[f"ann_level_5_transferred_label_unfiltered"],
    na_color="grey",
    size=2,
)
```



For your reference, these are the annotations of the reference atlas:

[70]:
```
sc.pl.umap(combined_emb, color="ann_finest_level", frameon=False, wspace=0.6)
```



To visualize the original labels that were transferred per level, run the code below:

[71]:
```
# # copy over labels from reference adata
# for cat in [f"labtransf_ann_level_{lev}" for lev in range(1,6)]:
#     combined_emb.obs.loc[adata_ref.obs.index, cat] = adata_ref.obs[cat]
```

[72]:
```
# sc.pl.umap(
#     combined_emb,
#     color=[f"labtransf_ann_level_{lev}" for lev in range(1,6)],
#     frameon=False,
#     wspace=1.4,
#     ncols=2,
```

(continues on next page)

(continued from previous page)

```
# )
```

19.7 Visualization of the query alone, using reference-based embedding and including original gene expression values:

To get a better look at the query and the uncertainty levels, we can go back to the original query object (without reference, and with all genes still included), add the learned embedding and transfered labels, and calculate the UMAP:

[73]:
```
adata_query_final = (
    adata_query_unprep.copy()
)  # copy the original query adata, including gene counts
```

[74]:
```
adata_query_final.obsm["X_scarches_emb"] = adata_query_latent[
    adata_query_final.obs.index, :
].X  # copy over scArches/reference-based embedding
```

If your original query_adata has gene ids instead of gene symbols as var.index, switch that here for easier gene querying. Adapt column names where necessary.

[75]:
```
adata_query_final.var["gene_ids"] = adata_query_final.var.index
adata_query_final.var.index = adata_query_final.var.gene_names
adata_query_final.var.index.name = None
```

normalize gene counts and log transform (we'll do a simple total counts normalization here for simplicity):

[76]:
```
sc.pp.normalize_per_cell(adata_query_final, counts_per_cell_after=10000)
sc.pp.log1p(adata_query_final)
```

copy over label transfer columns:

[77]:
```
for col in combined_emb.obs.columns:
    if col.startswith("ann_level") and "transfer" in col:
        adata_query_final.obs[col] = combined_emb.obs.loc[
            adata_query_final.obs.index, col
        ]
```

calculate neigbhor graph based on scArches embedding, and generate UMAP:

[78]:
```
sc.pp.neighbors(adata_query_final, use_rep="X_scarches_emb")
sc.tl.umap(adata_query_final)
```

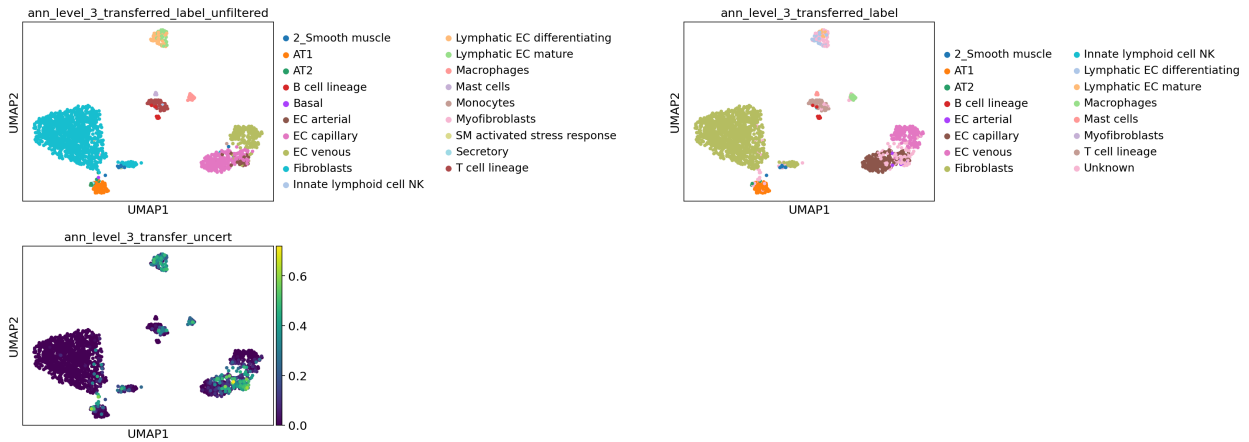
Visualize label transfer details for a level of choice:

[79]:
```
lev = 3
sc.pl.umap(
    adata_query_final,
    color=[
        f"ann_level_{lev}_transferred_label_unfiltered",
        f"ann_level_{lev}_transferred_label",
        f"ann_level_{lev}_transfer_uncert",
```

(continues on next page)

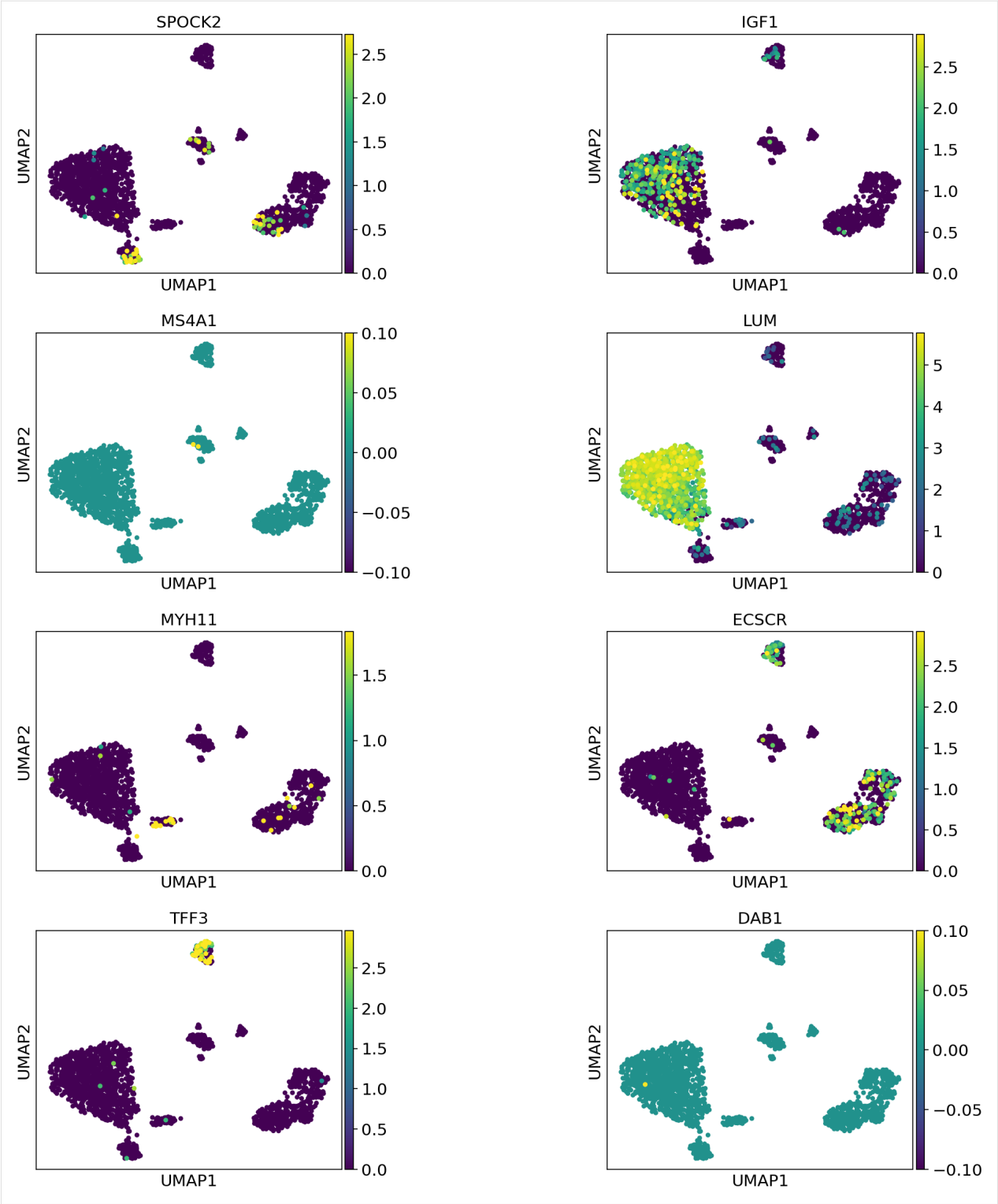
(continued from previous page)

```
    ],
    wspace=1.4,
    ncols=2,
)
```



Now visualize gene expression of any genes of your interest:

[80]:
```
sc.pl.umap(
    adata_query_final,
    color=["SPOCK2", "IGF1", "MS4A1", "LUM", "MYH11", "ECSCR", "TFF3", "DAB1"],
    vmax="p99",
    wspace=0.7,
    ncols=2,
)
```



Store the final adata_query if wanted:

[81]:
```
# adata_query_final.write_h5ad("./query_with_refbased_emb_and_anns.h5ad")
```

19.7. Visualization of the query alone, using reference-based embedding and including original gene expression values:

PYTHON MODULE INDEX

S

scarches.dataset, 7
scarches.plotting, 37
scarches.utils, 39
scarches.zenodo, 40
scarches.zenodo.deposition, 41
scarches.zenodo.file, 42

INDEX

A

add_annotations() (*in module scarches.utils*), 39
add_new_cell_type() (*scarches.models.scPoli method*), 16

C

classify() (*scarches.models.scPoli method*), 16
create_deposition() (*in module scarches.zenodo.deposition*), 41

D

delete_deposition() (*in module scarches.zenodo.deposition*), 41
differential_expression() (*scarches.models.TOTALVI method*), 29
download_file() (*in module scarches.zenodo.file*), 42
download_model() (*in module scarches.zenodo*), 40

E

EXPIMAP (*class in scarches.models*), 9

F

from_scvi_model() (*scarches.models.SCANVI class method*), 24

G

get_all_deposition_ids() (*in module scarches.zenodo.deposition*), 41
get_asw() (*scarches.plotting.SCVI_EVAL method*), 37
get_asw() (*scarches.plotting.TRVAE_EVAL method*), 38
get_classification_accuracy() (*scarches.plotting.SCVI_EVAL method*), 37
get_conditional_embeddings() (*scarches.models.scPoli method*), 16
get_ebm() (*scarches.plotting.SCVI_EVAL method*), 37
get_ebm() (*scarches.plotting.TRVAE_EVAL method*), 38
get_f1_score() (*scarches.plotting.SCVI_EVAL method*), 37
get_feature_correlation_matrix() (*scarches.models.TOTALVI method*), 31
get_knn_purity() (*scarches.plotting.SCVI_EVAL method*), 37
get_knn_purity() (*scarches.plotting.TRVAE_EVAL method*), 38
get_latent() (*scarches.models.EXPIMAP method*), 11
get_latent() (*scarches.models.scPoli method*), 17
get_latent_library_size() (*scarches.models.TOTALVI method*), 31
get_latent_score() (*scarches.plotting.SCVI_EVAL method*), 37
get_latent_score() (*scarches.plotting.TRVAE_EVAL method*), 38
get_likelihood_parameters() (*scarches.models.TOTALVI method*), 32
get_model_arch() (*scarches.plotting.SCVI_EVAL method*), 38
get_model_arch() (*scarches.plotting.TRVAE_EVAL method*), 38
get_nmi() (*scarches.plotting.SCVI_EVAL method*), 38
get_nmi() (*scarches.plotting.TRVAE_EVAL method*), 38
get_normalized_expression() (*scarches.models.TOTALVI method*), 32
get_protein_background_mean() (*scarches.models.TOTALVI method*), 33
get_protein_foreground_probability() (*scarches.models.TOTALVI method*), 33
get_prototypes_info() (*scarches.models.scPoli method*), 17

L

label_encoder() (*in module scarches.dataset*), 7
latent_as_anndata() (*scarches.plotting.SCVI_EVAL method*), 38
latent_as_anndata() (*scarches.plotting.TRVAE_EVAL method*), 38
latent_directions() (*scarches.models.EXPIMAP method*), 11
latent_enrich() (*scarches.models.EXPIMAP method*), 12
load_query_data() (*scarches.models.EXPIMAP class method*), 12

load_query_data() (*scarches.models.scPoli class method*), 17

M

mask_genes() (*scarches.models.EXPIMAP method*), 13
minify_adata() (*scarches.models.SCANVI method*), 25
minify_adata() (*scarches.models.SCVI method*), 20
module
    scarches.dataset, 7
    scarches.plotting, 37
    scarches.utils, 39
    scarches.zenodo, 40
    scarches.zenodo.deposition, 41
    scarches.zenodo.file, 42

N

nonzero_terms() (*scarches.models.EXPIMAP method*), 13

P

plot_abs_bfs() (*in module scarches.plotting*), 39
plot_history() (*scarches.plotting.SCVI_EVAL method*), 38
plot_history() (*scarches.plotting.TRVAE_EVAL method*), 38
plot_latent() (*scarches.plotting.SCVI_EVAL method*), 38
plot_latent() (*scarches.plotting.TRVAE_EVAL method*), 38
posterior_predictive_sample() (*scarches.models.TOTALVI method*), 34
predict() (*scarches.models.SCANVI method*), 25
publish_deposition() (*in module scarches.zenodo.deposition*), 41

R

remove_sparsity() (*in module scarches.dataset*), 7

S

sankey_diagram() (*in module scarches.plotting*), 39
SCANVI (*class in scarches.models*), 21
scarches.dataset
    module, 7
scarches.plotting
    module, 37
scarches.utils
    module, 39
scarches.zenodo
    module, 40
scarches.zenodo.deposition
    module, 41
scarches.zenodo.file
    module, 42
scPoli (*class in scarches.models*), 14
SCVI (*class in scarches.models*), 18
SCVI_EVAL (*class in scarches.plotting*), 37
setup_anndata() (*scarches.models.SCANVI class method*), 25
setup_anndata() (*scarches.models.SCVI class method*), 21
setup_anndata() (*scarches.models.TOTALVI class method*), 34
setup_mudata() (*scarches.models.TOTALVI class method*), 35

T

term_genes() (*scarches.models.EXPIMAP method*), 13
TOTALVI (*class in scarches.models*), 27
train() (*scarches.models.EXPIMAP method*), 13
train() (*scarches.models.SCANVI method*), 26
train() (*scarches.models.scPoli method*), 17
train() (*scarches.models.TOTALVI method*), 36
train() (*scarches.models.TRVAE method*), 9
TRVAE (*class in scarches.models*), 8
TRVAE_EVAL (*class in scarches.plotting*), 38
trVAEDataset (*in module scarches.dataset*), 7

U

update_deposition() (*in module scarches.zenodo.deposition*), 42
update_terms() (*scarches.models.EXPIMAP method*), 14
upload_file() (*in module scarches.zenodo.file*), 42
upload_model() (*in module scarches.zenodo*), 40

W

weighted_knn_trainer() (*in module scarches.utils*), 39
weighted_knn_transfer() (*in module scarches.utils*), 39