Tangram

Release 0.4.0

Department of AI/ML(Research Biology), Genentech

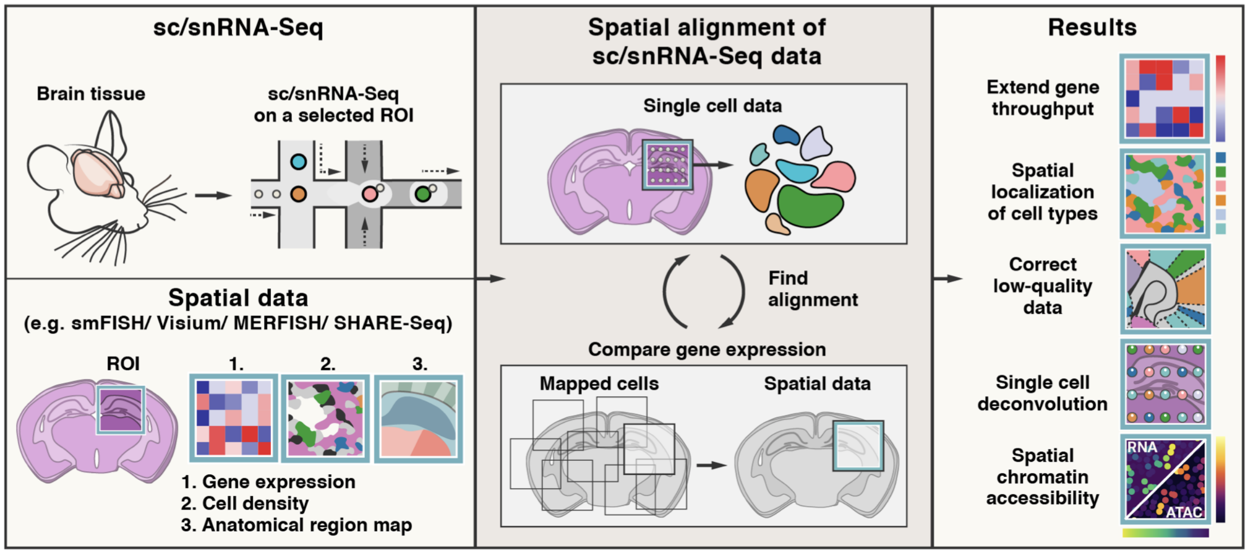
Sep 07, 2021

GENERAL

Contributors

- Tommaso Biancalani: author of method, PyPI maintainer `Contact: biancalt@gene.com`
- Gabriele Scalia: author of method `Contact: gabriele.scalia@roche.com`
- Ziqing Lu: PyPI maintainer `Contact: luz21@gene.com`
- Shreya Gaddam: PyPI maintainer `Contact: gaddams@gene.com`
- Anna Hupalowska: Artwork `Contact: ahupalow@broadinstitute.org`

Tangram is a Python package, written in PyTorch and based on scanpy , for mapping single-cell (or single-nucleus) gene expression data onto spatial gene expression data. The single-cell dataset and the spatial dataset should be collected from the same anatomical region/tissue type, ideally from a biological replicate, and need to share a set of genes. Tangram aligns the single-cell data in space by fitting gene expression on the shared genes. The best way to familiarize yourself with Tangram is to check out our *tutorials*.



CHAPTER

ONE

TANGRAM NEWS

- On Jan 28th 2021, Sten Linnarsson gave a talk at the WWNDev Forum and demostrated their mappings of the developmental mouse brain using Tangram.
- On Mar 9th 2021, Nicholas Eagles wrote a blog post about applying Tangram on Visium data.

CHAPTER

TWO

CITING TANGRAM

Tangram has been released in the following publication

Biancalani* T., Scalia* G. et al. - _Deep learning and alignment of spatially-resolved whole transcriptomes of single cells in the mouse brain with *Tangram* biorXiv 10.1101/2020.08.29.272831 (2020)

CHAPTER THREE

RELEASE NOTES

1.0.0 2021-08-06 - Initial Release

3.1 Getting Started

3.1.1 Installing Tangram

To install Tangram, make sure you have PyTorch and scanpy installed. If you need more details on the dependences, look at the environment.yml file.

Install Tangram from shell:

```
pip install tangram-sc
```

3.1.2 Running Tangram

Cell Level

To install Tangram, make sure you have PyTorch and scanpy installed. If you need more details on the dependences, look at the environment.yml file.

Install tangram-sc from shell:

```
pip install tangram-sc
```

Import tangram:

```
import tangram as tg
```

Then load your spatial data and your single cell data (which should be in AnnData format), and pre-process them using **tg.pp_adatas**:

```
ad_sp = sc.read_h5ad(path)
ad_sc = sc.read_h5ad(path)
tg.pp_adatas(ad_sc, ad_sp, genes=None)
```

The function **pp_adatas** finds the common genes between adata_sc, adata_sp, and saves them in two **adatas.uns** for mapping and analysis later. Also, it subsets the intersected genes to a set of training genes passed by **genes**. If **genes=None**, Tangram maps using all genes shared by the two datasets. Once the datasets are pre-processed we can map:

```
ad_map = tg.map_cells_to_space(ad_sc, ad_sp)
```

The returned AnnData, **ad_map** , is a cell-by-voxel structure where **ad_map.X[i, j]** gives the probability for cell *i* to be in voxel *j*. This structure can be used to project gene expression from the single cell data to space, which is achieved via **tg.project_genes**:

```
ad_ge = tg.project_genes(ad_map, ad_sc)
```

The returned **ad_ge** is a voxel-by-gene AnnData, similar to spatial data **ad_sp**, but where gene expression has been projected from the single cells. This allows to extend gene throughput, or correct for dropouts, if the single cells have higher quality (or more genes) than single cell data. It can also be used to transfer cell types onto space.

For more details on how to use Tangram check out our tutorial.

Cluster Level

To enable faster training and consume less memory, Tangram mapping can be done at cell cluster level.

Prepare the input data as the same you would do for cell level Tangram mapping. Then map using following code:

```
ad_map = tg.map_cells_to_space(
               ad_sc,
               ad_sp,
               mode='clusters',
               cluster_label='subclass_label')
```

Provided cluster_label must belong to ad_sc.obs. Above example code is to map at **subclass_label** level, and the **subclass_label** is in ad_sc.obs.

To project gene expression to space, use **tg.project_genes** and be sure to set the **cluster_label** argument to the same cluster label in mapping:

```
ad_ge = tg.project_genes(
              ad_map,
              ad_sc,
              cluster_label='subclass_label')
```

3.2 Tangram Under the Hood

Tangram instantiates a *Mapper* object passing the following arguments: * _S_: single cell matrix with shape cell-by-gene. Note that genes is the number of training genes. * _G_: spatial data matrix with shape voxels-by-genes. Voxel can contain multiple cells.

Then, Tangram searches for a mapping matrix *M*, with shape voxels-by-cells, where the element *M_ij* signifies the probability of cell *i* of being in spot *j*. Tangram computes the matrix *M* by minimizing the following loss:

where cos_sim is the cosine similarity. The meaning of the loss function is that gene expression of the mapped single cells should be as similar as possible to the spatial data *G*, under the cosine similarity sense.

The above accounts for basic Tangram usage. In our manuscript, we modified the loss function in several ways so as to add various kinds of prior knowledge, such as number of cell contained in each voxels.

3.3 Classes

tangram.mapping_optimizer	Library for instantiating and running the optimizer for Tangram.
tangram.mapping_utils	Mapping helpers
tangram.plot_utils	This module includes plotting utility functions.
tangram.utils	Utility functions to pre- and post-process data for Tangram.

3.3.1 tangram.mapping_optimizer

Description

Library for instantiating and running the optimizer for Tangram. The optimizer comes in two flavors, which correspond to two different classes: - Mapper: optimizer without filtering (i.e., all single cells are mapped onto space). At the end, the learned mapping matrix M is returned. - MapperConstrained: optimizer with filtering (i.e., only a subset of single cells are mapped onto space). At the end, the learned mapping matrix M and the learned filter F are returned.

Classes

Mapper(S, G[, d, d_source, lambda_g1, ...])	Allows instantiating and running the optimizer for Tangram, without filtering.
MapperConstrained(S, G, d[, lambda_d, ...])	Allows instantiating and running the optimizer for Tangram, with filtering.

tangram.mapping_optimizer.Mapper

`class tangram.mapping_optimizer.`**`Mapper`**(*S*, *G*, *d=None*, *d_source=None*, *lambda_g1=1.0*, *lambda_d=0*, *lambda_g2=0*, *lambda_r=0*, *device='cpu'*, *adata_map=None*, *random_state=None*)

Allows instantiating and running the optimizer for Tangram, without filtering. Once instantiated, the optimizer is run with the 'train' method, which also returns the mapping result.

Mapper.train(num_epochs[, learning_rate, ...])	Run the optimizer and returns the mapping outcome.

tangram.mapping_optimizer.Mapper.train

`Mapper.`**`train`**(*num_epochs*, *learning_rate=0.1*, *print_each=100*)

Run the optimizer and returns the mapping outcome.

Parameters

- **num_epochs** (*int*) – Number of epochs.
- **learning_rate** (*float*) – Optional. Learning rate for the optimizer. Default is 0.1.
- **print_each** (*int*) – Optional. Prints the loss each print_each epochs. If None, the loss is never printed. Default is 100.

Returns The optimized mapping matrix M (ndarray), with shape (number_cells, number_spots). training_history (dict): loss for each epoch

Return type output (ndarray)

tangram.mapping_optimizer.MapperConstrained

`class tangram.mapping_optimizer.`**`MapperConstrained`**(*S*, *G*, *d*, *lambda_d=1*, *lambda_g1=1*, *lambda_g2=1*, *lambda_r=0*, *lambda_count=1*, *lambda_f_reg=1*, *target_count=None*, *device='cpu'*, *adata_map=None*, *random_state=None*)

Allows instantiating and running the optimizer for Tangram, with filtering. Once instantiated, the optimizer is run with the 'train' method, which also returns the mapping and filter results.

MapperConstrained.train(num_epochs[, . . .])	Run the optimizer and returns the mapping outcome.

tangram.mapping_optimizer.MapperConstrained.train

`MapperConstrained.`**`train`**(*num_epochs*, *learning_rate=0.1*, *print_each=100*)

Run the optimizer and returns the mapping outcome.

Parameters

- **num_epochs** (*int*) – Number of epochs.
- **learning_rate** (*float*) – Optional. Learning rate for the optimizer. Default is 0.1.
- **print_each** (*int*) – Optional. Prints the loss each print_each epochs. If None, the loss is never printed. Default is 100.

Returns M (ndarray): is the optimized mapping matrix, shape = (number_cells, number_spots). f (ndarray): is the optimized filter, shape = (number_cells,). training_history (dict): loss for each epoch

Return type A tuple (output, F_out, training_history), with

3.3.2 tangram.mapping_utils

Description

Mapping helpers

Functions

adata_to_cluster_expression(adata, cluster_label)	Convert an AnnData to a new AnnData with cluster expressions.
map_cells_to_space(adata_sc, adata_sp[, . . .])	Map single cell data (*adata_sc*) on spatial data (*adata_sp*).
pp_adatas(adata_sc, adata_sp[, genes])	Pre-process AnnDatas so that they can be mapped.

tangram.mapping_utils.adata_to_cluster_expression

`tangram.mapping_utils.`**`adata_to_cluster_expression`**`(`*adata*, *cluster_label*, *scale=True*, *add_density=True*`)`

Convert an AnnData to a new AnnData with cluster expressions. Clusters are based on *cluster_label* in *adata.obs*. The returned AnnData has an observation for each cluster, with the cluster-level expression equals to the average expression for that cluster. All annotations in *adata.obs* except *cluster_label* are discarded in the returned AnnData.

Parameters

- **adata** (*AnnData*) – single cell data
- **cluster_label** (*String*) – field in *adata.obs* used for aggregating values
- **scale** (*bool*) – Optional. Whether weight input single cell by # of cells in cluster. Default is True.
- **add_density** (*bool*) – Optional. If True, the normalized number of cells in each cluster is added to the returned AnnData as obs.cluster_density. Default is True.

Returns aggregated single cell data

Return type AnnData

tangram.mapping_utils.map_cells_to_space

`tangram.mapping_utils.`**`map_cells_to_space`**`(`*adata_sc*, *adata_sp*, *cv_train_genes=None*, *cluster_label=None*, *mode='cells'*, *device='cpu'*, *learning_rate=0.1*, *num_epochs=1000*, *scale=True*, *lambda_d=0*, *lambda_g1=1*, *lambda_g2=0*, *lambda_r=0*, *lambda_count=1*, *lambda_f_reg=1*, *target_count=None*, *random_state=None*, *verbose=True*, *density_prior='rna_count_based'*`)`

Map single cell data (*adata_sc*) on spatial data (*adata_sp*).

Parameters

- **adata_sc** (*AnnData*) – single cell data
- **adata_sp** (*AnnData*) – gene spatial data
- **cv_train_genes** (*list*) – Optional. Training gene list. Default is None.
- **cluster_label** (*str*) – Optional. Field in *adata_sc.obs* used for aggregating single cell data. Only valid for *mode=clusters*.
- **mode** (*str*) – Optional. Tangram mapping mode. Currently supported: 'cell', 'clusters', 'constrained'. Default is 'cell'.
- **device** (*string or torch.device*) – Optional. Default is 'cpu'.
- **learning_rate** (*float*) – Optional. Learning rate for the optimizer. Default is 0.1.
- **num_epochs** (*int*) – Optional. Number of epochs. Default is 1000.
- **scale** (*bool*) – Optional. Whether weight input single cell data by the number of cells in each cluster, only valid when cluster_label is not None. Default is True.
- **lambda_d** (*float*) – Optional. Hyperparameter for the density term of the optimizer. Default is 0.

- **lambda_g1** (*float*) – Optional. Hyperparameter for the gene-voxel similarity term of the optimizer. Default is 1.
- **lambda_g2** (*float*) – Optional. Hyperparameter for the voxel-gene similarity term of the optimizer. Default is 0.
- **lambda_r** (*float*) – Optional. Strength of entropy regularizer. An higher entropy promotes probabilities of each cell peaked over a narrow portion of space. lambda_r = 0 corresponds to no entropy regularizer. Default is 0.
- **lambda_count** (*float*) – Optional. Regularizer for the count term. Default is 1. Only valid when mode == 'constrained'
- **lambda_f_reg** (*float*) – Optional. Regularizer for the filter, which promotes Boolean values (0s and 1s) in the filter. Only valid when mode == 'constrained'. Default is 1.
- **target_count** (*int*) – Optional. The number of cells to be filtered. Default is None.
- **random_state** (*int*) – Optional. pass an int to reproduce training. Default is None.
- **verbose** (*bool*) – Optional. If print training details. Default is True.
- **density_prior** (*str, ndarray or None*) – Spatial density of spots, when is a string, value can be 'rna_count_based' or 'uniform', when is a ndarray, shape = (number_spots,). This array should satisfy the constraints sum() == 1. If None, the density term is ignored. Default value is 'rna_count_based'.

Returns a cell-by-spot AnnData containing the probability of mapping cell i on spot j. The *uns* field of the returned AnnData contains the training genes.

tangram.mapping_utils.pp_adatas

tangram.mapping_utils.**pp_adatas**(*adata_sc*, *adata_sp*, *genes=None*)

Pre-process AnnDatas so that they can be mapped. Specifically: - Remove genes that all entries are zero - Find the intersection between adata_sc, adata_sp and given marker gene list, save the intersected markers in two adatas - Calculate density priors and save it with adata_sp

Parameters

- **adata_sc** (*AnnData*) – single cell data
- **adata_sp** (*AnnData*) – spatial expression data
- **genes** (*List*) – Optional. List of genes to use. If *None*, all genes are used.

Returns update adata_sc by creating *uns training_genes overlap_genes* fields update adata_sp by creating *uns training_genes overlap_genes* fields and creating *obs rna_count_based_density* & *uniform_density* field

3.3.3 tangram.plot_utils

Description

This module includes plotting utility functions.

Functions

`construct_obs_plot`(df_plot, adata[, perc, ...])	
`convert_adata_array`(adata)	
`ordered_predictions`(xs, ys, preds[, reverse])	Utility function that orders 2d points based on values associated to each point.
`plot_annotation_entropy`(adata_map[, annotation])	Utility function to plot entropy box plot by each annotation.
`plot_auc`(df_all_genes[, test_genes])	Plots auc curve which is used to evaluate model performance.
`plot_cell_annotation`(adata_map, adata_sp[, ...])	Transfer an annotation for a single cell dataset onto space, and visualize corresponding spatial probability maps.
`plot_cell_annotation_sc`(adata_sp, ...[, perc])	
`plot_gene_sparsity`(adata_1, adata_2[, ...])	Compare sparsity of all genes between *adata_1* and *adata_2*.
`plot_genes`(genes, adata_measured, ...[, x, ...])	Utility function to plot and compare original and projected gene spatial pattern ordered by intensity of the gene signal.
`plot_genes_sc`(genes, adata_measured, ...[, ...])	
`plot_test_scores`(df_gene_score[, bins, alpha])	Plots gene level test scores with each gene's sparsity for mapping result.
`plot_training_scores`(adata_map[, bins, alpha])	Plots the 4-panel training diagnosis plot
`q_value`(data, perc)	Computes min and max values according to percentile for colormap in plot functions
`quick_plot_gene`(gene, adata[, x, y, s, log, ...])	Utility function to quickly plot a gene in a AnnData structure ordered by intensity of the gene signal.

tangram.plot_utils.construct_obs_plot

tangram.plot_utils.**construct_obs_plot**(*df_plot*, *adata*, *perc=0*, *suffix=None*)

tangram.plot_utils.convert_adata_array

tangram.plot_utils.**convert_adata_array**(*adata*)

tangram.plot_utils.ordered_predictions

tangram.plot_utils.**ordered_predictions**(*xs*, *ys*, *preds*, *reverse=False*)

Utility function that orders 2d points based on values associated to each point.

Parameters

- **xs** (*Pandas series*) – Sequence of x coordinates (floats).
- **ys** (*Pandas series*) – Sequence of y coordinates (floats).
- **preds** (*Pandas series*) – Sequence of spatial prediction.

- **reverse** (*bool*) – Optional. False will sort ascending, True will sort descending. Default is False.

Returns Returns the ordered xs, ys, preds.

tangram.plot_utils.plot_annotation_entropy

`tangram.plot_utils.`**`plot_annotation_entropy`**(*adata_map*, *annotation='cell_type'*)

Utility function to plot entropy box plot by each annotation.

Parameters

- **adata_map** (*AnnData*) – cell-by-voxel tangram mapping result.
- **annotation** (*str*) – Optional. Must be a column in *adata_map.obs*. Default is 'cell_type'.

Returns None

tangram.plot_utils.plot_auc

`tangram.plot_utils.`**`plot_auc`**(*df_all_genes*, *test_genes=None*)

Plots auc curve which is used to evaluate model performance.

Parameters

- **df_all_genes** (*Pandas dataframe*) – returned by compare_spatial_geneexp(adata_ge, adata_sp);
- **test_genes** (*list*) – list of test genes, if not given, test_genes will be set to genes where 'is_training' field is False

Returns None

tangram.plot_utils.plot_cell_annotation

`tangram.plot_utils.`**`plot_cell_annotation`**(*adata_map*, *adata_sp*, *annotation='cell_type'*, *x='x'*, *y='y'*, *nrows=1*, *ncols=1*, *s=5*, *cmap='viridis'*, *subtitle_add=False*, *robust=False*, *perc=0*, *invert_y=True*)

Transfer an annotation for a single cell dataset onto space, and visualize corresponding spatial probability maps.

Parameters

- **adata_map** (*AnnData*) – cell-by-spot AnnData containing mapping result
- **adata_sp** (*AnnData*) – spot-by-gene spatial AnnData
- **annotation** (*str*) – Optional. Must be a column in *adata_map.obs*. Default is 'cell_type'.
- **x** (*str*) – Optional. Column name for spots x-coordinates (must be in *adata_map.var*). Default is 'x'.
- **y** (*str*) – Optional. Column name for spots y-coordinates (must be in *adata_map.var*). Default is 'y'.
- **nrows** (*int*) – Optional. Number of rows of the subplot grid. Default is 1.
- **ncols** (*int*) – Optional. Number of columns of the subplot grid. Default is 1.
- **s** (*float*) – Optional. Marker size. Default is 5.

- **cmap** (*str*) – Optional. Name of colormap. Default is 'viridis'.
- **subtitle_add** (*bool*) – Optional. If add annotation name as the subtitle. Default is False.
- **robust** (*bool*) – Optional. If True, the colormap range is computed with given percentiles instead of extreme values.
- **perc** (*float*) – Optional. percentile used to calculate colormap range, only used when robust is True. Default is zero.
- **invert_y** (*bool*) – Optional. If invert the y axis for the plot. Default is True.

Returns None

tangram.plot_utils.plot_cell_annotation_sc

`tangram.plot_utils.`**`plot_cell_annotation_sc`**`(`*adata_sp*, *annotation_list*, *perc=0*`)`

tangram.plot_utils.plot_gene_sparsity

`tangram.plot_utils.`**`plot_gene_sparsity`**`(`*adata_1*, *adata_2*, *xlabel='adata_1'*, *ylabel='adata_2'*, *genes=None*, *s=1*`)`

Compare sparsity of all genes between *adata_1* and *adata_2*.

Parameters

- **adata_1** (*AnnData*) – Input data
- **adata_2** (*AnnData*) – Input data
- **xlabel** (*str*) – Optional. For setting the xlabel in the plot. Default is 'adata_1'.
- **ylabel** (*str*) – Optional. For setting the ylabel in the plot. Default is 'adata_2'.
- **genes** (*list*) – Optional. List of genes to use. If *None*, all genes are used.
- **s** (*float*) – Optional. Controls the size of marker. Default is 1.

Returns None

tangram.plot_utils.plot_genes

`tangram.plot_utils.`**`plot_genes`**`(`*genes*, *adata_measured*, *adata_predicted*, *x='x'*, *y='y'*, *s=5*, *log=False*, *cmap='inferno'*, *robust=False*, *perc=0*, *invert_y=True*`)`

Utility function to plot and compare original and projected gene spatial pattern ordered by intensity of the gene signal.

Parameters

- **genes** (*list*) – list of gene names (str).
- **adata_measured** (*AnnData*) – ground truth gene spatial AnnData
- **adata_predicted** (*AnnData*) – projected gene spatial AnnData, can also be adata_ge_cv AnnData returned by cross_validation under 'loo' mode
- **x** (*str*) – Optional. Column name for spots x-coordinates (must be in *adata_measured.var* and *adata_predicted.var*). Default is 'x'.
- **y** (*str*) – Optional. Column name for spots y-coordinates (must be in *adata_measured.var* and *adata_predicted.var*). Default is 'y'.

- **s** (*float*) – Optional. Marker size. Default is 5.
- **log** – Optional. Whether to apply the log before plotting. Default is False.
- **cmap** (*str*) – Optional. Name of colormap. Default is 'inferno'.
- **robust** (*bool*) – Optional. If True, the colormap range is computed with given percentiles instead of extreme values.
- **perc** (*float*) – Optional. percentile used to calculate colormap range, only used when robust is True. Default is zero.
- **invert_y** (*bool*) – Optional. If invert the y axis for the plot. Default is True.

Returns None

tangram.plot_utils.plot_genes_sc

tangram.plot_utils.**plot_genes_sc**(*genes*, *adata_measured*, *adata_predicted*, *cmap='inferno'*, *perc=0*)

tangram.plot_utils.plot_test_scores

tangram.plot_utils.**plot_test_scores**(*df_gene_score*, *bins=10*, *alpha=0.7*)

Plots gene level test scores with each gene's sparsity for mapping result.

Parameters

- **df_gene_score** (*Pandas dataframe*) – returned by compare_spatial_geneexp(adata_ge, adata_sp, adata_sc); with "gene names" as the index and "score", "sparsity_sc", "sparsity_sp", "sparsity_diff" as the columns
- **bins** (*int or string*) – Optional. Default is 10.
- **alpha** (*float*) – Optional. Ranges from 0-1, and controls the opacity. Default is 0.7.

Returns None

tangram.plot_utils.plot_training_scores

tangram.plot_utils.**plot_training_scores**(*adata_map*, *bins=10*, *alpha=0.7*)

Plots the 4-panel training diagnosis plot

Parameters

- **adata_map** (*AnnData*) –
- **bins** (*int or string*) – Optional. Default is 10.
- **alpha** (*float*) – Optional. Ranges from 0-1, and controls the opacity. Default is 0.7.

Returns None

tangram.plot_utils.q_value

`tangram.plot_utils.q_value`(*data*, *perc*)

Computes min and max values according to percentile for colormap in plot functions

Parameters

- `data` (*numpy array*) – input
- `perc` (*float*) – percentile that between 0 and 100 inclusive

Returns will be later used to define the data range covers by the colormap

Return type tuple of floats

tangram.plot_utils.quick_plot_gene

`tangram.plot_utils.quick_plot_gene`(*gene*, *adata*, *x='x'*, *y='y'*, *s=50*, *log=False*, *cmap='viridis'*, *robust=False*, *perc=0*)

Utility function to quickly plot a gene in a AnnData structure ordered by intensity of the gene signal.

Parameters

- `gene` (*str*) – Gene name.
- `adata` (*AnnData*) – spot-by-gene spatial data.
- `x` (*str*) – Optional. Column name for spots x-coordinates (must be in *adata.var*). Default is 'x'.
- `y` (*str*) – Optional. Column name for spots y-coordinates (must be in *adata.var*). Default is 'y'.
- `s` (*float*) – Optional. Marker size. Default is 5.
- `log` – Optional. Whether to apply the log before plotting. Default is False.
- `cmap` (*str*) – Optional. Name of colormap. Default is 'viridis'.
- `robust` (*bool*) – Optional. If True, the colormap range is computed with given percentiles instead of extreme values.
- `perc` (*float*) – Optional. percentile used to calculate colormap range, only used when robust is True. Default is zero.

Returns None

3.3.4 tangram.utils

Description

Utility functions to pre- and post-process data for Tangram.

Functions

annotate_gene_sparsity(adata)	Annotates gene sparsity in given Anndatas.
compare_spatial_geneexp(adata_ge, adata_sp)	Compares generated spatial data with the true spatial data
count_cell_annotations(adata_map, adata_sc, . . .)	Count cells in a voxel for each annotation.
create_segment_cell_df(adata_sp)	Produces a Pandas dataframe where each row is a segmentation object, columns reveals its position information.
cross_val(adata_sc, adata_sp[, . . .])	Executes cross validation
cv_data_gen(adata_sc, adata_sp[, cv_mode])	Generates pair of training/test gene indexes cross validation datasets
deconvolve_cell_annotations(adata_sp[, . . .])	Assigns cell annotation to each segmented cell.
df_to_cell_types(df, cell_types)	Utility function that "randomly" assigns cell coordinates in a voxel to known numbers of cell types in that voxel.
eval_metric(df_all_genes[, test_genes])	Compute metrics on given test_genes set for evaluation
get_matched_genes(prior_genes_names, . . . [, . . .])	Given the list of genes in the spatial data and the list of genes in the single nuclei, identifies the subset of genes included in both lists and returns the corresponding matching indices.
one_hot_encoding(l[, keep_aggregate])	Given a sequence, returns a DataFrame with a column for each unique value in the sequence and a one-hot-encoding.
project_cell_annotations(adata_map, adata_sp)	Transfer *annotation* from single cell data onto space.
project_genes(adata_map, adata_sc[, . . .])	Transfer gene expression from the single cell onto space.
read_pickle(filename)	Helper to read pickle file which may be zipped or not.
transfer_annotations_prob(mapping_matrix, . . .)	Transfer cell annotations onto space through a mapping matrix.
transfer_annotations_prob_filter(. . .)	Transfer cell annotations onto space through a mapping matrix and a filter.

tangram.utils.annotate_gene_sparsity

tangram.utils.**annotate_gene_sparsity**(*adata*)

Annotates gene sparsity in given Anndatas. Update given Anndata by creating *var* "sparsity" field with gene_sparsity (1 - % non-zero observations).

Parameters **adata** (*Anndata*) – single cell or spatial data.

Returns None

tangram.utils.compare_spatial_geneexp

tangram.utils.**compare_spatial_geneexp**(*adata_ge*, *adata_sp*, *adata_sc=None*, *genes=None*)

Compares generated spatial data with the true spatial data

Parameters

- **adata_ge** (*AnnData*) – generated spatial data returned by *project_genes*
- **adata_sp** (*AnnData*) – gene spatial data
- **adata_sc** (*AnnData*) – Optional. When passed, sparsity difference between adata_sc and adata_sp will be calculated. Default is None.

- **genes** (*list*) – Optional. When passed, returned output will be subset on the list of genes. Default is None.

Returns

a dataframe with columns: 'score', 'is_training', 'sparsity_sp'(spatial data sparsity). Columns - 'sparsity_sc'(single cell data sparsity), 'sparsity_diff'(spatial sparsity - single cell sparsity) returned only when adata_sc is passed.

Return type Pandas Dataframe

tangram.utils.count_cell_annotations

`tangram.utils.`**`count_cell_annotations`**`(`*adata_map*, *adata_sc*, *adata_sp*, *annotation='cell_type'*, *threshold=0.5*`)`

Count cells in a voxel for each annotation.

Parameters

- **adata_map** (*AnnData*) – cell-by-spot AnnData returned by *train* function.
- **adata_sc** (*AnnData*) – cell-by-gene AnnData.
- **adata_sp** (*AnnData*) – spatial AnnData data used to save the mapping result.
- **annotation** (*str*) – Optional. Cell annotations matrix with shape (number_cells, number_annotations). Default is 'cell_type'.
- **threshold** (*float*) – Optional. Valid for using with adata_map.obs['F_out'] from 'constrained' mode mapping. Cell's probability below this threshold will be dropped. Default is 0.5.

Returns None. Update spatial AnnData by creating *obsm tangram_ct_count* field which contains a dataframe that each row is a spot and each column has the cell count for each cell annotation (number_spots, number_annotations).

tangram.utils.create_segment_cell_df

`tangram.utils.`**`create_segment_cell_df`**`(`*adata_sp*`)`

Produces a Pandas dataframe where each row is a segmentation object, columns reveals its position information.

Parameters **adata_sp** (*AnnData*) – spot-by-gene AnnData structure. Must contain obsm.['image_features']

Returns None. Update spatial AnnData.uns['tangram_cell_segmentation'] with a dataframe: each row represents a segmentation object (single cell/nuclei). Columns are 'spot_idx' (voxel id), and 'y', 'x', 'centroids' to specify the position of the segmentation object. Update spatial AnnData.obsm['trangram_spot_centroids'] with a sequence

tangram.utils.cross_val

`tangram.utils.`**`cross_val`**(*adata_sc*, *adata_sp*, *cluster_label=None*, *mode='clusters'*, *scale=True*, *lambda_d=0*, *lambda_g1=1*, *lambda_g2=0*, *lambda_r=0*, *lambda_count=1*, *lambda_f_reg=1*, *target_count=None*, *num_epochs=1000*, *device='cuda:0'*, *learning_rate=0.1*, *cv_mode='loo'*, *return_gene_pred=False*, *density_prior=None*, *random_state=None*, *verbose=False*)

Executes cross validation

Parameters

- **adata_sc** (*AnnData*) – single cell data
- **adata_sp** (*AnnData*) – gene spatial data
- **cluster_label** (*str*) – the level that the single cell data will be aggregate at, this is only valid for clusters mode mapping
- **mode** (*str*) – Optional. Tangram mapping mode. Currently supported: 'cell', 'clusters', 'constrained'. Default is 'clusters'.
- **scale** (*bool*) – Optional. Whether weight input single cell by # of cells in cluster, only valid when cluster_label is not None. Default is True.
- **lambda_g1** (*float*) – Optional. Strength of Tangram loss function. Default is 1.
- **lambda_d** (*float*) – Optional. Strength of density regularizer. Default is 0.
- **lambda_g2** (*float*) – Optional. Strength of voxel-gene regularizer. Default is 0.
- **lambda_r** (*float*) – Optional. Strength of entropy regularizer. Default is 0.
- **lambda_count** (*float*) – Optional. Regularizer for the count term. Default is 1. Only valid when mode == 'constrained'
- **lambda_f_reg** (*float*) – Optional. Regularizer for the filter, which promotes Boolean values (0s and 1s) in the filter. Only valid when mode == 'constrained'. Default is 1.
- **target_count** (*int*) – Optional. The number of cells to be filtered. Default is None.
- **num_epochs** (*int*) – Optional. Number of epochs. Default is 1000.
- **learning_rate** (*float*) – Optional. Learning rate for the optimizer. Default is 0.1.
- **device** (*str or torch.device*) – Optional. Default is 'cuda:0'.
- **cv_mode** (*str*) – Optional. cross validation mode, 'loo' ('leave-one-out') and '10fold' supported. Default is 'loo'.
- **return_gene_pred** (*bool*) – Optional. if return prediction and true spatial expression data for test gene, only applicable when 'loo' mode is on, default is False.
- **density_prior** (*ndarray or str*) – Spatial density of spots, when is a string, value can be 'rna_count_based' or 'uniform', when is a ndarray, shape = (number_spots,). This array should satisfy the constraints sum() == 1. If not provided, the density term is ignored.
- **random_state** (*int*) – Optional. pass an int to reproduce training. Default is None.
- **verbose** (*bool*) – Optional. If print training details. Default is False.

Returns a dictionary contains information of cross validation (hyperparameters, average test score and train score, etc.) adata_ge_cv (AnnData): predicted spatial data by LOOCV. Only returns when *return_gene_pred* is True and in 'loo' mode. test_gene_df (Pandas dataframe): dataframe with columns: 'score', 'is_training', 'sparsity_sp'(spatial data sparsity)

Return type cv_dict (dict)

tangram.utils.cv_data_gen

`tangram.utils.`**`cv_data_gen`**`(`*adata_sc*, *adata_sp*, *cv_mode='loo'*`)`

Generates pair of training/test gene indexes cross validation datasets

Parameters

- **`adata_sc`** (*`AnnData`*) – single cell data
- **`adata_sp`** (*`AnnData`*) – gene spatial data
- **`mode`** (*`str`*) – Optional. support 'loo' and '10fold'. Default is 'loo'.

Yields *tuple* – list of train_genes, list of test_genes

tangram.utils.deconvolve_cell_annotations

`tangram.utils.`**`deconvolve_cell_annotations`**`(`*adata_sp*, *filter_cell_annotation=None*`)`

Assigns cell annotation to each segmented cell. Produces an AnnData structure that saves the assignment in its obs dataframe.

Parameters

- **`adata_sp`** (*`AnnData`*) – Spatial AnnData structure.
- **`filter_cell_annotation`** (*`sequence`*) – Optional. Sequence of cell annotation names to be considered for deconvolution. Default is None. When no values passed, all cell annotation names in adata_sp.obsm["tangram_ct_pred"] will be used.

Returns Saves the cell annotation assignment result in its obs dataframe where each row representing a segmentation object, column 'x', 'y', 'centroids' contain its position and column 'cluster' is the assigned cell annotation.

Return type AnnData

tangram.utils.df_to_cell_types

`tangram.utils.`**`df_to_cell_types`**`(`*df*, *cell_types*`)`

Utility function that "randomly" assigns cell coordinates in a voxel to known numbers of cell types in that voxel. Used for deconvolution.

Parameters

- **`df`** (*`DataFrame`*) – Columns correspond to cell types. Each row in the DataFrame corresponds to a voxel and specifies the known number of cells in that voxel for each cell type (int). The additional column 'centroids' specifies the coordinates of the cells in the voxel (sequence of (x,y) pairs).
- **`cell_types`** (*`sequence`*) – Sequence of cell type names to be considered for deconvolution. Columns in 'df' not included in 'cell_types' are ignored for assignment.

Returns A dictionary <cell type name> -> <list of (x,y) coordinates for the cell type>

tangram.utils.eval_metric

`tangram.utils.`**`eval_metric`**(*df_all_genes*, *test_genes=None*)

Compute metrics on given test_genes set for evaluation

Parameters

- **`df_all_genes`** (*`Pandas dataframe`*) – returned by compare_spatial_geneexp(adata_ge, adata_sp);
- **`test_genes`** (*`list`*) – list of test genes, if not given, test_genes will be set to genes where 'is_training' field is False

Returns dict with values of each evaluation metric ("avg_test_score", "avg_train_score", "auc_score"), tuple of auc fitted coordinates and raw coordinates(test_score vs. sparsity_sp coordinates)

tangram.utils.get_matched_genes

`tangram.utils.`**`get_matched_genes`**(*prior_genes_names*, *sn_genes_names*, *excluded_genes=None*)

Given the list of genes in the spatial data and the list of genes in the single nuclei, identifies the subset of genes included in both lists and returns the corresponding matching indices.

Parameters

- **`prior_genes_names`** (*`sequence`*) – List of gene names in the spatial data.
- **`sn_genes_names`** (*`sequence`*) – List of gene names in the single nuclei data.
- **`excluded_genes`** (*`sequence`*) – Optional. List of genes to be excluded. These genes are excluded even if present in both datasets. If None, no genes are excluded. Default is None.

Returns

mask_prior_indices (list): List of indices for the selected genes in 'prior_genes_names'.
mask_sn_indices (list): List of indices for the selected genes in 'sn_genes_names'. selected_genes (list): List of names of the selected genes.

For each i, selected_genes[i] = prior_genes_names[mask_prior_indices[i]] = sn_genes_names[mask_sn_indices[i].

Return type A tuple (mask_prior_indices, mask_sn_indices, selected_genes), with

tangram.utils.one_hot_encoding

`tangram.utils.`**`one_hot_encoding`**(*l*, *keep_aggregate=False*)

Given a sequence, returns a DataFrame with a column for each unique value in the sequence and a one-hot-encoding.

Parameters

- **`l`** (*`sequence`*) – List to be transformed.
- **`keep_aggregate`** (*`bool`*) – Optional. If True, the output includes an additional column for the original list. Default is False.

Returns

A DataFrame with a column for each unique value in the sequence and a one-hot-encoding, and an additional
column with the input list if 'keep_aggregate' is True. The number of rows are equal to len(l).

tangram.utils.project_cell_annotations

`tangram.utils.`**`project_cell_annotations`**(*adata_map*, *adata_sp*, *annotation='cell_type'*, *threshold=0.5*)

Transfer *annotation* from single cell data onto space.

Parameters

- **`adata_map`** (*`AnnData`*) – cell-by-spot AnnData returned by *train* function.
- **`adata_sp`** (*`AnnData`*) – spatial data used to save the mapping result.
- **`annotation`** (*`str`*) – Optional. Cell annotations matrix with shape (number_cells, number_annotations). Default is 'cell_type'.
- **`threshold`** (*`float`*) – Optional. Valid for using with adata_map.obs['F_out'] from 'constrained' mode mapping. Cell's probability below this threshold will be dropped. Default is 0.5.

Returns None. Update spatial Anndata by creating *obsm tangram_ct_pred* field with a dataframe with spatial prediction for each annotation (number_spots, number_annotations)

tangram.utils.project_genes

`tangram.utils.`**`project_genes`**(*adata_map*, *adata_sc*, *cluster_label=None*, *scale=True*)

Transfer gene expression from the single cell onto space.

Parameters

- **`adata_map`** (*`AnnData`*) – single cell data
- **`adata_sp`** (*`AnnData`*) – gene spatial data
- **`cluster_label`** (*`AnnData`*) – Optional. Should be consistent with the 'cluster_label' argument passed to *map_cells_to_space* function.
- **`scale`** (*`bool`*) – Optional. Should be consistent with the 'scale' argument passed to *map_cells_to_space* function.

Returns spot-by-gene AnnData containing spatial gene expression from the single cell data.

Return type AnnData

tangram.utils.read_pickle

`tangram.utils.`**`read_pickle`**(*filename*)

Helper to read pickle file which may be zipped or not.

Parameters **`filename`** (*`str`*) – A valid string path.

Returns The file object.

tangram.utils.transfer_annotations_prob

`tangram.utils.`**`transfer_annotations_prob`**`(`*mapping_matrix*, *to_transfer*`)`

Transfer cell annotations onto space through a mapping matrix.

Parameters

- **`mapping_matrix`** (*ndarray*) – Mapping matrix with shape (number_cells, number_spots).
- **`to_transfer`** (*ndarray*) – Cell annotations matrix with shape (number_cells, number_annotations).

Returns A matrix of annotations onto space, with shape (number_spots, number_annotations)

tangram.utils.transfer_annotations_prob_filter

`tangram.utils.`**`transfer_annotations_prob_filter`**`(`*mapping_matrix*, *filter*, *to_transfer*`)`

Transfer cell annotations onto space through a mapping matrix and a filter. :param mapping_matrix: Mapping matrix with shape (number_cells, number_spots). :type mapping_matrix: ndarray :param filter: Filter with shape (number_cells,). :type filter: ndarray :param to_transfer: Cell annotations matrix with shape (number_cells, number_annotations). :type to_transfer: ndarray

Returns A matrix of annotations onto space, with shape (number_spots, number_annotations).

3.4 Frequently Asked Questions

Do I need a GPU for running Tangram?

A GPU is not required but is recommended. We run most of our mappings on a single P100 which maps ~50k cells in a few minutes.

How do I choose a list of training genes?

A good way to start is to use the top 1k unique marker genes, stratified across cell types, as training genes. Alternatively, you can map using the whole transcriptome. Ideally, training genes should contain high quality signals: if most training genes are rich in dropouts or obtained with bad RNA probes your mapping will not be accurate.

Do I need cell segmentation for mapping on Visium data?

You do not need to segment cells in your histology for mapping on spatial transcriptomics data (including Visium and Slide-seq). You need, however, cell segmentation if you wish to deconvolve the data (_ie_ deterministically assign a single cell profile to each cell within a spatial voxel).

I run out of memory when I map: what should I do?

Reduce your spatial data in various parts and map each single part. If that is not sufficient, you will need to downsample your single cell data as well.

3.5 Tutorials

3.5.1 Tutorial for mapping data with Tangram

by Tommaso Biancalani biancalt@gene.com and Ziqing Lu luz21@gene.com

- The notebook introduces to mapping single cell data on spatial data using the Tangram method.
- The notebook uses data from mouse brain cortex (different than those adopted in the manuscript).

Last changelog

- June 13th - Tommaso Biancalani biancalt@gene.com

Installation

- Make sure `tangram-sc` is installed via `pip install tangram-sc`.
- Otherwise, edit and uncomment the line starting with `sys.path` specifying the Tangram folder.
- The Python environment needs to install the packages listed in `environment.yml`.

[1]:
```
import os, sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scanpy as sc
import torch
#sys.path.append('/home/exx/git/Tangram/')  # uncomment for local import
import tangram as tg

%load_ext autoreload
%autoreload 2
%matplotlib inline

tg.__version__
```

[1]:
```
'1.0.0'
```

Download the data

- If you have `wget` installed, you can run the following code to automatically download and unzip the data.

[2]:
```
# Skip this cells if data are already downloaded
!wget https://storage.googleapis.com/tommaso-brain-data/tangram_demo/mop_sn_tutorial.
↪h5ad.gz -O data/mop_sn_tutorial.h5ad.gz
!wget https://storage.googleapis.com/tommaso-brain-data/tangram_demo/slideseq_MOp_1217.
↪h5ad.gz -O data/slideseq_MOp_1217.h5ad.gz
```

(continues on next page)

(continued from previous page)

```
!wget https://storage.googleapis.com/tommaso-brain-data/tangram_demo/MOp_markers.csv -O␣
↪data/MOp_markers.csv
!gunzip -f data/mop_sn_tutorial.h5ad.gz
!gunzip -f data/slideseq_MOp_1217.h5ad.gz
```

```
--2021-08-30 14:22:56--  https://storage.googleapis.com/tommaso-brain-data/tangram_demo/
↪mop_sn_tutorial.h5ad.gz
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.14.208, 172.217.14.
↪240, 142.250.69.208, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.14.208|:443...␣
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 474724402 (453M) [application/x-gzip]
Saving to: ‘data/mop_sn_tutorial.h5ad.gz’

data/mop_sn_tutoria 100%[===================>] 452.73M   136MB/s    in 3.3s

2021-08-30 14:22:59 (136 MB/s) - ‘data/mop_sn_tutorial.h5ad.gz’ saved [474724402/
↪474724402]

--2021-08-30 14:23:00--  https://storage.googleapis.com/tommaso-brain-data/tangram_demo/
↪slideseq_MOp_1217.h5ad.gz
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.33.112, 142.251.33.
↪80, 142.250.217.112, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.33.112|:443...␣
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 12614106 (12M) [application/x-gzip]
Saving to: ‘data/slideseq_MOp_1217.h5ad.gz’

data/slideseq_MOp_1 100%[===================>]  12.03M  65.4MB/s    in 0.2s

2021-08-30 14:23:01 (65.4 MB/s) - ‘data/slideseq_MOp_1217.h5ad.gz’ saved [12614106/
↪12614106]

--2021-08-30 14:23:01--  https://storage.googleapis.com/tommaso-brain-data/tangram_demo/
↪MOp_markers.csv
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.33.112, 142.251.33.
↪80, 142.250.217.112, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.33.112|:443...␣
↪connected.
HTTP request sent, awaiting response... 200 OK
Length: 2510 (2.5K) [text/csv]
Saving to: ‘data/MOp_markers.csv’

data/MOp_markers.cs 100%[===================>]   2.45K  --.-KB/s    in 0s

2021-08-30 14:23:02 (27.3 MB/s) - ‘data/MOp_markers.csv’ saved [2510/2510]
```

- If you do not have `wget` installed, manually download data from the links below:
  - snRNA-seq datasets collected from adult mouse cortex: 10Xv3 MOp.

- For spatial data, we will use one coronal slice of Slide-seq2 data (adult mouse brain; MOp area).
- We will map them via a few hundred marker genes, found in literature.
- All datasets need to be unzipped: resulting `h5ad` and `csv` files should be placed in the `data` folder.

Load spatial data

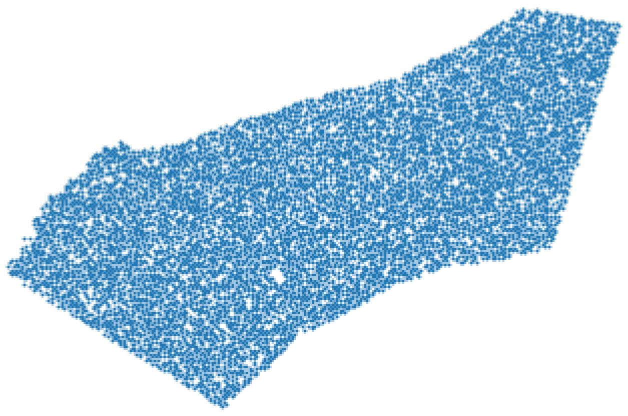
- Spatial data need to be organized as a voxel-by-gene matrix. Here, Slide-seq data contains 9852 spatial voxels, in each of which there are 24518 genes measured.

[2]:
```
path = os.path.join('data', 'slideseq_MOp_1217.h5ad')
ad_sp = sc.read_h5ad(path)
ad_sp
```

[2]:
```
AnnData object with n_obs × n_vars = 9852 × 24518
    obs: 'orig.ident', 'nCount_RNA', 'nFeature_RNA', 'x', 'y'
```

- The voxel coordinates are saved in the fields `obs.x` and `obs.y` which we can use to visualize the spatial ROI. Each "dot" is the center of a 10um voxel.

[3]:
```
xs = ad_sp.obs.x.values
ys = ad_sp.obs.y.values
plt.axis('off')
plt.scatter(xs, ys, s=.7);
plt.gca().invert_yaxis()
```



Single cell data

- By single cell data, we generally mean either scRNAseq or snRNAseq.
- We start by mapping the MOp 10Xv3 dataset, which contains single nuclei collected from a posterior region of the primary motor cortex.
- They are approximately 26k profiled cells with 28k genes.

[4]:
```
path = os.path.join('data','mop_sn_tutorial.h5ad')
ad_sc = sc.read_h5ad(path)
ad_sc
```

[4]:
```
AnnData object with n_obs × n_vars = 26431 × 27742
    obs: 'QC', 'batch', 'class_color', 'class_id', 'class_label', 'cluster_color',
↪'cluster_labels', 'dataset', 'date', 'ident', 'individual', 'nCount_RNA', 'nFeature_RNA
↪', 'nGene', 'nUMI', 'project', 'region', 'species', 'subclass_id', 'subclass_label'
    layers: 'logcounts'
```

- Usually, we work with data in raw count form, especially if the spatial data are in raw count form as well.
- If the data are in integer format, that probably means they are in raw count.

[5]:
```
np.unique(ad_sc.X.toarray()[0, :])
```

[5]:
```
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  33.,
        34.,  36.,  39.,  40.,  43.,  44.,  46.,  47.,  49.,  50.,  53.,
        56.,  57.,  58.,  62.,  68.,  69.,  73.,  77.,  80.,  85.,  86.,
        98., 104., 105., 118., 121., 126., 613.], dtype=float32)
```

- Here, we only do some light pre-processing as library size correction (in scanpy, via `sc.pp.normalize`) to normalize the number of count within each cell to a fixed number.
- Sometimes, we apply more sophisticated pre-processing methods, for example for batch correction, although mapping works great with raw data.
- Ideally, the single cell and spatial datasets, should exhibit signals as similar as possible and the pre-processing pipeline should be finalized to harmonize the signals.

[6]:
```
sc.pp.normalize_total(ad_sc)
```

- It is a good idea to have annotations in the single cell data, as they will be projected on space after we map.
- In this case, cell types are annotated in the `subclass_label` field, for which we plot cell counts.
- Note that cell type proportion should be similar in the two datasets: for example, if `Meis` is a rare cell type in the snRNA-seq then it is expected to be a rare one even in the spatial data as well.

[7]:
```
ad_sc.obs.subclass_label.value_counts()
```

[7]:
```
L5 IT        5623
Oligo        4330
L2/3 IT      3555
L6 CT        3118
Astro        2600
Micro-PVM    1121
Pvalb         972
```

(continues on next page)

(continued from previous page)

```
L6 IT          919
L5 ET          903
L5/6 NP        649
Sst            627
Vip            435
L6b            361
Endo           357
Lamp5          332
VLMC           248
Peri           187
Sncg            94
Name: subclass_label, dtype: int64
```

Prepare to map

- Tangram learns a spatial alignment of the single cell data so that *the gene expression of the aligned single cell data is as similar as possible to that of the spatial data*.
- In doing this, Tangram only looks at a subset genes, specified by the user, called the training genes.
- The choice of the training genes is a delicate step for mapping: they need to bear interesting signals and to be measured with high quality.
- Typically, a good start is to choose 100-1000 top marker genes, evenly stratified across cell types. Sometimes, we also use the entire transcriptome, or perform different mappings using different set of training genes to see how much the result change.
- For this case, we choose 253 marker genes of the MOp area which were curated in a different study.

[8]:
```
df_genes = pd.read_csv('data/MOp_markers.csv', index_col=0)
markers = np.reshape(df_genes.values, (-1, ))
markers = list(markers)
len(markers)
```

[8]:
```
253
```

- We now need to prepare the datasets for mapping by creating `training_genes` field in `uns` dictionary of the two AnnData structures.
- This `training_genes` field contains genes subset on the list of training genes. This field will be used later inside the mapping function to create training datasets.
- Also, the gene order needs to be the same in the datasets. This is because Tangram maps using only gene expression, so the j-th column in each matrix must correspond to the same gene.
- And if data entries of a gene are all zero, this gene will be removed
- This task is performed by the helper `pp_adatas`.

[9]:
```
tg.pp_adatas(ad_sc, ad_sp, genes=markers)
```

```
INFO:root:249 training genes are saved in `uns``training_genes` of both single cell and␣
↪spatial Anndatas.
INFO:root:18000 overlapped genes are saved in `uns``overlap_genes` of both single cell␣
↪and spatial Anndatas.
```

(continues on next page)

(continued from previous page)

```
INFO:root:uniform based density prior is calculated and saved in `obs``uniform_density`␣
↪of the spatial Anndata.
INFO:root:rna count based density prior is calculated and saved in `obs``rna_count_based_
↪density` of the spatial Anndata.
```

- You'll now notice that the two datasets now contain 249 genes, but 253 markers were provided.
- This is because the marker genes need to be shared by both dataset. If a gene is missing, `pp_adatas` will just take it out.
- Finally, the `assert` line below is a good way to ensure that the genes in the `training_genes` field in `uns` are actually ordered in both `AnnData`s.

[10]:
```
ad_sc
```

[10]:
```
AnnData object with n_obs × n_vars = 26431 × 26496
    obs: 'QC', 'batch', 'class_color', 'class_id', 'class_label', 'cluster_color',
↪'cluster_labels', 'dataset', 'date', 'ident', 'individual', 'nCount_RNA', 'nFeature_RNA
↪', 'nGene', 'nUMI', 'project', 'region', 'species', 'subclass_id', 'subclass_label'
    var: 'n_cells'
    uns: 'training_genes', 'overlap_genes'
    layers: 'logcounts'
```

[11]:
```
ad_sp
```

[11]:
```
AnnData object with n_obs × n_vars = 9852 × 20864
    obs: 'orig.ident', 'nCount_RNA', 'nFeature_RNA', 'x', 'y', 'uniform_density', 'rna_
↪count_based_density'
    var: 'n_cells'
    uns: 'training_genes', 'overlap_genes'
```

[12]:
```
assert ad_sc.uns['training_genes'] == ad_sp.uns['training_genes']
```

Map

- We can now train the model (*ie* map the single cell data onto space).
- Mapping should be interrupted after the score plateaus,which can be controlled by passing the `num_epochs` parameter.
- The score measures the similarity between the gene expression of the mapped cells vs spatial data: higher score means better mapping.
- Note that we obtained excellent mapping even if Tangram converges to a low scores (the typical case is when the spatial data are very sparse): we use the score merely to assess convergence.
- If you are running Tangram with a GPU, uncomment `device=cuda:0` and comment the line `device=cpu`. On a MacBook Pro 2018, it takes ~1h to run. On a P100 GPU it should be done in a few minutes.
- For this basic mapping, we do not use regularizers. More sophisticated loss functions can be used using the Tangram library (refer to manuscript or dive into the code). For example, you can pass your `density_prior` with the hyperparameter `lambda_d` to regularize the spatial density of cells. Currently `uniform`, `rna_count_based` and customized input array are supported for `density_prior` argument.

- Instead of mapping single cells, we can “average” the cells within a cluster and map the averaged cells instead, which drammatically improves performances. This suggestion was proposed by Sten Linnarsson. To activate this mode, select `mode='clusters'` and pass the annotation field to `cluster_label`.

```
[13]: ad_map = tg.map_cells_to_space(
          adata_sc=ad_sc,
          adata_sp=ad_sp,
          device='cpu',
          # device='cuda:0',
      )
```

```
INFO:root:Allocate tensors for mapping.
INFO:root:Begin training with 249 genes and None density_prior in cells mode...
INFO:root:Printing scores every 100 epochs.
```

```
Score: 0.103
Score: 0.802
Score: 0.819
Score: 0.822
Score: 0.824
Score: 0.825
Score: 0.826
Score: 0.826
Score: 0.827
Score: 0.827
```

```
INFO:root:Saving results..
```

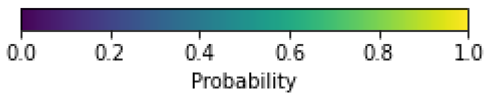
- The mapping results are stored in the returned `AnnData` structure, saved as `ad_map`, structured as following:
  - The cell-by-spot matrix `X` contains the probability of cell i to be in spot j.
  - The `obs` dataframe contains the metadata of the single cells.
  - The `var` dataframe contains the metadata of the spatial data.
  - The `uns` dictionary contains a dataframe with various information about the training genes (saved ad `train_genes_df`).
- We can now save the mapping results for post-analysis.

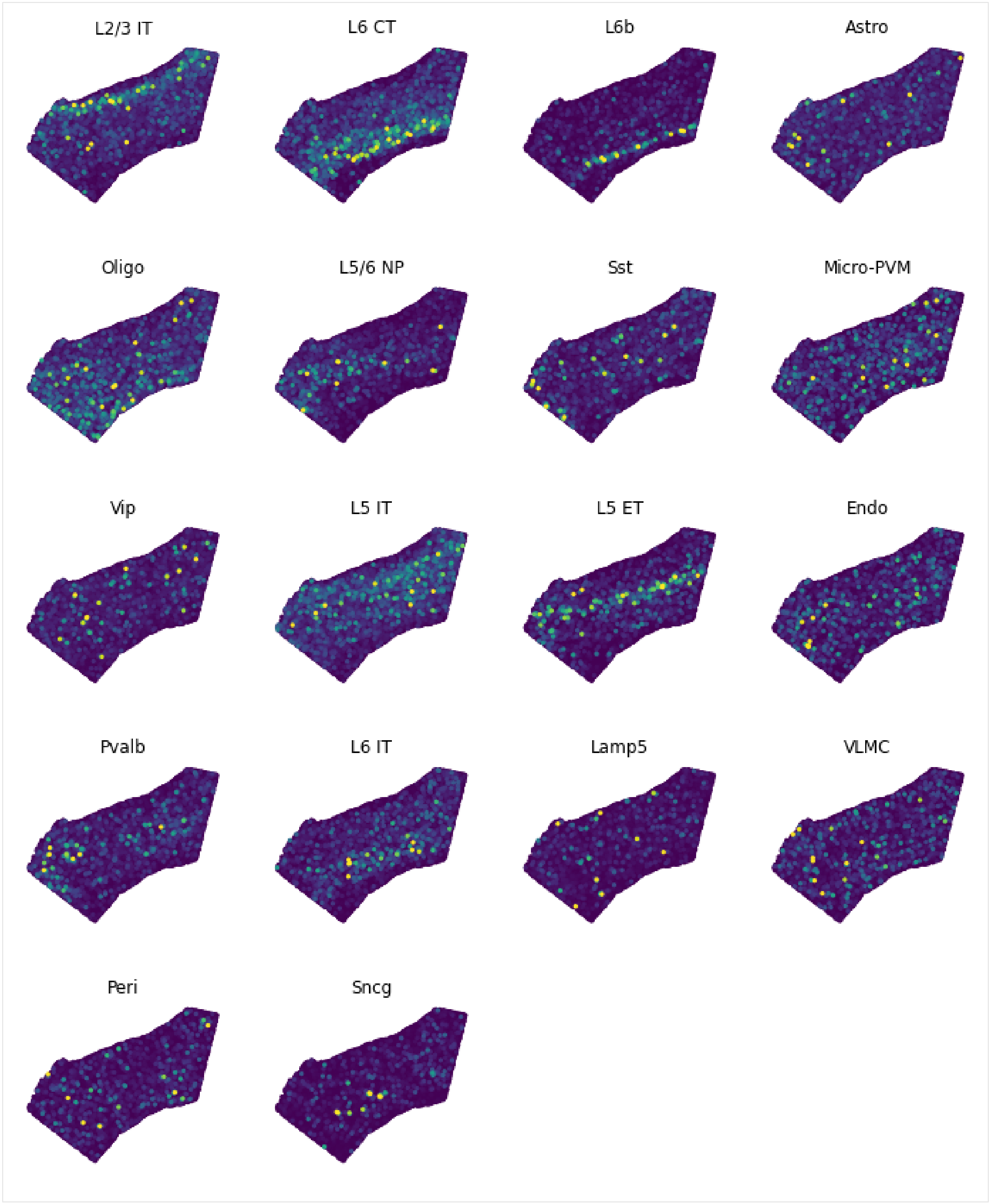
Analysis

- The most common application for mapping single cell data onto space is to transfer the cell type annotations onto space.
- This is dona via `plot_cell_annotation`, which visualizes spatial probability maps of the `annotation` in the `obs` dataframe (here, the `subclass_label` field). You can set `robust` argument to `True` and at the same time set the `perc` argument to set the range to the colormap, which would help remove outliers.
- The following plots recover cortical layers of excitatory neurons and sparse patterns of glia cells. The boundaries of the cortex are defined by layer 6b (cell type *L6b*) and oligodendrocytes are found concentrated into sub-cortical region, as expected.
- Yet, the *VLMC* cell type patterns does not seem correct: *VLMC* cells are clustered in the first cortical layer, whereas here are sparse in the ROI. This usually means that either (1) we have not used good marker genes for *VLMC* cells in our training genes (2) the present marker genes are very sparse in the spatial data, therefore they don’t contain good mapping signal.

[15]:
```
tg.plot_cell_annotation(ad_map, ad_sp, annotation='subclass_label', nrows=5, ncols=4,␣
↪robust=True, perc=0.05)
```

```
INFO:root:spatial prediction dataframe is saved in `obsm` `tangram_ct_pred` of the␣
↪spatial AnnData.
```



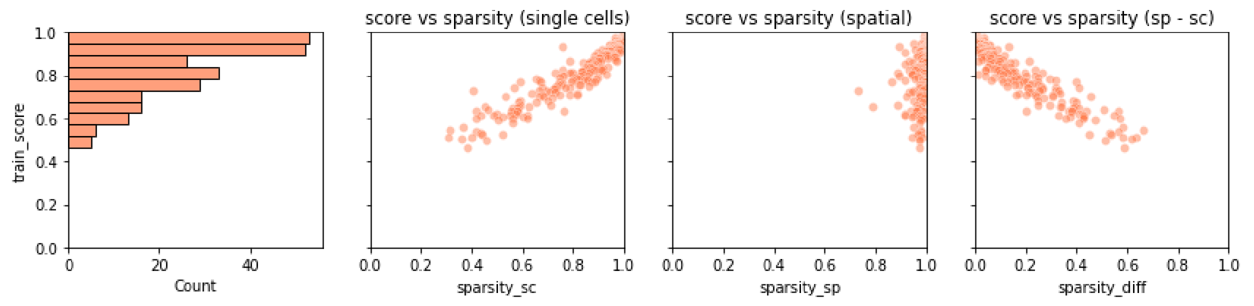


- Let's try to get a deeper sense of how good this mapping is. A good helper is `plot_training_scores` which gives us four panels:
  - The first panels is a histogram of the simlarity score for each training gene. Most genes are mapped with very high similarity (> .9) although few of them have score ~.5. We would like to understand why for these

genes the score is lower.

- The second panel shows that there is a neat correlation between the training score of a gene (y-axis) and the sparsity of that gene in the snRNA-seq data (x-axis). Each dot is a training gene. The trend is that the sparser the gene the higher the score: this usually happens because very sparse gene are easier to map, as their pattern is matched by placing a few "jackpot cells" in the right spots.
- The third panel is similar to the second one, but contains the gene sparsity of the spatial data. Spatial data are usually more sparse than single cell data, a discrepancy which is often responsible for low quality mapping.
- In the last panel, we show the training scores as a function of the difference in sparsity between the dataset. For genes with comparable sparsity, the mapped gene expression is very similar to that in the spatial data. However, if a gene is quite sparse in one dataset (typically, the spatial data) but not in other, the mapping score is lower. This occurs as Tangram cannot properly matched the gene pattern because of inconsistent amount of dropouts between the datasets.

```
[16]: tg.plot_training_scores(ad_map, bins=10, alpha=.5)
```



- Although the above plots give us a summary of scores at single-gene level, we would need to know *which* are the genes are mapped with low scores.
- These information can be access from the dataframe `.uns['train_genes_df']` from the mapping results; this is the dataframe used to build the four plots above.

```
[17]: ad_map.uns['train_genes_df']
```

```
[17]:                train_score  sparsity_sc  sparsity_sp  sparsity_diff
      igf2              0.999628     0.999924     0.994011      -0.005913
      chodl             0.997236     0.999016     0.999086       0.000070
      5031425f14rik     0.995950     0.998789     0.999594       0.000805
      car3              0.995266     0.999016     0.999695       0.000679
      scgn              0.994697     0.999205     0.999898       0.000693
      ...                    ...          ...          ...            ...
      5730522e02rik     0.514366     0.400401     0.984572       0.584171
      rgs6              0.508304     0.305172     0.941941       0.636769
      ptprk             0.507358     0.357800     0.974218       0.616419
      satb2             0.495069     0.455904     0.969549       0.513645
      cdh12             0.464210     0.384889     0.972594       0.587705

      [249 rows x 4 columns]
```

- We want to inspect gene expression of training genes mapped with low scores, to understand the quality of mapping.
- First, we need to generate "new spatial data" using the mapped single cell: this is done via `project_genes`.
- The function accepts as input a mapping (`adata_map`) and corresponding single cell data (`adata_sc`).

- The result is a voxel-by-gene `AnnData`, formally similar to `ad_sp`, but containing gene expression from the mapped single cell data rather than Slide-seq.

```
[18]: ad_ge = tg.project_genes(adata_map=ad_map, adata_sc=ad_sc)
      ad_ge
```

```
[18]: AnnData object with n_obs × n_vars = 9852 × 26496
          obs: 'orig.ident', 'nCount_RNA', 'nFeature_RNA', 'x', 'y', 'uniform_density', 'rna_
      ↪count_based_density'
          var: 'n_cells', 'sparsity', 'is_training'
          uns: 'training_genes', 'overlap_genes'
```

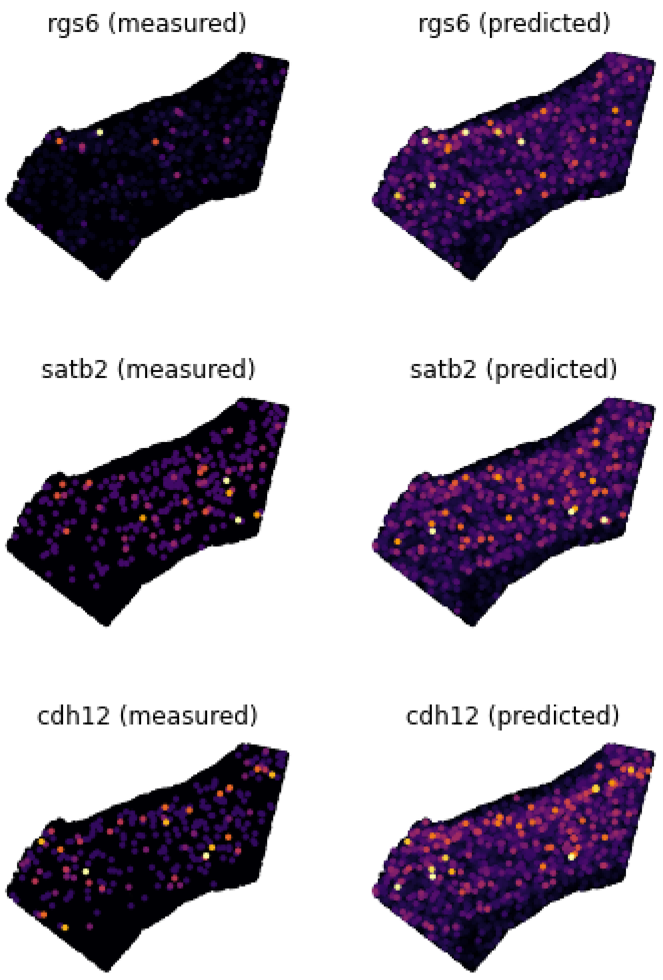
- We now choose a few training genes mapped with low score.

```
[19]: genes = ['rgs6', 'satb2',  'cdh12']
      ad_map.uns['train_genes_df'].loc[genes]
```

```
[19]:        train_score  sparsity_sc  sparsity_sp  sparsity_diff
      rgs6      0.508304     0.305172     0.941941       0.636769
      satb2     0.495069     0.455904     0.969549       0.513645
      cdh12     0.464210     0.384889     0.972594       0.587705
```

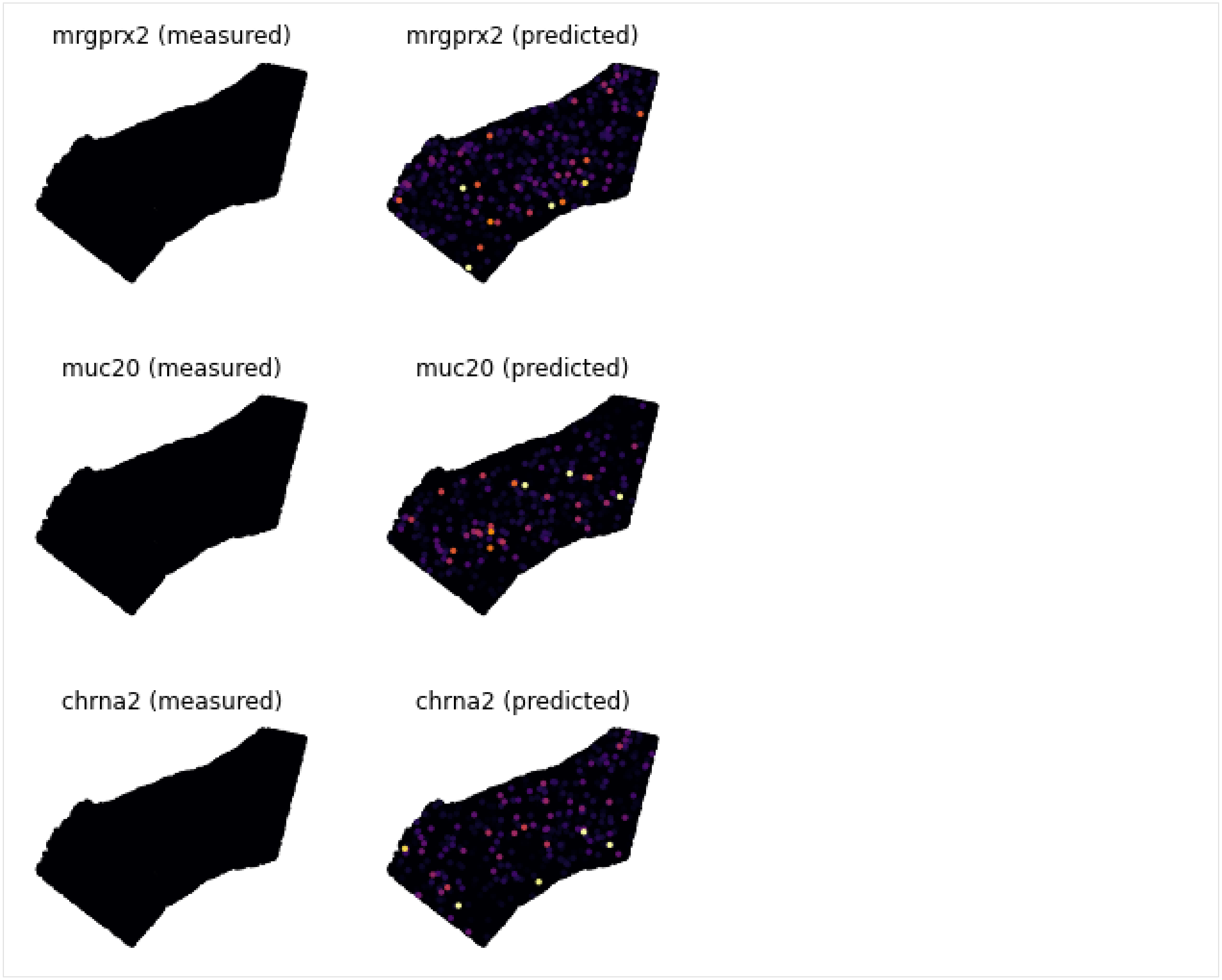
- To visualize gene patterns, we use the helper `plot_genes`. This function accepts two voxel-by-gene `AnnData`: the actual spatial data (`adata_measured`), and a Tangram spatial prediction (`adata_predicted`). The function returns gene expression maps from the two spatial `AnnData` on the genes `genes`.
- As expected, the predited gene expression is less sparse albeit the main patterns are captured. For these genes, we trust more the mapped gene patterns, as Tangram "corrects" gene expression by aligning in space less sparse data.

```
[23]: tg.plot_genes(genes, adata_measured=ad_sp, adata_predicted=ad_ge, robust=True, perc=0.02)
```



- An even stronger example is found in genes that are not detected in the spatial data, but are detected in the single cell data. They are removed before training with `pp_adatas` function. But tangram could still generate insight on how the spatial patterns look like.

```
[24]: genes=['mrgprx2', 'muc20', 'chrna2']
tg.plot_genes(genes, adata_measured=ad_sp, adata_predicted=ad_ge, robust=True, perc=0.02)
```



- So far, we only inspected genes used to align the data (training genes), but the mapped single cell data, `ad_ge` contains the whole transcriptome. That includes more than 26k test genes.

```
[25]: (ad_ge.var.is_training == False).sum()
```

```
[25]: 26247
```

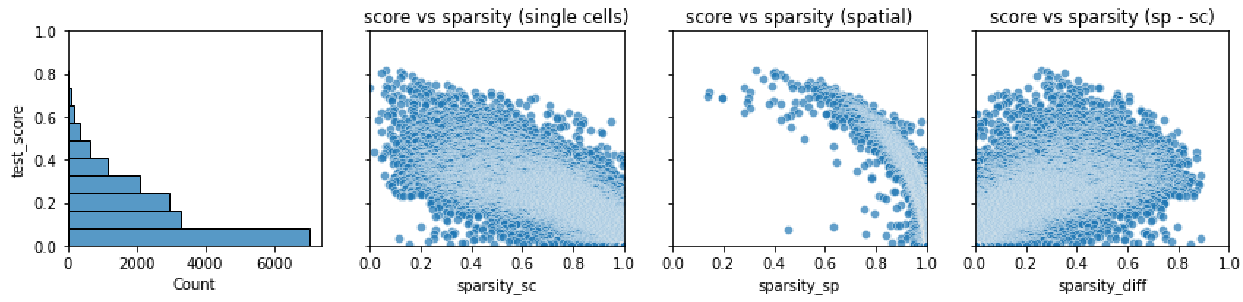
- We can use `plot_genes` to inspect gene expression of non training genes. This is an essential step as prediction of gene expression is the how we validate mapping.
- Before doing that, it is convenient to compute the similarity scores of all genes, which can be done by `compare_spatial_geneexp`. This function accepts two spatial `AnnDatas` (*ie* voxel-by-gene), and returns a dataframe with simlarity scores for all genes. Training genes are flagged by the Boolean field `is_training`.
- If we also pass single cell `AnnData` to `compare_spatial_geneexp` function like below, a dataframe with additional sparsity columns - sparsity_sc (single cell data sparsity) and sparsity_diff (spatial data sparsity - single cell data sparsity) will return. This is required if we want to call `plot_test_scores` function later with the returned datafrme from `compare_spatial_geneexp` function.

```
[26]: df_all_genes = tg.compare_spatial_geneexp(ad_ge, ad_sp, ad_sc)
      df_all_genes
```

[26]:
```
                       score  is_training  sparsity_sp  sparsity_sc  \
igf2           9.996283e-01         True     0.994011     0.999924
chodl          9.972357e-01         True     0.999086     0.999016
5031425f14rik  9.959502e-01         True     0.999594     0.998789
car3           9.952664e-01         True     0.999695     0.999016
scgn           9.946969e-01         True     0.999898     0.999205
...                     ...          ...          ...          ...
cyp1a2         2.775963e-08        False     0.999898     0.999962
ccdc185        1.200085e-08        False     0.999898     0.999962
alox12e        9.068886e-09        False     0.999898     0.999962
cd69           4.023093e-09        False     0.999898     0.999962
ms4a15         1.751993e-09        False     0.999898     0.999962

               sparsity_diff
igf2               -0.005913
chodl               0.000070
5031425f14rik       0.000805
car3                0.000679
scgn                0.000693
...                      ...
cyp1a2             -0.000064
ccdc185            -0.000064
alox12e            -0.000064
cd69               -0.000064
ms4a15             -0.000064

[18000 rows x 5 columns]
```

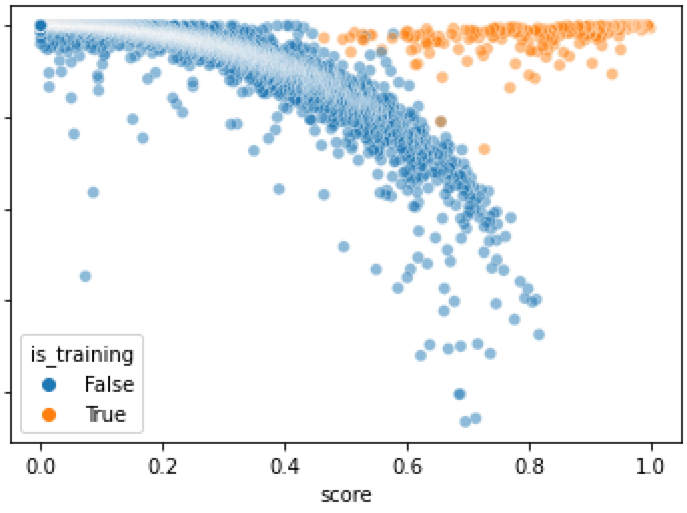
- The plot below give us a summary of scores at single-gene level for test genes

[27]:
```
tg.plot_test_scores(df_all_genes)
```



- Let's plot the scores of the test genes and see how they compare to the training genes. Following the strategy in the previous plots, we visualize the scores as a function of the sparsity of the spatial data.
- (We have not wrapped this call into a function yet).

[28]:
```
sns.scatterplot(data=df_all_genes, x='score', y='sparsity_sp', hue='is_training', alpha=.
↪5);
```



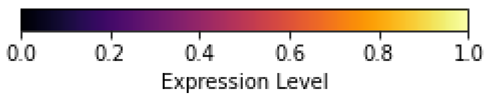
- Again, sparser genes in the spatial data are predicted with low scores, which is due to the presence of dropouts in the spatial data.
- Let's choose a few test genes with varied scores and compared predictions vs measured gene expression.

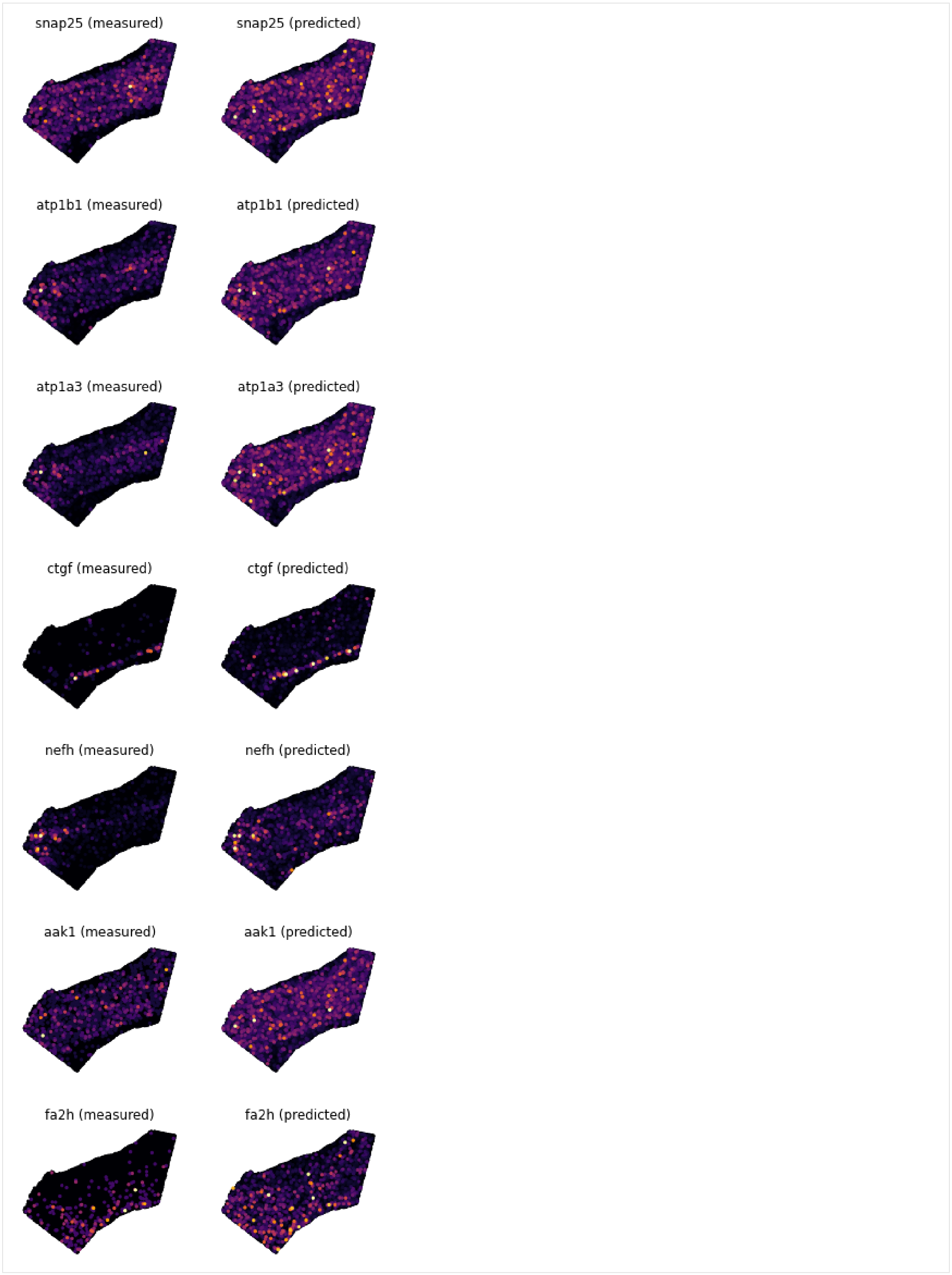
[29]:
```
genes = ['snap25', 'atp1b1', 'atp1a3', 'ctgf', 'nefh', 'aak1', 'fa2h', ]
df_all_genes.loc[genes]
```

[29]:
```
           score  is_training  sparsity_sp  sparsity_sc  sparsity_diff
snap25  0.812584        False     0.402253     0.120048       0.282205
atp1b1  0.770782        False     0.579984     0.175778       0.404205
atp1a3  0.697557        False     0.658343     0.319587       0.338757
ctgf    0.577751        False     0.981628     0.948243       0.033386
nefh    0.522629        False     0.909156     0.916083      -0.006928
aak1    0.492562        False     0.868047     0.179713       0.688334
fa2h    0.342086        False     0.972493     0.860845       0.111648
```

- We can use again `plot_genes` to visualize gene patterns.
- Interestingly, the agreement for genes `Atp1b1` or `Apt1a3`, seems less good than that for `Ctgf` and `Nefh`, despite the scores are higher for the former genes. This is because even though the latter gene patterns are localized correctly, their expression values are not so well correlated (for instance, in `Ctgf` the "bright yellow spot" is in different part of layer 6b). In contrast, for `Atpb1` the gene expression pattern is largely recover, even though the overall gene expression in the spatial data is more dim.

[30]:
```
tg.plot_genes(genes, adata_measured=ad_sp, adata_predicted=ad_ge, robust=True, perc=0.02)
```




snap25 (measured)
snap25 (predicted)
atp1b1 (measured)
atp1b1 (predicted)
atp1a3 (measured)
atp1a3 (predicted)
ctgf (measured)
ctgf (predicted)
nefh (measured)
nefh (predicted)
aak1 (measured)
aak1 (predicted)
fa2h (measured)
fa2h (predicted)

Leave-One-Out Cross Validation (LOOCV)

- If number of genes is small, Leave-One-Out cross validation (LOOCV) is supported in Tangram to evaluate mapping performance.
- LOOCV supported by Tangram:
  - Assume the number of genes we have in the dataset is N.
  - LOOCV would iterate over and map on the genes dataset N times.
  - Each time it hold out one gene as test gene (1 test gene) and trains on the rest of all genes (N-1 training genes).
  - After all trainings are done, average test/train score will be computed to evaluate the mapping performance.
- Assume all genes we have is the training genes in the example above. Here we demo the LOOCV mapping at cluster level.
- Restart the kernel and load single cell, spatial and gene markers data
- Run `pp_adatas` to prepare data for mapping

```
[ ]: import os, sys
     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import scanpy as sc
     import torch
     import tangram as tg
```

```
[ ]: path = os.path.join('data', 'slideseq_MOp_1217.h5ad')
     ad_sp = sc.read_h5ad(path)

     path = os.path.join('data','mop_sn_tutorial.h5ad')
     ad_sc = sc.read_h5ad(path)
     sc.pp.normalize_total(ad_sc)

     df_genes = pd.read_csv('data/MOp_markers.csv', index_col=0)
     markers = np.reshape(df_genes.values, (-1, ))
     markers = list(markers)

     tg.pp_adatas(ad_sc, ad_sp, genes=markers)
```

```
INFO:root:249 training genes are saved in `uns``training_genes` of both single cell and␣
↪spatial Anndatas.
INFO:root:18000 overlapped genes are saved in `uns``overlap_genes` of both single cell␣
↪and spatial Anndatas.
INFO:root:uniform based density prior is calculated and saved in `obs``uniform_density`␣
↪of the spatial Anndata.
INFO:root:rna count based density prior is calculated and saved in `obs``rna_count_based_
↪density` of the spatial Anndata.
```

```
[ ]: cv_dict, ad_ge_cv, df = tg.cross_val(ad_sc,
                                         ad_sp,
                                         device='cuda:0',
                                         mode='clusters',
                                         cv_mode='loo',
                                         num_epochs=1000,
                                         cluster_label='subclass_label',
                                         return_gene_pred=True,
                                         verbose=False,
                                         )
```

```
100%|| 249/249 [23:21<00:00,  5.63s/it]
```

```
cv avg test score 0.185
cv avg train score 0.296
```

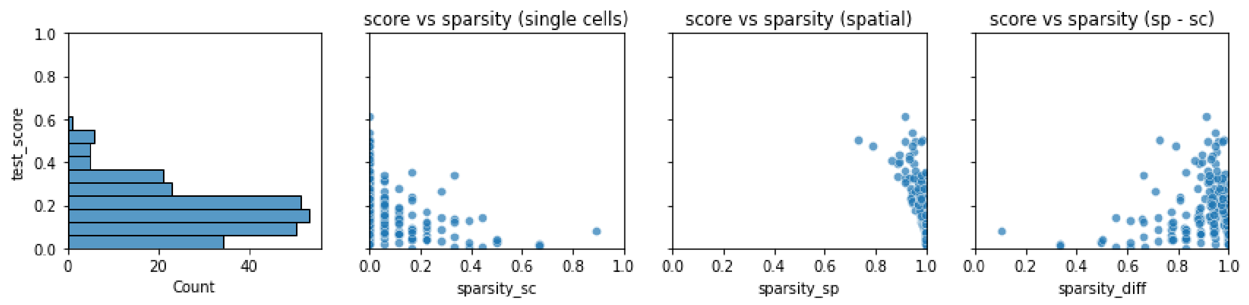
- `cross_val` function will return `cv_dict` and `ad_ge_cv` and `df_test_genes` in LOOCV mode. `cv_dict` contains the average score for cross validation, `ad_ge_cv` stores the predicted expression value for each gene, and `df_test_genes` contains scores and sparsity for each test genes.

```
[ ]: cv_dict
```

```
{'avg_test_score': 0.18502994, 'avg_train_score': 0.2960305045168084}
```

- We can use `plot_test_scores` to display an overview of the cross validation test scores of each gene vs. sparsity.

```
[ ]: tg.plot_test_scores(df, bins=10, alpha=.7)
```



- Now, let's compare a few genes between their ground truth and cross-validation predicted spatial pattern by calling the function `plot_genes`

```
[ ]: ad_ge_cv.var.sort_values(by='test_score', ascending=False)
```

```
               test_score
gad1             0.612824
gad2             0.538229
slc17a7          0.507557
vtn              0.503726
pvalb            0.498350
...                   ...
5031425f14rik    0.015658
prok2            0.008798
teddm3           0.003808
```

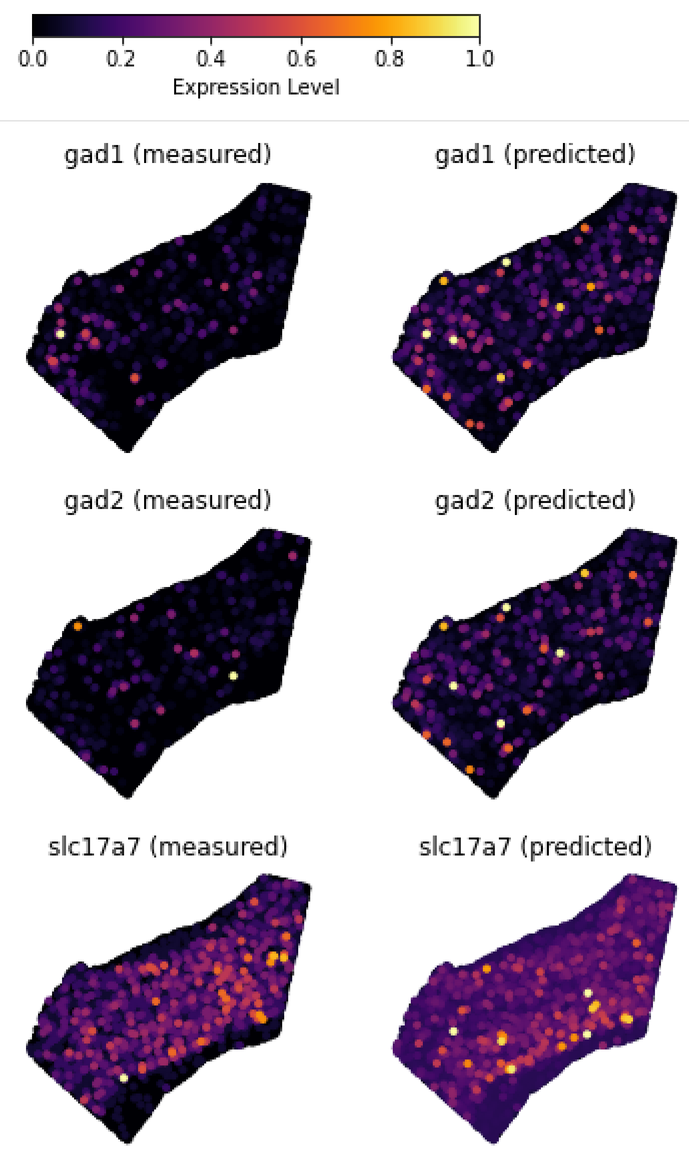
(continues on next page)

(continued from previous page)

```
scgn             0.002912
dnase1l3         0.000634

[249 rows x 1 columns]
```

[ ]:
```
ranked_genes = list(ad_ge_cv.var.sort_values(by='test_score', ascending=False).index.
↪values)
top_genes = ranked_genes[:3]
bottom_genes = ranked_genes[-3:]
```

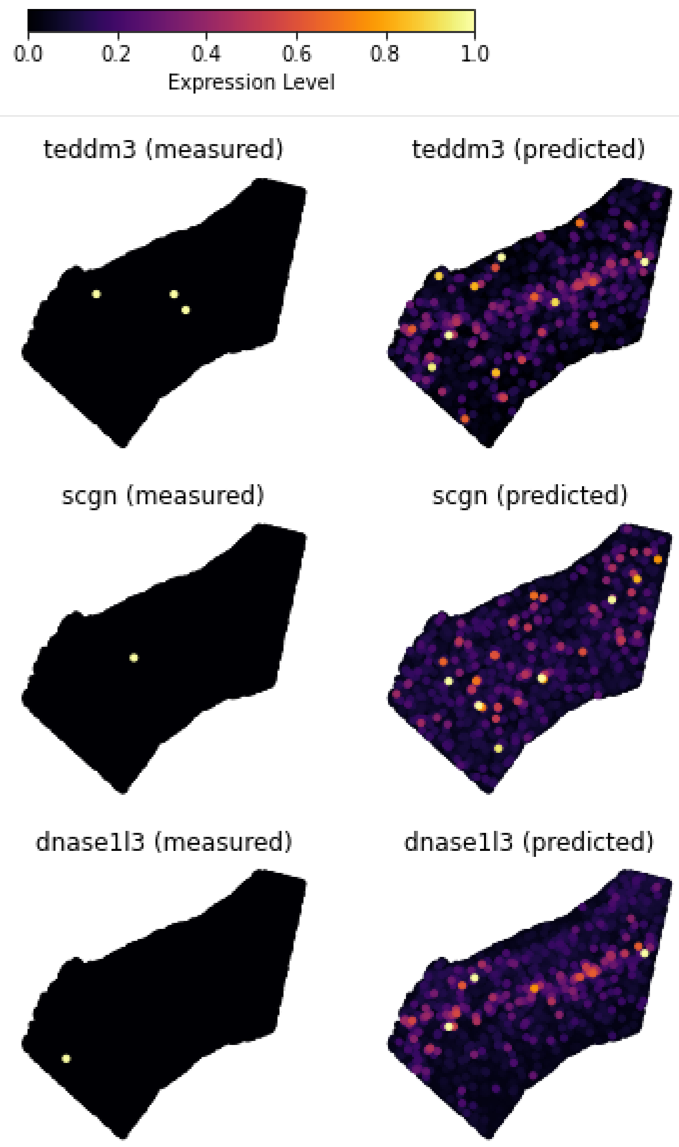
[ ]:
```
tg.plot_genes(genes=top_genes, adata_measured=ad_sp, adata_predicted=ad_ge_cv, s=10,␣
↪robust=True, perc=0.02)
```



[ ]:
```
tg.plot_genes(genes=bottom_genes, adata_measured=ad_sp, adata_predicted=ad_ge_cv, s=10,␣
↪robust=True, perc=0.02)
```

(continues on next page)

(continued from previous page)



3.5.2 Tutorial for spatial mapping using *Tangram*

- by Ziqing Lu luz21@gene.com and Tommaso Biancalani biancalt@gene.com.
- Last update: August 16th 2021

What is *Tangram*?

Tangram is a method for mapping single-cell (or single-nucleus) gene expression data onto spatial gene expression data. *Tangram* takes as input a single-cell dataset and a spatial dataset, collected from the same anatomical region/tissue type. Via integration, *Tangram* creates new spatial data by aligning the scRNAseq profiles in space. This allows to project every annotation in the scRNAseq (*e.g.* cell types, program usage) on space.

What do I use *Tangram* for?

The most common application of *Tangram* is to resolve cell types in space. Another usage is to correct gene expression from spatial data: as scRNA-seq data are less prone to dropout than (*e.g.*) Visium or Slide-seq, the "new" spatial data generated by *Tangram* resolve many more genes. As a result, we can visualize program usage in space, which can be used for ligand-receptor pair discovery or, more generally, cell-cell communication mechanisms. If cell segmentation is available, *Tangram* can be also used for deconvolution of spatial data. If your single cell are multimodal, *Tangram* can be used to spatially resolve other modalities, such as chromatin accessibility.

Frequently Asked Questions about *Tangram*

How is *Tangram* different, than all the other deconvolution/mapping method?

- Validation. Most methods "validate" mappings by looking at known patterns or proportion of cell types. These are good sanity checks, but are hardly useful when mapping is used for discovery. In *Tangram*, mappings are validated by inspective the predictions of holdout genes (test transcriptome).

My scRNAseq/spatial data come from different samples. Can I still use *Tangram*?

- Yes. There is a clever variation invented by Sten Linnarsson, which consists of mapping *average* cells of a certain cell type, rather than single cells. This method is much faster, and smooths out variation in biological signal from different samples via averaging. However, it requires annotated scRNA-seq, sacrifices resolving biological variability at single-cell level. To map this way, pass `mode=cluster`.

Does *Tangram* only work on mouse brain data?

- **No.** The original manuscript focused on mouse brain data b/c was funded by BICCN. We subsequently used *Tangram* for mapping lung, kidney and cancer tissue. If mapping doesn't work for your case, that is hardly due to the complexity of the tissue.

Why doesn't *Tangram* have hypotheses on the underlying model?

- Most models used in biology are probabilistic: they assume that data are generated according to a certain probability distribution, hence the hypothesis. But *Tangram* doesn't work that way: the hypothesis is that scRNA-seq and spatial data are generated with the same process (*i.e.* same biology) regardless of the process.

Where do I learn more about *Tangram*?

- Check out our documentation for learning more about the method, or our GitHub repo for the latest version of the code. Tangram has been released in **:cite:`tangram`**.

Setting up

Tangram is based on pytorch, scanpy and (optionally but highly-recommended) squidpy - this tutorial is designed to work with squidy. You can also check this tutorial, prior to integration with squidpy.

- To run the notebook locally, create a conda environment as `conda env create -f tangram_environment.yml` using our YAML file.
- This notebook is based on squidpy v1.1.0.

[2]:
```
import scanpy as sc
import squidpy as sq
import numpy as np
import pandas as pd
from anndata import AnnData
import pathlib
import matplotlib.pyplot as plt
import matplotlib as mpl
import skimage
import seaborn as sns
import tangram as tg

sc.logging.print_header()
print(f"squidpy=={sq.__version__}")

%load_ext autoreload
%autoreload 2
%matplotlib inline
```

```
scanpy==1.8.1 anndata==0.7.6 umap==0.5.1 numpy==1.20.0 scipy==1.5.2 pandas==1.2.0 scikit-
↪learn==0.24.2 statsmodels==0.12.2 python-igraph==0.9.6 pynndescent==0.5.4
squidpy==1.1.0
```

Loading datasets

Load public data available in Squidpy, from mouse brain cortex. Single cell data are stored in `adata_sc`. Spatial data, in `adata_st`.

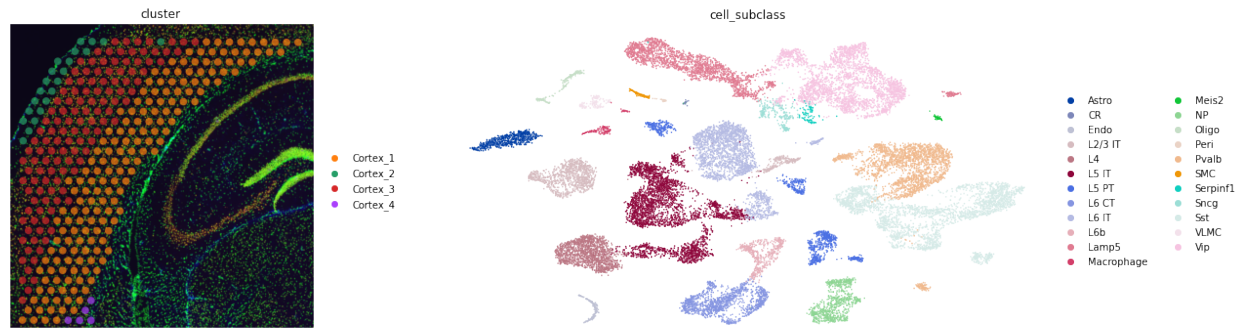
[3]:
```
adata_st = sq.datasets.visium_fluo_adata_crop()
adata_st = adata_st[
    adata_st.obs.cluster.isin([f"Cortex_{i}" for i in np.arange(1, 5)])
].copy()
img = sq.datasets.visium_fluo_image_crop()

adata_sc = sq.datasets.sc_mouse_cortex()
```

```
0%|          | 0.00/65.5M [00:00<?, ?B/s]
0%|          | 0.00/303M [00:00<?, ?B/s]
0%|          | 0.00/3.03G [00:00<?, ?B/s]
```

We subset the crop of the mouse brain to only contain clusters of the brain cortex. The pre-processed single cell dataset was taken from **:cite:`tasic2018shared`** and pre-processed with standard scanpy functions.

Let's visualize both spatial and single-cell datasets.

```
[5]: fig, axs = plt.subplots(1, 2, figsize=(20, 5))
sc.pl.spatial(
    adata_st, color="cluster", alpha=0.7, frameon=False, show=False, ax=axs[0]
)
sc.pl.umap(
    adata_sc, color="cell_subclass", size=10, frameon=False, show=False, ax=axs[1]
)
plt.tight_layout()
```



Tangram learns a spatial alignment of the single cell data by looking at a subset of genes, specified by the user, called the training genes. Training genes need to bear interesting signal and to be measured with high quality. Typically, we choose the training genes are 100-1000 differentially expressedx genes, stratified across cell types. Sometimes, we also use the entire transcriptome, or perform different mappings using different set of training genes to see how much the result change.

Tangram fits the scRNA-seq profiles on space using a custom loss function based on cosine similarity. The method is summarized in the sketch below:

figures/how_tangram_works.png

Pre-processing

For this case, we use 1401 marker genes as training genes.

```
[6]: sc.tl.rank_genes_groups(adata_sc, groupby="cell_subclass", use_raw=False)
markers_df = pd.DataFrame(adata_sc.uns["rank_genes_groups"]["names"]).iloc[0:100, :]
markers = list(np.unique(markers_df.melt().value.values))
len(markers)
```

```
WARNING: Default of the method has been changed to 't-test' from 't-test_overestim_var'
```

```
[6]: 1401
```

We prepares the data using `pp_adatas`, which does the following: - Takes a list of genes from user via the `genes` argument. These genes are used as training genes. - Annotates training genes under the `training_genes` field, in `uns` dictionary, of each AnnData. - Ensure consistent gene order in the datasets (*Tangram* requires that the the j-th column in each matrix correspond to the same gene). - If the counts for a gene are all zeros in one of the datasets, the gene is removed from the training genes. - If a gene is not present in both datasets, the gene is removed from the training genes.

```
[7]: tg.pp_adatas(adata_sc, adata_st, genes=markers)
```

```
INFO:root:1280 training genes are saved in `uns``training_genes` of both single cell and␣
↪spatial Anndatas.
INFO:root:14785 overlapped genes are saved in `uns``overlap_genes` of both single cell␣
↪and spatial Anndatas.
INFO:root:uniform based density prior is calculated and saved in `obs``uniform_density`␣
↪of the spatial Anndata.
INFO:root:rna count based density prior is calculated and saved in `obs``rna_count_based_
↪density` of the spatial Anndata.
```

Two datasets contain 1280 training genes of the 1401 originally provided, as some training genes have been removed.

Find alignment

To find the optimal spatial alignment for scRNA-seq profiles, we use the `map_cells_to_space` function: - The function maps iteratively as specified by `num_epochs`. We typically interrupt mapping after the score plateaus. - The score measures the similarity between the gene expression of the mapped cells vs spatial data on the training genes. - The default mapping mode is `mode='cells'`, which is recommended to run on a GPU. - Alternatively, one can specify `mode='clusters'` which averages the single cells beloning to the same cluster (pass annotations via `cluster_label`). This is faster, and is our chioce when scRNAseq and spatial data come from different specimens. - If you wish to run Tangram with a GPU, set `device=cuda:0` otherwise use the set `device=cpu`. - `density_prior` specifies the cell density within each spatial voxel. Use `uniform` if the spatial voxels are at single cell resolution (*ie* MERFISH). The default value, `rna_count_based`, assumes that cell density is proportional to the number of RNA molecules.

```
[8]: ad_map = tg.map_cells_to_space(adata_sc, adata_st,
    mode="cells",
#     mode="clusters",
#     cluster_label='cell_subclass',  # .obs field w cell types
    density_prior='rna_count_based',
    num_epochs=500,
    # device="cuda:0",
    device='cpu',
)
```

```
INFO:root:Allocate tensors for mapping.
INFO:root:Begin training with 1280 genes and rna_count_based density_prior in cells␣
↪mode...
INFO:root:Printing scores every 100 epochs.
```

```
Score: 0.613, KL reg: 0.001
Score: 0.733, KL reg: 0.000
Score: 0.736, KL reg: 0.000
Score: 0.737, KL reg: 0.000
Score: 0.737, KL reg: 0.000
```

```
INFO:root:Saving results..
```

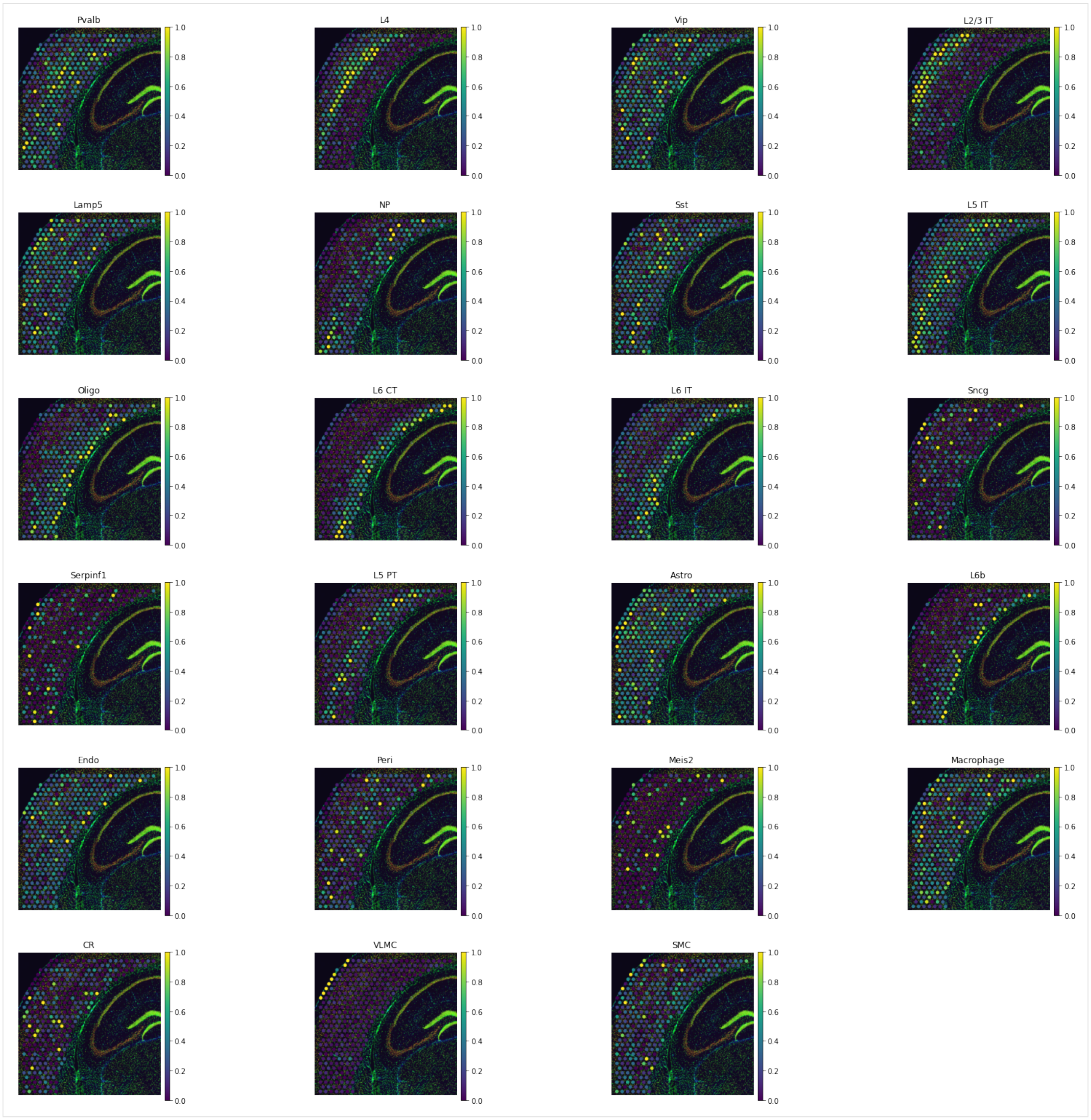
The mapping results are stored in the returned `AnnData` structure, saved as `ad_map`, structured as following: - The cell-by-spot matrix `X` contains the probability of cell `i` to be in spot `j`. - The `obs` dataframe contains the metadata of the single cells. - The `var` dataframe contains the metadata of the spatial data. - The `uns` dictionary contains a dataframe with various information about the training genes (saved as `train_genes_df`).

Cell type maps

To visualize cell types in space, we invoke `project_cell_annotation` to transfer the `annotation` from the mapping to space. We can then call `plot_cell_annotation` to visualize it. You can set the `perc` argument to set the range to the colormap, which would help remove outliers.

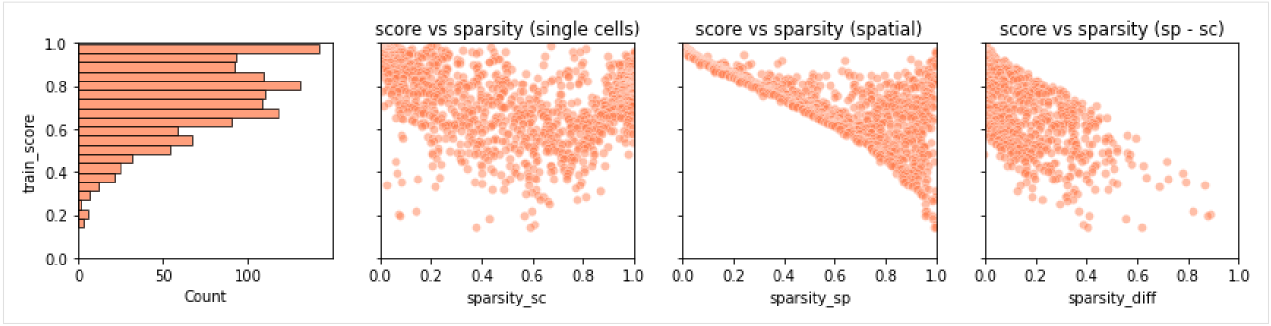
[10]:
```
tg.project_cell_annotations(ad_map, adata_st, annotation="cell_subclass")
annotation_list = list(pd.unique(adata_sc.obs['cell_subclass']))
tg.plot_cell_annotation_sc(adata_st, annotation_list, perc=0.02)
```

```
INFO:root:spatial prediction dataframe is saved in `obsm` `tangram_ct_pred` of the␣
↪spatial AnnData.
```



The first way to get a sense if mapping was successful is to look for known cell type patterns. To get a deeper sense, we can use the helper `plot_training_scores` which gives us four panels:

[11]:
```
tg.plot_training_scores(ad_map, bins=20, alpha=.5)
```



- The first panel is a histogram of the simlarity scores for each training gene.
- In the second panel, each dot is a training gene and we can observe the training score (y-axis) and the sparsity in the scRNA-seq data (x-axis) of each gene.
- The third panel is similar to the second one, but contains the gene sparsity of the spatial data. Spatial data are usually more sparse than single cell data, a discrepancy which is often responsible for low quality mapping.
- In the last panel, we show the training scores as a function of the difference in sparsity between the dataset. For genes with comparable sparsity, the mapped gene expression is very similar to that in the spatial data. However, if a gene is quite sparse in one dataset (typically, the spatial data) but not in other, the mapping score is lower. This occurs as Tangram cannot properly matched the gene pattern because of inconsistent amount of dropouts between the datasets.

Although the above plots give us a summary of scores at single-gene level, we would need to know *which* are the genes are mapped with low scores. These information are stored in the dataframe `.uns['train_genes_df']`; this is the dataframe used to build the four plots above.

```
[12]: ad_map.uns['train_genes_df']
```

```
[12]:           train_score  sparsity_sc  sparsity_sp  sparsity_diff
      ppia         0.998164     0.000092     0.000000      -0.000092
      ubb          0.997351     0.000092     0.000000      -0.000092
      atp1b1       0.997034     0.014334     0.000000      -0.014334
      tmsb4x       0.996996     0.002811     0.000000      -0.002811
      ckb          0.996355     0.002765     0.000000      -0.002765
      ...               ...          ...          ...            ...
      gabrb2       0.197021     0.078951     0.956790       0.877839
      cdyl2        0.186201     0.425911     0.981481       0.555570
      cntnap5c     0.159086     0.608241     0.993827       0.385586
      dlx1as       0.142598     0.587777     0.990741       0.402964
      kcnh6        0.140526     0.379131     0.996914       0.617783

      [1280 rows x 4 columns]
```

New spatial data via aligned single cells

If the mapping mode is `'cells'`, we can now generate the "new spatial data" using the mapped single cell: this is done via `project_genes`. The function accepts as input a mapping (`adata_map`) and corresponding single cell data (`adata_sc`). The result is a voxel-by-gene `AnnData`, formally similar to `adata_st`, but containing gene expression from the mapped single cell data rather than Visium. For downstream analysis, we always replace `adata_st` with the corresponding `ad_ge`.

```
[13]: ad_ge = tg.project_genes(adata_map=ad_map, adata_sc=adata_sc)
      ad_ge
```

```
[13]: AnnData object with n_obs × n_vars = 324 × 36826
          obs: 'in_tissue', 'array_row', 'array_col', 'n_genes_by_counts', 'log1p_n_genes_by_
      ↪counts', 'total_counts', 'log1p_total_counts', 'pct_counts_in_top_50_genes', 'pct_
      ↪counts_in_top_100_genes', 'pct_counts_in_top_200_genes', 'pct_counts_in_top_500_genes',
      ↪ 'total_counts_MT', 'log1p_total_counts_MT', 'pct_counts_MT', 'n_counts', 'leiden',
      ↪'cluster', 'uniform_density', 'rna_count_based_density'
          var: 'mt', 'n_cells_by_counts', 'mean_counts', 'log1p_mean_counts', 'pct_dropout_by_
      ↪counts', 'total_counts', 'log1p_total_counts', 'n_cells', 'highly_variable', 'highly_
      ↪variable_rank', 'means', 'variances', 'variances_norm', 'sparsity', 'is_training'
          uns: 'cell_class_colors', 'cell_subclass_colors', 'hvg', 'neighbors', 'pca', 'umap',
      ↪'rank_genes_groups', 'training_genes', 'overlap_genes'
```

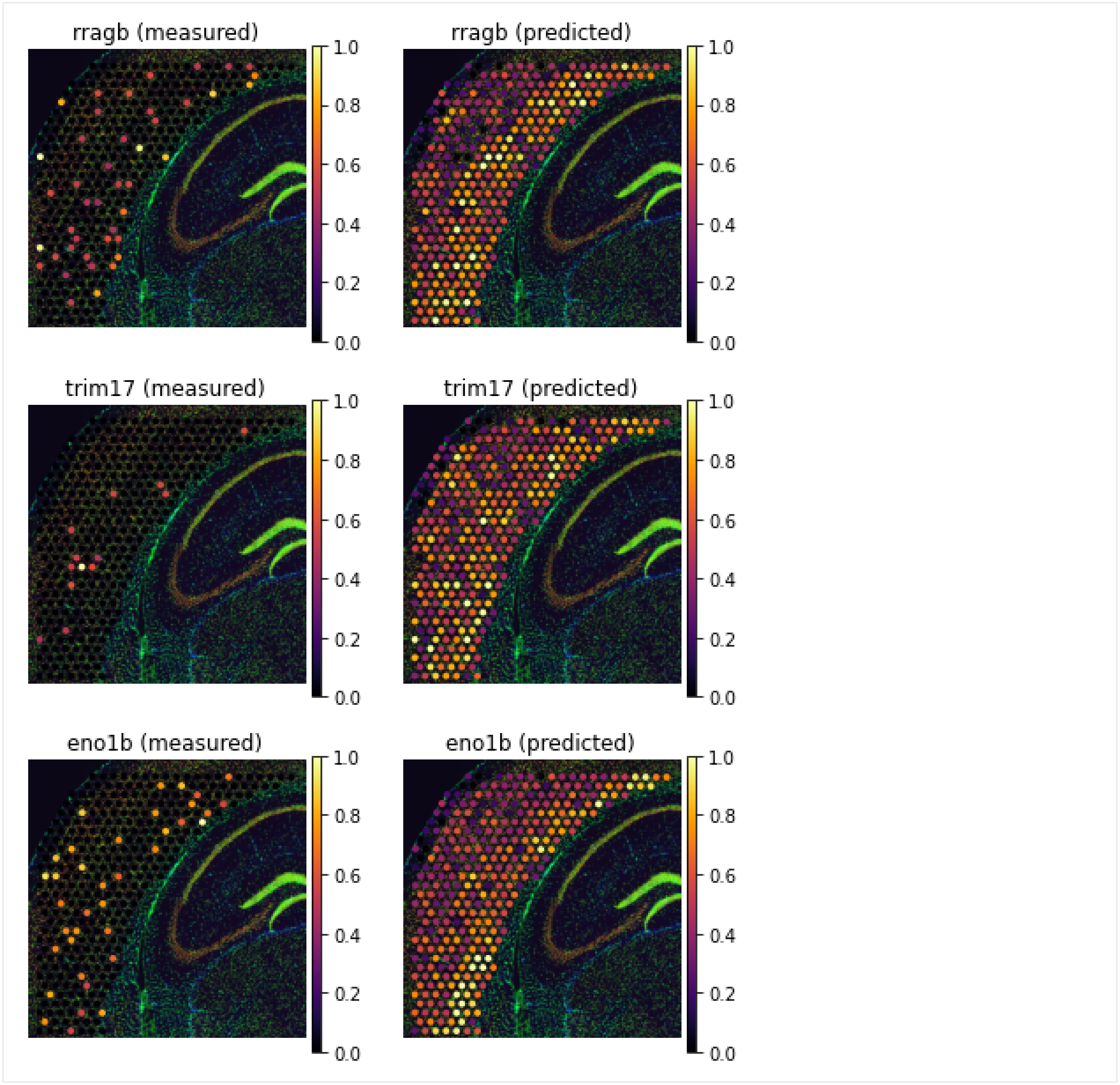
Let's choose a few training genes mapped with low score, to try to understand why.

```
[14]: genes = ['rragb', 'trim17', 'eno1b']
      ad_map.uns['train_genes_df'].loc[genes]
```

```
[14]:         train_score  sparsity_sc  sparsity_sp  sparsity_diff
      rragb      0.357352     0.079919     0.867284       0.787365
      trim17     0.203551     0.069641     0.959877       0.890236
      eno1b      0.341940     0.022492     0.885802       0.863311
```

To visualize gene patterns, we use the helper `plot_genes`. This function accepts two voxel-by-gene `AnnData`: the actual spatial data (`adata_measured`), and a Tangram spatial prediction (`adata_predicted`). The function returns gene expression maps from the two spatial `AnnData` on the genes `genes`.

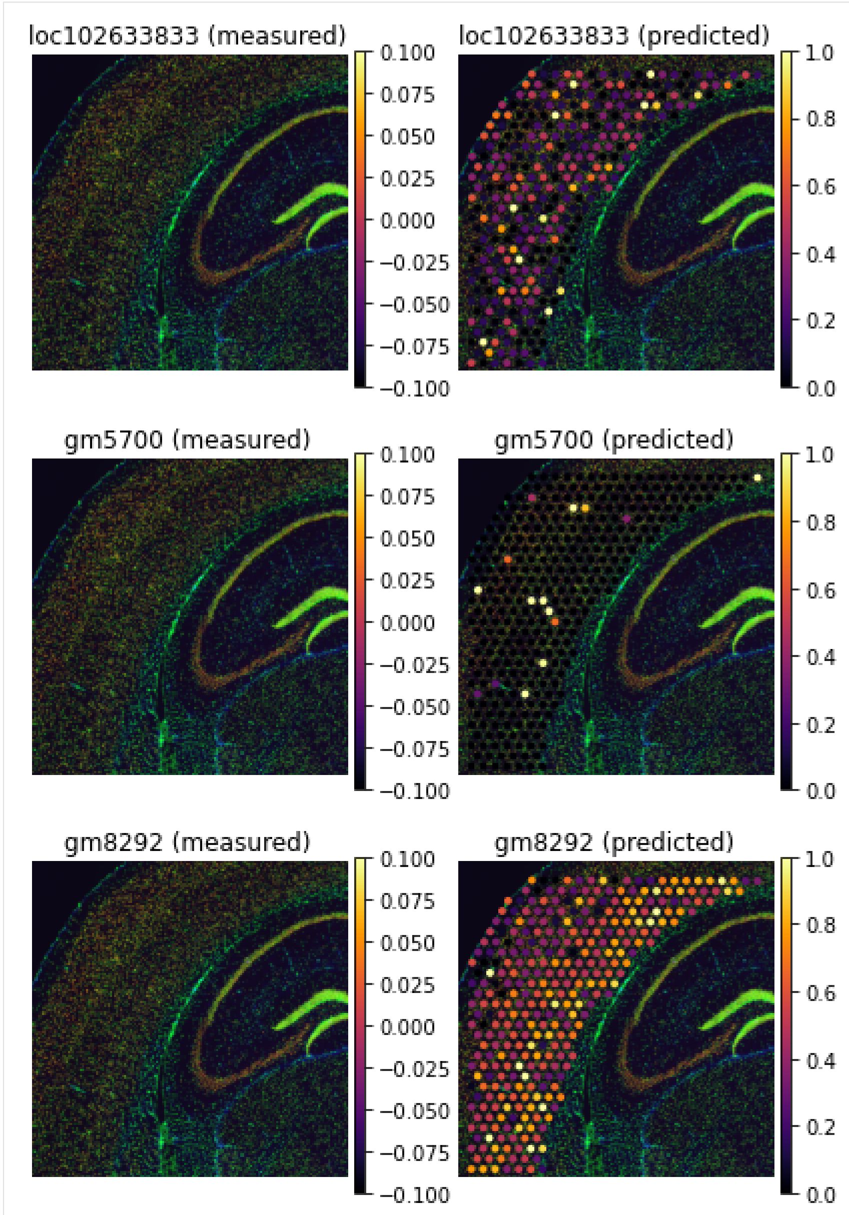
```
[15]: tg.plot_genes_sc(genes, adata_measured=adata_st, adata_predicted=ad_ge, perc=0.02)
```



The above pictures explain the low training scores. Some genes are detected with very different levels of sparsity - typically they are much more sparse in the scRNAseq than in the spatial data. This is due to the fact that technologies like Visium are more prone to technical dropouts. Therefore, *Tangram* cannot find a good spatial alignment for these genes as the baseline signal is missing. However, so long as *most* training genes are measured with high quality, we can trust mapping and use *Tangram* prediction to correct gene expression. This is an imputation method which relies on entirely different premises than those in probabilistic models.

Another application is found by inspecting genes that are not detected in the spatial data, but are detected in the single cell data. They are removed before training with `pp_adatas` function, but *Tangram* can predict their expression.

[16]:
```
genes=['loc102633833', 'gm5700', 'gm8292']
tg.plot_genes_sc(genes, adata_measured=adata_st, adata_predicted=ad_ge, perc=0.02)
```



- So far, we only inspected genes used to align the data (training genes), but the mapped single cell data, `ad_ge` contains the whole transcriptome. That includes more than 35k test genes.

```
[17]: (ad_ge.var.is_training == False).sum()
```

```
[17]: 35546
```

We can use `plot_genes` to inspect gene expression of test genes as well. Inspecting the test transcriptome is an essential to validate mapping. At the same time, we need to be careful that some prediction might disagree with spatial data because of the technical droputs.

It is convenient to compute the similarity scores of all genes, which can be done by `compare_spatial_geneexp`. This function accepts two spatial AnnDatas (ie voxel-by-gene), and returns a dataframe with simlarity scores for all genes. Training genes are flagged by the boolean field `is_training`. If we also pass single cell AnnData to `compare_spatial_geneexp` function like below, a dataframe with additional sparsity columns - sparsity_sc (single cell data sparsity) and sparsity_diff (spatial data sparsity - single cell data sparsity) will return. This is required if we want to call `plot_test_scores` function later with the returned datafrme from `compare_spatial_geneexp` function.

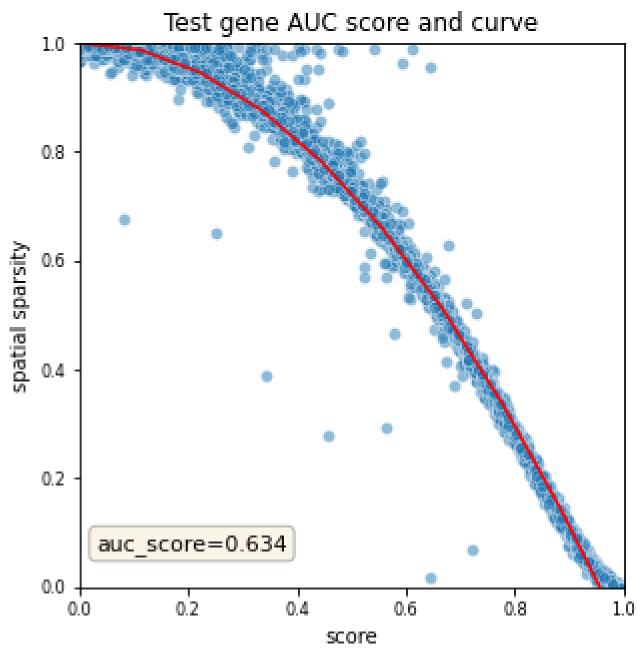
```
[18]: df_all_genes = tg.compare_spatial_geneexp(ad_ge, adata_st, adata_sc)
      df_all_genes
```

```
[18]:              score  is_training  sparsity_sp  sparsity_sc  sparsity_diff
      snap25    0.998254        False     0.000000     0.014610      -0.014610
      gapdh     0.998188        False     0.000000     0.000968      -0.000968
      ppia      0.998164         True     0.000000     0.000092      -0.000092
      calm1     0.997960        False     0.000000     0.000369      -0.000369
      calm2     0.997759        False     0.000000     0.001751      -0.001751
      ...            ...          ...          ...          ...            ...
      otx2      0.000013        False     0.996914     0.998341      -0.001427
      prr32     0.000010        False     0.993827     0.999263      -0.005435
      clec12a   0.000009        False     0.996914     0.998848      -0.001934
      sntn      0.000008        False     0.996914     0.999493      -0.002579
      cckar     0.000004        False     0.996914     0.999309      -0.002395

      [14785 rows x 5 columns]
```

The prediction on test genes can be graphically visualized using `plot_auc`:

```
[19]: # sns.scatterplot(data=df_all_genes, x='score', y='sparsity_sp', hue='is_training',␣
      ↪alpha=.5);  # for legacy
      tg.plot_auc(df_all_genes);
```

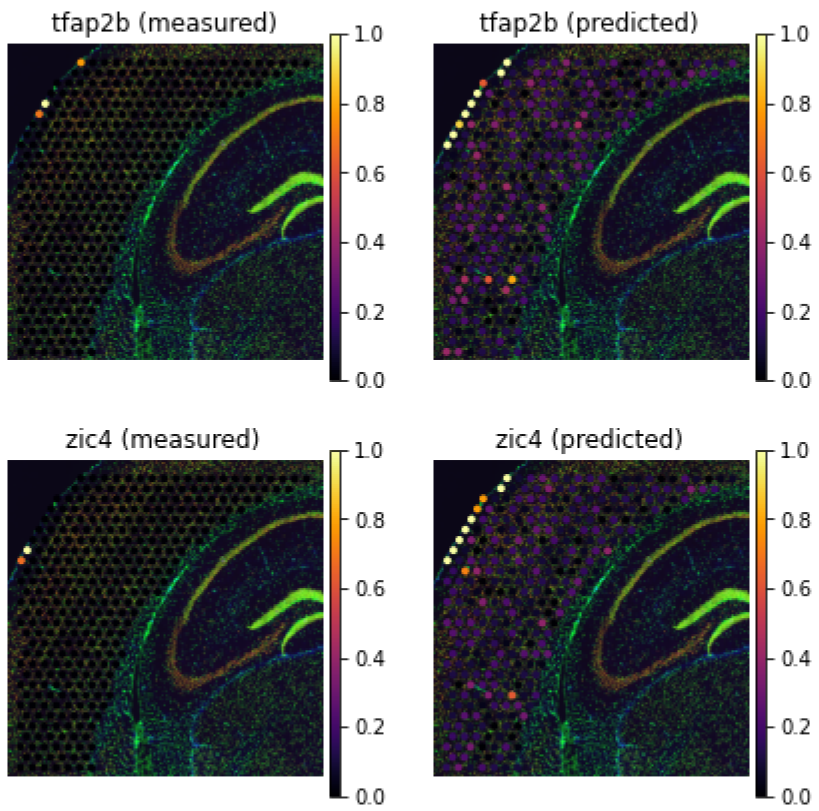
```
<Figure size 432x288 with 0 Axes>
```



This above figure is the most important validation plot in *Tangram*. Each dot represents a gene; the x-axis indicates the score, and the y-axis the sparsity of that gene in the spatial data. Unsurprisingly, the genes predicted with low score represents very sparse genes in the spatial data, suggesting that the *Tangram* predictions correct expression in those genes. Note that curve observed above is typical of *Tangram* mappings: the area under that curve is the most reliable metric we use to evaluate mapping.

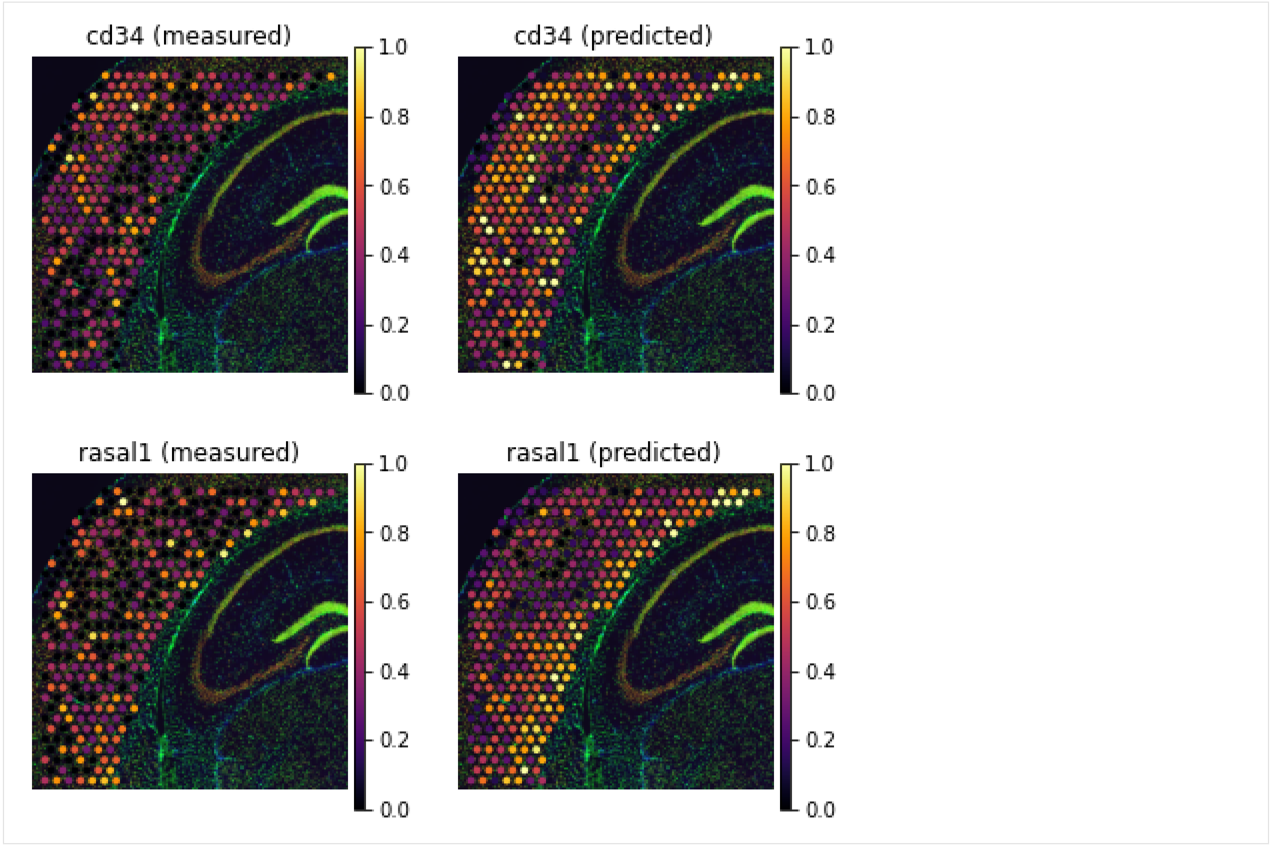
Let's inspect a few predictions. Some of these genes are biologically sparse, but well predicted:

[21]:
```
genes=['tfap2b', 'zic4']
tg.plot_genes_sc(genes, adata_measured=adata_st, adata_predicted=ad_ge, perc=0.02)
```



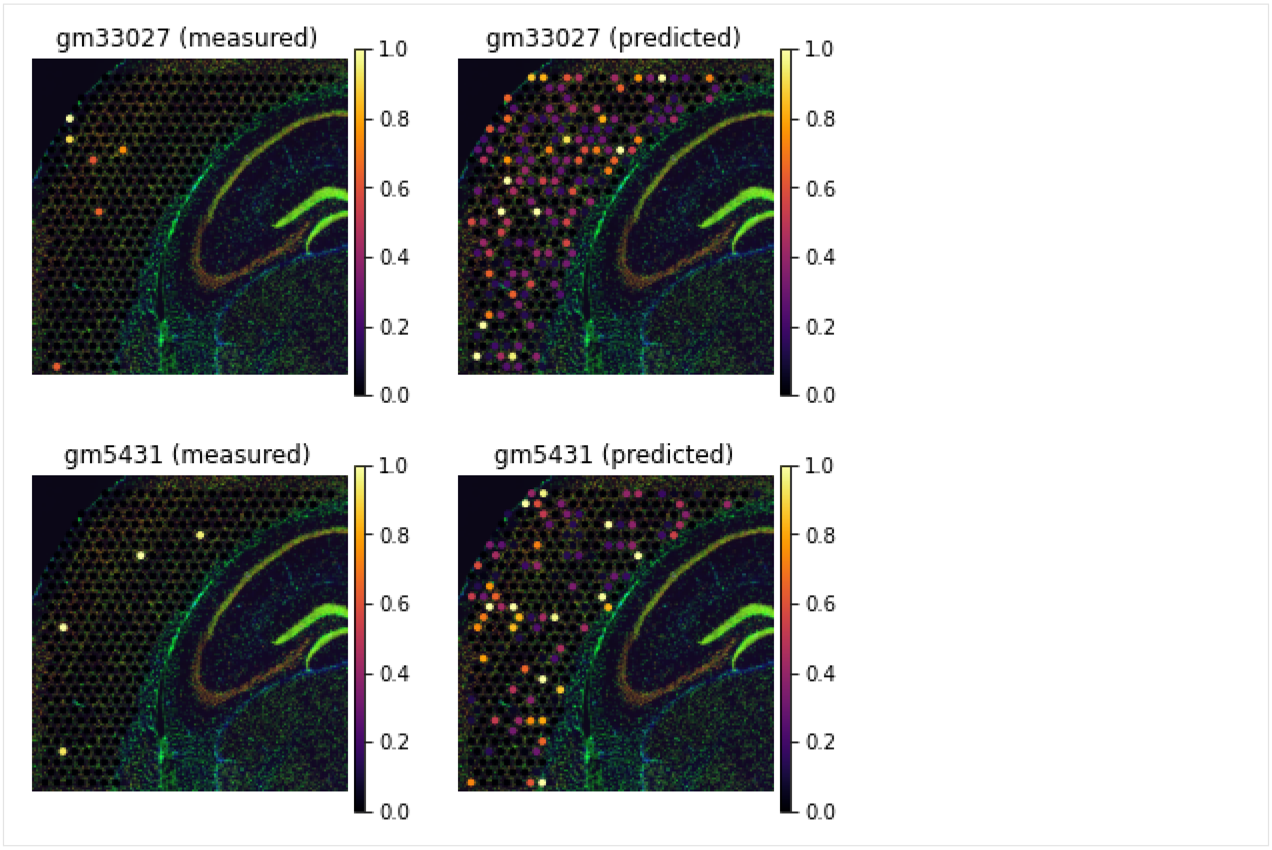
Some non-sparse genes present petterns, that *Tangram* accentuates:

[22]:
```
genes = ['cd34', 'rasal1']
tg.plot_genes_sc(genes, adata_measured=adata_st, adata_predicted=ad_ge, perc=0.02)
```



Finally, some unannotated genes have unknown function. These genes are often hardly detected in spatial data but *Tangram* provides prediction:

```
[23]: genes = ['gm33027', 'gm5431']
tg.plot_genes_sc(genes[:5], adata_measured=adata_st, adata_predicted=ad_ge, perc=0.02)
```



For untargeted spatial technologies, like Visium and Slide-seq, a spatial voxel may contain more than one cells. In these cases, it might be useful to disentangle gene expression into single cells - a process called deconvolution. Deconvolution is a requested feature, and also hard to obtain accurately with computational methods. If your goal is to study co-localization of cell types, we recommend you work with the spatial cell type maps instead. If your aim is discovery of cell-cell communication mechanisms, we suggest you compute gene programs, then use `project_cell_annotations` to spatially visualize program usage. To proceed with deconvolution anyways, see below.

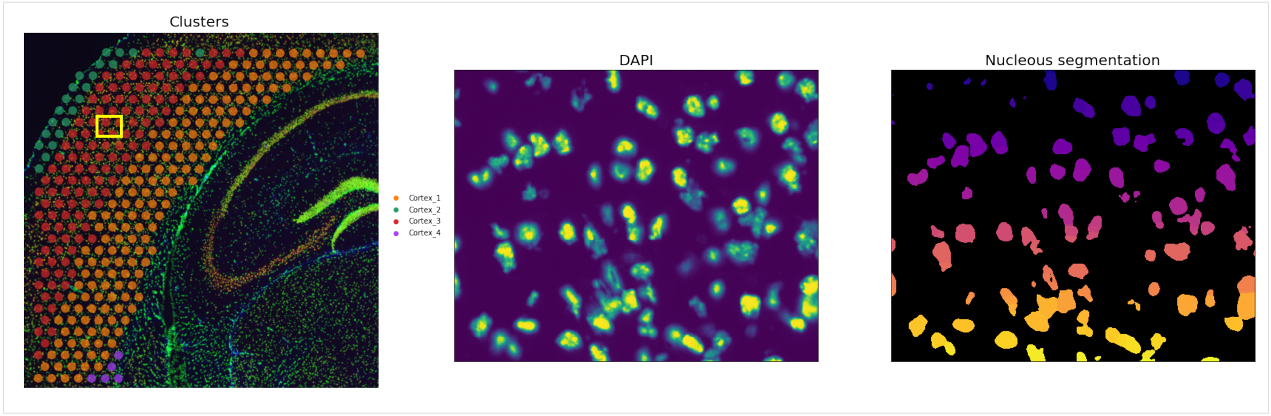
In order to deconvolve cells, *Tangram* needs to know how many cells are present in each voxel. This is achieved by segmenting the cells on the corresponding histology, which squidpy makes possible with two lines of code: - `squidpy.im.process` applies smoothing as a pre-processing step. - `squidpy.im.segment` computes segmentation masks with watershed algorithm.

Note that some technologies, like Slide-seq, currently do not allow staining the same slide of tissue on which genes are profiled. For these data, you can still attempt a deconvolution by estimating cell density in a rough way - often we just pass a uniform prior. Finally, note that deconvolutions are hard to validate, as we do not have ground truth spatially-resolved single cells.

[24]:
```
sq.im.process(img=img, layer="image", method="smooth")
sq.im.segment(
    img=img,
    layer="image_smooth",
    method="watershed",
    channel=0,
)
```

Let's visualize the segmentation results for an inset

[25]:
```
inset_y = 1500
inset_x = 1700
inset_sy = 400
inset_sx = 500

fig, axs = plt.subplots(1, 3, figsize=(30, 10))
sc.pl.spatial(
    adata_st, color="cluster", alpha=0.7, frameon=False, show=False, ax=axs[0], title=""
)
axs[0].set_title("Clusters", fontdict={"fontsize": 20})
sf = adata_st.uns["spatial"]["V1_Adult_Mouse_Brain_Coronal_Section_2"]["scalefactors"][
    "tissue_hires_scalef"
]
rect = mpl.patches.Rectangle(
    (inset_y * sf, inset_x * sf),
    width=inset_sx * sf,
    height=inset_sy * sf,
    ec="yellow",
    lw=4,
    fill=False,
)
axs[0].add_patch(rect)

axs[0].axes.xaxis.label.set_visible(False)
axs[0].axes.yaxis.label.set_visible(False)

axs[1].imshow(
    img["image"][inset_y : inset_y + inset_sy, inset_x : inset_x + inset_sx, 0, 0]
    / 65536,
    interpolation="none",
)
axs[1].grid(False)
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[1].set_title("DAPI", fontdict={"fontsize": 20})

crop = img["segmented_watershed"][
    inset_y : inset_y + inset_sy, inset_x : inset_x + inset_sx
].values.squeeze(-1)
crop = skimage.segmentation.relabel_sequential(crop)[0]
cmap = plt.cm.plasma
cmap.set_under(color="black")
axs[2].imshow(crop, interpolation="none", cmap=cmap, vmin=0.001)
axs[2].grid(False)
axs[2].set_xticks([])
axs[2].set_yticks([])
axs[2].set_title("Nucleous segmentation", fontdict={"fontsize": 20});
```



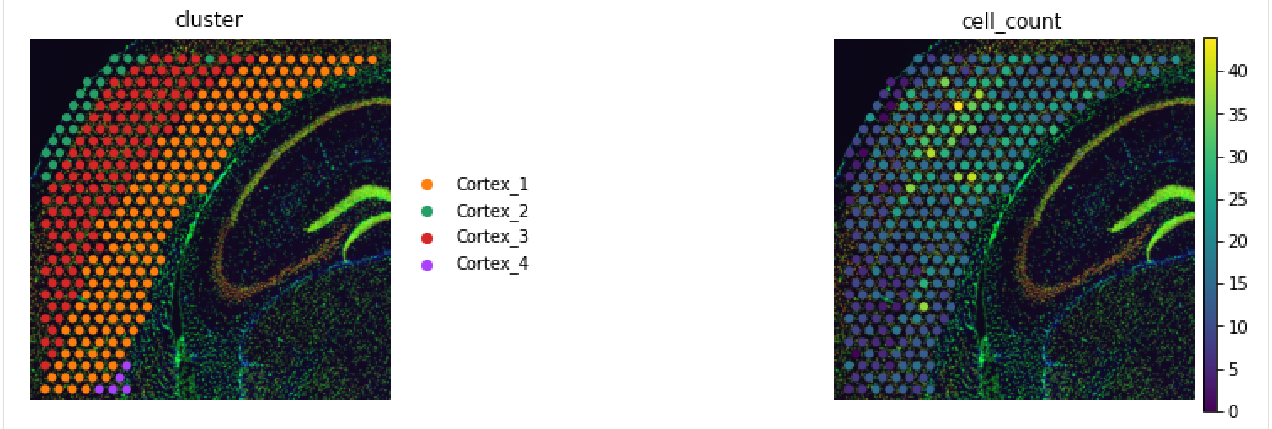
Comparison between DAPI and mask confirms the quality of the segmentation. We then need to extract some image features useful for the deconvolution task downstream. Specifically: - the number of unique segmentation objects (i.e. nuclei) under each spot. - the coordinates of the centroids of the segmentation object.

[26]:
```
# define image layer to use for segmentation
features_kwargs = {
    "segmentation": {
        "label_layer": "segmented_watershed",
        "props": ["label", "centroid"],
        "channels": [1, 2],
    }
}
# calculate segmentation features
sq.im.calculate_image_features(
    adata_st,
    img,
    layer="image",
    key_added="image_features",
    features_kwargs=features_kwargs,
    features="segmentation",
    mask_circle=True,
)
```

```
  0%|          | 0/324 [00:00<?, ?/s]
```

We can visualize the total number of objects under each spot with scanpy.

[27]:
```
adata_st.obs["cell_count"] = adata_st.obsm["image_features"]["segmentation_label"]
sc.pl.spatial(adata_st, color=["cluster", "cell_count"], frameon=False)
```



Deconvolution via alignment

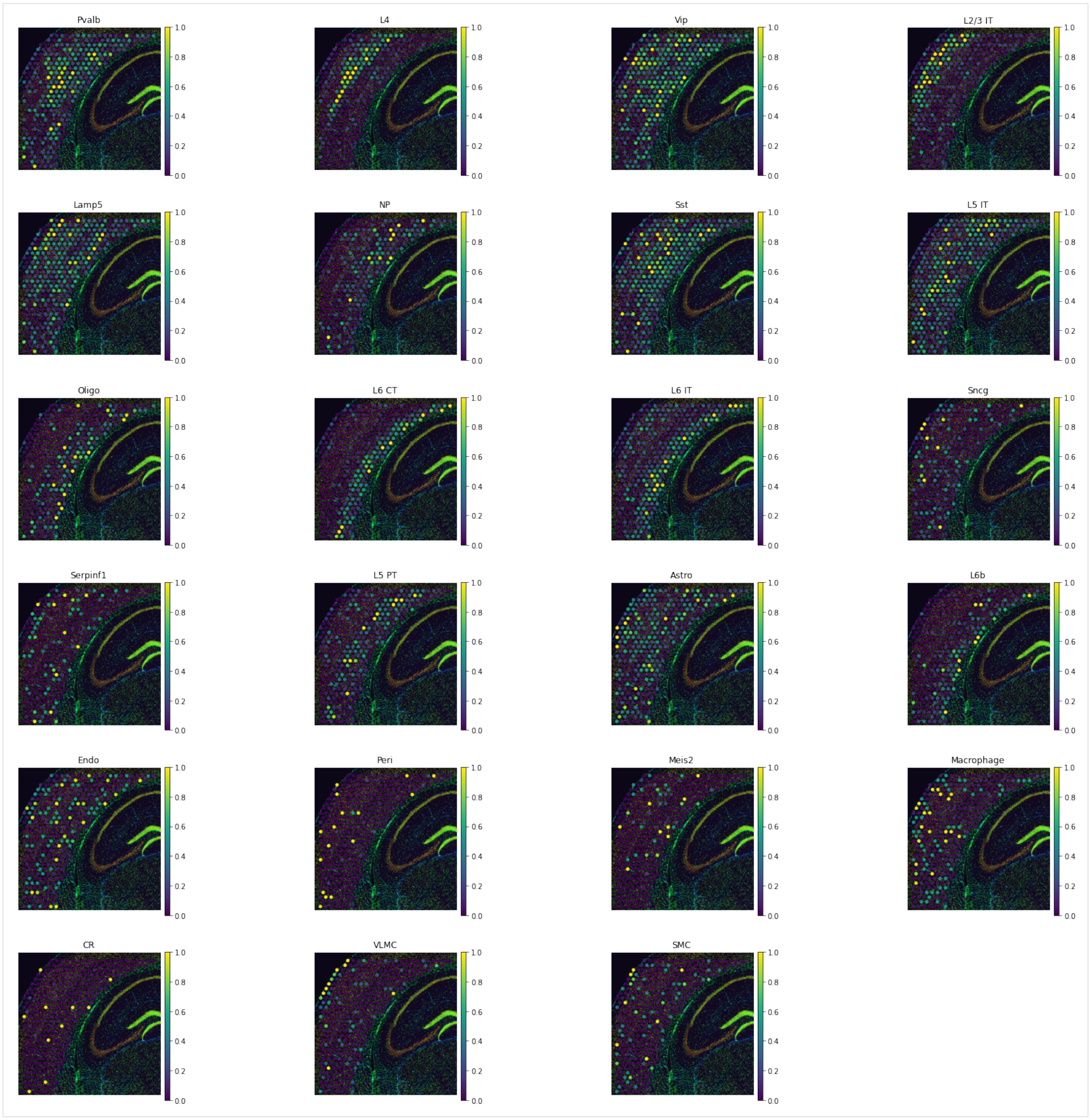
The rationale for deconvolving with Tangram, is to constrain the number of mapped single cell profiles. This is different that most deconvolution method. Specifically, we set them equal to the number of segmented cells in the histology, in the following way: - We pass `mode='constrained'`. This adds a filter term to the loss function, and a boolean regularizer. - We set `target_count` equal to the total number of segmented cells. *Tangram* will look for the best `target_count` cells to align in space. - We pass a `density_prior`, containing the fraction of cells per voxel.

[28]:
```
ad_map = tg.map_cells_to_space(
    adata_sc,
    adata_st,
    mode="constrained",
    target_count=adata_st.obs.cell_count.sum(),
    density_prior=np.array(adata_st.obs.cell_count) / adata_st.obs.cell_count.sum(),
    num_epochs=1000,
#     device="cuda:0",
    device='cpu',
)
```

```
Score: 0.613, KL reg: 0.125, Count reg: 5644.970, Lambda f reg: 4481.579
Score: 0.698, KL reg: 0.012, Count reg: 12.219, Lambda f reg: 725.094
Score: 0.700, KL reg: 0.012, Count reg: 2.325, Lambda f reg: 249.482
Score: 0.700, KL reg: 0.012, Count reg: 0.521, Lambda f reg: 171.574
Score: 0.701, KL reg: 0.012, Count reg: 1.381, Lambda f reg: 140.555
Score: 0.701, KL reg: 0.012, Count reg: 0.524, Lambda f reg: 125.590
Score: 0.701, KL reg: 0.012, Count reg: 0.990, Lambda f reg: 111.074
Score: 0.701, KL reg: 0.012, Count reg: 0.573, Lambda f reg: 99.697
Score: 0.701, KL reg: 0.012, Count reg: 0.889, Lambda f reg: 90.959
Score: 0.701, KL reg: 0.012, Count reg: 0.854, Lambda f reg: 80.407
```

In the same way as before, we can plot cell type maps:

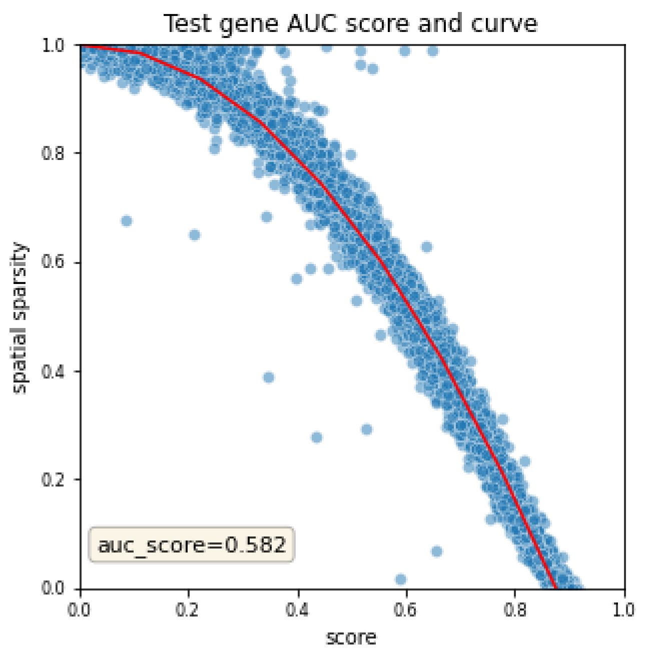
[29]:
```
tg.project_cell_annotations(ad_map, adata_st, annotation="cell_subclass")
annotation_list = list(pd.unique(adata_sc.obs['cell_subclass']))
tg.plot_cell_annotation_sc(adata_st, annotation_list, perc=0.02)
```



We validate mapping by inspecting the test transcriptome:

```
[30]: ad_ge = tg.project_genes(adata_map=ad_map, adata_sc=adata_sc)
df_all_genes = tg.compare_spatial_geneexp(ad_ge, adata_st, adata_sc)
tg.plot_auc(df_all_genes);
```

```
<Figure size 432x288 with 0 Axes>
```



And here comes the key part, where we will use the results of the previous deconvolution steps. Previously, we computed the absolute numbers of unique segmentation objects under each spot, together with their centroids. Let's extract them in the right format useful for *Tangram*. In the resulting dataframe, each row represents a single segmentation object (a cell). We also have the image coordinates as well as the unique centroid ID, which is a string that contains both the spot ID and a numerical index. *Tangram* provides a convenient function to export the mapping between spot ID and segmentation ID to `adata.uns`.

[31]:
```
tg.create_segment_cell_df(adata_st)
adata_st.uns["tangram_cell_segmentation"].head()
```

[31]:
```
             spot_idx            y           x             centroids
0  AAATGGCATGTCTTGT-1  5304.000000  731.000000  AAATGGCATGTCTTGT-1_0
1  AAATGGCATGTCTTGT-1  5320.947519  721.331554  AAATGGCATGTCTTGT-1_1
2  AAATGGCATGTCTTGT-1  5332.942342  717.447904  AAATGGCATGTCTTGT-1_2
3  AAATGGCATGTCTTGT-1  5348.865384  558.924248  AAATGGCATGTCTTGT-1_3
4  AAATGGCATGTCTTGT-1  5342.124989  567.208502  AAATGGCATGTCTTGT-1_4
```

We can use `tangram.count_cell_annotation()` to map cell types as result of the deconvolution step to putative segmentation ID.

[32]:
```
tg.count_cell_annotations(
    ad_map,
    adata_sc,
    adata_st,
    annotation="cell_subclass",
)
adata_st.obsm["tangram_ct_count"].head()
```

[32]:
```
                       x     y  cell_n  \
AAATGGCATGTCTTGT-1   641  5393      13
AACAACTGGTAGTTGC-1  4208  1672      16
AACAGGAAATCGAATA-1  1117  5117      28
```

(continues on next page)

(continued from previous page)

```
AACCCAGAGACGGAGA-1  1101  1274       5
AACCGTTGTGTTTGCT-1   399  4708       7

                                                     centroids  Pvalb  \
AAATGGCATGTCTTGT-1  [AAATGGCATGTCTTGT-1_0, AAATGGCATGTCTTGT-1_1, A...      1
AACAACTGGTAGTTGC-1  [AACAACTGGTAGTTGC-1_0, AACAACTGGTAGTTGC-1_1, A...      1
AACAGGAAATCGAATA-1  [AACAGGAAATCGAATA-1_0, AACAGGAAATCGAATA-1_1, A...      2
AACCCAGAGACGGAGA-1  [AACCCAGAGACGGAGA-1_0, AACCCAGAGACGGAGA-1_1, A...      3
AACCGTTGTGTTTGCT-1  [AACCGTTGTGTTTGCT-1_0, AACCGTTGTGTTTGCT-1_1, A...      2

                    L4  Vip  L2/3 IT  Lamp5  NP  ...  L5 PT  Astro  L6b  Endo  \
AAATGGCATGTCTTGT-1   0    2        0      2   0  ...      1      2    0     0
AACAACTGGTAGTTGC-1   0    4        0      2   2  ...      2      0    0     0
AACAGGAAATCGAATA-1   0    2        1      1   1  ...      0      0    0     0
AACCCAGAGACGGAGA-1   0    0        1      0   0  ...      0      1    0     0
AACCGTTGTGTTTGCT-1   0    0        0      0   0  ...      0      0    0     1

                    Peri  Meis2  Macrophage  CR  VLMC  SMC
AAATGGCATGTCTTGT-1     0      0           0   0     0    0
AACAACTGGTAGTTGC-1     0      0           1   0     0    0
AACAGGAAATCGAATA-1     0      0           0   0     0    0
AACCCAGAGACGGAGA-1     0      0           1   0     0    0
AACCGTTGTGTTTGCT-1     0      0           0   0     0    0

[5 rows x 27 columns]
```

And finally export the results in a new AnnData object.

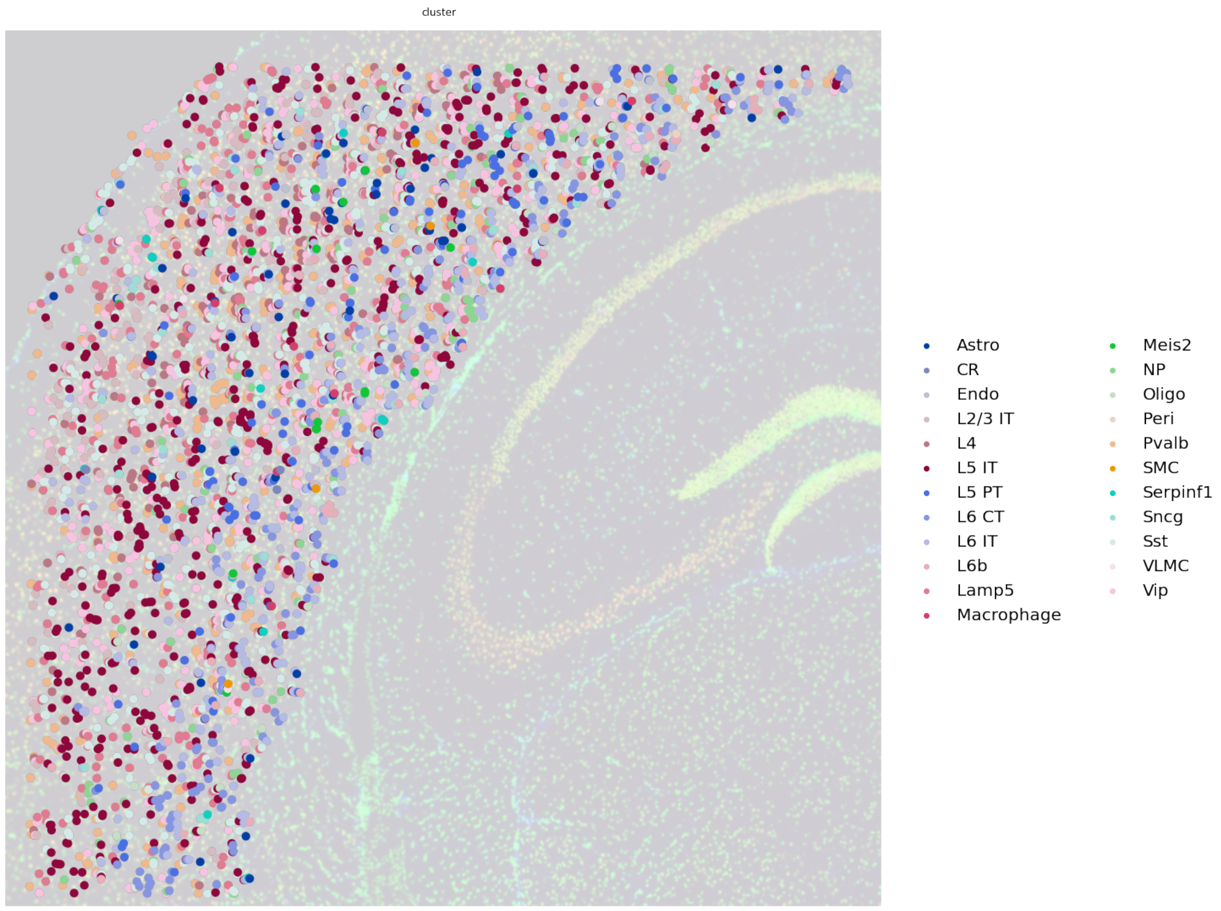
[33]:
```
adata_segment = tg.deconvolve_cell_annotations(adata_st)
adata_segment.obs.head()
```

[33]:
```
             y           x             centroids cluster
0  5304.000000  731.000000  AAATGGCATGTCTTGT-1_0   Pvalb
1  5320.947519  721.331554  AAATGGCATGTCTTGT-1_1     Vip
2  5332.942342  717.447904  AAATGGCATGTCTTGT-1_2     Vip
3  5348.865384  558.924248  AAATGGCATGTCTTGT-1_3   Lamp5
4  5342.124989  567.208502  AAATGGCATGTCTTGT-1_4   Lamp5
```

Note that the AnnData object does not contain counts, but only cell type annotations, as results of the Tangram mapping. Nevertheless, it's convenient to create such AnnData object for visualization purposes. Below you can appreciate how each dot is now not a Visium spot anymore, but a single unique segmentation object, with the mapped cell type.

[34]:
```
fig, ax = plt.subplots(1, 1, figsize=(20, 20))
sc.pl.spatial(
    adata_segment,
    color="cluster",
    size=0.4,
    show=False,
    frameon=False,
    alpha_img=0.2,
    legend_fontsize=20,
    ax=ax,
)
```

```
[34]: [<AxesSubplot:title={'center':'cluster'}, xlabel='spatial1', ylabel='spatial2'>]
```


cluster
Astro
CR
Endo
L2/3 IT
L4
L5 IT
L5 PT
L6 CT
L6 IT
L6b
Lamp5
Macrophage
Meis2
NP
Oligo
Peri
Pvalb
SMC
Serpinf1
Sncg
Sst
VLMC
Vip

```
[ ]:
```

PYTHON MODULE INDEX

INDEX