PolyA

Release 1.1.0

Kaitlin Carey, Audrey Shingleton, George Lesica, Travis Wheeler

Apr 25, 2023

CONTENTS

Welcome to the PolyA documentation!

CHAPTER

ONE

FULL CONTENTS

1.1 About

Preprint on bioRxiv https://www.biorxiv.org/content/10.1101/2021.02.13.430877v1/

Annotation of a biological sequence is usually performed by aligning that sequence to a database of known sequence elements. When that database contains elements that are highly similar to each other, the proper annotation may be ambiguous, because several entries in the database produce high-scoring alignments. Typical annotation methods work by assigning a label based on the candidate annotation with the highest alignment score; this can overstate annotation certainty, mislabel boundaries, and fails to identify large scale rearrangements or insertions within the annotated sequence.

PolyA is a software tool that adjudicates between competing alignment-based annotations by computing estimates of annotation confidence, identifying a trace with maximal confidence, and recursively splicing/stitching inserted elements. PolyA communicates annotation certainty, identifies large scale rearrangements, and detects boundaries between neighboring elements.

1.2 Installation

1.2.1 PyPI

PolyA may be consumed as an ordinary Python package and installed using the Pip tool:

```
pip install polyA

# Outside of a Virtual Environment, install as a user
# package, which doesn't require root.
pip install -U polyA
```

1.2.2 Docker

It is possible to run PolyA as a Docker container. This is simpler to install as it includes all required Python code and runtime dependencies.

The container image is on Docker Hub. See the README there for instructions on running it.

1.2.3 Runtime Dependencies

Easel

PolyA internally adjusts alignment scores to account for the scale multiplier implicit to each score matrix (the so-called *lambda* value). Unless the lambda value is given in the matrix file, PolyA must compute it; this is done using the `esl_scorematrix` utility found in the Easel software package.

To prepare that tool, follow these steps:

```
# get source code
cd $HOME/git  # or wherever you like to place repositories
git clone https://github.com/EddyRivasLab/easel
cd easel
autoconf
./configure
make
gcc -g -Wall -I. -L. \
    -o esl_scorematrix \
    -DeslSCOREMATRIX_EXAMPLE \
    esl_scorematrix.c \
    -leasel -lm

# add the easel directory to your path, e.g.
export PATH=$HOME/git/easel:$PATH
# you may wish to add this to your path permanently, e.g.
# https://opensource.com/article/17/6/set-path-linux

# alternatively, you can set the --easel-path argument to $HOME/git/easel
```

ULTRA

PolyA can optionally include tandem repeat annotations from ULTRA as competitors in the adjudication process. Options are:

(i) If you have an ultra annotation output on hand for the sequence being annotated, reference the ultra output file with the –ultra-data flag. For example:

```
polyA --ultra-data ultra.file ALIGNMENTS MATRICES
```

(ii) If you wish to run ULTRA on the fly, you'll need the software:

```
cd $HOME/git  # or wherever you like to place repositories
git clone git@github.com:TravisWheelerLab/ULTRA ultra
cd ultra
cmake .
make
```

After that, you'll reference the ultra binary in the PolyA command:

```
polyA --ultra-path $HOME/git/ultra/ultra \
      --sequences seq.fasta  ALIGNMENTS MATRICES
```

1.3 Running

Run PolyA from the command line:

```
polyA -h
```

or

```
python -m polyA -h
```

Command line usage is also included below for convenience.

```
usage: polyA [-h] [-v] [--chunk-size CHUNK_SIZE] [--trans-penalty TRANS_PENALTY]
          [--confidence] [--prior-counts FILE] [--shard-gap SHARD_GAP]
          [--sequences SEQS] [--ultra-data FILE] [--easel-path BIN]
          [--ultra-path BIN] [--log-file LOG] [--log-level LEVEL]
          [--matrix-position] [--output-path PATH] [--sequence-position] [--soda]
          ALIGNMENTS MATRICES

positional arguments:
  ALIGNMENTS              alignments file in Stockholm format

  MATRICES                substitution matrices file in PolyA matrix format

optional arguments:
  -h, --help              show this help message and exit
  -v, --version           show version and exit
  --chunk-size CHUNK_SIZE
                          size of the window in base pairs analyzed together
  --trans-penalty TRANS_PENALTY
                          penalty for changing annotations
  --confidence            run the confidence calculation and then exit
  --prior-counts FILE     file containing query genomic counts
  --shard-gap SHARD_GAP
                          maximum alignment gap before sharding occurs
  --sequences SEQS        FASTA file of the target sequence for using ULTRA
  --ultra-data FILE       text file of the output from ULTRA ran on the FASTA file
                          of the target sequence
  --easel-path BIN        path to the esl_scorematrix program, if necessary
                          (assumed to be in PATH)
  --ultra-path BIN        path to the ULTRA binary to use, if necessary (assumed
                          to be in PATH)
  --log-file LOG          file to store log output in, defaults to stderr
  --log-level LEVEL       logging level to use, 'debug' is the most noisy
  --matrix-position       produce output in terms of the matrix position
  --output-path PATH      directory to write output files to, defaults to
                          working directory
  --sequence-position     produce output in terms of the target sequence
                          position
  --soda                  write a SODA visualization file to the output
                          directory
  --complexity-adjustment complexity-adjust alignment scores (scores of matches
                          between sequence regions of biassed nucleotide
```

(continues on next page)

(continued from previous page)

```
composition are adjusted downwards)
```

1.3.1 Input Formats

PolyA accepts two required inputs and several optional inputs that affect its behavior. The required inputs are an alignment file which must contain alignments for all possible queries matching the target sequence. This file must be in Stockholm format with several custom metadata fields. The other required input is a set of substitution matrices. This file uses a custom, but extremely simple format.

Alignment File

Alignments for all possible queries matching the target sequence should be contained in a single file in Stockholm format.

There are several special metadata fields that must exist for each alignment in this file. See the example below. An explanation is indented to the right of each field with additional detail as noted.

```
#=GF ID  MERX#DNA/TcMar-Tigger    query sequence (1)
#=GF TR  chr1:11543-28567         target sequence
#=GF SC  1153                     alignment score
#=GF SD  +                        strand
#=GF TQ  -1                       (2)
#=GF ST  127                      alignment start position on target
#=GF SP  601                      alignment stop position on target
#=GF CST 135                      alignment start position on query
#=GF CSP 628                      alignment stop position on query
#=GF FL  128                      (3)
#=GF MX  matrix_name              (4)
#=GF GI -25                       gap init
#=GF GE -5                        gap extension
```

1. query sequence names must be in the format 'name#family/class'
2. valid values: 'q' if the alignment is on the reverse strand and the reversed sequence is the query; 't' if the alignment is on the reverse strand and the reversed sequence is the target; '-1' if the alignment is on the positive strand
3. the flanking region of the unaligned query sequence
4. the name of the substitution matrix file used to create alignment

Converting Alignments

PolyA can convert a Cross Match or Repeat Masker alignment file to the particular version of Stockholm format it requires. To do this, use either the `--cm-to-stockholm` or `--rm-to-stockholm` options, respectively, passing the path to the file to be converted.

The script will produce two files in the same directory as the input, one with a `.sto` extension and the other with a `.matrix` extension. These can be passed to the PolyA command line tool as `ALIGNMENTS` and `MATRICES`, respectively (see `--help`).

Example:

```
python -m polyA --cm-to-stockholm my_alignments.cm
```

Substitution Matrix File

The substitution matrix file example format (can include ambiguity codes):

- this file must include all of the matrices specified in the `#=GF MX` field of the alignment file, with corresponding and matching matrix names
- if lambda is not included polyA will use esl_scorematrix to calculate it for all matrices

```
matrix_name lambda(optional)
  A   G   C    T    N
  8  -6  -13  -15  -1
 -2  10  -13  -13  -1
-13  -13  10  -2   -1
-15  -13  -6   8   -1
 -1  -1   -1  -1   -1
//
matrix_name2 lambda2(optional)
  A   G   C    T    N
  8  -6  -13  -15  -1
 -2  10  -13  -13  -1
-13  -13  10  -2   -1
-15  -13  -6   8   -1
 -1  -1   -1  -1   -1
//
...
```

Sequence File

A FASTA file of the target sequence is needed when using ULTRA. The target sequence must be the same genomic region that was used to get the cross_match alignment file. This file must follow the format of

```
>chrom:start-end
target_sequence
```

as shown in the example below.

```
>chr1:152302175-152325203
AATAGTTTATTTTTAATTTAGATGCAGCTTACTATAATATTAATTATGTCCAAGATGATT
TTTTGAATACAGAATACTAGAATTCCAATAGAAGGATAATAGAGAAAGATGTGCTAGCCC
...
```

1.3.2 Output Formats

```
start   stop    ID  name
-----------------------------------------
11990879    11991268    eaa042dd09f944f68dba2fd4727c64e2    LTR40a#LTR/ERVL
11991272    11991444    fb5ef5e0e2ca4e05837ddc34ca7ef9e4    MSTA1#LTR/ERVL-MaLR
11991445    11991562    bdfc4039b7d947d0b25bf1115cc282ed    AluJr4#SINE/Alu
11991563    11991573    4871d91441a146209b98f645feae68c8    FLAM_C#SINE/Alu
11991574    11991818    fb5ef5e0e2ca4e05837ddc34ca7ef9e4    MSTB1#LTR/ERVL-MaLR
11991819    11991875    eaa042dd09f944f68dba2fd4727c64e2    LTR40a#LTR/ERVL

* Matching IDnums correspond to partial sequences that originate from
the same ancestral sequence.
```

Confidence Output Format

Computes confidence of a single input alignment region. Does not perform annotation or adjudication, simply outputs the confidence of all competing queries given in the input.

```
query_label         confidence
LTR40a#LTR/ERVL     0.875
LTR40b#LTR/ERVL     0.052
LTR40c#LTR/ERVL     0.001
...
```

1.3.3 Extensions

Visualizing Annotations

The command line option `--soda` will output the annotation data to a json file (output.0.viz) that can be used for visualization in SODA (linked below). The json file can be submitted on the browser to view the TE annotations from PolyA as well as the annotations from the UCSC Genome Browser for the same region of the human genome (hg38). The PolyA visualization can display the confidence values for all competing annotations of a selected region as well as their corresponding sequence alignments.

https://sodaviz.cs.umt.edu/polya-soda.html

Prior Counts Files

Default confidence calculations assume a uniform distribution over all competing queries. In the case of non uniform priors, the command line option –prior-counts prior_counts.txt includes prior genome counts in confidence calculations (see paper for more details).

https://www.biorxiv.org/content/10.1101/2021.02.13.430877v1

Prior counts file example format:

```
subfamily   genome_count
AluYk2      6855
LTR38           255
L1PA7_5end  13261
...
```

Using ULTRA

The optional use of ULTRA allows polyA to include tandem repeats (TRs) in the competing annotations of the target sequence. Doing so removes the dependency on pre-masking TRs prior to annotation, allows TRs to outcompete potentially weak fragmentary family annotation, and allows a family annotation to outcompete a TR. The command line option `--sequences seq.fasta` (with `--ultra-path` if necessary) will run ULTRA with polyA or `--ultra-data ultra_data.txt` can be used if ULTRA was ran on seq.fasta prior.

1.4 Tutorial

This tutorial covers running PolyA against region chr2:128348374-128348687.

1.4.1 Input Data

1. Get the FASTA file for the target region (chr2_128348379_128354757.fa)
2. Obtain the cross_match alignment file for this region (chr2_128348379_128354757.fa.cm)
3. **In the polyA directory, convert the cross_match file to stockholm format and get the required substitution matrices**
    - % polyA –cm-to-stockholm chr2_128348379_128354757.fa.cm
    - **This will output the following files needed to run polyA:**
        - chr2_128348379_128354757.fa.cm.sto
        - chr2_128348379_128354757.fa.cm.matrix

The files used here can found in the polyA tutorial directory.

1.4.2 Generating Output

Run polyA:
% polyA –sequence-position chr2_128348379_128354757.fa.cm.sto chr2_128348379_128354757.fa.cm.matrix

If you wish to include tandem repeats, you can run ULTRA on the fly:
% polyA –ultra-path /path/to/ultra –sequences chr2_128348379_128354757.fa chr2_128348379_128354757.fa.cm.sto chr2_128348379_128354757.fa.cm.matrix

or you can run ULTRA in advance:
% ./ultra -ss chr2_128348379_128354757.fa > chr2_128348379_128354757.fa.ultra

and then run polyA with the following command:
% polyA –sequence-position –ultra-data chr2_128348379_128354757.fa.ultra chr2_128348379_128354757.fa.cm.sto chr2_128348379_128354757.fa.cm.matrix

1.5 Development

This project uses Pipenv, which can be installed through Homebrew for Mac users. It must be installed before the Makefile targets, or the other commands listed in this document will work.

In order to run a command which relies on the project virtual environment, such as `python foo.py`, it is necessary to either run `pipenv shell` first, which will put you into a shell that has the correct Python installed, or prefix the command with `pipenv run` (e.g. `pipenv run python foo.py`).

```
# Get to work (from within project directory)
# This drops you into a shell inside the project virtual
# environment which means that commands that "pipenv run"
# may be elided from other commands.
pipenv shell
```

1.5.1 Makefile

A Makefile is available, run `make` or `make help` in the project root to see the available targets.

1.5.2 Dependencies

The Makefile has targets for managing dependencies, but it may be easier, particularly for Python developers, to simply invoke Pipenv directly. Example commands are listed below.

If you prefer not to use the Makefile, or if you need to add or remove dependencies, the following commands will allow you to manage dependencies.

```
# Fetch runtime dependencies
pipenv sync

# Fetch runtime and development dependencies
pipenv sync --dev

# Add a runtime dependency
pipenv install <package>

# Add a development dependency
pipenv install --dev <package>
```

1.5.3 Unit Tests

Unit tests use pytest. The linked documentation contains examples of how to use it to both write and run tests. The examples are quite extensive.

Documentation tests are also fine for simple cases.

```
# Run tests
pipenv run python -m pytest

# or
make check-fast check-slow
```

(continues on next page)

(continued from previous page)

```
# or
make check
```

1.5.4 Docker Container

There is a `Dockerfile` in the repo root. The image it describes is used for running tests in CI and can be used locally for convenience.

When dependencies change the image must be rebuilt and the new version pushed to Docker Hub. This can be done with `make container` if you have the correct permissions. Otherwise, ask a maintainer to do it for you.

There is also a file called `Dockerfile_run`. This is a runner image that is available for users who prefer to run PolyA through Docker (which is often more convenient). The runner image can be built with `tool/build-runner-image.sh` and, assuming sufficient permissions, pushed to Docker Hub with `tool/push-runner-image.sh`.

The scripts assume you are building the latest version of the image and so set the `latest` tag. If this is not the case, it is fairly simple to issue the correct Docker commands manually.

This image can be found on Docker Hub: https://hub.docker.com/repository/docker/traviswheelerlab/polya

1.5.5 Documentation

We use Sphinx for project documentation. Run `make docs` to update the documentation sources and build the HTML. Use `make docs-serve` to serve the documentation locally.

1.5.6 Release Process

Generating a new release is a four step process. The first step MUST be completed first, but the rest may be done in any order.

First, update the value of `VERSION` in `polyA/_version.py`, incrementing the various portions of the version number accordingly. We would like to follow Semantic Versioning, at least in general. Commit your changes to the *master* branch.

Second, create a "release" on GitHub. The tag should consist of a "v", followed immediately by the version number you chose above. For example: `v1.2.3`.

Third, run `make build-package`, followed by `make publish-package` assuming there aren't any build errors.

Fourth, and finally, create and push a new runner image to Docker Hub. Build the image by running `./tool/build-runner-image.sh`, and publish it with `./tool/push-runner-image.sh`. The version tag will be set automatically and the `latest` tag will also be updated.

1.6 polyA

1.6.1 polyA package

Subpackages

polyA.converters package

Submodules

polyA.converters.cm_to_stockholm module

polyA.converters.cm_to_stockholm.**convert**(*filename_cm: str*, *filename_out_sto: str*, *filename_out_matrix: str*)

polyA.converters.cm_to_stockholm.**get_alignment**(*alignment_array*)
takes cm alignment split on newline and return string of chrom seq and subfam seq

polyA.converters.cm_to_stockholm.**get_gap_penalties**(*file_contents*)
returns gap_init and gap_ext

polyA.converters.cm_to_stockholm.**get_info**(*info_array*)
takes info line split by white space and assigns all info fields to variables

polyA.converters.cm_to_stockholm.**get_score_matrix**(*file_contents*)
grabs score matrix info from alignment file, puts it in correct format, returns string

polyA.converters.cm_to_stockholm.**print_alignment**(*chrom_seq*, *subfam_seq*, *chrom*, *subfam*, *f_out_sto*)
takes chrom and subfam names and chrom and subfam seqs and prints in stockholm format

polyA.converters.cm_to_stockholm.**print_info**(*C*, *subfam*, *chrom*, *score*, *strand*, *start*, *stop*, *consensus_start*, *consensus_stop*, *flank*, *matrix_name*, *gap_init*, *gap_ext*, *f_out_sto*)
prints all info in correct format for stockholm

polyA.converters.cm_to_stockholm.**print_score_matrix**(*filename_out_matrix*, *score_matrix*, *matrix_name*, *background_freqs*)
print score matrix to its own output file with extension ".matrix"

polyA.converters.cm_to_stockholm.**read_file**(*filename_cm*)
opens file and returns contents of file as one string

polyA.converters.rm_to_stockholm module

polyA.converters.rm_to_stockholm.**convert**(*filename_rm: str*, *filename_out_sto: str*, *filename_out_matrix: str*)

polyA.converters.rm_to_stockholm.**get_alignment**(*alignment_array*)
takes cm alignment split on newline and return string of chrom seq and subfam seq

polyA.converters.rm_to_stockholm.**get_info**(*info_array*)
takes info line split by white space and assigns all info fields to variables

polyA.converters.rm_to_stockholm.**get_score_matrix**(*matrix_name*)

grabs score matrix info from original matrix file, puts it in correct format, returns string

polyA.converters.rm_to_stockholm.**print_alignment**(*chrom_seq*, *subfam_seq*, *chrom*, *subfam*, *f_out_sto*)

takes chrom and subfam names and chrom and subfam seqs and prints in stockholm format

polyA.converters.rm_to_stockholm.**print_info**(*C*, *subfam*, *chrom*, *score*, *strand*, *start*, *stop*, *consensus_start*, *consensus_stop*, *flank*, *matrix_name*, *f_out_sto*)

prints all info in correct format for stockholm

polyA.converters.rm_to_stockholm.**print_score_matrix**(*f_out_matrix*, *score_matrix*, *matrix_name*)

print score matrix to its output file with extension ".matrix"

polyA.converters.rm_to_stockholm.**read_file**(*filename_cm*)

opens file and returns contents of file as one string

Module contents

Submodules

polyA.alignment module

class polyA.alignment.**Alignment**(*subfamily: str*, *chrom_name: str*, *chrom_start: int*, *chrom_stop: int*, *score: int*, *start: int*, *stop: int*, *consensus_start: int*, *consensus_stop: int*, *sequences: List[str]*, *strand: str*, *flank: int*, *sub_matrix_name: str*, *gap_init: float*, *gap_ext: float*)

Bases: `tuple`

A container to hold data related to a single alignment.

```
>>> a = Alignment("", "ch", 25, 100, 0, 0, 0, 0, 0, [], "", 0, "", 0, 0)
>>> a.chrom_name
'ch'
>>> a.chrom_start
25
>>> a.chrom_stop
100
>>> a.chrom_length
76
```

property chrom_length: int

property chrom_name

Alias for field number 1

property chrom_start

Alias for field number 2

property chrom_stop

Alias for field number 3

property consensus_start

Alias for field number 7

property consensus_stop
Alias for field number 8

property flank
Alias for field number 11

property gap_ext
Alias for field number 14

property gap_init
Alias for field number 13

property score
Alias for field number 4

property sequence: str

property sequences
Alias for field number 9

property start
Alias for field number 5

property stop
Alias for field number 6

property strand
Alias for field number 10

property sub_matrix_name
Alias for field number 12

property subfamily
Alias for field number 0

property subfamily_sequence: str

polyA.alignment.**get_skip_state**()
Return the skip state singleton.

polyA.calc_repeat_scores module

polyA.calc_repeat_scores.**calculate_repeat_scores**(*tandem_repeats: List[*TandemRepeat*]*, *chunk_size: int*, *start_all: int*, *row_num: int*, *active_cells: Dict[int, List[int]]*, *align_matrix: Dict[Tuple[int, int], float]*, *consensus_matrix: Dict[Tuple[int, int], int]*, *shard_start: int*, *shard_stop: int*) → Dict[int, float]

Calculates score (according to ULTRA scoring) for every segment of size chunksize for all tandem repeats found in the target sequence. Scores are of the surrounding chunksize nucleotides in the sequence.

At same time, updates AlignMatrix, ActiveCells, and ConsensusMatrix to include tandem repeat scores.

input: tandem_repeats: list of tandem repeats from ULTRA output chunk_size: size of nucleotide chunks that are scored start_all: minimum starting nucleotide position row_num: number of rows in matrices columns: all columns that are not empty from alignment scores active_cells: dictionary that maps column number to a list of

active rows align_matrix consensus_matrix: dictionary that maps a tuple (row, col) to consensus position in the subfam/query sequence shard_start: start seq pos of current shard shard_stop: stop seq pos of current shard

output: repeat_scores: Hash implementation of sparse 1d array. Key is int that maps col index in the target sequence to it's tandem repeat score. Used in FillNodeConfidence.

tr_info - list of dictionaries - [{'Start': num, 'Length': num, 'PositionScoreDelta': '-0.014500:1.27896'}, ...] >>> repeats = [TandemRepeat.from_json(m) for m in [{'Start': 5, 'Length': 4, 'PositionScoreDelta': '-0.5:1:1.5:1', 'Consensus': 'AT'}, {'Start': 10, 'Length': 8, 'PositionScoreDelta': '0:0.5:0.5:1.5:1.5:1:0.5:-0.5', 'Consensus': 'GTT'}]] >>> align_mat = {} >>> active_cols = {} >>> rep_scores = calculate_repeat_scores(repeats, 5, 4, 1, active_cols, align_mat, {}, 1, 30) >>> rep_scores {2: -0.5, 3: 1.0, 4: 1.5, 5: 1.0, 7: 0.0, 8: 0.5, 9: 0.5, 10: 1.5, 11: 1.5, 12: 1.0, 13: 0.5, 14: -0.5} >>> align_mat {(1, 2): 3.3333333333333335, (1, 3): 3.75, (1, 4): 3.75, (1, 5): 5.833333333333333, (2, 7): 1.6666666666666667, (2, 8): 3.125, (2, 9): 4.0, (2, 10): 5.0, (2, 11): 5.0, (2, 12): 4.0, (2, 13): 3.125, (2, 14): 1.6666666666666667} >>> active_cols {2: [0, 1], 3: [0, 1], 4: [0, 1], 5: [0, 1], 7: [0, 2], 8: [0, 2], 9: [0, 2], 10: [0, 2], 11: [0, 2], 12: [0, 2], 13: [0, 2], 14: [0, 2]}

Shard cuts off TR from left side >>> repeats = [TandemRepeat.from_json(m) for m in [{'Start': 5, 'Length': 4, 'PositionScoreDelta': '-0.5:1:1.5:1', 'Consensus': 'AT'}, {'Start': 10, 'Length': 8, 'PositionScoreDelta': '0:0.5:0.5:1.5:1.5:1:0.5:-0.5', 'Consensus': 'GTT'}]] >>> align_mat = {} >>> active_cols = {} >>> rep_scores = calculate_repeat_scores(repeats, 5, 4, 1, active_cols, align_mat, {}, 6, 30) >>> rep_scores {3: 1.0, 4: 1.5, 5: 1.0, 7: 0.0, 8: 0.5, 9: 0.5, 10: 1.5, 11: 1.5, 12: 1.0, 13: 0.5, 14: -0.5} >>> align_mat {(1, 3): 3.75, (1, 4): 3.75, (1, 5): 5.833333333333333, (2, 7): 1.6666666666666667, (2, 8): 3.125, (2, 9): 4.0, (2, 10): 5.0, (2, 11): 5.0, (2, 12): 4.0, (2, 13): 3.125, (2, 14): 1.6666666666666667} >>> active_cols {3: [0, 1], 4: [0, 1], 5: [0, 1], 7: [0, 2], 8: [0, 2], 9: [0, 2], 10: [0, 2], 11: [0, 2], 12: [0, 2], 13: [0, 2], 14: [0, 2]}

Shard cuts off TR from right side in first half of chunk >>> repeats = [TandemRepeat.from_json(m) for m in [{'Start': 5, 'Length': 4, 'PositionScoreDelta': '-0.5:1:1.5:1', 'Consensus': 'AT'}, {'Start': 10, 'Length': 8, 'PositionScoreDelta': '0:0.5:0.5:1.5:1.5:1:0.5:-0.5', 'Consensus': 'GTT'}]] >>> align_mat = {} >>> active_cols = {} >>> rep_scores = calculate_repeat_scores(repeats, 5, 4, 1, active_cols, align_mat, {}, 1, 11) >>> rep_scores {2: -0.5, 3: 1.0, 4: 1.5, 5: 1.0, 7: 0.0, 8: 0.5} >>> align_mat {(1, 2): 3.3333333333333335, (1, 3): 3.75, (1, 4): 3.75, (1, 5): 5.833333333333333, (2, 7): 1.6666666666666667, (2, 8): 3.125} >>> active_cols {2: [0, 1], 3: [0, 1], 4: [0, 1], 5: [0, 1], 7: [0, 2], 8: [0, 2]}

polyA.calculate_score module

polyA.calculate_score.**calc_target_char_counts**(*query_seq: str*, *target_seq: str*, *target_chars: List[str]*) → Tuple[Dict[str, int], Set[str]]

Finds the char counts in the target sequence which will be used in the calculation to determine a character's contribution to the complexity adjusted score.

input: query_seq: query sequence from alignment target_seq: target sequence from alignment target_chars: chars in the target sequence that will contribute to the complexity adjusted score

output: target_char_counts: dictionary of the target chars to their count in the target sequence that will contribute to the adjusted score other_chars: other chars found in the target sequence that do not contribute to the complexity adjustment (gaps, padding)

polyA.calculate_score.**calculate_complexity_adjusted_score**(*char_background_freqs: Optional[Dict[str, float]]*, *query_seq: str*, *target_seq: str*, *lamb: float*) → Dict[str, float]

Calculates the per position contribution to the complexity adjusted score of the alignment for each valid char in the target sequence. This calculation is from the cross_match complexity adjustment functions:

$$t_i = \text{count of } i^{th} \text{ character}$$

$$p_i = \text{background frequency of } i^{th} \text{ character}$$

$$t_{factor} = \sum_{i=0}^{N} t_i ln(t_i) - \left[\left(\sum_{i=0}^{N} t_i \right) ln(\sum_{i=0}^{N} t_i) \right]$$

$$t_{sum} = \left[\sum_{i=0}^{N} t_i ln(p_i) \right] - t_{factor}$$

$$s_{adj} = s_{raw} + (\frac{t_{sum}}{\lambda}) + 0.999$$

input: char_background_freqs: background frequencies of the chars from the scoring matrix query_seq: query sequence from alignment target_seq: target sequence from alignment lamb: lambda value used from the scoring matrix

output: char_complexity_adjustments: a char's per position contribution to the complexity adjusted score

```
>>> query = "TCAGACTGTTCA-----ACTCACCTGGCAGCCACTTCCAGA"
>>> target = "TCAGACTGTTCATGAGTGCTCACCTGGTAGAGG-----AAA"
>>> lamb = 0.1227
>>> calculate_complexity_adjusted_score(None, query, target, lamb)
{'T': 0.0, 'C': 0.0, 'A': 0.0, 'G': 0.0, '-': 0.0}
```

```
>>> char_freqs = {'A': 0.295, 'G': 0.205, 'C': 0.205, 'T': 0.295}
>>> calculate_complexity_adjusted_score(char_freqs, query, target, lamb)
{'A': -0.08342236356495786, 'G': -0.11790190327726593, 'C': -0.11790190327726593, 'T
↪': -0.08342236356495786, '-': 0.0}
```

polyA.calculate_score.**calculate_score**(*gap_ext: float*, *gap_init: float*, *seq1: str*, *seq2: str*, *prev_char_seq1: str*, *prev_char_seq2: str*, *sub_matrix: Dict[str, int]*, *char_complexity_adjustment: Dict[str, float]*) → float

Calculate the score for an alignment between a subfamily and a target/chromsome sequence. Scores are calculated based on input sub_matrix, gap_ext, and gap_init.

prev_char_seq1, prev_char_seq2 are the single nucleotides in the alignment before the chunk - if chunk starts with '-' these tell us to use gap_init or gap_ext as the penalty

input: gap_ext: penalty to extend a gap gap_init: penalty to start a gap seq1: sequence chunk from alignment seq2: sequence chunk from alignment prev_char_seq1: character before alignment - tells if should use gap_ext or gap_init prev_char_seq2: character before alignment - tells if should use gap_ext or gap_init sub_matrix: input substitution matrix - dict that maps 2 chars being aligned to the score

output: alignment score

```
>>> sub_mat = {"AA":1, "AT":-1, "TA":-1, "TT":1}
>>> char_adjustments = {'T': 0.0, 'C': 0.0, 'A': 0.0, 'G': 0.0, '-': 0.0}
>>> calculate_score(-5, -25, "AT", "AT", "", "", sub_mat, char_adjustments)
2.0
>>> calculate_score(-5, -25, "-T", "AT", "A", "A", sub_mat, char_adjustments)
-24.0
>>> calculate_score(-5, -25, "-T", "AT", "-", "", sub_mat, char_adjustments)
-4.0
>>> char_adjustments = {'T': -0.4, 'C': 0.0, 'A': -0.2, 'G': 0.0, '-': 0.0}
>>> calculate_score(-5, -25, "-T", "AT", "A", "A", sub_mat, char_adjustments)
-24.4
>>> calculate_score(-5, -25, "-T", "AT", "-", "", sub_mat, char_adjustments)
-4.4
```

polyA.collapse_matrices module

polyA.collapse_matrices.**collapse_matrices**(*row_count: int*, *start_all: int*, *columns: List[int]*, *subfams: List[str]*, *strands: List[str]*, *starts: List[int]*, *stops: List[int]*, *active_cells: Dict[int, List[int]]*, *support_matrix: Dict[Tuple[int, int], float]*, *consensus_matrix: Dict[Tuple[int, int], int]*) → *CollapsedMatrices*

In all preceding matrices, collapse and combine rows that are the same subfam.

input: row_num: number of rows in matrices columns: list of non empty cols in matrices subfams: subfamily names for rows in matrices - each row is a different alignment strands: which strand each alignment is on active_cells: maps column number with rows that are active in that column support_matrix: uncollapsed support matrix - rows are number indices consensus_matrix: uncollapsed consensus matrix - rows are number indices

output: CollapsedMatrices container

```
>>> non_cols = [1, 2, 3]
>>> active = {1: [0, 1, 2], 2: [0, 1, 2], 3: [0, 1, 2]}
>>> subs = ["s1", "s2", "s3"]
>>> strandss = ["+", "-", "-"]
>>> sup_mat = {(0, 1): 0.5, (0, 2): 0.5, (0, 3): .1, (1, 1): 0.2, (1, 2): 0.2, (1,␣
↪3): .2, (2, 1): 0.1, (2, 2): 0.1, (2, 3): 0.9}
>>> con_mat = {(0, 1): 0, (0, 2): 1, (0, 3): 2, (1, 1): 0, (1, 2): 1, (1, 3): 2, (2,
↪ 1): 0, (2, 2): 3, (2, 3): 10}
>>> (r, con_mat_col, strand_mat_col, sup_mat_col, sub_col, active_col, sub_col_ind,␣
↪sub_aligns) = collapse_matrices(3, 1, non_cols, subs, strandss, [0,0,0], [2,2,2],␣
↪active, sup_mat, con_mat)
>>> r
3
>>> con_mat_col
{(0, 1): 0, (1, 1): 0, (2, 1): 0, (0, 2): 1, (1, 2): 1, (2, 2): 3, (0, 3): 2, (1,␣
↪3): 2, (2, 3): 10}
>>> strand_mat_col
{(0, 1): '+', (1, 1): '-', (2, 1): '-', (0, 2): '+', (1, 2): '-', (2, 2): '-', (0,␣
↪3): '+', (1, 3): '-', (2, 3): '-'}
>>> sup_mat_col
{(0, 1): 0.5, (1, 1): 0.2, (2, 1): 0.1, (0, 2): 0.5, (1, 2): 0.2, (2, 2): 0.1, (0,␣
↪3): 0.1, (1, 3): 0.2, (2, 3): 0.9}
>>> sub_col
['s1', 's2', 's3']
>>> active_col
{1: [0, 1, 2], 2: [0, 1, 2], 3: [0, 1, 2]}
>>> sub_col_ind
{'s1': 0, 's2': 1, 's3': 2}
>>> sub_aligns
{('s1', 1): (0, 0), ('s2', 1): (1, 0), ('s3', 1): (2, 0), ('s1', 2): (0, 1), ('s2',␣
↪2): (1, 1), ('s3', 2): (2, 3), ('s1', 3): (0, 2), ('s2', 3): (1, 2), ('s3', 3):␣
↪(2, 10)}
```

polyA.collapse_matrices.**dp_for_collapse**(*dp_rows: List[int]*, *active: Dict[int, List[int]]*, *support_matrix: Dict[Tuple[int, int], float]*, *non_empty: List[int]*) → List[int]

When there is more than one row with the same subfam label, do a mini DP trace to choose which row to use in collapsed matrices

input: dp_rows: rows in original matrices to be collapsed support_matrix: uncollapsed support matrix, use these

score for dp columns: list of non empty cols in matrices

output: path: list with labels for which row to collapse to for each column non_empty: list of non empty cols for this particular subfam (in the mini dp matrix)

```
>>> dp_r = [0, 2]
>>> act = {0:[0,2], 2:[0,2], 3:[0,2]}
>>> non_cols = [0, 2, 3]
>>> sup_mat = {(0, 0): 0.5, (0, 2): 0.5, (0, 3): .1, (2, 0): 0.1, (2, 2): 0.1, (2,␣
↪3): 0.9}
>>> p = dp_for_collapse(dp_r, act, sup_mat, non_cols)
>>> p
[2, 2, 2]
```

polyA.confidence_cm module

polyA.confidence_cm.**confidence_cm**(*region: List[float]*, *subfam_counts: Dict[str, float]*, *subfams: List[str]*, *subfam_rows: List[int]*, *repeats: int*, *node_confidence_bool: bool*) → List[float]

Computes confidence values for competing annotations using alignment and tandem repeat scores. Loops through the array once to find sum of 2^every_hit_score in region, then loops back through to calculate confidence. Converts the alignment score to account for lambda before summing.

If command line option for subfam_counts, this is included in confidence math.

input: region: list of scores for competing annotations subfam_counts: dict that maps subfam name to it's count info subfams: list of subfam names subfam_rows: list of subfamily rows that correspond to the the subfams of the region scores repeats: number of tandem repeat scores found at the end of the region node_confidence_bool: if False (0) - confidence for filling DP matrices, if True - confidence for nodes

output: confidence_list: list of confidence values for competing annotations, each input alignment and tandem repeat score will have one output confidence score

```
>>> counts = {"s1": .33, "s2": .33, "s3": .33}
>>> subs = ["s1", "s2", "s3"]
>>> conf = confidence_cm([2, 1, 1], counts, subs, [0, 1, 2], 0, False)
>>> f"{conf[0]:.2f}"
'0.50'
>>> f"{conf[1]:.2f}"
'0.25'
>>> f"{conf[2]:.2f}"
'0.25'
```

```
>>> conf = confidence_cm([0, 100, 100], 0, subs, [0, 1, 2], 0, False)
>>> f"{conf[0]:.2f}"
'0.01'
```

```
>>> conf = confidence_cm([0, 100, 100], 0, subs, [0, 1, 2], 0, True)
>>> f"{conf[0]:.2f}"
'0.00'
```

```
>>> counts = {"s1": .31, "s2": .31, "s3": .31, "Tandem Repeat": .06}
>>> subs = ["s1", "s2", "s3", "Tandem Repeat"]
```

(continues on next page)

(continued from previous page)

```
>>> conf = confidence_cm([2, 1, 0.7], counts, subs, [0, 1, 3], 1, False)
>>> f"{conf[0]:.2f}"
'0.65'
>>> f"{conf[1]:.2f}"
'0.32'
>>> f"{conf[2]:.2f}"
'0.03'
```

polyA.confidence_cm.**confidence_only**(*region: List[int]*, *lambs: List[float]*) → List[float]

Computes confidence values for competing annotations using alignment scores. Loops through the array once to find sum of 2^every_hit_score in region, then loops back through to calculate confidence. Converts the alignment score to account for lambda before summing.

input: lambdaa: lambda for score matrix used region: list of scores for competing annotations subfams: list of subfam names

output: confidence_list: list of confidence values for competing annotations

```
>>> reg = [100, 55, 1]
>>> lambs = [.1227] * 3
>>> conf = confidence_only(reg, lambs)
>>> conf
[0.9843787551069454, 0.015380918048546022, 0.0002403268445085316]
```

polyA.constants module

polyA.constants.**CROSS_MATCH_ADJUSTMENT = 0.999**

A constant value given from cross_match that is used to compute the complexity adjusted score of an alignment.

polyA.constants.**DEFAULT_CHUNK_SIZE = 31**

The width of the "window" used to break a sequence up into segments. Measured in base pairs.

polyA.constants.**DEFAULT_SHARD_GAP = -1**

Allowed gap between sequences for them to be included in the same shard.

polyA.constants.**DEFAULT_TRANS_PENALTY = 55**

The positive exponent value for the base probability of changing annotations. A value of 55 means the probability will be 1e-55.

polyA.constants.**INFINITE_SHARD_GAP = -1**

A magic value that indicates no sharding should occur.

polyA.constants.**NAN_STRING = 'NaN'**

The string used to represent "not a number" for serialization and deserialization.

polyA.constants.**PROB_SKIP = 0.4**

used for prior counts, about 60% of genome is TE derived, so the skip state will have a prob of 40%

polyA.constants.**PROB_SKIP_TR = 0.06**

About 6% of the genome is expected to be tandem repeats.

polyA.constants.**SAME_PROB_LOG = 0.0**

penalty for staying in the same state in the DP. mathematically not exactly 0, but in python log(1-10e-45) = 0 so set to 0 and avoid doing the math

polyA.constants.**SKIP_ALIGN_SCORE = 10.0**

alignment score given to skip state, no lambda adjustment needed

polyA.constants.**START_ID = 1111**

The arbitrary temporary ID used to bootstrap the graph process. The value doesn't matter, but it's here for clarity.

polyA.edges module

polyA.edges.**edges**(*starts: List[int]*, *stops: List[int]*) → Tuple[int, int]

Find and return the min start and max stop positions for the entire region included in the alignments.

Inputs:

starts - start positions on the target sequence from the input alignment stops - stop positions on the target sequence from the input alignment

Outputs:

minimum and maximum start and stop positions on the chromosome/target sequences for whole alignment

```
>>> b, e = edges(starts=[0, 1, 4, 7], stops=[0, 3, 10, 9])
>>> b
1
>>> e
10
```

polyA.exceptions module

exception polyA.exceptions.**FileFormatException**(*path: str*, *line_number: int*, *message: str = ''*)

Bases: Exception

An exception that should be raised whenever we attempt to parse an input file and find that it is in the incorrect format.

line_number: int

message: str

path: str

polyA.extract_nodes module

polyA.extract_nodes.**extract_nodes**(*nodes: int*, *columns: List[int]*, *changes_position: List[int]*, *path_graph: List[int]*) → None

finds nodes that only have one (or less) incoming and one (or less) outgoing edge and adds them to remove_starts and remove_stops so they can be extracted from the alignment - all columns in removed nodes are removed from NonEmptyColumns (columns) so during next round of DP calculations, those nodes are ignored and surrounding nodes can be stitched if necessary,

input: num_col: number of columns in matrices nodes: number of nodes in graph columns: non empty columns in matrices changes_position: node boundaries path_graph: 2D array that represents edges in the graph

output: updated_num_col: updated number of columns after nodes are extracted

```
>>> non_cols = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> change_pos = [0, 3, 7]
>>> path_graph = [0, 1, 1, 0, 0, 1, 0, 0, 0]
>>> extract_nodes(3, non_cols, change_pos, path_graph)
>>> non_cols
[0, 1, 2, 7, 8, 9]
```

polyA.fill_align_matrix module

polyA.fill_align_matrix.**fill_align_matrix**(*lambda_values: List[float]*, *column_count: int*, *edge_start: int*, *chunk_size: int*, *gap_inits: List[float]*, *gap_exts: List[float]*, *skip_align_score: float*, *subfams: List[str]*, *chroms: List[str]*, *starts: List[int]*, *stops: List[int]*, *sub_matrices: List[Dict[str, int]]*, *background_freqs: List[Optional[Dict[str, float]]]*) → Dict[Tuple[int, int], float]

Fills an alignment score matrix by calculating an alignment score (according to crossmatch scoring) for every segment of size *chunk_size* for all sequences.

Scores are based on the surrounding *chunk_size* nucleotides in the alignment. Ex: column 15 in the matrix holds the aligment score for nucleotides at positons 0 - 30, assuming *chunk_size* = 31.

Starting and trailing cells are treated differently. For example, column 0 in the matrix holds the alignment score for nucleotides 0 - 15, column 1 represents nucleotides 0 - 16, etc. Scores are weighted based on number of nucleotides that contribute to the score.

This algorithm ignores all padded indices (".") in the chroms and subfams lists.

Speed up by computing base score for the first segment, moves to next column but adding next chars score to base score and subtracting previous chars score from base score. Restarts and recomputes new base score when necessary.

Inputs:

everything needed for CalcScore() edge_start: where alignment starts on the target/chrom sequence chunk_size: size of nucletide chunks that are scored skip_align_score: alignment score to give the skip state (default = 0) subfams: alignments to calculate scores from chroms: alignments to calculate scores from starts: where in the target sequence the competing alignments start

Outputs:

align_matrix: Hash implementation of sparse 2D matrix used in pre-DP calculations. Key is tuple[int, int] that maps row, col to the value held in that cell of matrix. Rows are subfamilies in the input alignment file, cols are nucleotide positions in the target sequence.

```
>>> chros = ["", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAT..............",
↪"TAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAT..............."]
>>> subs = ["", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA..............",
↪"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA--A..............."]
>>> strts = [0, 2, 0]
>>> stps = [0, 39, 39]
>>> sub_mat = [{"AA":1, "AT":-1, "TA":-1, "TT":1, "..":0}] * 3
>>> back_freqs = [None] * 3
>>> m = fill_align_matrix([0.0, 0.1, 0.1], 41, 0, 31, [0, -25, -25], [0, -5, -5], 1.
↪0, subs, chros, strts, stps, sub_mat, back_freqs)
>>> m
```

(continues on next page)

(continued from previous page)

```
{(1, 3): 3.1, (1, 4): 3.1, (1, 5): 3.1, (1, 6): 3.1, (1, 7): 3.1, (1, 8): 3.1, (1,␣
↪9): 3.1, (1, 10): 3.1, (1, 11): 3.1, (1, 12): 3.1, (1, 13): 3.1, (1, 14): 3.1, (1,
↪ 15): 3.1, (1, 16): 3.1, (1, 17): 3.1, (1, 18): 3.1, (1, 19): 3.1, (1, 20): 3.1,␣
↪(1, 21): 3.1, (1, 22): 3.1, (1, 23): 3.1, (1, 24): 3.1, (1, 25): 2.
↪9000000000000004, (1, 26): 2.8933333333333335, (1, 27): 2.8862068965517245, (1,␣
↪28): 2.878571428571429, (1, 29): 2.8703703703703702, (1, 30): 2.861538461538462,␣
↪(1, 31): 2.8520000000000003, (1, 32): 2.841666666666667, (1, 33): 2.
↪8304347826086955, (1, 34): 2.8181818181818183, (1, 35): 2.804761904761905, (1,␣
↪36): 2.7900000000000005, (1, 37): 2.7736842105263158, (1, 38): 2.7555555555555555,
↪ (1, 39): 2.735294117647059, (1, 40): 2.7125000000000004, (2, 1): 2.
↪7125000000000004, (2, 2): 2.735294117647059, (2, 3): 2.755555555555556, (2, 4): 2.
↪7736842105263158, (2, 5): 2.79, (2, 6): 2.804761904761905, (2, 7): 2.
↪8181818181818183, (2, 8): 2.830434782608696, (2, 9): 2.841666666666667, (2, 10):␣
↪2.8520000000000003, (2, 11): 2.861538461538462, (2, 12): 2.8703703703703702, (2,␣
↪13): 2.8785714285714286, (2, 14): 2.8862068965517245, (2, 15): 2.8933333333333335,
↪ (2, 16): 2.9000000000000004, (2, 17): 3.1, (2, 18): 3.1, (2, 19): 3.1, (2, 20):␣
↪3.1, (2, 21): 3.1, (2, 22): 3.1, (2, 23): 0.5, (2, 24): -0.1, (2, 25): -0.
↪30000000000000004, (2, 26): -0.41333333333333333, (2, 27): -0.5344827586206897,␣
↪(2, 28): -0.6642857142857143, (2, 29): -0.8037037037037037, (2, 30): -0.
↪9538461538461539, (2, 31): -1.116, (2, 32): -1.291666666666667, (2, 33): -1.
↪482608695652174, (2, 34): -1.6909090909090907, (2, 35): -1.9190476190476191, (2,␣
↪36): -2.17, (2, 37): -2.447368421052632, (2, 38): -2.755555555555555, (2, 39): -
↪3.1, (2, 40): -3.4875000000000003, (0, 0): 1.0, (0, 1): 1.0, (0, 2): 1.0, (0, 3):␣
↪1.0, (0, 4): 1.0, (0, 5): 1.0, (0, 6): 1.0, (0, 7): 1.0, (0, 8): 1.0, (0, 9): 1.0,
↪ (0, 10): 1.0, (0, 11): 1.0, (0, 12): 1.0, (0, 13): 1.0, (0, 14): 1.0, (0, 15): 1.
↪0, (0, 16): 1.0, (0, 17): 1.0, (0, 18): 1.0, (0, 19): 1.0, (0, 20): 1.0, (0, 21):␣
↪1.0, (0, 22): 1.0, (0, 23): 1.0, (0, 24): 1.0, (0, 25): 1.0, (0, 26): 1.0, (0,␣
↪27): 1.0, (0, 28): 1.0, (0, 29): 1.0, (0, 30): 1.0, (0, 31): 1.0, (0, 32): 1.0,␣
↪(0, 33): 1.0, (0, 34): 1.0, (0, 35): 1.0, (0, 36): 1.0, (0, 37): 1.0, (0, 38): 1.
↪0, (0, 39): 1.0, (0, 40): 1.0}
```

polyA.fill_confidence_matrix module

polyA.fill_confidence_matrix.**fill_confidence_matrix**(*column_count: int*, *subfam_counts: Dict[str, float]*, *subfams: List[str]*, *active_cells: Dict[int, List[int]]*, *align_matrix: Dict[Tuple[int, int], float]*) → Dict[Tuple[int, int], float]

Fills confidence matrix from alignment matrix. Each column in the alignment matrix is a group of competing annotations that are input into confidence_cm, the output confidence values are used to populate confidence_matrix.

Inputs:

columns - non empty matrix column indices subfam_counts - mapping of subfamily names to their prior counts subfams - list of subfamily names taken from the original alignments active_cells - map of column indices (from columns) into list of non-empty rows for the given column align_matrix - the alignment matrix from fill_align_matrix

Outputs:

confidence_matrix - hash implementation of sparse 2D matrix used in pre-DP calculations. Key is tuple[int, int] that maps row, col with the value held in that cell of matrix. Rows are subfamilies in the input alignment file, cols are nucleotide positions in the alignment. Each cell in matrix is the confidence score calculated from all the alignment scores in a column of the alignment matrix.

```
>>> align_mat = {
...     (0, 0): 10, (0, 1): 10, (0, 2): 10, (0, 3): 10, (0, 4): 10,
...     (1, 1): 42, (1, 2): 41, (1, 3): 41, (2, 1): 45, (2, 2): 43, (2, 3): 39,
... }
>>> active = {0: [0], 1: [0, 1, 2], 2: [0, 1, 2], 3: [0, 1, 2], 4: [0]}
>>> col_count = 5
>>> counts = {"skip": 0.4, "s1": .33, "s2": .33}
>>> subs = ["skip", "s1", "s2"]
>>> conf_mat = fill_confidence_matrix(col_count, counts, subs, active, align_mat)
>>> f"{conf_mat[1,1]:.4f}"
'0.1100'
>>> f"{conf_mat[2,1]:.4f}"
'0.8800'
>>> f"{conf_mat[1,2]:.4f}"
'0.1980'
>>> f"{conf_mat[2,2]:.4f}"
'0.7920'
>>> f"{conf_mat[1,3]:.4f}"
'0.7920'
>>> f"{conf_mat[2,3]:.4f}"
'0.1980'
```

polyA.fill_consensus_position_matrix module

polyA.fill_consensus_position_matrix.**fill_consensus_position_matrix**(*row_count: int*, *column_count: int*, *start_all: int*, *subfams: List[str]*, *chroms: List[str]*, *starts: List[int]*, *stops: List[int]*, *consensus_starts: List[int]*, *strands: List[str]*) → *ConsensusMatrixContainer*

Fills matrix that holds the consensus position for each subfam at that position in the alignment. Walks along the alignments one nucleotide at a time adding the consensus position to the matrix.

At same time, fills ActiveCells.

input:

column_count: number of columns in alignment matrix - will be same number of columns in consensus_matrix row_count: number of rows in matrices start_all: min start position on chromosome/target sequences for whole alignment subfams: actual subfamily/consensus sequences from alignment chroms: actual target/chromosome sequences from alignment starts: start positions for all competing alignments (on target) stops: stop positions for all competing alignments (on target) consensus_starts: where alignment starts in the subfam/consensus sequence strands: what strand each of the alignments are on - reverse strand will count down instead of up

output:

ConsensusMatrixContainer

```
>>> subs = ["", ".AA", "TT-"]
>>> chrs = ["", ".AA", "TTT"]
>>> strts = [0, 1, 0]
```

(continues on next page)

(continued from previous page)

```
>>> stps = [0, 2, 2]
>>> con_strts = [-1, 0, 10]
>>> strandss = ["", "+", "-"]
>>> active, con_mat = fill_consensus_position_matrix(3, 3, 0, subs, chrs, strts,␣
↪stps, con_strts, strandss)
>>> con_mat
{(1, 2): 0, (1, 3): 1, (2, 1): 10, (2, 2): 9, (2, 3): 9, (0, 0): 0, (0, 1): 0, (0,␣
↪2): 0}
>>> active
{2: [0, 1, 2], 3: [0, 1, 2], 1: [0, 2], 0: [0]}
```

polyA.fill_node_confidence module

polyA.fill_node_confidence.**fill_node_confidence**(*nodes: int*, *start_all: int*, *gap_inits: List[float]*, *gap_exts: List[float]*, *columns: List[int]*, *starts: List[int]*, *stops: List[int]*, *changes_position: List[int]*, *subfams: List[str]*, *subfam_seqs: List[str]*, *chrom_seqs: List[str]*, *subfam_countss: Dict[str, float]*, *sub_matrices: List[*SubMatrix*]*, *repeat_scores: Dict[int, float]*, *tr_count: int*) → Dict[Tuple[str, int], float]

for a particular annotated node, takes all competing alignments and calculates the confidence for that node

first fills matrix with node alignment scores, then reuses matrix for confidence scores Using changes_position indentifies boundaries for all nodes, and computes confidence values for each node.

input: all input needed for CalcScore() and ConfidenceCM() nodes: number of nodes changes_pos: node boundaries columns: all non empty columns in matrices

output: node_confidence: Hash implementation of sparse 2D matrix that holds confidence values for whole nodes. Used during stitching process. Tuple[str, int] is key that maps a subfamily and node number to a confidence score.

```
>>> non_cols = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> strts = [0, 0, 0]
>>> stps = [10, 10, 10]
>>> change_pos = [1, 3, 7, 10]
>>> names = ["skip", "n1", "n2"]
>>> s_seqs = ['', 'AAA-TTTT-T', 'TTTTTTTTTT']
>>> c_seqs = ['', 'TTTTTTTTTT', 'TTTTTTTTTT']
>>> counts = {"skip": .33, "n1": .33, "n2": .33}
>>> sub_mat = SubMatrix("", 0.1227)
>>> sub_mat.scores = {"AA": 10, "AT": -1, "TA": -1, "TT": 10, "..":0}
>>> sub_mat.background_freqs = None
>>> sub_mats = [sub_mat] * 3
>>> rep_scores = {}
>>> node_conf = fill_node_confidence(3, 0, [0, -25, -25], [0, -5, -5], non_cols,␣
↪strts, stps, change_pos, names, s_seqs, c_seqs, counts, sub_mats, rep_scores, 0)
>>> node_conf
{('n1', 0): 0.19999999999999998, ('n2', 0): 0.7999999999999999, ('n1', 1): 0.
↪058823529411764705, ('n2', 1): 0.9411764705882353, ('n1', 2): 0.1111111111111111,␣
↪('n2', 2): 0.8888888888888888}
```

(continues on next page)

(continued from previous page)

```
>>> s_seqs = ['', 'AAA-T--TT-', 'TTTTTTTTTT']
>>> c_seqs = ['', 'TTTTTTTTTT', 'TTTTTTTTTT']
>>> node_conf2 = fill_node_confidence(3, 0, [0, -25, -25], [0, -5, -5], non_cols,␣
↪strts, stps, change_pos, names, s_seqs, c_seqs, counts, sub_mats, rep_scores, 0)
>>> node_conf2
{('n1', 0): 0.19999999999999998, ('n2', 0): 0.7999999999999999, ('n1', 1): 0.
↪001949317738791423, ('n2', 1): 0.9980506822612085, ('n1', 2): 0.1111111111111111,␣
↪('n2', 2): 0.8888888888888888}
```

polyA.fill_path_graph module

polyA.fill_path_graph.**fill_path_graph**(*nodes: int*, *columns: List[int]*, *changes: List[str]*, *changes_position: List[int]*, *consensus_matrix_collapse: Dict[Tuple[int, int], int]*, *strand_matrix_collapse: Dict[Tuple[int, int], str]*, *node_confidence: Dict[Tuple[str, int], float]*, *subfams_collapse_index: Dict[str, int]*) → List[int]

finds alternative paths through the nodes - used for stitching to find nested elements and to stitch back together elements that have been inserted into.

Alternative edges only added when on same strand, and confidence of other node label is above a certain threshold and when the alignments are relatively contiguous in the consensus sequence.

input: nodes: number of nodes columns: list with all non empty columns in matrices changes: list of node boundaries changes_position: list of subfams identified for each node consensus_matrix_collapse: matrix holding alignment position in consensus/subfam seqs - used to identify whether nodes are spacially close enough for an alternative edge strand_matrix_collapse: matrix holding strand for alignments node_confidence: competing annoation confidence for nodes subfams_collapse_index: maps subfam names to their row indices in collapsed matrices

output: path_graph: Flat array reprepsentation of 2D matrix. Graph used during stitching. Maps nodes to all other nodes. 1 if nodes are connected, 0 if not

```
>>> non_cols = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> changess = ["s1", "s2", "s1"]
>>> changes_pos = [0, 3, 6, 10]
>>> con_mat_col = {(0, 0): 0, (0, 1): 1, (0, 2): 2, (0, 3): 3, (0, 6): 6, (0, 7): 7,
↪ (0, 8): 8, (0, 9): 9, (1, 3): 1, (1, 4): 2, (1, 5): 3, (1, 6): 4}
>>> strand_mat_col = {(0, 0): '+', (0, 1): '+', (0, 2): '+', (0, 3): '+', (0, 6): '+
↪', (0, 7): '+', (0, 8): '+', (0, 9): '+', (1, 3): '-', (1, 4): '-', (1, 5): '-',␣
↪(1, 6): '-'}
>>> node_conf = {('s1', 0): 0.9, ('s1', 1): 0.5, ('s1', 2): 0.9, ('s1', 0): 0.1, (
↪'s2', 1): 0.5, ('s2', 2): 0.1}
>>> sub_col_ind = {'s1': 0, 's2': 1}
>>> fill_path_graph(3, non_cols, changess, changes_pos, con_mat_col, strand_mat_col,
↪ node_conf, sub_col_ind)
[0, 1, 1, 0, 0, 1, 0, 0, 0]
```

polyA.fill_probability_matrix module

polyA.fill_probability_matrix.**fill_probability_matrix**(*same_prob_skip: float*, *same_prob: float*, *change_prob: float*, *change_prob_skip: float*, *columns: List[int]*, *collapsed_matrices:* CollapsedMatrices) → Tuple[List[float], Dict[Tuple[int, int], int], Dict[Tuple[int, int], int]]

Calculates the probability score matrix to find most probable path through the support matrix. Fills the origin matrix for easier backtrace.

The basic algorithm is described below.

1. Look at all i's in j-1
2. Multiply by confidence in current cell
3. If it comes from same i, multiply by the higher probability
4. Else multiply by the lower probability divided by (numseqs-1) so sum of all probs == 1
5. Return max

Note: All probabilities are in log space.

Parameters

- **same_prob_skip** – penalty given to staying in the skip state
- **same_prob** – penalty given to staying in the same row
- **change_prob** – penalty given for changing rows
- **change_prob_skip** – penalty given for changing rows in or out of skip state
- **columns** – list that holds all non empty columns in matrices CollapsedMatrices container

Returns

1. `col_list`: last column of prob matrix, needed to find max to know where to start the backtrace.
2. `origin_matrix`: Hash implementation of sparse 2D DP matrix. This is a collapsed matrix. Holds which cell in previous column the probability in the DP matrix came from. Used when doing backtrace through the DP matrix.
3. `same_subfam_change_matrix`: parallel to origin_matrix, if 1 - came from same subfam, but got a change probability. When doing backtrace, have to note this is same subfam name, but different annotation.

Return type
Tuple

polyA.fill_support_matrix module

polyA.fill_support_matrix.**fill_support_matrix**(*chunk_size: int*, *starts: List[int]*, *stops: List[int]*, *confidence_matrix: Dict[Tuple[int, int], float]*) → Dict[Tuple[int, int], float]

Fill the support_matrix using values in *confidence_matrix*. Average confidence values for surrounding *chunk_size* confidence values - normalized by dividing by number of segments.

The score for subfam x at position i is the sum of all confidences for subfam x for all segments that overlap position i - divided by the number of segments.

Inputs:

chunk_size - the width of the window to use when averaging confidence scores. This is assumed to be an odd number >= 3.

starts - a list of start columns for each row in the confidence matrix. These positions come should come directly from the alignments without any offsetting. The exception is the first value, which should contain the minimum of the remaining values, less one, in other words, *starts[0] == min(starts[1:]) - 1*. This is the implicit start position for the skip state, which occupies the first row and is the only row that has data in the first column of the confidence matrix.

stops - equivalent to the starts list, but contains the last column in each row that contains data. Again, the first value is special and should contain the maximum of the remaining values. In other words, *stops[0] == max(stops[1:]) + 1*.

confidence_matrix - confidence values computed with fill_confidence_matrix. The confidence matrix should have one contiguous run of values per row.

Outputs:

support_matrix - support values represent average confidence values over a window chunk_size wide at each position in the confidence matrix. This is the "support score".

```
>>> conf_mat = {
...     (0, 0): 1.0, (0, 1): 0.1, (0, 2): 0.2, (0, 3): 0.4, (0, 4): 0.5, (0, 5): 1.
↪0,
...     (1, 1): 0.9, (1, 2): 0.5, (1, 3): 0.5, (1, 4): 0.5,
...     (2, 2): 0.3, (2, 3): 0.1,
... }
>>> supp_mat = fill_support_matrix(
...     chunk_size=3,
...     starts=[0, 1, 2],
...     stops=[5, 4, 3],
...     confidence_matrix=conf_mat)
>>> len(supp_mat) == len(conf_mat)
True
>>> round(supp_mat[1, 1], 4)
0.7
>>> round(supp_mat[1, 2], 4)
0.6333
>>> round(supp_mat[1, 3], 4)
0.5
>>> round(supp_mat[1, 4], 4)
0.5
>>> (1, 0) in supp_mat
False
```

(continues on next page)

(continued from previous page)

```
>>> round(supp_mat[2, 2], 4)
0.2
>>> round(supp_mat[2, 3], 4)
0.2
>>> (2, 4) in supp_mat
False
>>>
>>> conf_mat = {
...     (0, 0): 1.0, (0, 1): 0.1, (0, 2): 0.05, (0, 3): 0.01, (0, 4): 0.01, (0, 5): 0.01, (0, 6): 0.9, (0, 7): 1.0,
...     (1, 1): 0.9, (1, 2): 0.75, (1, 3): 0.5, (1, 4): 0.39, (1, 5): 0.1, (1, 6): 0.1,
...     (2, 2): 0.2, (2, 3): 0.49, (2, 4): 0.6, (2, 5): 0.89,
... }
>>> supp_mat = fill_support_matrix(
...     chunk_size=3,
...     starts=[9, 10, 11],
...     stops=[16, 15, 14],
...     confidence_matrix=conf_mat)
>>> len(supp_mat) == len(conf_mat)
True
>>> supp_mat[1, 1]
0.825
>>> round(supp_mat[1, 2], 4)
0.7167
>>> round(supp_mat[1, 3], 4)
0.5467
>>> round(supp_mat[1, 4], 4)
0.33
>>> round(supp_mat[1, 5], 4)
0.1967
>>> round(supp_mat[1, 6], 4)
0.1
>>> (2, 1) in supp_mat
False
>>> round(supp_mat[2, 2], 4)
0.345
>>> round(supp_mat[2, 3], 4)
0.43
>>> round(supp_mat[2, 4], 4)
0.66
>>> round(supp_mat[2, 5], 4)
0.745
>>> (2, 6) in supp_mat
False
```

polyA.get_path module

polyA.get_path.**get_path**(*columns: List[int]*, *ids: List[str]*, *subfams_collapse: List[str]*, *last_column: List[float]*, *active_cells_collapse: Dict[int, List[int]]*, *origin_matrix: Dict[Tuple[int, int], int]*, *same_subfam_change_matrix: Dict[Tuple[int, int], int]*) → Tuple[List[int], List[str]]

back traces through origin matrix to get most probable path through the DP matrix. Finds where the path switches to a different row and populates Changes and ChangesPosition. Reverses Changes and ChangesPosition because it's a backtrace so they are initially backwards. Jumps over removed/empty columns when necessary.

assigns IDs to each col in matrices (corresponds to a nucleotide position in target/chrom sequence) - cols with same ID are part of same initial subfam

input: temp_id: current id number being used - makes it so new ids are unique columns: list of non empty columns in matrices ids: list of ids for each column in matrices changes_orig: original changes from first DP trace changes_pos_orig: original changes_position from first DP trace columns_orig: original list of non empty columns before any nodes are extracted subfams_collapse: subfamily names for the rows last_column: last column in probability matrix. Used to find where to start backtrace. active_cells_collapse: holds which rows haves values for each column origin_matrix: origin matrix same_subfam_change_matrix: parallel to origin_matrix, if 1 - came from the same subfam, but got a change transition probability

output: temp_id: updated current id number being used after function completes changes_position: which columns (positions in target/chrom seq) switch to different subfam changes: parallel array to changes_position - what subfam is being switches to updates input list ids

```
>>> non_cols = [0, 1, 2, 3]
>>> idss = ["", "", "", ""]
>>> subs = ["s1", "s2"]
>>> active_col = {0: [0, 1], 1: [0, 1], 2: [0, 1], 3: [0, 1]}
>>> last_col = [-100, -10]
>>> orig_mat = {(0, 0): 0, (1, 0): 1, (0, 1): 0, (1, 1): 0, (0, 2): 0, (1, 2): 0,␣
↪(0, 3): 0, (1, 3): 1}
>>> same_sub_mat = {}
>>> (changes_pos, changess) = get_path(non_cols, idss, subs, last_col, active_col,␣
↪orig_mat, same_sub_mat)
>>> changes_pos
[0, 2, 3]
>>> changess
['s1', 's2']
>>> idss[0] == idss[1]
True
>>> idss[2] == idss[3]
True
>>> idss[0] != idss[2]
True
```

polyA.lambda_provider module

class polyA.lambda_provider.**ConstantLambdaProvider**(*value: float*)

Bases: object

A test double that returns a fixed value regardless of the value passed when called.

```
>>> p: LambdaProvider = ConstantLambdaProvider(1.0)
>>> p({})
1.0
```

class polyA.lambda_provider.**EaselLambdaProvider**(*easel_path: str*)

Bases: object

A provider that executes Easel behind the scenes to recover a lambda value for the given matrix.

polyA.lambda_provider.**LambdaProvider**

A callback that accepts a substitution matrix name and returns the corresponding lambda value for that matrix.

alias of Callable[[Dict[str, int]], float]

exception polyA.lambda_provider.**LambdaProviderException**(*return_code: int*, *stdout: str*, *stderr: str*)

Bases: Exception

return_code: int

stderr: str

stdout: str

polyA.load_alignments module

class polyA.load_alignments.**Shard**(*start*, *stop*, *alignments*)

Bases: tuple

property alignments

Alias for field number 2

property start

Alias for field number 0

property stop

Alias for field number 1

polyA.load_alignments.**load_alignment_tool**(*file: TextIO*) → str

Return tool used to produce the alignments in the input file.

polyA.load_alignments.**load_alignments**(*file: TextIO*, *add_skip_state: bool = False*) → Iterable[*Alignment*]

Load a set of alignments in Stockholm format. See below for an example.

```
#=GF ID  MERX#DNA/TcMar-Tigger
#=GF TR  chr0:0000-0000
#=GF SC  1153
#=GF ST  +
#=GF TQ  -1
#=GF ST  127
```

(continues on next page)

(continued from previous page)

```
#=GF SP  601
#=GF CST 135
#=GF CSP 628
#=GF FL  128
#=GF MX  20p41g.matrix
#=GF GI -25
#=GF GE -5
```

polyA.load_alignments.**shard_overlapping_alignments**(*alignments: Iterable[*Alignment*]*, *shard_gap: int*, *add_skip_state: bool = True*) → Iterable[*Shard*]

Shard the given alignments into overlapping groups, separated by no more than *shard_gap* nucleotides. This allows for more efficient processing for alignments over large regions of sequence (such as an entire genome) where many regions will be empty. Precondition: alignments are sorted by their start position and all alignments have start position <= stop position.

```
>>> skip = get_skip_state()
>>> a0 = Alignment("", "a", 1, 100, 0, 1, 10, 0, 0, [], "", 0, "", 0, 0)
>>> a1 = Alignment("", "a", 1, 100, 0, 21, 30, 0, 0, [], "", 0, "", 0, 0)
>>> shards = list(shard_overlapping_alignments([a0, a1], 10))
>>> len(shards)
2
>>> shards[0].start
1
>>> shards[0].stop
15
>>> len(shards[0].alignments)
2
>>> shards[1].start
16
>>> shards[1].stop
100
>>> len(shards[1].alignments)
2
```

polyA.matrices module

class polyA.matrices.**CollapsedMatrices**(*row_num_update*, *consensus_matrix*, *strand_matrix*, *support_matrix*, *subfamilies*, *active_rows*, *subfamily_indices*, *subfam_alignments_matrix*)

Bases: `tuple`

property active_rows

Alias for field number 5

property consensus_matrix

Alias for field number 1

property row_num_update

all matrices used in DP are "collapsed" meaning if there are duplicate rows with the same subfamily they are collapsed into the same row. We choose which original row to use in the collapsed version by doing a mini DP to find the most probable.

updates number of rows in matrices

property strand_matrix

Alias for field number 2

property subfam_alignments_matrix

Alias for field number 7

property subfamilies

Alias for field number 4

property subfamily_indices

Alias for field number 6

property support_matrix

Alias for field number 3

class polyA.matrices.**ConsensusMatrixContainer**(*active_rows: Dict[int, List[int]]*, *matrix: Dict[Tuple[int, int], int]*)

Bases: tuple

A container for a consensus matrix along with its context.

property active_rows

Mapping from column index to a list of row indices in that column that are "active" (have data in them). This lets us avoid looping through unnecessary rows.

property matrix

Hash implementation of a sparse 2D matrix used along with DP matrices. The key is a 2-tuple of row and column indices (respectively) and the values are the alignment positions in the consensus subfamily sequence.

polyA.output module

class polyA.output.**Output**(*output_path: str*, *output_to_file: bool*)

Bases: object

A class to manage creating output file objects.

get_heatmap(*index: int*) → TextIO

get_results() → TextIO

get_soda(*index: int*) → Tuple[TextIO, TextIO]

polyA.pad_sequences module

polyA.pad_sequences.**pad_sequences**(*chunk_size: int*, *subfam_seqs: List[str]*, *chrom_seqs: List[str]*, *padding_char: str = '.'*) → None

Right-pad sequences with *(chunk_size - 1) / 2* copies of ".".

The first sequence is ignored since that is the skip state.

The *padding_char* parameter exists because the doctest doesn't "see" the periods, so the test will pass no matter how many are added to each sequence.

Inputs:

chunk_size: size (in nucleotides) of each segment subfam_seqs: actual subfamily / consensus sequences from alignment chrom_seqs: actual target / chromosome sequences from alignment padding_char: for testing, specifies the character to use for padding

Side effects:

Updates subfam_seqs and chrom_seqs with padded sequences

```
>>> s_seq = ['', 'a', 'aaa']
>>> c_seq = ['', 'a', 't-t']
>>> pad_sequences(31, s_seq, c_seq, padding_char="-")
>>> s_seq
['', 'a----------------', 'aaa----------------']
>>> c_seq
['', 'a----------------', 't-t----------------']
```

polyA.performance module

polyA.performance.**reset_timeit_file**()

polyA.performance.**reset_timeit_logger**()

polyA.performance.**set_timeit_file**(*file: TextIO*)

polyA.performance.**set_timeit_logger**(*logger: Logger*)

polyA.performance.**timeit**(*name: Optional[str] = None*)

A benchmarking helper intended for us as a decorator. To use, decorate a function with *@timeit()*, then set the *POLYA_PERFORMANCE* environment variable to any value when the program is run.

Output will be written to stderr unless a *file* is provided by calling *set_timeit_file*, in which case that will be used. If a logger is passed to *set_timeit_logger* then that will be used for output preferentially.

```
>>> import os, io
>>> stream = io.StringIO()
>>> os.environ[ENV_VAR_NAME] = 'true'
>>> set_timeit_file(stream)
>>> perf_sum = timeit()(sum)
>>> perf_sum([1, 2])
3
>>> stream.getvalue()[:31]
'[PERFORMANCE] time spent in sum'
```

polyA.print_helpers module

polyA.print_helpers.**find_consensus_lengths**(*alignments: List[*Alignment*]*) → Dict[str, int]

Consensus sequence lengths used by the SODA printer.

polyA.printers module

class polyA.printers.**Printer**(*output_file: Optional[TextIO] = None*, *print_id: bool = False*, *soda_viz_file: Optional[TextIO] = None*, *use_matrix_position: bool = False*, *use_sequence_position: bool = False*)

Bases: object

print_results_chrom(*start_all: int*, *chrom_start: int*, *changes: List[str]*, *tr_changes: Dict[int, str]*, *position_changes: List[int]*, *columns: List[int]*, *ids: List[str]*) → None

Print final results. Start and stop are in terms of the chromosome / target position.

print_results_header()

print_results_matrix(*changes: List[str]*, *tr_changes: Dict[int, str]*, *position_changes: List[int]*, *columns: List[int]*, *ids: List[str]*) → None

Print final results. Start and stop are in terms of matrix position.

print_results_sequence(*start_all: int*, *changes: List[str]*, *tr_changes: Dict[int, str]*, *position_changes: List[int]*, *columns: List[int]*, *ids: List[str]*) → None

Print final results. Start and stop are in terms of the target sequence.

print_results_simple(*start: int*, *stop: int*, *node_id: str*, *node_name: str*) → None

A version of the output printer that takes the exact values to output.

print_results_soda(*start_all: int*, *chrom: str*, *chrom_start: int*, *chrom_end: int*, *subfams: List[str]*, *changes_orig: List[str]*, *tr_consensus_changes: Dict[int, str]*, *changes_position_orig: List[int]*, *columns_orig: List[int]*, *consensus_lengths: Dict[str, int]*, *strand_matrix_collapse: Dict[Tuple[int, int], str]*, *consensus_matrix_collapse: Dict[Tuple[int, int], int]*, *subfams_collapse_index: Dict[str, int]*, *node_confidence_orig: Dict[Tuple[str, int], float]*, *ids: List[str]*, *subfam_alignments: List[str]*, *chrom_alignments: List[str]*, *consensus_starts: List[int]*, *consensus_stops: List[int]*, *chrom_starts: List[int]*, *chrom_stops: List[int]*, *alignments_matrix: Dict[Tuple[str, int], Tuple[int, int]]*, *matrix: Dict[Tuple[int, int], float]*, *subfams_collapse: List[str]*, *num_col: int*) → None

Produce a polya-soda html file with the results.

set_soda_files(*viz_file: Optional[TextIO]*)

property use_matrix_position: bool

property use_sequence_position: bool

property use_soda_output: bool

polyA.printers.**calc_relative_start_and_end**(*confidence_start: int*, *alignment_start: int*, *confidence_length: int*, *target_seq: str*)

polyA.printers.**complement**(*sequence: str*) → str

polyA.prior_counts module

polyA.prior_counts.**read_prior_counts**(*prior_counts_file: TextIO*, *use_trs: bool*) → Dict[str, float]

polyA.substitution_matrix module

exception polyA.substitution_matrix.**ParseError**

Bases: RuntimeError

class polyA.substitution_matrix.**SubMatrix**(*name: str*, *lamb: float*)

Bases: object

A single substitution matrix, including its name, lambda value, and assumed character background frequencies. The lambda value may be *None*, in which case it must be computed before the matrix can be used. The background frequencies may also be 'None' if complexity adjusted scoring will not be used.

A single matrix maps '<char1><char2>' (a pair of characters from the input alphabet) to the score from the input substitution matrix.

For example: 'AA' => 8

background_freqs: Optional[Dict[str, float]]

lamb: float

name: str

scores: Dict[str, int]

polyA.substitution_matrix.**load_substitution_matrices**(*file: TextIO*, *lambda_provider: Callable[[Dict[str, int]], float]*, *complexity_adjustment: bool*, *alphabet_chars: str = 'AGCTYRWSKMDVHBXN'*, *ambiguity_chars: str = '-.'*) → Dict[str, *SubMatrix*]

Reads a set of score matrices from a file and returns a *SubMatrixCollection* that contains the matrices and any lambda values associated with them.

Each matrix has a name, which is then specified for each alignment that is to be adjudicated.

```
>>> _lamb_provider = ConstantLambdaProvider(0.1227)
>>> matrices_file = "fixtures/ultra_test_files/ex13.fa.cm.matrix"
>>> with open(matrices_file) as _sub_matrices_file:
...     sub_matrices = load_substitution_matrices(_sub_matrices_file, _lamb_
↪provider, False)
>>> len(sub_matrices)
1
>>> "matrix1" in sub_matrices
True
>>> matrix = sub_matrices["matrix1"]
>>> matrix.lamb
0.1227
>>> matrix.name
'matrix1'
>>> matrix.scores['AA']
8
```

(continues on next page)

(continued from previous page)

```
>>> matrix.scores['AT']
-15
>>> matrix.scores['TT']
8
>>> matrix.scores['GA']
-2
>>> matrix.scores['GC']
-13
>>> matrix.scores['CC']
10
>>> matrix.background_freqs is None
True
>>> with open(matrices_file) as _sub_matrices_file:
...     sub_matrices = load_substitution_matrices(_sub_matrices_file, _lamb_
↪provider, True)
>>> matrix = sub_matrices["matrix1"]
>>> matrix.background_freqs is None
False
>>> matrix.background_freqs
{'A': 0.295, 'G': 0.205, 'C': 0.205, 'T': 0.295}
```

polyA.sum_repeat_scores module

polyA.sum_repeat_scores.**sub_repeat_scores**(*start: int*, *end: int*, *repeat_scores: Dict[int, float]*) → float

Sums tandem repeat scores from columns start to end.

input: start: column index to start sum end: column index to end sum repeat_scores: hash implementation of 1d array that maps column index to tandem repeat score

output: summ: sum of the tandem repeat scores

polyA.ultra_provider module

class polyA.ultra_provider.**ApplicationUltraProvider**(*sequence_path: str = ''*, *ultra_output_path: str = ''*, *ultra_path: str = 'ultra'*)

Bases: object

class polyA.ultra_provider.**TandemRepeat**(*consensus*, *start*, *length*, *stop*, *position_scores*)

Bases: tuple

property consensus

Alias for field number 0

static from_json(*json_map: Dict[str, Any]*)

property length

Alias for field number 2

property position_scores

Alias for field number 4

property start

Alias for field number 1

property stop

Alias for field number 3

class polyA.ultra_provider.**UltraOutput**(*tandem_repeats*)

Bases: tuple

static from_json(*json_map: Dict[str, Any]*)

property tandem_repeats

Alias for field number 0

property tr_count: int

exception polyA.ultra_provider.**UltraProviderException**(*return_code: int*, *stdout: str*, *stderr: str*)

Bases: Exception

return_code: int

stderr: str

stdout: str

Module contents

PolyA is a sequence annotation adjudicator.

CHAPTER

TWO

INDICES AND SEARCH

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

G

I

L

M

N

O

P

R

S

T

U