mousestyles Documentation

Release 0.1

2016 UC Berkeley MA students

May 14, 2016

CHAPTER

ONE

PROJECT REPORT

1.1 Overview

1.1.1 Statement of Problem

For this project, our primary data source is mouse behavioral data from the Tecott Lab at UCSF. [1] The lab has recently developed a method for continuous high-resolution behavioral data collection and analysis, which enables them to observe and study the structure of spontaneous patterns of behavior ("Lifestyles") in the mouse *[Tec03][TN04][GSJ+08][AP14]*. They have found that using this method: 1) reveals a set of fundamental principles of behavioral organization that have not been previously reported, 2) permits classification by genotype with unprecedented accuracy, and 3) enables fine dissection of behavioral patterns.

Project Goal

The goal of this project is to explore the effects of genetics on behavior in mice and extrapolate these findings to improve treatment of psychiatric diseases in humans.

Computational Ethology

Ethology is the study of animal behavior, and it generally follows one of two approaches. The first approach was developed by B.F. Skinner and relies heavily on laboratory experiments. Skinner postulated that since behavior is predictable, it should be controllable. His most prominent experiment involved training pigeons through *operant conditioning*. He found that the pigeons had the ability to learn new behaviors under a reward system. However, the setting of the experiment was incredibly artificial and controlled. In contrast, Konrad Lorenz believed that the only way to truly understand animal behavior is to observe them in their natural context as behavior induced by an artificial environment may not reflect animal behavior in their natural environment. Lorenz's most prominent experiment discovered that a young goose will instinctively bond with the first moving object that it perceives in order to better recognize its own species. This experiment was conducted in a more natural setting to minimize human manipulation.

This project adopts Lorenz's approach—observing the whole spectrum of animal behavior in their natural state. Previous approaches relied on human observation to score animal behavior. However, slow data collection, low dimensional data, imprecise and subjective measurements, and human visual and language limitations impeded data collection and analysis. Using modern quantitative tools for measurement, description, and analysis, the field of *computational ethology* has emerged to solve these issues. Together, modern mathematics, engineering, and computer science have the potential to establish a causal relationship between genetics and behavior.

[1] http://www.neuroscience.ucsf.edu/neurograd/faculty/tecott.html

Why Mice?

Although the human genome was first sequenced over a decade ago, the relationship between genetics and behavior is still not well understood. For ethical reasons, genes cannot be systematically manipulated in humans. Therefore, the familiar scientific testing approach must be edited—enter mice. There are several reasons the mouse has become the mammal of choice in human behavioral studies. Chief among them is the fact that "approximately 99% of mouse genes have human counterparts—conversely, mouse versions (orthologs) can be identified for 99% of human genes." Additionally, the brain organization and behavioral responses of humans and mice display many similarities. For example, both mammals display complex processes like hunger and fear. This allows scientists to more easily identify and track these behaviors. In addition, the extreme similarities of intra-strain mice, the space efficiency of maintaining their caged environments, and the speed of reproduction make mice a logical alternative to testing on humans.

1.1.2 Methodology

The traditional technology to analyze behavior is time-intensive and labor-intensive. For example, recording data requires researchers to track behavior uninterruptedly. This experiment utilized a method for continuous high-resolution behavioral data collection and analysis. The *home cage monitoring (HCM) system* utilized is a network of photobeam feeding detection, drinking detection, and activity platform sensors that records mouse *active states* and *inactive states* automatically and incessantly over 24-hour periods where 1 hour per day was reserved for HCM maintenance. This facilitates objective, multi-dimensional computer tracking, providing a higher degree of accuracy compared to human observation.

In this project, the HCM system tracked 170 mice, representing 16 *strains* or approximately 94% of the genome, logging 500,000 behavioral events per mouse per day over 12 days of data collection after 5 days of acclimation. The scope of the study was limited to male mice as females tended to display cyclical changes in behavior. Mice within each strain display strong homogeneity and low variability in genetic composition. This is in part due to high levels of inbreeding which decreases the randomness between individual mouse samples. By studying the behaviors of theses 16 different strains – each strain with their own unique genetic makeup – we can begin to understand the importance of genetic makeup on mice behavior. We can then extrapolate to humans thanks to the remarkable genetic similarity between the two species.

1.1.3 Statement of Statistical Problems

This project utilized sophisticated statistical methods to process data, including machine learning algorithms and statistical inference. The whole project can be divided into 6 sub-projects:

- *Behavioral Model*
- *Exploration and Path Diversity*
- *Dynamics of AS Patterns*
- *Ultradian and Circadian Analysis*
- *Classification and Clustering of Mice*
- *Power Laws and Universality*

1.1.4 Glossary

- **Active State (AS):** The active state in this model is when the mouse is using energy, such as foraging, patrolling, eating, or drinking. Active states are energetically costly and can be dangerous in a natural environment.
- **Computational Ethology:** The use of mathematics, engineering, and computer science to overcome the difficulties that come from using humans to score animal behavior.

- **Ethology:** The study of animal behavior, including the phenomenological, causal, genetic, and evolutionary aspects.
- **HCM System:** The system used in this experiment to track variables of interest. The HCM System included photobeam sensors at the feeding stations, capacity based sensors at the drinking station, and an activity platform for position detection using an (x,y) system.
- **Home Environment:** The home environment is the cage of each mouse containing a home base, a food station, and a water station.
- **Inactive State (IS):** The inactive state in this model is when the mouse is in a state of energy conservation, such as sleeping or resting at the home base.
- **Operant Conditioning:** Altering of behavior through the use of positive reinforcement which is given to the subject after eliciting a desired response.
- **Phenotype:** The set of observable characteristics of an individual resulting from the interaction of its genotype with the home environment.
- **Strain:** A strain here is a genetic variant or sub-type of of the more general mouse population.

1.1.5 Data

The data includes two directories, intervals and txy_coords, and a npy file named all_features_mousedays_11bins. The all_features_mousedays_11bins.npy contains a $9 \times 1921 \times 11$ matrix, which represents 9 features among 1921 mouse days in 11 2 hour bins for a day, the 9 features are:

- **Food (F):** records the food consumption (g) for a certain mouse day and a certain time bin.
- **Water (W):** records the water consumption (g) for a certain mouse day and a certain time bin.
- **Distance (D):** records the movement distance for a certain mouse day and a certain time bin.
- **ASProbability (ASP):** records the AS time proportion in the certain time bin.
- **ASNumbers (ASN):** records the numbers of AS in the certain time bin.
- **ASDurations (ASD):** records the total duration of AS in a certain bin.
- **ASFoodIntensity (ASFI):** equals F/ASP.
- **ASWaterIntensity (ASWI):** equals W/ASP.
- **MoveASIntensity (ASMI):** equals D/ASP.

The intervals directory has 6 sub-directories, all sub-directories have about 33 files for 3 strains, and for each strain there are 11 days data:

- **F:** records start and stop time of eating behaviors for a certain strain and a certain day.
- **W:** records start and stop time of drinking behaviors for a certain strain and a certain day.
- **AS:** records start and stop time of AS for a certain strain and a certain day.
- **M_AS:** records start and stop time of movements in AS for a certain strain and a certain day.
- **IS:** records start and stop time of IS for a certain strain and a certain day.
- **M_IS:** records start and stop time of movements in IS for a certain strain and a certain day.

The txy_coords directory has 5 sub-directories,all sub-directories have about 33 files for 3 strains, and for each strain there are 11 days data:

- **CY,CX,CY:** records the position (x,y) in time t for a certain strain and a certain day.
- **C_idx_HB:** indicates whether the mouse is in HB or not at time t.

- **recordingStartTimeEndTime:** records the start and stop time of tracking (x,y,t) for a certain strain and a certain day.

1.2 Behavioral Model

1.2.1 Statement of Problem

In order to differentiate mice, we need to create a detailed behavior profile to describe:

1. how they drink, feed or move (locomotion) and
2. how they translate between active state or inactive state.

We have the following key background information from the paper:

- **Home Cage Monitoring System(HCM) HCM cages were spatially** discretized into a 12 x 24 array of cells and occupancy times for each MD were computed as the proportion of time spent within each of the 288 cells. To determine whether animals establish a Home base, HCM cages were spatially discretized into a 2 x 4 array of cells, and occupancy times for each mouse day were calculated as above. In the experiment, 56/158 animals displayed largest occupancy times in the cell containing the niche area, which was considered to be their Home base location.
- **Active and Inactive State Mice react differently during** Active state and Inactive states and all behavioral record should be classified into 2 mutually exclusive categories, Active States (ASs) and Inactive States (ISs). To designate ISs, we examined all time intervals occurring between movement, feeding, and drinking events while the animal was outside the Home base. Those time intervals exceeding an IS Threshold (IST) duration value were classified as ISs; the set of ASs was then defined as the complement of these ISs. Equivalent mathematically, ASs can also be defined as those intervals resulting from connecting gaps between events outside the Home base of length at most IST; ISs are then defined as the complement of these ASs. (*Active State Organization of Spontaneous Behavior Patterns*, C. Hillar et al.)

1.2.2 Statement of Statistical Problems

The above flowchart shows the key metrics that are required by the study to capture the behavioral profile:

The main focus is 3 key states of the mice i.e. *Drinking*, *Feeding* and *Locomotion*.

Each of these metrics can be seen visually in the slides referenced below. Each metric is a tree, decomposed into two child node metrics, whereby when the child nodes are multiplied together, they yield the parent metric. We illustrate the relevant calculations in the case of **drinking state**:

Drinking Consumption Rate: `Total Drinking Amount/ Total Time` (mg/s)

Active State Prob: `Active Time/ Total Time`

Drinking Intensity: `Total Drinking Amount/ Active Time` (mg/s)

- **Drinking** Bout Rate: `Number of Bouts/ Active Time` (bouts/s)
- **Drinking Bout Size:** **`Total Drinking Amount/ Number of Bouts`** **(mg/bout)**
  - **Drinking** Bout Duration: `Total Drinking Time/ Number of Bouts` (s/bout)
  - **Drinking Bout Intensity:** **`Drinking Amount/ Total Drinking Time`** **(mg/s)**
    - **Drinking** Bout Event Rate: `Number of Events/ Total Drinking Time` (events/s)
    - **Drinking** Event Size: `Total Drinking Amount/ Number of Events` (mg/event)

Project 1: Behavioral Model

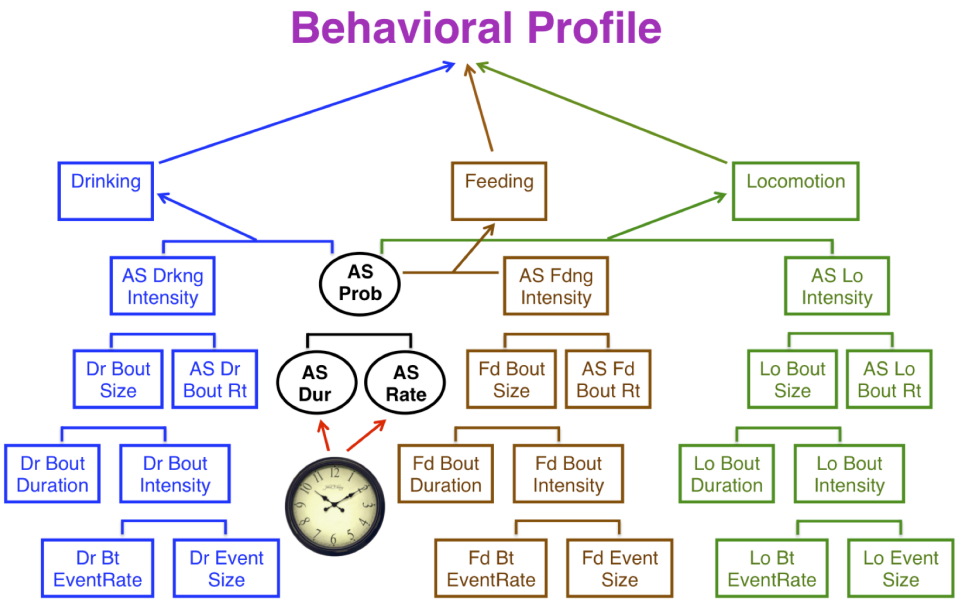


Fig. 1.1: Behavioral Profile (image courtesy of Tecott Lab)

1.2.3 Data

Our underlying functions depend on the following key data requirements for each **mouse, strain and day**:

- Active State
- Inactive State
- Moving Active State
- Moving Inactive State
- Total Distance Travelled (meters)
- Food Consumption (grams)
- Water Consumption (milligrams)

These data requirements are sourced via the `mousestyles data loader` utilities.

The Food and Water consumption data was only provided on a daily basis and our `behavior` utilities assume that such quantities are uniformly consumed over the time period studied.

1.2.4 Illustrative Examples

We illustrate the use of the `behavior` utilties with a motivating research question:

How do do the key feeding metrics compare across 2 different mice for the entire 11 days?

In this code, we see a few important features. First, we create behavior trees using the `compute_tree` function, and demonstrate their pretty-printing functionality. Then, we merge lists of trees into one larger tree of lists, which we can then summarize or turn into a Pandas data frame. Finally, we show how to plot the results using `matplotlib`.

```
#!/usr/bin/python

import mousestyles.behavior as bh
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# get a tree for each day for two different mice
mouse1_trees = [bh.compute_tree('F', 0, 0, d) for d in range(11)]
mouse2_trees = [bh.compute_tree('F', 0, 1, d) for d in range(11)]
print(mouse1_trees[0])
print(mouse2_trees[0])

# merge each of the trees for the two mice
mouse1_merged = bh.BehaviorTree.merge(*mouse1_trees)
mouse2_merged = bh.BehaviorTree.merge(*mouse2_trees)
print(mouse1_merged)
print(mouse2_merged)

# get the means for the two mice
print(mouse1_merged.summarize(np.mean))
print(mouse2_merged.summarize(np.mean))

# turn the tree into pandas DataFrame
mouse1_df = pd.DataFrame.from_dict(mouse1_merged.contents)
# this gives us another way to compute summary statistics
print(mouse1_df.describe())
```

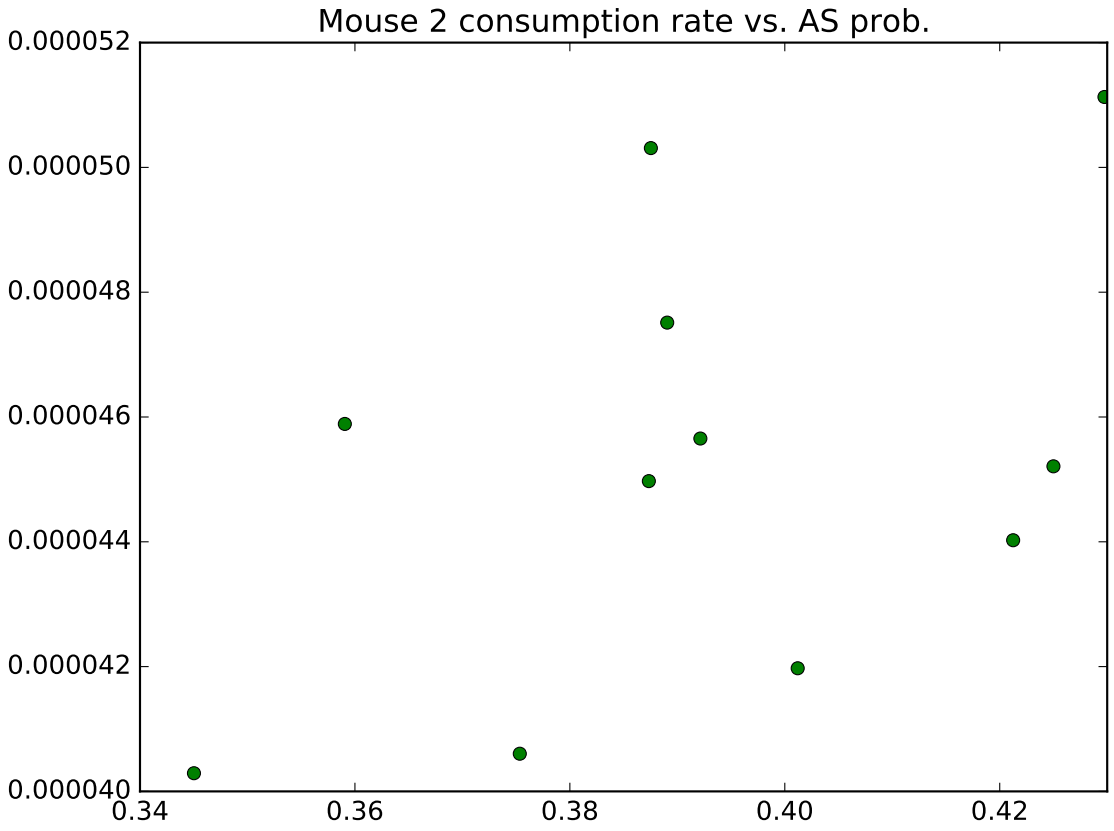
```
# plot the AS Probability over time
plt.plot(range(11),
         mouse1_merged['AS Prob'], 'r',
         mouse2_merged['AS Prob'], 'g')
plt.title('Mouse 1 and 2 AS prob. over time')
plt.show()
```


```
# plot consumption vs. AS prob
plt.plot(mouse2_merged['AS Prob'],
         mouse2_merged['Consumption Rate'], 'go')
plt.title('Mouse 2 consumption rate vs. AS prob.')
plt.show()
```

1.3 Exploration and Path Diversity

1.3.1 Statement of Problem

The movements of mice are theorized to be correlated with physical, neural, and environmental attributes associated with mice such as strain, health, time-of-day, and day-of-week. The aim of this subproject was to discover whether or not mouse locomotion patterns are unique to each strain. To achieve this objective, we studied the paths the mice took


Mouse 2 consumption rate vs. AS prob.
0.000052
0.000050
0.000048
0.000046
0.000044
0.000042
0.000040
0.34
0.36
0.38
0.40
0.42

throughout the days of the experiment. This involved engineering path features such as length, speed, acceleration, and angle, followed by incorporating visualization techniques to discover previously hidden patterns.

1.3.2 Statement of Statistical Problems

As stated previously, this subproject attempted to discover differences in path patterns that were unique to each mouse strain and each mouse within a strain. The major statistical problem associated with this project was to collapse the data in such a way as to increase our understanding of physical and psychological behavior through visualization of mouse paths. With the Tecott lab's visualization graphics as a launching point (see the image below), we continued on to create features that could be adapted to their plots and also generated new plotting methods to find the optimal method of expression. These features might prove useful in the classification subproject and potentially help us understand the relationship between behavior and genetics in mice and humans.

1.3.3 Exploratory Analysis

The initial exploratory data analysis focused on how to define a path, how to generate metrics from the data, and how to visualize paths at different points in the day. Based on advice from the Tecott lab and our initial data exploration, we separated paths by interruptions in movement that were more than one second long. The path metrics we chose to include were length, speed, acceleration, angles, radius, center angles, area covered, area of the rectangle that contains the path, and absolute distance between the first and last points of a path. With the results of these features, we hoped to gain a better understanding of the more granular differences in paths between strains than with just visualization alone. Having considered these questions, we also realized the need for data cleaning functions to filter out any noise from the data.

The figure below is an example of a day in the life of a mouse in terms of path movements. We see high density movement near the home base, food, and water locations of the cage.

1.3.4 Data

The data required to perform our analysis included the `<x, y, t>` coordinates for the mice as well as a boolean indicating whether the mouse was situated in its home base. Additionally, we required the daily coordinates of the home base for each mouse.

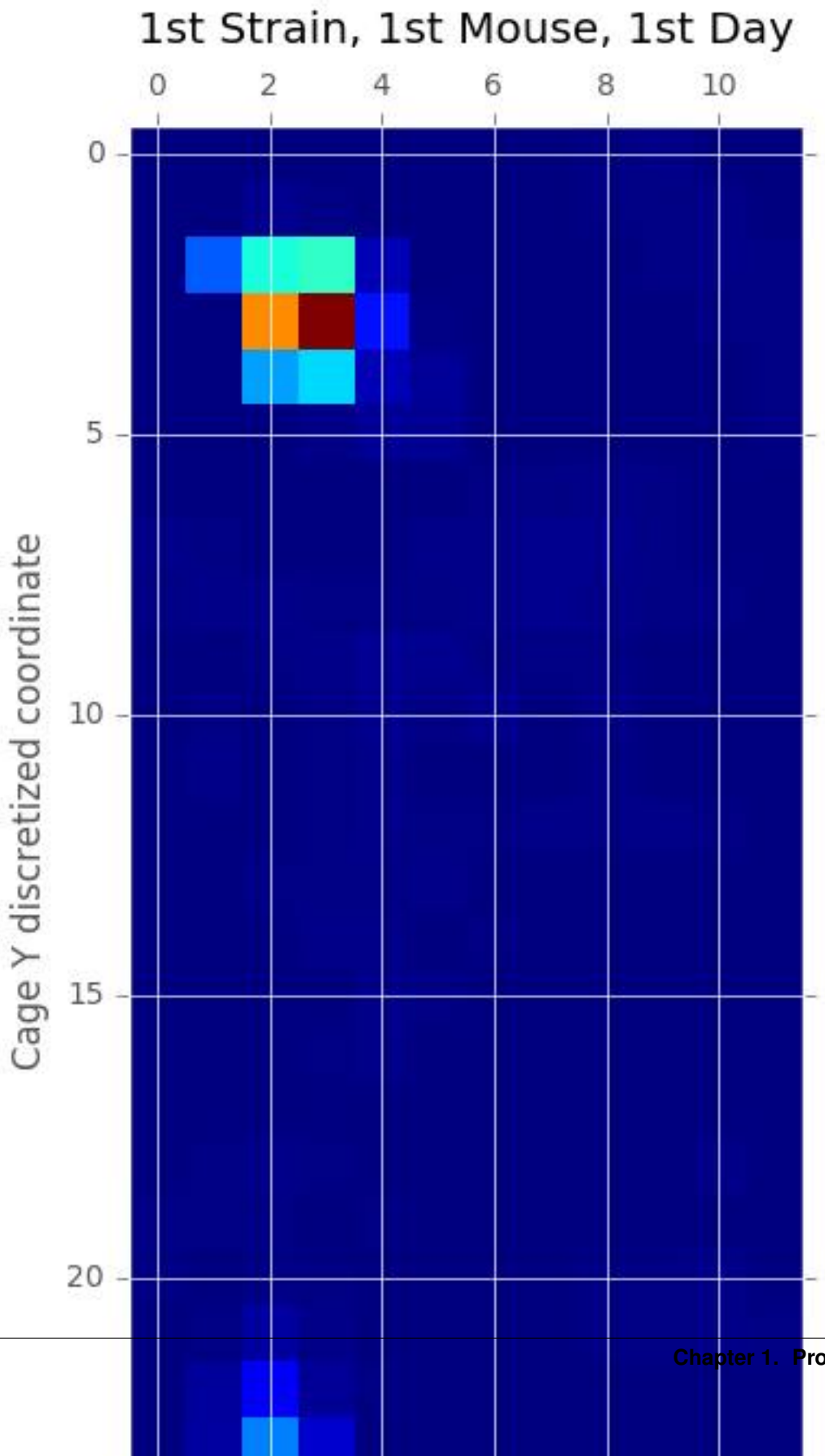
1.3.5 Methodology

Step 1 : Define "Path"

We define a path based on a specific time threshold between movements. In more detail, we created a function that considers that time difference. If the time between movements exceeded the threshold, we considered a new path to have begun. That path was considered to have ended once the threshold had been exceeded again. Based on the advice of the Tecott lab, the default threshold was one second.

Step 2: Clean Data

Based on our definition of a path, we realized that we needed to clean the data to remove obvious outliers and noise. For instance, we found that the sensor platform generated extreme outliers such as an acceleration of -4000 centimeters per second. In order to remove this noise, we created two functions: one to filter the paths and one to remove duplicate rows. The filter paths function uses a minimum number of points in a path to filter out paths that create noise. For example, a path that is only three points could simply be a mouse shifting weight from one foot to the other and back again. Since this is not the type of path we are interested in analyzing, it makes sense to filter them out. The remove duplicates function deals with an issue in the data where the same x and y coordinates appear in adjacent rows but with different timestamps. This caused a problem when computing the angles of the paths so, while it is a trivial removal, it is necessary.


1st Strain, 1st Mouse, 1st Day
0
2
4
6
8
10
0
5
10
15
20
Cage Y discretized coordinate

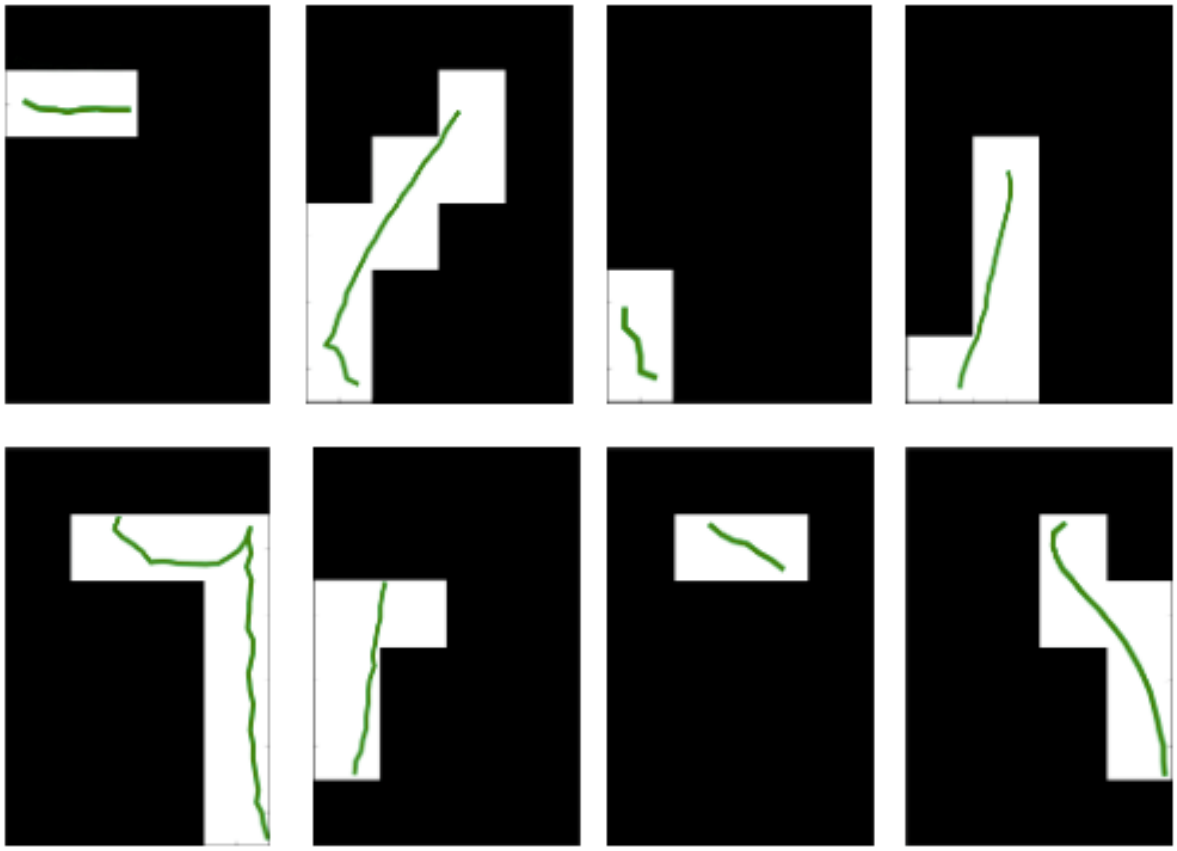
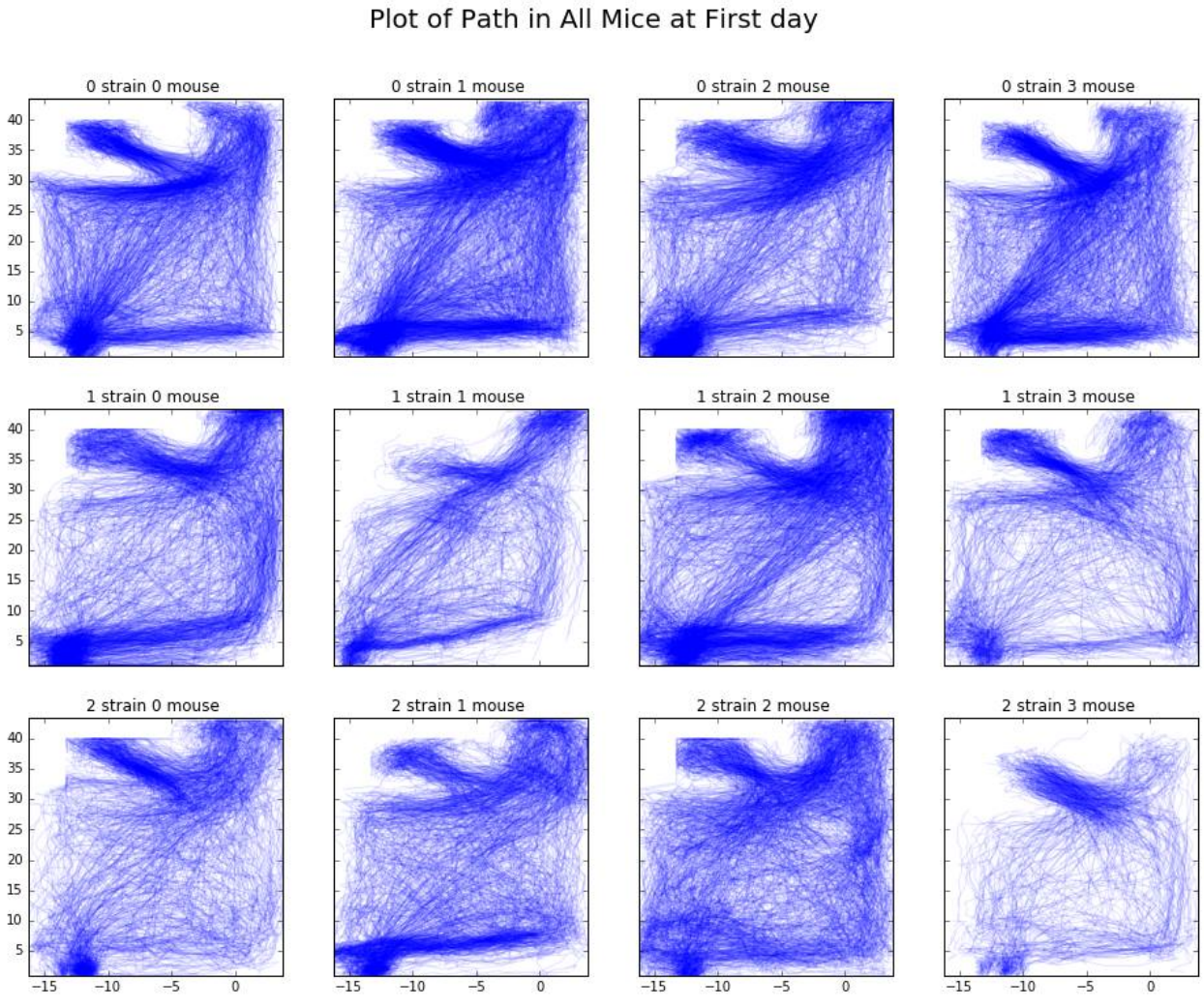


Fig. 1.3: Path (image courtesy of Tecott Lab)


Plot of Path in All Mice at First day
0 strain 0 mouse
0 strain 1 mouse
0 strain 2 mouse
0 strain 3 mouse
1 strain 0 mouse
1 strain 1 mouse
1 strain 2 mouse
1 strain 3 mouse
2 strain 0 mouse
2 strain 1 mouse
2 strain 2 mouse
2 strain 3 mouse

Step 3: Choose Key Features

The path features we wrapped up into functions are as follows:

- Path Length: Total distance covered
- Path Speed: Average speed
- Acceleration: Ratio of speed to the time difference at each point
- Angles: Angle between adjacent vectors
- Radius of Path: Distance between the center point and every other point
- Center Angles: Angle created by two adjacent radius vectors
- Area Covered: Area covered by the path, computed with center angle and radii
- Area of Rectangle: Area of rectangle that contains the entire path
- Absolute Distance: Distance between the first and last points

Step 4: Interpretation

Below are the plots generated based on the features we calculated above, per strain, mouse, and day. With these plots, we can draw some initial conclusions about differences between strains in the results section.

1.3.6 Results

The main findings in this subproject involved the distributions of path distance, average speed, acceleration, and angle. The plots illustrating these distributions can be found below.

In the distribution of path distances plot, we see that Strain 0 appeared to be the most active overall. For all mice in this strain, activeness peaked around days 6-8 and fell off after that. In contrast, there appears to be huge variation in the activeness among mice of Strain 1 with Mouse 2 seeming to exhibit odd behavior compared to the other mice in that strain. Finally, Strain 2 appeared to be the least active strain. However, further inspection is necessary.

From the distribution of average speed plot, we see similar patterns as that of path distance. This is likely due to the correlation between the two features.

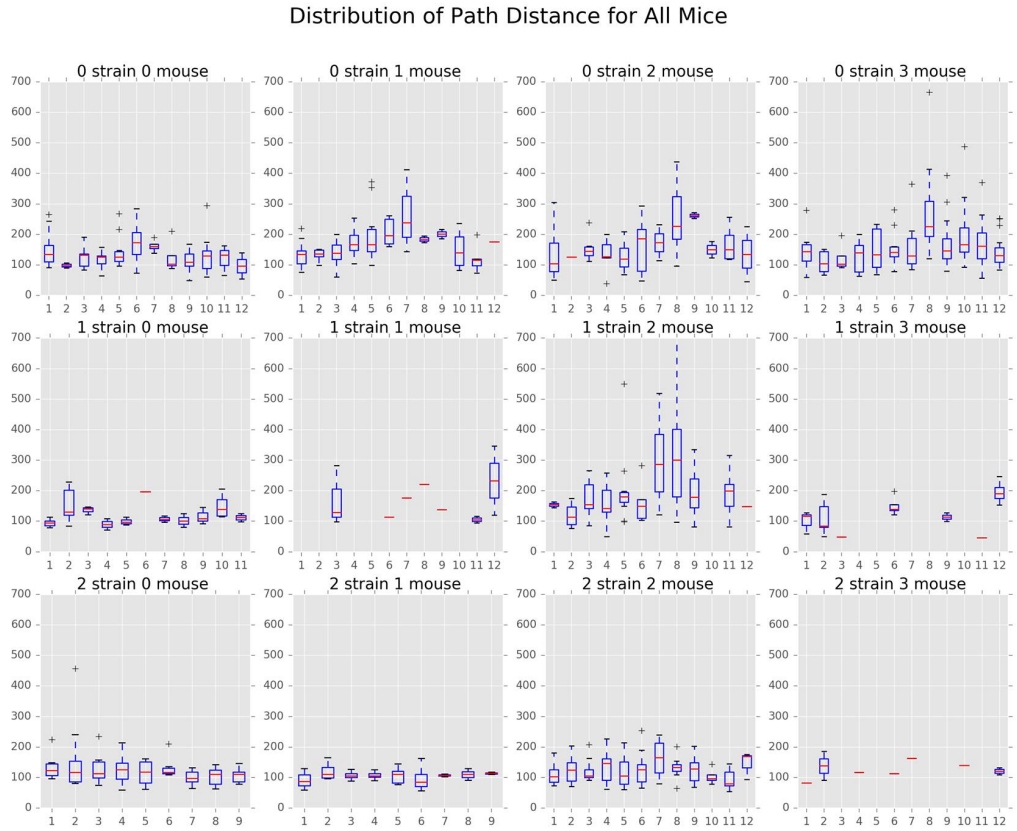
We have also included a distribution of acceleration plot that goes along nicely with the average speed and distance plots. The distributions were centered around 0, symmetrical across the mean. In general, Strain 2 had the least variation while Strain 0 had the most. This implies that Strain 0 reached higher speeds than the other two strains.

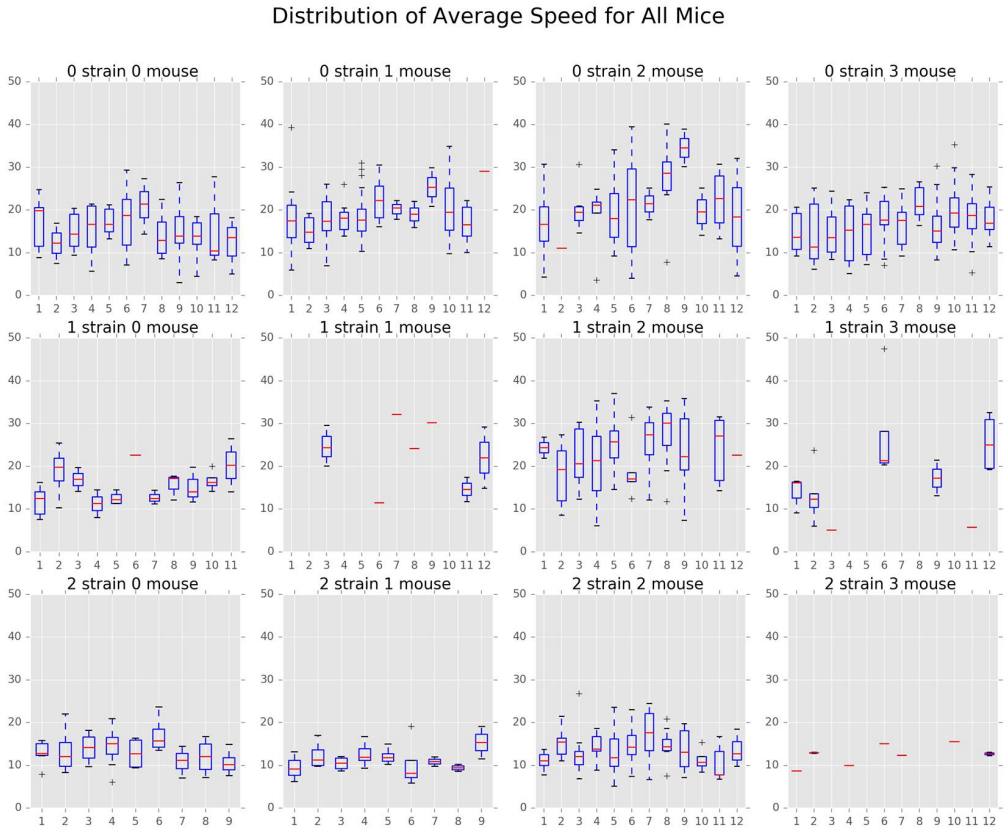
In terms of the distribution of angle, we found that Strain 1 Mouse 1 and 3 along with Strain 2 Mouse 3 had the least sharp turns compared to the other mice. However, this result is likely skewed by the presence of noise in the sensor data.

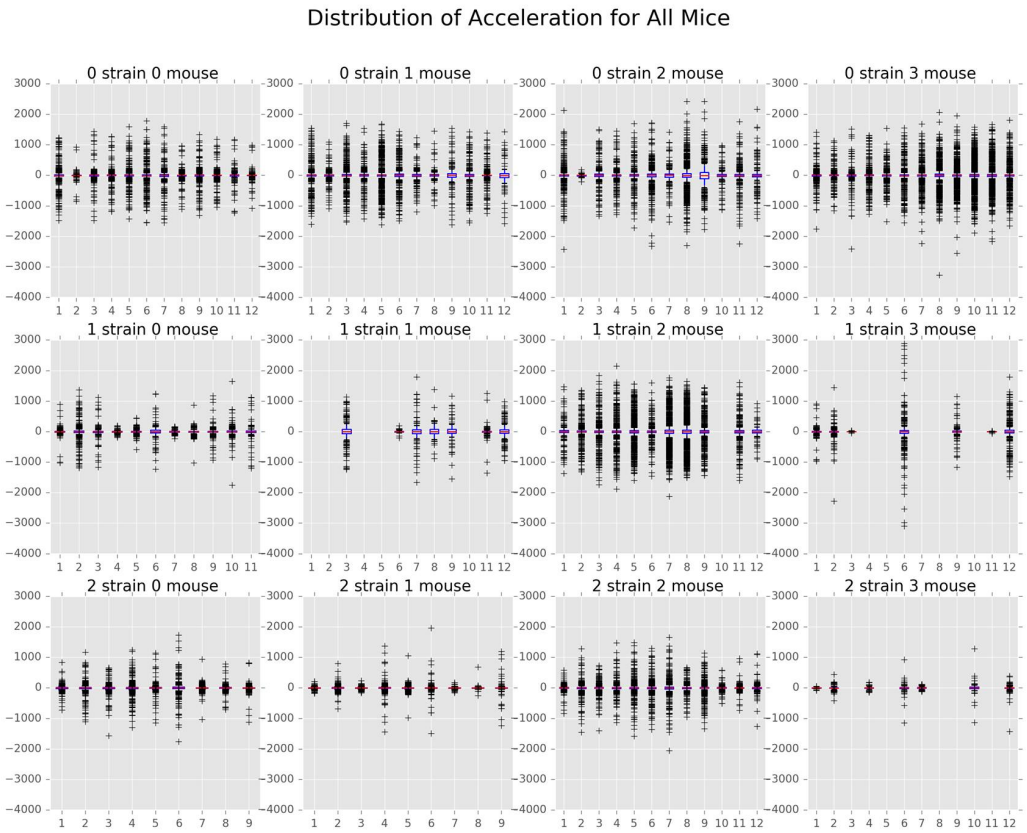
1.3.7 Future Work

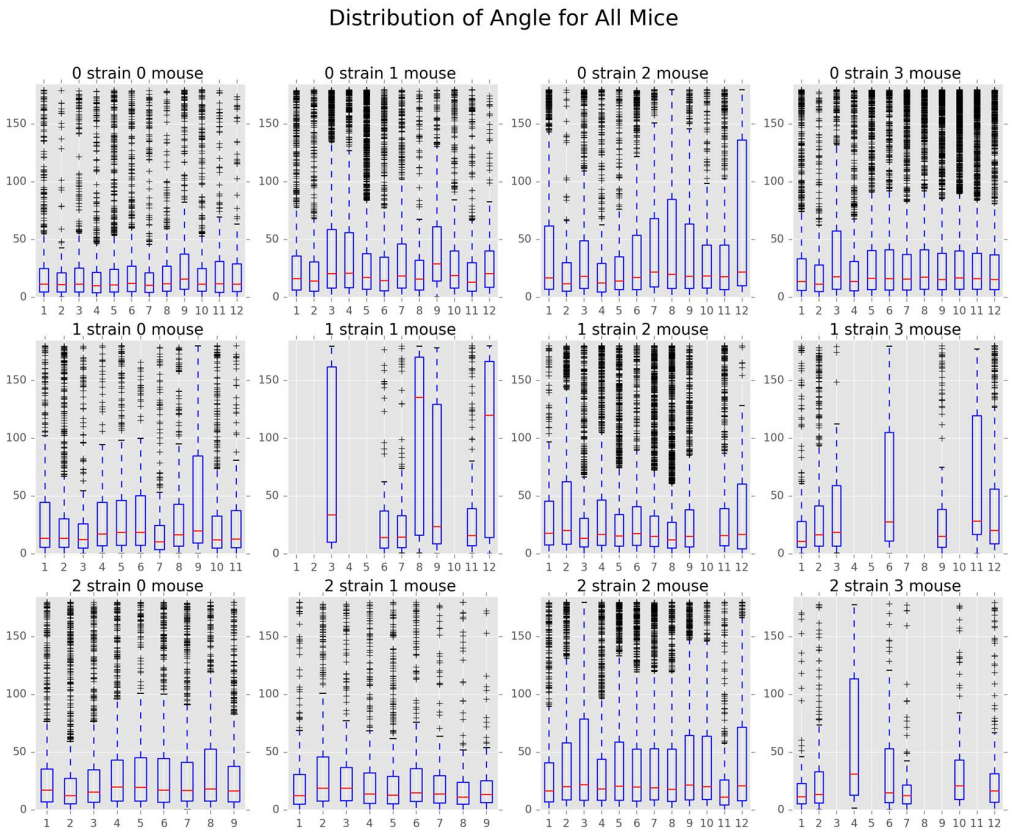
In the future, we would like to explore the following areas:

- Further data smoothing: We would like to further develop a robust methodology to detect points which could be deemed 'measurement noise'. Measurement noise can be defined as consecutive observations which indicate high-turning angles (i.e., above 150 degrees) over very short time intervals. Such observations would skew results on number of sharp turns, average turning angles, etc. It is possible the measurement noise obscures the path behavior differences between strains.
- Path behavior as it relates to active state lifestyle: We would like to explore the influence of a mouse's active state behaviors on its path behavior. A possible question we may be answer is the following: how does hunger


Distribution of Path Distance for All Mice
0 strain 0 mouse
0 strain 1 mouse
0 strain 2 mouse
0 strain 3 mouse
1 strain 0 mouse
1 strain 1 mouse
1 strain 2 mouse
1 strain 3 mouse
2 strain 0 mouse
2 strain 1 mouse
2 strain 2 mouse
2 strain 3 mouse


Distribution of Average Speed for All Mice
0 strain 0 mouse
0 strain 1 mouse
0 strain 2 mouse
0 strain 3 mouse
1 strain 0 mouse
1 strain 1 mouse
1 strain 2 mouse
1 strain 3 mouse
2 strain 0 mouse
2 strain 1 mouse
2 strain 2 mouse
2 strain 3 mouse


Distribution of Acceleration for All Mice
0 strain 0 mouse
0 strain 1 mouse
0 strain 2 mouse
0 strain 3 mouse
1 strain 0 mouse
1 strain 1 mouse
1 strain 2 mouse
1 strain 3 mouse
2 strain 0 mouse
2 strain 1 mouse
2 strain 2 mouse
2 strain 3 mouse


Distribution of Angle for All Mice
0 strain 0 mouse
0 strain 1 mouse
0 strain 2 mouse
0 strain 3 mouse
1 strain 0 mouse
1 strain 1 mouse
1 strain 2 mouse
1 strain 3 mouse
2 strain 0 mouse
2 strain 1 mouse
2 strain 2 mouse
2 strain 3 mouse

and thirst affect the mouse's path diversity? (i.e., does the mouse get sluggish after having eaten and therefore takes shorter, slower paths?)

- Obtaining optimal tuning parameters: We would like to determine which parameters would be optimal for analysis purposes. These parameters include, but are not limited to, the cutoff time for a path, the minimum path length (in observations) and turning angle threshold for noise detection.
- Use path features for classification: We would like to use the features calculated within the functions and apply machine learning techniques to determine whether path features (speed, turning angles, distance traveled, etc.) can be used for classification.

1.3.8 Additional Remarks

While working on this subproject, we came across several questions that required clarification from the lab. Researchers should be cautious regarding these points in future studies.

First, we noted that the locomotive observations of the mice were recorded at unevenly spaced intervals (i.e., delta-t varies from point to point). In our exploratory data analysis, we assumed that observations were recorded whenever the mouse was in motion, and during large delta-t intervals, we assumed the mouse was stationary. This is an important point that should be taken into account before moving forward with any analysis.

According to the authors, a mouse 'movement event' was numbered in the tens of thousands per day. Each event was described by a location and time stamp when the distance from the prior recorded location exceeded 1 cm. Despite this, we noted an instance in the data where the coordinates from (t) to (t+1) did not change, but resulted in a new observation. This was accounted for in the data cleaning process.

1.3.9 Reference Reading

- Spatial memory: the part of memory that is responsible for recording information about one's environment and its spatial orientation
- Wikipedia
- Mouse Cognition-Related Behavior in the Open-Field: Emergence of Places of Attraction

1.4 Dynamics of AS Patterns

1.4.1 Introduction

The objective of Dynamics Analysis is to analyze and characterize the state transforming behaviors of different strains of mice. The analysis is mainly focused on the three strains of mice included in the interval dataset. Using a Markov Chain Model, a sequence of state transition probability matrix is retrieved for each strain depending on the time intervals. A simulation mouse is then generated based on the model and an evaluated. The evaluation system is created using the time interval that produces the most realistic simulation mouse. Further visualization of the fake mice is done to explore the discrepancy between different strain's behaviors.

1.4.2 Data

We are using data from intervals directory for estimating the dynamics pattern.

1.4.3 Methodology

1. Defining states of interest:

- Feeding: labeled by event F
- Drinking: labeled by event W
- Other active state behaviors: This could possibly include all other movements in the AS state of a mouse besides drinking and eating.
- Inactive State: labeled by event I.

2. Data Preprocessing:

We first transform data from intervals directory to a time matrix. The rows of the matrix indicate a specific strain, specific mouse and specific day. The columns of the matrix represent the 24 hours in a day. The content of the matrix indicates the state of a specific strain, specific mouse and specific day in the column time. Here we define 0 as IS, 1 as F, 2 as W, and 3 as Others. Note that there are two important preprocessing methods we do to create such matrix:

- Combining small intervals: Some of the intervals, especially drinking behavior intervals are significantly small, say 0.001 from the beginning to the end for an interval, which does not make sense. So we combined these intervals if the next interval start time minus the last interval stop time is smaller than a threshold. We are using 4 seconds as the default threshold.
- Choosing the gap between columns: After combining small intervals, all the current behavior intervals are at least 4 seconds long. Now we want to set a gap between columns so that we can divide 24 hours equally and map the small columns time into the behavior intervals to determine the state at that column time. We are using 1 second as the default gap.

To create this matrix, one can do:

```
from mousestyles.dynamics import create_time_matrix
data_df = create_time_matrix()
```

Or, simply import the matrix from data directory:

```
from mousestyles.data import load_time_matrix_dynamics
data_df = load_time_matrix_dynamics()
```

3. Estimating Transition Probability:

Estimate the transition probability matrix of the Markov Chain using the data given. One of the key challenges to estimate the transition matrix is that the model used is actually time continuous non-homogeneous Markov Chain, and the parameters are too difficult to estimate given the current data. Therefore, a new way is discovered to make the model a composite of small homogeneous discrete time Markov Chains, so that performing a rough estimation of the original time continuous non-homogeneous Markov Chain is possible. The following steps are followed to get the transistion probability matrix:

- Divide each mouse day into small time intervals, say 5 minutes. (The time interval is optimized in the future analysis)
- For each of the small time intervals, aggregate the data from all mice in the same strain for all mouse days and estimate the transition probability matrix of a discrete homogeneous Markov Chain model just for this small time interval.
- each of these transition probability matrices is estimated by MLE method, where e.g.:

$$P(F_{t+1}|W_t) = \frac{N_{WF}}{N_{W.}}$$

where N_{WF} indicates the counts of transitions from W to F and $N_{W.}$ indicates the counts of transitions starting from W, no matter where it ends.

- Build the whole model by compositing the models for each small time intervals.

```
strain_df = data_df[data_df.strain == 2]
get_prob_matrix_list(time_df=strain_df,interval_length=1000)
# [array([[ 1.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]]),
# array([[  9.99854291e-01,   0.00000000e+00,   4.85696246e-05,
           9.71392491e-05],
        [  0.00000000e+00,   9.10714286e-01,   0.00000000e+00,
           8.92857143e-02],
        [  0.00000000e+00,   0.00000000e+00,   5.00000000e-01,
           5.00000000e-01],
        [  0.00000000e+00,   1.64835165e-02,   0.00000000e+00,
           9.83516484e-01]]),
  ...
  ...
```

4. Simluation:

Use the Transition Model generated to simluate a typical mouse day for a typical strain. The simulation is time sensitive which means the simulation mouse is generated depending on the time interval chosen in the transition model. The time interval length is optimized in the future analysis so as the generate the best fake mouse for each strain.

```
trans_matrix = get_prob_matrix_list(time_df=strain_df,interval_length=1000)
mcmc_simulation(trans_matrix, n_per_int=1000)
# array([0, 0, 0, ..., 0, 0, 0])
```

5. Evaluation System:

How to get a score for the performance of our simulation? The evaluation system is trying to create a score to evaluate the performance of the simulation. Higher the score, better the simulation is doing in catching the pattern in the mouse day. Because we have unbalanced data, meaning most of the statuses in one typical mouse day are IS and OTHERS and only a few of them are DRINK and EAT, we should give more weights on correct predictions about DRINK and EAT. We first calculate the proportion of the four status within different days we have for different mice from different strains and then take average to get the ratio of different status. And then we choose the initial weights based on the numbers we get. For example, a mouse day has 21200 timestamps on average. And on average 10000 of them are IS, 1000 are EAT, 200 are DRINK, and the left 10000 are OTHERS. The ratio is 10000:1000:200:10000 = 1:0.1:0.02:0.1. Thus, the weights should be the inverse, 1:10:50:1.

But of course, users can define their own initial weights to fit in their purpose. If the user prefers equal weights, he/she can simply set the weight to be (1,1,1,1).

The final score is calculated as the number of correct predictions times the weights divided by the number of status. For example, if our observed data is "IS IS DRK DRK EAT OTHERS", and our predicted value based on it is "IS IS IS DRK EAT OTHERS", the score would be (1+1+0+50+10+1)/6 = 10.5. But because our data contains a lot more IS and OTHERS, the score ranges from 0 to 1 for our data.

6. Comparison:

The comparison function plots the simulated behavior dynamics obtained from the evaluation function for the given strain during the given time period.

The x-axis is the time range, and the default time range is from 36,000 to 36,100. It is because the behavior dynamics show clear differences across the strains during this time range and because the behavior dynamics have more active actions than the other time period. Users can, of course, define the time range of their interest. But they should be aware of the lower and upper bounds of the time range before doing comparisons. For the first strain (strain_num =

0), the time range should be between 1 and 92,400. For the second one (strain_num = 1), it should be between 1 and 90,000. The time range should be between 1 and 88,800 for the last strain (strain_num = 2).

Of note is that, based on our understanding and explanations from the research team, 1 indicates the time when a mouse wakes up and starts its day. The unit of time is understood as one second. Thus, the default time range (from 36,000 to 36,100) represents around 10 hours after the start of the day. Based on this definition of the time range, users can change it and compare simulated behavior dynamics across the three strains.

We do not define the y-axis for this comparison function, because our interest is to visually understand the dynamics of the four states during the given time period. For better understanding of the behavior dynamics, we assign different colors for different states. In the plot, blue represents IS, bright green represents eating, yellow represents drinking, and red represents other activities in AS.

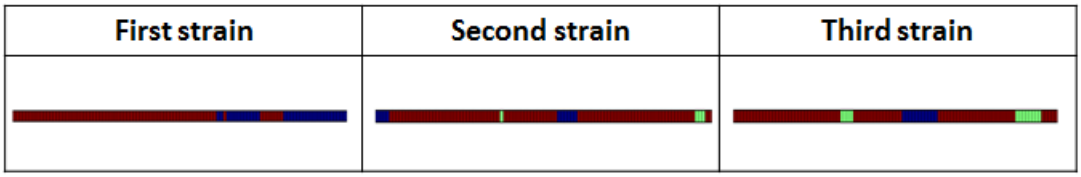
1.4.4 Result

The problem we are insterested in here is whether the three strains of mice are indeed acting differently in a time series manner. The behaviors are compared using the three simulation mice, each for one strain. Therefore, as the first step, the optimal time interval is selected so as to have the most-real simulation mouse that behaves the most similarly to its strain, which is evaluated using the score system we created. As a result, the best time interval length is selected for each strain as well as the corresponding simulation mice behavior and the comparison score.

```
find_best_interval(data_df,strain=0)
# (600, array([0, 0, 0, ..., 0, 0, 0]), 0.70509736459572225)
find_best_interval(data_df,strain=1)
# (600, array([0, 0, 0, ..., 0, 0, 0]), 0.91423472578532516)
find_best_interval(data_df,strain=2)
# (1800, array([0, 0, 0, ..., 0, 0, 0]), 0.83975976073161329)
```

As the script suggested, the best time interval selected is 600 seconds for the first strain. This generates a simulation mouse that is 71% similar to the real mice in that strain. Note that the simulation behavior is quoted in the numpy array format. Similarly, it shows the 600s optimal time interval with 91% similarity for the second strain and the 1800s optimal time interval with 84% similarity for the third strain.

With the resulting best simulation mice, we move forward to compare the between-strain behaviors using visulization. The simulated mice show different behavior dynamics across the strains, during the given time period (36,000 to 36,100; around 10 hours after the start of the day). The fake mouse from the first strain does the other active actions and stays inactive for a short period of time. The fake mice from the other two strains, on the other hand, show different dynamic patterns: do the other active actions, consume food, and stay inactive. The fake mice from the third strain consumes food for the longer time for the given time period.



However, if we let the plot time range from 1 to 100, all of the three types of fake mice stay inactive. It is intuitive under the daily lives of humans. They are likely to stay home and rest right after getting up. Users can change the plot time range and compare the three mice, which helps understand any differences among the three strains over time.

Users can also follow the example codes inside at mousestyles/doc/sources/report/plots - plot_dynamics0.py, plot_dynamics1.py, and plot_dynamics2.py - for more details.

First strain	Second strain	Third strain

```
from mousestyles.data import load_time_matrix_dynamics
from mousestyles.visualization.dynamics import plot_dynamics
time_matrix = load_time_matrix_dynamics()
plot_dynamics(time_matrix, 0)
```

1.4.5 Discussion

We have written the functions and tried understanding dynamic behaviors of mice. There are three issues for further analysis. First, we can apply more statistical approach to detect the discrepancy of behaviors across the strains. Second, larger sample size of mice and longer time period can help understand behavior dynamics more accurately. Third, the inactive states and the other active states can be specified, which can help detect more detailed dynamics of mice behaviors.

1.4.6 References Reading

http://scikit-learn.sourceforge.net/stable/modules/hmm.html

https://github.com/hmmlearn/hmmlearn

https://en.wikipedia.org/wiki/Hidden_Markov_model

1.4.7 Contribution

- Data Preprocessing: Hongfei
- Modeling: Jianglong
- Simlation: Chenyu
- Score: Weiyan
- Evaluation: Luyun Zhao
- Comparison: Mingyung

1.5 Ultradian and Circadian Analysis

1.5.1 Statement of Problem

Ultradian and circadian rhythm are widely observed in mammalian behavioral patterns. The ultradian analysis aims to find the time-specific patterns in behavioral records, without specifying the length of the cycle in advance (but need to be within 1 hour to 1 day). The typical ultradian period includes 8 and 12 hours. The circadian rhythm refers to a roughly 24-hour cycle. For example, we expect the rats to be inactive in the nighttime and ingestions and movements mostly happened in the daytime.

The variables of interests are the summary statistics of mouse activity including food (F) and water (W) ingestion, distance traveled (D), Active State probability (AS), movement inside the home base (M_IS) and movement outside the home base (M_AS). We may also consider spatial variables, for example, we can spatially discretize the data to cells containing the primary functions, like a food cell or a water cell, and examine the ultradian cycle of the spatial probability densities of the occupancy time in each cell.

In this report, we assume strain to be the primary influence on the variation of ultradian rhythms across individual mice and examine the difference in rhythms across strains. Currently, we lack basic information such as weight, age, and gender. With more data available in the future, we may look into the cycle for each mouse and detect the most important factors influencing the ultradian rhythms.

The ultradian and circadian analysis is closely related to other subprojects. Ultradian rhythms could be treated as one feature for clustering the 16 strains. We may also subset the data using the results of the cluster and analyze the rhythm similarities and differences across clusters.

1.5.2 Statement of Statistical Problem

Our statistical problems are three parts: data preparation, choice of the frequency or period, and modeling of rhythm patterns.

- Data preparation:

  Mouse behaviors are recorded based on time intervals or time points, like the beginning and ending time stamp of one food ingestion or the coordinates of mouse position at a particular time point. We aggregate the data based on given time bins and thus convert the raw data to time series. Then how to determine the optimal bin intervals for constructing the time series? Bin interval examples include 5 min, 30 min, 1 hour, etc.

- Choice of the frequency or period:

  For ultradian rhythms, the significant period length may vary according to the variables of interests. The Lomb-Scargle (LS) periodogram spectral analysis technique, a widely used tool in period detection and frequency analysis, is applied.

- Modeling of rhythm patterns:

  We are not only interested in the period length of circadian and ultradian rhythms, but also the variation of features over time within the discovered circles. To model the time trajectory of features, we propose to use seasonal decomposition and mixed effects model.

1.5.3 Data

In order to retrieve the data needed for analyzing mouse ultradian behaviors, we get input from users, including features (water, food, active state, etc.), strain number, mouse number and time bin width. Also, we have another function called “aggregate movements” to track mice’s movements intensity based on time bins. They enable us to specify which mouse and which kind of features to study. After getting bin width from users, we allocate the time used by each feature into selected time intervals. For example, if “30 minutes” is selected to be bin width and “food” is selected to be the feature, then the eating time intervals are separated are relocated into 30-minute long bins. For one mouse day, using 30-minute time bin gives 48 records as a result in total.

For eating and drinking behavior, instead of using time consumption as intensity, we choose to use food or water consumption amount by acquiring food/water consuming data, and assuming the food/water expended during feeding is proportional to the time used. For calculating the movements data, we figured out the position change by generating Euclidean distances first and then distributing them into different time bins.

The output is a pandas time series, with time index representing time intervals and values including feature data (such as food/water consumption, active state time as well as movement distance).

- **Input:** records for each strain (total of 16), each feature of interest (food, water, distance, active_state probability, ...), in a duration of 12 days (excluding 4 acclimation days).
- **Processed:** using one-minute time bins of movement records to binary score the activity into 0 (IS: inactive state) and 1 (AS: active state); using thirty-minute bins of food records to calculate the amount of chows consumed by mice; using LS periodogram technique to select the appropriate time bins for above.
- **Output:** different patterned visualization for each feature, with the appropriate time bins that present the most significant ultradian pattern.

1.5.4 Methodology

Seasonal decomposition

Seasonal decomposition is a very common method used in time series analysis. One of the main objectives for a decomposition is to estimate seasonal effects that can be used to create and present seasonally adjusted values.

Two basic structures are commonly used:

```
1. Additive:  x_t = Trend + Seasonal + Random

2. Multiplicative:  x_t= Trend * Seasonal * Random
```

The "Random" term is often called "Irregular" in software for decompositions.

Basic steps:

```
1. Estimate the trend

2. "De-trend" the data

3. Estimate seasonal factors by using the "de-trended" series

4. Determine the "random" term
```

The following plot shows the seasonal variation of AS using circadian period. The difference between strains in rhythm patterns is evident.

Seasonal variation of AS probability (circadian, 24 hours). There appear to be significant patterns within each group. Strain0 tend to have two active stages every day, while strain1 will only have one active stage that lasts about 5 hours. The pattern in strain3 is not as strong as other strains, for that different mice seem to have larger variations. Nevertheless, the behavior with periods of 24 hours can be more regularly distributed than other periods showed below.

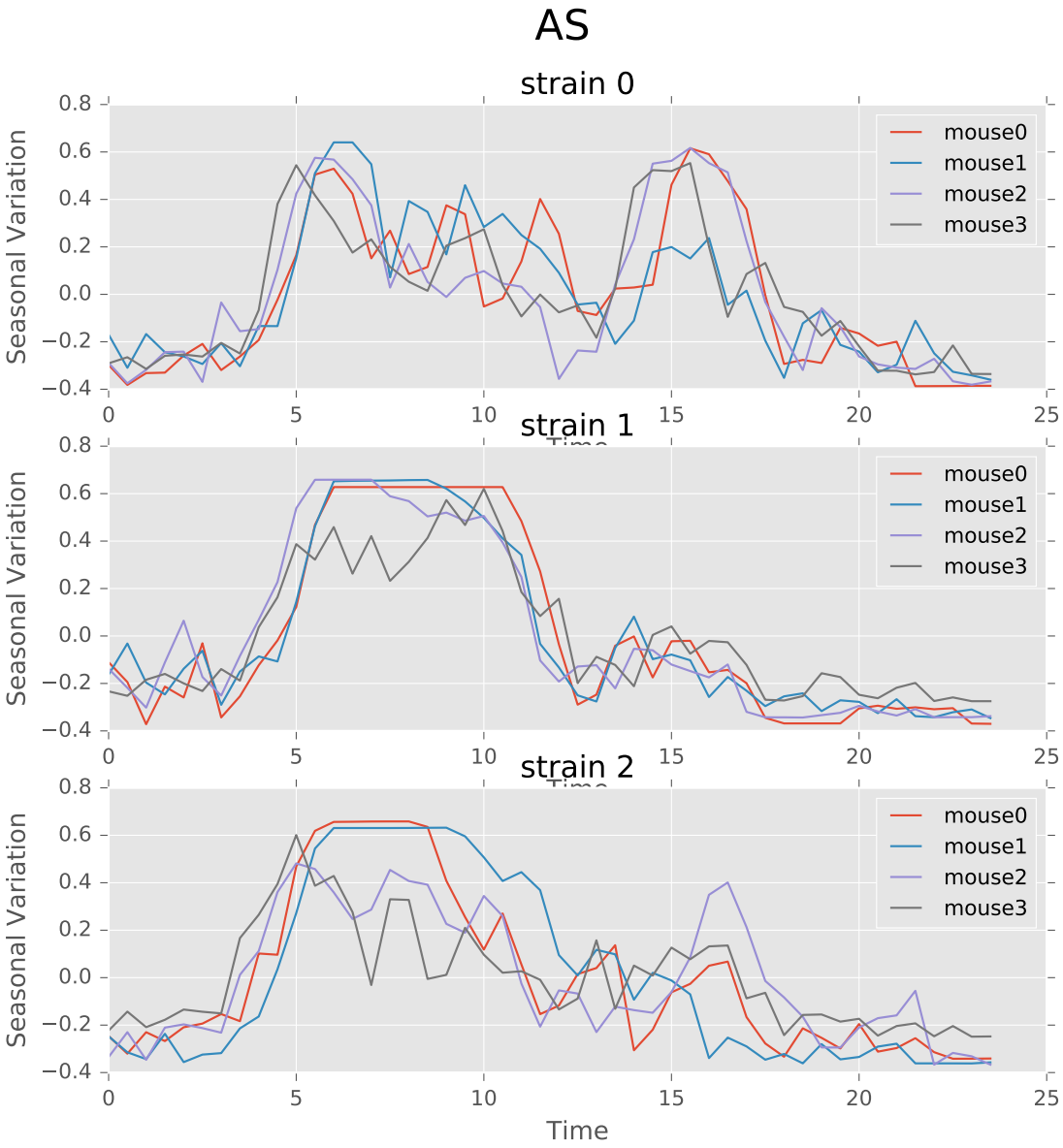
Lomb-Scargle Period Test

Similar to Fourier analysis, the Lomb-Scargle periodogram is a common tool in the frequency analysis of unequally spaced data equivalent to least-squares fitting of sine waves. Basically, we want to fit sine waves of the form:

$$y = a\cos\omega t + b\sin\omega t$$

While standard fitting procedures require the solution of a set of linear equations for each sampled frequency, the Lomb-Scargle method provides an analytic solution and is therefore both convenient to use and efficient. In this case, we want to test whether each mouse/strain has a significant cycle less than 24 hours.

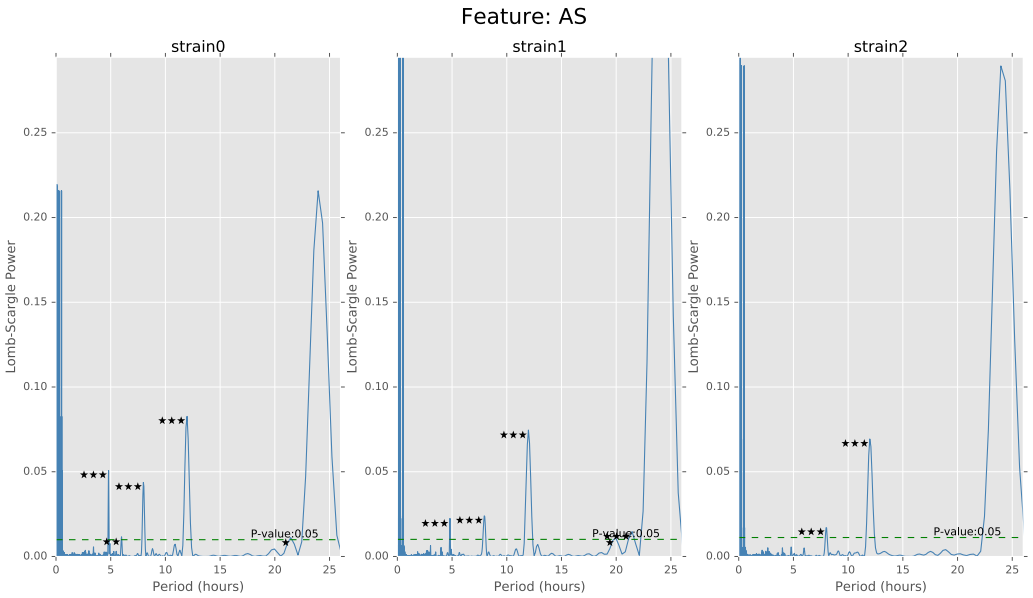
For the mouse behavior data, we use Lomb-Scargle method on different strain and mouse's data to find the best possible periods with highest p-values. The function can be used on specific strains and specific mice, as well as just certain strains without specifying mouse number. We use the $O(N \log N)$ fast implementation of Lomb-Scargle from


AS
strain 0
strain 1
strain 2
Seasonal Variation
Time
mouse0
mouse1
mouse2
mouse3
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
5
10
15
20
25

the gatspy package, but the LS power around 0 period is a little bit noisy. The other algorithm can give smooth results around 0 point but suffer $O(N^2)$ time complexity. Also, we need to add small uniformly distributed noise on the regularly sampled time sequence to avoid singular matrix problems.

The function can give the LS power as well as the P values for the corresponding periods, with respect to the time bins chosen to combine the data. There will also be stars and horizontal lines indicating the p-values of significance. Three stars will be p-values in [0,0.001], two stars will be p-values in [0.001,0.01], one star will be p-values in [0.01,0.05]. The horizontal line is the LS power that has p-value of 0.05.

Below are the ultradian analysis results found by combining seasonal decomposition with best periods returned by Lomb Scargle periodogram. Here we use features "AS" (active state probability) and "M_IS" (movement time inside home base) as two examples, because other features like food, water, movement distance, movement time outside home base all have similar LS plot to "AS" and we show them in Appendix. "M_IS" shows a rather different pattern.



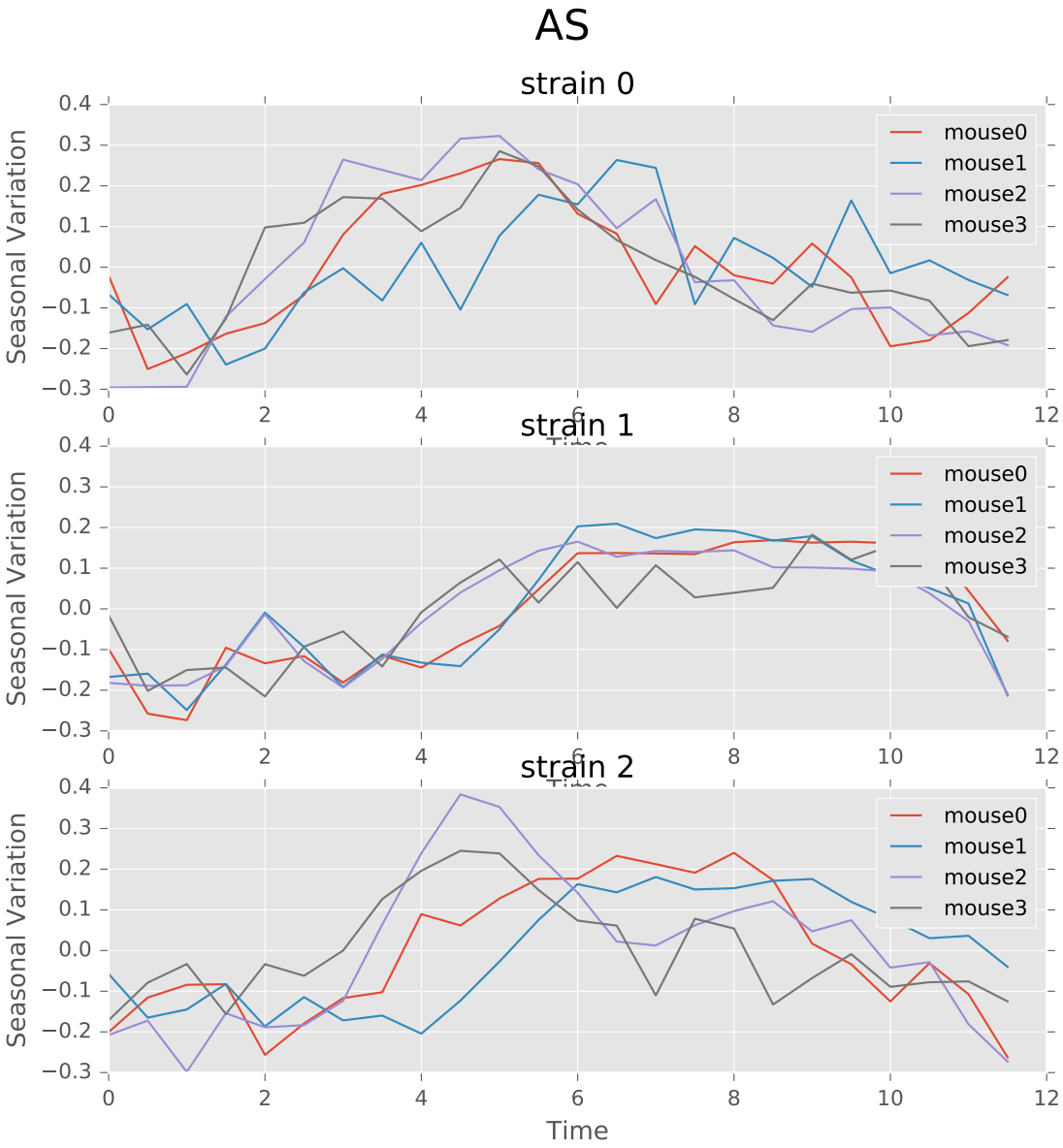
Longitudinal data analysis

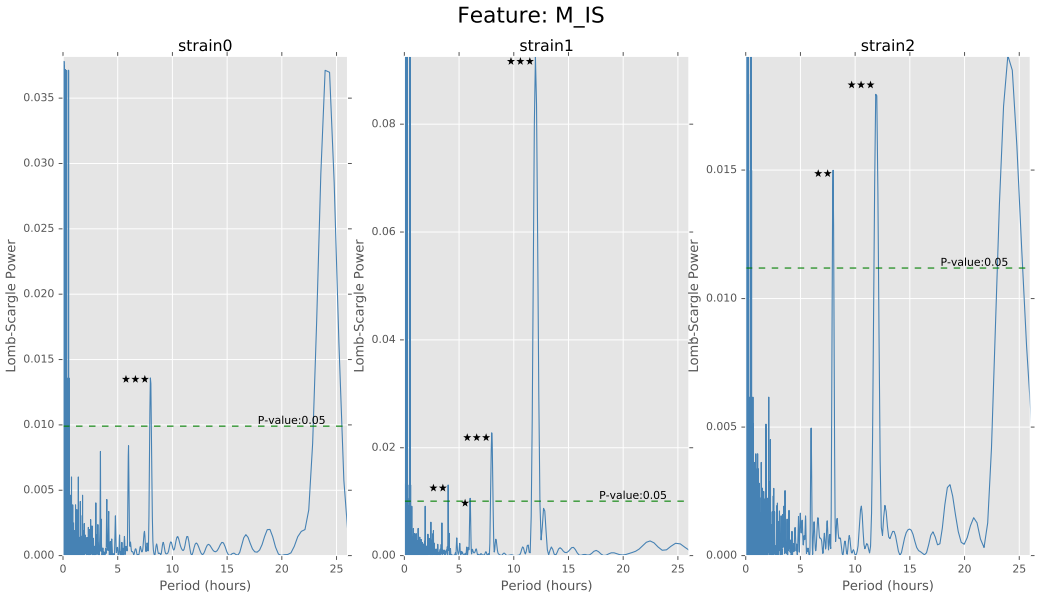
- Attempts for mixed models

  The mixed model is frequently used for longitudinal analysis. We should specify the random effects and fixed effects first. Since it is ultradian analysis, we only need to focus on the hour factor and their cycle which we can get from the previous LS test. The random effect is the mouse id. We have four different mice in one strain and only want to compare the different patterns among these three strains. So if we set the random effect to be mouse id, the effects from different mouses will be cancelled out and we can also test the significance of these effects. The response variable will be one of the six features listed before. After that we can use the mixed model to get the pattern of the movements in different time period.

- Build the model

  Take *Food* feature as an example. Here strain0 is a dummy variable indicating whether the mouse belongs to strain 0 or not and similarly for strain1. strain0hour and strain1hour indicate the interaction terms, which we add in order to figure out whether the strain and hour have some interaction effect in the Food feature.(*i denote ith strain, j denote the jth mouse*)


AS
strain 0
strain 1
strain 2
Seasonal Variation
Time
mouse0
mouse1
mouse2
mouse3
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0
2
4
6
8
10
12



$$Food_{ij} = f(strain0_{ij}, strain1_{ij}, hour_{ij}, cycle_{ij}) + interactions + \beta_j mouse$$

- Perform significance test

  Here we have two purposes. The first is to figure out if the effects from different mouses are significant. The second is to figure out if the patterns for different strains are significantly different. To test the first one, we just need to use the t-test and get the p-value from the result by using the *statsmodels.formula.api* package. package. For the second one, we can perform the likelihood ratio test on the interaction terms.

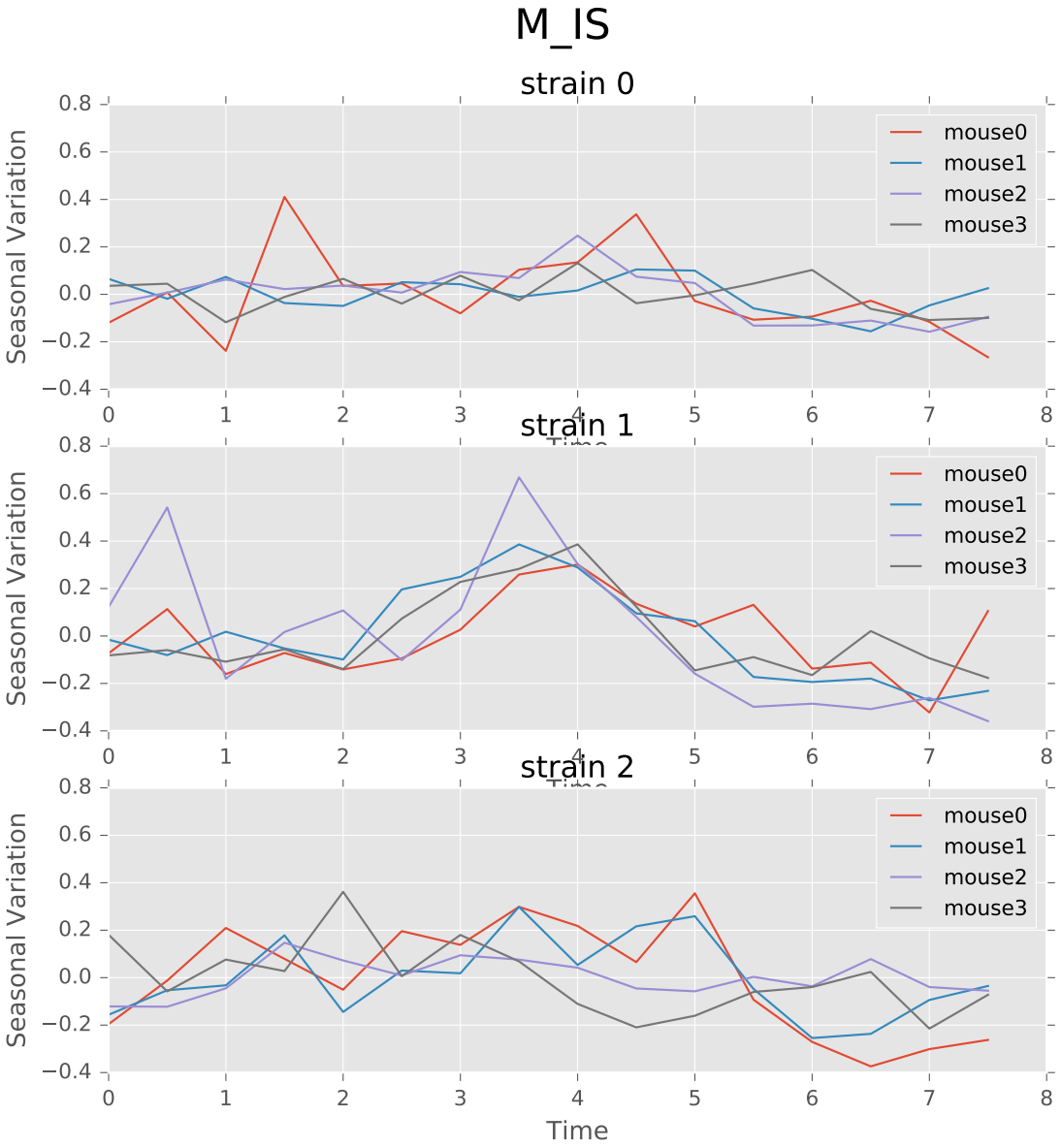
First, we look at the summary of the full model (also take the Food feature as an example). To get this result, we fit the second-degree function. Since the cycles from the previous study are very similar between strains, we did not include it here. We can see that the effects of the mouse from the same strain are not significant. However, the p-value here seems to indicate that the interaction terms is not as significant as the other factors. So we consider conducting the likelihood ratio test.

factors	Coef.	Std.Err.	z	P>|z|
Intercept	0.049	0.008	6.095	0.000
hour	-0.005	0.001	14.961	0.000
hour2	-0.001	0.000	-18.767	0.000
strain0	-0.027	0.010	-2.624	0.009
strain1	0.045	0.010	4.332	0.000
strain0:hour	-0.002	0.002	-0.778	0.437
strain1:hour	-0.004	0.000	-1.76	0.078
strain0:hour2	0.000	0.000	2.019	0.043
strain1:hour2	0.000	0.000	0.540	0.589
RE	0.000	0.000		

Secondly we did likelihood ratio test between the two models: full model and reduced interaction terms model. We found that the p values for 6 features below:

Water	Food	AS	M_AS	M_IS	Distance
3.08e-9	2.50e-9	9.39e-12	5.11e-5	0.002	1.53e-8

We can see that the Water, Food, AS, M_AS, Distance have significantly different patterns of different strains.


M_IS
strain 0
strain 1
strain 2
Seasonal Variation
Time
mouse0
mouse1
mouse2
mouse3

1.5.5 Testing Framework Outline

Step 1: Generating random samples for testing:

- Split the data based on the Mouse Day Cycle
- Number the splits and use numpy.random to subset from these splits

Step 2: Conduct Lomb-Scargle (LS) test to detect the period. Implement the three different models onto the certain period and get the patterns/ estimated coefficients for the model.

Step 3: Compare the result with our hypothesis.

1.5.6 Appendix

1.6 Classification and Clustering of Mice

1.6.1 Statement of Overall Problem

The mouse style research investigates mouse behaviors of different genes. The researchers hope to gain insight to human behaviors using mouse data, since experimenting directly on human is difficult.

The important concern is whether we can classify different strains of mice based on mouse behaviors through machine learning algorithms. Important features in classification models will be extracted for future study. The fundamental hypothesis is that different mouse strains have different behaviors, and we are able to use these behaviors as features to classify mice successfully. On the other hand, we are also trying to use unsupervised clustering algorithms to cluster mice, and ideally different strains should have different clustering distributions.

1.6.2 Statement of Statistical Problems

The researchers design the experiment as follows: they have 16 different strains of mice, and each strain has 9 to 12 almost identical male mice in terms of genes.

We first need to verify the assumption that mouse of different strains do exhibit different behaviors. One way to do this is to perform hypothesis testing based on the joint distribution of all the features. A simpler alternative is to perform EDA on each of the features. If we observe that each mouse behaves very similarly to its twins, but differently from other strains of mice, we can conclude that genetic differences affect mouse behaviors and that behavioral features might be important for classification. Notice that here we can evaluate the difference by assessing the classification performance of models based on behavioral features.

Classification

Based on the assumption, the problem is inherently a multiclass classification problem based on behavioral features either directly obtained during the experiment or artificially constructed. Here we have to determine the feature space both from the exploratory analysis and biological knowledge. If the classification models performed well, we may conclude that behavioral differences indeed reveal genetic differences, and dig into the most important features needed seeking the biological explanations. Otherwise, say if the model fails to distinguish two different strains of mice, we may study that whether those strains of mice are genetically similar or the behavior features we selected are actually homogeneous through different strains of mice.

Clustering

To extend the scope of the analysis, the clustering analysis may also be used to analyze mouse behaviors. Though these mice had inherent strain labels, the clusters may not follow the strain labels because genetical differences are not fully capturing the behavioral differences. Moreover, determining the best number of clusters is key in assessing the performance of a clustering model. Notice that the best number of clusters are neither necessarily equal to 16

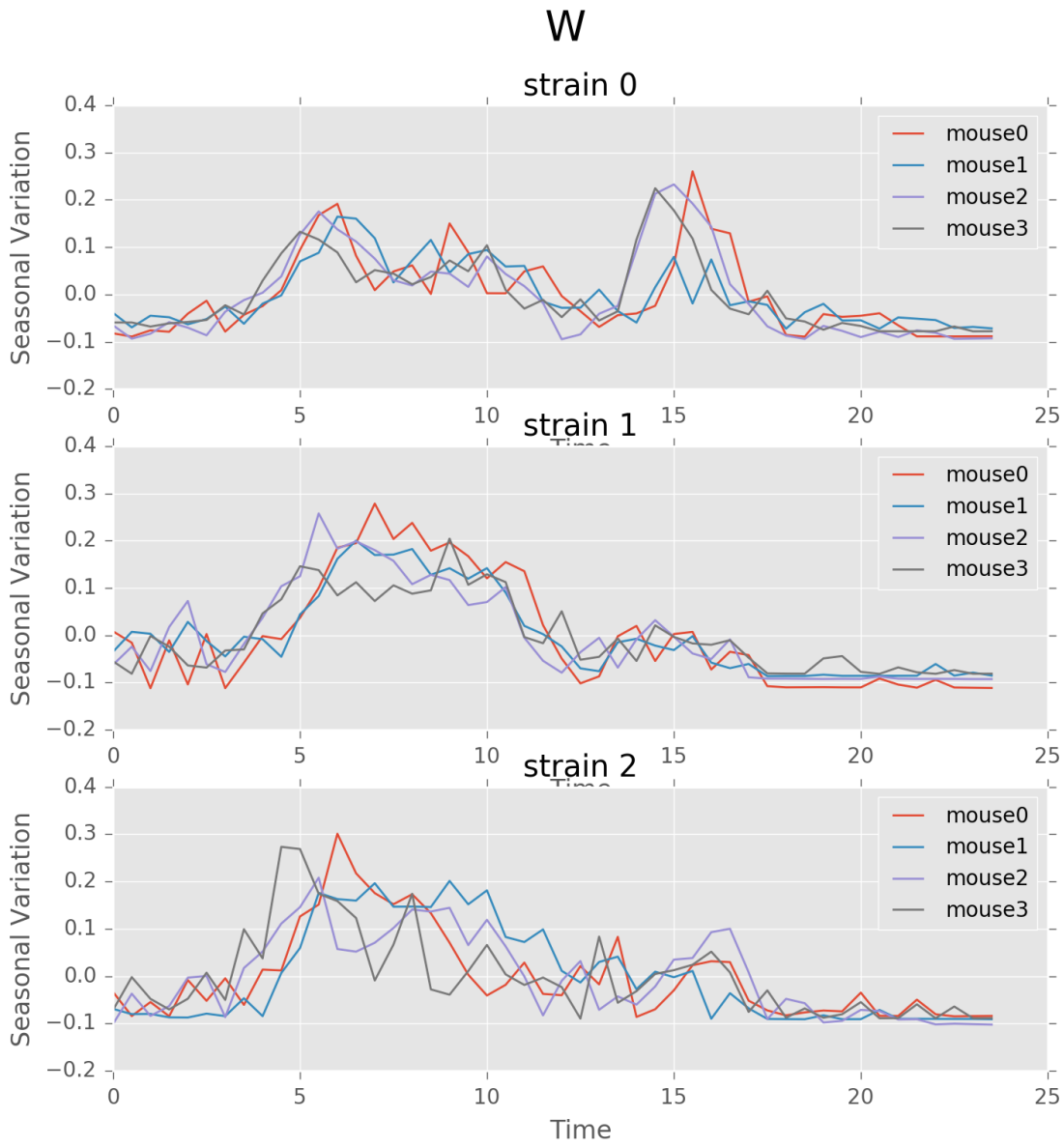


Fig. 1.4: Seasonal variation of other features (circadian). Strain0 and strain1 have more obvious patterns than strain3, which is consistent with the findings in longitudinal data analysis.

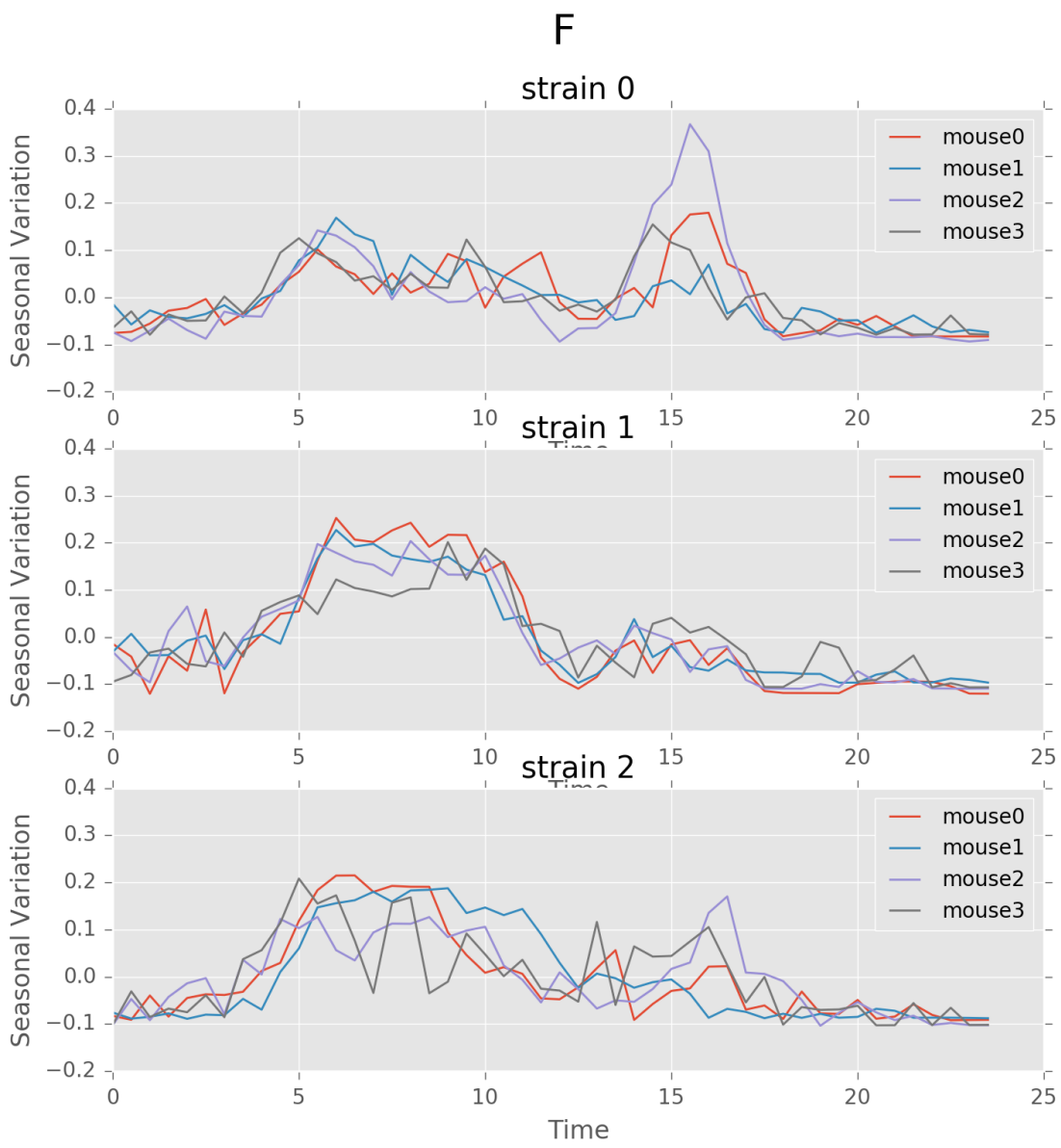


Fig. 1.5: Seasonal variation of other features (circadian).

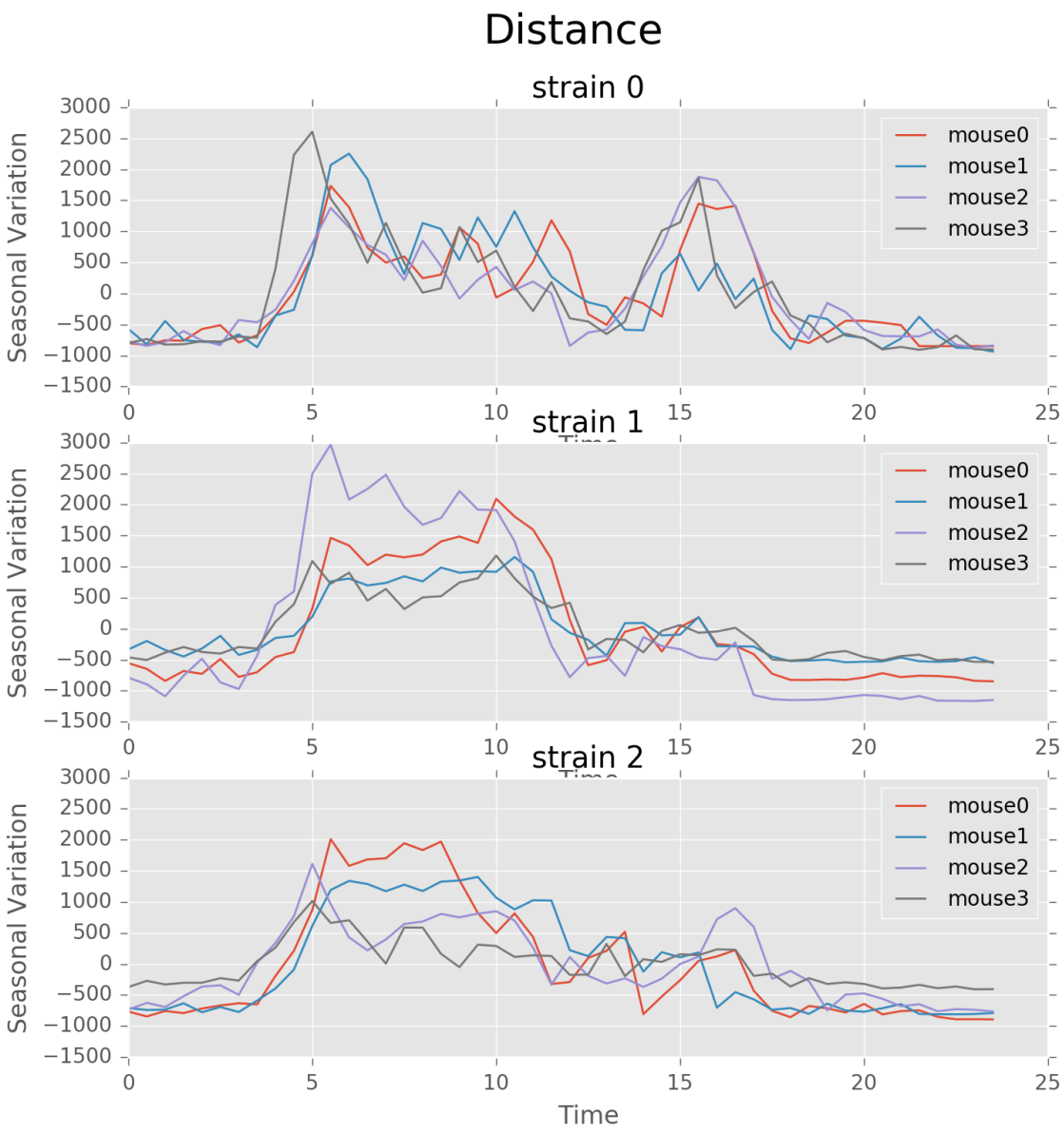


Fig. 1.6: Seasonal variation of other features (circadian).

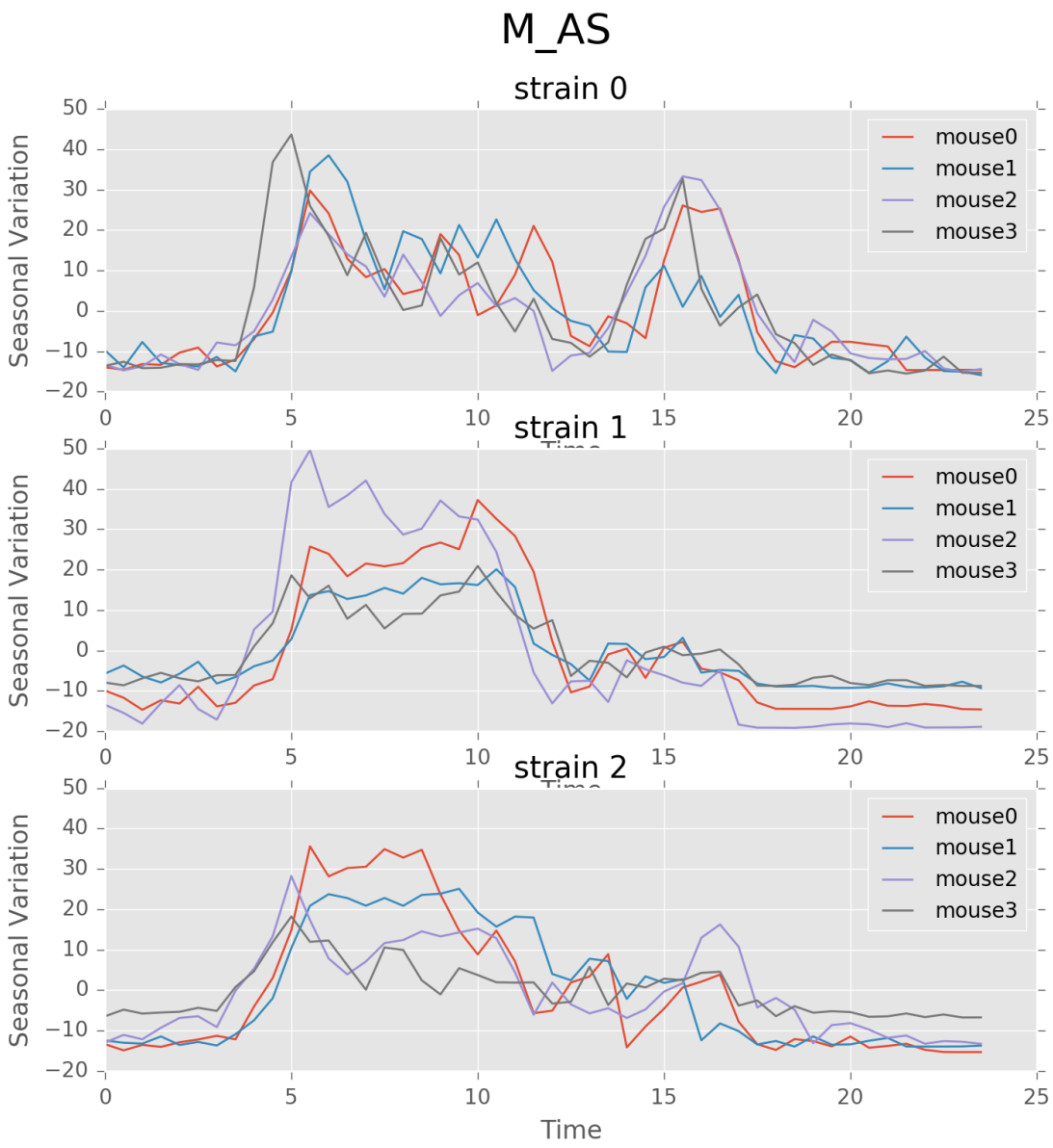


Fig. 1.7: Seasonal variation of other features (circadian).

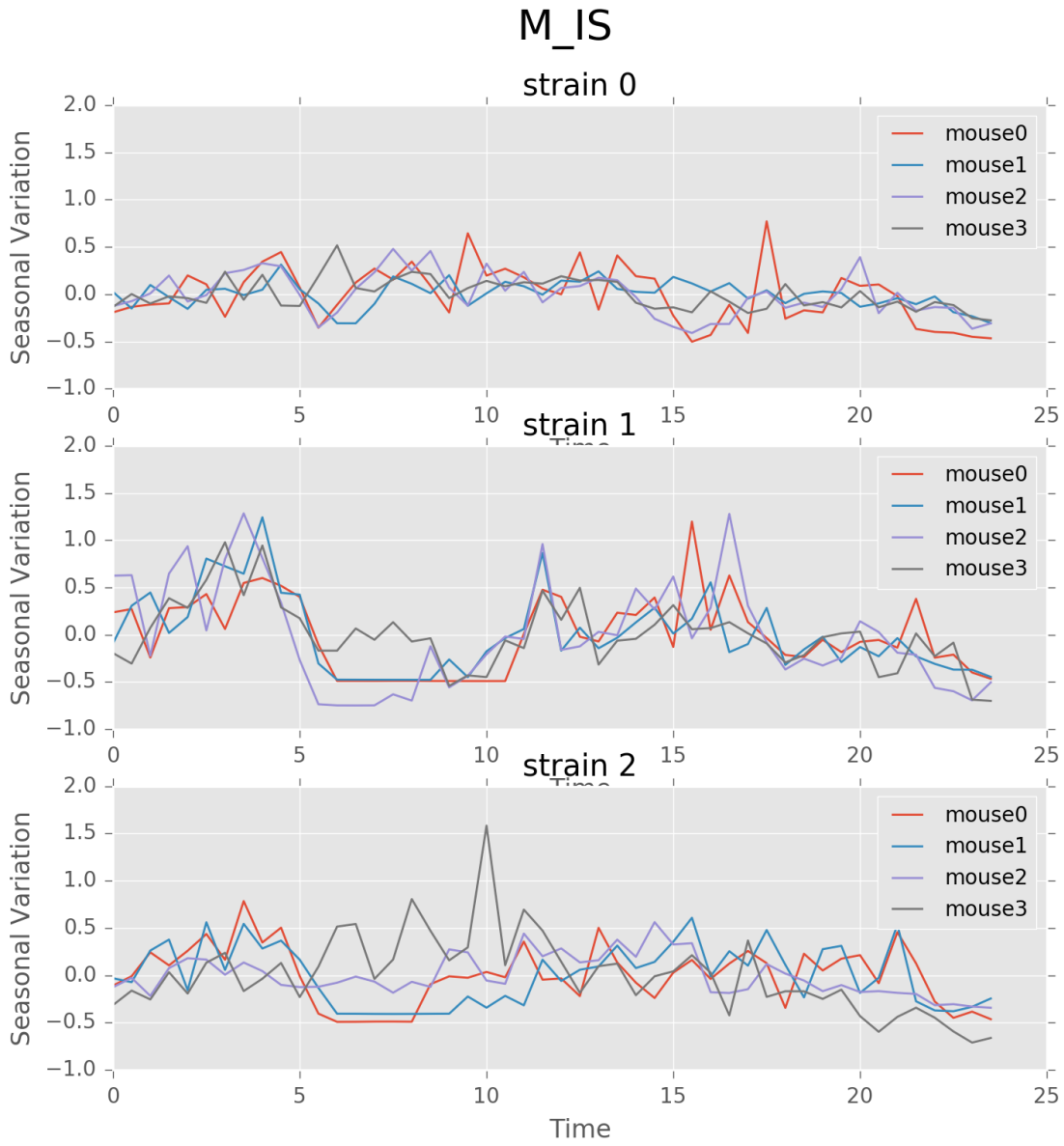


Fig. 1.8: Seasonal variation of other features (circadian). Strain0 and strain1 have more obvious patterns than strain3, which is consistent with the findings in longitudinal data analysis.

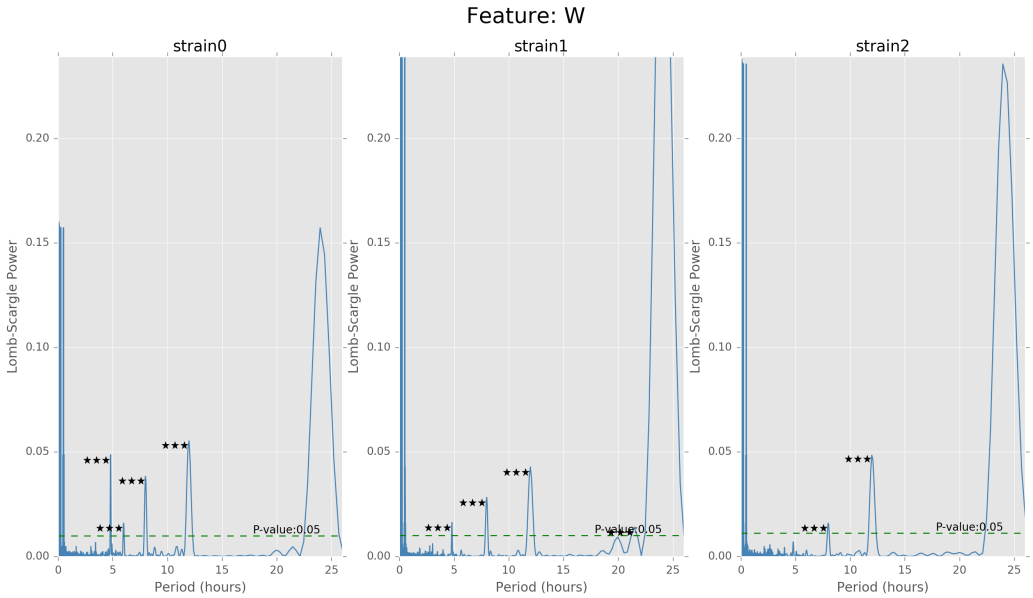
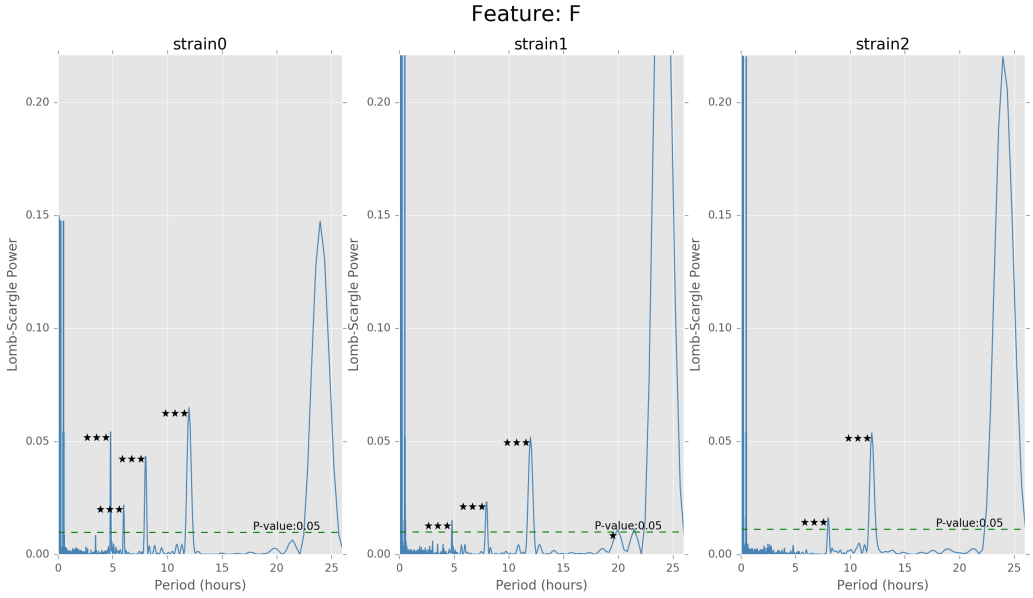
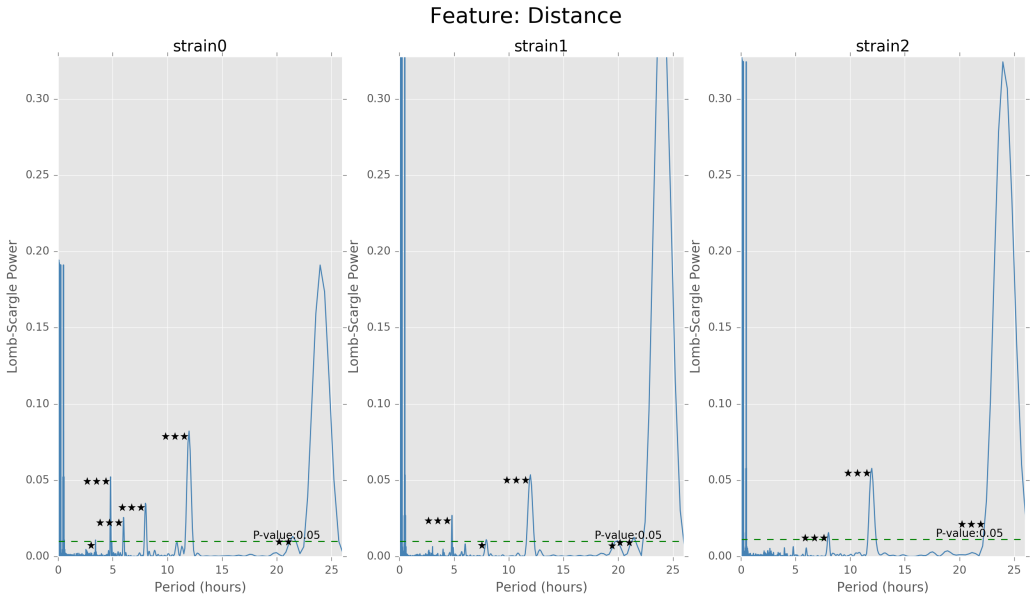
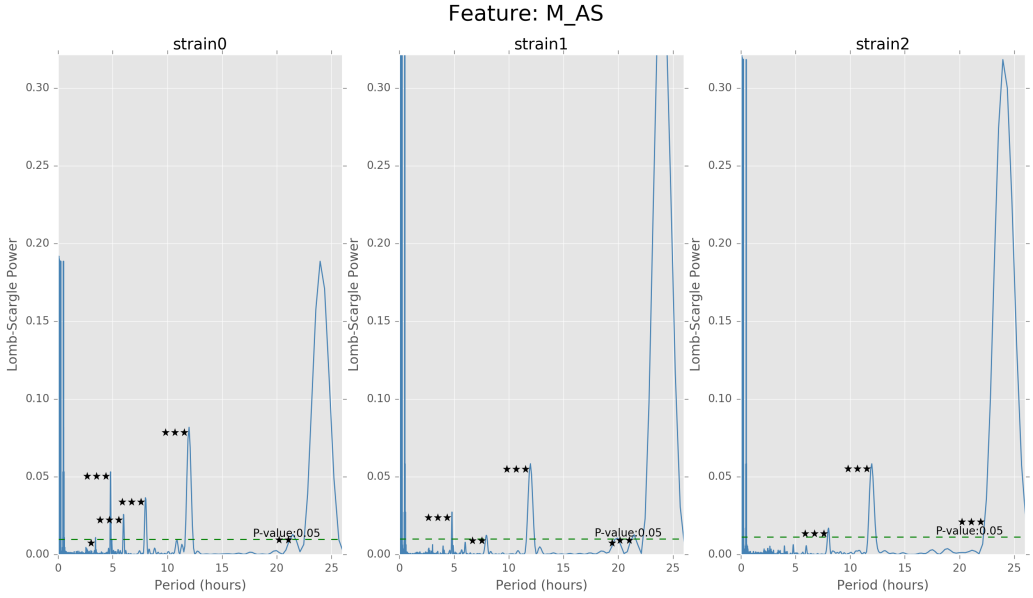


Fig. 1.9: Lomb scargle plot for different features. Different strains have different ultradian periods, differing also in p-values. Here $O(N \log N)$ algorithms suffer an instability around 0 points while $O(N^2)$ algorithms can be more smooth. We here compare the significant ultradian periods between strains and ignore the highest LS power appearing near 24 hours.




Feature: Distance
strain0
strain1
strain2
Lomb-Scargle Power
Period (hours)
P-value:0.05


Feature: M_AS
strain0
strain1
strain2
Lomb-Scargle Power
Period (hours)
P-value:0.05

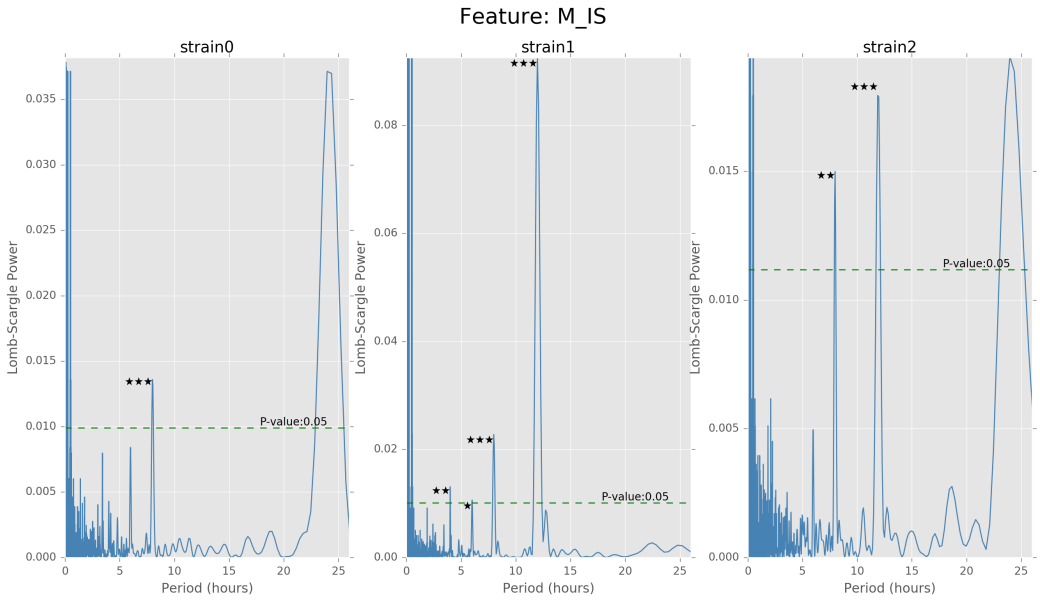


Fig. 1.10: Lomb scargle plot for different features. Different strains have different ultradian periods, differing also in p-values. Here $O(N \log N)$ algorithms suffer an instability around 0 points while $O(N^2)$ algorithms can be more smooth. We here compare the significant ultradian periods between strains and ignore the highest LS power appearing near 24 hours.

nor the same across different clustering methods. Criterion like silhouette scores would be evaluated to choose the best number of clusters. Petkov's paper also give us ideas about clustering and the difference between each strains *[PDC+04]*.

1.6.3 Exploratory Analysis & Classification Models

In 1D, box plots of each feature, say food consumption or sleeping time, of each strain are plotted below. In 2D, PCA can be performed on the feature data set and the data are then plotted along the first and the second principal axes colored in different strains. These plots are useful in verifying assumptions. For instance, we could box-plot different strains of mice against food consumption to see whether different strains of mice eat distinctly. If the number of variables needed to be evaluated is large, we might also use five number summaries to study the distributions.

Since each strain (each class) only has 9 to 12 mice, inputting too many features to the classification model is unwise. The exploratory data analysis will be an important step for hypothesis testing and feature selection. The process will also help us to find outliers and missing values in each behavioral variable, and we will decide how to handle those values after encountering that.

1.6.4 Data Requirements Description

For this project, we use behavioral data recorded for each mouse and each two-hour time bin of the day. There are in total 170 mice across 16 different strains. Behavioral features include measurements of daily activities, such as food and water consumption, distance moved, the amount of time a mouse is in active state, and the amount of time a mouse spends eating, drinking and moving while in active state.

For classification and clustering subsections, we require different processed data.

For **classification** project, we have two different datasets – mouseday dataset or individual mouse dataset. For the mouseday dataset, we take each combination of mouse and day as a unique observation, resulting in 1921 observations.

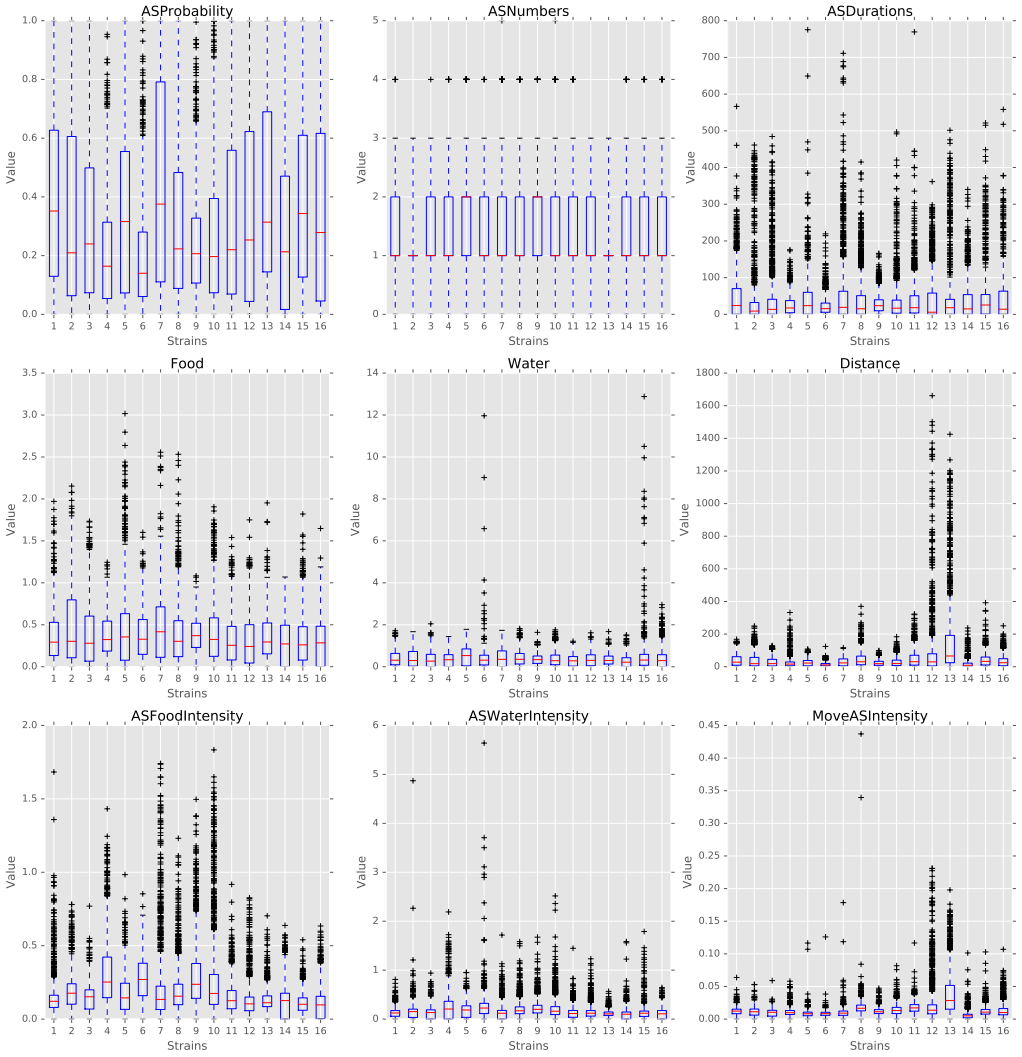


Fig. 1.11: Boxplots of the features over 170 mice

For the individual mouse dataset, we take the average measures of different days for each mouse to form 170 unique observations. In each dataset, we use the measures for different time bins as different features. Therefore, we have a maximum of 9 types of features * 11 time bins = 99 features. Users can also use a subset of these 99 features to train models. The mouse strain number will serve as the label for the dataset.

For **clustering project**, we used the individual mouse dataset which has 170 observations. For clustering, we do not want too many features since these features might have directions very close to each other and it would be repetitive to use all of them. Therefore, instead taking each hour bin as different features like in the classification dataset, we take each hour bin as different observations, resulting in only 9 features for the clustering data. Then we take average of days for each mouse. While preparing the input data, users will also have the choice to standardize data and/or add the standard deviation for each of the 9 features, resulting in possibly 18 final features. One thing to note is that we did not use mouseday data to perform clustering, although it is possible to do with our function. The reason we are not using big dataset is because the hierarchical clustering is computationally expensive and silhouette score also takes fairly long time to calculate.

1.6.5 Methodology

Classification

For this project, we mainly focus on three classification algorithms, which are random forests, gradient boosting and support vector machines (SVM).

Introduction

Random forests is a notion of the general technique of random decision forests that are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. The method combines Breiman's "bagging" idea and the random selection of features, correcting for decision trees' habit of overfitting to their training set.

Gradient boosting is another machine learning algorithm for classification. It produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. Gradient boosting fits an additive model in a forward stage-wise manner. In each stage, it introduces a weak learner to compensate the shortcomings of existing weak learners, which allows optimization of an arbitrary differentiable loss function.

Support vector Machines (SVM) are set of related supervised learning methods for classification and regression, which minimize the empirical classification error and maximize the geometric margin. SVMs map the input vector into a higher dimensional space where the maximal separating hyper plane is constructed. By maximizing the distance between different parallel hyper planes, SVMs come up with the classification of the input vector.

Tuning Parameters

For each of the algorithms, we create functions to fit them on the dataset. There are two different ways to fit these methods: if the user pre-defines the set of the parameters, we will use cross validation to find the best estimators and their relative labels; if the user does not define the parameters, the functions will use the default values to fit the models.

For random forests, we tune n_estimators, max_feature and importance_level. n_estimators represents the number of trees in the forest. The larger, the more accurate. However, it takes considerable amount of computational time when increasing forest size. max_features represents the number of features to consider when looking for the best split. max_depth represents the maximum depth of the tree. The larger, the more accurate. However, it takes considerable amount of computational time when increasing tree size.

For gradient boosting, we tune n_estimators and learning_rate. n_estimators represent the number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting, therefore, a larger number represents more performing stages, usually leading to better performance. learning_rate will shrink the contribution of each tree by the value of learning_rate. There is a trade-off between learning_rate and n_estimators. We use GridSearch to tune the learning_rate in order to find the best estimator.

For SVM, we tune C and gamma. C represents the penalty parameter of the error term. It trades off misclassification of training examples against simplicity of the decision surface. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly. Gamma is the Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. It defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'.

Model Assessment

After tuning our parameters, we apply our models to testing set and compare the prediction labels with the true labels. There are two major ways to measure the quality of the prediction process, one is a confusion matrix and the other is percentage indicators including precision, recall, and F-1 measure. A confusion matrix is a specific table layout that allows visualization of the performance of an algorithm. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class. The name stems from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabeling one as another).

precision(P) = $\frac{\#label\ y\ \ predicted\ correctly}{\#label\ y\ predicted}$

recall(R) = $\frac{\#label\ y\ \ predicted\ correctly}{\#label\ y\ true}$

F-1 = $\frac{2*P*R}{P+R}$

Thus, precision for each label is the corresponding diagonal value divided by row total in the confusion matrix and recall is the diagonal value divided by column total.

Clustering

Unsupervised learning clustering algorithms, K-means and hierarchical clustering, are included in the subpackage `classification`. Unlike other clustering problems where no ground truth is available, the biological information of the mice allows us to group the 16 strains into 6 larger mouse families, although the 'distances' among the families are unknown and may not be comparable at all. Hence, cluster numbers from 2 to 16 should all be tried out to find the optimal. Here, we briefly describe the two algorithms and the usage of the related functions.

Above all, note that unlike the supervised classification problem where we have 11 levels for one feature (so we have up to 99 features in the classification problem), the unsupervised clustering methods could suffer from curse of high dimensionality when we input a large amount of features. In high dimension, every data point is far away from each other, and the useful feature may fail to stand out. Thus we decided to use the average amount of features over a day and the standard deviation of those features for the individual mouse (170 data points) case.

K-means

To begin with, *K-means* minimizes the within-cluster sum of squares to search for the best clusters set. Then the best number of clusters was determined by a compromise between the silhouette score and the interpretability. K-means is computationally inexpensive so we can either do the individual mouse options (170 data points). However, the nature of K-means makes it perform poorly when we have imbalanced clusters.

Hierarchical Clustering

Given the above, the potentially uneven cluster sizes lead us to consider an additional clustering algorithm, *hierarchical clustering*, the functionality of which is included in the subpackage. Generally, hierarchical clustering seeks to build a hierarchy of clusters and falls into two types: agglomerative and divisive. The agglomerative approach has a "richer get richer" behavior and hence is adopted, which works in a bottom-up manner such that each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. The merges are determined in a greedy manner in the sense that the merge resulting in the greatest reduction in the total distances is chosen at each step. The results of hierarchical clustering are usually presented in a dendrogram, and thereby one may choose the cutoff to decide the optimal number of clusters.

Below is a demo to fit the clustering algorithm. The loaded data is firstly standardized, and then the optimal distance measure and the optimal linkage method are determined. We have restricted the distance measure to be l1-norm (Manhattan distance), l2-norm (Euclidean distance) and infinity-norm (maximum distance), and the linkage method to be ward linkage, maximum linkage and average linkage. The maximum linkage assigns the maximum distance

between any pair of points from two clusters to be the distance between the clusters, while the average linkage assigns the average. The ward linkage uses the Ward variance minimization criterion. Then, the optimal linkage method and distance measure are input to the model fitting function, and the resulting clusters and corresponding silhouette scores are recorded for cluster number determination. A plotting function from the subpackage is also called to output a plot. The output plot is included in the result section of the report.

```
from mousestyles import data
from mousestyles.classification import clustering
from mousestyles.visualization import plot_clustering

# load data
mouse_data = data.load_all_features()

# rescaled mouse data
mouse_dayavgstd_rsl = clustering.prep_data(
mouse_data, melted=False, std=True, rescale=True)

# get optimal parameters
method, dist = clustering.get_optimal_hc_params(mouse_day=mouse_dayavgstd_rsl)

# fit hc
sils_hc, labels_hc = clustering.fit_hc(
    mouse_day_X=mouse_dayavgstd_rsl[:,2:],
    method=method, dist=dist, num_clusters=range(2,17))

# plot
plot_clustering.plot_dendrogram(
    mouse_day=mouse_dayavgstd_rsl, method=method, dist=dist)
```

1.6.6 Testing Framework Outline

To ensure our functions do the correct steps and return appropriate results, we also implemented test functions. For clustering, we first perform basic testing of whether our output has appropriate number of values or values we expect. One more advanced check we perform is to test whether we successfully assign cluster numbers to every observation. Also, since we compute silhouette score for each cluster and silhouette score is defined to be between -1 and 1, we also checked that whether our silhouette score is appropriate. For classification, we also checked whether our final predictions of mouse strains only include numbers 0 through 15 since they are the only strains for data we have and we should predict those strains.

1.6.7 Result

Classification

For three models, after tuning the parameters and output the prediction result, we create the side-by-side barplot for the different measurement of accuracy, which are precision, recall and F1.

Random Forest

Random Forest shows a very promising result. For each strain, prediction, recall and F-1 measure are very close to each other. Except for predicting strain 15, all the other prediction has F-1 measure exceeding 0.8.

We also select the most important features, including ASProbability_2, Distance_14, ASProbability_16, Distance_2, Food_4, MoveASIntensity_2, ASProbability_4, Distance_4, Distance_16.

Gradient Boosting

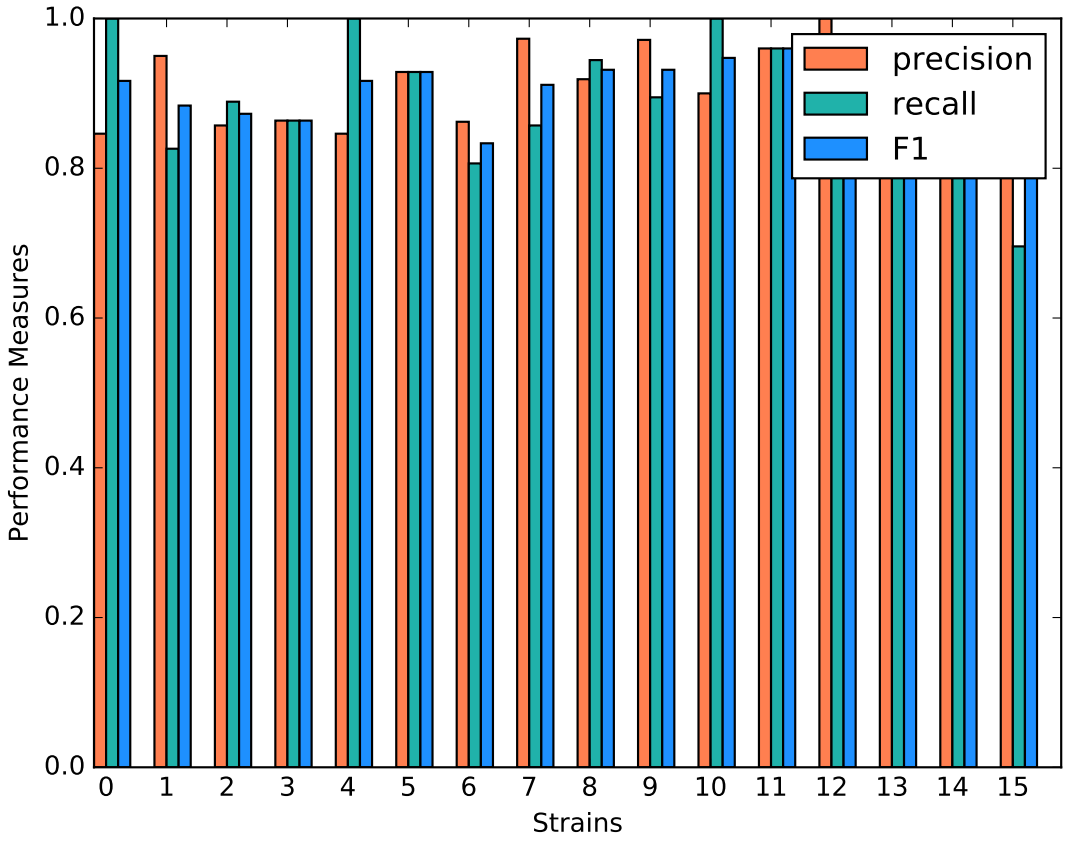


Fig. 1.12: Classification Performance of Random Forest

Gradient Boosting shows a decent performance on the prediction. There is no huge difference in precision and recall for predicting each strain, but bigger than Random Forest. It is shown that strain 3, 7 and 10 shows obvious higher prediction than recall. Almost all the accuracy measurement is above 0.8.

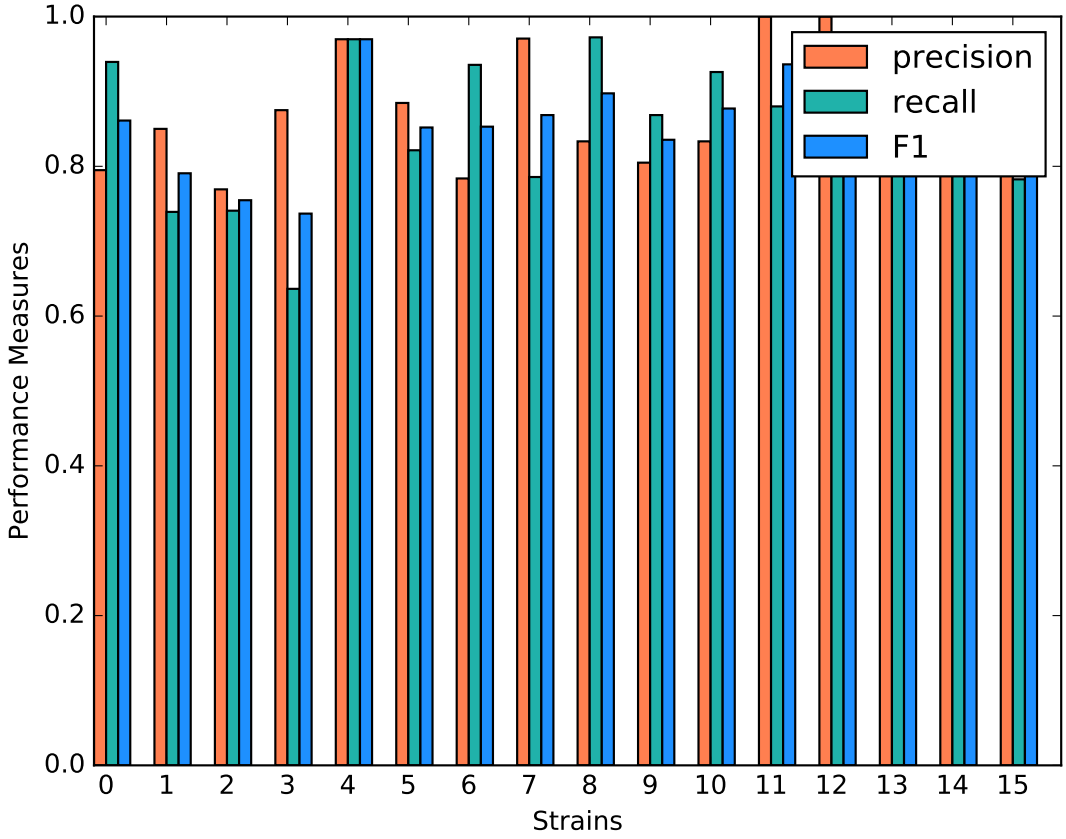


Fig. 1.13: Classification Performance of Gradient Boosting

SVM

SVM model shows a very inconsistent performance on the prediction. For example, the precision for predicting strain 3,4,11,12,15 is 1 while the precision for predicting strain 6,9 is below 0.5. Although precision for predicting strain 3,11,12,15 is very high, the recall for predicting these strains are much lower, resulting in a low F-1 measurement. The high precision and low recall indicates that we can trust the classification judgements, however the low rate of recall indicates that SVM is very conservative. This might be good if we are worried about incorrectly classifying the strains.

Comparison

By plotting side-by-side barplot of F-1 measurement among the three models, we can clearly see that Random Forest model provides the best result and SVM is the worst. Performance of Random Forest and Gradient Boosting are similar, but the SVM is obviously weak. So we recommend predicting strains by implementing the Random Forest model.

Clustering

K-means

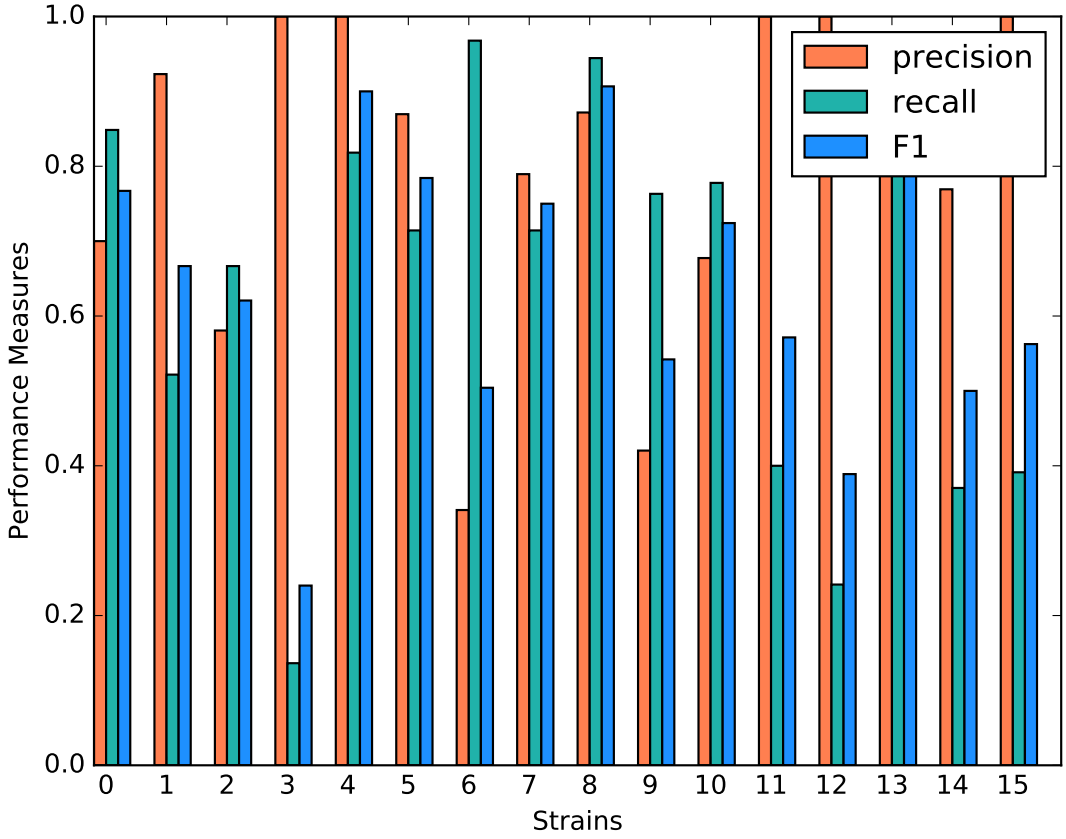


Fig. 1.14: Classification Performance of SVM

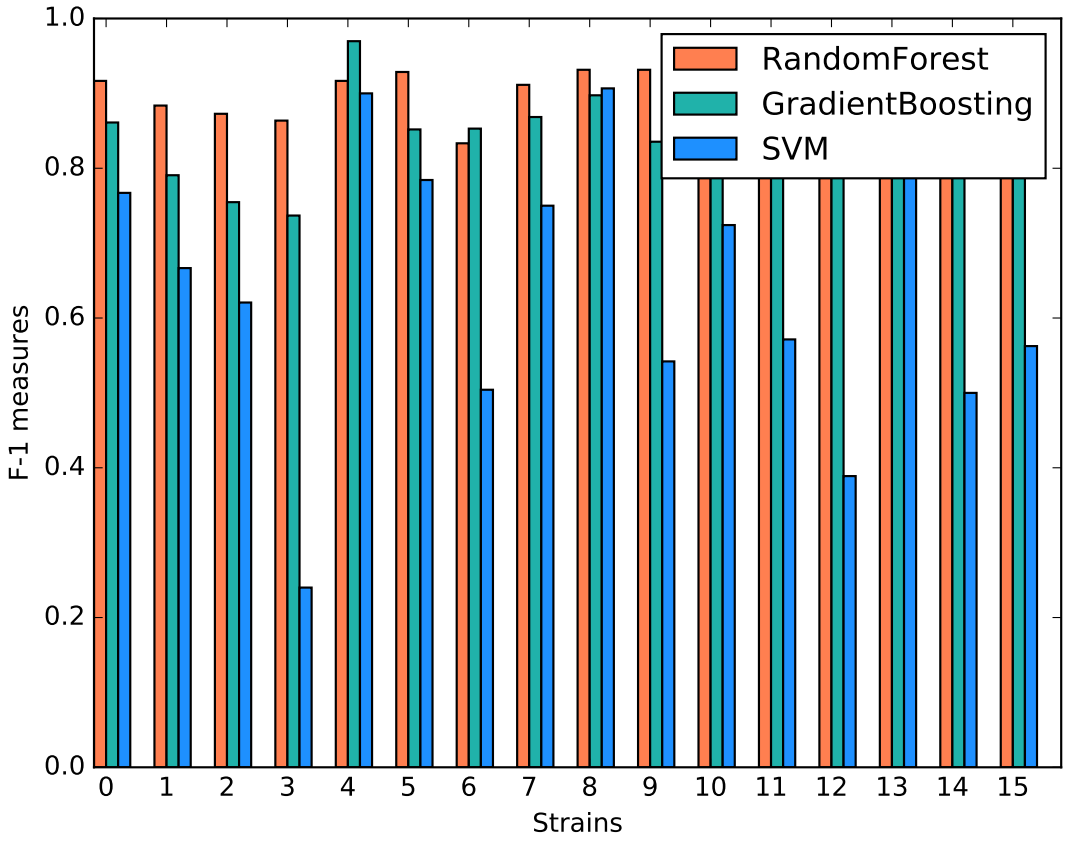


Fig. 1.15: Comparison of F1 measures of Different Classification Models

The silhouette scores corresponding to the number of clusters ranging from 2 to 16 are: 0.835, 0.775, 0.423, 0.415, 0.432, 0.421, 0.404, 0.383, 0.421, 0.327, 0.388, 0.347, 0.388, 0.371,0.362. We plot 6 clusters here to show, and found that Czech and CAST mice behaved quite differently from each other.

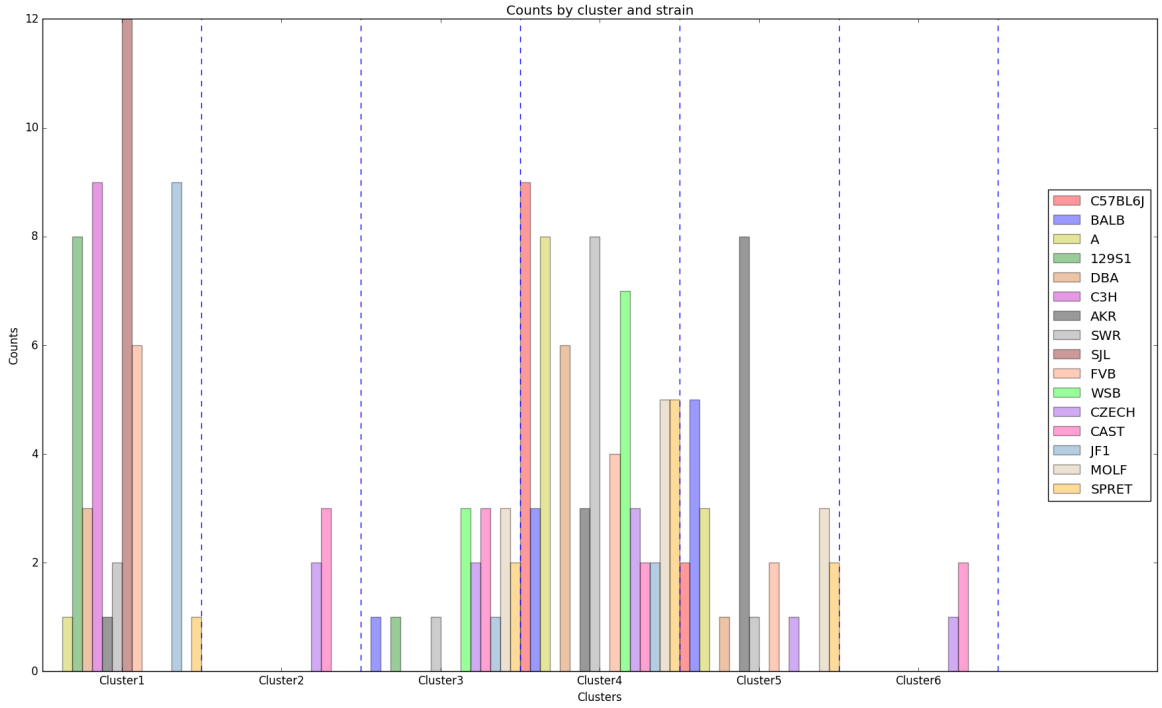


Fig. 1.16: Distribution of strains in clusters by K-means algorithm (Generated by *plot_strain_cluster* function; script can be found in *report/plots* directory.)

Hierarchical Clustering

The optimal distance measure is l1-norm and the optimal linkage method is average linkage method. The silhouette scores corresponding to the number of clusters ranging from 2 to 16 are: 0.8525, 0.7548, 0.7503, 0.6695, 0.6796, 0.4536, 0.4557, 0.4574, 0.3997, 0.4057, 0.3893, 0.3959, 0.4075, 0.4088, 0.4179. It seems 6 clusters is a good choice from the silhouette scores.

However, the clustering dendrogram tells a different story. Below shows the last 10 merges of the hierarchical clustering algorithm. The black dots indicate the earlier merges. The leaf texts are either the mouse id (ranges from 0 to 169) or the number of mice in that leaf. Clearly, we see that almost all the mice are clustered in 2 clusters, very far from the rest individuals. Thus, the hierarchical clustering fails to correctly cluster the mice in the case.

The failure of the the algorithm might be due to the different importance levels of the features in determining which cluster a mouse belongs to. One improvement could be that using only the important features determined in the classification algorithms to cluster the mice, but given the unsupervised learning nature of the algorithm, not using the results from the classification is fair for clustering tasks.

The distribution of strains in each cluster in the case of using 6 clusters are shown below. Obviously, the mice almost fall into the same cluster.

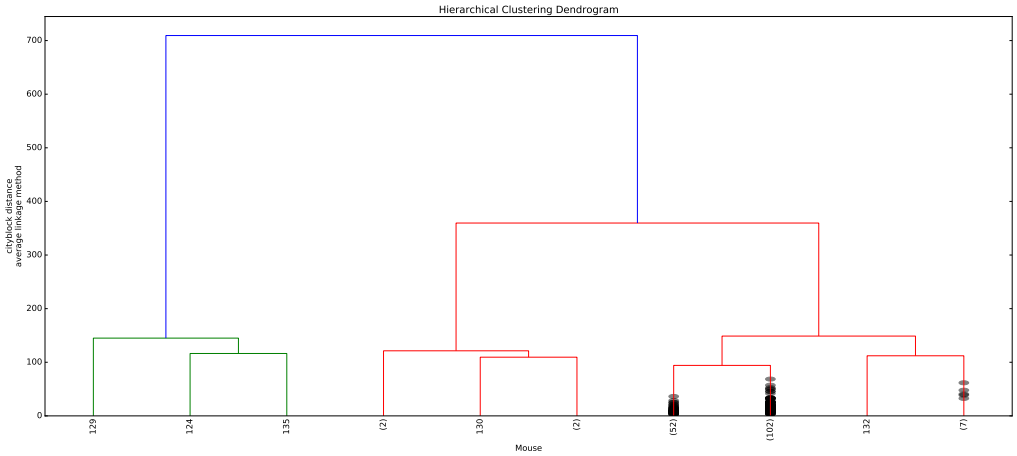


Fig. 1.17: Dendrogram of the hierarchical clustering

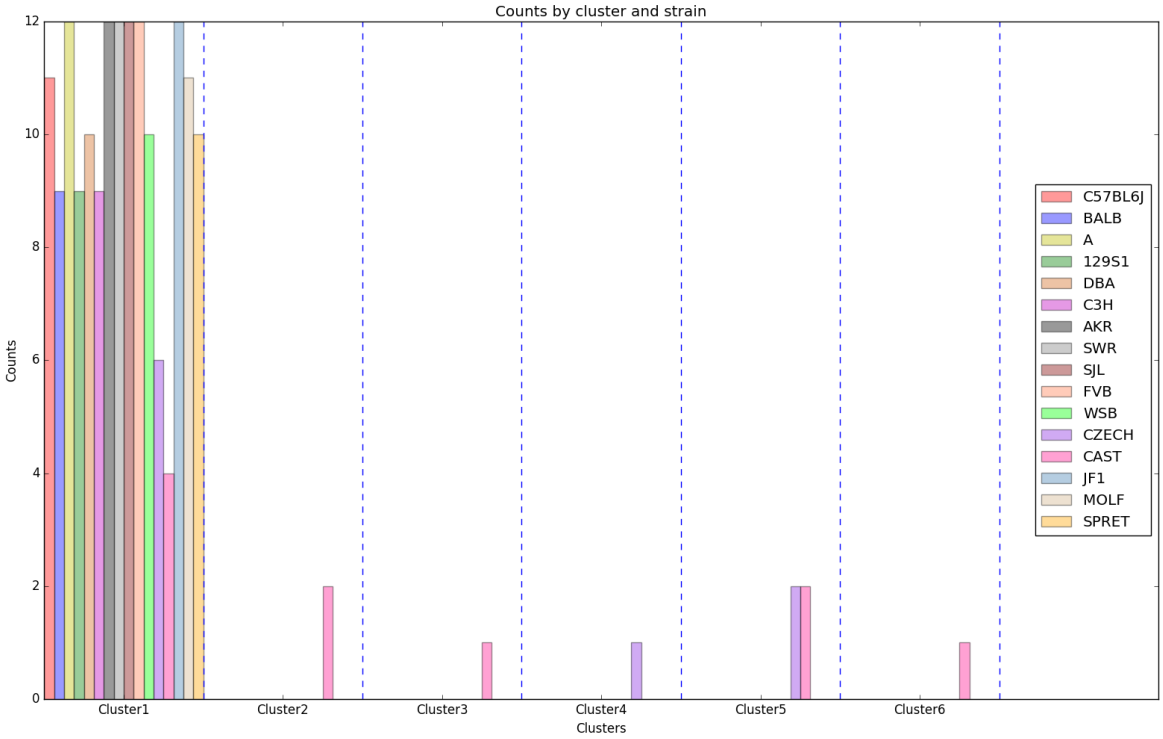


Fig. 1.18: Distribution of strains in clusters by agglomerative hierarchical clustering (Generated by *plot_strain_cluster* function; script can be found in *report/plots* directory.)

1.6.8 Future Work

The future research should focus more on feature engineering, including the questions of whether more features could be added to the model. Moreover, even though we have extracted the important features from the random forest to evaluate the performance of the smaller model, it seemed that the economized model did not perform as expected. In the future, other techniques like PCA might be performed to reduce the complexity of the model in order to train classification models faster.

To understand more about the nature of the strain difference, it would be better to have a sense of relationships between different strains of mice. For instance, we have explored that these 16 strains of mice belong to 7 different groups, which implied that some strains were genetically similar. Considering the time limit, we have put it to the future work.

1.7 Power Laws and Universality

1.7.1 Statement of Problem

Construct Statistical Model for Locomotion Distribution

To take a closer look at the tail property of locomotion, we analyze the moving distance distribution for each string. Fitting those distributions using some prior knowledge and obtain estimator for each mouse. With expectation, it is possible to cluster mice by the their unique distribution estimator, and further figure figure out with strain they belong to.

Main Questions to Answer

- What is the distribution model we could use to fit?
- How can we estimate parameters for our distribution?
- How can we do clustering strains based on model parameters?

1.7.2 Statement of Statistical Problem

Distribution Selection

The major statistical question is how to choose fitted distribution family. Based on conventions and data we have, we propose two distributions: law decay distribution and exponenital distribution:

In statistics, power law, also known as a pareto, is a functional relationship between two quantities, where a relative change in one quantity results in a proportional relative change in the other quantity, independent of the initial size of those quantities: one quantity varies as a power of another. $F(x) = kx^{-a}$.

There is a problem with power law or pareto distribution when took a close look at the distance distribution. This power law decay only works for monotone decreasing distribution. While it is not for our case. Therefore, two methods are designed to handle this.

First of all, we proposed to only estimate the powerlaw on the tail of distribution. Namely, truncate the small unstable distances on the left tail of distribution. Secondly, the power function may not be realistic by nature as the observed distribution is not monotone decreasing. In this case, the distribution is left skewed with one peak (see exploratory analysis). Thus we can make more general assumption, such as exponential distribution.

Fitness Comparsion

Previously we proposed two different parametric families and each of them make sense but we need to have more quantitative evidence to see the fittness or, hopefully, compare their goodness of fit.

The check of fitness is in needed after obtaining the parameter. Kolmogorov-Smirnov test enable us to test the fitness of power law and exponential distribution. For them, we choose the null to be selected distribution, and if we reject K-S test, we will conclude the the sample distribution is not familiar with theoretical distribution; thus, we have enough evidence against our null.

After that, to further compare these two distribution's capability of capturing the pattern of distances, we conducted a generalized likelihood ratio test (GLRT), with both two to be null hypothesis consecutively. Moreover, it may be hard to know the exact distribution of our test statistics and thus extremely hard to calculate the critical value or p-values. Therefore, we may need simulation methods to estimate critial value of p-value, and the major question is how can we generate random number from our fitted model.

1.7.3 Exploratory Analysis

The difference between "home base" and "non home base": "home base", which means a favored location at which long periods of inactivity (ISs) occur, is a post defined characteristic of the mouse and it is classified by its behavior. For instance, if one mice spend about 80% of the time outside of the nest, then we may conclude it is not home base and we will cluster it as non-home based mouse. This classification is not done by one specific mouse, but a strain of mouse. By the Exploratory Analysis, we will find significant difference between two classes and hopefully we can make connection about the speed of mice and its home based or non home based property.

The definition of inter-event distance: literally, inter-event distance is the distance within one single event. Investigating the data of the distance between each two consecutive points recorded by the detector of one mouse day, we found the shape of the histogram is similar to the plot on the slide, except the value of frequency is bigger. This inconsistency gives us the motivation to calculate the inter event distance instead of the distance between each two points. For this purpose, we need a vector of the index of events for each mouse day. In particular, the mapping vector is connected by time since xy coordinates are recorded according to time and the event is defined based on time.

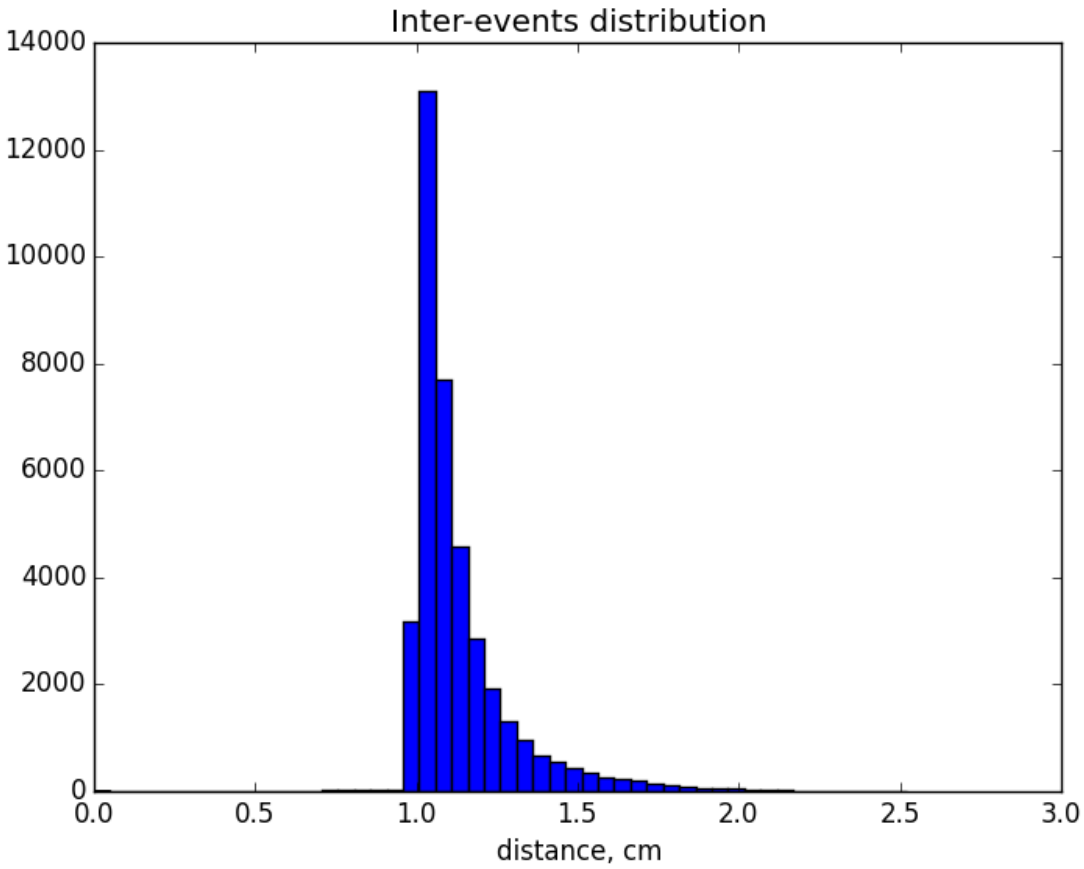
Preferred choice of distribution: the power law is a monotone decreasing, however our plot indicates a peak. Therefore, we can either choose differernt parametric distribution family, such as gamma distribution, or choose a threshold to make our distribution monotonic decreasing. Based on what Chris said about the sensor, as it will only detect when a mice moves more than 1cm within in 20ms. It does not make sense to have observations less than 1cm and outliers needs to be cleaned up. After we put a threshold with 1cm, we find the distribution is monotonic decreasing and we can fit power law distribution, as well as exponential distribution.

1.7.4 Distribution

- Preferred choice of distribution: the power law is a monotone decreasing, however our plot indicates a peak, in which gamma distribution may fit better.
- Another idea would be to record the position of the mouse at regular time intervals and compute the distances between them. This will give us an idea of the speed distribution of the mouse.

1.7.5 Data Requirements

- Label of "home base" or "non home base": generated in the process of data pre-processing by the definition. The build-in function enable us load data directly.
- Event index corresponding to the time: a vector mapped the time indicating the events.
- Distance: calculated by the square root of the sum of the difference x,y coordinates
- Speed: calculated by the distance divided by the duration of this distance.


Inter-events distribution
14000
12000
10000
8000
6000
4000
2000
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
distance, cm

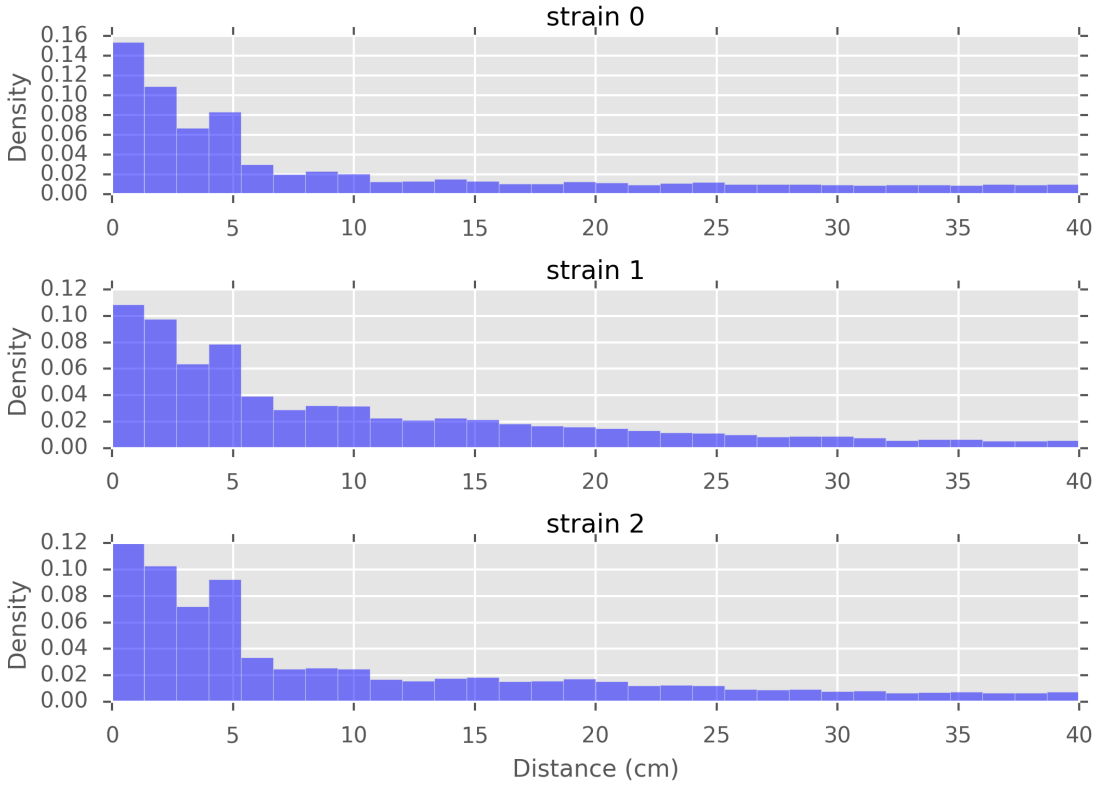


Fig. 1.19: Distances recorded every second, for different strains of mice

1.7.6 Methodology

Given the estimated parameter for each distribution, we can learn more about its distribution and the information lies mainly in the decay rate of the tail.

Here are our algorithms:

- Draw the histogram for our data. For each mouse day, observe the distribution and explore whether our distribution assumption makes sense.
- Estimate parameters based on MLE of truncated powerlaw, aka Pareto distribution and truncated exponential.
- Add the density function to our histogram, see the fitness of our distribution.
- Conduct statistical test to quantitatively analysis the fitness. For testing the hypothetical distributions of a given array, there are several existing commonly used methods:
  - Kolmogorov–Smirnov test
  - Cramér–von Mises criterion
  - Anderson–Darling test
  - Shapiro–Wilk test
  - Chi-squared test
  - Akaike information criterion
  - Hosmer–Lemeshow test

  However, each approach has their pros and cons. We adopt KS test since the Kolmogorov–Smirnov statistic quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. We recommend that all the methods are to be tried to get a comprehensive understanding of the inter-event step distributions.
- Conduct Generalized Likehood Ratio Test to compare the fitness of powerlaw and exponential. GLRT will calculate Likelihood Ratio which is the fraction of likelihood function with smallest KL divergence in two separate parametric space and then compare their peroformance.

1.7.7 Testing Framework Outline

The potential functions are recommended to implement:

- Retrieve data function (*retrieve_data*): Given the number of mouse and the date, create a data frame containing follow variables. 1) position: x,y coordinates 2) time: detecting time stamp for each pair of coordinates, time interval label for events, time interval label for active state and inactive state.
- Retrieve event function (*retrieve_event*): Given an event label (e.g. Food), subset respective part of data from the data frame we got in *retrieve_data*
- Compute the distance (*compute_distance*): Given event label, compute the distance between each time stamp. As we already know the x, y coordinates from the dataframe in *retrieve_event*, the simplest way to implement this function is that:

$$distance = ((x_t2 - x_t1)^2 + (y_t2 - y_t1)^2)^(1/2)$$

- Draw histogram (*draw_histogram*): Given a sub-array, using the plt built-in histogram function to draw the plot. Test distribution (*fit_distr*): Given the testing methods (e.g. "ks"), implement the corresponding fitting methods. The potential output could be p-value of the hypothesis test.

Based on the potential functions to be implemented, the following is the guide of testing:

- *test_retrieve_data*: attain a small subset of data from x,y coordinate and t, and feed in the function. Compare the results with the counted number.
- *test_retrieve_event*: Use the small data frame we get in test_retrieve_data, given different events/state. Compare the results with our counted number.
- *test_compute_distance*: Given x = 3, y =4, the output should be 5.
- *test_fit_distr*: randomly draw samples from widely used distributions (e.g. uniform). Test it with right(e.g. uniform) and wrong(e.g. gamma) distributions. Compare the p-values with given threshold (e.g. alpha = 95%)

1.7.8 Result

We fit the power law and exponential distribution for each mouse day. For each, we got an estimator of alpha for power law and an estimator of lambda for exponential. We store our result in a dataframe called estimation which has five columns: strain, mouse, day alpha and lambda. Draw histogram of the estimator where red, blue and green stands for different strains.

- The histogram of estimators from powerlaw:

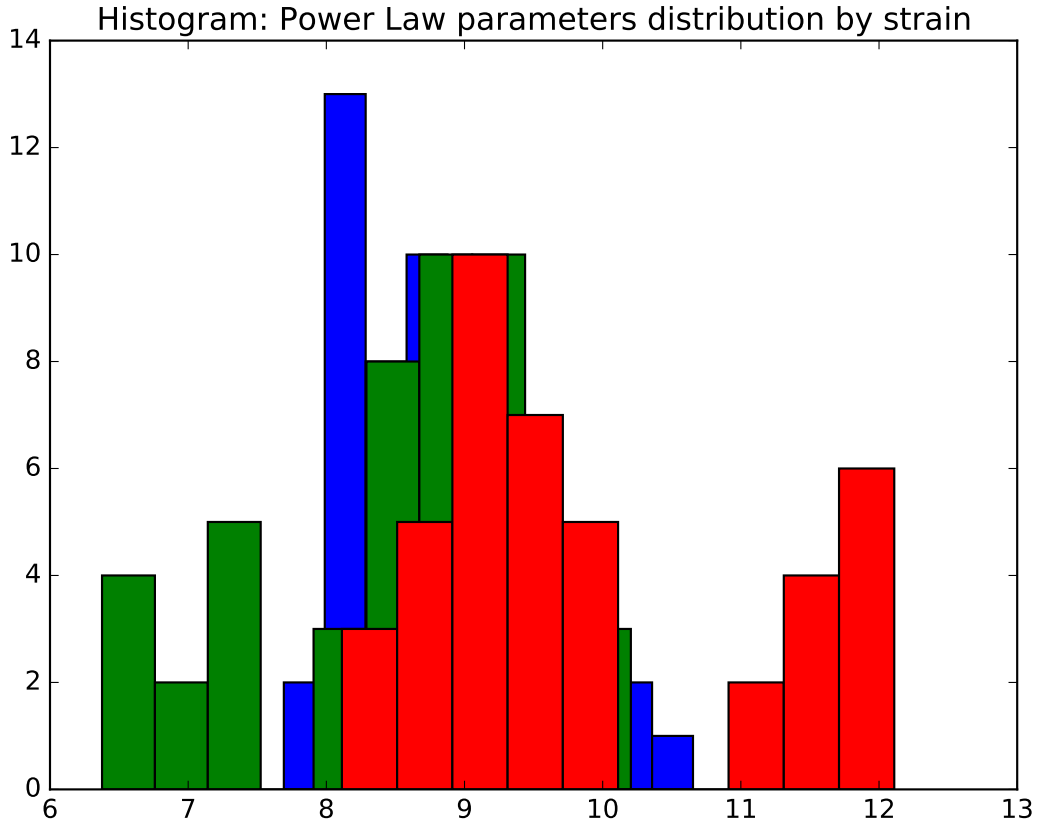


Fig. 1.20: Histogram of the parameters of powerlaw.

- The histogram of estimators from exponential:

We want to check the fitted curve with the original histogram of distance so we write of function to draw the power law and exponential curve with corresponding estimator with the original histogram of distance with the input of strain,

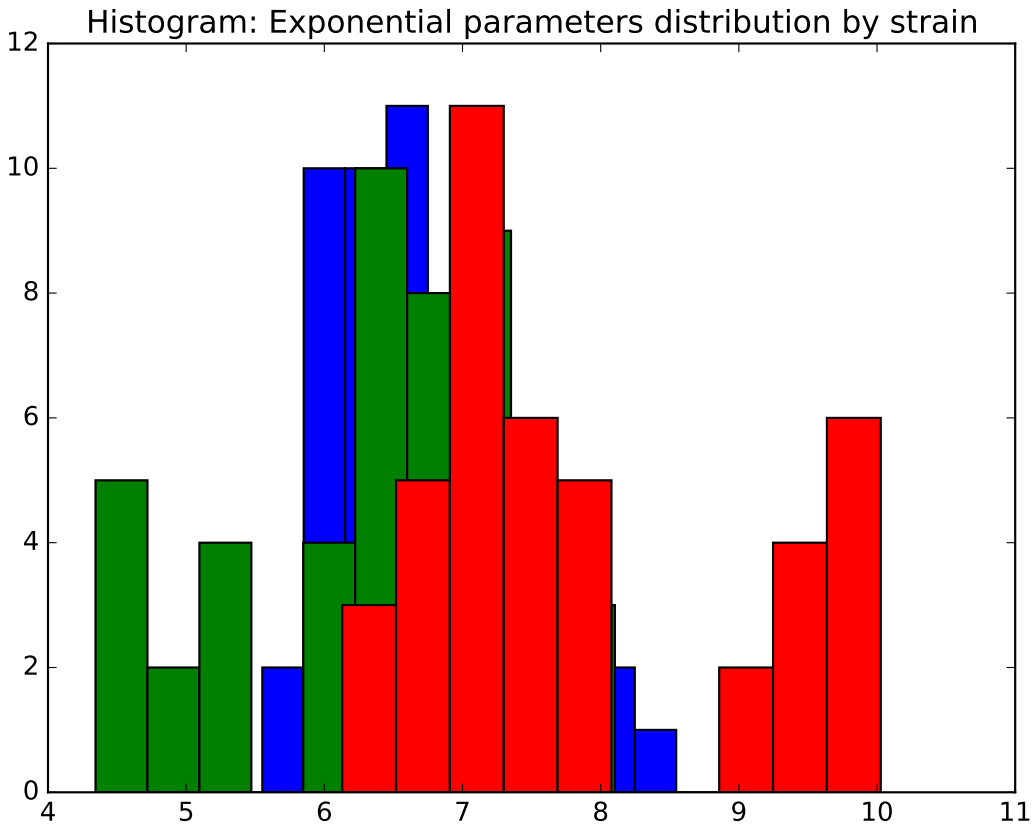


Fig. 1.21: Histogram of the parameters of exponential.

mouse and day. In particular, some normalization may be needed when doing the camparison and drawing the plot, for example, we intutively times (alpha-1) for the histogram. Here is an example of strain 0, mouse 2, day 5. From the plot we can see the fitting is pretty well.

- The histogram of data and fitted curve for strain 0, mouse 2, day 5:

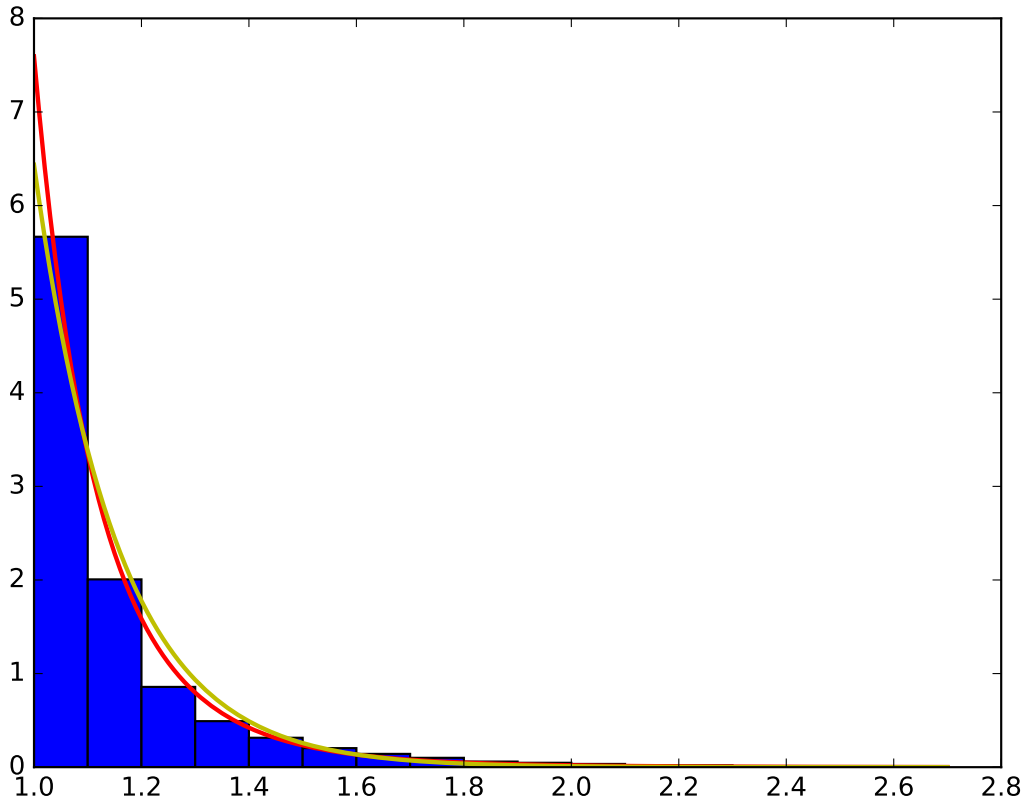


Fig. 1.22: Histogram and fitted curve for strain 0, mouse 2, day 5.

After visualize the fitting, we want to evaluate our fitting in statistical ways. There are several tests to quantify the performance and we adopt the KS test to evaluate the goodness of fit and GLRT test to compare fitness.

- Fitting power law distribution and gamma distribution for strain 0, mouse 0, and day 0; fitting by Maximum Likelihood, and by minimizing Kolmogorov CDF distances:
- Comparsion Between truncated Exponential and Powerlaw (Pareto) distribution.

One major question we want to answer: which distribution fits better, truncated exponential or truncated power law, aka pareto, distribution. To measure the distribution of the speed, the major difference is the tail distribution. You can also see it from the fitted plot. Both exponential distribution and pareto distribution fits quite well and they are actually very similar with each other, and the difference is barely noticeable. Therefore, it is hard to simply tell which distribution fits better. However, although the distribution is quite similar at the beginning, it diverse in the tail distribution. For exponential distribution, the tail decays with the rate e^{-x}, which is much faster than that of pareto distribution x^{-a}. Therefore, the goodness of fit is mainly determined by the tail distribution. We tried Kolmogorov test to determine whether our sample fits the theoretical distribution, but it does not compare two distributions.

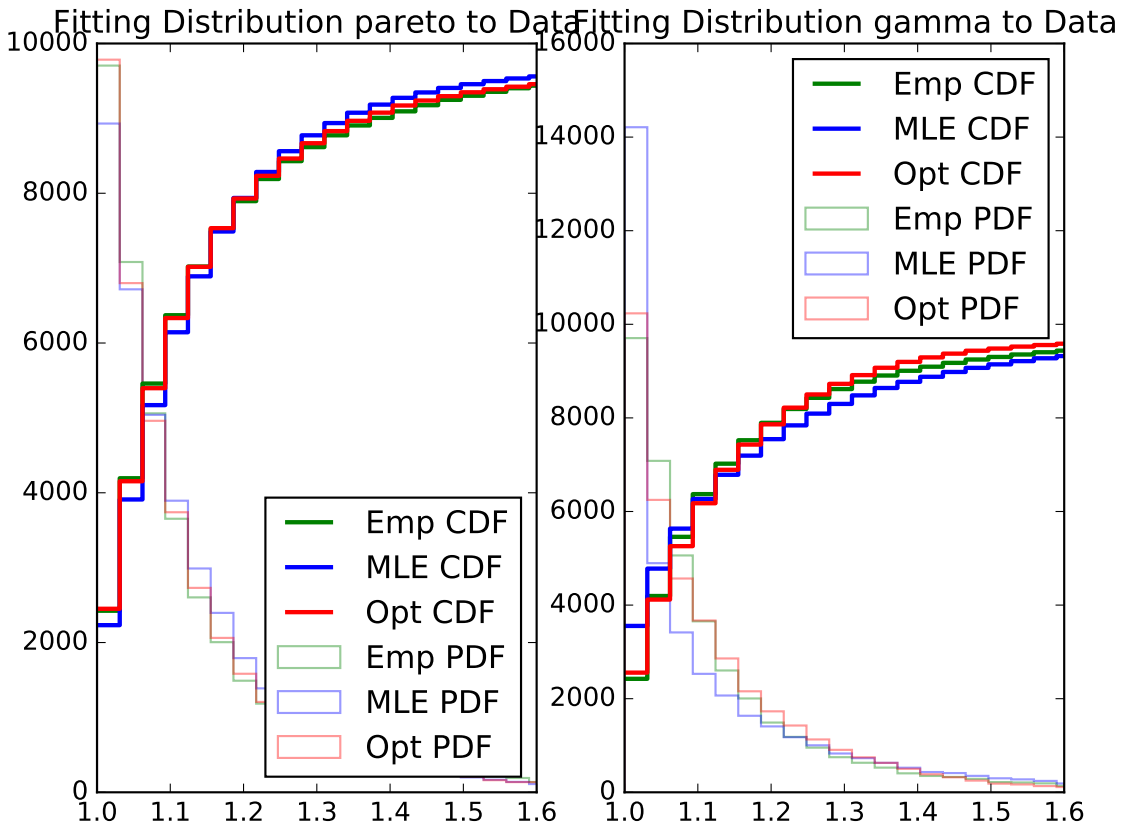


Fig. 1.23: Histogram of distances travelled in 20ms by strain 0, mouse 0, day 0.

To make comparison between two distributions, we used Generalized Likelihood Ratio Test to do hypothesis testing. As we cannot actually treat different distribution equally, with that being said, to do hypothesis testing, we must have null hypothesis and alternative hypothesis, where we tend to protect it and only reject when the we have strong evidence against it. Thus, we will conduct two hypothesis testings, with null being either exponential or power law distribution. We will expect there to be three possible outcomes.

- Exponential null rejected but power law null not rejected. In this case, we conclude power law is better than exponential.
- Power law null rejected but exponential null not rejected. In this case, we conclude exponential is better than power law.
- Both two tests not rejected. In this case, we conclude both two fits similarly and there is no one significantly better than another.

Although theoretically we should consider the case when both two tests are rejected, it is highly unlikely this thing happens. Because rejecting both two means we have enough evidence to say exponential is better and power law is also better, while not rejecting two might happen, as we tends to protect the null and if they react similarly, we don't have enough evidence to reject any of them.

Here is the algorithm to conduct the test. The GLRT test statistics is the ratio of likelihood, with numerator being likelihood under null set while that under alternative in numerator. It is intuitively right that we shall reject the null if our test statistics is too small. To make the significance level being 0.05, it is essential to find the critical value. However, it is hard for us to derive the distribution of test statistics and thus we use simulation to estimate it. Thus, we draw random number from null distribution and then calculate the test statistics. Also, p-value is a better statistics and it will not only tell us whether we should reject the null, but also tell us what is the confidence that we reject the null.

From the outcome of our function, we actually find the p-value from exponential null is very close to 1, while that from power law null is very Small, next to 0.0005. This is a strong evidence that we should not think power law is a better fit than exponential. Thus, we conclude that we should use exponential to fit and do further research.

1.7.9 Relative distribution with kernel smooting

In the previous section, the result of K-S test suggests that the actual distribution of distance doesn't follow the power law family but may follow in the exponential family. In this section, we characterize discrepency of the actual distribution to these two by looking at their relative distribution.

In the relative distribution framework, we call the actual distribution G comparison distribution and call the proposed distribution F_0, e.g. power law with parameter α, the reference distribution. The idea of the relative distribution is to characterize the change in distributions by using F_0 to transform X into $[0, 1]$, and then look at how the density departs from uniformity.

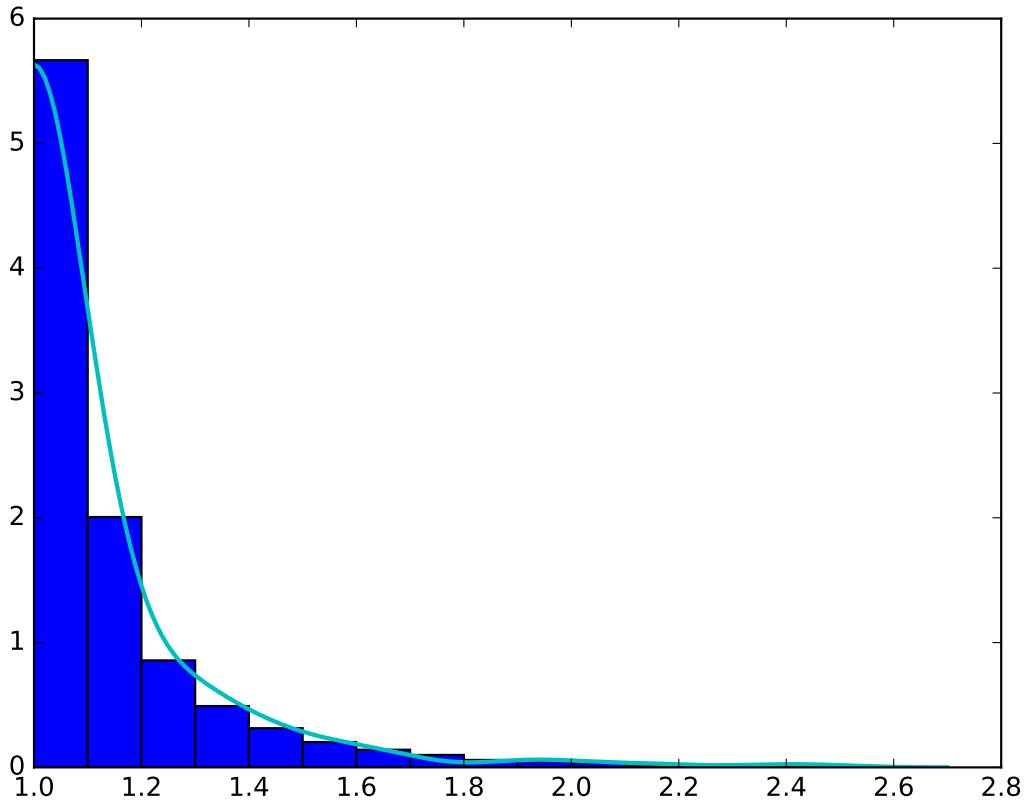
The relative data, or grades, are $r_i = F_0(x_i)$, which lies in $[0, 1]$. Its probability density function is \$f_r(x) = frac{g(F_0^{-1}(x))}{f_0(F_0^{-1}(x))}\$. r has uniform density in the special case when $G = F$.

As shown in the previous formula, we need to get the density g of the observed data to calculate the relative distribution. However, the naive nonparametric density estimation with gaussian kernel doesn't work well when the random variable is nonnegative. We propose a strategy using symmetric correction to work it around.

In the first step, we create the symmetric-corrected data X' by concatenating the original data X and its reflection around its left boundary point 1, $2 - X$. In the second step, we get f_1, the density estimate using gaussian kernel with bandwidth maximizing 5-fold cross-validated score. In the third step, we delete $2 - X$ and set the density estimate of X to be $f = 2f_1$. An example is shown in the plot.

We construct the relative distribution to the best-fitted power law distribution and exponential distribution using this density. The result is shown in the plotted density curve.

Now, the fine relative density structure is very clear. We see our actual distribution has a smaller density at the beginning and a thicker tail compared to both power law distribution and exponential distribution. Exponential has a


6
5
4
3
2
1
0
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
2.8

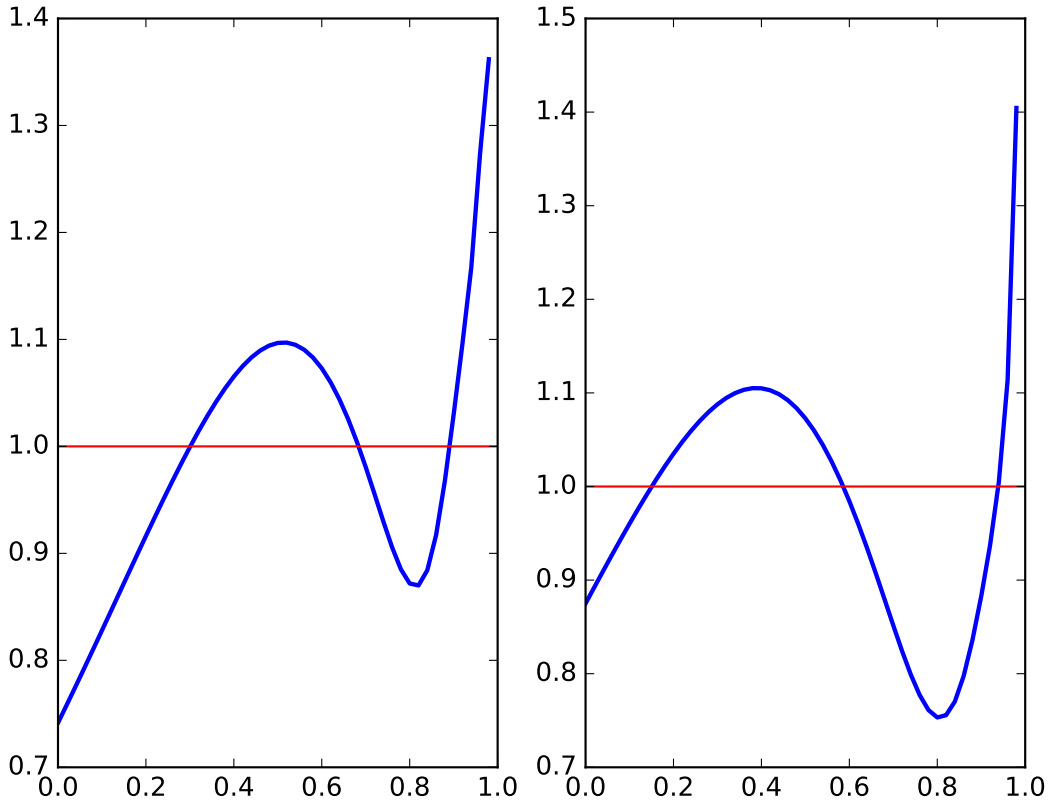


Fig. 1.24: Relative distributions, power law on the left, exponential on the right

better fit at the beginning and a worse fit for the tail, which gives rise to smaller discrepency in CDF and smaller K-S statistic. However, neither family seems to be good enough in terms of density characterization of distribution.

1.7.10 Mann-Whitney U test on distances

Given the distribution of distances, we perform a hypothesis test on the distributions of these distances. The goal is to identify some similarity on mice in the same strain when only looking at the distances covered every second.

To do so we chose a non parametric test since we only have access to sample distributions. The Kolmogorov Smirnov test is a very popular test used in this case but I chose to explore the Mann-Whitney U test instead for the following reasons:

- The KS test is sensitive to any differences in the two distributions. Substantial differences in shape, spread or median will result in a small P value. (see here for more details). Here we can feel that the distances have high variance. Therefore, the KS test would be too strict for our study.
- In contrast, the MW test is mostly sensitive to changes in the median, which is less sensitive of noise in the case of mice.

The MannWhitney U test is a test for assessing whether two independent samples come from the same distribution. The null hypothesis for this test is that the two groups have the same distribution, while the alternative hypothesis is that one group has larger (or smaller) values than the other.

- H_0: $P(X > Y) = P(Y > X)$
- H_1: $P(X > Y) \neq P(Y > X)$

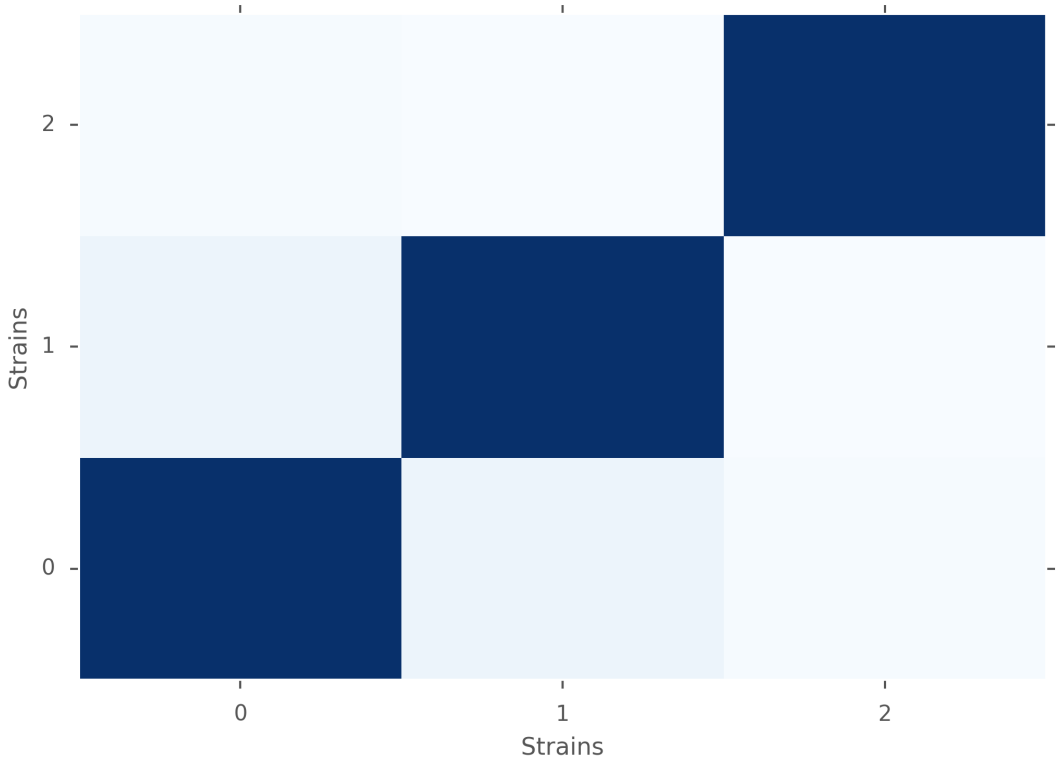


Fig. 1.25: MW U test p-values for different strains of mice. Dark blue is close to 1 and very light blue is close to zero.

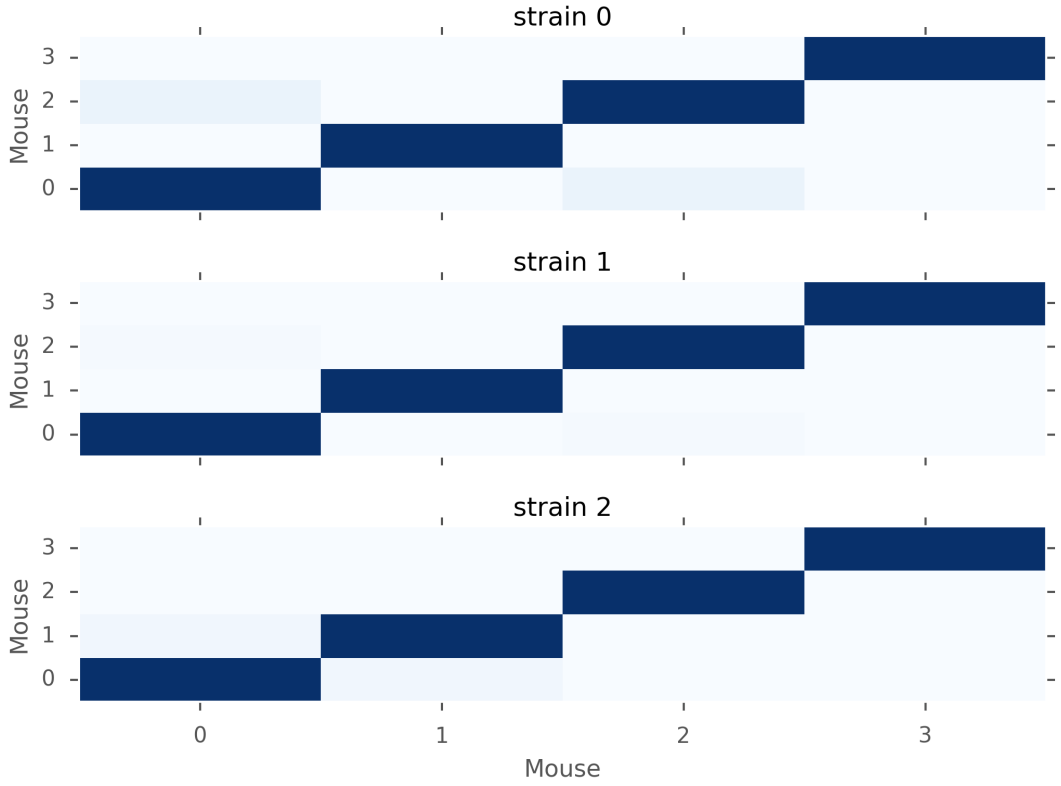


Fig. 1.26: MW U test p-values for mice within the same strains. Dark blue is close to 1 and very light blue is close to zero.

From the figures above, we notice that in terms of p-values, strain 1 is closer to strain 0 than strain 2. But we also notice that the p-values are still very low, which means that there is evidence for rejecting H_0. Moreover, when looking closer within each strain of mice, we can see that mice have different distributions too. Therefore, based on the inter-distances, we cannot conclude that mice behave similarly depending on their strains.

1.7.11 Future Work

Here are some further research we could do. However, because of Incomplete sample we have, we cannot do it for now, but it is easy to fix the function

- K means clustering:

One major goal of this project is to measure similarity between different strain and hopefully make clusters based on our data. But one difficulty is that we cannot plug in the information we have to a known machine learning clustering algorithm. However, as truncated exponential to be a good fit. We can use the parameters to measure the similarity and transform our sample data to one scalar. One classic unsupervised learning algorithm is K-means and we can definitely use it to make clusters. However, one drawback is the distance between our parameters is not uniform but as long as there exists significant difference, it will not harm that much.

1.7.12 Reference reading

- https://en.wikipedia.org/wiki/Power_law
- http://arxiv.org/pdf/0706.1062v2.pdf

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Contributing

2.1.1 Development process

Here's the long and short of it:

1. If you are a first-time contributor:
   - Go to https://github.com/berkeley-stat222/mousestyles and click the "fork" button to create your own copy of the project.
   - Clone the project to your local computer:

```
git clone git@github.com:*your-Github-username*/mousestyles.git
```

   - Add the upstream repository:

```
git remote add upstream git@github.com:berkeley-stat222/mousestyles.git
```

   - Now, you have remote repositories named:
     - `upstream`, which refers to the `berkeley-stat222/mousestyles` repository
     - `origin`, which refers to your personal fork
2. Develop your contribution:
   - Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

   - Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name based on the feature you want to improve or add such as '*feature*-issue#*number*' (e.g. `git checkout -b contributing-issue#14`):

```
git checkout -b contributing-issue#14
```

     See *tools/bash-and-git.sh*, which contains useful configurations for visualizing and keeping track of branches more easily.
   - Commit locally as you progress (`git add` and `git commit`). It is highly recommended that you use the provided pre-commit hook (described in the tools README), which automatically runs tests and style checks before every commit. This way, you can see if Travis (described below) will fail before you push.
3. To submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin contributing-issue#14
```

- Go to GitHub. The new branch will show up with a green Pull Request button - click it.
- If you want, comment on the issue posting in Github to explain your changes or to ask for review.

4. Review process:
   - Reviewers (the other developers and interested community members) will write inline and/or general comments on your Pull Request (PR) to help you improve its implementation, documentation and style. Every single developer working on the project has their code reviewed, and we've come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don't let the review discourage you from contributing: its only aim is to improve the quality of the project, not to criticize (we are, after all, very grateful for the time you're donating!). See here for some nice code review guidelines.
   - To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.
   - Travis-CI, a continuous integration service, is triggered after each Pull Request update to build the code, run unit tests, measure code coverage and check coding style (PEP8) of your branch. The Travis tests must pass before your PR can be merged. If Travis fails, you can find out why by clicking on the "failed" icon (red cross) and inspecting the build and test log.
5. Document changes

   Before merging your commits, you must add a description of your changes to the release notes of the upcoming version in `doc/release/release_dev.rst`.

Note: To reviewers: if it is not obvious, add a short explanation of what a branch did to the merge message and, if closing a bug, also add "Closes #14" where 14 is the issue number.

2.1.2 Divergence between `upstream master` and your feature branch

Do *not* ever merge the main branch into yours. If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, rebase onto master:

```
git checkout master
git pull upstream master
git checkout contributing-issue#14
git rebase master
```

If any conflicts occur in (e.g. conflict-file1 and conflict-file2), fix the according files and continue:

```
git add conflict-file1 conflict-file2
git rebase --continue
```

Help in resolving merge conflicts is provided here.

In some cases, you'll want to edit your history while rebasing. This can be accomplished with the `-i` or `--interactive` option of `git rebase`. Running `rebase` with this option will open a text editor, where you can choose to remove or edit some commits, or squash several together. This allows you to (for example) edit commit messages, or merge together repetitive small commits like "fixed typo." See this tutorial for more details.

Note: you should only rebase your own branches and must generally not rebase any branch which you collaborate on with someone else.

Finally, you must push your rebased branch:

```
git push --force origin contributing-issue#14
```

(If you are curious, here's a further discussion on the dangers of rebasing. Also see this LWN article.)

2.1.3 Guidelines

- All code should have tests (see *test coverage* below for more details).
- All code should be documented, to the same standard as NumPy and SciPy.
- No changes are ever committed without review. Ask on the mailing list if you get no response to your pull request. **Never merge your own pull request.**

2.1.4 Stylistic Guidelines

- Set up your editor to remove trailing whitespace. Follow PEP08. Check code with pyflakes / flake8.
- Use numpy data types instead of strings (`np.uint8` instead of `"uint8"`).
- Use the following import conventions:

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt

cimport numpy as cnp # in Cython code
```

2.1.5 Commit message codes

Please prefix all commit summaries with one (or more) of the following labels. This should help others to easily classify the commits into meaningful categories:

- *BUG* : bug fix
- *RFT* : refactoring
- *ENH* : new feature or extended functionality
- *BKW* : addresses backward-compatibility
- *OPT* : optimization
- *BRK* : breaks something and/or tests fail
- *DOC*: for all kinds of documentation related commits
- *TST* : for adding or changing tests
- *DAT* : for adding or changing data files
- *STY* : PEP8 conformance, whitespace changes etc that do not affect function.

So your commit message might look something like this:

```
TST: relax test threshold slightly

Attempted fix for failure on windows test run when arrays are in fact
very close (within 6 dp).
```

Keeping up a habit of doing this is useful because it makes it much easier to see at a glance which changes are likely to be important when you are looking for sources of bugs, fixes, large refactorings or new features.

2.1.6 Pull request codes

When you submit a pull request to github, github will ask you for a summary. If your code is not ready to merge, but you want to get feedback, please consider using `WIP - experimental optimization` or similar for the title of your pull request. That way we will all know that it's not yet ready to merge and that you may be interested in more fundamental comments about design.

When you think the pull request is ready to merge, change the title (using the *Edit* button) to something like `MRG - optimization`.

2.1.7 Test coverage

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, install coverage.py (e.g., using `pip install coverage`) and then run:

```
$ make coverage
```

This will print a report with one line for each file in *mousestyles*, detailing the test coverage:

```
Name                  Stmts   Miss Branch BrMiss  Cover   Missing
-----------------------------------------------------------------
mousestyles              43      6     10      1    87%   72, 77-88
mousestyles.core         55      0     30      4    95%
mousestyles.data         45      0      2      0   100%
mousestyles.eda          22      0      8      0   100%
mousestyles.irr          52      0     20      2    97%
mousestyles.stratified   44      0     16      4    93%
-----------------------------------------------------------------
TOTAL                   261      6     86     11    95%
----------------------------------------------------------------------
Ran 35 tests in 37.199s

OK
```

2.1.8 Bugs

Please report bugs on GitHub.

2.2 Documentation and Coding Standards

2.2.1 Naming Conventions

- Choose short and concise names that reflect usage
- Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names
- *Package and Module Names*:
- Use all-lowercase names

- Use underscores when appropriate in module names but not package names
- *Class Names*:
- Use `CapWords` convention
- *Function and Variable Names*:
- `lower_case_with_underscores`
- Use a descriptive verb in function names
- *Constants*:
- Use all capital letters with underscores separating words (e.g. `YELLOW_MARINE`)

2.2.2 Indentation

- Use 4 spaces per indentation level
- Continuation lines should align wrapped elements vertically using Python's implicit line joining inside parentheses, brackets and braces

```
    # Aligned with opening delimiter
    nacho_cheese = not_your_cheese(cheddar, pepperjack,
                                   provolone, brie)
```

- The closing brace/bracket/parenthesis on multi-line constructs should line up under the first non-whitespace character of the last line of list

```
    six_afraid_of_seven = [
        7, 8, 9,
        1, 0, 1
        ]
```

2.2.3 Docstrings

- We will be following numpy conventions.
- What is a Docstring?
- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the *_doc_* special attribute of that object.
- When do we use docstrings?
- All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the *_init_* constructor) should also have docstrings. A package may be documented in the module docstring of the *_init_*.py file in the package directory.
- Formatting by type of object
- The docstring of a **script** should be usable as its "usage" message, printed when the script is invoked with incorrect or missing arguments. Such a docstring should document the script's function and command line syntax, environment variables, and files. Usage messages should be sufficient for a new user to use the command properly, as well as a complete quick reference to all options and arguments for the sophisticated user.
- If the stand-alone script uses another module for handling options, such as the argparse module, then option information is moved from the docstring to the module's utilities.

- The docstring for a **module** should generally list the classes, exceptions, and functions that are exported by the module, with a one-line summary of each.
- The docstring for a **package** (i.e., the *_init_*.py module) should also list the modules and subpackages exported by the package.
- The docstring of a **function** or **method** is a phrase ending in a period. It prescribes the function or method's effect as a command ("Do this", "Return that"), not as a description; e.g. don't write "Returns the pathname ...". A multiline-docstring for a function or method should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called. Optional arguments should be indicated. It should be documented whether keyword arguments are part of the interface.
- The docstring for a **class** should summarize its behavior and list the public methods and instance variables. If the class is intended to be subclassed, and has an additional interface for subclasses, this interface should be listed separately. The class constructor should be documented in the docstring for its *_init_* method. Individual methods should be documented by their own docstring.
- Example docstring
- For consistency, always use `"""triple double quotes"""` around docstrings. Use `r"""raw triple double quotes"""` if you use any backslashes in your docstrings. For Unicode docstrings, use `u"""Unicode triple-quoted strings"""`.

```
    def foo(var1, var2, long_var_name='hi') :
        r"""A one-line summary that does not use variable names or the
        function name.

        Several sentences providing an extended description. Refer to
        variables using back-ticks, e.g. `var`.

        Parameters
        ----------
        var1 : array_like
            Array_like means all those objects -- lists, nested lists, etc. --
            that can be converted to an array.  We can also refer to
            variables like `var1`.
        var2 : int
            The type above can either refer to an actual Python type
            (e.g. ``int``), or describe the type of the variable in more
            detail, e.g. ``(N,) ndarray`` or ``array_like``.
        long_var_name : {'hi', 'ho'}, optional
            Choices in brackets, default first when optional.

        Returns
        -------
        type
            Explanation of anonymous return value of type ``type``.
        describe : type
            Explanation of return value named `describe`.
        out : type
            Explanation of `out`.

        Other Parameters
        ----------------
        only_seldom_used_keywords : type
            Explanation
        common_parameters_listed_above : type
            Explanation

        Raises
```

```
        ------
        BadException
            Because you shouldn't have done that.

        See Also
        --------
        otherfunc : relationship (optional)
        newfunc : Relationship (optional), which could be fairly long, in which
            case the line wraps here.
        thirdfunc, fourthfunc, fifthfunc

        Notes
        -----
        Notes about the implementation algorithm (if needed).

        This can have multiple paragraphs.

        You may include some math:

        .. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}

        And even use a greek symbol like :math:`omega` inline.

        References
        ----------
        Cite the relevant literature, e.g. [1]_.  You may also cite these
        references in the notes section above.

        .. [1] O. McNoleg, "The integration of GIS, remote sensing,
            expert systems and adaptive co-kriging for environmental habitat
            modelling of the Highland Haggis using object-oriented, fuzzy-logic
            and neural-network techniques," Computers & Geosciences, vol. 22,
            pp. 585-588, 1996.

        Examples
        --------
        These are written in doctest format, and should illustrate how to
        use the function.

        >>> a=[1,2,3]
        >>> print [x + 3 for x in a]
        [4, 5, 6]
        >>> print "a\n\nb"
        a
        b

        """

        pass

        ...
```

2.2.4 Comments

- Use complete sentences; if a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lowercase letter

- Use two spaces after a sentence-ending period
- Indent block comments to the same level as the code below; start each line of a block comment with a # and a single space (unless it is indented text inside the comment)
- Separate paragraphs inside a block comment by a line containing a single #

`python # How dows a mouse feel after it takes a shower?  # Squeaky clean. # # How do you save a drowning mouse?  # Use mouse to mouse resuscitation. not_your_cheese(marscapone, maasdam, camembert, roquefort)` - Use inline comments sparingly

`python not_your_cheese(gorgonzola, munster, limburger, doppelrhamstufel) # Not your cheese, my cheese.`

2.2.5 Blank Lines

- Surround top-level function and class definitions with two blank lines
- Surround method definitions inside a class with a single blank line
- Use extra blank lines sparingly to separate groups of related functions; omit blank lines between related one-liners (e.g. a set of dummy implementations) if desired
- Use blank lines in functions sparingly to indicate logical sections

2.2.6 Whitespace

- Avoid extraneous whitespace:
- Immediately inside parentheses, brackets, or braces
- Immediately before a comma, semicolon, or colon
- Immediately before the open parenthesis that starts the argument list of a function call
- Immediately before the open parenthesis that starts an indexing or slicing
- More than one space around an assignment (or other) operator to align it with another
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+= , -= etc.), comparisons (== , < , > , != , <> , <= , >= , in , not in , is , is not), Booleans (and , or , not)
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies); never use more than one space, and always have the same amount of whitespace on both sides of a binary operator

`python eyes = e + yes cyclops = eyes*4 - 3 c = (a+b) * (a-b)` - Do not use spaces around the = sign when used to indicate a keyword argument or a default parameter value

```
```python
```

def not_your_cheese(cheese1=smelly, cheese2=stinky, cheese3=noxious, cheese4=bad) return my_cheese(r=real, i=imag) ‘‘‘ - Avoid compound statements (multiple statements on the same line) - Never put an if/for/while with a multi-clause statement on the same line
```

2.2.7 String Quotes

- Generally, use double-quotes for strings, but if a string contains a double-quote, then use single quotes
- Keep string quotes consistent for readability

2.2.8 Imports

- Put imports on separate lines, e.g.:

`python import os import sys from subprocess import Popen, PIPE` - Put imports at the top of the file, just after any module comments and docstrings, and before module globals and constants - Avoid wildcard imports (from import *) as they make it unclear which names are present in the namespace, confusing both readers and many automated tools

2.2.9 Maximum Line Length

- Limit all lines to a maximum of 79 characters
- Use backslashes when implicit continuation fails

```
python with open("/why/did/the/chicken/cross/the/road") as
chicken, \ open("/to/get/to/the/other/side", "w") as waffles:
waffles.write(chicken.read())
```

For more information, please refer to: Style Guide for Python Code and Docstring Guide

2.3 Writing tests

We need to create unit tests for each module and test each function of the module.

2.3.1 An introductory example

Suppose we have a file called *data_utils.py* with two functions `func1` and `func2`, we should have a corresponding test file called *test_data_utils.py* with two corresponding test functions called `test_func1` and `test_func2` and *data_utils.py* imported.

- Note that we use `pytest` in our framework. Always import pytest at the top of the test file.
- Please follow the naming convention: *test_*.py* for test file and *test_** for test functions.

```
# content of mousestyles/mousestyles/utils.py
def func1():
   return 3

def func2(x):
   return x + 1
```

```
# content of mousestyles/mousestyles/tests/test_utils.py
from __future__ import (absolute_import,
                        division, print_function,
                        unicode_literals)

import pytest
```

```
from mousestyles.utils import func1, func2

def test_func1():
   assert f() == 3

def test_func2():
   assert func(3) == 4
```

2.3.2 Run all the tests for mousestyles

- For our `mousestyles` package, you can run all the tests with `make test`. Note that `make test` always run make `make install`, thus you do not need to install beforehand.
- Run `make test-fast` for quicker (but less foolproof) development.
- If you want to look at the details, code for `make` command can be found here.

2.3.3 Pytest: basic usage

Installation:

```
pip install -U pytest
```

Run all files in the current directory and its subdirectories of the form *test_*.py* or **_test.py*:

```
$ py.test
```

Or you may run one test file with or without "quiet" reporting mode:

```
$ py.test test_data_utils.py
$ py.test -q test_data_utils.py
```

2.3.4 Pytest: writing and reporting of assertions

`pytest` allows you to use the standard python `assert` for verifying expectations and values. For example:

```
# contents of example1.py
def f():
   return 3

def test_function():
   assert f() == 4
```

If we run `example1.py` using `py.test example1.py`, it will fail because the expected value is 4, but `f` return 3.

We can also specify a message with the assertion like this:

```
assert a % 2 == 0, "value was odd, should be even"
```

In order to write assertions about raised exceptions, you can use pytest.raises as a context manager like this:

```
import pytest

def test_zero_division():
   with pytest.raises(ZeroDivisionError):
```

```
    1 / 0

def test_exception():
    with pytest.raises(Exception):
        x = 1 / 0
```

See Built-in Exceptions for more about raising errors.

If you need to have access to the actual exception info you may use:

```
def test_recursion_depth():
    with pytest.raises(RuntimeError) as excinfo:
        def f():
            f()
        f()
 assert 'maximum recursion' in str(excinfo.value)
```

Pytest also support expected warnings, see pytest.warn

For more about assertions, see Assertions in pytest

2.3.5 What aspects of a function need to be tested:

- check if it returns correct type of object
- check if it returns correct dimension
- check if it returns the correct value, by
  - prior knowledge
  - different implementation: i.e. use R vs Python; use $Var(X)$ vs. $E(X^2) - E(X)^2$
  - theoretical derivation
- "regression": if the function is improved (by speed, efficiency) while has same functionality, check the output is the same with older version.
- assert errors occured: i.e. when the function takes three arguements while we only give two, make sure the function will throw an error message

2.3.6 Reference

Most example above comes from Pytest documentation. See Pytest Documentation for more detail.

2.4 Set up SSH key

SSH, also known as Secure Socket Shell, is a network protocol that provides us with a secure way to access a remote computer. Once you set up your SSH key and add it to your Github account, you don't have to type username and password when you push your commits to the remote repository.

1. Checking for existing SSH keys:
   - Go to your local repository and check whether you have SSH key or not:

```
ls -al ~/.ssh
```

   - If you already have the SSH key, you will see the one of the following filenames for the public SSH key:

  - `id_rsa.pub`, `id_ecdsa.pub`, `id_ed25519.pub`, or `id_rsa.pub`
- If you don't see SSH key or don't want to use the one you have, you can generate a new one (see 2).

2. Generating a new SSH key and adding it to the ssh-agent:
   - Generate a SSH key:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

   - After you paste the above command to the terminal, you will get the following message:
     - `Enter a file in which to save the key (/Users/you/.ssh/id_rsa)` Hit enter to continue.
     - `Enter passphrase (empty for no passphrase)` Add passphrase if you want; otherwise, hit enter to continue.
     - `Enter same passphrase again` Type the passphrase again or hit enter to continue.
   - You will see:
     - `Your identification has been saved in /home/oski/.ssh/id_rsa.`
     - `Your public key has been saved in /home/oski/.ssh/id_rsa.pub.`, which means that you have SSH key now!
   - Add it to the ssh-agent:

```
eval $(ssh-agent -s)
ssh-add ~/.ssh/id_rsa
```

   - Now, you are ready to add the SSH key to Github.
3. Adding a new SSH key to your Github account:
   - Copy the SSH key to your clipboard:

```
clip < ~/.ssh/id_rsa.pub
```

   - Or you can use and copy the output:

```
cat ~/.ssh/id_rsa.pub
```

   - Go to Github and click your profile in the top-right corner.
   - Click the SSH and GPG keys on the right panel and click New SSH key.
   - Give a label to this SSH key, paste the key into Key field and click Add SSH key.
4. Testing your SSH connection:
   - Enter:

```
ssh -T git@github.com
```

   - You will see
     - `The authenticity of host 'github.com (192.30.252.1)' can't be established. RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a Are you sure you want to continue connecting (yes/no)?`
   - Type yes and you should see:
     - `Hi username! You've successfully authenticated, but GitHub does not provide shell access.`

5. Switching remote URLs from HTTPS to SSH:

   - Since you probably already add your remote URL using HTTPS (check by `git remote -v`, you need to change it into SSH:

```
git remote set-url origin git@github.com:USERNAME/OTHERREPOSITORY.git
```

   - You can verify again by `git remote -v` and next time you push your commits to the origin, you don't have to type username and password anymore!

For more detailed discussion, read SSH key documentation on Github.

CHAPTER

THREE

API REFERENCE

3.1 mousestyles package

3.1.1 Subpackages

mousestyles.behavior package

Submodules

mousestyles.behavior.behavior module

Behavior Analysis

mousestyles.behavior.behavior_tree module

class mousestyles.behavior.behavior_tree.**BehaviorTree**(*structure=None*, *contents=None*, *units=None*)

Object representing a decomposed tree of behavior. Behaves much like a dictionary, with an additional representation of the tree structure of the data.

Parameters

- **structure** (*list, optional*) – Representation of parent-child relationships between nodes
- **contents** (*defaultdict, optional*) – Initial node values
- **units** (*defaultdict, optional*) – Mapping of node names to units

Examples

```
>>> tree = BehaviorTree(['Consumption',
                         ['AS Prob', 'Intensity']])
>>> tree['Consumption'] = 5
>>> tree['AS Prob'] = 1
>>> tree['Intensity'] = 2
>>> tree.contents
defaultdict(float, {'AS Prob': 1, 'Consumption': 5, 'Intensity': 2})
>>> tree.structure
['Consumption', ['AS Prob', 'Intensity']]
>>> print(tree)
```

Consumption: 5 AS Prob: 1 Intensity: 2

copy()
Return a copy of the tree.

static merge(**args*)
Merge several trees into one compound tree.

Parameters ***args** (*variable-length argument list of BehaviorTree objects*) – One or more trees to be merged.

Returns **new_tree** – Tree with same structure as input trees, where each node is a list containing the node values from the input trees.

Return type *BehaviorTree*

Examples

```
>>> tree = BehaviorTree(contents={'a': 1, 'b': 2})
>>> tree2 = BehaviorTree(contents={'a': -1.5, 'b': 20})
>>> BehaviorTree.merge(tree, tree2).contents
defaultdict(list, {'a': [1, -1.5], 'b': [2, 20]})
```

summarize(*f*)
Computes a summary statistic for the nodes in a tree.

Parameters **f** (*function*) – A function to be applied to the contents in each node. Most often, this will take in a list and return a single number.

Returns

- *A new tree with the same structure, where each node value tree[k]*
- *is replaced by f(tree[k]).*

mousestyles.behavior.construct_trees module

Construct behavior trees

mousestyles.behavior.construct_trees.**compute_tree**(*feature*, *strain*, *mouse*, *day*, *epsilon=1*)
Compute and return a tree decomposition of a feature for a single mouse-day

Parameters

- **feature** (*{'F', 'W', 'L'}*) – The feature used. 'F' for food, 'W' for water, 'L' for locomotion.
- **strain** (*int*) – Integer representing the strain of the mouse
- **mouse** (*int*) – Integer representing the specific mouse
- **day** (*int*) – Integer representing the day to produce the metrics for
- **epsilon** (*float, optional*) – tolerance for merging events into bouts

Returns **behavior** – dictionary-like object representing breakdown of consumption into various nodes

Return type *BehaviorTree*

mousestyles.behavior.construct_trees.**process_raw_intervals**(*feature*, *consumption*, *intervals*, *active_time*, *total_time*, *epsilon*)

Takes *total_consumption* and the data in *intervals* and decomposes it into a tree of behavioral summary statistics.

Parameters

- **feature** (*{'F', 'W', 'L'}*) – The feature used. 'F' for food, 'W' for water, 'L' for locomotion.
- **consumption** (*float*) – total consumption (food, water, or movement) in the time period
- **intervals** (Intervals) – Intervals object representing intervals of behavior
- **active_time** (*float*) – amount of time the mouse was active in the day
- **total_time** (*float*) – total amount of recording time in the day
- **epsilon** (*float*) – tolerance for merging events into bouts

Returns **behavior** – dictionary-like object representing breakdown of consumption into various nodes

Return type *BehaviorTree*

Examples

```
>>> from mousestyles import data, intervals
>>> ints = intervals.Intervals(data.load_intervals('AS'))
>>> results = process_raw_intervals(1000, ints, 1)
```

mousestyles.behavior.metrics module

Metrics for behavior

mousestyles.behavior.metrics.**active_time**(*strain*, *mouse*, *day*)

Returns the total amount of active time recorded for a certain mouse-day.

Parameters

- **strain** (*int*) –
- **mouse** (*int*) –
- **day** (*int*) –

Returns

Return type The total amount of active time in seconds for the specified mouse-day.

mousestyles.behavior.metrics.**create_collapsed_intervals**(*intervals*, *bout_threshold*)

Returns a collapsed interval object (within a given threshold) on a certain mouse-strain-day for a given activity type.

Parameters

- **intervals** (*interval object on a certain mouse-strain-day*) –
- **a given activity type** (*for*) –
- **bout_threshold** (*: float*) – Float representing the time threshold to use for collapsing separate events into bouts

Returns

- An intervals object *intervals* for the given activity type and
- *mouse-strain-day within a bout threshold*

Examples

```
>>> ints=behavior.create_intervals('F', strain = 0
                                   , mouse = 0, day = 0
                                   , bout_threshold = 0.001)
>>> collapsed_ints=behavior.create_collapsed_intervals(intervals = ints
                                                       , bout_threshold = 0.001)
>>> print(collapsed_ints)
```

mousestyles.behavior.metrics.**create_intervals**(*activity_type*, *strain*, *mouse*, *day*)

Returns an interval object on a certain mouse-strain-day for a given activity type.

Parameters

- **activity_type** (*str*) – String specifying activity type {"AS", "F", "IS", "M_AS", "M_IS", "W"}
- **strain** (*int*) – Integer representing the strain of the mouse
- **mouse** (*int*) – Integer representing the specific mouse
- **day** (*int*) – Integer representing the day to produce the metrics for
- **bout_threshold** (*: float*) – Float representing the time threshold to use for collapsing separate events into bouts

Returns

- An intervals object *intervals* for the given activity type and
- *mouse-strain-day within a bout threshold*

Examples

```
>>> ints=behavior.create_intervals(activity_type = 'F', strain = 0
                                   , mouse = 0, day = 0
                                   , bout_threshold = 0.001)
>>> print(sum(ints))
```

mousestyles.behavior.metrics.**total_amount**(*strain*, *mouse*, *day*, *feature*)

Returns the total amount consumed/moved for one of the following features: food ("F"), water ("W"), and locomotion ("L").

Parameters

- **strain** (*int*) –
- **mouse** (*int*) –
- **day** (*int*) –
- **feature** (*{'F', 'W', 'L'}*) – The feature used. 'F' for food, 'W' for water, 'L' for locomotion.

Returns

Return type The total amount consumed/moved for the given feature in the mouse-day.

mousestyles.behavior.metrics.**total_time**(*strain*, *mouse*, *day*)
Returns the total amount of time recorded for a certain mouse-day.

Parameters

- **strain** (*int*) –
- **mouse** (*int*) –
- **day** (*int*) –

Returns

Return type The total amount of time in seconds of the specified mouse-day.

Module contents

mousestyles.classification package

Submodules

mousestyles.classification.classification module

mousestyles.classification.classification.**fit_gradient_boosting**(*train_y*, *train_x*, *test_x*, *n_estimators=None*, *learning_rate=None*)
Returns a DataFrame of Gradient Boosting results, containing prediction strain labels and printing the best model. The model's parameters will be tuned by cross validation, and accepts user-defined parameters. :param train_y: labels of classification results, which are predicted strains. :type train_y: pandas.Series :param train_x: features used to predict strains in training set :type train_x: pandas.DataFrame :param test_x: features used to predict strains in testing set :type test_x: pandas.DataFrame :param n_estimators: tuning parameter of GradientBoosting, which is the number of

boosting stages to perform

Parameters **learning_rate** (*list, optional*) – learning_rate shrinks the contribution of each tree learning_rate

Returns **GradientBoosting results** – Prediction strain labels

Return type pandas.DataFrame

mousestyles.classification.classification.**fit_random_forest**(*train_y*, *train_x*, *test_x*, *n_estimators=None*, *max_feature=None*, *importance_level=None*)
Returns a DataFrame of RandomForest results, containing prediction strain labels and printing the best model. The model's parameters will be tuned by cross validation, and accepts user-defined parameters. :param train_y: labels of classification results, which are predicted strains. :type train_y: pandas.Series :param train_x: features used to predict strains in training set :type train_x: pandas.DataFrame :param test_x: features used to predict

strains in testing set :type test_x: pandas.DataFrame :param n_estimators: tuning parameter of RandomForest, which is the number of

trees in the forest

Parameters

- **max_feature** (*list, optional*) – tuning parameter of RandomForest, which is the number of features to consider when looking for the best split
- **importance_level** (*int, optional*) – the minimum importance of features

Returns **RandomForest results** – The first element is the dataframe of prediction strain labels. The second element is the list of tuples of score and important features larger than the importance level.

Return type list

mousestyles.classification.classification.**fit_svm**(*train_y*, *train_x*, *test_x*, *c=None*, *gamma=None*)

Returns a DataFrame of svm results, containing prediction strain labels and printing the best model. The model's parameters will be tuned by cross validation, and accepts user-defined parameters. :param train_y: labels of classification results, which are predicted strains. :type train_y: pandas.Series :param train_x: features used to predict strains in training set :type train_x: pandas.DataFrame :param test_x: features used to predict strains in testing set :type test_x: pandas.DataFrame :param c: tuning parameter of svm, which is penalty parameter of the error term :type c: list, optional :param gamma: tuning parameter of svm, which is kernel coefficient :type gamma: list, optional

Returns **svm results** – Prediction strain labels

Return type pandas.DataFrame

mousestyles.classification.classification.**get_summary**(*predict_labels*, *true_labels*)

Returns a DataFrame of classification result summary, including precision, recall, F1 measure in terms of different strains. :param predict_labels: prediction strain labels :type predict_labels: pandas.DataFrame :param true_labels: true strain labels, used to measure the prediction

accuracy

Returns **classification result summary** – 16 rows, for each strain 0-15 Column 0: precision Column 1: recall Column 2: F-1 measure

Return type pandas.DataFrame, shape (16,3).

mousestyles.classification.classification.**prep_data**(*strain*, *features*, *rseed=222*)

Returns a list of 4: [train_y, train_x, test_y, test_x] train_y: pandas.Series of strain labels in train data sets, train_x: pandas.DataFrame of features in train data sets, test_y: pandas.Series of strain labels in test data sets, test_x: pandas.DataFrame of features in train data sets

Parameters

- **strain** (*pandas.Series*) – classification labels
- **features** (*pandas.DataFrame*) – classification features
- **rseed** (*int, optional*) – random seed for shuffling the data set to separate train and test

Returns **splitted data** – A list of 4 as explained above

Return type list

mousestyles.classification.clustering module

mousestyles.classification.clustering.**cluster_in_strain**(*labels_first*, *labels_second*)

Returns a dictionary object indicating the count of different clusters in each different strain (when put cluster labels as first) or the count of different strain in each clusters (when put strain labels as first).

Parameters

- **labels_first** (*numpy arrary or list*) – A numpy arrary or list of integers representing which cluster the mice in, or representing which strain mice in.
- **labels_second** (*numpy arrary or list*) – A numpy array or list of integers (0-15) representing which strain the mice in, or representing which cluster the mice in

Returns **count_data** – A dictioanry object with key is the strain number and value is a list indicating the distribution of clusters, or the key is the cluster number and the value is a list indicating the distribution of each strain.

Return type dictionary

Examples

```
>>> count_1 = cluster_in_strain([1,2,1,0,0],[0,1,1,2,1])
```

mousestyles.classification.clustering.**fit_hc**(*mouse_day_X*, *method*, *dist*, *num_clusters=[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]*)

Returns a list of 2: [silhouettes, cluster_labels] silhouettes: list of float, cluster_labels: list of list,

each sublist is the labels corresponding to the silhouette

Parameters

- **mouse_day_X** (*a 170 * M numpy array,*) – all columns corresponding to feature avg/std of a mouse over 16 days
- **method** (*str,*) – method of calculating distance between clusters
- **dist** (*str,*) – distance metric
- **num_clusters** (*range*) – range of number of clusters

Returns **A list of 2**

Return type [silhouettes, cluster_labels]

mousestyles.classification.clustering.**get_optimal_fit_kmeans**(*mouse_X*, *num_clusters*, *raw=False*)

Returns a list of 2: [silhouettes, cluster_labels] silhouettes: list of float, cluster_labels: list of list,

each sublist is the labels corresponding to the silhouette

Parameters

- **mouse_X** (*a 170 * M numpy array or 21131 * M numpy array,*) – all columns corresponding to feature avg/std of a mouse over 16 days or the raw data without averaging over days

- **num_clusters** (*range or a list or a numpy array*) – range of number of clusters
- **raw** (*a boolean with default is False*) – False if using the 170 * M array

Returns A list of 2

Return type [silhouettes, cluster_labels]

mousestyles.classification.clustering.**get_optimal_hc_params**(*mouse_day*)

Returns a list of 2: [method, dist] method: {'ward', 'average', 'complete'} dist: {'cityblock', 'euclidean', 'chebychev'}

Parameters **mouse_day** (*a 170 * M numpy array,*) – column 0 : strain, column 1: mouse, other columns corresponding to feature avg/std of a mouse over 16 days

Returns **method_distance** – [method, dist]

Return type list

mousestyles.classification.clustering.**prep_data**(*mouse_data*, *melted=False*, *std=True*, *rescale=True*)

Returns a ndarray data to be used in clustering algorithms: column 0 : strain, column 1: mouse, other columns corresponding to feature avg/std of a mouse over 16 days

that may or may not be rescaled to the same unit as specified

Parameters

- **mouse_data** –

  1. a 21131 * (4 +) pandas DataFrame, column 0 : strain, column 1: mouse, column 2: day, column 3: hour, other columns corresponding to features

  or (ii) a 1921 * (3 +) pandas DataFrame,

  column 0: strain, column 1: mouse, column 2: day, other columns corresponding to features

- **melted** (*bool,*) – False if the input mouse_data is of type (i)
- **std** (*bool,*) – whether the standard deviation of each feature is returned
- **rescale** (*bool,*) – whether each column is rescaled or not (rescale is performed by the column's maximum)

Returns

Return type The ndarray as specified

Module contents

mousestyles.data package

Submodules

mousestyles.data.utils module

Data utilities.

mousestyles.data.utils.**day_to_mouse_average**(*features*, *labels*, *num_strains=16*, *stdev=False*, *stderr=False*)

first three columns of labels are strain num, mouse number, day number features is an M x N matrix of mouse day x features

Returns new data matrix with a mean and stdev/stderr for each mouse over mouse days

mousestyles.data.utils.**idx_restrict_to_rectangles**(*TXY*, *rects=[(0, 0)]*, *xlims=(0, 1)*, *ylims=(0, 1)*, *xbins=2*, *ybins=4*, *eps=0.01*)

given (3 x T) TXY with 0th row array of times [ASSUMED SORTED] and rows 1,2 are x,y coords

returns new interval array which is E minus those things occuring outside of given rectangle

mousestyles.data.utils.**map_xbins_ybins_to_cage**(*rectangle=(0, 0)*, *xbins=2*, *ybins=4*, *YLower=1.0*, *YUpper=43.0*, *XUpper=3.75*, *XLower=-16.25*)

converts a rectangle in xbins x ybins to corresponding rectangle in Cage coordinates

format is [[p1, p2], [p3, p4]] where pi = (cage_height_location, cage_length_location)

??? -chris ### THIS GIVES WRONG CAGE LOCATIONS for top bottom left right # # # xbins ybins do NOT reflect cage geometry perfectly

mousestyles.data.utils.**mouse_to_strain_average**(*features*, *labels*, *num_strains=16*, *stdev=False*, *stderr=False*)

first two columns of M x N data matrix are strain num (1 - num_strains), mouse number other columns are features

Returns: new data matrix with a mean and stdev/stderr for each strain over mice

mousestyles.data.utils.**pull_locom_tseries_subset**(*M*, *start_time=0*, *stop_time=300*)

given an (m x n) numpy array M where the 0th row is array of times [ASSUMED SORTED]

returns a new array (copy) that is a subset of M corresp to start_time, stop_time

returns [] if times are not in array

(the difficulty is that if mouse does not move nothing gets registered so we should artificially create start_time, stop_time movement events at boundries)

mousestyles.data.utils.**split_data_in_half_randomly**(*features*, *labels*)

given an array of the form: features = M x A x B x C x ...

where M is the number of mouse days

and an array labels for this data of the form: labels = M x 2

where labels[:, 0] are strain numbers and the labels[:, 1] are mice numbers

returns bootstrap_data_1 = a random half of the mouse days bootstrap_labels_1 bootstrap_data_2 = the other half bootstrap_labels_2

mousestyles.data.utils.**total_time_rectangle_bins**(*M*, *xlims=(0, 1)*, *ylims=(0, 1)*, *xbins=5*, *ybins=10*)

given an (3 x n) numpy array M where the 0th row is array of times [ASSUMED SORTED]

returns a new (xbins x ybins) array (copy) that contains PDF of location over time

Module contents

mousestyles.data.**distances**(*strain*, *mouse*, *day*, *step=50*)

Return a numpy array object of project movement data for the specified combination of strain, mouse and day.

At regular timesteps, defined by the step parameter, compute the euclidian distance between the positions of the mouse at two consecutive times.

More specifically:

•let delta_t be the step parameter.

•let t_n be the sequance of non negative numbers such that $t_0 = 0$ and $t_(n+1) = t_n + delta_t$. The sequence is defined for all n such that $n >= 0$ and $t_n <= time$ of the experiment

•let d_n be the sequence of non negative numbers such that $d_0 = 0$ and d_n equals the position of the mouse at a particular day at time t_n. d_n is then defined on the same set of integers as the sequence t_n.

•The function returns the sequence d_n.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **day** (*int*) – nonnegative integer indicating the day number
- **step** (*float*) – positive float defining the time between two observations default corresponds to 1 second

Returns **movement**

Return type numpy array

Examples

```
>>> dist = distances(0, 0, 0, step=1e2)
```

mousestyles.data.**distances_bymouse**(*strain*, *mouse*, *step=50*, *verbose=False*)

Aggregates 'distances' for all days of recorded data for one particular mouse.

More specifically:

•let $d^1, ..., d^D$ be the sequence of distances for one particular mouse for days 1 to D.

•The function returns the concatenation of the d^i.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **step** (*float*) – positive float defining the time between two observations default corresponds to 1 second

Returns **movement**

Return type numpy array

Examples

```
>>> dist = distances_bymouse(0, 0, step=1e2)
```

mousestyles.data.**distances_bystrain**(*strain*, *step=50*, *verbose=False*)

Aggregates distances_bymouse for all mice in one given strain.

More specifically:

•let $d^1, ..., d^M$ be the sequence of distances for one particular strain for mouses 1 to M.

•The function returns the sequence concatenation of the d^i.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **step** (*float*) – positive float defining the time between two observations default corresponds to 1 second

Returns **movement**

Return type numpy array

Examples

```
>>> dist = distances_bystrain(0, step=1e2)
```

mousestyles.data.**load_all_features**()

Returns a (21131, 13) size pandas.DataFrame object corresponding to 9 features over each mouse's 2-hour time bin. The first four columns index each mouses's 2-hour bin:

Column 0: the strain of the mouse (0-15) Column 1: the mouse number (number depends on strain) Column 2: the day number (5-16) Column 3: the 2-hour time bin (e.g., value 4 corresponds to hours 4 to 6)

The remaining 9 columns are the computed features.

Returns **features_data_frame** – A dataframe of computed features.

Return type pandas.DataFrame

mousestyles.data.**load_intervals**(*feature*)

Return a pandas.DataFrame object of project interval data for the specified feature.

There are 5 columns in the dataframe: strain: the strain number of the mouse mouse: the mouse number in its strain day: the day number start: the start time stop: the stop time

Parameters **feature** (*{"AS", "F", "IS", "M_AS", "M_IS", "W"}*) –

Returns **intervals** – All data of the specified feature as a dataframe

Return type pandas.DataFrame

Examples

```
>>> AS = load_intervals('AS')
>>> IS = load_intervals('IS')
```

mousestyles.data.**load_mouseday_features**(*features=None*)

Returns a (1921, 3+11*n) size pandas.DataFrame object corresponding to each 2-hour time bin of the n inputted features over each mouse. The first three columns index each mouse:

Column 0: the strain of the mouse (0-15) Column 1: the mouse number (number depends on strain) Column 2: the day number (5-16)

The remaining 3*n columns are the values for each 2-hour time bin of the n inputted features.

Parameters **features** (*list, optional*) – A list of one or more features chosen from {"AS-Probability", "ASNumbers", "ASDurations", "Food", "Water", "Distance", "ASFoodIntensity", "ASWaterIntensity", "MoveASIntensity"} Default all features when optional

Returns **features_data_frame** – A dataframe of computed features.

Return type pandas.DataFrame

Examples

```
>>> mouseday = load_mouseday_features()
>>> mouseday = load_mouseday_features(["Food"])
>>> mouseday = load_mouseday_features(["Food", "Water", "Distance"])
```

mousestyles.data.**load_movement**(*strain*, *mouse*, *day*)

Return a pandas.DataFrame object of project movement data for the specified combination of strain, mouse and day.

There are 4 columns in the dataframe: t: Time coordinates (in seconds) x: X coordinates indicating the left-right position of the cage y: Y coordinates indicating the front-back position of the cage isHB: Boolean indicating whether the point is in the home base or not

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **day** (*int*) – nonnegative integer indicating the day number

Returns **movement** – CT, CX, CY coordinates and home base status of the combination of strain, mouse and day

Return type pandas.DataFrame

Examples

```
>>> movement = load_movement(0, 0, 0)
>>> movement = load_movement(1, 2, 1)
```

mousestyles.data.**load_movement_and_intervals**(*strain, mouse, day, features=[u'AS', u'F', u'IS', u'M_AS', u'M_IS', u'W']*)

Return a pandas.DataFrame object of project movement and interval data for the specified combination of strain, mouse and day.

There are 4 + len(features) columns in the dataframe: t: Time coordinates (in seconds) x: X coordinates indicating the left-right position of the cage y: Y coordinates indicating the front-back position of the cage isHB: Boolean indicating whether the point is in the home base or not Additonal columns taking their names from features: Boolean indicating whether the time point is in an interval of behavior of the given feature.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **day** (*int*) – nonnegative integer indicating the day number
- **features** (*list (or other iterable) of strings*) – list of features from {"AS", "F", "IS", "M_AS", "M_IS", "W"}

Returns **movement** – coordinates, home base status, and feature interval information for a given srain, mouse and day

Return type pandas.DataFrame CT, CX, CY

Examples

```
>>> m1 = load_movement(1, 1, 1)
>>> m2 = load_movement_and_intervals(1, 1, 1, []) # don't add any features
>>> np.all(m1 == m2)
True
>>> m3 = load_movement_and_intervals(1, 1, 1, ['AS'])
>>> m3.shape[1] == m1.shape[1] + 1 # adds one column
True
>>> m3.shape[0] == m1.shape[0] # same number of rows
True
>>> m3[29:32]
             t      x       y   isHB     AS
29  56448.333 -6.289  34.902  False  False
30  56448.653 -5.509  34.173   True   True
31  56449.273 -5.048  33.284   True   True
```

mousestyles.data.**load_start_time_end_time**(*strain*, *mouse*, *day*)

Returns the start and end times recorded for the mouse-day. The first number indicates the number of seconds elapsed since midnight, the second number indicates when the cage is closed for cleaning. In other words, this is the interval for which all sensors are active.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **day** (*int*) – nonnegative integer indicating the day number

Returns **times** – the start time and end time

Return type a tuple of (float, float)

mousestyles.data.**load_time_matrix_dynamics**()

Load the time matrix for dynamics pattern project

mousestyles.distributions package

Submodules

mousestyles.distributions.kolmogorov_test module

mousestyles.distributions.kolmogorov_test.**get_travel_distances**(*strain=0*, *mouse=0*, *day=0*)

Get distances travelled in 20ms for this strain, this mouse, on this day.

Parameters

- **strain** (*int {0, 1, 2}*) – The strain of mouse to test
- **mouse** (*int {0, 1, 2, 3}*) – The mouse twin id with in the strain

- **day** (*int {0, 1, ..., 11}*) – The day to calculate the distance

Returns

- **x** (*np.ndarray shape (n, 1)*) – The distances travelled in 20ms for this mouse on this day, truncated at 1cm (i.e. only record mouse movement when it moves more than 1cm)
- *Examples*
- **>>> get_travel_distances(0, 0, 0)[** (*3]*)
- *array([1.00648944, 1.02094319, 1.0178885])*

mousestyles.distributions.kolmogorov_test.**perform_kstest**(*x*, *distribution=<scipy.stats._continuous_distns.pareto_gen object>*, *verbose=True*)

This function fits a distribution to data, and then test the fit of the distribution using Kolmogorov-Smirnov test.

The Kolmogorov-Smirnov test constructs the test statistic, which is defined as $\sup |F_n(x) - F(x)|$, for F_n is the sample CDF, and F is the theoretical CDF. This statistics can be considered as a measure of distance between the sample distribution and the theoretical distribution. The smaller it is, the more similar the two distributions.

We first estimate the parameter using MLE, then by minimizing the KS test statistic.

The Pareto distribution is sometimes known as the Power Law distribution, with the PDF: $b/x * *(b + 1)$ for $x >= 1, b > 0$. The truncated exponential distribution is the same as the rescaled exponential distribution.

Parameters

- **x** (*np.ndarray (n,)*) – The sample data to test the distribution
- **distribution** (*A Scipy Stats Continuous Distribution*) – {stats.pareto, stats.expon, stats.gamma} The distribution to test against. Currently support pareto, expon, and gamma, but any one-sided continuous distribution in Scipy.stats should work.
- **verbose** (*boolean*) – If True, will print out testing result

Returns

- **params** (*np.ndarray shape (p,)*) – The optimal parameter for the distribution. Optimal in the sense of minimizing K-S statistics.
- *The function also print out the Kolmogorov-Smirnov test result for three*
- *cases*
- *1. When comparing the empirical distribution against the distribution with*
- *parameters estimated with MLE*
- *2. When comparing the empirical distribution against the distribution with*
- *parameters estimated by explicitely minimizing KS statistics*
- *3. When comparing a resample with replacement of the empirical distribution*
- *against the Pareto in 2.*
- *A p-value > 0.05 means we fail to reject the Null hypothesis that the*
- *empirical distribution follow the specified distribution.*
- *Notes*
- ——
- *The MLE often does not fit the data very well. We instead minimizing the*
- *K-S distance, and obtain a better fit (as seen by the PDF and CDF*

- *similarity between sample data and the fit)*
- *References*
- ———— –

1. **Kolmogorov-Smirnov test:** https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test
2. **Pareto Distribution (also known as power law distribution)** https://en.wikipedia.org/wiki/Pareto_distribution

- *Examples*
- ———
- >>> *x = get_travel_distances(0, 0, 0)*
- >>> *res = perform_kstest(x, verbose=False)*
- >>> *np.allclose(res, np.array([3.67593246, 0.62795748, 0.37224205]))*
- *True*

Module contents

mousestyles.dynamics package

Module contents

mousestyles.dynamics.**create_time_matrix**(*combined_gap=4*, *time_gap=1*, *days_index=137*, *verbose=False*)

Return a time matrix for estimate the MLE parobability. The rows are 137 mousedays. The columns are time series in a day. The data are the mouse activity at that time. 0 represents IS, 1 represents eating, 2 represents drinking, 3 represents others activity in AS.

Parameters

- **combined_gap** (*nonnegative float or int*) – The threshold for combining small intervals. If next start time minus last stop time is smaller than combined_gap than combined these two intervals.
- **time_gap** (*positive float or int*) – The time gap for create the columns time series
- **days_index** (*nonnegative int*) – The number of days to process, from day 0 to day days_index.
- **verbose** (*bool*) – If True, print out helpful information to the screen

Returns **time** – a matrix represents the activity for a certain mouse day and a certain time.

Return type Pandas.DataFrame

Examples

```
>>> time = create_time_matrix(combined_gap=4, time_gap=1).iloc[0, 0:10]
>>> strain    0
    mouse     0
    day       0
    48007     0
    48008     0
```

```
48009     0
48010     0
48011     0
48012     0
48013     0
Name: 0, dtype: float64
```

mousestyles.dynamics.**find_best_interval**(*df*, *strain_num*, *interval_length_initial=array([600, 1200, 1800, 2400, 3000, 3600, 4200, 4800, 5400, 6000, 6600, 7200])*)

Returns the optimized time interval length and the corresponding fake mouse behavior string with the evaluation score for a particular mouse strain. The input data are the pandas DataFrame generated by function create_time_matrix which contains all strains information, the strain number that we want to develop, and some initial time interval length we want to optimize on. The outputs are the optimized time interval length, the simulated mouse states string using that optimized time interval and the evaluation score comparing this simulated mouse with the real mice behavior in the strain using the same optimized time interval length.

Parameters

- **df** (*Pandas.DataFrame*) – a huge data frame containing info on strain, mouse no., mouse day, and different states at chosen time points.
- **strain_num** (*int*) – an integer of 0, 1, 2 that specifying the desired mouse strain
- **interval_length_initial** (*numpy.ndarray*) – a numpy.ndarray specifying the range of time interval that it optimizes on, with the default value of a sequence from 600s to 7200s with 600s step since 10min to 2h is a reasonable choice.

Returns

- **best_interval_length** (*int*) – a interger indicating the optimized time interval in terms of the evaluation score
- **best_fake_mouse** (*list*) – a list of around 88,283 integers indicating the simulated states using the best_interval_length
- **best_score** (*float*) – a float between 0 and 1 representing the evaluation score comparing the best_fake_mouse with the real mice behavior in the strain under the same optimized time interval length. higher the score, better the simulation behavior.

Examples

```
>>> row_i = np.hstack((np.zeros(40)))
>>> time_df_eg = np.vstack((row_i, row_i, row_i))
>>> time_df_eg = pd.DataFrame(time_df_eg)
>>> time_df_eg.rename(columns={0:'strain'}, inplace=True)
>>> find_best_interval(time_df_eg, 0, np.arange(10, 40, 10))
(10, array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]), 0.98245614035087725)
```

mousestyles.dynamics.**get_prob_matrix_list**(*time_df*, *interval_length=1000*)

returns a list of probability transition matrices that will be later used to characterize and simu- late the behavior dynamics of different strains of mice. The data used as input is the pandas DataFrame generated by function create_time_matrix with de- fault parameters. The output is a list of numpy arrays, each being a transition matrix characterizing one small time interval. The interval length could be chosen.

Parameters

- **time_df** (*Pandas.DataFrame*) – a huge data frame containing info on strain, mouse no., mouse day, and different states at chosen time points.
- **interval_length** (*int*) – an integer specifying the desired length of each small time interval.

Returns **matrix_list** – a list of the mle estimations of the probability tran- sition matrices for each small time interval stored in the format of numpy array. Each element of this list is a numpy array matrix.

Return type list

Examples

```
>>> row_i = np.hstack((np.zeros(13), np.ones(10),
                       np.ones(10)*2, np.ones(10)*3))
>>> time_df_eg = np.vstack((row_i, row_i,row_i))
>>> time_df_eg = pd.DataFrame(time_df_eg)
>>> mat_list = get_prob_matrix_list(time_df_eg,
                                    interval_length=10)
>>> mat_list[0]
>>> array([[ 1.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.]])
>>> mat_list[1]
>>> array([[ 0.,  0.,  0.,  0.],
           [ 0.,  1.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.]])
>>> mat_list[2]
>>> array([[ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  1.,  0.],
           [ 0.,  0.,  0.,  0.]])
>>> mat_list[3]
>>> array([[ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  0.],
           [ 0.,  0.,  0.,  1.]])
```

mousestyles.dynamics.**get_prob_matrix_small_interval**(*string_list*, *verbose=False*)

return the MLE estimate of the probability matrix of the markov chain model. The data used as input is a list of strings that contains the information regarding the transition of states of the mouse be- havior. The output is a matrix stored in the format of numpy array, where the i,j th term indicates the probability of transiting from state i to state j.

Parameters

- **string_list** (*list*) – a list of strings of the states in the given time slot.
- **verbose** (*bool*) – If True, print out helpful information to the screen

Returns **M** – the MLE estimation of the probability tran- sition matrix. Each entry M_ij represents the probability of transiting from state i to state j.

Return type numpy.ndarray

Examples

```
>>> time_list = ['002', '001', '012']
>>> get_prob_matrix_small_interval(time_list)
>>> array([[ 0.4,  0.4,  0.2,  0. ],
           [ 0. ,  0. ,  1. ,  0. ],
           [ 0. ,  0. ,  0. ,  0. ],
           [ 0. ,  0. ,  0. ,  0. ]])
```

mousestyles.dynamics.**get_score**(*true_day, simulated_day, weight=[1, 10, 50, 1]*)

Returns the evaluation score for the simulted day that will be later used to choose the best time interval for different strains. The input data should be two numpy arrays and one list, the two arrays are possibly with different lengths, one being the activities for one particular day of one particular mouse, the other array is our simulated day for this mouse from the mcmc_simulation function. And the list is the weight for different status. We should give different rewards for making correct simulations on various status. For example, the average mouse day have 21200 timestamps. 10000 of them are IS, 1000 are EAT, 200 are drink, and the left 10000 are OTHERS. So we should weigh more on drink and eat, their ratio is 10000:1000:200:10000 = 1:0.1:0.02:0.1. So I took the inverse of them to be 1:10:50:1. The output will be one number between 0 and max(weight), indicating the similiary of the true day of a mouse and a simulated day of the same mouse. We will use this function to measure the performance of the simulation and then choose the appropriate time interval.

Parameters

- **true_day** (*numpy.array*) – a numpy.array containing the activities for one particular mouse on a specific day
- **simulated_day** (*numpy.array*) – a numpy.array containing the simulated activities for this particular mouse.
- **weight** (*list*) – a list with positive numbers showing the rewards for making the right predictions of various status.

Returns **score** – a float from 0 to max(weight), indicating the similarity of the simulated data with the actual value, and therefore, the performance of the simulation, with max(weight) being the most similar, and 0 being the least similar.

Return type float

Examples

```
>>> true_day_1 = np.zeros(13)
>>> simulated_day_1 = np.ones(13)
>>> get_score(true_day_1, simulated_day_1)
>>> 0.0
>>> true_day_2 = np.ones(13)
>>> simulated_day_2 = np.ones(13)
>>> get_score(true_day_2, simulated_day_2)
>>> 10.0
```

mousestyles.dynamics.**mcmc_simulation**(*mat_list, n_per_int*)

This function gives the Monte Carlo simulation of the stochastic process modeling the dynamic changes of states of the mice behavior. The in- put of this function is a list of probability transition matrices and an integer indicates how many outputs for each matrix. This number is related to the interval_length parameter in function get_prob_matrix_list. The output is an array of numbers, each indicates one state.

Parameters

- **mat_list** (*List*) – a list of numpy arrays storing the probabi- lity transition matrices for each small time interval chosen.
- **n_per_int** (*int*) – an integer specifying the desired output length of each probability transition matrix. This is the same as the parameter interval_length used in the function get_prob_matrix_small_interval

Returns **simu_result** – an array of integers indicating the simulated states given a list of probability transition matrices.

Return type numpy.array

Examples

```
>>> mat0 = np.zeros(16).reshape(4, 4)
>>> np.fill_diagonal(mat0, val=1)
>>> mat1 = np.zeros(16).reshape(4, 4)
>>> mat1[0, 1] = 1
>>> mat1[1, 0] = 1
>>> mat1[2, 2] = 1
>>> mat1[3, 3] = 1
>>> mat_list_example = [mat0, mat1]
>>> mcmc_simulation(mat_list_example, 10)
>>> array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
```

mousestyles.path_diversity package

Submodules

mousestyles.path_diversity.clean_movements module

mousestyles.path_diversity.clean_movements.**clean_movements**(*movements*, *keep_index=False*)

Returns a list of cleaned movement objects.

For each element of the input *movements* object, detects the existence of adjacent rows which have i) same x,y-coordinates, or ii) same timestamps. For case i), and removes the duplicated rows except for the first. For case ii), raises value error.

Parameters

- **movements** (*list*) – each element is pandas.DataFrame containing CT, CX, CY coordinates. Should have length greater than 1.
- **keep_index** (*boolean*) – whether or not keep the original index. Deafalt is False. If false, cleaned movement object is re-indexed.

Returns **cleaned-movements** – each element is cleaned movement object.

Return type list

Examples

```
>>> strain_nums = [0,0,1,1,2,2]
>>> mouse_nums = [0,0,1,1,2,2]
>>> day_nums = [0,11,0,11,0,11]
>>> movements = [data.load_movement(strain_num, mouse_num, day_num)
                 for strain_num, mouse_num, day_num in
                 zip(strain_nums, mouse_nums, day_nums)]
>>> cleaned_movements = clean_movements(movements)
```

mousestyles.path_diversity.detect_noise module

mousestyles.path_diversity.detect_noise.**detect_noise**(*movement*, *paths*, *angle_threshold*, *delta_t*)

Return a list object containing boolean values at points where measurement noise is detected and will be passed to a smoothing function

Parameters

- **movement** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse and day
- **index** (*paths*) –
- **angle_threshold** (*float*) – positive number indicating the minimum turning angle to flag as noise
- **delta_t** (*float*) – positive number indicating the delta_time interval

Returns

- **noise index** (*a pandas series containing the indices at which*)
- *noise, as defined by input parameters, is detected*

Examples

```
>>> movement = data.load_movement(1, 2, 1)
>>> paths = path_diversity.path_index(movement, 1, 1)
>>> noise = detect_noise(movement, paths, 135, .1)
```

mousestyles.path_diversity.filter_path module

mousestyles.path_diversity.filter_path.**filter_path**(*movement*, *paths*, *time_threshold*)

Return a list object containing start and end indices for movements lasting equal to or longer than the specified time threshold

Parameters

- **movement** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse and day
- **paths** (*list*) – a list containing the indices for all paths
- **time_threshold** (*float*) – positive number indicating the time threshold

Returns

- **paths index** (*a list containing the indices for all paths*)
- *that the spending times are larger than a time threshold*

Examples

```
>>> movement = data.load_movement(1, 2, 1)
>>> paths = path_index(movement, 1, 1)
>>> filter_path(movement, paths, 20)
[[26754, 26897], [28538, 28627]]
```

mousestyles.path_diversity.get_dist_speed module

mousestyles.path_diversity.get_dist_speed.**get_dist_speed**(*movement*, *start*, *end*, *return_array=True*)

Return a list containing distance(s) covered in a path and average speed(s) within a path.

Parameters

- **movement** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse, and day
- **start** (*int*) – positive integer indicating the starting index of a path
- **end** (*int*) – positive integer indicating the end index of a path
- **return_array** (*bool*) – boolean indicating whether an array of distances and average speeds are returned or the summation of those distances and speeds

Returns

- **dist** (*distance(s) travelled along a path*)
- **speed** (*average speed(s) along a path*)

Examples

```
>>> movement = data.load_movement(1, 2, 1)
>>> dist, speed = get_dist_speed(movement, 0, 3)
>>> print(dist)
[0.0, 0.17999999999999972, 0.19446593532030554]
>>> print(speed)
[0.0, 0.9999999999983815, 0.055246004352409776]
>>> dist, speed = get_dist_speed(movement, 0, 3, return_array=False)
>>> print(dist)
0.37446593532030525
>>> print(speed)
0.096661315260887087
```

mousestyles.path_diversity.path_features module

mousestyles.path_diversity.path_features.**angle_between**(*v1*, *v2*)

Returns the angle in radians between vectors *v1* and *v2*. Both vectors must have same length.

Parameters **v2** (*v1,*) – Vectors whose angle would be calculated. Should have same lengths. Should not be zero vector.

Returns **angle** – nan if either of *v1* or *v2* is zero vector,

Return type numpy float object

Examples

```
>>> angle_between([1, 0],[1, 0])
0.0
>>> angle_between([1, 0],[0, 1])
1.5707963267948966
>>> angle_between([1, 0],[-1, 0])
3.1415926535897931
```

mousestyles.path_diversity.path_features.**compute_accelerations**(*speeds*, *timestamps*)

Returns a list of the accelerations along the path. Each element of the list is the ratio of the speed to the time difference. The length of the list is equal to the length of speeds minus 1.

Parameters

- **speeds** (*list*) – the traveled distances along the path. Expecting the output of compute_distances.
- **timestamps** (*list*) – the time difference within the path. Its length must be equal to the length of speeds plus 1. Should not contain same time in adjacent rows; otherwise output would contain inf.

Returns **accel** – contains the speeds along the path.

Return type list

Examples

```
>>> compute_accelerations([1, 2, 0], [3, 4, 5, 6])
[0.5, -1.0]
```

mousestyles.path_diversity.path_features.**compute_angles**(*path_obj*, *radian=False*)

Returns a list of the angles in the path. Each element of the list is the angle between the adjacent vectors in the path. The length of the list is equal to the length of speeds minus 2.

Parameters

- **path_obj** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status. Must have length greater than 3.
- **radian** (*boolean*) – True for the output in radians. False for in turns (i.e. 360 for a full turn). Default is False.

Returns **angles** – contains the angles in the path. The first and last elements are None.

Return type list

Examples

```
>>> path = pd.DataFrame({'t':[2, 4.5, 10.5], 'x':[0, 1, 1],\
'y':[0, 0, 1], 'isHB':[True, True, False]})
>>> compute_angles(path)
[None, 90.0, None]
```

mousestyles.path_diversity.path_features_advanced module

mousestyles.path_diversity.path_features_advanced.**compute_advanced**(*path_obj*)

Returns dictionary containing several advanced features of path.

The features are the radius, center angles, area covered by the path, area of the rectangle spanned by the path, and absolute distance between start and end points.

Parameters **path_obj** (*pandas.DataFrame*) – CX and CY must be contained. The length must be greater than 2.

Returns

- **radius** (*list*) – each element is the distance between center point and each point in the path. The length equals to the length of the path_obj.
- **center_angles** (*list*) – each element is the center angle generated by 2 adjacent radius. The length equals to the length of the radius minus 1.
- **area_cov** (*numpy float object*) – area covered by the path. Computed by radius and center angles.
- **area_rec** (*numpy float object*) – area of the rectangle spanned by the path.
- **abs_distance** (*numpy float object*) – the distance between the start and end points in a path.

Examples

```
>>> movement = data.load_movement(1,2,1)
>>> adv_features = compute_advanced(movement[5:10])
```

mousestyles.path_diversity.path_index module

mousestyles.path_diversity.path_index.**path_index**(*movement*, *stop_threshold*, *min_path_length*)

Return a list object containing start and end indices for a specific movement. Each element in the list is a list containing two indices: the first element is the start index and the second element is the end index.

Parameters

- **movement** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse and day
- **stop_threshold** (*float*) – positive number indicating the path cutoff criteria if the time difference between two observations is less than threhold, they will be in the same path
- **min_path_length** (*int*) – positive integer indicating how many observations in a path

Returns **paths index**

Return type a list containing the indices for all paths

Examples

```
>>> movement = data.load_movement(1, 2, 1)
>>> paths = path_index(movement, 1, 1)[:5]
>>> paths
[[0, 2], [6, 8], [107, 113], [129, 131], [144, 152]]
```

mousestyles.path_diversity.smooth_noise module

mousestyles.path_diversity.smooth_noise.**smooth_noise**(*movement*, *paths*, *angle_threshold*, *delta_t*)

Return a new smoothed movement pandas DataFrame object containing CT, CX, CY coordinates.

The inputted movement DataFrame is passed through a noise detection function. At points where noise is detected, as defined by the input parameters (i.e., angle_threshold and delta_t), this function returns a new movement DataFrame by averaging points where noise is detected.

Parameters

- **movement** (*pandas.DataFrame*) – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse and day
- **index** (*paths*) –
- **angle_threshold** (*float*) – positive number indicating the minimum turning angle to flag as noise
- **delta_t** (*float*) – positive number indicating the delta_time interval

Returns **smoothed movement** – CT, CX, CY coordinates and homebase status for the unique combination of strain, mouse and day

Return type pandas.DataFrame

Examples

```
>>> movement = data.load_movement(1, 2, 1)
>>> paths = path_index(movement, 1, 1)
>>> smoothed_movement = smooth_noise(movement, paths, 135, .1)
```

Module contents

mousestyles.ultradian package

Module contents

mousestyles.ultradian.**aggregate_data**(*feature*, *bin_width*, *nmouse=4*, *nstrain=3*)

Aggregate all the strains and mouses with any feature together in one dataframe. It combines the results you got from aggregate_movements and aggregate_interval. It will return a dataframe with three variables: mouse, strain, feature and hour.

Parameters

- **feature** – {"AS", "F", "IS", "M_AS", "M_IS", "W", "Distance"}
- **bin_width** (*int*) – Number of minutes, the time interval for data aggregation.

Returns

- *pandas.dataframe*
- *describe* – Column 0: the mouse number (number depends on strain)(0-3) Column 1: the strain of the mouse (0-2) Column 2: hour(numeric values below 24 accourding to bin_width) Column 3: feature values

Examples

```
>>> test = aggregate_data("Distance",20)
>>> print(np.mean(test["Distance"]))
531.4500177747973
```

mousestyles.ultradian.**aggregate_interval**(*strain*, *mouse*, *feature*, *bin_width*)

Aggregate the interval data based on n-minute time intervals, return a time series.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **feature** (*{"AS", "F", "M_AS", "M_IS", "W"}*) – "AS": Active state probability "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g)
- **bin_width** (*number of minutes of time interval for data aggregation*) –

Returns **ts** – a pandas time series of length 12(day)*24(hour)*60(minute)/n

Return type pandas.tseries

mousestyles.ultradian.**aggregate_movement**(*strain*, *mouse*, *bin_width*)

Aggregate the movement data based on n-minute time intervals, return a time series.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **bin_width** (*number of minutes of time interval for data aggregation*) –

Returns **ts** – a pandas time series of length (#day)*24(hour)*60(minute)/n

Return type pandas.tseries

mousestyles.ultradian.**find_cycle**(*feature*, *strain*, *mouse=None*, *bin_width=15*, *methods=u'LombScargleFast'*, *disturb_t=False*, *gen_doc=False*, *plot=True*, *search_range_fit=None*, *nyquist_factor=3*, *n_cycle=10*, *search_range_find=(2, 26)*, *sig=array([0.05])*)

Use Lomb-Scargel method on different strain and mouse's data to find the best possible periods with highest p-values. The function can be used on specific strains and specific mouses, as well as just specific strains without specifying mouse number. We use the O(NlogN) fast implementation of Lomb-Scargle from the gatspy package, and also provide a way to visualize the result.

Note that either plotting or calculating L-S power doesn't use the same method in finding best cycle. The former can use user-specified search_range, while the latter uses default two grid search_range.

Parameters

- **feature** (*string in {"AS", "F", "M_AS", "M_IS", "W", "Distance"}*) – "AS": Active state probalibity "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g) "Distance": Distance traveled
- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int, default is None*) – nonnegative integer indicating the mouse number
- **bin_width** (*int, minute unit, default is 15 minutes*) – number of minutes, the time interval for data aggregation
- **methods** (*string in {"LombScargleFast", "LombScargle"}*) – indicating the method used in determining periods and best cycle. If choose 'LombScargle', 'disturb_t' must be True.
- **disturb_t** (*boolean, default is False*) – If True, add uniformly distributed noise to the time sequence which are used to fit the Lomb Scargle model. This is to avoid the singular matrix error that could happen sometimes.
- **plot** (*boolean, default is True*) – If True, call the visualization function to plot the Lomb Scargle power versus periods plot. First use the data (either strain specific or strain-mouse specific) to fit the LS model, then use the search_range_fit as time sequence to predict the corresponding LS power, at last draw the plot out. There will also be stars and horizontal lines indicating the p-value of significance. Three stars will be p-value in [0,0.001], two stars will be p-value in [0.001,0.01], one star will be p-value in [0.01,0.05]. The horizontal line is the LS power that has p-value of 0.05.
- **search_range_fit** (*list, numpy array or numpy arange, hours unit,*) – default is None list of numbers as the time sequence to predict the corrsponding Lomb Scargle power. If plot is 'True', these will be drawn as the x-axis. Note that the number of search_range_fit points can not be too small, or the prediction smooth line will not be accurate. However the plot will always give the right periods and their LS power with 1,2 or 3 stars. This could be a sign to check whether search_range_fit is not enough to draw the correct plot. We recommend the default None, which is easy to use.
- **nyquist_factor** (*int*) – If search_range_fit is None, the algorithm will automatically choose the periods sequence. 5 * nyquist_factor * length(time sequence) / 2 gives the number of power and periods used to make LS prediction and plot the graph.
- **n_cycle** (*int, default is 10*) – numbers of periods to be returned by function, which have the highest Lomb Scargle power and p-value.
- **search_range_find** (*list, tuple or numpy array with length of 2, default is*) – (2,26), hours unit Range of periods to be searched for best cycle. Note that the minimum should be strictly larger than 0 to avoid 1/0 issues.
- **sig** (*list or numpy array, default is [0.05].*) – significance level to be used for plot horizontal line.
- **gen_doc** (*boolean, default is False*) – If true, return the parameters needed for visualize the LS power versus periods

Returns

- **cycle** (*numpy array of length 'n_cycle'*) – The best periods with highest LS power and p-values.
- **cycle_power** (*numpy array of length 'n_cycle'*) – The corrsponding LS power of 'cycle'.
- **cycle_pvalue** (*numpy array of length 'n_cycle'*) – The corrsponding p-value of 'cycle'.

- **periods** (*numpy array of the same length with 'power'*) – use as time sequence in LS model to make predictions.Only return when gen_doc is True.
- **power** (*numpy array of the same length with 'periods'*) – the corresponding predicted power of periods. Only return when gen_doc is True.
- **sig** (*list, tuple or numpy array, default is [0.05].*) – significance level to be used for plot horizontal line. Only return when gen_doc is True.
- **N** (*int*) – the length of time sequence in the fit model. Only return when gen_doc is True.

Examples

```
>>> a,b,c = find_cycle(feature='F', strain = 0,mouse = 0, plot=False,)
>>> print(a,b,c)
>>> [ 23.98055016   4.81080233  12.00693952   6.01216335   8.0356203
     3.4316698    2.56303353   4.9294791   21.37925713   3.5697756 ]
    [ 0.11543449  0.05138839  0.03853218  0.02982237  0.02275952
    0.0147941  0.01151601  0.00998443  0.00845883  0.0082382 ]
    [  0.00000000e+00   3.29976046e-10   5.39367189e-07   8.10528027e-05
      4.71001953e-03   3.70178834e-01   9.52707020e-01   9.99372657e-01
     9.99999981e-01   9.99999998e-01]
```

mousestyles.ultradian.**mix_strain**(*data*, *feature*, *print_opt=True*, *nstrain=3*, *search_range=(3, 12)*, *degree=1*)

Fit the linear mixed model onto our aggregate data. The fixed effects are the hour, strain, interactions between hour and strain; The random effect is mouse because we want to make sure that the different mouses will not give out any differences. We added two dummy variables: strain0 and strain1 to be our fixed effects.

Parameters

- **data** (*data frame output from aggregate_data function*) –
- **feature** (*{"AS", "F", "IS", "M_AS", "M_IS", "W", "Distance"}*) –
- **print_opt** (*True or False*) –
- **nstrain** (*positive integer*) –
- **range** (*array contains two elements*) –
- **degree** (*positive integer*) –

Returns

- *Two mixed model regression results which includes all the coefficients,*
- *t statistics and p values for corresponding coefficients; The first model*
- *includes interaction terms while the second model does not include the*
- *interaction terms*
- *Likelihood ratio test p values, if it is below our significance level,*
- *we can conclude that the different strains have significantly different*
- *time patterns*

Examples

```
>>> result = mix_strain(data = aggregate_data("F",30), feature = "F",
>>>          print_opt = False, degree = 2)
>>> print(result)
2.5025846540930469e-09
```

mousestyles.ultradian.**seasonal_decomposition**(*strain*, *mouse*, *feature*, *bin_width*, *period_length*)

Apply seasonal decomposition model on the time series of specified strain, mouse, feature and bin_width.

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*int*) – nonnegative integer indicating the mouse number
- **feature** (*{"AS", "F", "M_AS", "M_IS", "W", "Distance"}*) – "AS": Active state probalibity "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g) "Distance": Distance traveled
- **bin_width** (*int*) – number of minutes, the time interval for data aggregation
- **period_length** (*int or float*) – number of hours, usually the significant period length indicated by Lomb-scargle model

Returns **res** – seasonal decomposition result for the mouse. Check the seasonal decomposition plot by res.plot(), seasonl term and trend term by res.seasonal and res.trend separately.

Return type statsmodel seasonal decomposition object

Examples

```
>>> res = seasonal_decomposition(strain=0, mouse=0, feature="W",
                                 bin_width=30, period_length = 24)
```

mousestyles.ultradian.**strain_seasonal**(*strain*, *mouse*, *feature*, *bin_width*, *period_length*)

Use seansonal decomposition model on the time series of specified strain, mouse, feature and bin_width. return the seasonal term and the plot of seasonal term by mouse of a set of mouses in a strain

Parameters

- **strain** (*int*) – nonnegative integer indicating the strain number
- **mouse** (*list, set or tuple*) – nonnegative integer indicating the mouse number
- **feature** (*{"AS", "F", "M_AS", "M_IS", "W", "Distance"}*) – "AS": Active state probalibity "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g) "Distance": Distance traveled
- **bin_width** (*int*) – number of minutes, the time interval for data aggregation
- **period_length** (*int or float*) – number of hours, usually the significant period length indicated by Lomb-scargle model

Returns **seasonal_all** – mouse indicated by the input parameter

Return type numpy array containing the seasonal term for every

Examples

```
>>> res = strain_seasonal(strain=0, mouse={0, 1, 2, 3}, feature="W",
                          bin_width=30, period_length = 24)
```

mousestyles.visualization package

Submodules

mousestyles.visualization.distribution_plot module

mousestyles.visualization.distribution_plot.**plot_exponential**(*estimation*)
Return the histogram of all estimators of exponential to check the distribution.

Parameters **estimation** (*dataframe*) – dataframe of strain, mouse, day and the estimator

Returns **plot** – The histogram of all estimators of exponential.

Return type histogram

mousestyles.visualization.distribution_plot.**plot_fitted**(*strain*, *mouse*, *day*, *hist=True*, *density=False*)
Return the plot of one single mouse day -fitted power law -fitted exponential -histogram of distance -kernel density curve

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number
- **hist** (*boolean*) – Plot histogram if True
- **density** (*boolean*) – plot density if True

Returns **plot**

Return type 1 histogram (blue) + 2 fitted curve + 1 density (cyan)

mousestyles.visualization.distribution_plot.**plot_powerlaw**(*estimation*)
Return the histogram of all estimators of power law to check the distribution.

Parameters **estimation** (*dataframe*) – dataframe of strain, mouse, day and the estimator

Returns **plot** – The histogram of all estimators of power law

Return type histogram

mousestyles.visualization.distribution_plot.**plot_relative_dist**(*strain*, *mouse*, *day*)
Return the relative distribution of distance of a single mouse day The relative distribution of a comparison distribution with density g vs a reference distribution with density f is g(F^(-1)(x))/f(F^(-1)(x)) It's a prob distribution on [0, 1].

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number

Returns

- **pdf_rel_power** (*numpy.ndarray*) – the relative distribution, referencing the best fit power-law distribution, at grid points np.arange(0, 1.0, 0.02)
- **pdf_rel_exp** (*numpy.ndarray*) – the relative distribution, referencing the best fit exponential distribution, at grid points np.arange(0, 1.0, 0.02)

mousestyles.visualization.dynamics module

mousestyles.visualization.dynamics.**plot_dynamics**(*df*, *strain_num*, *interval_length_initial=array([600, 1200, 1800, 2400, 3000, 3600, 4200, 4800, 5400, 6000, 6600, 7200])*, *plot_time_range=array([36000, 36001, 36002, 36003, 36004, 36005, 36006, 36007, 36008, 36009, 36010, 36011, 36012, 36013, 36014, 36015, 36016, 36017, 36018, 36019, 36020, 36021, 36022, 36023, 36024, 36025, 36026, 36027, 36028, 36029, 36030, 36031, 36032, 36033, 36034, 36035, 36036, 36037, 36038, 36039, 36040, 36041, 36042, 36043, 36044, 36045, 36046, 36047, 36048, 36049, 36050, 36051, 36052, 36053, 36054, 36055, 36056, 36057, 36058, 36059, 36060, 36061, 36062, 36063, 36064, 36065, 36066, 36067, 36068, 36069, 36070, 36071, 36072, 36073, 36074, 36075, 36076, 36077, 36078, 36079, 36080, 36081, 36082, 36083, 36084, 36085, 36086, 36087, 36088, 36089, 36090, 36091, 36092, 36093, 36094, 36095, 36096, 36097, 36098, 36099])*)

returns a plot that can help understand the behavior dynamics that are obtained from the best simulation. The data used as input is the pandas DataFrame generated by function create_time_matrix. The output is a plot that summarizes the dynamics of a fake mouse of the given strain_num. The strain_num could be chosen. Of note is that 0 represents IS, 1 represents eating, 2 represents drinking, 3 represents others activity in AS. In the plot, blue represents IS, bright green represents eating, yellow represents drinking, and red represents other activities in AS.

Parameters

- **df** (*Pandas.DataFrame*) – a huge data frame containing info on strain, mouse no., mouse day, and different states at chosen time points.
- **starin_num** (*int*) – an integer specifying the desired mouse strain. strain_num is 0, 1, or 2.
- **interval_length_initial** (*numpy.ndarray*) – a numpy.ndarray specifying the range of time interval that it optimizes on.
- **plot_time_range** (*numpy.ndarray*) – a numpy.ndarray specifying the range of time range of the plot.

Returns **dynamics_plot** – a plot of behavior dynamics of a fake mouse of the given strain_num. The x-axis is the time stamps that start from 0. For strain_num = 0, the x-aixs is from 0 to 92,400. For stain_num = 1, the x-axis is from 0 90,000. For strain_num = 2, the x-axis is from 0 to 88,800. We assign different colors for different states. In the plot, blue represents IS, bright green represents eating, yellow represents drinking, and red represents other activities in AS.

Return type plot

Examples

```
>>> row_i = np.hstack((np.zeros(40)))
>>> time_df_eg = np.vstack((row_i, row_i, row_i))
>>> time_df_eg = pd.DataFrame(time_df_eg)
>>> time_df_eg.rename(columns={0:'strain'}, inplace=True)
>>> plot_dynamics_plot(time_df_eg, 0,
                       np.arange(10, 40, 10), np.arange(0, 40, 1))
```

mousestyles.visualization.path_diversity_plotting module

mousestyles.visualization.path_diversity_plotting.**plot_box**(*list_of_arrays*, *title=u'Box Plot of Distribution'*, *width=4*, *height=4*)

Make a box plot of the desired metric (path length, speed, angle, etc) per mouse, day or strain.

Parameters

- **list_of_arrays** (*list*) – each element of the list is a numpy array containing data on the metric to be plotted per mouse, day, or strain.Data is typically generated from another function in the package.
- **title** (*str*) – desired title of the plot
- **width** (*int*) – first argument in figure size specifying width of plot
- **height** (*int*) – second argument in figure size specifying height of plot

Returns **box plot**

Return type box plot of the desired metric combinations

mousestyles.visualization.path_diversity_plotting.**plot_hist**(*list_of_arrays*, *title=u'Histogram of Distribution'*, *xlab=u'X Values'*, *ylab=u'Y Values'*, *leg=True*)

Make a histogram of the desired metric (path length, speed, angle, etc) per mouse, day, or strain.

Parameters

- **list_of_arrays** (*list*) – each element of the list is a numpy array containing data on the metric to be plotted per mouse, day, or strain.Data typically generated from another function in the package.
- **title** (*str*) – desired title of the plot
- **xlab** (*str*) – desired x axis label of the plot
- **ylab** (*str*) – desired y axis label of the plot

- **leg** (*bool*) – indicates whether a legend for the plot is desired

Returns **hist plot**

Return type histogram of the desired metric combinations

mousestyles.visualization.plot_classification module

mousestyles.visualization.plot_classification.**plot_comparison**(*comparison*)

Plots the F1 Measure of different classification models. It is a side-by-side barplot. For each strain, it plots the F-1 measure of RandomForest, GradientBoosting, SVM. :param comparison: :type comparison: dataframe, columns of F-1 measures of 3 :param methods:

Returns

Return type None

mousestyles.visualization.plot_classification.**plot_performance**(*result*)

Plots the performance of classification model.It is a side-by-side barplot. For each strain, it plots the precision, recall and F-1 measure. :param result: F-1 measure. :type result: dataframe, columns of precision, recall and

Returns

Return type None

mousestyles.visualization.plot_clustering module

mousestyles.visualization.plot_clustering.**plot_dendrogram**(*mouse_day*, *method*, *dist*)

Returns a linkage matrix and plot the dendrogram

Parameters

- **mouse_day** (*a 170 * M numpy array,*) – column 0 : strain, column 1: mouse, other columns corresponding to feature avg/std of a mouse over 16 days
- **method** (*string,*) – method of calculating distance between clusters
- **dist** (*string,*) – distance metric

Returns **Z** – The hierarchical clustering encoded as a linkage matrix.

Return type ndarray

mousestyles.visualization.plot_clustering.**plot_lastp_dist**(*Z*, *method*, *dist*, *p=10*)

Plot the distances between clusters of last p merges in hierarchical clustering

Parameters

- **Z** (*ndarray*) – The hierarchical clustering encoded as a linkage matrix
- **p** (*int, optional*) – number of merges to plot

mousestyles.visualization.plot_clustering.**plot_strain_cluster**(*count_data*, *groupby_cluster=True*)

Plot the side by side bar chart showing the strain distribution of mice in different clusters or the cluster distribution in different strains

Parameters

- **count_data** (*Dictionary*) – A dictionary object with the keys are the cluster numbers and the values are lists containing strain counts

- **groupbu_cluster** (*Boolean*) – If True, then the barchart is group by cluster otherwise by strain

mousestyles.visualization.plot_lomb_scargle module

mousestyles.visualization.plot_lomb_scargle.**lombscargle_visualize**(*periods*, *power*, *sig*, *N*, *cycle*, *cycle_power*, *cycle_pvalue*)

Use Lomb-Scargel method on different strain and mouse's data to find the best possible periods with highest p-values, and plot the Lomb Scargle power versus periods plot. use the periods as time sequence to predict the corresponding LS power, draw the plot.

There will also be stars and horizontal lines indicating the p-value of significance. Three stars will be p-value in [0,0.001], two stars will be p-value in [0.001,0.01], one star will be p-value in [0.01,0.05]. The horizontal line is the LS power that has p-value of 0.05.

Parameters

- **periods** (*numpy array of the same length with 'power'*) – use as time sequence in LS model to make predictions
- **power** (*numpy array of the same length with 'periods'*) – the corresponding predicted power of periods
- **sig** (*list, tuple or numpy array, default is [0.05].*) – significance level to be used for plot horizontal line.
- **N** (*int*) – the length of time sequence in the fit model
- **cycle** (*numpy array*) – periods
- **cycle_power** (*numpy array*) – LS power corrsponding to the periods in 'cycle'
- **cycle_pvalue** (*numpy array*) – p-values corresponding to the periods in 'cycle'

Returns

Return type Lomb Scargle Power versus Periods (hours) plot with significant levels.

Examples

mousestyles.visualization.plot_path module

mousestyles.visualization.plot_path.**plot_path**(*movement, paths, title=u'example plot of path', alpha=0.1, linewidth=1.0, xlim=[-16.24, 3.76], ylim=[0.9, 43.5]*)

Plot the lines along paths.

Parameters

- **movement** (*pandas.DataFrame*) – CX, CY coordinates. Must have length greater than 1.
- **paths** (*list*) – a list containing the indices for all paths.
- **title** (*str*) – the title of the plot. Default is 'example plot of path'

- **alpha** (*numeric*) – graphical parameter which determines strongness of each line. Default is .1.
- **linewidth** (*numeric*) – graphical parameter which determines the width of each line. Default is 1.
- **ylim** (*xlim,*) – list of length 2 indicating the end points of the plot.

Returns

Return type Drawing the plot of the path.

Examples

```
>>> movement = data.load_movement(1,2,1)
>>> sep = path_index(movement, 1, 1)
>>> plot_path(movement, sep)
```

mousestyles.visualization.plot_ultradian module

mousestyles.visualization.plot_ultradian.**compare_strain**(*feature*, *n_strain=3*, *bin_width=15*, *disturb_t=False*)

Use the data from function find_cycle and plotting method from function lombscargle_visualize to compare the Lomb-Scargle plots between different strains.

Parameters

- **feature** (*string in {"AS", "F", "M_AS", "M_IS", "W", "Distance"}*) – "AS": Active state probalibity "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g) "Distance": Distance traveled
- **n_strain** (*int, defalt is 3*) – nonnegative integer indicating total number of strains to be compared
- **bin_width** (*int, minute unit, default is 15 minutes*) – number of minutes, the time interval for data aggregation
- **disturb_t** (*boolean, default is False*) – If True, add uniformly distributed noise to the time sequence which are used to fit the Lomb Scargle model. This is to avoid the singular matrix error that could happen sometimes.

Returns

Return type Lomb Scargle Power versus Periods (hours) plot with significant levels.

Examples

mousestyles.visualization.plot_ultradian.**plot_strain_seasonal**(*strains*, *mouse*, *feature*, *bin_width*, *period_length*)

Use seansonal decomposition model on the time series of specified strain, mouse, feature and bin_width. return the seasonal term and the plot of seasonal term by mouse of a set of mouses in a strain

Parameters

- **strain** (*list, set or tuple*) – nonnegative integer indicating the strain number
- **mouse** (*list, set or tuple*) – nonnegative integer indicating the mouse number
- **feature** (*{"AS", "F", "M_AS", "M_IS", "W", "Distance"}*) – "AS": Active state probalibity "F": Food consumed (g) "M_AS": Movement outside homebase "M_IS": Movement inside homebase "W": Water consumed (g) "Distance": Distance traveled
- **bin_width** (*int*) – number of minutes, the time interval for data aggregation
- **period_length** (*float or int*) – number of hours, usually the significant period length indicated by Lomb-scargle model

Returns **seasonal_plot**

Return type plot of seasonal term by mouse

Examples

```
>>> res = plot_strain_seasonal(strains={0, 1, 2,}, mouse={0, 1, 2, 3},
                               feature="W",
                               bin_width=30, period_length = 24)
```

Module contents

3.1.2 Submodules

3.1.3 mousestyles.GLRT_distribution module

mousestyles.GLRT_distribution.**hypo_exp_null**(*strain*, *mouse*, *day*, *law_est=0*, *exp_est=0*, *seed=-1*)

Return the outcome from GLRT with null hypothesis law distribution.

This function also used the Generalized Likelihood Ratio Test to test goodness of fit: in other words, which distribution is more likely.

In this function, we choose the exponential distributin to be the null and powerlaw distribution to be the alternative. We derived the test statistics by theory and pluged in MLE as our estimation of best parameters.

After we calculated the paramters, we need to find the rejection region, critical value or pvalue. To get a more general test, we want to use pvalue, instead of critical value under certain significance level.

To find the p-value, we use simulation methods, and all random numbers are drawn from previous functions. Therefore, although p value should be a constant given data, it is not a constant in our function, if we did not set the seed.

In general, in this function, if the p value is too small, then we will reject the null, and we say exponential is not a better fit compared to exponential distribution.

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number
- **law_est** (*double (optional)*) – the estimated parameter in law distribution

- **exp_est** (*double (optional)*) – the estimated parameter in exponential distribution

Returns the probablity under null reject.

Return type p_value

Examples

```
>>> hypo_exp_null (0, 0, 0)
1.0
```

mousestyles.GLRT_distribution.**hypo_powerLaw_null**(*strain*, *mouse*, *day*, *law_est=0*, *seed=-1*)

Return the outcome from GLRT with null hypothesis law distribution.

This function used the Generalized Likelihood Ratio Test to test the goodness of fit: in other words, which distribution is more likely.

In this function, we choose the powerLaw distributin to be the null and exponential distribution to be the alternative. We derived the test statistics by theory and pluged in MLE as our estimation of best parameters.

After we calculated the paramters, we need to find the rejection region, critical value or pvalue. To get a more general test, we want to use pvalue, instead of critical value under certain significance level.

To find the p-value, we use simulation methods, and all random numbers are drawn from previous functions. Therefore, although p value should be a constant given data, it is not a constant in our function, if we did not set the seed.

In general, in this function, if the p value is too small, then we will reject the null, and we say powerlaw is not a better fit compared to exponential distribution.

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number
- **law_est** (*double (optional)*) – the estimated parameter in law distribution

Returns the probablity under null reject.

Return type p_value

Examples

```
>>> hypo_law_null (0, 0, 0)
0.0070000000000000001
```

mousestyles.GLRT_distribution.**random_exp**(*n*, *l*, *seed=-1*)

Random generate points of truncated exponential.

The method we generate is to use the memorylessness property of exponential distribution. As the survival function of exponential distribution is always the same, for truncated exponential distribution, it is just the same to draw from regular exponential distribtion and shift the truncated value.

Parameters

- **n** (*int*) – number of points
- **l** (*int*) – exponential parameter lambda

Returns **points** – n points have the target distribution.

Return type a vector of float number

Examples

```
>>> random_exp(4,2)
array([ 1.07592496,  1.19789646,  1.19759663,  1.03993227])
```

mousestyles.GLRT_distribution.**random_powerlaw**(*n*, *a*, *seed=-1*)

Random generate points of truncated power law.

The method we generate is to inverse Cumulative Density Function of truncated powerlaw function, and put random number draw from Unif[0,1]. The theory behind it is $F^{-1}(U) \sim F$.

Parameters

- **n** (*int*) – number of points
- **a** (*int>1*) – power law parameter alpha

Returns **points** – n points have the target distribution.

Return type a vector of float number

Examples

```
>>> random_powerlaw(4,2)
array([  1.18097435,   1.04584078,   1.4650779 ,  36.03967524])
```

3.1.4 mousestyles.distribution module

mousestyles.distribution.**exp_inverse_cdf**(*y*, *l*)

The inverse CDF function of truncated (at 1) exponential distribution

Parameters

- **y** (*float in [0, 1], or a np.dnarray*) – y in formula F^(-1)(y) = 1 - log(1 - y) / l
- **l** (*float > 0*) – a in formula F^(-1)(y) = 1 - log(1 - y) / l

Returns **x** – The inverse CDF function of truncated (at 1) exponential distribution distribution with parameter l at point y

Return type float

Examples

```
>>> exp_inverse_cdf(0.6,2)
1.4581453659370776
```

mousestyles.distribution.**exp_pdf**(*x*, *l*)

The probability density function of truncated exponential.

Parameters

- **x** (*float, or a np.dnarray*) – x in formula p(x)=lambda*exp(-lambda*x).

- **l** (*float*) – lambda in formula p(x)=lambda*exp(-lambda*x).

Returns **probability density** – The probability density of power law at x.

Return type float

Examples

```
>>> exp_pdf(1, 1)
0.36787944117144233
```

mousestyles.distribution.**powerlaw_inverse_cdf**(*y*, *a*)

The inverse CDF function of power law distribution

Parameters

- **y** (*float in [0, 1], or a np.dnarray*) – y in formula F^(-1)(y) = (1 - y)^(1/(1-a))
- **a** (*float > 1*) – a in formula F^(-1)(y) = (1 - y)^(1/(1-a))

Returns **x** – The inverse CDF function of power law distribution with parameter a at point y

Return type float

Examples

```
>>> powerlaw_inverse_cdf(0.5, 5)
1.189207115002721
```

mousestyles.distribution.**powerlaw_pdf**(*x*, *a*)

The probability density function of truncated power law.

Parameters

- **x** (*float > 0, or a np.dnarray*) – x in formula p(x)=(alpha-1)*x^(-alpha).
- **a** (*float > 1*) – alpha in formula p(x)=(alpha-1)*x^(-alpha).

Returns **probability density** – The probability density of power law at x.

Return type float

Examples

```
>>> powerlaw_pdf (2, 2)
0.25
```

3.1.5 mousestyles.est_power_param module

mousestyles.est_power_param.**fit_dist_all**()

Return the estimators of truncated power law and exponential for each mouse day.

Returns **estimator** – The estimator of truncated exponential distribution.

Return type a float number

Examples

```
>>> fit()
7.385844980814098
```

mousestyles.est_power_param.**fit_exponential**(*strain*, *mouse*, *day*)

Return the estimator of truncated exponential.

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number

Returns **estimator** – The estimator of truncated exponential distribution.

Return type a float number

Examples

```
>>> fit_exponential (0, 0, 0)
7.385844980814098
```

mousestyles.est_power_param.**fit_powerlaw**(*strain*, *mouse*, *day*)

Return the estimator of truncated power law.

Parameters

- **strain** (*int*) – the strain number of the mouse
- **mouse** (*int*) – the mouse number in its strain
- **day** (*int*) – the day number

Returns **estimator** – The estimator of truncated power law.

Return type a float number

Examples

```
>>> fit_powerlaw (0, 0, 0)
9.4748705008269827
```

mousestyles.est_power_param.**getdistance**(*strain*, *mouse*, *day*)

Return the distance of each two consecutive points among coordinates which is bigger than 1cm(truncated).

Parameters

- **strain** (*int*) –
- **strain number of the mouse** (*the*) –
- **mouse** (*int*) –
- **mouse number in its strain** (*the*) –
- **day** (*int*) –
- **day number** (*the*) –

Returns

- **cut_dist** (*an array of number*)
- *The vector of truncated distance.*

Examples

```
>>> getdistance (0, 0, 0)
array([ 1.00648944,  1.02094319,  1.0178885 , ...,  1.00099351,
1.01191156,  1.00423354])
```

3.1.6 mousestyles.intervals module

Finite Union of Intervals

Implements a class to handle finite (disjoint) unions of intervals.

- assumes that intervals are always closed and that the union is disjoint
- open intervals remaining at the end of any operations (eg. complement)
- are always made closed. e.g. [0,1]^C = [-np.inf,0] [1,np.inf]
- end intervals being unbounded is handled using -np.inf and np.inf
- does some okay handling for point intervals [a,a]

Darren Rhea, 2012; Chris Hillar revised, April 30, 2013; Ram Mehta revised, 2013; Copyright (c) 2013, All rights reserved; Chris Hillar revised, 2015

class mousestyles.intervals.**Intervals**(*intervals=None*)

Bases: object

Finite Union of Intervals [ai,bi] backed by sorted lists.

parameters intervals: (M x 2) numpy np.double array

ASs(*ISDT=20*)

returns new object of Active States given self as Events

ISs(*ISDT=20*)

returns new object of Inactive States given self as Events

complement()

New Intervals object which is the complement of self.

connect_gaps(*eps=0.001*)

connects consecutive intervals separated by lengths <= eps

connect_gaps_by_rule(*rule*)

Returns a new object with gaps connected when rule returns True.

Parameters rule: Callable that takes parameters start_time and end_time.

contains(*x*)

Check if x is in the Finite Union of Intervals.

copy()

index_of_first_intersection(*x*, *find_nearest=False*)

finds interval nearest to given number x and containing x if find_nearest=False: doesn't require x to be in the interval

intersect(*F*)
New Intervals object which is the intersection of self and Intervals F.

intersect_with_interval(*a*, *b*)
returns (not a copy) Intervals object which is the intersection of self and [a, b] (faster than intersect)

is_empty()

load(*filename=u'Intervals_save.npz'*)

measure()

num()

remove(*other*)

save(*filename=u'Intervals_save'*)

subordinate_to_array(*arr*)
returns a new Intervals object with only intervals containing elements of arr (NOTE: arr is assumed sorted)

symmetric_difference(*other*)

trim(*eps=0.001*)
Removes intervals with lengths <= eps.

union(*F*)
New Intervals object which is the union of self and Intervals F.

mousestyles.intervals.**binary_from_intervals**(*intervals*, *length=None*)
From an intervals object produce a binary sequence of size length

mousestyles.intervals.**intervals_from_binary**(*bin_array*, *times*)
Given a one dimensional bin_array of 0s and 1s, returns a Intervals object of times corresponding to consecutives 1s

mousestyles.intervals.**timestamps_to_interval**(*array*, *eps=0.01*)
given a 1D array with event timestamps, returns an interval centered on timestamp and eps wide. default 0.01 is half of minimum HCM sampling rate

3.1.7 mousestyles.kde module

mousestyles.kde.**kde**(*x*, *x_grid*, *symmetric_correction=False*, *cutoff=1*)
Return a numpy.ndarray object of estimated density

Parameters

- **x** (*numpy.ndarray*) – data, as realiztions of variable X
- **x_grid** (*numpy.ndarray*) – the grid points for the estimated density
- **symmetric_correction** (*boolean*) – a method indicator. If False, do common gaussian kernel density estimation (kde). If True, do common gaussian kde on data generated from x concatenating with its reflection around the cutoff point. Then transform the estimated kde back by a factor of 2. Used for e.g. kde for nonnegative kernel estimation
- **cutoff** (*float*) – the axis of symmetry for symmetric correction

Returns **pdf** – estimated density at the specified grid points x_grid

Return type numpy.ndarray

Examples

```
>>> kde(x = np.array([2,3,1,0]), x_grid=np.linspace(0, 5, 10))
array([ 0.17483395,  0.21599529,  0.23685855,  0.24007961,  0.22670763,
    0.19365019,  0.14228937,  0.08552725,  0.04043597,  0.01463953])
>>> x1 = np.concatenate([norm(-1, 1.).rvs(400), norm(1, 0.3).rvs(100)])
>>> pdf1 = kde(x=x1, x_grid=np.linspace(0, 5, 100), symmetric_correction
                =True, cutoff=1)
array([ 0.26625297,  0.26818492,  0.27105849,  0.27489486,  0.27968752, ...
    0.07764054,  0.07239964,  0.06736559,  0.06254175,  0.05793043])
```

3.1.8 mousestyles.mww module

mousestyles.mww.**MWW_allmice**(*step=50*, *verbose=False*)

Aggregates MWW_mice data for all available strains of mice.

Parameters

- **step** (*time interval length used to compute distances. Default is 1s.*) – See data.distances_bymouse for more information.
- **verbose** (*boolean*) –

Returns

- **mww_values** (*MWW_mice outputs for each strain.*) – mww_values[i] corresponds to the ith strain.
- *Examples*
- ———
- >>> *mww_values = MWW_allmice()*

mousestyles.mww.**MWW_mice**(*strain*, *step=50*, *verbose=False*)

Compare distributions of distances among mice of the same strain. Use p-values of the Mann-Whitney U test.

Parameters

- **strain** (*integer*) – Number of the strain.
- **step** (*floeat*) – Time interval length used to compute distances. Default is 1s. See data.distances_bymouse for more information.
- **verbose** (*boolean*) –

Returns

- **cor** (*pvalues of the Mann-Whitney U test for each couple of distances*) – samples among mice of the corresponding strain.
- *Examples*
- ———
- >>> *cor = MWW_mice(0)*

mousestyles.mww.**MWW_strains**(*step=50*, *verbose=False*)

Compare distributions of distances among strains. Proceed as if the mice in each strain are i.i.d. samples, and compare the p-values of the Mann-Whitney U test.

Parameters

- **step** (*time interval length used to compute distances. Default is 1s.*) – See data.distances_bymouse for more information.
- **verbose** (*boolean*) –

Returns

- **cor** (*pvalues of the Mann-Whitney U test for each couple of distances*) – samples among strains of mice.
- *Examples*
- ———
- >>> *cor = MWW_strains()*

mousestyles.mww.**get_pvalues**(*m*)

This function takes a bunch of sampled distributions and compute the p-values of the two sided Mann Whitney U test for each couple of samples.

The Mann-Whitney U test is a test for assessing whether two independent samples come from the same distribution. The null hypothesis for this test is that the two groups have the same distribution, while the alternative hypothesis is that one group has larger (or smaller) values than the other.

Null hypothesis H_0: $P(X > Y) = P(Y > X)$. Alternative H_1: not H_0.

The Mann-Whitney U test is similar to the Wilcoxon test, but can be used to compare multiple samples that aren't necessarily paired.

Parameters **m** (*list of numpy arrays*) – Sampled distributions.

Returns

- **cor** (*2 dimensional array of pvalues.*) – cor[i,j] is the p-value of the MWW test between the samples i and j.
- *Notes*
- ——
- *A p-value < 0.05 means that there is strong evidence to reject the null*
- *hypothesis.*
- *References*
- ——— –

1. **Mann-Whitney U test:** http://tqmp.org/RegularArticles/vol04-1/p013/p013.pdf
2. **Non parametric tests** http://www.mit.edu/~6.s085/notes/lecture5.pdf

- *Examples*
- ———
- >>> *cor = get_pvalues([np.array([1, 2, 3]), np.array([1, 1, 2])])*

mousestyles.mww.**plot_cor**(*data*)

Plot the p-values outputed by the Mann-Whitney U test using a correlation matrix representation.

Parameters **data** (*MWW_allmice output*) –

Returns

- **plot** (*correlation matrix*)
- *Examples*

- ———
- >>> *strains = MWW_strains()*
- >>> *plot_cor(strains)*

mousestyles.mww.**plot_cor_multi**(*mww_values*)

Vectorized version of plot_cor. Plot several correlation matrices side by side using plot_cor.

Parameters **data** (*MWW_allmice output*) –

Returns

- **plot** (*correlation matrix*)
- *Examples*
- ———
- >>> *allmice = MWW_allmice()*
- >>> *plot_cor_multi(allmice)*

3.1.9 Module contents

mousestyles is the final project for UC Berkeley's 2016 Masters Capstone Project class. It is based on ideas, code, and data from the Tecott Lab at UCSF.

[AP14] David J Anderson and Pietro Perona. Toward a science of computational ethology. *Neuron*, 84(1):18–31, 2014. http://www.sciencedirect.com/science/article/pii/S0896627314007934.

[GSJ+08] Evan H Goulding, A Katrin Schenk, Punita Juneja, Adrienne W MacKay, Jennifer M Wade, and Laurence H Tecott. A robust automated system elucidates mouse home cage behavioral structure. *Proceedings of the National Academy of Sciences*, 105(52):20575–20582, 2008. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2634928/.

[PDC+04] Petko M Petkov, Yueming Ding, Megan A Cassell, Weidong Zhang, Gunjan Wagner, Evelyn E Sargent, Steven Asquith, Victor Crew, Kevin A Johnson, Phil Robinson, and others. An efficient snp system for mouse genome scanning and elucidating strain relationships. *Genome research*, 14(9):1806–1811, 2004. http://genome.cshlp.org/content/14/9/1806/F3.expansion.

[Tec03] Laurence H Tecott. The genes and brains of mice and men. *American Journal of Psychiatry*, 2003. http://dx.doi.org/10.1176/appi.ajp.160.4.646.

[TN04] Laurence H Tecott and Eric J Nestler. Neurobehavioral assessment in the information age. *Nature neuroscience*, 7(5):462–466, 2004. http://www.nature.com/neuro/journal/v7/n5/full/nn1225.html.

m

N

P

R

S

T

U