
JABS

Release 0.1

KumarLab

Jul 24, 2023

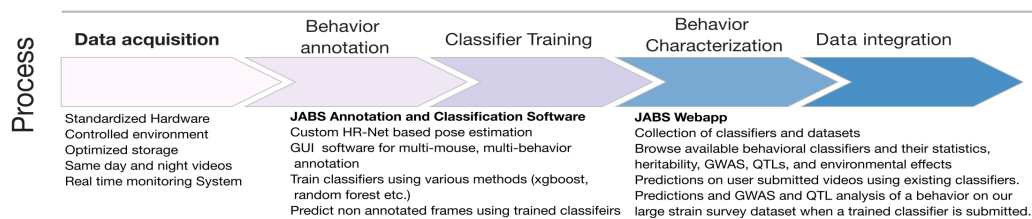
CONTENTS

1	Contents	3
1.1	Set up	3
1.2	JABS User Guide	5
1.3	JABS Video Tutorial	18
1.4	Advanced JABS usage	18
1.5	Introduction to the JABS downstream analysis	19
1.6	2022 Short Course on the Application of Machine Learning for Automated Quantification of Behavior	19

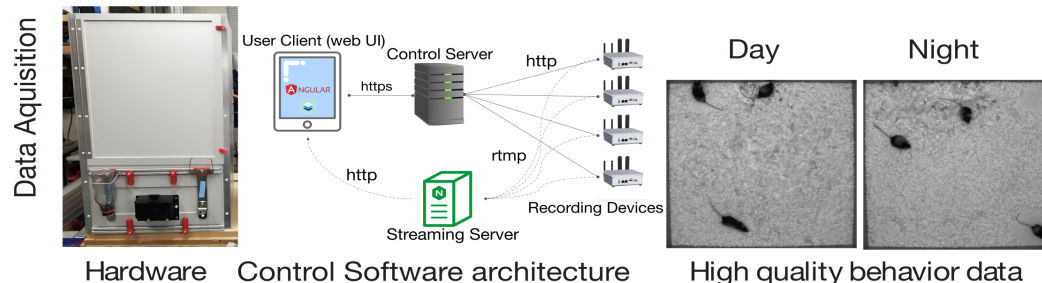
JABS (JAX Animal Behavior System) is an integrated rodent phenotyping platform to the community for data acquisition, machine learning based behavior annotation, classifier sharing and automated genetic analysis.

A

JAX Animal Behavior System (JABS)



B



Check out the [Set up](#) section for further information, including how to [Installation](#) the project.

Note: This project is under active development.

CONTENTS

1.1 Set up

1.1.1 Installation

Creating the Virtual Environment

You will need to create the virtual environment before you can run the labeler for the first time. The following commands will create a new Python3 virtual environment, activate it, and install the required packages. Note, your python executable may be named `python` or `python3` depending on your installation.

```
cd <path-to-JABS-folder>
python -m venv jabs.venv
source jabs.venv/bin/activate
pip install -r requirements.txt
```

Activating

The virtual environment must be activated before you can run the labeling interface. To activate, run the following command:

```
source jabs.venv/bin/activate
```

Deactivating

The virtual environment can be deactivated if you no longer need it:

```
deactivate
```

Installing on Apple M1/M2 Silicone

To install the app on Apple's newer M1/M2 macbooks, the user needs to install via [anaconda](#) using the following instructions:

```
conda env create -n jabs -f environment_jabs.yml
conda activate jabs
python app.py
```

Enabling XGBoost Classifier

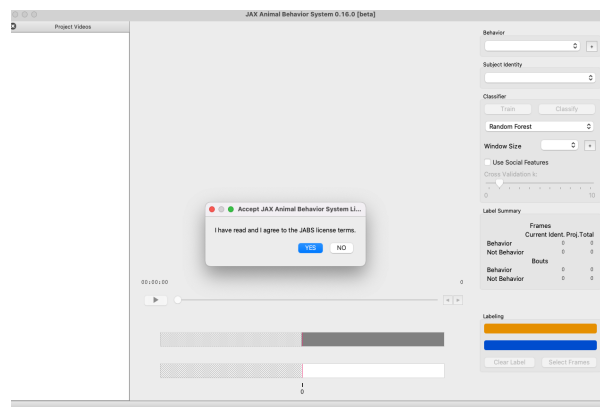
The XGBoost Classifier has a dependency on the OpenMP library. This does not ship with MacOS. XGBoost should work “out of the box” on other platforms. On MacOS, you can install libomp with Homebrew (preferred) with the following command `brew install libomp`.

1.1.2 Launching JABS GUI

To launch JABS from the command prompt, open a command prompt in the JABS directory and run the following commands:

```
jabs.venv\Scripts\activate.bat
python app.py
```

If everything runs smoothly, you should see a JABS startup window like the following:



1.1.3 Preparing the JABS Project

Once the JABS environment is activated, prepare your project folder. The folder should contain the videos for labeling and the corresponding pose file for each video. Once prepared, you may either proceed to open the JABS GUI or initialize the project folder prior to working using `initialize_project.py`.

```
python initialize_project.py <project_dir>
```

This will generate the JABS features for the project for the default window size of 5. The argument ‘-w’ can be used to set the initial window size for feature generation.

Starting up

You can open the JABS GUI with the command:

















```
python app.py
```

1.2 JABS User Guide

1.2.1 The JABS Project Directory

A JABS project is a directory of video files and their corresponding pose estimation files. The first time a project directory is opened in JABS, it will create a subdirectory called “rotta”, which contains various files created by JABS to save project state, including labels and current predictions.

Example JABS project directory listing:

Name	Kind
 007713_B6NJ_F_vehicle__36013_pose_est_v2.h5	HDF Files
 007713_B6NJ_F_vehicle__36013.avi	AVI movie
 007738_B6NJ_M_vehicle__11000_pose_est_v2.h5	HDF Files
 007738_B6NJ_M_vehicle__11000.avi	AVI movie
 007741_B6NJ_M_vehicle__29360_pose_est_v2.h5	HDF Files
 007741_B6NJ_M_vehicle__29360.avi	AVI movie
 L4-B2B+2019-10-11_KK+LL4-1_LL4-1_B6JF_027_2_68000_pose_est_v2.h5	HDF Files
 L4-B2B+2019-10-11_KK+LL4-1_LL4-1_B6JF_027_2_68000.avi	AVI movie
 Left_Turn_training_20220922_151104.h5	HDF Files
 LL1-B2B+2019-10-11_KK+LL1-1_LL1-1_B6JM_000_2_25710_pose_est_v2.h5	HDF Files
 LL1-B2B+2019-10-11_KK+LL1-1_LL1-1_B6JM_000_2_25710.avi	AVI movie
 LL1-B2B+2019-10-11_KK+LL1-1_LL1-1_B6JM_000_2_68000_pose_est_v2.h5	HDF Files
 LL1-B2B+2019-10-11_KK+LL1-1_LL1-1_B6JM_000_2_68000.avi	AVI movie
 LL1-B6+2021-06-01_SPD+AgedB6-0312_1500_pose_est_v2.h5	HDF Files
 LL1-B6+2021-06-01_SPD+AgedB6-0312_1500.avi	AVI movie

1.2.2 Initializing A JABS Project Directory

The first time you open a project directory in with JABS it will create the “rotta” subdirectory. Features will be computed the first time the “Train” button is clicked. This can be very time consuming depending on the number and length of videos in the project directory.

The `initialize_project.py` script can also be used to initialize a project directory before it is opened in the JABS GUI. This script checks to make sure that a pose file exists for each video in the directory, and that the pose file and video have the same number of frames. Then, after these basic checks, the script will compute features for all of the videos in the project. Since `initialize_project.py` can compute features for multiple videos in parallel, it is significantly faster than doing so through the GUI during the training process.

`initialize_project.py` usage:

```
initialize_project.py [-h] [-f] [-p PROCESSES] [-w WINDOW*SIZE]
                                [-\-\-force\-\-pixel\-\-distances]
                                project\_dir
```

positional arguments:

project_dir

optional arguments:

-h, --help show this help message and exit

-f, --force recompute features even if file already exists

-p PROCESSES, --processes PROCESSES

number of multiprocessing workers

-w WINDOW_SIZE Specify window sizes to use for computing window features.↵
↪Argument can

be repeated to specify multiple sizes (e.g.

↪ \-w 2 \-w 5). Size is number

of frames before and after the current.↵

↪frame to include in the window.

For example, '\-w 2' results in a window.↵

↪size of 5 (2 frames before, 2

frames after, plus the current frame). If.↵

↪no window size is specified,

a default of 5 will be used.

--force-pixel-distances

use pixel distances when computing.↵

↪features even if project supports cm

example initialize_project.py command

The following command runs the initialize_project.py script to compute features

using window sizes of 2, 5, and 10. The script will use up to 8 processes for

computing features (-p8). If no -p argument is passed, initialize_project.py

will use up to 4 processes.

1.2.3 The Rotta Directory

JABS creates a subdirectory called “rotta” inside the project directory (this directory is called “rotta” for historical reasons and may change prior to the 1.0.0 release of JABS). This directory contains app-specific data such as project settings, generated features, user labels, cache files, and the latest predictions.

project.json This file contains project settings and metadata.

rotta/annotations

This directory stores the user’s labels, stored in one JSON file per labeled video.

rotta/archive

This directory contains archived labels. These are compressed files (gzip) containing labels for behaviors that the user has removed from the project. Rotta only archives labels. Trained classifiers and predictions are deleted if a user removes a behavior from a project.

rotta/cache

Files cached by JABS to speed up performance. Some of these files may not be portable, so this directory should be deleted if a JABS project is copied to a different platform.

rotta/classifiers

This directory contains trained classifiers. Currently, these are stored in Python Pickle files and should be considered non-portable.

rotta/features

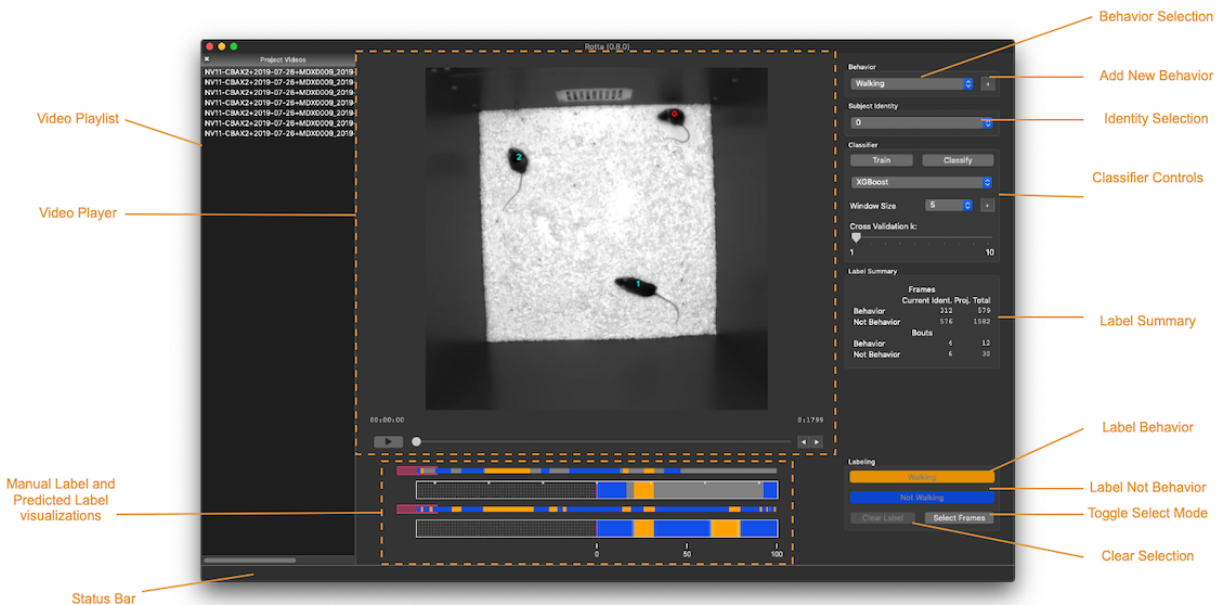
This directory contains the computed features. There is one directory per project video, and within each video directory there will be one feature directory per identity. Feature files are usually portable, but JABS may need to recompute the features if they were created with a different version of JABS.

rotta/predictions

This directory contains prediction files. There will be one subdirectory per behavior containing one prediction file per video. Prediction files are automatically opened and displayed by JABS if they exist. Prediction files are portable, and are the same format as the output of the command line classifier tool (*classify.py*).

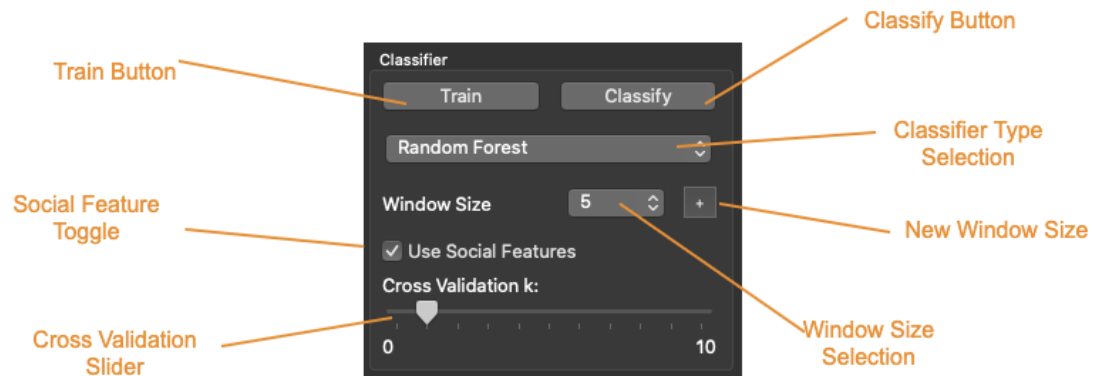
1.2.4 GUI

Main Window



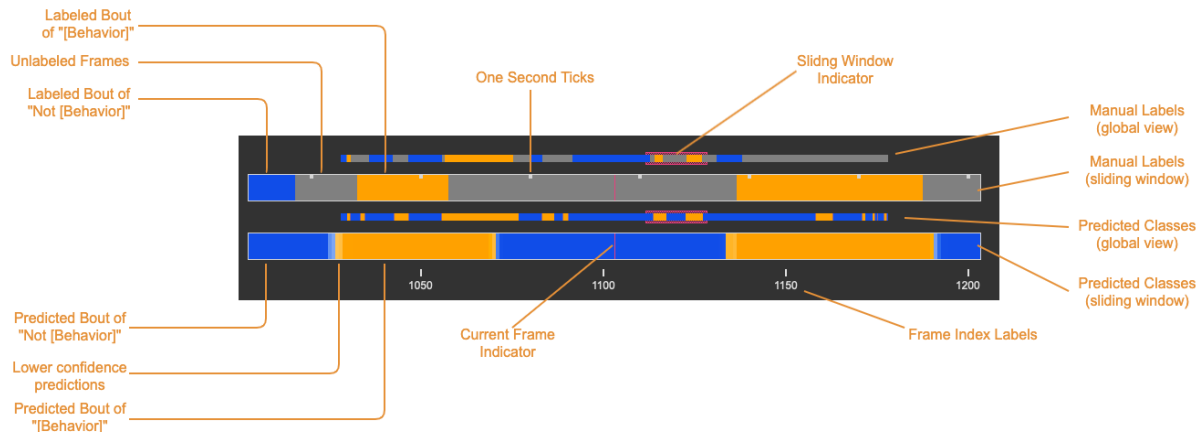
- **Behavior Selection:** Select current behavior to label
- **Add New Behavior Button:** Add new behavior label to project
- **Identity Selection:** Select subject mouse to label (subject can also be selected by clicking on mouse in the video)
- **Classifier Controls:** Configure and train classifier. Use trained classifier to infer classes for unlabeled frames. See “Classifier Controls” section for more details.
- **Label Summary:** Counts of labeled frames and bouts for the subject identity in the current video and across the whole project.
- **Label “Behavior” Button:** Label current selection of frames as showing behavior. This button is labeled with the current behavior name.
- **Label “Not Behavior” Button:** Label current selection of frames as not showing behavior. This button is labeled with “Not <current behavior name>”.
- **Clear Selection Button:** remove labels from current selection of frames
- **Toggle Select Mode Button:** toggle select mode on/off (turning select mode on will begin selecting frames starting from that point)
- **Video Playlist:** list of videos in the current project. Click a video name to make it the active video.
- **Video Player:** Displays the current video. See “Video Player” section for more information.
- **Manual Label and Predicted Label Visualizations:** see “Label Visualizations” for more information.
- **Status Bar:** Displays periodic status messages.

Classifier Controls



- **Train Button:** Train the classifier with the current parameters. This button is disabled until minimum number of frames have been labeled for a
minimum number of mice (increasing the cross validation k parameter increases
the minimum number of labeled mice)
- **Classify Button:** Infer class of unlabeled frames. Disabled until classifier is trained. Changing classifier parameters may require retraining
before the Classify button becomes active again.
- **Classifier Type Selection:** Users can select from a list of supported classifiers.
- **Window Size Selection:** Number of frames on each side of the current frame to include in window feature calculations for that frame. A “window size” of 5
means that 11 frames are included into the window feature calculations for
each frame (5 previous frames, current frame, 5 following frames).
- **New Window Size:** Add a new window size to the project.
- **Cross Validation Slider:** Number of “Leave One Out” cross validation iterations to run while training.
- **Social Feature Toggle:** Turn on/off social features (disabled if project includes pose file version 2). Allows training a classifier backwards
compatible with V2 pose files using V3 or higher poses.

Label and Prediction Visualizations

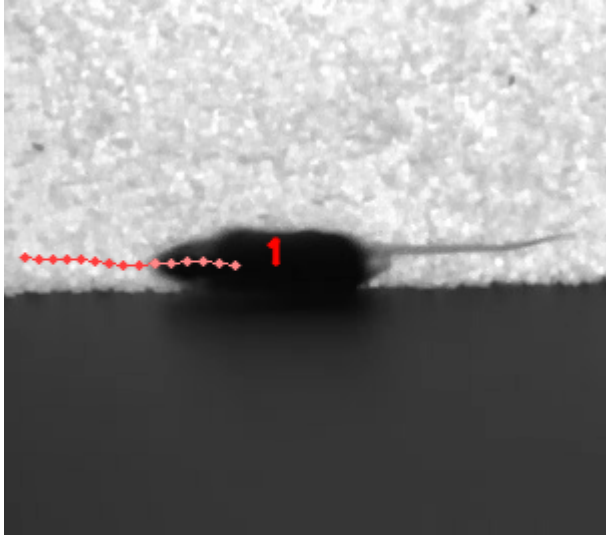
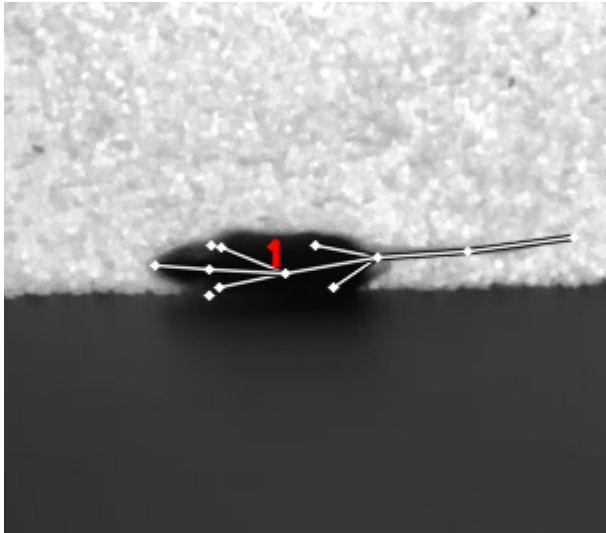


- **Manual Labels (sliding window):** Displays manually assigned labels for a sliding window of frames. The window range is the current frame ± 50 frames.
Orange indicates frames labeled as showing the behavior, blue indicates frames labeled as not showing the behavior. Unlabeled frames are colored gray.
- **Manual Labels (global view):** Displays a zoomed out view of the manual labels for the entire video
- **Predicted Classes (sliding window):** Displays predicted classes (if the classifier has been run). Color opacity indicates prediction probability for the predicted class. Manually assigned labels are also displayed with probability of 100%.
- **Predicted Class (global view):** Displays a zoomed out view of the predicted classes for the entire video.
- **Sliding Window Indicator:** highlights the section of the global views that correspond to the frames displayed in the “sliding window” views.

Menu

- **JABS→About:** Display About Dialog
- **JABS→User Guide:** Display User Guide
- **JABS→Quit JABS:** Quit Program
- **File→Open Project:** Select a project directory to open. If a project is already opened, it will be closed and the newly selected project will be opened.
- **File→Export Training Data:** Create a file with the information needed to share a classifier. This exported file is written to the project directory and has the form `<Behavior*Name>*training*<YYYYMMDD*hhmmss>.h5`. This file is used as one input for the `classify.py` script.

- **View→View Playlist:** can be used to hide/show video playlist
- **View→Show Track:** show/hide track overlay for the subject. The track overlay shows the nose position for the previous 5 frames and the next 10 frames. The nose position for the next 10 frames is colored red, and the previous 5 frames it is a shade of pink.
- **View→Overlay Pose:** toggle the overlay of the pose on top of the subject mouse
- **View→Overlay Landmarks:** toggle the overlay of arena landmarks over the video.

Track Overlay Example:**Pose Overlay Example:**

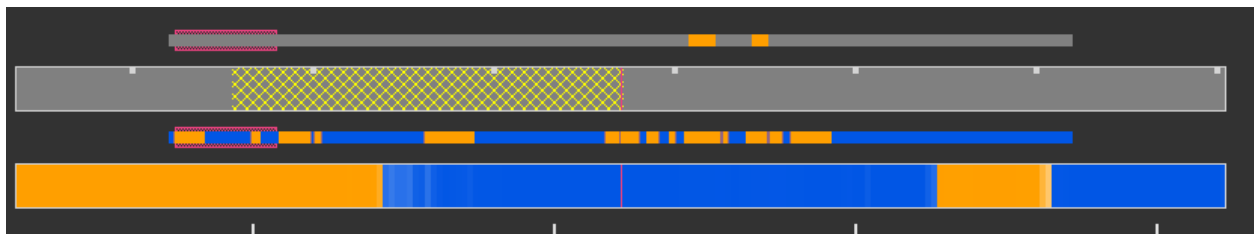
1.2.5 Labeling

This section describes how a user can add or remove labels. Labels are always applied to the subject mouse and the current subject can be changed at any time. A common way to approach labeling is to scan through the video for the behavior of interest, and then when the behavior is observed select the mouse that is showing the behavior. Scan to the start of the behavior, and begin selecting frames. Scan to the end of the behavior to select all of the frames that belong to the bout, and click the label button.

Selecting Frames

When “Select Mode” is activated, JABS begins a new selection starting at that frame. The current selection is from the selection start frame through the current frame. Applying a label, or removing labels from the selection clears the current selection and leaves “Select Mode”.

The current selection range is shown on the “Manual Labels” display:



Clicking the “Select Frames” button again or pressing the Escape key will unselect the frames and leave select mode without making a change to the labels.

Applying Labels

The “Label Behavior Button” will mark all of the frames in the current selection as showing the behavior. The “Label Not Behavior” button will mark all of the frames in the current selection as not showing the behavior. Finally, the “Clear Labels” button will remove all labels from the currently selected frames.

The “Label Behavior Button” will mark all of the frames in the current selection as showing the behavior. The “Label Not Behavior” button will mark all of the frames in the current selection as not showing the behavior. Finally, the “Clear Labels” button will remove all labels from the currently selected frames.

Keyboard Shortcuts

Using the keyboard controls can be the fastest way to label.

Navigation Keyboard Controls

The arrow keys can be used for stepping through video. The up arrow skips ahead 10 frames, and the down arrow skips back 10 frames. The right arrow advances one frame, and the left arrow goes back one frame.

Labeling Controls

The z, x, and c keys can be used to apply labels.

If in select mode:

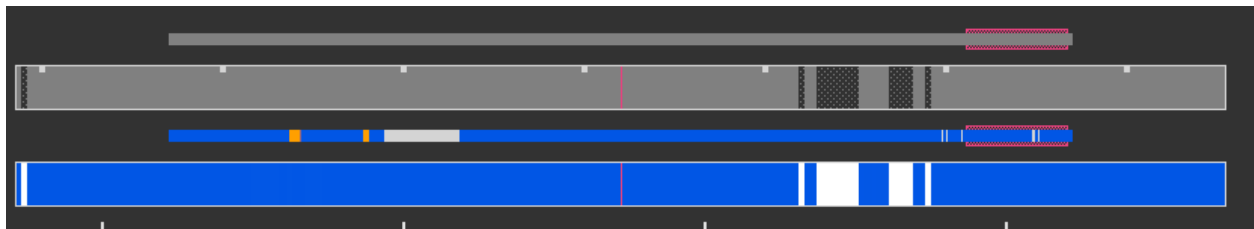
- **z:** label current selection as “behavior”
- **x:** clear labels from current selection
- **c:** label current selection as “not behavior”

If not in select mode:

- **z, x, c:** start selecting frames.

Identity Gaps

Identities can have gaps if the mouse becomes obstructed or the pose estimation failed for those frames. In the manual label visualization, these gaps are indicated with a pattern fill instead of the solid gray/orange/blue colors. In the predicted class visualization, the gaps are colored white.



1.2.6 All Keyboard Shortcuts

File Menu

Actions under the file menu have keyboard shortcuts.

- Control Q (Command Q on Mac) quit JABS
- Control T (Command T on Mac) export training data

Navigation

- left arrow: move to previous frame
- right arrow: move to next frame
- up arrow: move forward 10 frames (TODO: make configurable)
- down arrow: move back 10 frames (TODO: make configurable)
- space bar: toggle play/pause

Labeling

while in select mode:

- z: label current selection <behavior> and leave select mode
- x: clear current selection labels and leave select mode
- c: label current selection not <behavior> and leave select mode
- Escape: exit select mode without applying/clearing labels for current selection

while not in select mode:

- z, x, c: enter select mode

Other

- t: toggle track overlay for subject
- p: toggle pose overlay for subject
- l: toggle landmark overlay

1.2.7 The Command Line Classifier

JABS includes a script called *classify.py*, which can be used to classify a single video from the command line.

```
usage: classify.py COMMAND COMMAND_ARGS
```

```
commands:
```

```
classify    classify a pose file
```

```
train       train a classifier that can be used to classify multiple pose files
```

```
See `classify.py COMMAND --help` for information on a specific command.
```

```
usage: classify.py classify [-h] [--random-forest | --gradient-boosting | --xgboost]
                             (\-\-training TRAINING | \-\-
→ classifier CLASSIFIER) \-\-input\-\-pose
```

(continues on next page)

(continued from previous page)

```

INPUT\_POSE \- \-out \-dir OUT\_DIR_

[ \- \-fps FPS]

[ \- \-feature \-dir FEATURE\_DIR]

optional arguments:

  -h, --help            show this help message and exit
  --fps FPS              frames per second, default=30
  --feature-dir FEATURE_DIR

                          Feature cache dir. If present, look_
here for features before computing.

                          If features need to be computed, they_
will be saved here.

required arguments:

  --input-pose INPUT_POSE

                          input HDF5 pose file (v2, v3, or v4).

  --out-dir OUT_DIR      directory to store classification output
optionally override the classifier specified in the training file:
Ignored if trained classifier passed with --classifier option.

(the following options are mutually exclusive):

  --random-forest        Use Random Forest
  --gradient-boosting     Use Gradient Boosting
  --xgboost               Use XGBoost

```

Classifier Input (one of the following is required):

--training TRAINING Training data h5 file exported from JABS

—classifier CLASSIFIER

Classifier file produced from the *classify.py train* command

```

usage: classify.py train [-h] [--random-forest | --gradient-boosting | --xgboost]

                          training\_file out\_file

positional arguments:

  training_file          Training h5 file exported by JABS

```

(continues on next page)

(continued from previous page)

```
out_file          output filename

optional arguments:

-h, --help          show this help message and exit

optionally override the classifier specified in the training file:

(the following options are mutually exclusive):

--random-forest      Use Random Forest

--gradient-boosting  Use Gradient Boosting

--xgboost            Use XGBoost
```

Note: xgboost may be unavailable on Mac OS if libomp is not installed.

See *classify.py classify -help* output for list of classifiers supported in the current execution environment.

Note: fps parameter is used to specify the frames per second (used for scaling time unit for speed and velocity features from “per frame” to “per second”).

1.2.8 File Formats

This section documents the format of JABS output files that may be needed for downstream analysis.

Inference File

An inference file represents the predicted classes for each identity present in one video file.

Location

The prediction files are saved in *<JABS project dir>/rotta/predictions/<behavior*name>/<video*name>.h5* if they were generated by the JABS GUI. The *classify.py* script saves inference files in *<out-dir>/<behavior*name>/<video*name>.h5*

Contents

The H5 file contains one group, called “predictions”. This group contains three datasets

predictions

- predicted_class
- probabilities
- identity_to_track

The file also has some attributes:

- version: This attribute contains an integer version number, and will be incremented if an incompatible change is made to the file format.
- source*pose*major_version: integer containing the major version of the pose file that was used for the prediction

predicted_class

- dtype: 8-bit integer
- shape: #identities x #frames

This dataset contains the predicted class. Each element contains one of three values:

- 0: “not behavior”
- 1: “behavior”
- -1: “identity not present in frame”.

probabilities

- dtype: 32-bit floating point
- shape: #identities x #frames

This dataset contains the probability (0.0-1.0) of each prediction. If there is no prediction (the identity doesn’t exist at a given frame) then the prediction probability is 0.0.

identity_to_track

- dtype: 32-bit integer
- shape: #identities x #frames

This dataset maps each JABS-assigned identity (Pose version 3) back to the original track ID from the pose file at each frame. -1 indicates the identity does not map to a track for that frame. For Pose File Version 4 and greater, JABS uses the identity assignment contained in the pose file. For pose version 2, there will be exactly one identity (0).

1.3 JABS Video Tutorial

1.4 Advanced JABS usage

1.4.1 Active Learning Strategy in JABS

Active Learning is a semi-supervised Machine Learning (ML) strategy. The basic principle is that the data set and the model are built simultaneously through a ML-human loop. The labeled data is incrementally added to the model, retraining it. The retrained model is used to synthetically label the unlabeled data set. The labeler is then able to pick the most useful unlabeled data to label next (Correcting mistakes). Since not all datapoints are equally valuable, Active Learning can lower the amount of labels needed to produce the best results so it is especially suited for videos.

Therefore, when you are labeling in JABS, employ an iterative process of labeling a couple videos, training+predicting, and then correcting some more labels, then training and predicting again. This will cut down on the total amount of labels needed to train a strong classifier by allowing you to only label frames which are most useful to the classifier.

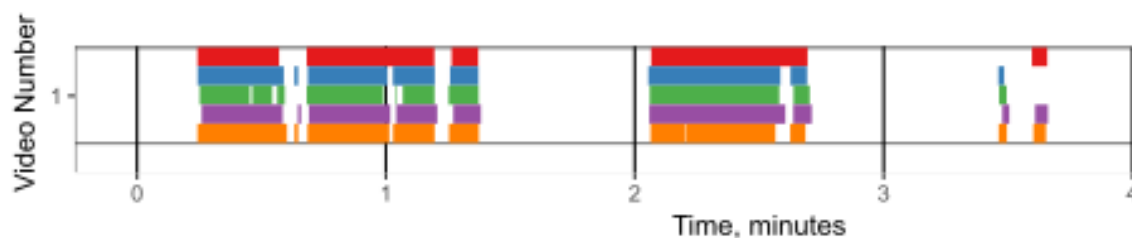
1.4.2 Finding new videos

If you have labelled most of the videos in your project and are not satisfied with the amount of examples of that behavior in the project, you can use a weaker classifier trained on that project to find new videos to add to your project. Use your current classifier to infer on a larger available dataset. The videos which have the most predictions for your behavior can be added to your folder for more behavior examples and more examples of mistakes to correct.

1.4.3 Ground Truth Validation

Once you have a classifier you want to validate, it may be helpful to densely label a set of ground truth videos to test your classifier on. The simplest way to validate is a frame-based approach of counting the amount of true positives, false positives, true negatives, and false negatives for your classifier. You can use these to calculate Precision, Recall, Accuracy, and F1 beta.

However, it is important to note that while machine learning techniques are often evaluated using frame accuracies, behaviorists may find detecting the same bouts of behavior more important than the exact starting and ending frame of these bouts. Even between two humans labeling the same behavior, there are unavoidable discrepancies in the exact frames of the behavior. Consider the following example where 4 annotators labeled grooming bouts:



Though many of the same bouts are found by all annotators, taking a frame-based approach may cause the agreement between annotators to seem deceptively low. Therefore, you may wish to pursue a bout-based agreement for validation.

1.5 Introduction to the JABS downstream analysis

1. Importing necessary libraries

We are just going to import basic python libraries to help us analyze predicted behavior

```
[7]: # auto reload
%load_ext autoreload
%autoreload 2
```

```
[8]: from stuff import *
from ipywidgets import FloatSlider, interactive, fixed
```

2. Visualizing the Ethogram for a sample prediction

```
[14]: # Reading the sample file
df = load_file('data/sample_file.h5')
int_plot = interactive(draw_ethogram, df=fixed(df['Class'].values.astype(int)),
    i=FloatSlider(min=1, max=100, step=1), save_fig=False);
int_plot

interactive(children=(FloatSlider(value=1.0, description='i', min=1.0, step=1.0),
    Checkbox(value=False, descri...
```

2. Frame counting statistics

1.6 2022 Short Course on the Application of Machine Learning for Automated Quantification of Behavior

1.6.1 Day 4 Workshop - Training Classifiers to Classify Animal Behavior

1.6.2 Purpose/Expectations

The contents of this course will grant insight into important characteristics while training behavioral classifiers. The usage of JABS code in this tutorial is strictly limited to reading in project annotations and feature vectors.

Within this tutorial, you will be provided with 2 example annotated projects that contain sparse labels that will be used for training and dense labels that you will use for final evaluation of your best model.

Expected knowledge before coming doing the course

- Familiarity with animal pose data and frame-level features derived from that data
- Basic python experience
- Some numpy experience
- Familiarity with generating plots in python (preference to using plotnine, which is ggplot-style syntax)

Expected takeaways of the course

- Learn the core process to train a frame-wise predictor of behavior
- Become familiar with key components to take into account when training these types of classifiers
- Gain insight for some of the design decision considerations our group integrated into the JABS software

This is a static web-version of the ipython notebook found [here](#).

Step 0

Import a lot of libraries that we will be using in this tutorial

```
import numpy as np
import pandas as pd
from itertools import chain
import sklearn
# JABS install directory
import sys
sys.path.append('JABS-behavior-classifier/')
from src.project import Project
from src.classifier import Classifier
# Plotting library
import plotnine as p9
%matplotlib notebook
```

Step 1

Import the project annotations that we are providing These annotations were created using our JABS software, so we can use that library to extract the annotations in python

```
project = Project('JABS-Training-Data/')
# We're picking the first behavior in the project. Each JABS project can contain
↳ multiple behavior annotations.
behavior = project.load_metadata()['behaviors'][0]
# Here we read in the all important information for all labels
training_data = project.get_labeled_features(behavior, window_size=5, use_social_
↳ features=True)
# Since this is more of an internal format, we can separate out the data to be more
↳ easily accessible
all_labels = training_data[0]['labels']
annotation_animal = training_data[0]['groups']
annotation_animal_names = [training_data[1][x]['video'] + ' animal ' + str(training_
↳ data[1][x]['identity']) for x in annotation_animal]
feature_names = training_data[0]['column_names']
all_features = Classifier().combine_data(training_data[0]['per_frame'], training_data[0][
↳ 'window'])
```

Also extract some bout-level groups, something that JABS doesn't normally do

```
raw_annotations = [project.load_video_labels(training_data[1][x]['video']).as_dict()[
↳ 'labels'][str(training_data[1][x]['identity'])][behavior] for x in np.
↳ unique(annotation_animal)]
num_animals_annotated = len(raw_annotations)
bout_data = [[np.repeat(animal_idx+bout_idx*num_animals_annotated, bout['end']-bout[
↳ 'start']+1) for bout_idx, bout in enumerate(animal_data)] for animal_idx, animal_data
↳ in enumerate(raw_annotations)]
bout_data = np.concatenate(list(chain.from_iterable(bout_data)))
```


At this point we have extracted feature data and annotations from a JABS project. Here's a description of important variables:

- `all_features` contains the matrix of all the features. The first dimension is the frame index and the second dimension is the feature index.
- `all_labels` contains a vector of labels. 0 = not behavior, 1 = behavior
- `annotation_animal` contains a vector with a unique number for each animal
- `bout_data` contains a vector with a unique bout number for each bout annotated * `feature_names` contains the name of each feature. Each value here is the name of the 2nd dimension of the feature vector.

Step 2

Read in the held-out test set

This set is distinct from the validation dataset we'll be splitting in 2 key factors: 1. This set will be held out and should not be used in tuning model parameters. 2. It is also densely annotated (all frames contain a label), rather than the sparse annotation provided for training.

```
test_project = Project('JABS-Test-Data/')
test_data = test_project.get_labeled_features(behavior, window_size=5, use_social_
↪ features=True)
test_labels = test_data[0]['labels']
test_animals = test_data[0]['groups']
test_features = Classifier().combine_data(test_data[0]['per_frame'], test_data[0]['window
↪ '])
```

Step 3

Inspect characteristics of the dataset Since the annotations are located in `all_labels`, we should inspect some characteristics of it

Question 1

How many annotations do we have?

```
print(len(all_labels))
```

```
3407
```

Question 2

How many labels do we have of each class: 0 (not-behavior) and 1 (behavior)

```
num_not_behavior = sum(all_labels==0)
print(num_not_behavior)
num_behavior = sum(all_labels==1)
print(num_behavior)
```

```
2294
1113
```

Question 3-4

How many bouts were annotated?

How many animals were annotated?

```
num_bouts = len(np.unique(bout_data))
print(num_bouts)
num_animals = len(np.unique(annotation_animal))
print(num_animals)
```

```
235
21
```

Discussion 1

What are characteristics of the dataset that may be important about creating a good model?

Hypotheticals to think about:

- If you have a rare behavior, is it okay to label a lot more “not behavior”? What might the model try and do to improve accuracy if that ratio becomes 1000:1?
- If one animal tends to express the behavior more than another, is it okay to provide more labels on from that individual?

Discussion on the balancing of data

Rules of thumb:

- Balanced annotations tend to create better classifiers. The higher the degree of imbalance, the more likely the classifier will just cheat and learn to predict one state over another. Generally, you should aim to not have more than a 2:1 ratio, but performance only particularly degrades when worse than a 5:1 or 10:1 ratio. Re-sampling can address this.
- More bouts is generally better, because within-bout annotations have high temporal correlation and are therefore less informative for making decisions that generalize in favor of making decisions specific to that bout.
- More animals is generally better, because some animals may express behavior in a more unique style. Having more animals enables better generalization. We’ve observed in the past that limiting to specific strains of animals in training hurts generalization to different strains. This keys in on adequately sampling from the population variation of the experiments you plan on running inferences on - include at least some of each genotype.

Extra visualizations of the dataset

```
plotting_df = pd.DataFrame({'annotations':all_labels, 'bout':bout_data, 'animal':
    ↳annotation_animal, 'annotation_idx':np.arange(len(all_labels))})
# Histogram of the class labels
plot_labels = p9.ggplot() + \
    p9.geom_histogram(mapping=p9.aes(x='factor(annotations)'), data=plotting_df,
    ↳bins=2, fill='#9e9e9e', color='#000000', size=2) + \
    p9.labs(title='Class Labels', x='Class', y='Count') + \
    p9.scale_x_discrete(labels=['Not-Behavior',behavior]) + \
    p9.theme_bw()
plot_labels.draw().show()

# Class labels per-animal
plot_animals = p9.ggplot() + \
    p9.geom_histogram(mapping=p9.aes(x='factor(annotations)'), data=plotting_df,
    ↳bins=2, fill='#9e9e9e', color='#000000', size=2) + \
    p9.labs(title='Class Labels by Animal', x='Class', y='Count') + \
    p9.scale_x_discrete(labels=['Not-Behavior',behavior]) + \
    p9.facet_wrap('animal') + \
    p9.theme_bw()
plot_animals.draw().show()
```

Example Output:

Step 4

Splitting the training data into train and validation

We will be using the training portion of the data to allow the algorithm to learn the best parameters to make predictions

The validation will be held out to evaluate performance of the classifier

Here we define a handful of functions that accepts the features and labels and returns the split data

```
# Defining functions to split the data
# Here we're going to manually shuffle and split
# Naive approach 1 - random sorted split (deterministic)
def split_data(features, labels, percent_train=0.75):
    available_examples = np.arange(len(labels))
    train_idx = available_examples[:int(len(labels)*percent_train)]
    test_idx = available_examples[int(len(labels)*percent_train):]
    # Separate out the training data
    train_features = features[train_idx]
    train_labels = labels[train_idx]
    # Separate out the validation data
    valid_features = features[test_idx]
    valid_labels = labels[test_idx]
    return train_features, train_labels, valid_features, valid_labels
```

```
# Naive approach 2 - random shuffle
def random_split_data(features, labels, percent_train=0.75):
    available_examples = np.arange(len(labels))
```

(continues on next page)

(continued from previous page)

```

np.random.shuffle(available_examples)
train_idx = available_examples[:int(len(labels)*percent_train)]
test_idx = available_examples[int(len(labels)*percent_train):]
# Separate out the training data
train_features = features[train_idx]
train_labels = labels[train_idx]
# Separate out the validation data
valid_features = features[test_idx]
valid_labels = labels[test_idx]
return train_features, train_labels, valid_features, valid_labels

```

```

# Using sklearn to split the data
# Stratified splitting:
# Attempts to preserve the train/valid class representations
def sklearn_stratified_split(features, labels, percent_train=0.75):
    train_idx, test_idx = list(sklearn.model_selection.StratifiedShuffleSplit(n_
↳ splits=1, train_size=percent_train).split(features, labels))[0]
    # Separate out the training data
    train_features = features[train_idx]
    train_labels = labels[train_idx]
    # Separate out the validation data
    valid_features = features[test_idx]
    valid_labels = labels[test_idx]
    return train_features, train_labels, valid_features, valid_labels

```

```

# Group splitting
# Leaves one group out within the split
# Note that this one group that is left out is not guaranteed to contain both labels, so
↳ sometime performance will contain things like division by 0.
def sklearn_logo_split(features, labels, groups, percent_train=0.75):
    train_idx, test_idx = list(sklearn.model_selection.LeaveOneGroupOut().
↳ split(features, labels, groups))[0]
    # Separate out the training data
    train_features = features[train_idx]
    train_labels = labels[train_idx]
    # Separate out the validation data
    valid_features = features[test_idx]
    valid_labels = labels[test_idx]
    return train_features, train_labels, valid_features, valid_labels

```

```

# Group splitting #2
# Leaves multiple groups out with a target percent annotations that fall into the
↳ training set
def sklearn_group_split(features, labels, groups, percent_train=0.75):
    train_idx, test_idx = list(sklearn.model_selection.GroupShuffleSplit(n_splits=1,
↳ train_size=percent_train).split(features, labels, groups))[0]
    # Separate out the training data
    train_features = features[train_idx]
    train_labels = labels[train_idx]
    # Separate out the validation data
    valid_features = features[test_idx]

```

(continues on next page)

(continued from previous page)

```
valid_labels = labels[test_idx]
return train_features, train_labels, valid_features, valid_labels
```

Experiment 1

Inspect/Visualize the effects of different splits of the annotations

```
train_features, train_labels, valid_features, valid_labels = sklearn_group_split(all_
↪ features, all_labels, annotation_animal)
train_df = pd.DataFrame({'state': 'train', 'label': train_labels})
valid_df = pd.DataFrame({'state': 'valid', 'label': valid_labels})
plot_df = pd.concat([train_df, valid_df])

split_plot = p9.ggplot(plot_df) + \
    p9.geom_bar(p9.aes(x='state', fill='factor(label)')) + \
    p9.theme_bw()

split_plot.draw().show()
```

Example Output:

Label Balancing

Since we're aware that we have roughly a 3x more annotations for not-behavior, we should probably consider balancing the labels

We can take multiple approaches to creating a more balanced dataset: 1. Downsampling: Discard data from the class which has the most annotation until the labels are balanced 2. Upsampling: Duplicate data from the class which has the least annotation until the labels are balanced

```
# Downsamples the features, labels, and groups such that the labels are balanced
def downsample_balanced(features, labels, groups=None):
    _, label_counts = np.unique(labels, return_counts=True)
    smallest_class_count = np.min(label_counts)
    class_0_idx = np.where(labels==0)[0]
    class_1_idx = np.where(labels==1)[0]
    class_0_idx = np.random.choice(class_0_idx, smallest_class_count, replace=False)
    class_1_idx = np.random.choice(class_1_idx, smallest_class_count, replace=False)
    selected_samples = np.sort(np.concatenate([class_0_idx, class_1_idx]))
    new_features = features[selected_samples,:]
    new_labels = labels[selected_samples]
    if groups is not None:
        new_groups = groups[selected_samples]
        return new_features, new_labels, new_groups
    else:
        return new_features, new_labels, None
```

Question 5

Write a function to upsample the data

```
# Downsamples the features, labels, and groups such that the labels are balanced
def upsample_balanced(features, labels, groups=None):
    _, label_counts = np.unique(labels, return_counts=True)
    largest_class_count = np.max(label_counts)
    class_0_idx = np.where(labels==0)[0]
    class_1_idx = np.where(labels==1)[0]
    class_0_idx = np.random.choice(class_0_idx, largest_class_count, replace=True)
    class_1_idx = np.random.choice(class_1_idx, largest_class_count, replace=True)
    selected_samples = np.sort(np.concatenate([class_0_idx, class_1_idx]))
    new_features = features[selected_samples,:]
    new_labels = labels[selected_samples]
    if groups is not None:
        new_groups = groups[selected_samples]
        return new_features, new_labels, new_groups
    else:
        return new_features, new_labels, None
```

Experiment 2

Inspect/Visualize the effects of downsampling the annotations

```
balanced_features, balanced_labels, balanced_groups = downsample_balanced(all_features,
    ↪all_labels, annotation_animal)
train_features, train_labels, valid_features, valid_labels = sklearn_group_
    ↪split(balanced_features, balanced_labels, balanced_groups)
train_df = pd.DataFrame({'state':'train', 'label':train_labels})
valid_df = pd.DataFrame({'state':'valid', 'label':valid_labels})
plot_df = pd.concat([train_df, valid_df])

split_plot = p9.ggplot(plot_df) + \
    p9.geom_bar(p9.aes(x='state', fill='factor(label)')) + \
    p9.theme_bw()

split_plot.draw().show()
```

Example Output:

Step 5

Train classifiers

For the purposes of this tutorial, we will be relying on using SKLearn to training classifiers.

Good recommended classifiers: 1. [Adaboost](#) 2. [Random Forest](#)

```
# Define how to train a decision tree classifier
def train_dt_classifier(train_features, train_labels):
    # Create a classifier object
```

(continues on next page)

(continued from previous page)

```

# Note here we are using a bunch of default values that SKLearn has picked for us.
# Check the documentation of the various parameters you can adjust for the decision.
→ trees:
# https://scikit-learn.org/stable/modules/generated/sklearn.tree.
→ DecisionTreeClassifier.html#sklearn-tree-decisiontreeclassifier
classifier = sklearn.tree.DecisionTreeClassifier()
# Train the classifier on our training features + labels
classifier = classifier.fit(train_features, train_labels)
return classifier

```

Question 6

Write functions to train an adaboost and random forest classifier

```

# Adaboost Classifier
def train_ada_classifier(train_features, train_labels):
    classifier = sklearn.ensemble.AdaBoostClassifier()
    classifier = classifier.fit(train_features, train_labels)
    return classifier

```

```

# Random forest Classifier
def train_rf_classifier(train_features, train_labels):
    classifier = sklearn.ensemble.RandomForestClassifier()
    classifier = classifier.fit(train_features, train_labels)
    return classifier

```

Step 6

Frame-wise evaluation of the classifier

Since our classifier predicts for every frame, we can estimate its performance by comparing the held-out validation data with the predictions it is making.

For evaluating classifier performance, we must first calculate the 4 values of a confusion matrix: 1. True positives (TP): When both the classifier and the ground truth assign “behavior” 2. False negatives (FN): When the classifier predicts “not behavior” and the ground truth assigns “behavior” 3. False positives (FP): When the classifier predicts “behavior” and the ground truth assigns “not behavior” 4. True negatives (TN): When both the classifier and the ground truth assign “not behavior”

From these, we can calculate 4 additional useful performance metrics for describing the classifier. Wikipedia has a [good article](#) outlining these as well as more, but these are the 4 most common. 1. Accuracy: The total number of predictions that were correct. 2. Precision: Of the predictions for “behavior”, how many were correct? 3. Recall: Of the ground truths labeled “behavior”, how many did the classifier predict? 4. F-beta (F1, F-score): Harmonic mean of precision and recall

Question 7

Complete the definitions of the confusion matrix and the performance metrics below

```
# Routine for evaluating the classifier performance
def eval_classifier_performance(classifier, features, labels, print_results=True):
    # Predict on held-out set
    predictions = classifier.predict(features)
    # Accuracy
    # Students will write these
    acc = np.mean(predictions == labels)
    # Confusion matrix values for a binary classifier
    tp_count = np.sum(np.logical_and(predictions==1, labels==1))
    tn_count = np.sum(np.logical_and(predictions==0, labels==0))
    fp_count = np.sum(np.logical_and(predictions==1, labels==0))
    fn_count = np.sum(np.logical_and(predictions==0, labels==1))
    # Calculate other metrics
    # Of the predictions we made, how many were correct
    precision = tp_count/(tp_count + fp_count)
    # Of the things we were looking for, how many did we find
    recall = tp_count/(tp_count + fn_count)
    # Harmonic mean of Pr and Re
    f1 = 2*(precision * recall)/(precision + recall)
    if print_results:
        print('Accuracy: ' + str(acc))
        print('Precision: ' + str(precision))
        print('Recall: ' + str(recall))
        print('F1-score: ' + str(f1))
    # We can also show that sklearn has tools for providing these metrics
    # Note that we've calculated the "precision" and "recall" for the behavior (value == 1)
    # print(sklearn.metrics.classification_report(labels, predictions))
    # Manually obtaining each
    # sklearn.metrics.precision_score(labels, predictions)
    # sklearn.metrics.recall_score(labels, predictions)
    # sklearn.metrics.f1_score(labels, predictions)
    return acc, precision, recall, f1
```

Step 7

Run through the steps: 1. Split the training data into train/valid 2. Train a classifier on the train portion of the split 3. Evaluate the classifier

```
# Step 1: Split the data into train/valid
train_features, train_labels, valid_features, valid_labels = random_split_data(all_
    features, all_labels)
# Train a classifier
dt_classifier = train_dt_classifier(train_features, train_labels)
# Evaluate the classifier
eval_classifier_performance(dt_classifier, valid_features, valid_labels)
# Repeat these steps to test how different parameters influence performance
```



```
Accuracy: 0.9647887323943662
Precision: 0.9651567944250871
Recall: 0.9326599326599326
F1-score: 0.9486301369863013
```

Experiment 3

See performance of different classifiers and sampling strategies

```
# Example with extra step for balanced labels
balanced_features, balanced_labels, balanced_groups = downsample_balanced(all_features,
    ↪ all_labels, annotation_animal)
train_features, train_labels, valid_features, valid_labels = sklearn_group_
    ↪ split(balanced_features, balanced_labels, balanced_groups)
dt_classifier = train_dt_classifier(train_features, train_labels)
eval_classifier_performance(dt_classifier, valid_features, valid_labels)
```

```
Accuracy: 0.7521663778162911
Precision: 0.7605633802816901
Recall: 0.7422680412371134
F1-score: 0.7513043478260869
```

Experiment 4

Expand the search of parameters by also adjusting the parameters of the model, eg the number of trees in the random forest.

We provide an example structure for looping through 10 training splits for the decision tree classifier.

Suggested tests: 1. Change the classifier type 2. Change the train/valid split approach 3. Include balanced data 4. Test various parameters of the classifier

```
all_results = []
# random splits
for test_model in np.arange(10):
    train_features, train_labels, valid_features, valid_labels = random_split_data(all_
    ↪ features, all_labels)
    dt_classifier = train_dt_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(dt_classifier, valid_features, valid_
    ↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier': ['decision tree'], 'split_method': [
    ↪ 'random'], 'accuracy': [acc], 'precision': [pr], 'recall': [re], 'f1': [f1]}))
    ada_classifier = train_ada_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(ada_classifier, valid_features, valid_
    ↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier': ['ada boost'], 'split_method': ['random
    ↪ '], 'accuracy': [acc], 'precision': [pr], 'recall': [re], 'f1': [f1]}))
    rf_classifier = train_rf_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(rf_classifier, valid_features, valid_
    ↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier': ['random forest'], 'split_method': [
```

(continues on next page)

(continued from previous page)

```

↪ 'random'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]))

# leave bouts out splits
for test_model in np.arange(10):
    train_features, train_labels, valid_features, valid_labels = sklearn_group_split(all_
↪ features, all_labels, bout_data)
    dt_classifier = train_dt_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(dt_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['decision tree'], 'split_method':[
↪ 'leave bouts out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))
    ada_classifier = train_ada_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(ada_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['ada boost'], 'split_method':['leave_
↪ bouts out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))
    rf_classifier = train_rf_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(rf_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['random forest'], 'split_method':[
↪ 'leave bouts out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))

# leave animals out splits
for test_model in np.arange(10):
    train_features, train_labels, valid_features, valid_labels = sklearn_group_split(all_
↪ features, all_labels, annotation_animal)
    dt_classifier = train_dt_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(dt_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['decision tree'], 'split_method':[
↪ 'leave animal out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))
    ada_classifier = train_ada_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(ada_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['ada boost'], 'split_method':['leave_
↪ animal out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))
    rf_classifier = train_rf_classifier(train_features, train_labels)
    acc, pr, re, f1 = eval_classifier_performance(rf_classifier, valid_features, valid_
↪ labels, print_results=False)
    all_results.append(pd.DataFrame({'classifier':['random forest'], 'split_method':[
↪ 'leave animal out'], 'accuracy':[acc], 'precision':[pr], 'recall':[re], 'f1':[f1]}))

# Flatten the list of dataframes into one
results_df = pd.concat(all_results)

```

Inspect the results of your scan(s)

```

accuracy_plot = p9.ggplot(data=results_df) + \
    p9.geom_point(p9.aes(x='classifier', y='accuracy')) + \
    p9.facet_wrap('split_method') + \
    p9.labs(title='Classifier accuracy', x='Classifier type', y='Accuracy') + \
    p9.theme_bw()

```

(continues on next page)

(continued from previous page)

```
accuracy_plot.draw().show()
```

Example Output:

Step 8

Evaluate your best classifier on the held-out test dataset

You could also potentially just train a classifier on the entire training set (after you've tuned the numbers with the validation set)

```
#best_classifier = dt_classifier
best_classifier = train_rf_classifier(all_features, all_labels)
eval_classifier_performance(best_classifier, test_features, test_labels)
```

```
Accuracy: 0.9844444444444445
Precision: 0.8036332179930796
Recall: 0.6761280931586608
F1-score: 0.7343873517786562
```

You may observe that performance has dropped a bit, but if you selected the correct train/valid split approach and didn't over-fit on your validation data, it should still perform well.

Discussion 2

Up until now, we've been focussing on frame-level agreement, which is simple enough for training classifiers.

Are there better ways to evaluate performance of a behavior classifier?

One approach could be evaluating dense annotations for bout-level agreement Other notes on the importance of bout-level agreement – eg filter/stitching/other post-processing steps

Step 9

Bout-level agreement on dense annotation

In order to measure bout-level agreement, we need to transform the data into starts and ends

We can either write the detection of bouts within predictions manually or re-use a classical compression algorithm called run length encoding

Note that typical RLE algorithms will encode all states (both "behavior" and "not-behavior"), while we primarily care about only 1 of the 2 states.

```
# Run length encoding, implemented using numpy
# Accepts a 1d vector
# Returns a tuple containing (starts, durations, values)
def rle(inarray):
    ia = np.asarray(inarray)
    n = len(ia)
    if n == 0:
```

(continues on next page)

(continued from previous page)

```

        return (None, None, None)
    else:
        y = ia[1:] != ia[:-1]
        i = np.append(np.where(y), n - 1)
        z = np.diff(np.append(-1, i))
        p = np.cumsum(np.append(0, z))[:-1]
        return(p, z, ia[i])

```

Use the RLE algorithm to encode predictions

We also need to separate the data by animal, because bouts can only exist within animal

```

test_predictions = best_classifier.predict(test_features)

# Arrays to store the list of bouts by animal
all_pr_bouts = []
all_gt_bouts = []
# Loop over each animal and add their bouts to the lists
for animal_id in np.unique(test_animals):
    animal_indices = test_animals==animal_id
    test_bout_predictions = rle(test_predictions[animal_indices])
    # Only store the bouts of behavior
    behavior_bouts = test_bout_predictions[2]==1
    if np.any(behavior_bouts):
        test_bout_predictions = (test_bout_predictions[0][behavior_bouts], test_bout_
↳ predictions[1][behavior_bouts])
        all_pr_bouts.append(test_bout_predictions)
    else:
        all_pr_bouts.append(([], []))
    test_bout_gt = rle(test_labels[animal_indices])
    # Only store the bouts of behavior
    behavior_bouts = test_bout_gt[2]==1
    if np.any(behavior_bouts):
        test_bout_gt = (test_bout_gt[0][behavior_bouts], test_bout_gt[1][behavior_bouts])
        all_gt_bouts.append(test_bout_gt)
    else:
        all_gt_bouts.append(([], []))

```

Display the predictions next to their GT

```

bout_data = []
for animal_idx, animal_id in enumerate(np.unique(test_animals)):
    gt_bout_data = all_gt_bouts[animal_idx]
    if len(gt_bout_data[0])>0:
        gt_bout_df = pd.DataFrame({'state':'GT', 'animal':animal_id, 'animal_idx':animal_
↳ idx, 'start_time':gt_bout_data[0], 'end_time':gt_bout_data[0]+gt_bout_data[1]})
        bout_data.append(gt_bout_df)
    pr_bout_data = all_pr_bouts[animal_idx]
    if len(pr_bout_data[0])>0:
        pr_bout_df = pd.DataFrame({'state':'PR', 'animal':animal_id, 'animal_idx':animal_
↳ idx, 'start_time':pr_bout_data[0], 'end_time':pr_bout_data[0]+pr_bout_data[1]})
        bout_data.append(pr_bout_df)

```

(continues on next page)

(continued from previous page)

```

bout_data = pd.concat(bout_data)
bout_plot = p9.ggplot(bout_data) + \
    p9.geom_rect(p9.aes(xmin='start_time', xmax='end_time', ymin='animal_idx-0.25', ymax=
    ↪ 'animal_idx+0.25', fill='factor(state)'), alpha=0.5) + \
    p9.labs(title='Bout annotations and predictions', x='Time, frame', y='Animal index',
    ↪ fill='State') + \
    p9.theme_bw()

bout_plot.draw().show()

```

Example Output:

Step 10

Strategies for evaluating performance of bout-based metrics

Since predictions are almost never going to be identical to the ground truth, we need to adjust our strategy for calling correct and incorrect classifications of the confusion matrix. To do this, we should attempt to detect how much the predictions overlap.

In classical image processing, this was done via calculating the intersection over union (IoU) between predictions and ground truth. We can adopt the same technique for our 1-D problem (time).

```

# Calculates the intersection of 2 bouts
# Each bout is a tuple of (start, duration)
def calculate_intersection(gt_bout, pr_bout):
    # Detect the larger of the 2 start times
    max_start_time = np.max([gt_bout[0], pr_bout[0]])
    # Detect the smaller of the 2 end times
    gt_bout_end = gt_bout[0]+gt_bout[1]
    pr_bout_end = pr_bout[0]+pr_bout[1]
    min_end_time = np.min([gt_bout_end, pr_bout_end])
    # Detect if the 2 bouts intersected at all
    if max_start_time < min_end_time:
        return min_end_time-max_start_time
    else:
        return 0

# Calculates the union of 2 bouts
# Each bout is a tuple of (start, duration)
def calculate_union(gt_bout, pr_bout):
    # If the 2 don't intersect, we can just sum up the durations
    if calculate_intersection(gt_bout, pr_bout) == 0:
        return gt_bout[1] + pr_bout[1]
    # They do intersect
    else:
        min_start_time = np.min([gt_bout[0], pr_bout[0]])
        gt_bout_end = gt_bout[0]+gt_bout[1]
        pr_bout_end = pr_bout[0]+pr_bout[1]
        max_end_time = np.max([gt_bout_end, pr_bout_end])
        return max_end_time - min_start_time

```

```

all_intersections = []
all_unions = []
all_iious = []

# Loop over the animals again
num_test_animals = len(np.unique(test_animals))
for animal_idx in np.arange(num_test_animals):
    # For each animal, we want a matrix of intersections, unions, and ious
    num_gt_bouts = len(all_gt_bouts[animal_idx][0])
    num_pr_bouts = len(all_pr_bouts[animal_idx][0])
    animal_intersection_mat = np.zeros([num_gt_bouts, num_pr_bouts])
    animal_union_mat = np.zeros([num_gt_bouts, num_pr_bouts])
    # Do a second loop for each GT bout
    for gt_idx in np.arange(num_gt_bouts):
        gt_bout = [all_gt_bouts[animal_idx][0][gt_idx], all_gt_bouts[animal_idx][1][gt_
        idx]]
        # Final loop for each proposed bout
        for pr_idx in np.arange(num_pr_bouts):
            pr_bout = [all_pr_bouts[animal_idx][0][pr_idx], all_pr_bouts[animal_
            idx][1][pr_idx]]
            # Calculate the intersections, unions, and IoUs
            animal_intersection_mat[gt_idx, pr_idx] = calculate_intersection(gt_bout, pr_
            bout)
            animal_union_mat[gt_idx, pr_idx] = calculate_union(gt_bout, pr_bout)
        # The IoU matrix will just be i / u
        animal_iou_mat = animal_intersection_mat/animal_union_mat
        # Add the animal data to the resulting lists
        all_intersections.append(animal_intersection_mat)
        all_unions.append(animal_union_mat)
        all_iious.append(animal_iou_mat)

```

Now that we have IoUs, we can try and apply thresholds for calculating things like precision/recall. Since it's hard to define "True Negatives" with the IoU technique, we can skip that for now.

```

# Define detection metrics, given a IoU threshold
def calc_temporal_iou_metrics(iou_data, threshold):
    tp_counts = 0
    fn_counts = 0
    fp_counts = 0
    for cur_iou_mat in iou_data:
        tp_counts += np.sum(np.any(cur_iou_mat>threshold, axis=1))
        fn_counts += np.sum(np.all(cur_iou_mat<threshold, axis=1))
        fp_counts += np.sum(np.all(cur_iou_mat<threshold, axis=0))
    precision = tp_counts/(tp_counts + fp_counts)
    recall = tp_counts/(tp_counts + fn_counts)
    f1 = 2*(precision * recall)/(precision + recall)
    return precision, recall, f1

```

```

all_iou_results = []

ious_to_evaluate = all_iious

for cur_iou in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]:

```

(continues on next page)

(continued from previous page)

```

performance = calc_temporal_iou_metrics(ious_to_evaluate, cur_iou)
all_iou_results.append(pd.DataFrame({'iou_threshold':[cur_iou], 'precision':
↪ [performance[0]], 'recall':[performance[1]], 'f1':[performance[2]]}))

iou_df = pd.concat(all_iou_results)
iou_df = pd.melt(iou_df, id_vars = ['iou_threshold'], value_vars = ['precision','recall',
↪ 'f1'])

iou_plot = p9.ggplot(iou_df) + \
    p9.geom_line(p9.aes(x='iou_threshold', y='value', color='factor(variable)')) + \
    p9.theme_bw()

iou_plot.draw().show()

```

Example Output:

Discussion 3

Pros and Cons to bout based analysis

False positives are overestimated, because there may be multiple predicted bouts within 1 real bout.

Are there potential methods to try and fix this issue?

Possible techniques include: 1. Filtering out short/spurious predictions 2. Stitching together multiple predictions close in time 3. Adjusting what we're calling TP/FN/FP to be more lenient on the exact number of bouts

Step 11

Post-processing techniques

```

def merge_behavior_gaps(bout_starts, bout_durations, bout_states, max_gap_size, state_to_
↪ merge = False):
    gaps_to_remove = np.logical_and(bout_states==state_to_merge, bout_durations<max_gap_
↪ size)
    new_durations = np.copy(bout_durations)
    new_starts = np.copy(bout_starts)
    new_states = np.copy(bout_states)
    if np.any(gaps_to_remove):
        # Go through backwards removing gaps
        for cur_gap in np.where(gaps_to_remove)[0][::-1]:
            # Nothing earlier or later to join together, ignore
            if cur_gap == 0 or cur_gap == len(new_durations)-1:
                pass
            else:
                cur_duration = np.sum(new_durations[cur_gap-1:cur_gap+2])
                new_durations[cur_gap-1] = cur_duration
                new_durations = np.delete(new_durations, [cur_gap, cur_gap+1])
                new_starts = np.delete(new_starts, [cur_gap, cur_gap+1])
                new_states = np.delete(new_states, [cur_gap, cur_gap+1])
    return new_starts, new_durations, new_states

```

Experiment 5

Test performance across multiple filtering parameters Run the next 2 cells and change the 2 variables at the top to be more/less stringent on bouts

```
# Filter out short bouts
min_bout_duration = 9
# Remove short breaks in bouts
min_gap_duration = 5

filtered_pr_bouts = []
for animal_id in np.unique(test_animals):
    animal_indices = test_animals==animal_id
    raw_bout_predictions = rle(test_predictions[animal_indices])
    # Remove short breaks in bouts
    filtered_bout_predictions = merge_behavior_gaps(raw_bout_predictions[0], raw_bout_
    predictions[1], raw_bout_predictions[2], min_bout_duration, state_to_merge=False)
    # Filter out short bouts
    filtered_bout_predictions = merge_behavior_gaps(filtered_bout_predictions[0],
    filtered_bout_predictions[1], filtered_bout_predictions[2], min_bout_duration, state_
    to_merge=True)
    behavior_bouts = filtered_bout_predictions[2]==1
    if np.any(behavior_bouts):
        filtered_bout_predictions = (filtered_bout_predictions[0][behavior_bouts],
        filtered_bout_predictions[1][behavior_bouts])
        filtered_pr_bouts.append(filtered_bout_predictions)
    else:
        filtered_pr_bouts.append(([], []))

filtered_intersections = []
filtered_unions = []
filtered_iou = []

# Loop over the animals again
num_test_animals = len(np.unique(test_animals))
for animal_idx in np.arange(num_test_animals):
    # For each animal, we want a matrix of intersections, unions, and iou
    num_gt_bouts = len(all_gt_bouts[animal_idx][0])
    num_pr_bouts = len(filtered_pr_bouts[animal_idx][0])
    animal_intersection_mat = np.zeros([num_gt_bouts, num_pr_bouts])
    animal_union_mat = np.zeros([num_gt_bouts, num_pr_bouts])
    # Do a second loop for each GT bout
    for gt_idx in np.arange(num_gt_bouts):
        gt_bout = [all_gt_bouts[animal_idx][0][gt_idx], all_gt_bouts[animal_idx][1][gt_
        idx]]
        # Final loop for each proposed bout
        for pr_idx in np.arange(num_pr_bouts):
            pr_bout = [filtered_pr_bouts[animal_idx][0][pr_idx], filtered_pr_
            bouts[animal_idx][1][pr_idx]]
            # Calculate the intersections, unions, and IoUs
            animal_intersection_mat[gt_idx, pr_idx] = calculate_intersection(gt_bout, pr_
            bout)
            animal_union_mat[gt_idx, pr_idx] = calculate_union(gt_bout, pr_bout)
```

(continues on next page)

(continued from previous page)

```
# The IoU matrix will just be i / u
animal_iou_mat = animal_intersection_mat/animal_union_mat
# Add the animal data to the resulting lists
filtered_intersections.append(animal_intersection_mat)
filtered_unions.append(animal_union_mat)
filtered_iou.append(animal_iou_mat)
```

Plot the performance with post-processing!

```
all_iou_results = []

for cur_iou in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]:
    performance = calc_temporal_iou_metrics(filtered_iou, cur_iou)
    all_iou_results.append(pd.DataFrame({'iou_threshold':[cur_iou], 'precision':
    ↳[performance[0]], 'recall':[performance[1]], 'f1':[performance[2]]}))

iou_df_filtered = pd.concat(all_iou_results)
iou_df_filtered = pd.melt(iou_df_filtered, id_vars = ['iou_threshold'], value_vars = [
    ↳'precision','recall','f1'])

iou_df_filtered['filtered'] = True
iou_df['filtered'] = False
iou_df_filtered = pd.concat([iou_df_filtered, iou_df])

iou_plot = p9.ggplot(iou_df_filtered) + \
    p9.geom_line(p9.aes(x='iou_threshold', y='value', color='filtered')) + \
    p9.facet_wrap('variable') + \
    p9.theme_bw()

iou_plot.draw().show()
```

Example Output:

```
bout_data = []
for animal_idx, animal_id in enumerate(np.unique(test_animals)):
    gt_bout_data = all_gt_bouts[animal_idx]
    if len(gt_bout_data[0])>0:
        gt_bout_df = pd.DataFrame({'state':'GT', 'animal':animal_id, 'animal_idx':animal_
    ↳idx, 'start_time':gt_bout_data[0], 'end_time':gt_bout_data[0]+gt_bout_data[1]})
        bout_data.append(gt_bout_df)
    pr_bout_data = filtered_pr_bouts[animal_idx]
    if len(pr_bout_data[0])>0:
        pr_bout_df = pd.DataFrame({'state':'PR', 'animal':animal_id, 'animal_idx':animal_
    ↳idx, 'start_time':pr_bout_data[0], 'end_time':pr_bout_data[0]+pr_bout_data[1]})
        bout_data.append(pr_bout_df)

bout_data = pd.concat(bout_data)
bout_plot = p9.ggplot(bout_data) + \
    p9.geom_rect(p9.aes(xmin='start_time', xmax='end_time', ymin='animal_idx-0.25', ymax=
    ↳'animal_idx+0.25', fill='factor(state)'), alpha=0.5) + \
```

(continues on next page)

(continued from previous page)

```
p9.labs(title='Bout annotations and predictions', x='Time, frame', y='Animal index',  
↪ fill='State') + \  
  p9.theme_bw()  
  
bout_plot.draw().show()
```

Example Output: