A guide to the single-cell epigenomics analysis

Kai Zhang

Table of contents

Preface

This book is used to complement the documentation of the SnapATAC2 Python/Rust package.

1 Introduction

This is a book created from markdown and executable code.

2 AnnData – Annotated Data

2.1 Introduction

AnnData is both a data structure and an on-disk file specification that facilitates the sharing of labeled data matrices.

The Python anndata package supports both in-memory and on-disk representation of AnnData object. For detailed descriptions about the AnnData format, please read `anndata`'s documentation.

Despite being an excellent package, the `anndata` package falls shall of its support for the on-disk representation or backed mode of AnnData object. When opened in the backed mode, the in-memory snapshot and on-disk data of AnnData are not in sync with each other, causing inconsistent and unexpected behaviors. For example in the backed mode, `anndata` only supports updates to the `X` slot in the AnnData object, which means any changes to other slots like `obs` will not be written to disk. This make the backed mode very cumbersome to use and often lead to unexpected outcomes. Also, as it still reads all other componenets except `X` into memory, it uses a lot of memory for large datasets.

To address these limitations, SnapATAC2 implements its own **out-of-core** AnnData object with the following key features:

- AnnData is fully backed by the underlying hdf5 file. Any operations on the AnnData object will be reflected on the hdf5 file.

- All elements are lazily loaded. No matter how large is the file, opening it consume almost zero memory. Matrix data can be accessed and processed by chunks, which keeps the memory usage to the minimum.
- In-memory cache can be turned on to speed up the repetitive access of elements.
- Featuring an AnnDataSet object to lazily concatenate multiple AnnData objects.

2.2 A tutorial on using backed AnnData objects

In this section, we will learn the basics about SnapATAC2's AnnData implementation.

2.2.1 Reading/opening a h5ad file.

SnapATAC2 can open `h5ad` files in either in-memory mode or backed mode. By default, `snapatac2.read` open a `h5ad` file in backed mode.

```
import snapatac2 as snap
adata = snap.read(snap.datasets.pbmc5k(type='h5ad'))
adata
```

```
AnnData object with n_obs x n_vars = 4363 x 6176550 backed at '/home/kaizhang
    obs: 'tsse', 'n_fragment', 'frac_dup', 'frac_mito', 'doublet_score', 'is_
    var: 'selected'
    uns: 'reference_sequences', 'scrublet_sim_doublet_score', 'peaks', 'scrul
    obsm: 'X_umap', 'X_spectral', 'insertion'
    obsp: 'distances'
```

You can turn the backed mode off using `backed=False`, which will use the Python `anndata` package to read the file and create an in-memory AnnData object.

```
import snapatac2 as snap
adata = snap.read(snap.datasets.pbmc5k(type='h5ad'), backed=False)
adata
```

```
Updating file 'atac_pbmc_5k.h5ad' from 'http://renlab.sdsc.edu/kai/public_datasets/single_ce
```

```
AnnData object with n_obs × n_vars = 4363 × 6176550
    obs: 'tsse', 'n_fragment', 'frac_dup', 'frac_mito', 'doublet_score', 'is_doublet', 'leid
    var: 'selected'
    uns: 'peaks', 'reference_sequences', 'scrublet_sim_doublet_score', 'scrublet_threshold',
    obsm: 'X_spectral', 'X_umap', 'insertion'
    obsp: 'distances'
```

2.2.2 Closing a backed AnnData object

The backed AnnData object in SnapATAC2 does not need to be saved as it is always in sync with the data on disk. However, if you have opened the `h5ad` file in write mode, it is important to remember to close the file using the `AnnData.close` method. Otherwise, the underlying hdf5 file might be corrupted.

```
adata = snap.read(snap.datasets.pbmc5k(type='h5ad'))
adata.close()
adata
```

```
Closed AnnData object
```

2.2.3 Creating a backed AnnData object

You can use the `AnnData` constructor to create a new AnnData object.

```
adata = snap.AnnData(filename='adata.h5ad')
adata
```

```
AnnData object with n_obs x n_vars = 0 x 0 backed at 'adata.h5ad'
```

You can then modify slots in the AnnData object.

```
import numpy as np
adata.X = np.ones((3, 4))
adata.obs_names = ["1", "2", "3"]
adata.var_names = ["a", "b", "c", "d"]
adata.obsm['matrix'] = np.ones((3, 10))
adata.varm['another_matrix'] = np.ones((4, 10))
adata
```

```
AnnData object with n_obs x n_vars = 3 x 4 backed at 'adata.h5ad'
    obsm: 'matrix'
    varm: 'another_matrix'
```

The matrices are now saved on the backing hdf5 file and will be cleared from the memory.

2.2.4 Accessing elements in a backed AnnData object

Slots in backed AnnData object, *e.g.,* `AnnData.X`, `AnnData.obs`, store references to the actual data. Accessing those slots does not automatically perform dereferencing or load the data into memory. Instead, a lazy element will be returned, as demonstrated in the example below:

```
adata.X
```

```
3 x 4 Array(Float(U8)) element, cache_enabled: no, cached: no
```

However, asscessing the slots by keys will automatically read the data:

```
adata.obsm['matrix']
```

```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

To retreive the lazy element from `obsm`, you can use:

```
adata.obsm.el('matrix')
```

```
3 x 10 Array(Float(U8)) element, cache_enabled: no, cached: no
```

Several useful methods haven been implemented for lazy elements. For example, you can use the slicing operator to read the full data or a part of the data:

```
adata.X[:]
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

```
adata.X[:2, :2]
```

```
array([[1., 1.],
       [1., 1.]])
```

You can also iterate over the chunks of the matrix using the `chunked` method:

```
for chunk, fr, to in adata.obsm.el('matrix').chunked(chunk_size=2):
    print("from row {} to {}: {}".format(fr, to - 1, chunk))
```

```
from row 0 to 1: [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
from row 2 to 2: [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

By default AnnData will read from the disk each time you request the data. This will incur a lot of IO overheads if you do this repetitively.

```
%%time
for _ in range(1000):
    adata.obsm['matrix']
```

```
CPU times: user 14.2 ms, sys: 153 µs, total: 14.4 ms
Wall time: 14.3 ms
```

One solution to this is to turn on the cache for the element you want to repetitively read from.

```
%%time
adata.obsm.el('matrix').enable_cache()
for _ in range(1000):
    adata.obsm['matrix']
```

```
CPU times: user 644 µs, sys: 0 ns, total: 644 µs
Wall time: 650 µs
```

The data will be cached the first time you request it and the subsequent calls will make use of the cached data.

2.2.5 Subsetting the AnnData

The backed AnnData object does not have "views". Instead, you need to use the `AnnData.subset` method to create a new AnnData object.

```
adata_subset = adata.subset([0, 1], [0, 1], out="subset.h5ad")
adata_subset
```

```
AnnData object with n_obs x n_vars = 2 x 2 backed at 'subset.h5ad'
    obsm: 'matrix'
    varm: 'another_matrix'
```

You could also do this inplace without the `out` parameter:

```
adata_subset.subset([0])
adata_subset
```

```
AnnData object with n_obs x n_vars = 1 x 2 backed at 'subset.h5ad'
    obsm: 'matrix'
    varm: 'another_matrix'
```

2.2.6 Convert to in-memory representation

Finally, you can convert a backed AnnData to `anndata`'s in-memory AnnData object using:

```
adata.to_memory()
```

```
AnnData object with n_obs × n_vars = 3 × 4
    obsm: 'matrix'
    varm: 'another_matrix'
```

2.3 Combining multiple AnnData objects into a AnnDataSet object

Oftentimes you want to combine and deal with multiple h5ad files simultaniously. In this section you will learn how to do this efficiently.

First, let us create a bunch of AnnData objects.

```
def create_anndata(index: int):
    adata = snap.AnnData(
        X=np.ones((4, 7))*index,
        filename=str(index) + ".h5ad",
    )
    adata.var_names = [str(i) for i in range(7)]
    adata.obs_names = [str(i) for i in range(4)]
    adata.obsm['matrix'] = np.random.rand(4,50)
    return adata
list_of_anndata = [(str(i), create_anndata(i)) for i in range(10)]
```

We can then use the `AnnDataSet` constructor to horizontally concatenate all AnnData objects.

```
dataset = snap.AnnDataSet(
    adatas=list_of_anndata,
    filename="dataset.h5ads",
    add_key="id",
)
dataset
```

```
AnnDataSet object with n_obs x n_vars = 40 x 7 backed at 'dataset.h5ads'
contains 10 AnnData objects with keys: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
    obs: 'id'
    uns: 'AnnDataSet'
```

AnnDataSet is just a special form of AnnData objects. It inherits most of the methods from AnnData. It carries its own annotations, such as `obs`, `var`, `obsm`, *etc.* Besides, it grants you the access to component AnnData objects as well, as shown in the example below:

```
dataset.adatas.obsm['matrix']
```

```
array([[0.50191366, 0.16120245, 0.02598691, ..., 0.14598419, 0.60951903,
        0.88647748],
       [0.85330824, 0.11952603, 0.27757905, ..., 0.27949759, 0.82678061,
        0.77015056],
       [0.75666633, 0.10675303, 0.32092797, ..., 0.0948781 , 0.659396  ,
        0.5274284 ],
       ...,
       [0.49019988, 0.17998617, 0.8488963 , ..., 0.23487292, 0.0684726 ,
        0.7806173 ],
       [0.44687947, 0.90347629, 0.62437934, ..., 0.02896885, 0.73274449,
        0.57782651],
       [0.94853876, 0.8984447 , 0.92410241, ..., 0.9648648 , 0.16504023,
        0.51376406]])
```

2.3.1 Subsetting an AnnDataSet object

AnnDataSet can be subsetted in a way similar to AnnData objects. But there is one caveat: subsetting an AnnDataSet will not rearrange the rows across component AnnData objects.

2.3.2 Converting AnnDataSet to AnnData

An in-memory AnnData can be made from AnnDataSet using:

```
dataset.to_adata()
```

```
AnnData object with n_obs × n_vars = 40 × 7
    obs: 'id'
    uns: 'AnnDataSet'
```

3 Input data format

SnapATAC2 accepts BAM or BED-like tabular file as input. The BED-like tabular file can be used to represent fragments (paired-end sequencing) or insertions (single-end sequencing). BAM files can be converted to BED-like files using `snapatac2.pp.make_fragment_file`.

3.1 Fragment interval format

Fragments are created by two separate transposition events, which create the two ends of the observed fragment. Each unique fragment may generate multiple duplicate reads. These duplicate reads are collapsed into a single fragment record. **A fragment record must contain exactly five fields**:

1. Reference genome chromosome of fragment.
2. Adjusted start position of fragment on chromosome.
3. Adjusted end position of fragment on chromosome. The end position is exclusive, so represents the position immediately following the fragment interval.
4. The cell barcode of this fragment.
5. The total number of read pairs associated with this fragment. This includes the read pair marked unique and all duplicate read pairs.

During data import, a fragment record is converted to two insertions corresponding to the start and end position of the fragment interval.

3.2 Insertion format

Insertion records are used to represent single-end reads in experiments that sequence only one end of the fragments, e.g., Paired-Tag experiments. While fragment records are created by two transposition events, insertion records correspond to a single transposition event.

Each insertion record must contain six fields:

1. Reference genome chromosome.
2. Adjusted start position on chromosome.
3. Adjusted end position on chromosome. The end position is exclusive.
4. The cell barcode of this fragment.
5. The total number of reads associated with this insertion.
6. The strandness of the read.

During data import, the 5' end of an insertion record is converted to one insertion count.

Note: in both cases, the fifth column (duplication count) is not used during reads counting. In other words, we count duplicated reads only once. If you want to count the same record multiple times, you need to duplicate them in the input file.

4 Dimension reduction

Single-cell ATAC-seq (scATAC-seq) produces large and highly sparse cell by feature count matrix. Working directly with such a large matrix is very inconvinent and computational intensive. Therefore typically, we need to reduce the dimensionality of the count matrix before any downstream analysis. Most of the counts in this matrix are very small. For example, ~50% of the counts are 1 in deeply sequenced scATAC-seq data. As a result, many methods treat the count matrix as a binary matrix.

Different from most existing approaches, the dimension reduction method used in SnapATAC2 is a pairwise-similarity based method, which requires defining and computing similarity between each pair of cells in the data. This method was first proposed in (Fang et al. 2021), the version 1 of SnapATAC, and was called "diffusion map". In SnapATAC2, we reformulate this approach as spectral embedding, *a.k.a.*, Laplacian eigenmaps.

4.1 Spectral embedding

Start with $n \times p$ cell by feature count matrix M, we first compute the $n \times n$ pairwise similarity matrix S such that $S_{ij} = \delta(M_{i*}, M_{j*})$, where $\delta : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$ is the function defines the similarity between any two cells. Typical choices of δ include the jaccard index and the cosine similarity.

We then compute the normalized graph Laplacian $L = I - D^{-1/2} S D^{-1/2}$, where I is the identity matrix and D is a diagonal matrix such that $D_{ii} = \sum_k S_{ik}$.

The eigenvectors correspond to the k+1-smallest eigenvalues of L are selected as the lower dimensional embedding.

4.2 Nyström method

For samples with large numbers of cells, computing the full similarity matrix is slow and requires a large amount of memory. To address this limitation and increase the scalability of spectral embedding, we used the Nystrom method to perform a low-rank approximation of the full similarity matrix.

We will be focusing on generating an approximation $\tilde{S}$ of S based on a sample of $l \ll n$ of its columns.

Suppose $S = \begin{bmatrix} A & B \\ B^T & C \end{bmatrix}$ and columns $\begin{bmatrix} A \\ B^T \end{bmatrix}$ are our samples. We first perform eigendecomposition on $A = U\Lambda U^T$. The nystrom method approximates the eigenvectors of matrix S by $\tilde{U} = \begin{bmatrix} U \\ B^T U \Lambda^{-1} \end{bmatrix}$.

We can then compute $\tilde{S}$:

$$\begin{aligned}
\tilde{S} &= \tilde{U}\Lambda\tilde{U}^T \\
&= \begin{bmatrix} U \\ B^T U \Lambda^{-1} \end{bmatrix} \Lambda \begin{bmatrix} U^T & \Lambda^{-1} U^T B \end{bmatrix} \\
&= \begin{bmatrix} U\Lambda U^T & U\Lambda\Lambda^{-1}U^T B \\ B^T U \Lambda^{-1} \Lambda U^T & B^T U \Lambda^{-1} \Lambda \Lambda^{-1} U^T B \end{bmatrix} \\
&= \begin{bmatrix} A & B \\ B^T & B^T U \Lambda^{-1} U^T B \end{bmatrix}
\end{aligned}$$

In practice, $\tilde{S}$ does not need to be computed. Instead, it is used implicitly to estimate the degree normalization vector:

$$\tilde{d} = \tilde{S}\mathbf{1} = \begin{bmatrix} A\mathbf{1} + B\mathbf{1} \\ B^T\mathbf{1} + B^T A^{-1} B\mathbf{1} \end{bmatrix}$$

References

Fang, Rongxin, Sebastian Preissl, Yang Li, Xiaomeng Hou, Jacinta Lucero, Xinxin Wang, Amir Motamedi, et al. 2021. “Comprehensive analysis of single cell ATAC-seq data with SnapATAC.” *Nature Communications* 12 (1): 1337. https://doi.org/10.1038/s41467-021-21583-9.