autostreamtree v1.1.0 2024-01-23

Contents

Module autostreamtree

Sub-modules

Module `autostreamtree.aggregators`

Functions

Function `adjusted_harmonic_mean`

```
def adjusted_harmonic_mean(
    stuff
)
```

Computes an adjusted harmonic mean that is corrected for non-positive values.

Args —–= `stuff` : `array-like` : The input array.

Returns —–= float : The adjusted harmonic mean.

Function `aggregate_dist`

```
def aggregate_dist(
    method,
    stuff
)
```

Aggregates the given array according to the specified method.

Args —–= `method` : str : The aggregation method to use (e.g. “HARM”, “ARITH”, “GEOM”, etc.).

`stuff` **:** `array-like` The input array to be aggregated.

Returns —–= float : The aggregated result.

Module `autostreamtree.cli`

Functions

Function `main`

```
def main()
```

Module `autostreamtree.cluster_pops`

Functions

Function `coords_to_dataframe`

```
def coords_to_dataframe(
    coords
)
```

Convert a dictionary of geographic coordinates to a Pandas DataFrame.

Args: - coords (dict): A dictionary containing the coordinates in the format {id:(long, lat)}.

Returns: - A Pandas DataFrame of coordinates with columns “long” and “lat”.

Function `coords_to_matrix`

```
def coords_to_matrix(
    coords
)
```

Convert a dictionary of geographic coordinates to a NumPy array.

Args: - coords (dict): A dictionary containing the coordinates in the format {id:(long, lat)}.

Returns: - A NumPy array of coordinates in the format [[long, lat], …].

Function `dbscan_cluster`

```
def dbscan_cluster(
    coords: Dict[str, Tuple[float, float]],
    epsilon: float,
    min_samples: int,
    out: Optional[str] = None
) -> sortedcontainers.sorteddict.SortedDict
```

Cluster geographic coordinates using DBSCAN algorithm.

Parameters: coords (Dict[str, Tuple[float, float]]): A dictionary of coordinate pairs with the key as the location identifier and values as tuples of (latitude, longitude). epsilon (float): Maximum distance between two samples for them to be considered as in the same neighborhood. min_samples (int): The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. out (Optional[str]): Output file path to save cluster labels.

Returns: SortedDict: A sorted dictionary of cluster labels with keys as cluster name and values as lists of location identifiers in that cluster.

Raises: ValueError: If any coordinate value is missing or not a float.

Examples:

```
>>> coords = {'A': (33.4484, -112.0740), 'B': (37.7749,
              -122.4194), 'C': (40.7128, -74.0060)}
>>> dbscan_cluster(coords, 300, 2)
SortedDict({'DB_-1': ['C'], 'DB_0': ['A'], 'DB_1': ['B']})
```

Function `flatten_popmap`

```
def flatten_popmap(
    popmap
)
```

Flattens the popmap dictionary from a form of key=pop; value=list(inds) to key=ind; value=pop.

Args ——= `popmap` : dict : Dictionary in key=pop; value=list(inds) format.

Returns ——= dict : Dictionary in key=ind; value=pop format.

Function `get_cluster_centroid`

```
def get_cluster_centroid(
    coords,
    popmap,
    out=None
)
```

Function `get_pop_coords_matrix`

```
def get_pop_coords_matrix(
    d,
    subset
)
```

Function `plot_clustered_points`

```
def plot_clustered_points(
    point_coords,
    popmap,
    out,
    centroids=None
)
```

Function `plot_histogram`

```
def plot_histogram(
    dat,
    out
)
```

Plots a histogram of snap distances and saves to a PDF file.

Args ——= `dat` : list : List of snap distances.

`out` **: str** Output file path without extension.

Returns ——= None

Module `autostreamtree.functions`

Functions

Function `block_print`

```
def block_print()
```

Disables standard output by redirecting it to a null device, effectively blocking any print statements.

Function `custom_warn_handler`

```
def custom_warn_handler(
    message,
    category,
    filename,
    lineno,
    file=None,
    line=None
)
```

Function `enable_print`

```
def enable_print()
```

Restores standard output to its original state, allowing print statements to be displayed again.

Function `extract_full_subgraph`

```
def extract_full_subgraph(
    subgraph,
    graph,
    nodelist,
    id_col,
    len_col,
    path
)
```

Extracts the full subgraph from the given nodes.

Args —–= `subgraph` : NetworkX Graph : The subgraph to be modified.

`graph` **: NetworkX Graph** The input graph.
`nodelist` **: list** The list of nodes.
`id_col` **: str** The column name for node ID.
`len_col` **: str** The column name for edge length.
`path` **: list** The path between nodes.

Function `extract_minimal_subgraph`

```
def extract_minimal_subgraph(
    subgraph,
    graph,
    nodelist,
    id_col,
    len_col,
    path
)
```

Extracts a simplified subgraph from paths, keeping only terminal and junction nodes.

Args —–= `subgraph` : NetworkX Graph : The subgraph to be modified.

`graph` **: NetworkX Graph** The input graph.
`nodelist` **: list** The list of nodes.
`id_col` **: str** The column name for edge ID.
`len_col` **: str** The column name for edge length.
`path` **: list** The path between nodes.

Function `find_pair`

```
def find_pair(
    lst,
    x,
    y
)
```

Check if two elements are consecutive in a list, irrespective of their order.

Args —–= `lst` : list : The list to search for the pair.

`x` **: Any** The first element of the pair.
`y` **: Any** The second element of the pair.

Returns —–= bool : True if the elements are consecutive in the list, False otherwise

Function `fit_least_squares_distances`

```
def fit_least_squares_distances(
    D,
```

```
        X,
        iterative,
        out,
        weight='CSE67'
    )
```

Computes least-squares branch lengths from a vector of genetic distances D and incidence matrix X. When iterative=True, negative distances are constrained to 0 and then recomputed.

Args —–= `D` : numpy.ndarray : Vector of genetic distances.

`X` **: numpy.ndarray** Incidence matrix.
`iterative` **: bool** Whether to use an iterative approach to constrain negative distances.
`out` **: str** Output file prefix.
`weight` Weight type. Defaults to CSE67.

Returns —–= numpy.ndarray : Least-squared optimized distances.

Function `generate_weights_matrix`

```
    def generate_weights_matrix(
        d,
        weight
    )
```

Generates a weights matrix for the least-squares method, where weights are on the diagonals.

Args —–= `d` : numpy.ndarray : Vector of genetic distances.

`weight` **: str** Weighting method to use, options: ‘CSE67’, ‘BEYER74’, ‘FM67’.

Returns —–= numpy.ndarray : Weights matrix.

Function `get_fitted_d`

```
    def get_fitted_d(
        points,
        genmat,
        inc,
        r
    )
```

Calculates predicted genetic distances based on fitted streamtree distances.

Args —–= `points` : dict : A dictionary of points with their latitude and longitude values.

`genmat` **: ndarray** A pairwise genetic distance matrix.
`inc` **: ndarray** An incidence matrix representing the presence or absence of streams for each point.
`r` **: ndarray** A fitted streamtree distance matrix.

Returns —–= pandas.DataFrame : A dataframe with columns for ‘from’, ‘to’, ‘observed_D’, ‘predicted_D’, and ‘abs_diff’.

Function `get_gendist_mats`

```
    def get_gendist_mats(
        params,
        point_coords,
        popmap,
        seqs
    )
```

Returns population genetic distance matrices.

Args —–= `params` : object : An object that contains parameters for the analysis.

`point_coords` **: list** A list of coordinates for the sampled points.
`popmap` **: dict** A dictionary that maps each sample to its corresponding population.
`seqs` **: list** A list of DNA sequences for the sampled points.

Returns —–= tuple : A tuple containing two matrices. The first is a pairwise distance matrix for all

samples. The second is a matrix of pairwise genetic distances between populations.

Raises —–= SystemExit : If distance metric is not possible without population data.

Function `get_loc_data`

```
def get_loc_data(
    seqs
)
```

Generator function that yields a dictionary of individual loci data for each locus in the input sequences.

Args —–= `seqs` : A dictionary containing sequences as values and individual

identifiers as keys.

Returns —–= A generator that yields a dictionary with individual identifiers as keys and a list containing the corresponding locus as the value.

Function `get_lower_tri`

```
def get_lower_tri(
    mat
)
```

Extracts the lower triangular elements from a square matrix.

Args —–= `mat` : numpy.ndarray : Input square matrix.

Returns —–= numpy.ndarray : 1D array containing the lower triangular elements of the input matrix.

Function `get_point_table`

```
def get_point_table(
    points
)
```

Returns a pandas DataFrame from a dictionary of points.

Args —–= `points` : dict : A dictionary of points with their latitude and longitude values.

Returns —–= pandas.DataFrame : A dataframe with columns for ‘sample’, ‘lat’, and ‘long’.

Function `get_stream_mats`

```
def get_stream_mats(
    points,
    graph,
    len_col
)
```

Computes pairwise stream distances and 0/1 incidence matrix for StreamTree calculations.

Args —–= `points` : dict : Dictionary of point indices and their corresponding node IDs in the graph.

`graph` **: networkx.Graph** NetworkX graph object representing the stream network.
`len_col` **: str** Attribute name for the length of the edges in the graph.

Returns —–= tuple : Pair of numpy.ndarray representing the pairwise stream distance matrix and incidence matrix.

Function `great_circle`

```
def great_circle(
    lon1,
    lat1,
    lon2,
    lat2,
    thresh=1e-07
)
```

Calculates the great circle distance between two points on a sphere, using their longitudes and latitudes.

Args ——= `lon1` : float : Longitude of the first point.

`lat1` **: float** Latitude of the first point.
`lon2` **: float** Longitude of the second point.
`lat2` **: float** Latitude of the second point.
`thresh` **: float, optional** Threshold for determining if the points are the same. Defaults to 0.0000001.

Returns ——= float : Great circle distance in kilometers.

Function `nCr`

```
def nCr(
    n,
    k
)
```

Calculate the number of combinations, n choose k.

Args ——= `n` : int : The number of elements.

`k` **: int** The number of elements to choose.

Returns ——= int : The number of possible combinations.

Function `output_fitted_d`

```
def output_fitted_d(
    pred,
    out
)
```

Function `parse_input_genmat`

```
def parse_input_genmat(
    params,
    inmat,
    point_coords,
    popmap,
    seqs=None
)
```

Parses an input genetic distance matrix and verifies if it matches the user input parameters. Aggregates individual distances if required by the user input.

Args ——= `params` : Input parameters provided by the user.

`inmat` **: pd.DataFrame** The input genetic distance matrix.
`point_coords` **: dict** Dictionary containing point coordinates.
`popmap` **: dict** Dictionary containing the population map.

Returns ——= tuple : A tuple containing the genetic distance matrix (gen) and

population genetic distance matrix (pop_gen).

Function `parse_subgraph_from_points`

```
def parse_subgraph_from_points(
    params,
    point_coords,
    pop_coords,
    G
)
```

Extracts a subgraph from a given graph based on input points.

Args —–= `params` : A custom object containing various input parameters.

`point_coords` A list of point coordinates to use for extracting the subgraph.
`pop_coords` A list of population coordinates to use for extracting the subgraph.
`G` A NetworkX Graph object representing the input graph.

Returns —–= A NetworkX Graph object representing the extracted subgraph.

Function `path_edge_attributes`

```
def path_edge_attributes(
    graph,
    path,
    attribute
)
```

Get the attribute values for edges in a given path.

Args —–= `graph` : NetworkX Graph : The graph containing the edges.

`path` **: list** The list of nodes forming the path.
`attribute` **: str** The edge attribute to get the values for.

Returns —–= list : A list of attribute values for the edges in the path.

Function `path_subgraph`

```
def path_subgraph(
    graph,
    nodes,
    method,
    id_col,
    len_col
)
```

Find and extract paths between points from a graph.

Args —–= `graph` : NetworkX Graph : The input graph.

`nodes` **: dict** A dictionary of nodes to extract paths between.
`method` **: callable** The method to build the subgraph.
`id_col` **: str** The column name for node ID.
`len_col` **: str** The column name for edge length.

Returns —–= NetworkX Graph : A subgraph containing the extracted paths.

Function `plot_gen_by_geo`

```
def plot_gen_by_geo(
    gen,
    sdist,
    out,
    log=False
```

```
)
```

Plots genetic distance against geographic distance to visualize isolation by distance.

Args ——= `gen` : numpy.ndarray : Genetic distance matrix.

`sdist` **: numpy.ndarray** Spatial distance matrix.
`out` **: str** Output file prefix for the generated plot.
`log` **: bool, optional** If True, the geographic distance axis will be log-transformed. Defaults to False.

Function `process_samples`

```
def process_samples(
    params,
    points,
    G
)
```

Processes input sample data by snapping points to a graph, calculating coordinates, and processing populations if required.

Args ——= `params` : Input parameters provided by the user.

`points` **: pd.DataFrame** DataFrame containing sample points.
`G` **: networkx.Graph** Graph object representing the road network.

Returns ——= tuple : A tuple containing point coordinates, population coordinates,

and the population map.

Function `prune_graph`

```
def prune_graph(
    G,
    edge_list,
    reachid_col
)
```

Prunes a graph to only retain edges whose ‘reachid_col’ matches values in ‘edge_list’.

Args ——= `G` : The input NetworkX Graph.

`edge_list` A list of values to filter edges.
`reachid_col` The edge attribute to be checked against ‘edge_list’.

Returns ——= A pruned NetworkX Graph.

Function `r2`

```
def r2(
    x,
    y
)
```

Returns the Pearson correlation coefficient squared between two arrays.

Args ——= `x` : array : An array of values.

`y` **: array** An array of values.

Returns ——= float : The squared Pearson correlation coefficient between the two arrays.

Function `read_network`

```
def read_network(
    network,
    shapefile
)
```

Reads a network from a saved file or builds a network from a shapefile.

Args ——= `network` : Path to the saved network file (pickle format). If provided, the function will read the network from this file.

`shapefile` Path to the shapefile to build the network from. This is used if the network parameter is not provided.

Returns ——= A NetworkX Graph object representing the network.

Function `read_popmap`

```
def read_popmap(
    popmap
)
```

Reads a population map file and returns a dictionary with individuals as keys and populations as values.

Args ——= `popmap` : str : Path to the population map file.

Returns ——= dict : A dictionary with individuals as keys and populations as values.

Function `read_vcf`

```
def read_vcf(
    vcf,
    concat='none',
    popmap=None
)
```

Reads a VCF file and returns a dictionary of sample genotypes.

Args ——= `vcf` : Path to the input VCF file.

`concat` Specifies the concatenation mode for genotypes. Options are “all”, “loc”, and “none”. “all”: Concatenate genotypes of all loci for each sample. “loc”: Concatenate genotypes within the same chromosome for each sample. “none”: Do not concatenate genotypes.

`popmap` Optional dictionary that maps populations to a list of samples. If provided, only samples in the popmap will be retained in the output dictionary.

Returns ——= A dictionary with sample names as keys and lists of genotypes as values

Function `replace_zeroes`

```
def replace_zeroes(
    data
)
```

Replaces zeroes in the input array with the smallest non-zero value.

Args ——= `data` : numpy.ndarray : Input array.

Returns ——= numpy.ndarray : Array with zeroes replaced by the smallest non-zero value.

Function `report_genmats`

```
def report_genmats(
    params,
    gen,
    pop_gen,
    point_coords,
    pop_coords
)
```

Prints genetic distance matrices and writes them to files.

Args ——= `params` : A custom object containing various input parameters.

`gen` A NumPy array representing the individual genetic distance matrix.
`pop_gen` A NumPy array representing the population genetic distance matrix.
`point_coords` A dictionary containing individual point coordinates.
`pop_coords` A dictionary containing population point coordinates.

Function `report_genmats_list`

```
def report_genmats_list(
    params,
    genlist,
    popgenlist,
    point_coords,
    pop_coords
)
```

Writes individual and population genetic distance matrices to files for each locus in genlist and popgenlist.

Args ——= `params` : A namespace object containing parameters, including the output directory.

`genlist` A list of individual genetic distance matrices for each locus.
`popgenlist` A list of population genetic distance matrices for each locus.
`point_coords` A dictionary containing individual point coordinates.
`pop_coords` A dictionary containing population point coordinates.

Function `snap_to_node`

```
def snap_to_node(
    graph,
    pos
)
```

Finds the closest node to the given [x, y] coordinates in the graph.

Args ——= `graph` : NetworkX Graph : The input graph.

`pos` **: tuple** A tuple of [x, y] coordinates.

Returns ——= tuple : The closest node to the input coordinates.

Function `test_ibd`

```
def test_ibd(
    gen,
    geo,
    out,
    perms,
    log=False
)
```

Function `vectorize_mat`

```
def vectorize_mat(
    mat
)
```

Converts a pairwise matrix to a 1D vector.

Args ——= `mat` : numpy.ndarray : Pairwise matrix.

Returns ——= numpy.ndarray : 1D vector of matrix elements.

Function `write_geodataframe`

```
def write_geodataframe(
    gdf,
    output_prefix,
    output_driver
)
```

Module `autostreamtree.genetic_distances`

Functions

Function `clean_inds`

```
def clean_inds(
    inds
)
```

Removes individuals with unknown or gap alleles.

Args ——= `inds` : list : A list of individuals.

Returns ——= list : A list of individuals without unknown or gap alleles.

Function `clean_list`

```
def clean_list(
    to_clean,
    bads
)
```

Removes bad items from a list.

Args ——= `l` : list : A list to clean.

`bads` **: list** A list of items to remove.

Returns ——= list : A cleaned list.

Function `get_alleles`

```
def get_alleles(
    s
)
```

Splits a string of alleles separated by “/” into a list.

Args ——= `s` : str : A string of alleles separated by “/”.

Returns ——= list : A list of alleles.

Function `get_average_het`

```
def get_average_het(
    s1,
    s2
)
```

Computes the mean expected heterozygosity (Ht) from two populations.

Args ——= `s1` : list : A list of alleles from population 1.

`s2` **: list** A list of alleles from population 2.

Returns ——= float : The Ht estimate per locus.

Function `get_genmat`

```
def get_genmat(
    dist,
    points,
    seqs,
    ploidy,
    het,
    loc_agg
)
```

Function `get_global_het`

```
def get_global_het(
    seqs
)
```

Computes global expected heterozygosity (Ht) from a set of sequences.

Args: seqs (list): A list of sequences.

Returns: float: The Ht estimate for the given sequences.

Function `get_het_from_phased`

```
def get_het_from_phased(
    allele,
    phasedList,
    count=False
)
```

Returns observed heterozygosity of an allele given a list of phased genotypes (e.g. allele1/allele2 for each individual). Assumes diploid.

Args: allele (str): The allele for which to compute heterozygosity. phasedList (list): A list of phased genotypes, where each element is a string of the form 'allele1/allele2'. count (bool): Whether to return the number of heterozygotes instead of the proportion.

Returns: float: The proportion of heterozygotes for the given allele, unless count is True, in which case the count of heterozygotes is returned.

Function `get_pop_genmat`

```
def get_pop_genmat(
    dist,
    indmat,
    popmap,
    dat,
```

```
    seqs,
    pop_agg='ARITH',
    loc_agg='ARITH',
    ploidy=2,
    global_het=False
)
```

Function `hamming_distance`

```
def hamming_distance(
    seq1,
    seq2
)
```

Calculate the Hamming distance between two sequences.

Args ——= `seq1` : str : First sequence.

`seq2` **: str** Second sequence.

Returns ——= float : Hamming distance.

Function `p_distance`

```
def p_distance(
    seq1,
    seq2
)
```

Calculate p-distance, accounting for IUPAC ambiguity codes and partial matches.

Args ——= `seq1` : str : First sequence.

`seq2` **: str** Second sequence.

Returns ——= float : p-distance.

Function `two_pop_chord_dist`

```
def two_pop_chord_dist(
    s1,
    s2
)
```

Function `two_pop_jost_d`

```
def two_pop_jost_d(
    seqs1,
    seqs2,
    global_het=False
)
```

Function `two_pop_weir_cockerham_fst`

```
def two_pop_weir_cockerham_fst(
    s1,
    s2
)
```

Computes Weir and Cockerham's THETAst Fst approximation for two populations

Args ——= `s1` : list : A list of phased genotypes for population 1.

`s2` **: list** A list of phased genotypes for population 2.

Returns —–= tuple : A tuple containing two floats. The first float is the numerator

of the THETAst estimator for the locus. The second float is the denominator of the THETAst estimator for the locus.

Raises —–= ValueError : If the inputs are not valid.

Function `uniq_alleles`

```
def uniq_alleles(
    s
)
```

Returns the unique alleles in a list of alleles.

Args —–= `s` : list : A list of alleles.

Returns —–= set : A set of unique alleles.

Module `autostreamtree.params`

Classes

Class `parseArgs`

```
class parseArgs
```

Methods

Method `display_help`

```
def display_help(
    self,
    message=None
)
```

Module `autostreamtree.report_refs`

Functions

Function `fetch_references`

```
def fetch_references(
    params
)
```

Build an output reference string for all methods a user chose in a run.

Parameters:

params : object An object containing all user input parameters.

Returns:

refs : str A string containing all relevant references to be cited.

Module `autostreamtree.sequence`

Functions

Function `decode`

```
def decode(
    gt: tuple,
    ref: str,
    alts: list,
    as_iupac: bool = False,
    as_tuple: bool = False,
    as_list: bool = False
) -> str
```

Decode a genotype from a VCF file and return the corresponding DNA sequence

Args —–= `gt` : tuple : The genotype as a tuple of integers representing the indices of the reference and alternate alleles.

`ref` **: str** The reference allele.
`alts` **: list** A list of alternate alleles.
`as_iupac` **: bool** Whether to return the consensus IUPAC symbol if the genotype is heterozygous. Default is False.
`as_tuple` **: bool** Whether to return the result as a tuple of two strings Default is False.
`as_list` **: bool** Whether to return the result as a list of two strings. Default is False.

Returns —–= ret (str): The decoded genotype as a string. By default, this is a string in the format “ref/alt”, but the format can be customized using the optional arguments.

Function `dna_consensus`

```
def dna_consensus(
    seq: str
) -> str
```

Make a consensus DNA sequence from alleles separated by a “/” character.

Args —–= `seq` : str : A string of DNA sequences separated by a “/” character.

Returns —–= consens (str): A consensus DNA sequence derived from the input sequences.

Function `get_iupac_caseless`

```
def get_iupac_caseless(
    char: str
) -> List[str]
```

Split a character to IUPAC codes, assuming diploidy.

Args —–= `char` : str : A character to be split.

Returns —–= codes (list): A list of IUPAC codes corresponding to the input character.

Function `get_nuc_freqs`

```
def get_nuc_freqs(
    seqs: dict,
    ploidy: int
) -> list
```

Compute the nucleotide frequencies of a set of DNA sequences.

Args —–= `seqs` : dict : A dictionary of DNA sequences. The keys are sample names and the values are strings of nucleotides.

`ploidy` **: int** The ploidy of the sequences. If ploidy=1, ambiguities will be skipped. If ploidy=2, ambiguities will be resolved.

Returns ——= freqs (list): A list of dictionaries, where each dictionary contains the nucleotide frequencies for a single position in the sequences.

Raises ——= ValueError : If any sequence is empty or contains only invalid characters, or if sequences are not all the same length.

Function `list_to_sort_unique_string`

```
def list_to_sort_unique_string(
    input: list
) -> str
```

Convert a list of characters to a sorted, unique string.

Args ——= `l` : list : A list of characters.

Returns ——= s (str): A string containing the characters from the input list, sorted and with duplicates removed.

Function `phase_snp`

```
def phase_snp(
    snp: str
) -> str
```

Phase a single nucleotide polymorphism (SNP) by splitting its IUPAC code into two alleles.

Args ——= `snp` : str : The IUPAC code for the SNP.

Returns ——= phase (str): The SNP alleles separated by a "/" character.

Function `reverse_iupac_case`

```
def reverse_iupac_case(
    char: str
) -> str
```

Translate a string of DNA bases to an IUPAC ambiguity code, retaining case.

Args ——= `char` : str : A string of DNA bases.

Returns ——= iupac (str): An IUPAC ambiguity code.

Generated by *pdoc* 0.10.0 (https://pdoc3.github.io).