/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/AverageIdendity_RelativeToShockIdentities.py

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/AveragingIdentitySignals.py

---

This script is performing analysis on some kind of cell data that is stored in CSV files. The main functionality can be broken down into a few steps:

Locating relevant files: The script first defines two methods, find_paths and find_paths_conditional_endswith, which are used to traverse a given root directory and find all files that meet certain conditions based on the filename. It searches through all subdirectories recursively and collects the paths of these files. The first method finds files whose names contain a specified middle portion and end with a specified string. The second method looks for files whose names either exactly match a given string (cond_lookfor) or, failing that, another given string (og_lookfor).

Cell class definition: The script defines a class named Cell which represents individual cell data. Each cell has various properties (such as time bounds, a reference pair, etc.) and methods that perform analyses on the cell data. The wilcoxon_analysis method, for example, performs the Mann-Whitney U test on two subwindows of the cell's data. The custom_standardize method standardizes the cell data by computing z-scores.

Main function: This script defines a main function where the bulk of the analysis occurs. It's working on a series of "sessions", each of which corresponds to a directory containing CSV files of cell data. It identifies the cells with '+' (positive), '-' (negative), and 'Neutral' status based on the wilcoxon_analysis method. This classification is used later to sort cells into three categories.

Then, for each CSV file found in the session's directory, the script does the following:

Read the data into a pandas DataFrame.
Initialize a new Cell object for each column in the DataFrame (each column represents a cell).
Group the cells into three groups: '+', '-', and 'Neutral', based on their classifications determined earlier.
For each group, it calculates the average of the corresponding dff_trace data, as well as the standard error of the mean (SEM).
It then plots the mean and SEM for each group on a graph, with the time on the x-axis and the Z-score on the y-axis. The graph is then saved to a PNG file.
It also writes the mean and SEM data to a new CSV file.
Execution: Lastly, the script checks if it is being run directly (as opposed to being imported as a module), and if so, calls the main function to start the analysis.

Please note, this script seems to be working with some specific biological or experimental data, possibly related to neuroscience given the naming conventions and use of terms like 'cell', 'hertz', 'z-score', etc. You may need to have a background in the specific field to fully understand the purpose and implications of the analysis. The script also seems to make some assumptions about the structure of the data and the filesystem, which could affect its robustness if these assumptions are not met.

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/avg_traces.py

---

This Python script is intended to perform two tasks, each defined in a separate `main1` and `main2` function, related to processing and averaging data from data frames stored in CSV files.

1. The `main1` function:
   - It first defines a ROOT directory where it expects to find a number of CSV files whose names end with "sorted_traces_z_fullwindow_id_auc_bonf0.05.csv".
   - The function `find_paths_endswith` is used to find all such CSV files under the ROOT directory.
   - For each of the found CSV files, it reads the file into a pandas DataFrame.
   - It then averages across three columns: `+_mean`, `-_mean`, and `Neutral_mean`, for each row, which might represent different measurements taken over time.
   - The average is then written to a new column of a new DataFrame and this DataFrame is saved in the same directory as the original file but with a name "avg_dff_trace.csv".

2. The `main2` function:
   - It starts by specifying a single CSV file path, which is to be opened and read into a pandas DataFrame.
   - For each column in the DataFrame (presumably representing a different cell's measurements over time), it appends the data to a list of traces.
   - It then averages and calculates the standard error of the mean (SEM) for each row across all columns, effectively giving you an average trace with uncertainty estimation.
   - The results are stored in a new DataFrame with columns "Avg dff trace" and "SEM", and this DataFrame is saved in the same directory as the original file but with a new name "avg_dff_trace_w_sem.csv".

The script is currently set to run the `main2` function when it is run as a script. This suggests that the user has the option to comment/uncomment the function call under `if __name__ == "__main__"` to switch between the two different tasks.

Note: This script assumes a very specific file and directory structure and specific names of the columns in the CSV files. If these assumptions are not met, the script may not work as expected.

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/BetweenCellAlignment.py

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/BetweenCellAlignment_AUCid.py

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/BetweenCellAlignmentShock.py

---

This script, written in Python, seems to be for processing and organizing data related to an experiment, possibly related to neuroscience given the use of terms like "cell" and "trace". It does this by locating files containing cell data, reading the data, and reorganizing it based on certain criteria.

Here's a brief overview of what each function does:

1. `find_avg_dff_of_cell_for_event(session_path, startswith)`: This function locates all CSV files that start with a certain string within a given directory and all of its subdirectories, and returns a list of their file paths.

2. `create_concat_csv(lst_of_all_avg_cell_csv_paths, root_path)`: This function reads the CSV files located by `find_avg_dff_of_cell_for_event()`, parses the cell data, and organizes it into a new dictionary structure based on the event and combo names. The restructured data is then written to new CSV files in the directory specified by `root_path`.

3. `main()`: This is the main function that orchestrates the whole process. It specifies a list of directories to process and a list of session types and events to look for. It then walks through the file system starting from each specified directory, looking for directories that match the specified session types. For each matching directory, it locates the CSV files, processes the data, and writes the restructured data to new CSV files.

It's important to note that this script seems highly tailored to a specific file and directory structure, and may not work properly if the files and directories are organized differently.

/media/rory/Padlock_DT/Rodrigo/Behavioral_Calcium_DLC_Analysis/BetweenCellsAnalysis.py

---

1. **find_paths**: This function accepts three arguments - the root directory path, a middle string, and an ending string. It uses `glob.glob` to find all files that match the pattern from the root directory to any subdirectories. The files must include the middle string and end with the ending string. It returns a list of paths.

2. **find_paths_endswith**: Similar to `find_paths`, but this function only needs the root directory and the ending string. It returns a list of all file paths that end with the specified string.

3. **find_paths_startswith**: This function accepts the root directory and a starting string, returning a list of all file paths that start with the specified string.

4. **find_paths_conditional_endswith**: This function checks all files in the specified directory and subdirectories. If the 'cond_lookfor' string is in the filename, it's added to the list; if not, the 'og_lookfor' string is appended to the root directory and added to the list.

5. **indv_events_spaghetti_plot**: This function accepts a list of individual event traces for a cell and plots them. The plot is saved as a .png file.

6. **sort_cells**: This function sorts the cells based on their z_score attribute.

7. **heatmap**: This function accepts a DataFrame, a file_path, an output_path, and additional optional arguments, and creates a heatmap of the data. The heatmap is saved as a .png file.

8. **spaghetti_plot**: This function accepts a DataFrame, file_path, and output_path, and creates a "spaghetti plot" of the data. The plot is saved as a .png file.

9. **custom_standardize**: This function accepts a DataFrame and several additional arguments. It performs a custom standardization (z-score) calculation on the data in the DataFrame.

10. **custom_standardize_limit**: Similar to custom_standardize, but it stops z-scoring at a certain limit.

11. **custom_standardize_limit_fixed**: Similar to custom_standardize_limit, but the limit is based on the index, not the z-score.

12. **zscore**: This function performs the calculation for a z-score given an observed value, a mean (mu), and a standard deviation (sigma).

13. **convert_secs_to_idx**: This function accepts a minimum and maximum time, a reference pair (0 seconds to a certain index), and a frequency (hertz). It returns the start and end indices.

14. **convert_secs_to_idx_single_timepoint**: This function converts a single time point into an index.

15. **gaussian_smooth**: This function applies a Gaussian filter to the DataFrame.

16. **change_cell_names**: This function replaces the "BLA-Insc-" part of each column name in the DataFrame.

17. **scatter_plot**: This function creates a scatter plot of the DataFrame. The plot is saved as a .png file.

18. **subdf_of_df**: This function creates a sub DataFrame for a specific time window.

19. **create_subwindow_for_col**: This function creates a subwindow for a specific column.

20. **insert_time_index_to_df**: This function inserts a time index into a DataFrame.

main function:

- performs several operations on data, such as loading a CSV file, transforming and standardizing the data, sorting the cells, and then creating visualizations.

Here's an outline of the steps this script is performing:

1. It sets a root path and names for the original and conditional CSV files to look for.

2. It uses a function `find_paths_conditional_endswith` to find all paths that end with either of the file names specified above.

3. For each path found:

   - It attempts to load a DataFrame from the CSV file at the path.

   - It transforms the DataFrame using functions `change_cell_names`, `custom_standardize`, `gaussian_smooth`, and `sort_cells`.

   - It then saves the transformed DataFrame to a new CSV file, replacing the original filename in the path with a new one that includes "baseline-10_0_gauss1.5".

   - It then tries to insert a time index into the DataFrame using the function `insert_time_index_to_df`.

   - If this results in a ValueError (presumably because there are fewer than 200 data points), it repeats the index insertion with a different range.

4. Finally, it generates two types of visualizations (a heatmap and a "spaghetti plot") of the sorted DataFrame and saves these as PNG files.

Behavioral_Calcium_DLC_Analysis/BetweenMiceAlignment_1.py

This is a Python script that appears to be used for processing and analyzing data from a neuroscience experiment. More specifically, the data likely pertains to calcium imaging recordings of neuronal activity across different mice and experimental sessions. The script organizes and concatenates data across similar experimental sessions and across multiple mice.

1. `find_csv_files` function: This function is a helper to find CSV files in a nested folder structure. Given a root directory (session_path) and a filename starting substring (startswith), it recursively searches the directory structure and returns a list of files that match the given criteria.

2. `concat_all_cells_across_similar_sessions` function: This function does the heavy lifting of the script. It concatenates cell data from similar experimental sessions. The logic of the function can be summarized as follows:

- Iterate over each CSV file provided in `lst_of_all_avg_concat_cells_path` (list of file paths to CSVs). For each file:
    - Parse details from the file path, such as `mouse_name`, `session_type`, `combo` (combination), and `subcombo` (subcombination). These details will be used as keys to organize data in a nested dictionary (`between_mice_d`).
    - Read the CSV file into a pandas DataFrame.
    - Iterate over each column (cell) in the DataFrame, and for each cell:
        - Rename the cell to include the mouse name, ensuring that cell names are unique across mice.
        - Add the cell's data to the nested dictionary (`between_mice_d`), using the keys derived from the file path.
- After all CSV files have been processed, iterate over the nested dictionary (`between_mice_d`), and for each unique combination of `session_type`, `combo`, and `subcombo`:
    - Create a new DataFrame from the cells data.
    - If the lengths of cell data arrays are not equal (jagged arrays), it creates the DataFrame by filling missing values with NaN.
    - Write the DataFrame to a new CSV file in a corresponding directory.

3. `main` function: This is where the script execution begins. It sets the `ROOT_PATH`, which is the base directory where data is located. It also specifies a list of experimental session types to process. For each session type, it uses `find_paths` to find all CSV files that contain cell data. It then calls `concat_all_cells_across_similar_sessions` to process the cell data, concatenating data from similar sessions across multiple mice.

Behavioral_Calcium_DLC_Analysis/BetweenMiceAlignment_2.py

Its main purpose is to process CSV files in a directory, truncate rows past a certain length, and handle "jagged" CSVs by dropping the last row.

Here's a detailed explanation:

1. `find_csv_files`: This function, like the one in your previous script, is a helper to find CSV files in a nested folder structure. It accepts a root directory and a filename starting substring, and it returns a list of file paths matching this criteria.

2. `truncate_csvs_in_root`: This function is responsible for managing the truncation process. It finds the relevant CSV files using the `find_csv_files` function, and for each CSV file, it creates a `Table` object and checks if the CSV file needs to be truncated.

3. `TableDatabase`: This class serves as a database for tracking the number of "jagged" CSV files, i.e., CSV files where columns have different lengths. The class has a class variable `_number_of_jagged_dfs` which tracks the total number of jagged dataframes.

4. `Table`: This class extends `TableDatabase` and represents a single CSV file. It has several methods for manipulating and checking the CSV data:
   - `check_if_df_len_equals_thres`: Checks if all columns in the DataFrame are of equal length, i.e., checks if the DataFrame is "jagged".
   - `drop_last_row_df`: Drops the last row of the DataFrame.
   - `include_table`: Checks if the DataFrame is "jagged" and performs various operations based on the `drop_row` parameter. If `drop_row` is `True`, it drops the last row if the DataFrame is "jagged", updates the count of jagged DataFrames in `TableDatabase`, and saves the DataFrame. If `drop_row` is `False`, it only updates the count of jagged DataFrames without making changes to the DataFrame.
   - `save_table`: Saves the DataFrame to a new CSV file, appending "_truncated" to the filename.
   - `truncate_past_len_threshold`: Checks if the DataFrame has more rows than the threshold. If so, it truncates the DataFrame to the threshold length and saves it.

5. `main`: This is where the script execution begins. It sets the `ROOT_PATH`, which is the base directory where data is located. It then calls `truncate_csvs_in_root` to process CSV files, truncating them if necessary and handling "jagged" CSVs by dropping the last row. The `len_threshold` parameter is set to 200, meaning all CSV files will be truncated to a maximum length of 200 rows.

In summary, this script processes CSV files in a directory, truncating them to a specified length and handling "jagged" CSV files by dropping the last row. It also keeps track of the number of "jagged" CSV files in the directory.

Behavioral_Calcium_DLC_Analysis/Cell.py

This script defines a `Cell` class. The purpose of the `Cell` class, as defined by the comment, is to create an object that represents a cell in order to allow for comparisons between cells based on their attributes.

The `Cell` class takes the following attributes during initialization:

1. `cell_name`: The name of the cell.
2. `dff_traces`: A list containing the fluorescence traces of the cell over time.
3. `unknown_time_min` and `unknown_time_max`: These attributes define a time window for which the cell's fluorescence traces are of interest.
4. `reference_pair`: A dictionary that pairs a reference time with an index.
5. `hertz`: The rate at which the cell's fluorescence was sampled.

The `Cell` class has four methods:

1. `average_zscore`: This method calculates and returns the average z-score of the cell's fluorescence trace within the specified time window. The z-score is a measure of how many standard deviations an element is from the mean. The method first checks if the distribution of fluorescence traces is normal using the `is_normal_distribution` method. If the distribution is normal, it calculates the z-score using the mean (`mu`) and standard deviation (`sigma`) of a standard normal distribution (mean of 0 and standard deviation of 1). If the distribution is not normal, the method does nothing and returns `None`.

2. `is_normal_distribution`: This method checks if the distribution of the cell's fluorescence traces within the specified time window is normal. According to the method, if the number of samples is greater than or equal to 30, the distribution is assumed to be normal. If not, it prints a message that the cell is not in a normal distribution and returns `True` regardless.

3. `make_arr_of_focus`: This method extracts the cell's fluorescence traces within the specified time window and returns them as a list. It first calculates the starting and ending indices of the window based on the minimum and maximum times of interest and the sampling rate (`hertz`). It then uses these indices to slice the `dff_traces` list and return the slice.

In the initialization method, the `arr_of_focus` and `z_score` attributes are calculated using the `make_arr_of_focus` and `average_zscore` methods, respectively. The `arr_of_focus` attribute is the list of fluorescence traces within the time window of interest, and the `z_score` attribute is the average z-score of these traces.

Behavioral_Calcium_DLC_Analysis/DataCleaning.py

The script has a couple of utility functions that deal with file operations such as deleting directories and files.

Here's what each function does:

1. `dir_del_recursively(foldername, root)`: This function takes in a `foldername` and a `root` directory, then walks through the directory tree rooted at `root` and deletes any directory named `foldername`. The function uses `os.walk()` to iterate over all subdirectories and `shutil.rmtree()` to remove the directories. It keeps a count of the number of directories deleted and prints that out at the end.

2. `delete_recursively(root_path, name_endswith_list)`: This function takes a root directory and a list of file name endings, then searches the root directory and all its subdirectories for any files ending with one of the specified endings. If it finds any, it deletes them using `os.remove()`. It prints out each file it deletes, or an error message if it fails to delete a file.

3. `purge(dir, pattern)`: This function takes a directory and a regex pattern. It looks for any files in the given directory (not recursively) whose names match the regex pattern, and prints out their paths. The line of code to actually delete these files (`os.remove()`) is commented out.

In the end, it uses the `delete_recursively(ROOT, del_list)` function with a root path and a list of file name endings to delete. The specified file name endings are "all_concat_cells_sorted_hm_baseline-10_-1_gauss1.5.png" and "all_concat_cells_sorted_spaghetti_baseline-10_-1_gauss1.5.png". It will recursively search the root directory for these files and delete them.

Behavioral_Calcium_DLC_Analysis/Identity_ShockRelative_SingleMouse.py

---

1. Defining a set of mice and the associated sessions for the experiment.

2. For each mouse and session, it identifies the appropriate CSV file (the identity determinator) that contains data for all the cells.

3. It defines the time parameters for the baseline and test periods and some other parameters such as the sampling rate (`hertz`) and the significance level (`alpha`).

4. Then, for each cell, it creates a `Cell` object which is responsible for performing the Wilcoxon test and classifying the cell as "+", "-", or "Neutral". These identities are stored in a dictionary (`d_cells`).

5. It then identifies all other CSV files in the root directory, and for each of these files, it again creates a `Cell` object for each cell.

6. The cells' dF/F traces are added to a dictionary (`d`) under the appropriate identity as determined earlier.

7. The function then averages these dF/F traces for each identity, and calculates the standard error of the mean.

8. Finally, it plots these average traces (with error bars) and saves the plots as PNG images. It also saves the averaged traces and their standard errors to new CSV files.

Behavioral_Calcium_DLC_Analysis/OneEvent_Pipeline.py

This script standardizes, smooths, and averages neuronal activity data, producing a summary visualization and processed data files for further analysis.

1. It first reads a CSV file where each row corresponds to a different event (likely a neuronal spike), and each column (after the first) is a different time point.

2. It transposes this dataframe so that each row is a different time point and each column is a different event. The first row (which contains the "Event #" label) is removed as it's not relevant for further analysis.

3. Then, it applies a z-scoring operation on each event (column), but only up to a certain time point (`limit_idx`). The z-score is calculated based on the mean and standard deviation of a defined baseline period (`baseline_min` to `baseline_max`). For time points beyond `limit_idx`, the original value is retained.

4. The dataframe is then transposed back to its original format.

5. A Gaussian smoothing operation is applied across each row (i.e., across time for each event), with a standard deviation of 1.5.

6. The function `avg_cell_eventrace` is then called. This function computes the average dF/F value for each time point (across all events) and plots this average trace. It also saves this average trace to a new CSV file.

7. Finally, the original "Event #" column is reinserted at the start of the dataframe and the processed data is saved to a new CSV file.

Overall, this script is used to preprocess cell imaging data by applying z-scoring, Gaussian smoothing, and averaging operations. This processed data can then be used for further analysis or visualization.

Behavioral_Calcium_DLC_Analysis/SingleCellAlignment.py

This script further processes and analyzes neuronal activity data from calcium imaging, specifically focusing on single-cell activity. It has several functionalities, summarized as follows:

1. **Custom Standardization**: The script reads in a dataframe from a CSV file, transposes the dataframe, then applies z-scoring to a specified baseline period. The standardized values are then applied to all columns in the dataframe up to a specific limit. This standardization method aims to normalize data based on an initial period of observation.

2. **Gaussian Smoothing**: The data is then smoothed using a Gaussian filter to reduce noise and highlight the underlying trends.

3. **Averaging and Plotting**: The script computes the average activity trace across all events for each time point, and generates a plot of this trace.

4. **File Output**: The processed data (including the z-scores and the average trace), as well as the plot, are saved for further use.

5. **Path finding**: Functions `find_paths` and `find_paths_2mids` are used for traversing through directories and finding relevant files.

6. **Main Function**: The `main` function iterates over different mice and sessions, applying the above processes to each "plot_ready.csv" file found in the specified directory. It utilizes a try-except block to handle potential errors and keep the program running.

7. **onecell Function**: This function allows the user to apply the same processes to specific CSV files, useful for testing or handling special cases.

Please note that the actual execution will depend on whether you run `main()` or `onecell()`, as they are the two entry points of the script.

Behavioral_Calcium_DLC_Analysis/SingleCellAlignment_AUC.py
Behavioral_Calcium_DLC_Analysis/SingleCellAlignment_AUC_nobonf.py
Behavioral_Calcium_DLC_Analysis/SingleCellAlignment_noAUC_bonf.py

1. **Defining the Area Under the Curve (AUC):** AUC is a method of quantifying the entire two-dimensional area underneath an entire curve (or any section of the curve). In the script, `subwindow_auc` function calculates the area under the curve for a subsection of a DataFrame column.

2. **Finding files:** The `find_paths` function is used to find files in a directory structure that matches a specific pattern.

3. **Performing Wilcoxon test:** The `wilcoxon_analysis` function carries out a statistical analysis (Wilcoxon-Mann-Whitney U Test) comparing the AUCs of two different time windows (pre-choice and post-choice) for a cell. It checks if the post-choice AUC is statistically greater, less, or neutral compared to pre-choice AUC. The Bonferroni correction is applied to the p-values to account for multiple testing.

4. **Main function:** The script uses a root directory path and some other parameters to locate all csv files that need to be processed. For each found file, the script reads it into a DataFrame, calculates the AUC for both pre-choice and post-choice windows and saves this data into a new CSV file. It also performs the Wilcoxon analysis on these AUCs and saves the result to another CSV file. Note that it only considers specific events, "Block_Reward Size_Shock Ocurred_Choice Time (s)".

This script is more specific and performs a more complex analysis. It focuses on calculating and comparing AUCs of specific time windows of each cell's response, and it uses a different statistical test (Mann-Whitney U test as opposed to t-test). The output is not a plot but two types of CSV files containing the AUCs and the result of the statistical test.

Behavioral_Calcium_DLC_Analysis/SingleCellAlignment_noAUC_nobonf.py

---

Here are some of the differences between these two scripts:

1. Calculation of Area Under the Curve (AUC):

In the first script, the function `subwindow_auc` is used to compute the AUC in pre-defined windows for every column of the dataframe (presumably, each column corresponds to a trial). The resultant AUC values for prechoice and postchoice segments are then saved in a new dataframe and written to a CSV file.

In the second script, the `subwindow_auc` function is not used. Instead, the script slices the data into prechoice and postchoice segments directly in the main function and does not compute AUC values.

2. Additional Functions:

The second script has additional functions, `avg_cell_eventrace` and `export_avg_cell_eventraces`, which compute the average trace of the cell event across all trials and can save the resulting average trace plot or data. These functions are not present in the first script.

3. Wilcoxon Analysis:

The function `wilcoxon_analysis` is used in both scripts to compare the prechoice and postchoice segments. In the first script, the comparison is made using the AUC values, while in this script, it directly compares the prechoice and postchoice segments. Also, in the first script, the p-value from the Wilcoxon analysis is adjusted using Bonferroni correction for multiple comparisons by dividing the alpha level by the total number of cells. In this script, there's no adjustment for multiple comparisons.

4. Cell and Session Selection:

The first script has an option to choose a subset of mice (cells) from a larger group, while this script analyzes all available cells. Also, the sessions considered are different in each script.

5. Error Handling:

The first script includes a try-except block to handle any FileNotFoundError that might occur when trying to open the CSV files. This is not included in this script.

6. File Paths and Naming:

The file paths and the specific files each script is looking for are different, which means they are likely performing analyses on different sets of data or different stages in a pipeline.

Overall, the main logical difference is that the above script computes the AUC for prechoice and postchoice segments and then performs a comparison of these values using a Wilcoxon analysis with Bonferroni correction. In contrast, this script computes the average trace of a cell event, and then performs a Wilcoxon analysis on prechoice and postchoice segments without any Bonferroni correction.

Behavioral_Calcium_DLC_Analysis/avg_traces.py

`main1()`: This function first calls the `find_paths_endswith()` function to get a list of file paths that end with the string "sorted_traces_z_fullwindow_id_auc_bonf0.05.csv". For each file in this list, it reads the file into a pandas DataFrame and then calculates the average of three columns ("+_mean", "-_mean", and "Neutral_mean") for each row in the DataFrame. The average is saved into a new DataFrame and then written to a new .csv file in the same directory as the original file.

`main2()`: This function reads in a specific csv file into a pandas DataFrame, then for each column in the DataFrame, it calculates the average and standard error of the mean (SEM) of the column values. It then writes these averages and SEMs into a new DataFrame and saves it to a new .csv file, replacing "all_concat_cells_z_fullwindow.csv" in the original file name with "avg_dff_trace_w_sem.csv".

Behavioral_Calcium_DLC_Analysis/get_sortedtraces_from_aucid.py

1. `find_paths(root_path: Path, endswith: str) -> List[str]`: This function finds all files in a directory and its subdirectories (specified by 'root_path') that end with a certain string (specified by 'endswith'). It uses the glob function with the recursive option enabled, which means it will search all subdirectories of the specified directory.

2. `find_paths_conditional_endswith(root_path, og_lookfor: str, cond_lookfor: str) -> list`: This function walks through the directory tree starting from 'root_path'. For each directory, it checks if there is a file that matches 'cond_lookfor'. If such a file is found, its path is added to the result list. If not, the path of a file matching 'og_lookfor' is added to the result list instead. In the end, it returns a list of all found file paths.

3. `main()`: The main function first specifies some paths and filenames. Then, for each session (only "RDT D1" is considered in the given example), it calls the `find_paths()` function to get a list of file paths that end with the string specified in 'look_for'. For each of these paths, it reads in two csv files ('id_csv' and 'dff_csv') as pandas DataFrames. The 'id_csv' DataFrame seems to contain identifiers for different types of data (denoted by "+", "-", "Neutral"), while 'dff_csv' DataFrame contains the corresponding data.

   The function groups the data from the 'dff_csv' DataFrame according to their types from the 'id_csv' DataFrame, calculates the mean and standard error of the mean (SEM) for each group and saves them to a dictionary (`d_description`). It also plots the means and the SEMs for each group.

   The function creates a pandas DataFrame from the dictionary, saves it as a csv file, and creates a plot with the mean and SEM values for each group. It also handles exceptions during the file reading and data processing steps, printing any errors encountered and continuing with the next file.

In general, this script is used for processing and visualizing data from multiple csv files that contain grouped data. Each group's data is averaged and plotted with error bars representing the standard error of the mean. The results are saved as both csv files and image files.

Behavioral_Calcium_DLC_Analysis/p_values_stacked_bar.py

---

analyze some neuroscience or behavior data, and you have collected counts of different experimental outcomes ('+', '-', 'Neutral') across different experimental sessions and conditions.

The python code you provided calculates the Chi-square p-values to determine if there is a significant difference in the distributions of the '+', '-', 'Neutral' outcomes among the different experimental sessions. This is achieved by comparing the observed frequencies of the outcomes to the expected frequencies under the assumption of independence (null hypothesis).

The Chi-square test is performed on these categories for three different sets of experimental conditions (Pre RDT RM, RDT D1, and RDT D2) and the p-values are printed out. If any of these p-values is smaller than the threshold you set for significance (often 0.05), you would reject the null hypothesis and conclude that there is a significant difference in the distribution of the '+', '-', 'Neutral' outcomes among the sessions.

The `compute_expected_frequencies` function calculates the expected frequencies of the three outcomes under the null hypothesis, assuming that they are equally distributed among the sessions.

The last part of your script starts to combine dictionaries from different conditions, presumably to compare the distributions of '+', '-', 'Neutral' outcomes across the conditions, but it seems to be incomplete. You have combined the first dictionary of the RDT D1 Session with the dictionaries from the Pre RDT RM Session, but you have not yet used these in any statistical test.

Behavioral_Calcium_DLC_Analysis/plot_indv_trials_per_cell.py

This code appears to be written to process, analyze, and visualize neurophysiological data, possibly related to neuroscience experiments involving cell activity measurements during certain events. The naming conventions hint towards measurements taken from the basolateral amygdala (BLA) region of the brain. Let's break down the components and their potential functions:

1. **Imports**: The code imports a variety of libraries, such as `numpy`, `matplotlib`, `pandas`, and `scipy`, among others. These libraries are essential for numerical operations, data processing, statistical analysis, and plotting.

2. **Function `avg_cell_eventrace`**:
   - This function plots the average of all rows (possibly representing individual trials) for each column (possibly representing timepoints).
   - If a column's average is greater than 10000, the column name and its values are printed.
   - The resulting plot has a title, x-axis labeled as "Time (s)", and y-axis labeled with "Average DF/F", where DF/F probably stands for "Delta F over F", a measure of fluorescence intensity changes in calcium imaging experiments.
   - The figure is saved with a specific name.

3. **Function `export_avg_cell_eventraces`**:
   - Exports the average data to a CSV file.

4. **Function `subwindow_auc`**:
   - Calculates the Area Under the Curve (AUC) for a specified sub-window of the data.

5. **Function `find_paths`**:
   - Searches and returns a list of file paths that match a specific pattern, recursively, from a root directory.

6. **Function `wilcoxon_analysis`**:
   - Performs a Mann-Whitney U test (also known as the Wilcoxon rank-sum test) to determine whether values in one list tend to be larger or smaller than those in another list. This is a non-parametric alternative to the independent two-sample t-test.

7. **Function `main`**:
   - Defines the main directory containing data, as well as a list of mice identifiers.
   - Defines timepoints (`t_pos`, `t_neg`, and `t`) for plotting.
   - Iterates over each mouse and session (only "RDT D1" is used in this case).
   - For each session, it retrieves file paths containing specific data for that session.
   - The cell name, type of event, and sub-event type are extracted from each file path.
   - Reads the data from each CSV file, transposes it to have timepoints as rows and trials as columns.
   - Each column (trial) is then plotted against the timepoints. This is referred to as the "spaghetti plot" (a term sometimes used in bioinformatics to represent individual traces in a single plot).
   - The resulting plot is saved.

8. **Execution**:
   - If this script is run directly, the `main()` function will be executed.

Based on the functionality provided by this script, it seems that it's part of a larger pipeline used to process, analyze, and visualize data from neuroscience experiments. The specific type of experiment isn't fully detailed, but there are hints pointing towards it being related to the BLA region of the brain and involving stimuli (events and sub-events) that elicit neuronal responses in different cells.

Behavioral_Calcium_DLC_Analysis/plt_indv_trials_per_cell_onecell.py

---

This script appears to be part of a larger data analysis pipeline, designed to process neuroscience-related data. Let's break down the code step by step:

Imports
The script starts by importing various required libraries and functions. These libraries include utilities for numerical processing (`numpy`), plotting (`matplotlib.pyplot`), directory and file operations (`glob`, `os`, `pathlib`), data processing (`pandas`), statistical analysis (`scipy.stats`, `sklearn.metrics`), and a custom module (`Cell`).

avg_cell_eventrace
This function takes a dataframe (`df`), the path to a CSV file, the cell name, and two flags (`plot` and `export_avg`). It calculates the trimmed mean of the columns in the dataframe. If the trimmed mean exceeds 10,000, it prints the column name and values. The function can either plot the data (based on the `plot` flag) or save it (based on the `export_avg` flag). The plots display an "average DF/F trace", which seems related to neuroscience imaging.

export_avg_cell_eventraces
This function saves the average DF/F trace data to a CSV file. It takes in the cell name, a list of average values, and an output path.

subwindow_auc
This function calculates the Area Under Curve (AUC) for a sub-window of each column in a dataframe. The AUC is a measure often used in statistics, and in this context, it likely relates to the integral of some signal over time. The function returns a list of AUC values.

find_paths
It takes a root path, a middle substring, and an end substring to search recursively for matching files within the root directory and its subdirectories. It returns a list of file paths that match the specified pattern.

wilcoxon_analysis
This function performs a Mann-Whitney U test (also known as the Wilcoxon rank-sum test) on two lists of values (`postchoice_list` and `prechoice_list`). This is a non-parametric test used to determine if two independent samples were drawn from populations with the same distribution. The function returns a symbol (`+`, `-`, or `Neutral`) based on the p-value of the test and a given significance level (`alpha`).

main
This is the primary function that executes when the script runs:

1. It defines a root directory (`MASTER_ROOT`) and other initial variables.
2. It processes a specific CSV file (specified by the `csv` variable). This file appears to contain data for a specific cell.
3. It reads and transposes the data, then plots individual traces from the data. This type of plot is often called a "spaghetti plot" in neuroscience because it displays multiple traces that can resemble spaghetti noodles.
4. The generated plot is then saved to a file with a specific naming convention.

Conclusion
The code processes and visualizes neuroscience-related data, specifically calcium imaging or electrophysiological traces from cells, likely neurons. The data appears to be organized by individual cells, sessions, and events. The script allows for calculating average traces, plotting individual traces, and performing statistical tests to compare different conditions or events.

PCA Methods

Behavioral_Calcium_DLC_Analysis/Methods/PCA/0_OrganizingData.py

---

It looks like you provided code that is designed to organize and prepare data from a neuroscience experiment (potentially involving mice) for analysis. The data seems to be structured in terms of 'trials' that have been recorded under various conditions.

1. **Utility Functions:**
   - `find_paths` & `find_paths_endswith`: These help in locating files within a given root directory based on certain criteria.
   - `strip_outcome`: Strips certain characters from a string and then splits it based on comma.
   - `custom_dissect_path` & `shock_dissect_path`: Extract useful metadata from the path of a file, such as the experimental block, event category, outcome, mouse id, session, and cell id.

2. **Trial Class:**
   - This class is used to represent a single trial. It contains metadata like block, event_category, outcome, and so on. Additionally, it also contains methods to retrieve specific segments of the recorded data (`get_prechoice_dff_trace`, `get_postchoice_dff_trace`, etc.)

3. **main function:**
   - This function's purpose seems to be to parse and reorganize data from its original structure into a new directory structure that makes it easier to analyze.
   - It navigates the directory structure to find CSV files that contain the data for individual cells in various trials.

    - It then reads these CSVs, extracts relevant data, and saves them under a new directory structure that is more conducive for analysis.
    - The directory structure is decided based on extracted metadata like block, event category, outcome, mouse id, etc.
    - If a CSV for a particular trial already exists, it appends the data. If not, it creates a new CSV.

4. **main_shock function:**
    - This function appears to be a specialized version of the main function but specifically tailored for processing data related to 'shock' trials.
    - Its structure and function are largely similar to the main function, with differences mainly in how metadata is extracted and how the directory structure is set up.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/0_OrganizingData_3bins.py

---

The Assistant sees that this is a script for organizing and processing data. Specifically, it appears to be related to neuroscience data with terms like "SingleCellAlignmentData", "trial", "cell", "timepoints", and "dff_trace", which might be deltaF/F calcium imaging data.

The general logic of the script:

1. **Helper Functions**:
    * `find_paths`: This returns file paths based on a defined structure, using `glob`.
    * `find_paths_endswith`: Finds paths that end with a specific substring.
    * `strip_outcome`: Process strings to extract outcomes from them.
    * `custom_dissect_path`: Dissect the parts of a path to get relevant data about the trial.

2. **Trial Class**:
    * This class defines an individual trial and provides methods to get data before and after a particular time point, based on `idx_at_time_zero`.

3. **Main Function Logic**:
    * Paths are set up.
    * Sessions are identified.
    * For each CSV file path found:
        1. Dissect the path to get metadata about the trial.
        2. Create directories to store this data.
        3. Read the CSV and process the data.
        4. For each row of data in the CSV:
            * Construct a trial object.
            * Create a new CSV file for this trial or append to it if it already exists.

4. Execution of the main function if the script is run as a standalone program.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/1_average_trial_mice.py

---

1. **Import Required Libraries**: The necessary libraries for data processing, file handling, PCA analysis, and plotting are imported.

2. **Define Helper Function - `find_paths_endswith`**:
   - **Purpose**: To find all the file paths in a given root directory that end with a specific pattern (like "trial_*.csv").
   - **Input**: A root directory and a string pattern the files should end with.
   - **Output**: A list containing all the paths that match the given pattern.

3. **Define Main Function - `main`**:
   - **Preliminaries**:
     - Lists of different `blocks`, `rew`, `mice`, and `sessions` values are defined. They dictate the directory structure and will be used to loop over different sets of files.
   - **Nested Loop**:
     - There's a 4-layer nested loop to go over every combination of `block`, `r` (reward), `mouse`, and `session`.
   - **For each Combination**:
     - **Determine Root Directory**: The root directory (`ROOT`) for the current loop iteration is determined based on the loop variables.
     - **Find Relevant CSV Files**: All the CSV files in this directory that follow the pattern "trial_*.csv" are fetched using the helper function.
     - **Data Collection**:
       - A dictionary `d` is initialized to hold cell data.
       - For each CSV file in the found list (`files`):
         - The file is read into a dataframe.
         - For each row in the dataframe (representing a cell's data in a trial):
           - If this cell's name is already a key in dictionary `d`, the trial data is appended to the corresponding list.
           - If not, a new key-value pair is added to `d` with the cell's name as the key and its trial data as the value in a list.
       - A dictionary `d_avg` is initialized.
       - For each cell (key) in `d`, the averages across all trials (averaging values at the same time point) are calculated and stored in `d_avg`.
     - **Data Storage**:
       - The average data in `d_avg` is converted into a dataframe `avg_df` and saved as "trials_average.csv" in the directory being considered.
     - **Error Handling**: If any error occurs while processing the CSV files or calculating averages, it prints the error and continues with the next file or directory.
   - **PCA Analysis (Commented Out)**:
     - For each time point column in the `avg_df` dataframe:

- The code seems to be aiming to run PCA analysis for the data of that time point, but this section is commented out, so it's not executed in the current script.

4. **Execution**: If this script is run as the main program, the `main()` function is executed, processing all the files and calculating the averages as per the logic described above.

In essence, this code aims to process neural trial data stored in CSV files. For each specific combination of parameters (`block`, `reward`, `mouse`, and `session`), it calculates the average neural activity across all trials and saves this average data in a new CSV file.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/2_generalized_avg_trial.py

---

1. Import necessary modules and packages.

2. `find_paths_endswith`: A function to retrieve paths ending with a specific string. It uses the `glob` module to retrieve these paths.

3. `Cell` class: Represents a cell with its name, data trace, and mean value derived from the trace.

4. `which_batch` function: Given a mouse name, determine its batch number.

5. `get_max_of_df` and `get_min_of_df` functions: Retrieve the maximum and minimum values, respectively, from a DataFrame.

6. `Trial` class: Represents a trial with specific attributes including mouse, session, event, sub-event, trial number, cell, and the data trace for the trial.

7. `find_different_subevents` function: From a list of CSV paths, finds distinct sub-events.

8. `fill_points_for_hm`: Fills any points in the heatmap that have a value of 0 with the corresponding value from the original DataFrame.

9. `sort_cells`: Sorts cells in a DataFrame based on the mean of their dFF traces.

10. `main` function:
   - Sets up paths and conditions to iterate over.
   - For each block, reward, and session, it finds all corresponding "trials_average.csv" files across different mice.
   - Creates a new directory to store the processed data.
   - For each of the mice files, it transposes and processes the data, appending new columns to the master DataFrame (`all_cells_df`) that will contain the data from all mice.
   - Checks and avoids adding columns where NaN values exist.

  - After processing all mice files, it sorts the cells based on their means.
  - Saves the sorted DataFrame to a CSV.

11. If the script is run as the main program, it will execute the `main` function.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/2_generalized_avg_trial_3bins.py

---

This script appears to be focused on processing some specific neuroscience (likely calcium imaging) data. Here's a brief overview of what it does:

1. **Import Libraries**: Libraries such as pandas, numpy, os, etc. are imported for use.

2. **Utility Functions**:

   * **find_paths_endswith**: Scans a given directory and its subdirectories to find files that end with a specific substring.
   * **get_max_of_df**: Finds the global maximum value from a DataFrame.
   * **get_min_of_df**: Finds the global minimum value from a DataFrame.
   * **find_different_subevents**: Retrieves unique subevent names from given paths.
   * **fill_points_for_hm**: A function that seems to fill in zeros in a transposed DataFrame with the original DataFrame values. The name suggests this might be used for a heatmap, but this isn't entirely clear.
   * **sort_cells**: Orders cell data by the mean of their calcium traces in descending order.

3. **Classes**:

   * **Cell**: Represents a single cell with a name and its dF/F trace. It also calculates the mean of this trace.
   * **Trial**: Represents a single trial with various properties like mouse, session, event, subevent, etc. and a dF/F trace for a given cell.

4. **which_batch Function**: Based on a mouse string, this determines which batch a mouse belongs to.

5. **Main Function**:
   * The main function is iterating over certain conditions (blocks, reward sizes, shock status, and sessions).
   * For each condition combination, it searches for specific CSV files and reads their data.
   * It then transposes this data and renames the columns to include the mouse number.
   * For each subsequent CSV file in the condition, it appends the data to a master DataFrame (`all_cells_df`).
   * If there are NaN values in a column, it seems to skip adding that column to the master DataFrame.

* Once all the data is aggregated, it sorts the cells using the `sort_cells` function.
  * Finally, the sorted DataFrame is saved to a new CSV file.

6. **Execution**: When the script is run directly, it will execute the main function.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/2_generalized_avg_trial_norew.py

---

It seems you have provided a Python script. This script performs some operations on .csv files, particularly trial data related to mice.

From what I can gather, the code does the following:

1. **find_paths_endswith**: Returns all paths from a root directory that end with a specified string.

2. **Cell Class**: A representation of a cell with attributes like cell name, dff trace, and its mean.

3. **which_batch**: Determines the batch of a mouse based on its ID.

4. **get_max_of_df**: Gets the maximum value across the entire DataFrame.

5. **get_min_of_df**: Gets the minimum value across the entire DataFrame.

6. **Trial Class**: Represents a trial with various attributes such as mouse, session, event, etc.

7. **find_different_subevents**: Determines different subevents by checking a particular section of the CSV paths provided.

8. **fill_points_for_hm**: Fills in 0 values for a heatmap (as suggested by the function name) from its transposed counterpart.

9. **sort_cells**: Sorts cells based on their mean in descending order.

10. **main Function**:
   - Iterates through various blocks, sessions, and shock conditions.
   - For each combination, it identifies the respective CSV files for mice.
   - It then reads the first CSV and transposes it.
   - Iterates over the remaining CSVs, transposing and adding their columns to a master DataFrame `all_cells_df`.
   - Once all the cells are added to `all_cells_df`, it sorts them and saves the result to a CSV file.

It's a decently long script, so let me know if you have a specific question or if there's a particular portion of the script you'd like assistance with!

Behavioral_Calcium_DLC_Analysis/Methods/PCA/comparing_two_sessions_pca_time.py

---

Your code appears to conduct a PCA (Principal Component Analysis) on some data from `.csv` files and visualize the results using 3D scatter plots. Let's break down what each part of the code is doing:

1. **Imports**: Libraries and functions required for your script.

2. `find_paths_endswith()`: This function uses the `glob` library to search for files in a given directory (`root_path`) that end with a specific string (`endswith`).

3. `pca_df()`: This function:
   - Receives a DataFrame `df`.
   - Applies PCA to the data.
   - Returns the transformed data (PCA scores), explained variance ratio, and the PC labels.

4. `combiner_same_block()`: This function:
   - Combines multiple CSV files into a single DataFrame.
   - Transposes the DataFrame (swapping rows with columns).
   - Creates a MultiIndex for the columns using information extracted from the CSV filenames.

5. `main()`: This is the main function that ties everything together:
   - Specifies paths to `.csv` files.
   - Specifies the colors and time vectors for the scatter plot.
   - Combines the data from multiple CSVs and then standardizes it (z-score).
   - Conducts PCA on the combined and standardized data.
   - Plots a scree plot to visualize the variance explained by each PC.
   - Plots a 3D scatter plot showing the first and second PCs.

6. `if __name__ == "__main__":`: This ensures that the `main()` function is only executed when the script is run as a standalone file, and not when it's imported as a module elsewhere.

Key Notes:

- **PCA** (Principal Component Analysis): A dimensionality reduction technique that identifies the axes in the dataset that maximize variance. It's used to capture the most essential features of the data. The code processes this using the `PCA()` function from `sklearn.decomposition`.

- **Standardization (z-score)**: Before applying PCA, it's common to standardize the data so that each feature has a mean of 0 and a standard deviation of 1. This ensures that all features contribute equally to the PCA. In your code, this is done using the `zscore` function you've defined, which standardizes values based on the overall mean and standard deviation of all data points.

- **Visualization**: After PCA, you're visualizing the data in two ways:
  1. **Scree Plot**: This shows the percentage of variance explained by each principal component. It helps in deciding how many components to retain.
  2. **3D Scatter Plot**: This visualizes the data in the space defined by the first three principal components. You're distinguishing different data groups using different colors.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/distance_bw_PCs.py

---

Your script seems to be performing principal component analysis (PCA) on some CSV data files and then plotting the difference in the first principal component (PC1) across different blocks. Here is a brief overview of what each function is doing:

1. `is_same_vector_dim(p, q)` - Checks if two vectors (lists or arrays) `p` and `q` have the same dimension.

2. `vectors_dim(p, q)` - Returns the dimension (length) of the vectors if they have the same dimension, otherwise returns `None`.

3. `squared_dist(p_ith, q_ith)` - Computes the squared distance between two scalar values.

4. `euclid_dist(p, q)` - Computes the Euclidean distance between two vectors, returns a list of Euclidean distances.

5. `distance(p, q)` - Computes the absolute difference between each corresponding element of two lists.

6. `euclid_dist_alex(t1, t2)` - Another function to compute the Euclidean distance between two vectors (but it isn't used in your script).

7. `find_paths_endswith(root_path, endswith)` - Fetches all the file paths from a given directory that end with a specified string.

8. `pca_df(df)` - Performs PCA on a given dataframe `df`, returning the transformed dataframe, the explained variance ratio, and the labels for the principal components.

9. `combiner_same_block(csvs_to_concat)` - Reads CSV files specified in a list, transposes them, and concatenates them such that they share the same multi-index block.

10. `main()` - Main function that specifies paths to three CSV files, reads and combines them, z-scores the combined data, and then performs PCA on the z-scored data. It then plots the difference in PC1 values across different blocks.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pc_v_time.py

Your code appears to be performing Principal Component Analysis (PCA) on a dataset, and then plotting the results. To help you better, I'll provide a general breakdown and analysis of the given code:

1. **Libraries & Functions**:
   - The code starts by importing various libraries necessary for file manipulation, data analysis, and plotting.
   - There are multiple custom functions such as `find_paths_endswith`, `pca_df`, and `combiner_same_block` which help in achieving different tasks throughout the main workflow.

2. **`find_paths_endswith`**:
   - This function takes in a root path and a string (`endswith`). It returns all files in the directory and its subdirectories that end with the given string.

3. **`pca_df`**:
   - This function takes a DataFrame and applies PCA to it, returning the transformed data, the percentage variance explained by each principal component, and labels for the principal components.

4. **`combiner_same_block`**:
   - It takes a list of CSV file paths, reads them into DataFrames, transposes them, and then concatenates them side-by-side. The column headers are modified to reflect the block name and the original columns from each CSV.

5. **`main` function**:
   - The main function is where the primary workflow of the code exists.
   - It starts by defining paths to three CSV files.
   - These CSVs are then combined into one DataFrame using the `combiner_same_block` function.
   - The combined DataFrame is then standardized (z-scored).
   - PCA is performed on the standardized data, and a scree plot is displayed to show the variance explained by the principal components.
   - The first principal component (PC1) values for three blocks are plotted against time.

6. **`if __name__ == "__main__":`**:

   - Ensures that the `main` function is only called when the script is run directly and not when imported elsewhere.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca.py

Both of the scripts you provided perform Principal Component Analysis (PCA) on some dataset, and they aim to visualize the results in different ways.

Here's a breakdown of the functionality for both of them:

1. **First Script**:
   - Combines data from multiple CSV files into one DataFrame.
   - Applies z-scoring to standardize the data in the concatenated DataFrame.
   - Performs PCA on the standardized data.
   - Displays a scree plot showing the percentage of explained variance for each principal component.
   - Plots the first principal component for different blocks against time.

2. **Second Script**:
   - Performs PCA for various data files categorized by block and size (Large or Small).
   - Plots the data points for the first two principal components on a scatter plot, with different colors representing different categories.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca_all_blocks_3D_time.py

---

Your code appears to read and process data files using PCA (Principal Component Analysis) and then visualizes the results using 2D and 3D scatter plots. You're comparing PCA across blocks for the given conditions (e.g., "Large" or "Small" rewards, whether there was a "shock", and the "session").

Here are some specific improvements and a brief breakdown of what you've done:

1. **File Paths and Dynamic Variables**: You use fixed paths for your files. Consider using `pathlib.Path` for creating and managing paths as you have already imported it. This allows for cleaner and more Pythonic path management.

2. **PCA Function**: `pca_df(df: pd.DataFrame)` is more versatile since it takes a DataFrame as an argument, instead of directly reading the CSV within the function. This is good as it allows for more flexibility in how you might use this function in the future.

3. **Data Combination**: You use `combiner_same_block()` to concatenate CSV files and provide multi-level indexing. This function now returns the data in a transposed form (`.T`).

4. **Data Standardization**: You have standardized the concatenated dataframe using a `zscore` function. This is important before applying PCA to ensure each feature has the same scale.

5. **PCA Visualization**: You have separated the plotting code for the Scree plot and the 3D PCA scatter plot. This makes it easier to understand and modify.

6. **Saving Plots**: You save the 3D scatter plot as an SVG file. This is beneficial if you need a scalable vector format for further editing or high-quality printing.

7. **3D Plotting**: When plotting in 3D, you've opted to plot the first two principal components across time. This provides a clearer view of how data points are distributed across different blocks over time.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca_multiindex_all_blocks.py

---

Your code essentially concatenates data from multiple CSV files, standardizes the data, performs Principal Component Analysis (PCA), and then visualizes the PCA in a 3D plot. Here's a breakdown of your code:

1. **Imports**: You import necessary libraries for data processing, analysis, and visualization.

2. **find_paths_endswith**: Finds all the file paths under the specified root path that ends with the given suffix.

3. **pca_df**: Takes a DataFrame and performs PCA on it. This function returns the DataFrame after PCA, the explained variance ratio for each principal component, and the labels for each principal component.

4. **combiner_same_block**: Concatenates data from multiple CSV files. The structure of the output DataFrame is such that the columns are multi-indexed, with the outermost index being the "block" and the innermost index being the column name from the original CSV.

5. **main**:
   - Specifies paths of CSV files and combines them using `combiner_same_block`.
   - The concatenated DataFrame is standardized (z-scored). For standardization, you calculate the mean and standard deviation of all data points across all CSVs, and then apply the z-score formula.
   - PCA is performed on the standardized DataFrame.

   - The explained variance for the first 10 principal components is plotted in a bar chart called a Scree Plot.
   - Data is then visualized in a 3D plot, where each point represents data from a CSV file and is plotted in the space defined by the first three principal components. The data is color-coded based on the block.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca_multiindex_oneblock.py

---

This script seems to be written to process and visualize data through Principal Component Analysis (PCA). Let's break down what this script does:

1. **Imports and Setup**:
    The script begins by importing necessary modules. These include modules for data processing (like `pandas`, `numpy`, and `os`), PCA (`sklearn.decomposition`), visualization (`matplotlib.pyplot`), and some others.

2. **Utility Functions**:

   a. `find_paths_endswith(root_path, endswith)`: This function is meant to return file paths from a root directory that end with a specified string. It's not used in the main function though.

   b. `pca_df(df)`: The function that performs PCA on the given DataFrame `df`. It returns the transformed data, the explained variance ratios, and the labels for each principal component. There's been a major change in the way the PCA data is structured. In the previous version, the data was transposed before returning; in this version, the data is transposed before performing PCA.

   c. `combiner_same_block(csvs_to_concat)`: This function reads in multiple CSV files, and concatenates them, creating a multi-level column index in the process. The higher level of the column index represents a certain category derived from the file path, while the lower level is the actual column names of the CSVs.

3. **Main Process**:

   a. The script sets a hardcoded list of paths to CSV files which need to be concatenated.

   b. These CSV files are combined using the `combiner_same_block` function.

   c. The combined DataFrame is then passed to the `pca_df` function, which returns the PCA transformed data, the explained variance ratios, and labels.

   d. A bar chart (Scree plot) is created to visualize the variance explained by each principal component.

e. Finally, a scatter plot is produced to visualize the data in the space of the first two principal components, color-coding the points based on a certain category ('Small' or 'Large') which is inferred from the multi-level index of the DataFrame.

From the changes in the script, the main difference seems to be the perspective of the analysis. In the first script, the data was interpreted with a 3D plot, looking at three principal components, whereas in the second script, the focus is on a 2D plane, looking at just the first two principal components. The categorization in the first script is based on 'Block 1', 'Block 2', and 'Block 3', whereas in the second script, it's based on 'Small' and 'Large'. This might suggest that the same analysis method is being applied to different datasets or different categorizations of the same dataset.

If you have any questions or if there's a specific aspect you'd like more insight into, please let me know!

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca_oneblock_oneevent.py

---

This version of the script has further modified the data processing and visualization method. Let's analyze this revised code:

1. **Imports and Setup**:
   The imported modules remain largely the same.

2. **Utility Functions**:

   a. `find_paths_endswith(root_path, endswith)`: This function still finds file paths that end with a specific string, but it's not used in the main code.

   b. `pca_df(df)`: The PCA processing function is consistent with the previous version. It takes in a dataframe, transposes it, and returns the PCA transformed data, the explained variance ratios, and labels.

   c. `combiner_same_block(csvs_to_concat)`: This function combines CSV files with a multi-level column index. It has remained consistent, but it's not used in the main function of this version.

3. **Main Process**:

   a. The script now only reads a single CSV file directly into a DataFrame called `concatenated_df`, and then it trims the first column. Previously, two CSV files were combined.

b. The script then performs PCA on this single DataFrame.

c. A Scree plot is generated, visualizing the percentage of variance explained by the first 10 principal components.

d. Instead of categorizing and coloring the scatter plot based on 'Small' or 'Large' as previously, the script now plots all the data points in a single color ('indianred') with the label "Large". This could be a placeholder, or it might mean that the focus is no longer on comparing two groups. All the data points are plotted on a 2D scatter plot using the first and second principal components.

The modified script has streamlined the PCA process for a single dataset, rather than comparing multiple datasets or subsets. It's simpler and more straightforward in its goal: perform PCA on a dataset and visualize the first two principal components.

The 3D visualization and data categorization based on 'Small' and 'Large' have been removed, indicating a possible shift in the analytical goals or the nature of the dataset being analyzed.

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pca_same_pcs.py

This code provides functionality to perform Principal Component Analysis (PCA) on the data contained in CSV files, and then visualizes the first two principal components in a scatter plot.

Here's a breakdown of the script:

1. **Imports and Setup**:
 - The necessary libraries and modules are imported.
 - The setup is similar to the previous versions with the addition of removing some unnecessary imports.

2. **Utility Functions**:

a. `find_paths_endswith(root_path, endswith)`: This function finds file paths that end with a specific string but is not utilized in the main function.

b. `pca_df(df)`: This function performs PCA on a dataframe `df`. It first trims the first column, transposes the dataframe, and then applies PCA. It returns the transformed PCA dataframe and the percentage variance explained by each component.

c. `combiner(csvs_to_concat)`: This function reads multiple CSV files from a list and concatenates them along the columns. The resulting dataframe will have columns from all the CSVs.

3. **Main Process**:

   a. The code contains several commented-out lists of file paths. Each list corresponds to a set of CSV files. The active list for this script's run contains CSVs corresponding to the '3.0/Small' and '3.0/Large' categories. You can uncomment other lists and use them if needed.

   b. The script combines the data from these CSV files into a single dataframe named `concatenated`.

   c. PCA is performed on this concatenated dataframe.

   d. The first two principal components are then plotted in a scatter plot. The data points are labeled as "Large". This might be a placeholder since the dataset includes both 'Small' and 'Large' categories.

   e. The axes of the scatter plot are labeled with the corresponding percentage of variance explained by the two principal components.

The script has become cleaner and more concise compared to the previous versions. This version focuses on reading data from a chosen set of CSV files, performing PCA, and visualizing the results. It provides flexibility in the choice of datasets by allowing the user to switch between different sets of file paths (although only one set is active per run).

Behavioral_Calcium_DLC_Analysis/Methods/PCA/pearson_corrmaps.py

---

The provided code seems to be a script used for analyzing experimental data. Here's a brief overview of the code:

1. **Imports**:
   - Modules for data manipulation (`pandas` and `numpy`), file and system operations (`os`, `glob`, `pathlib`), statistics (`statistics`, `scipy.stats`), and visualization (`matplotlib.pyplot`, `seaborn`) are imported.

2. **Function Definitions**:
   - `find_paths_endswith`: Finds all file paths ending with a specific substring.
   - `which_batch`: Determines the batch based on the mouse number.
   - `get_max_of_df`: Fetches the maximum value across a DataFrame.
   - `get_min_of_df`: Fetches the minimum value across a DataFrame.
   - `find_different_subevents`: Retrieves unique subevents from a list of CSV paths.
   - `fill_points_for_hm`: A method that updates a heatmap DataFrame with values from another DataFrame.
   - `heatmap`: Creates and saves a heatmap visualization.

   - `sort_cells`: Sorts cells based on their mean value.

3. **Class Definitions**:
   - `Cell`: Represents a cell with attributes such as name and its data.
   - `Trial`: Represents an experimental trial with various attributes like mouse, session, event, etc.

4. **Main Procedure - `make_pearson_corrmaps`**:
   - This procedure reads the data, combines it from different sources, sorts cells based on their means, and then creates a Pearson correlation map. The correlation map is saved as both an image and CSV file.

5. **Execution Point**:
   - If the script is run as the main program, it calls the `make_pearson_corrmaps` function.

Behavioral_Calcium_DLC_Analysis/Opto/0_singleCellAlignment_opto.py

---

The provided script is an analysis pipeline tailored for specific scientific data – presumably related to studying speed, and some form of neuronal activity or physiological recordings. I'll provide a high-level summary of what the script does:

1. **Import Modules**: Essential Python libraries for data processing and visualization (e.g., `numpy`, `matplotlib`, `pandas`, `seaborn`, and `scipy`) are imported.

2. **avg_cell_eventrace Function**: Plots and exports an average speed from the dataframe. This is based on the mean of each column, which is then plotted if `plot` is set to `True`, and exported to CSV if `export_avg` is set to `True`.

3. **make_avg_speed_table Function**: Generates a table for average speed over a defined time window and exports the table to CSV. The table structure will contain time on the x-axis and average speed on the y-axis.

4. **plot_avg_speed Function**: This function generates a plot of average speed based on the CSV created by the previous function.

5. **export_avg_cell_eventraces Function**: Exports the average speed of each time window event to CSV.

6. **zscore Function**: Computes the z-score of an observation.

7. **custom_standardize_limit_fixed Function**: This function z-scores (standardizes) the dataframe values based on a specific baseline. The z-scoring stops at a certain limit.

8. **find_paths Function**: It searches and returns file paths based on a given root path and filename patterns.

9. **gaussian_smooth Function**: Applies a 1D Gaussian smoothing filter to a dataframe. This can help in reducing noise and highlighting actual patterns.

10. **main Function**: The core execution pipeline:

  - It has a list of event types it cares about, `list_of_combos_we_care_about`.
  - For each combo/event type, it searches for CSV files using `find_paths`.
  - For each found CSV file:
    - Reads the file into a dataframe.
    - Z-scores the dataframe using `custom_standardize_limit_fixed`.
    - Optionally applies Gaussian smoothing to the dataframe.
    - Computes average speed for each time window event using `avg_cell_eventrace`.
    - Saves the processed dataframe to a new CSV file.

11. The script ends with an `if __name__ == "__main__":` guard to ensure that the `main` function is called only when the script is run directly, and not when it's imported elsewhere.

General Observations:

- **Logging and Exception Handling**: The script contains basic exception handling (`try...except` blocks) that catch `TypeError` and `AttributeError` exceptions and simply prints them without stopping the script.

- **File and Directory Operations**: The script makes extensive use of file path manipulations to save and load data. This includes the use of `os.path.join`, `str.replace`, and `glob.glob` for searching patterns in filenames.

- **Pandas Dataframe Manipulation**: The script does a lot of dataframe reshaping using `pandas` – for example, reading CSVs, transposing dataframes, adding columns, etc.

- **Statistical Analysis**: Several functions, like `stats.tmean` and `stats.tstd` (from `scipy`), are used for statistical analysis.

In summary, this script is a comprehensive data processing pipeline designed for a specific set of scientific data. To further improve or adapt this script, it's essential to understand the biological or scientific context in which it's being used.

Behavioral_Calcium_DLC_Analysis/Opto/1_apply_filter_indv_trials.py

---

Your code appears to be a script that processes, smooths, and visualizes speed data over time, stored in `.csv` files. Here's a brief rundown of its structure and what each part does:

1. **Imports**:
   - Libraries necessary for data manipulation (`numpy`, `pandas`), visualization (`matplotlib.pyplot`), and some others are imported.

2. **Utility Functions**:
   - `find_paths`: Searches recursively for files matching a pattern.
   - `str_to_int`: Parses a list of string numbers (could be positive or negative) and converts them into rounded decimal strings.
   - `add_val`: Appends a value to a numpy array.
   - `plot_trace_deco`: A decorator for the plotting function, set up to standardize the plotting process.
   - `plot_trace`: Uses the decorated version to plot the trace.
   - `zscore`: Computes the z-score for a value given the mean and standard deviation.

3. **Main Functionality**:
   - `main`:
     - Defines a root directory to search for `.csv` files.
     - For each combination of events (as named in `list_of_combos_we_care_about`), it finds the relevant CSV files.
     - For each of these files, it does the following:
       * Reads the CSV into a DataFrame.
       * For each row in the DataFrame (representing an individual event trace):
         - It applies a Savitzky-Golay filter to the data to smooth it.
         - Plots the smoothed data.
       * Saves the smoothed data back to a new CSV.
       * Plots all of the smoothed traces on a single graph and saves it to an image.

4. **Execution**:
   - If the script is the main file being run, it will execute the `main` function.

Behavioral_Calcium_DLC_Analysis/Opto/2_avg_indv_traces.py

Your code looks like it's designed to handle a fairly specialized analysis related to some form of behavioral analysis data. Here's a summary of what each part of the code does:

1. **Imports**: The necessary libraries and functions for processing and plotting are imported.

2. **`make_avg_speed_table` Function**:
   - A function to generate an average speed table.

   - It reads a CSV file into a DataFrame, modifies the structure and columns of the DataFrame, calculates the time and average speed, and then saves this data to a new CSV file.

3. **`add_val` Function**:
   - A simple function that appends a value to an array and returns the resulting array.

4. **`plot_avg_speed` Function**:
   - This function plots the average speed of trials from a given CSV file.
   - The resulting graph shows the average speed of trials relative to some trigger time and saves it as a PNG.

5. **`find_paths` Function**:
   - This function uses glob to search for files in directories recursively. It returns all file paths that match a specific pattern.

6. **`main` Function**:
   - This is the primary execution point when the script is run.
   - It sets the `session_root` which seems to be a base directory from where files are to be searched.
   - A list `list_of_combos_we_care_about` is provided which seems to filter or specify certain types of data/events for which analysis should be performed.
   - For each `combo` in that list, it finds the paths of files matching a specific pattern.
   - For each such file, it computes the average speed table, saves this data, and then plots the average speed.

7. **`one_process` Function**:
   - This seems to be a test or example function where a specific file is taken and processed to generate the average speed table and then plotted. This could be used for debugging or checking the performance on a single file before executing the script on all files.

8. The final lines of the script specify that when the script is run, the `main()` function should be executed. The `one_process()` function is commented out, so it won't execute unless uncommented.

Behavioral_Calcium_DLC_Analysis/Opto/3_rename_cols.py

Your script appears to be handling CSV files related to some kind of behavioral analysis, where the focus seems to be on renaming columns based on the specific mouse involved in the analysis. Let's break down the primary components:

1. **`find_paths` Function**:
   - This function searches for files recursively in the provided directory that match the pattern defined by the combination of `root_path`, `middle`, and `endswith`.
   - It returns all the matching file paths.

2. **`main` Function**:
   - The function sets a `session_root` that is the base directory from where files will be searched.
   - A list `list_of_combos_we_care_about` is defined, which seems to filter or specify certain types of data/events for which the renaming operation should be performed.
   - For each `combo` in the list, it finds the paths of files matching the given filename pattern (`speeds_z_-5_5_savgol_avg.csv`).
   - For each such file, it identifies the mouse from the directory structure, reads the CSV into a DataFrame, and renames a column based on the identified mouse's name. The updated DataFrame is then saved back to the original CSV.

3. The last section (`if __name__ == "__main__":`) indicates that when the script is run directly (not imported as a module), the `main()` function should be executed.

Behavioral_Calcium_DLC_Analysis/Opto/4_downsample_velocity.py

---

It seems you've created a script to process and downsample CSV files based on their size (frames per second). Here's a brief overview of each function:

1. **`find_paths` Function**: Finds and returns file paths that end with a specified string.

2. **`main` Function**: Processes CSV files named `speeds_z_-5_5_savgol.csv`. Depending on the file's length, it either saves the original file as an "unsampled" version or downsamples it by skipping rows.

3. **`main_avg` Function**: Similar to the `main` function but for CSV files named `speeds_z_-5_5_savgol_avg.csv`.

4. **`main2` & `main3` Functions**: Process the "unsampled" files in another directory, downsampling them.

5. **`one` & `one_avg` Functions**: They are like the `main` and `main_avg` functions but for a specific file.

In the end, you're calling `main()` and `main_avg()` to process the files.

Behavioral_Calcium_DLC_Analysis/Opto/5_combining_mice_avgs.py

---

Your code seems to focus on extracting, analyzing, and plotting data related to mouse sessions. Before diving into what the code does step by step, here are some general observations:

1. **Imported Libraries:** The code imports various libraries that help in file handling (`os`, `pathlib`, `glob`), data processing (`pandas`, `numpy`), and plotting (`matplotlib`).

2. **Utility Functions:**

    - `find_csv_files()`: Searches for `.csv` files with specific starting names within a session path.

    - `add_val()`: Appends a value to a NumPy array.

    - `plot_indv_speed()`: Reads a given `.csv` file, processes its columns, and creates a plot to visualize the data.

3. **Main Data Processing Function: `concat_all_cells_across_similar_sessions()`**:

    - This function processes a list of paths to `.csv` files and organizes the data into a hierarchical dictionary structure (`between_mice_d`).

    - It categorizes data by mouse name, session type (Choice/Outcome), and other criteria.

    - After processing and organizing data, it creates a dataframe for each session type, combo, and subcombo grouping, which is then saved to a `.csv` file. Then, it calls the `plot_indv_speed()` function to generate a plot for each `.csv` file.

4. **Main Executable Logic (`main()` function)**:

    - The main function defines the `ROOT_PATH`, finds all `.csv` files within that path that meet a certain condition, and then calls `concat_all_cells_across_similar_sessions()` to process these `.csv` files.

    - It seems you are working on data related to opto speed analysis for mice.

5. **Data and Directory Structures**:

    - The `concat_all_cells_across_similar_sessions` function seems to be working with a specific directory structure. It uses this structure to extract mouse names, session types, combos, and subcombos from the path names of the `.csv` files. Adjusting this function to work with a different directory structure would require modifying how it interprets and splits the paths.

6. **Error Handling and Logging**:

   - The function `concat_all_cells_across_similar_sessions` contains some error handling for cases when the lengths of data columns do not match, leading to a 'jagged array'. In such cases, it logs the session type, combo, and subcombo associated with the jagged array and then attempts to create a dataframe using a different method.

7. **Plot Customization**:

   - In the `plot_indv_speed()` function, every nth x-axis tick label is made visible while the others are hidden, for better readability.

Behavioral_Calcium_DLC_Analysis/Opto/6_apply_filters_trials.py

---

Your code is quite comprehensive. It primarily works with time-series data that represents some kind of speed measure taken at different time intervals. This data is loaded from a CSV file, Z-score standardized, and then smoothed using a Savitzky-Golay filter. The result is both plotted and saved as a new CSV file.

Here's a brief outline:

1. **Functions**:
   - `find_paths`: Finds paths that match the given criteria in a root directory.
   - `zscore`: Computes the Z-score for a value.
   - `custom_standardize_limit_fixed`: Z-scores the values in a DataFrame based on a specified baseline range.
   - `gaussian_smooth`: Applies Gaussian smoothing to a DataFrame.
   - `str_to_int`: Converts an array of strings, possibly containing dashes, into a corresponding array of integers.
   - `add_val`: Appends a value to an array.

2. **Main Logic**:
   - The `main` function processes a set of CSV files in a root directory. For each file:
      1. Load the data.
      2. Apply Z-score standardization based on a specified baseline range.
      3. Apply the Savitzky-Golay filter to smooth the data.
      4. Plot and save the results.

   - The `process_one` function does the same as the `main` function but only for one specific CSV file.

3. **Execution**:
   - If the script is run as the main module, it calls the `main` function.

Behavioral_Calcium_DLC_Analysis/Opto/7_avg_and_sem.py

This code provides functionality to process certain types of CSV files (speed data) and generate averages and standard error of the mean (SEM) for each time point across all mice in the dataset. It does this for either a single file (with `process_one`) or for all matching files in a specified root directory (with `main`).

Here's a brief explanation:

1. **Functions**:
   - `find_paths_endswith`: This function finds all files that end with the specified suffix within a root path and its subdirectories.
   - `main`: This function scans the root directory for files named "all_speeds_z_-5_5_savgol_renamed.csv", reads each of them, and computes the average and SEM for each time point across all mice. It then saves the results to a new CSV named "all_mice_avg_speed_w_sem.csv".
   - `process_one`: This does the same operation as `main`, but for only one specific CSV file.

2. **Execution**:
   - If the script is run as the main module, it calls the `main` function. However, there's also a commented-out call to `process_one`, suggesting that you might want to run it sometimes.

Behavioral_Calcium_DLC_Analysis/Opto/8_visualize_stack.py

The code you've provided seems to be for data visualization and analysis for an optogenetics experiment. Here's a brief overview:

1. **Imports:** Necessary libraries like matplotlib, pandas, scipy, seaborn, etc., are imported.

2. **Cell Class:** Defines a cell with attributes such as cell name and `dff_trace` (which could be related to calcium imaging). Each cell object also calculates its own mean value for a particular time range.

3. **sort_cells Function:** This function sorts cells based on their mean value and returns a dataframe that's been reorganized accordingly.

4. **get_max_of_df and get_min_of_df Functions:** Return the maximum and minimum values from the entire dataframe, respectively.

5. **main_gridspec Function:** Creates a plot of the z-scored average speed for two treatments across a time series. The paths seem hardcoded, and the figure saved has a predefined name and path.

6. **find_subcombos Function:** Retrieves the sub-combinations of trials or conditions.

7. **main Function:** This is the primary function that puts everything together.
   - Loads data files from a defined ROOT directory.
   - Iterates through different treatment combinations and generates plots comparing their effects on the 'Avg. Savgol Z-scored Speed'.
   - It calculates the Pearson correlation coefficient between the two treatments for each plot.
   - Fills the area between the mean and standard error of the mean (SEM) for both treatments.
   - Saves these plots to a defined directory.

8. The call to `main()` at the bottom executes everything.