IGVF CRISPR Jamboree 2024: Perturb-seq Inference (Python)

Gene Katsevich

February 9, 2024

1 Overview

1.1 Perturb-seq inference

The goal of perturb-seq inference is to quantify the extent to which the perturbation of a given genomic element impacts the expression of a given gene. We allow a range of statistical interpretations of this task. In a frequentist framework, this task can be viewed as testing the null hypothesis that the perturbation of the genomic element has no effect on the gene's expression, or as estimating the effect size of the perturbation on the gene's expression. In a Bayesian framework, this task can be viewed as estimating the posterior probability of the presence of a non-zero effect, or as a posterior mean of the effect size.

1.2 Jamboree goals

The goal of the perturb-seq inference portion of the Jamboree is to implement a number of perturb-seq inference methodologies using common input and output formats. Following the Jamboree, these implementations will be added as modules to a Nextflow pipeline, which will then be used to benchmark their statistical and computational performance. This benchmarking effort will suggest best practices for perturb-seq inference, and will be used to inform the development of the IGVF perturb-seq analysis pipeline.

1.3 Data format overview

The primary input to a perturb-seq inference module is a `MuData` object, which contains both the perturb-seq data and a set of element-gene pairs for which the inference is to be performed. The output of each method should be the same `MuData` object, except with an additional table containing one or more measures of association for each element-gene pair. The `MuData` format is an HDF5-based language-agnostic format compatible with import into both R and Python. Each `MuData` object will contain a minimal set of fields required for inference, and potentially one or more optional fields that provide additional information. For the purposes of this Jamboree, we have provided `MuData` objects for subsets of the Gasperini et al (2019) and Papalexi et al (2021) datasets. For each dataset, we have provided a minimal `MuData` object that contains just the required fields, as well as a more fleshed out object that contains additional optional fields.

1.4 Requested function API, documentation, and demonstration

Please write a function in your language of choice with the following arguments:

- The first argument should be `mudata_input_fp`, a filepath to a `MuData` object.
- The second argument should be `mudata_output_fp`, a filepath to an output `MuData` object.
- There may be one or more additional arguments specific to your method.

The function should read the `MuData` object from `mudata_input_fp`, perform the inference, and write the resulting `MuData` object to `mudata_output_fp` (in Python, via `mudata.read_h5mu()`). The function should include documentation of any additional arguments used. Furthermore, please include a demonstration of the use of your function on at least one of the sample datasets provided, and a brief discussion of the results.

2 MuData format

Let us walk through the input and output format specifications, from the perspective of Python, using a subset of the Gasperini et al (2019) dataset as an example.

```
import mudata as md
import pandas as pd
pd.set_option('display.max_columns', None)
```

2.1 Required input fields

We start with an example of the minimal `MuData` object required for perturb-seq inference.

```
mudata_input_fp = "data/gasperini_inference_input_minimal.h5mu"
input_minimal = md.read_h5mu(mudata_input_fp)
input_minimal
```

```
## MuData object with n_obs × n_vars = 9704 × 167
##   uns:   'pairs_to_test'
##   2 modalities
##     gene:    9704 x 112
##     guide:   9704 x 55
##       var:   'targeting', 'intended_target_name'
##       uns:   'capture_method', 'moi'
##       layers:    'guide_assignment'
```

The minimal `MuData` object for perturb-seq inference contains two modalities: `gene` and `guide`.

2.1.1 `gene` modality

The `gene` modality just needs to have a `.X` matrix containing the RNA UMI counts.

```
input_minimal['gene'].X
```

```
## <9704x112 sparse matrix of type '<class 'numpy.float64'>'
##  with 355109 stored elements in Compressed Sparse Row format>
```

```
input_minimal['gene'][:2,:2].to_df()
```

```
##                                  ENSG00000008853  ENSG00000104679
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2              0.0              2.0
## AGCTTGATCGAGAGCA-1_1A_2_SI-GA-E3              0.0              0.0
```

2.1.2 `guide` modality

The `guide` modality needs to have a `.X` matrix containing the gRNA UMI counts, as well as a `.layers['guide_assignment']` matrix containing the binary gRNA assignments.

```
input_minimal['guide'].X
```

```
## <9704x55 sparse matrix of type '<class 'numpy.float64'>'
##  with 11868 stored elements in Compressed Sparse Row format>
```

```
input_minimal['guide'].layers['guide_assignment']
```

```
## <9704x55 sparse matrix of type '<class 'numpy.float64'>'
##  with 10563 stored elements in Compressed Sparse Row format>
```

We can view a couple rows and columns of each:

```
cell_ids = [
    "GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2",
    "GGTGAAGCACCAGGCT-1_1A_6_SI-GA-E7"
]
grna_ids = [
  "GCCCTGCTACCCACTTACAG",
  "ATGTAGAAGGAGACACCGGG"
]
pd.DataFrame(input_minimal['guide'][cell_ids, grna_ids].X.toarray(),
             index = cell_ids,
             columns = grna_ids)
```

```
##                                  GCCCTGCTACCCACTTACAG  ATGTAGAAGGAGACACCGGG
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2                  9.0                  0.0
## GGTGAAGCACCAGGCT-1_1A_6_SI-GA-E7                  0.0                 18.0
```

```
pd.DataFrame(input_minimal['guide'][cell_ids, grna_ids].layers['guide_assignment'].toarray(),
             index = cell_ids,
             columns = grna_ids)
```

```
##                                  GCCCTGCTACCCACTTACAG  ATGTAGAAGGAGACACCGGG
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2                  1.0                  0.0
## GGTGAAGCACCAGGCT-1_1A_6_SI-GA-E7                  0.0                  1.0
```

In addition to the guide UMI counts and assignments, the `guide` modality must contain certain metadata information. This includes a `.var` data frame containing at least the binary variable `targeting` (`TRUE` if the guide targets a genomic element of interest or `FALSE` if it is safe- or non-targeting) and the string `intended_target_name` (the name of the genomic element targeted by the guide).

```
input_minimal['guide'].var.iloc[[0, 1, 20, 21, 30, 31]]
```

```
##                      targeting intended_target_name
## ATGTAGAAGGAGACACCGGG      TRUE      ENSG00000012660
## GCGCAGAGGCGGATGTAGAG      TRUE      ENSG00000012660
## ACACCCTCATTAGAACCCAG      TRUE      candidate_enh_1
## TTAAGAGCCTCGGTTCCCCT      TRUE      candidate_enh_1
## GACCTCCTGTGATCAGGTGG     FALSE        non-targeting
## ATTGGTATCCGTATAAGCAG     FALSE        non-targeting
```

Note that the `targeting` column is a string rather than a Boolean due to type compatibility issues involving R, Python, and HDF5. It can be cast to a Boolean if desired.

Finally, the guide modality must contain `uns` fields called `moi` (`low` or `high`) and `capture_method` (`"CROP-seq"` or `"direct capture"`):

```
input_minimal['guide'].uns['capture_method'][0]
```

```
## 'CROP-seq'
```

```
input_minimal['guide'].uns['moi'][0]
```

```
## 'high'
```

2.1.3 Global `.uns`

The input `MuData` object is also required to have a global `.uns` field named `pairs_to_test`, which is a data frame containing the pairs of elements (specified via `intended_target_name`) and genes (specified via `gene_id`) for which the inference is to be performed.

```
pd.DataFrame(input_minimal.uns['pairs_to_test'])
```

```
##              gene_id intended_target_name
## 0    ENSG00000187109      ENSG00000187109
## 1    ENSG00000114850      ENSG00000114850
## 2    ENSG00000134851      ENSG00000134851
## 3    ENSG00000163866      ENSG00000163866
## 4    ENSG00000181610      ENSG00000181610
## ..               ...                  ...
## 105  ENSG00000106789      candidate_enh_2
## 106  ENSG00000125482      candidate_enh_3
## 107  ENSG00000095380      candidate_enh_2
## 108  ENSG00000158941      candidate_enh_1
## 109  ENSG00000167123      candidate_enh_3
##
## [110 rows x 2 columns]
```

2.2 Optional input fields

Next we consider optional fields that can be included in the input `MuData` object.

```
mudata_input_fp = "data/gasperini_inference_input.h5mu"
input_optional = md.read_h5mu(mudata_input_fp)
input_optional
```

```
## MuData object with n_obs × n_vars = 9704 × 167
##   obs:   'prep_batch', 'within_batch_chip', 'within_chip_lane'
##   uns:   'pairs_to_test'
##   2 modalities
##     gene:    9704 x 112
##       obs:   'num_expressed_genes', 'total_gene_umis'
##       var:   'symbol', 'gene_chr', 'gene_start', 'gene_end'
##     guide:   9704 x 55
##       obs:   'num_expressed_guides', 'total_guide_umis'
##       var:   'targeting', 'intended_target_name', 'intended_target_chr', 'intended_target_start', 'i
##       uns:   'capture_method', 'moi'
##       layers:    'guide_assignment'
```

2.2.1 `gene` modality

The `MuData` object may include cellwise covariates for the `gene` modality in `.mod['gene'].obs`, such as number of genes with nonzero UMI counts (`num_expressed_genes`) and total RNA UMIs (`total_gene_umis`):

```
input_optional['gene'].obs
```

```
##                                  num_expressed_genes  total_gene_umis
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2                   41            280.0
## AGCTTGATCGAGAGCA-1_1A_2_SI-GA-E3                   35            192.0
## CCCAATCTCCTCAATT-1_1B_1_SI-GA-F2                   41            781.0
## CGCGGTACACTGTCGG-1_1A_2_SI-GA-E3                   37            189.0
## GGACGTCTCATGTCTT-1_1B_8_SI-GA-F9                   32            262.0
## ...                                               ...              ...
## CGCTATCTCTATCGCC-1_2A_4_SI-GA-G5                   23            203.0
## TCACAAGCAGCCTTGG-1_2A_6_SI-GA-G7                   30            173.0
## GCTGCAGGTGAAGGCT-1_2B_6_SI-GA-H7                   37            428.0
## GGATTACCATGTTGAC-1_2A_4_SI-GA-G5                   47            658.0
```

```
## GTGCTTCTCGGATGTT-1_2A_1_SI-GA-G2                  23           166.0
##
## [9704 rows x 2 columns]
```

The `MuData` object may include per-gene metadata in `.mod['gene'].var`, such as the HGNC gene symbol (`symbol`), the gene chromosome (`chr`), start (`gene_start`), and end (`gene_end`) coordinates:

```
input_optional['gene'].var
```

```
##                     symbol gene_chr   gene_start     gene_end
## ENSG00000008853    RHOBTB2     chr8   22844930.0   22844931.0
## ENSG00000104679     R3HCC1     chr8   23145421.0   23145422.0
## ENSG00000104689  TNFRSF10A     chr8   23082573.0   23082574.0
## ENSG00000120889  TNFRSF10B     chr8   22926533.0   22926534.0
## ENSG00000120896     SORBS3     chr8   22409208.0   22409209.0
## ...                    ...      ...          ...          ...
## ENSG00000114850       SSR3     chr3  156271913.0  156271914.0
## ENSG00000072274       TFRC     chr3  195808960.0  195808961.0
## ENSG00000134851    TMEM165     chr4   56262124.0   56262125.0
## ENSG00000198899                               NaN          NaN
## ENSG00000228253                               NaN          NaN
##
## [112 rows x 4 columns]
```

2.2.2 `guide` modality

The `MuData` object may include cellwise covariates for the `guide` modality in `.mod['guide'].obs`, such as number of guides with nonzero UMI counts (`num_expressed_guides`) and total guide UMIs (`total_guide_umis`):

```
input_optional['guide'].obs
```

```
##                                  num_expressed_guides  total_guide_umis
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2                     1               9.0
## AGCTTGATCGAGAGCA-1_1A_2_SI-GA-E3                     1              18.0
## CCCAATCTCCTCAATT-1_1B_1_SI-GA-F2                     1              24.0
## CGCGGTACACTGTCGG-1_1A_2_SI-GA-E3                     1              26.0
## GGACGTCTCATGTCTT-1_1B_8_SI-GA-F9                     1              12.0
## ...                                                ...               ...
## CGCTATCTCTATCGCC-1_2A_4_SI-GA-G5                     1               5.0
## TCACAAGCAGCCTTGG-1_2A_6_SI-GA-G7                     1              39.0
## GCTGCAGGTGAAGGCT-1_2B_6_SI-GA-H7                     1              21.0
## GGATTACCATGTTGAC-1_2A_4_SI-GA-G5                     1              73.0
## GTGCTTCTCGGATGTT-1_2A_1_SI-GA-G2                     1              12.0
##
## [9704 rows x 2 columns]
```

The `MuData` object may include per-guide metadata in `.mod['guide'].var` in addition to the required `targeting` and `intended_target_name` fields, such as the chromosome (`intended_target_chr`), start (`intended_target_start`), and end (`intended_target_end`) of the targeted element:

```
input_optional['guide'].var.iloc[[0, 1, 20, 21, 30, 31]]
```

```
##                     targeting intended_target_name intended_target_chr  \
## ATGTAGAAGGAGACACCGGG      TRUE      ENSG00000012660               chr6
## GCGCAGAGGCGGATGTAGAG      TRUE      ENSG00000012660               chr6
## ACACCCTCATTAGAACCCAG      TRUE      candidate_enh_1               chr8
## TTAAGAGCCTCGGTTCCCCT      TRUE      candidate_enh_1               chr8
```

```
## GACCTCCTGTGATCAGGTGG     FALSE        non-targeting
## ATTGGTATCCGTATAAGCAG     FALSE        non-targeting
##
##                       intended_target_start  intended_target_end
## ATGTAGAAGGAGACACCGGG              53213723.0           53213738.0
## GCGCAGAGGCGGATGTAGAG              53213738.0           53213754.0
## ACACCCTCATTAGAACCCAG              23366136.0           23366564.0
## TTAAGAGCCTCGGTTCCCCT              23366564.0           23366992.0
## GACCTCCTGTGATCAGGTGG                    -9.0                 -9.0
## ATTGGTATCCGTATAAGCAG                    -9.0                 -9.0
```

2.2.3 Global `.obs`

Optionally, the `MuData` input object can contain a global `obs` field containing cell-level information that is not specific to modality, such as batch information. Here is what it looks like for the Gasperini data:

```
input_optional.obs[['prep_batch', 'within_batch_chip', 'within_chip_lane']]
```

```
##                                    prep_batch    within_batch_chip  \
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2  prep_batch_1  within_batch_chip_B
## AGCTTGATCGAGAGCA-1_1A_2_SI-GA-E3  prep_batch_1  within_batch_chip_A
## CCCAATCTCCTCAATT-1_1B_1_SI-GA-F2  prep_batch_1  within_batch_chip_B
## CGCGGTACACTGTCGG-1_1A_2_SI-GA-E3  prep_batch_1  within_batch_chip_A
## GGACGTCTCATGTCTT-1_1B_8_SI-GA-F9  prep_batch_1  within_batch_chip_B
## ...                                        ...                  ...
## CGCTATCTCTATCGCC-1_2A_4_SI-GA-G5  prep_batch_2  within_batch_chip_A
## TCACAAGCAGCCTTGG-1_2A_6_SI-GA-G7  prep_batch_2  within_batch_chip_A
## GCTGCAGGTGAAGGCT-1_2B_6_SI-GA-H7  prep_batch_2  within_batch_chip_B
## GGATTACCATGTTGAC-1_2A_4_SI-GA-G5  prep_batch_2  within_batch_chip_A
## GTGCTTCTCGGATGTT-1_2A_1_SI-GA-G2  prep_batch_2  within_batch_chip_A
##
##                                   within_chip_lane
## GCTTGAATCGAATGCT-1_1B_1_SI-GA-F2  within_chip_lane_1
## AGCTTGATCGAGAGCA-1_1A_2_SI-GA-E3  within_chip_lane_2
## CCCAATCTCCTCAATT-1_1B_1_SI-GA-F2  within_chip_lane_1
## CGCGGTACACTGTCGG-1_1A_2_SI-GA-E3  within_chip_lane_2
## GGACGTCTCATGTCTT-1_1B_8_SI-GA-F9  within_chip_lane_8
## ...                                            ...
## CGCTATCTCTATCGCC-1_2A_4_SI-GA-G5  within_chip_lane_4
## TCACAAGCAGCCTTGG-1_2A_6_SI-GA-G7  within_chip_lane_6
## GCTGCAGGTGAAGGCT-1_2B_6_SI-GA-H7  within_chip_lane_6
## GGATTACCATGTTGAC-1_2A_4_SI-GA-G5  within_chip_lane_4
## GTGCTTCTCGGATGTT-1_2A_1_SI-GA-G2  within_chip_lane_1
##
## [9704 rows x 3 columns]
```

2.2.4 Pairs to test

Optionally, `.uns['pairs_to_test']` can have a third column: `pair_type`:

```
pd.DataFrame(input_optional.uns['pairs_to_test'])
```

```
##             gene_id intended_target_name         pair_type
## 0    ENSG00000187109      ENSG00000187109  positive_control
## 1    ENSG00000114850      ENSG00000114850  positive_control
## 2    ENSG00000134851      ENSG00000134851  positive_control
```

```
## 3    ENSG00000163866      ENSG00000163866  positive_control
## 4    ENSG00000181610      ENSG00000181610  positive_control
## ..               ...                  ...               ...
## 105  ENSG00000106789      candidate_enh_2         discovery
## 106  ENSG00000125482      candidate_enh_3         discovery
## 107  ENSG00000095380      candidate_enh_2         discovery
## 108  ENSG00000158941      candidate_enh_1         discovery
## 109  ENSG00000167123      candidate_enh_3         discovery
##
## [110 rows x 3 columns]
```

This optional column classifies pairs based on whether they are intended to be positive controls (an association is known to exist), negative controls (an association is known not to exist), or discovery pairs (pairs where it is unknown whether an association exists). This information need not be used by the inference module, but it is useful for downstream analysis.

2.3 Output fields

The output should be the same `MuData` object as the input, with the addition of a `test_results` field to the global `.uns`:

```
mudata_output_fp = "data/gasperini_inference_output.h5mu"
output_optional = md.read_h5mu(mudata_output_fp)
output_optional
```

```
## MuData object with n_obs × n_vars = 9704 × 167
##   obs:   'prep_batch', 'within_batch_chip', 'within_chip_lane'
##   uns:   'pairs_to_test', 'test_results'
##   2 modalities
##     gene:    9704 x 112
##       obs:   'num_expressed_genes', 'total_gene_umis'
##       var:   'symbol', 'gene_chr', 'gene_start', 'gene_end'
##     guide:   9704 x 55
##       obs:   'num_expressed_guides', 'total_guide_umis'
##       var:   'targeting', 'intended_target_name', 'intended_target_chr', 'intended_target_start', 'i
##       uns:   'capture_method', 'moi'
##       layers:    'guide_assignment'
```

```
pd.DataFrame(output_optional.uns['test_results'])
```

```
##              gene_id intended_target_name   log2_fc       p_value  \
## 0    ENSG00000187109      ENSG00000187109 -0.774367  3.217223e-85
## 1    ENSG00000114850      ENSG00000114850 -1.849572  2.414163e-79
## 2    ENSG00000134851      ENSG00000134851 -0.893860  4.309833e-50
## 3    ENSG00000163866      ENSG00000163866 -1.223700  4.704066e-49
## 4    ENSG00000181610      ENSG00000181610 -1.314285  3.766690e-42
## ..               ...                  ...       ...           ...
## 105  ENSG00000106789      candidate_enh_2  0.079632  6.660000e-01
## 106  ENSG00000125482      candidate_enh_3  0.144014  8.900000e-01
## 107  ENSG00000095380      candidate_enh_2 -0.165492  3.400000e-02
## 108  ENSG00000158941      candidate_enh_1  0.117617  7.980000e-01
## 109  ENSG00000167123      candidate_enh_3 -0.482057  8.800000e-02
##
##             pair_type
## 0    positive_control
## 1    positive_control
```

```
## 2    positive_control
## 3    positive_control
## 4    positive_control
## ..               ...
## 105         discovery
## 106         discovery
## 107         discovery
## 108         discovery
## 109         discovery
##
## [110 rows x 5 columns]
```

This is a data frame containing the same columns as the `pairs_to_test` data frame, plus at least one column containing a measure of the association for each pair. These columns can be `p_value`, `log2_fc`, `posterior_probability`, or any other measure of association.

3 Sample submission

Here we present a sample Jamboree submission.

3.1 Function

Here is a sample function that computes a p-value based on a Wilcoxon test:

3.2 Demonstration

Here is a demonstration of this function on the Gasperini data: