|  |
| --- |
| Allen Institute |
| Mahdi Ramadan’s Summer Internship at the Allen |
| A Stream of Consciousness Perspective |

|  |
| --- |
| Mahdi Ramadan  6-24-2016 |

Table of Contents

[2 Introduction 2](#_Toc459641926)

[2.1 The Project 2](#_Toc459641927)

[2.2 Background 2](#_Toc459641928)

[2.3 Process Overview 2](#_Toc459641929)

[2.4 Annotation 2](#_Toc459641930)

[3 Initial Setup 4](#_Toc459641931)

[3.1 Python Code 4](#_Toc459641932)

[3.1.1 BehaviorAnnotation.py 4](#_Toc459641933)

[3.1.2 Excel\_processing.py 6](#_Toc459641934)

[3.1.3 Image\_processing.py 10](#_Toc459641935)

[3.1.4 Machine\_learning.py 12](#_Toc459641936)

[4 Machine Learning 12](#_Toc459641937)

[4.1 Models 12](#_Toc459641938)

[4.1.1 Error Measures: 12](#_Toc459641939)

[4.1.2 Control: 13](#_Toc459641940)

[4.1.3 Table Summary of Runs: 16](#_Toc459641941)

[4.2 Features 17](#_Toc459641942)

[4.2.1 Pixel information: 19](#_Toc459641943)

[4.2.2 Optical flow: 20](#_Toc459641944)

[4.2.3 Optical angle: 20](#_Toc459641945)

# Introduction

Hello reader,

My name is Mahdi Ramadan and I am currently a rising Senior at the University of Washington studying Neurobiology, minoring in Computational Neuroscience and Neural Engineering. This summer I interned at the Allen Institute for Brain Science under the guidance of my mentor Jerome Lecoq.

## The Project

My project focused on exploring the possibility of leveraging computer vision and machine learning techniques for automatic identification and classification of mouse behavior from video. In the bigger scheme of science at the Allen Institute, understanding behavioral constructs is a crucial portion of the Allen Observatory initiative, which aims to understand the neural correlates of visually induced decisions and behavior. The first step to understanding such a correlation is to have a clear annotation and labeling of behavior.

## Background

Each mouse experiment is set-up with a two-photon microscope to image neural fluorescence, an eye tracking camera, and a behavioral camera capturing the full body of the mouse. We will be studying and analyzing the behavioral camera. Each experiment/video is approximately one hour long.

## Process Overview

The first step in starting this project was determining the mouse behaviors our Optical Physiology team would like annotated. The first obvious split was movement versus non-movement, but more refined behaviors were also interesting. Some of these behaviors include fidgeting, which at first I termed as startle response. The change came about after talking to a team member who brought to my attention that extreme and exaggerated nature of the startle response, which up to that point had been annotated with much smaller twitched. For that reason, it was changed to fidget. Another interesting behavior observed was mouth chattering, wherein the mouse rapidly moves its mouth in a chewing pattern. See pg. 4 for a full behavior decision tree.

## Annotation

At first, I was unsure of how to annotate the videos. I thought that I could write the frame number on the videos using a python script, and then using an excel sheet record the behaviors displayed during a certain frame duration. As you might have guessed, and considering that each video is an hour long, the process took way to long. Each minute of video took around ten minutes to complete. To expedite the process, I developed an html based tool for annotation, which sends the annotation data to a personal web database structure. The information transaction is mediated with AJAX and MySQL structures. Please see my Git repository for detailed code and line by line commenting.

# Initial Setup

Once my annotation tool was in place, I tried annotating as many videos as I could. However, even with this optimized tool, each video took around 3-4 hours to annotate. For this reason, I was motivated to recruit a larger group of personal to help me annotate the videos. To account for the subjective nature of behavior identification and interpretation, I wrote up a Standard Operating Procedure document explaining and detailing the annotation protocol expected. Each behavior has its criteria and specific examples to go with it.

Why collect so much annotation data? Considering the mouse behavior video database consisted of around 300 videos with more added everyday, it made sense that our training data should be a least 10-20 annotated videos, especially considering that this project might require the use of deep neural networks, which are known to need very large amounts of data.

## Python Code

All programming for this project was done in python. First I needed a way to organize the python code inherited from my team, as well the new code I will write throughout my internship. For this, I created a parent class called behehaviorAnnotation.py, which imports and initializes every other script pertaining to the project. In this manner, I can call on multiple classes from one code. Since no virtual environment is set-up and is purely structured as a project in the PyCharm IDE, any scripts to be used must be in the same folder as BehaviorAnnotation.py.

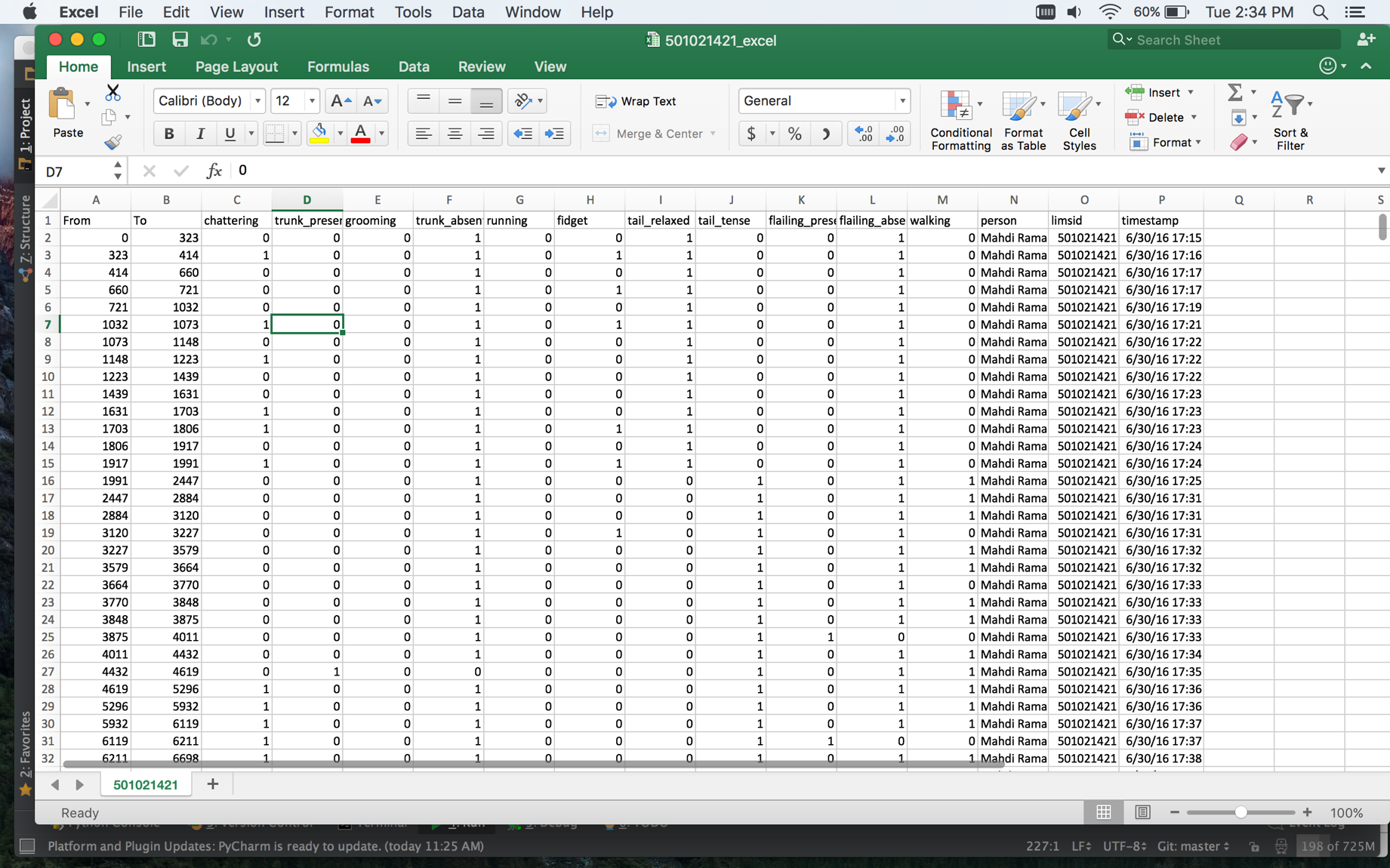
### BehaviorAnnotation.py

# behaviorAnnotation.py must be in same folder as raw\_behavior.py (to avoid adding files to path issues)  
  
**from** raw\_behavior **import** RawBehavior **as** rb  
**from** stimulus\_behavior **import** StimulusBehavior **as** sb  
**from** synced\_videos **import** SyncedVideos **as** sv  
**from** excel\_processing **import** ExcelProcessing **as** ep  
  
**from** wheel\_data **import** WheelData **as** wd  
# from machine\_learning import MachineLearning as ml  
**from** Visualize\_optical **import** VisualizeOptical **as** vo  
# from clustering\_visualization import ClusteringVisualization as cv  
**from** sync\_meta **import** SyncMeta **as** sm  
**from** lims\_database **import** LimsDatabase **as** ld  
**from** sync\_meta **import** SyncMeta **as** sm  
**import** matplotlib.pyplot **as** plt  
**import** numpy **as** np  
**import** math  
**import** os  
**import** cv2  
**import** sys  
**import** pandas  
**from** numpy.fft **import** fft, ifft, fft2, ifft2, fftshift  
  
**class DataAnalysis**:  
 **def** \_\_init\_\_(self,exp\_folder, lims\_ID):  
 # set up objects with parameters to be used  
 # If class calls on another class that requires a lims\_ID,  
 # class must have a lims\_ID input (add input below)  
 self.rb = rb(exp\_folder, lims\_ID)  
 self.sb = sb(exp\_folder, lims\_ID)  
 self.sv = sv(exp\_folder, lims\_ID)  
 self.ep = ep(exp\_folder, lims\_ID)  
 # self.wd = wd(exp\_folder, lims\_ID)  
 # self.ml = ml(exp\_folder, lims\_ID)  
 # self.vo = vo(exp\_folder, lims\_ID)  
 self.sm = sm(exp\_folder)  
 # self.cv = cv(exp\_folder, lims\_ID)  
  
 # self.ld = ld(lims\_ID)  
 # self.sm = sm(exp\_folder, lims\_ID)  
  
# Actual running script below  
  
# videos on this laptop stored in "/Users/mahdiramadan/Documents/Allen\_Institute/code\_repository/Videos"  
# example LIMS ID is 501021421  
  
# input LIMS ID or directory to files of interest!  
# RawBehavior, Stimulusbehavior, SyncedVideos, ExcelProcessing take in video directory  
# LimsDatabase takes in LIMS ID  
  
# initializes all DataAnalysis objects, takes video dire ctory and lims ID  
DataAnalysis = DataAnalysis("/Users/mahdiramadan/Documents/Allen\_Institute/code\_repository/Data/501560436", "501560436")  
  
# data labels for annotations are: "From", "To", "chattering", "trunk\_present", "grooming", "trunk\_absent", "running"  
# "fidget", "tail\_relaxed", "tail\_tense", "flailing\_present", "flailing\_absent", "walking", "person", "limsid", "timestamp"  
  
data = DataAnalysis.ep.get\_per\_frame\_data()  
  
**print**(data)

* BehaviorAnnotation.py is the master script from which other classes can be called from. The script first imports the classes of interest such as StimulusBehavior.py. The classes must be in the same folder or the environment/path of the class must be provided.
* Each class of interest is initialized with a folder path containing files of interest, such as the excel file containing annotation labels or pkl file containing wheel data. The class is also given the LIMS ID of interest in order to distinguish from other experiments.
* Once each class is initialized, we can call on our script to run a particular class and its associated method, or multiple classes and their associated methods. In this case for example, we are running the excel processing class (named ep) and specifically calling on the get\_per\_frame\_data method.

### Excel\_processing.py

From our annotation tool, we can extract an excel file from the MySQL database containing our annotation labels. The excel file is structured as shown below:



Each row represents one entry submission from the annotator, with each behavior label given a 0 or 1 (not present vs. present), and a range of frames for the entry as described by the “From” and “To” columns. The first task after retrieving the excel document is to quality check the data entries. In excel\_processing.py, there are two useful methods for making sure the entry submissions are make sense. The first is the frames\_continuous method, which looks for continuity between entries. It makes sure the “To” frame of a previous entry is equal to the “From” frame of the next entry.

**def frames\_continuous**(self):  
 # this method checks to see if the labeled frames are continuous. Make sure the labeled frame data is continuous  
 # for the rest of the code to work!  
 # gets the to and from frames  
 To\_frames = self.get\_to()  
 From\_frames = self.get\_from()  
  
 # for the each iteration, check whether the "to" frame is equal to the "from" frame in the next row  
 # if not continuous, returns which rows are discontinuous  
 **for** k **in** range(len(From\_frames)-1):  
 **if** To\_frames[k] != From\_frames[k+1]:  
 **return** "Frames are not continuous between row number %r and %r of the data" % (k+2, k+3)  
 **else**:  
 **continue  
 return** "Frames are continuous!"

If the entries are not continuous, it will print out for you the two rows of entries that do not match. The row numbers will correspond to the row numbers in the excel file for ease of locating the mismatched entries. Most errors arise from the annotator forgetting to press the “From” button, and occasionally forgetting to press the “submit” button. In all cases, the person checking the data should use the tool to re-watch the frames of interest and fix or fill in information.

Excel\_processing.py (EP) also has an is\_from\_smaller\_than\_to method, which makes sure the “To” frame of an entry is bigger than the “From” frame of the same entry. This is important for detecting errors arising from forgetting to press the “To” button. This method will also let you know which rows need to be looked at.

Once the data is checked for quality, the excel data needs to be formatted into an organized table that python scripts can use. The get\_per\_frame\_data method creates a table with a column for each column label in the excel file. The column labels are stored and called from a python list:

**def get\_labels**(self):  
 # column labels of interest for data (ignoring name, lims id, date)  
 # make sure to update if columns change  
 labels = ["chattering", "trunk\_present", "grooming", "trunk\_absent", "running",  
 "fidget", "tail\_relaxed", "tail\_tense", "flailing\_present", "flailing\_absent", "walking"]  
 **return** labels

For each excel entry, the method converts the the frame-range based labels to a frame by frame label. The output is a table with a column for each data label, and a row for each frame of the video that was annotated with corresponding 0s and 1s to denote the presence or absence of that label.

**def get\_per\_frame\_data**(self):  
 # This method takes in the annotated excel data with frame ranges, and returns a data matrix of  
 # each frame number annotated, along with the annotation scheme of each label (0 vs. 1) at that frame  
  
 # Important! I just found out that my mac and my work desktop gave me two different fps  
 # rates for the videos (29.411 vs. 30.0). For this reason, I had to apply a rate transform  
 # for the annotated labels in th excels files (From and To columns) of  
 # form =INT(previous To or From Cell/29.411765\*30.0)  
  
 # initiates data lists  
 frame\_start = []  
 frame\_end = []  
 labels = self.get\_labels()  
 # initiates the labels x number of frames data list, the minus 3 is to ignore the columns of  
 # name, mousid and date  
 frame\_data = [[] **for** \_ **in** range(len(self.get\_labels()) + 1)]  
 # first column set to frame numbers between first and last frame  
 frame\_data[0].extend(range(self.get\_first\_frame(), self.get\_last\_frame()+1))  
  
 # initiates all column labels to either 0 or 1  
 **for** k **in** range(len(self.get\_labels())):  
 frame\_data[k + 1].append(self.get\_true\_false(labels[k], 0))  
  
 # gets the frame start and end of each row in the excel file  
 **for** p **in** range(len(self.get\_column("From"))):  
 frame\_start.insert(0, self.get\_frame\_start(p))  
 frame\_end.insert(0, self.get\_frame\_end(p))  
  
 # for each frame, puts a 0 or 1 for each column label  
 # if you have frames 0 to 10 == 1, 10 to 20 == 0, frame 10 == 1 due to how code is set-up  
 **for** k **in** range(len(self.get\_labels())):  
 frame\_data[k + 1].extend([self.get\_true\_false(self.get\_labels()[k], p)] \* (frame\_end[0] - frame\_start[0]))  
  
 **return** frame\_data

Along with the ability to extract frame label data, EP has a method that calculates the normalized frequency of behavior during specific visual stimuli, and subsequently creates a visualization illustrating this relationship.

The main method responsible for this functionality is get\_frequency\_plot, which counts the number of frames with a certain behavior present over a time interval (in this case set to 5 seconds considering an fps of 30).

**def get\_frequency\_plot**(self, label):  
 # returns a plot of the change in frequency of a label per SECOND  
 data = self.get\_per\_frame\_data()  
 fps = 30.0  
 # get cumsum of column data  
 label\_data = np.cumsum(data[self.get\_labels().index(label) + 1])  
 # initiate counters and lists  
 frequency\_data = []  
 time = []  
 count = 0  
 n = 0  
 # determines over how many frames we calculate annotation frequency  
 # 147 frames is approximately 5 seconds  
 interval = 150  
 interval\_seconds = round(interval / fps)  
  
 # iterate over each frame  
 **for** k **in** range(len(label\_data)):  
 count += 1  
 # if count mod interval size = equal, then calculate the difference of the cumsum associated  
 # with this frame minus the cumsum at the frame one interval before  
 **if** count % interval == 0:  
 frequency\_data.append(label\_data[k] - label\_data[k - (interval - 1)])  
 n += 1  
 # round time in seconds to nearest integer  
 second = round(n \* interval / fps)  
 time.append(second)  
 **else**:  
 **continue**

To sync our 5 second bin count of behavior frequency with the visual stimulus being presented, we must access the NWB file associated with the experiment we are interested in. In the NWB data structure, we will find the timing of visual stimuli types during the experiment. The NWB is opened with the open\_nwb method.

**def open\_nwb**(self):  
 # opens nwb file  
 **for** file **in** os.listdir(self.directory):  
 **if** file.endswith("nwb"):  
 # make sure file is in there!  
 nwb\_path = os.path.join(self.directory, file)  
 # input file path, r is for read only  
 nwb\_file = h5py.File(nwb\_path, "r")  
 **if not** file:  
 **print** ("nwb file not found.")  
  
 **return** nwb\_file

Once the NWB file is accessible, the unique frames of each visual stimulus presented during the experiment are visualized by a colored rectangle starting and ending at times corresponding to the frame numbers (fps of 30.0). Each rectangle color is specific to a stimulus type. Right below is a bar plot of the frequency of behavior (binned every five seconds).

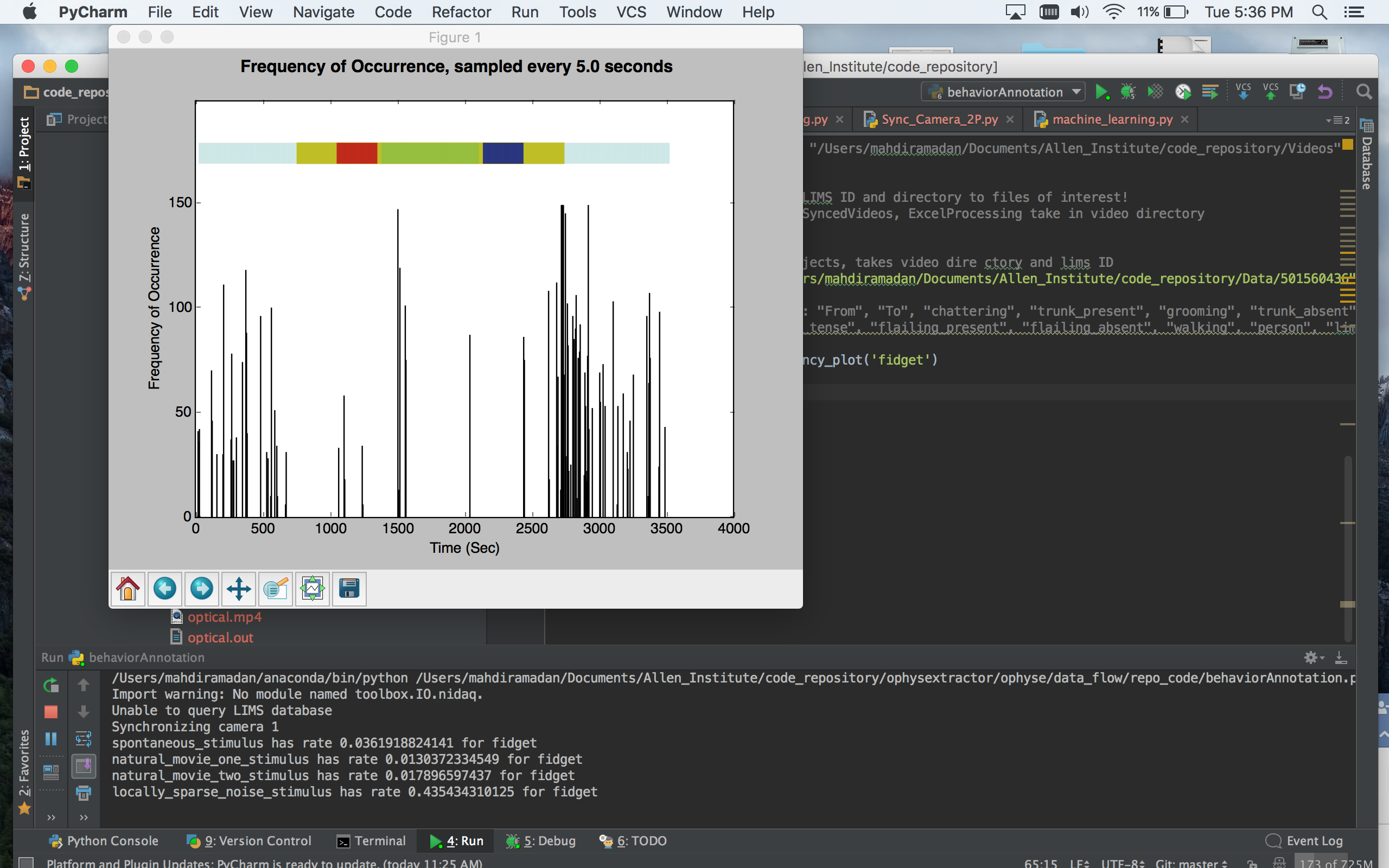
The frame numbers however are synced relative to the 2p imaging camera, which is not in sync with the behavior video camera we are interested in. To account for this situation, the code calls on another class, Sync\_Camera\_Stimulus, which calculates the offset between the two cameras.

**def create\_stimulus\_definition**(self, m, ax, fps, freq, label):  
 # create CAM stimulus definition visual  
 # Must call on get\_frequency\_plot first to get correct inputs, this is  
 # NOT a stand alone method  
  
 iframe = self.sc2p.Sync\_Camera\_Stimulus()[0]

The offset is incorporated to ensure correct behavior video to visual stimulus timing. Using this information, the frequency of a behavior per visual stimulus can be calculated, and is normalized by the proportion of time that the visual stimulus was presented during the experiment. The code will print out your behavior rate for your specific behavior.

sum = np.sum(cqg)  
**print** (str(stim) + ' has rate '+ str(float(sum)/fn) + ' for ' + str(label))

The final visualization looks like this:



To determine the color legend, just match the index of the visual stimuli in the visual stimulus list

stimuli = ['spontaneous\_stimulus','drifting\_gratings\_stimulus','natural\_movie\_one\_stimulus', 'natural\_movie\_two\_stimulus',  
 'natural\_movie\_three\_stimulus', 'static\_gratings\_stimulus',  
 'locally\_sparse\_noise\_stimulus']

with the same index number in the colors array:

# diff colors for each stim (black should be for stim that needs an edge color as explained below  
colors = ('y','k','r','b','g','m','c')

### Image\_processing.py

Image\_processing.py will calculate and store the feature vectors that will be used for training and prediction in our machine learning model. Due to the large amount of data involved, the features are stored in a .h5 file.

For every frame of video, the script calculates a sliding window histogram vector for the raw frame pixels, the optical flow map and the optical angle map.

Optical flow and angle are calculated using a Farneback based algorithm, which utilizes efficient polynomial expansion transforms to estimate pixel neighborhood differences between two frames. To read more, see original paper: <http://www.diva-portal.org/smash/get/diva2:273847/FULLTEXT01.pdf>.

Once calculated, the optical angles are scaled to a 0-180 range, which was later used to color code movement direction based on a HSV color scale.

**def optical\_flow**(prvs, next):  
  
 # calculate optical flow and angle of out two frame  
 flow = cv2.calcOpticalFlowFarneback(prvs, next, 0.5, 3, 15, 3, 5, 1.2, 0)  
 mag, ang= cv2.cartToPolar(flow[..., 0], flow[..., 1])  
  
 # get histograms of optical flow and angles (these are our features)  
 mag = process\_input(mag)  
 ang = process\_input((ang\*180/np.pi/2))  
  
 **return** {'mag': mag, 'ang': ang}

Each window is size 30 x 30, with a step of 30 pixels, for a matrix size of 540 x 240 (the frame size). For each frame size, 144 sliding windows will be computed.

**def process\_input**(input):  
  
 frame\_data = []  
 # for each defined window over the data, bin values and return a  
 # properly scaled list of expectation values  
 **for** (x, y, window) **in** sliding\_window(input, 30, (30, 30)):  
 hist, bin = np.histogram(window, 10)  
 center = (bin[:-1] + bin[1:]) / 2  
 hist\_x = np.multiply(center, hist)  
 hist\_x = preprocessing.MinMaxScaler((-1, 1)).fit(hist\_x).transform(hist\_x)  
 frame\_data = np.concatenate((frame\_data, hist\_x))  
 **return** frame\_data  
  
**def sliding\_window**(image, stepSize, windowSize):  
 # slide a window across the image  
 **for** y **in** xrange(0, image.shape[0], stepSize):  
 **for** x **in** xrange(0, image.shape[1], stepSize):  
 # yield the current window  
 **yield** (x, y, image[y:y + windowSize[1], x:x + windowSize[0]])  
  
**def get\_file\_string**(exp\_folder,lims\_ID):

For each 30x30 window, values are categorized into 10 discrete bins, and then the expectation value of each window is calculated by multiplying the bin value by the count value (number of occurrences in the bin). The resultant 10x1 vector is our feature descriptor for that sliding window. For each frame matrix therefore, the feature vector will contain 10 x 144 expectation values.

Each vector calculated is normalized to a -1 to +1 range, considering that many models such as the Support Vector Machine need data inputs of the same scale, and more specifically lie in a 0 to 1 or -1 to 1 range.

The final feature vector contains the wheel velocity corresponding to the frame (one value), 1440 values for the raw pixels, 1440 values for optical flow, and 1440 values for optical angle. The final feature size is 4321 for each video frame.

Each new feature vector is stored in an h5 file, as described by this code:

# create hdf file  
hf = h5py.File('data\_' + str(lims\_ID) + '.h5', 'w')  
g = hf.create\_group('feature space')  
vector = np.zeros((limit, 4321))  
table = g.create\_dataset('features', data = vector, shape =(limit, 4321))

Each one-hour video will create a ~4 GB H5 file.

### Machine\_learning.py

# Machine Learning

## Models

Throughout my internship there has always been an ongoing discussion about which machine learning model to use for the project. One of the top candidates was a deep neural network, the other a support vector machine. Due to the time constraints and computational power of my internship, and in conjunction with another machine learning expert at the Allen Institute, we decided to first try the SVM and see how it does. If anything, we get a baseline to compare any further models to. It has been widely known in the artificial intelligence realm that simpler models such as nearest neighbors or decision trees perform poorly in computer vision problems.

After discussion with various machine learning experts, it seemed like the best route for starting on the machine learning project was to try a simpler machine learning model such as a Support Vector Machine (SVM) before moving onto a model as complex and intensive as a deep neural network. SVM’s take into consideration various parameters, but two of the most sensitive are the C and Gamma constants. The C value determines how sensitive the SVM should be to edge points of clusters, while Gamma determines the step size the SVM should take as it iterates towards a possible solution.

I used a module classed GRIDSEARCHCV to iterate through multiple C and Gamma values of different orders of magnitude to get an idea of the range I should be working with. The search consistently gave an optimal C: 10 and Gamma: 0.001 no matter the training data size or behavior of interest.

### Error Measures:

Recall:

Proportion of correct positive classifications from cases that are actually positive

Precision:

Proportion of correct positive classification from cases that are predicted to be positiv

### Control:

#### Human Baseline:

For a baseline comparison, two humans were asked to annotate the same video (LIMS ID: 502665019 and 509904120). The precision and recall measures were then calculated for a variety of behavior categories. Below is a summary:

##### LIMS ID: 502665019

* Fidget (label 1 == present):

precision recall f1-score support

0 0.90 0.80 0.85 27510

1 0.68 0.83 0.75 14490

avg / total 0.82 0.81 0.81 42000

* Movement (label 1 == present):

precision recall f1-score support

0 0.66 0.98 0.79 4125

1 1.00 0.95 0.97 37875

avg / total 0.96 0.95 0.95 42000

* Fidget (label 0 == present) vs. movement (label 1 == present) vs everything else:

precision recall f1-score support

0 0.98 0.66 0.79 6108

1 0.80 0.90 0.85 24473

2 0.60 0.55 0.57 11419

avg / total 0.77 0.77 0.76 42000

##### LIMS ID: 509904120

* Fidget:

precision recall f1-score support

0 0.64 0.96 0.77 25286

1 0.49 0.06 0.11 14714

avg / total 0.58 0.63 0.53 40000

* Movement:

precision recall f1-score support

0 1.00 1.00 1.00 26109

1 1.00 1.00 1.00 13891

avg / total 1.00 1.00 1.00 40000

* Fidget vs movement vs everything else:

precision recall f1-score support

0 0.49 0.06 0.11 14714

1 1.00 0.98 0.99 13891

2 0.44 0.94 0.60 11395

avg / total 0.65 0.63 0.56 40000

#### Chance baseline:

To get an idea of what the random baseline looks like, I compared the annotations of different videos.

The results are as shown below:

##### LIMS ID 500860585 and 497060401 (same mouse different stim):

* Fidget vs movement vs neither:

precision recall f1-score support

0 0.09 0.13 0.11 4994

1 0.03 0.06 0.04 9696

2 0.79 0.67 0.73 65310

avg / total 0.65 0.57 0.61 80000

##### LIMS ID 501773889 and 497256116 (different mouse different stim):

* Fidget vs. movement vs. no movement:

0 0.02 0.39 0.03 2229

1 0.21 0.27 0.24 16736

2 0.68 0.12 0.20 61035

avg / total 0.56 0.16 0.20 80000

##### LIMS ID 501021421 and 50292794 (different mouse same stim):

* Fidget vs movement vs neither:

precision recall f1-score support

0 0.16 0.15 0.15 11041

1 0.29 0.24 0.26 13143

2 0.70 0.74 0.72 55816

avg / total 0.56 0.57 0.56 80000

### Table Summary of Runs:

Below is a table summary of all the experiments run with the support vector machine:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Run # | Behavior | Model | Training data size | Precision | Recall | Notes: | Video |
| 1 | Fidget vs. rest | SVM | 15000 | 0 | 0 | Initial try. Only optical flow and angle | Train and test:  501560436 |
| 2 | Fidget vs. movement | SVM | 25000 | 0.36 | 0.4 | More balanced counts of fidget vs rest. Only optical flow and angle | Train and Test:  501560436 |
| 3 | Fidget vs. movement | SVM | 50000 | 0.97 | 0.97 | Only fidget and movement frames, added wheel data and frame HOG descriptor. Random sampling. | Train and Test:  501560436 |
| 4 | Fidget vs movement vs rest | SVM | 50000 | 0.94 | 0.94 | Fidget vs. movement vs. the rest of all other behaviors, updated wheel sync data method (was previously offset). Random sampling. | Train and test:  501560436 |
| 5 | Fidget vs movement vs rest | SVM | 60000 | 0.80 | 0.81 | Fidget vs. movement vs. the rest of all other behaviors,  Training data from three videos,  test set was from a completely unused fourth video | Train:  501560436, 501021421, 500860585,  Test:  501004031 |
| 6 | Chattering vs. not chattering | SVM | 50000 | 0.636 | 0.64 | Mouth chattering vs. all other behaviors, training and testing data from same video | Train and Test:  LIMS ID 501021421 |
| 7 | Fidget vs movement vs neither | SVM | 68000 | 0.65 | 0.63 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 8 | Movement vs no movement | SVM | 68000 | 0.72 | 0.70 | Video made for the first 30,000 frames | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 9 | Movement vs. no movement | SVM (only wheel data) | 60000 | 0.72 | 0.75 | Trained with only wheel data!!!! | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 10 | Movement vs. no movement | LDA  (Wheel and optical flow) | 60000 | 0.70 | 0.72 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 11 | Fidget and movement vs. neither | SVM | 60000 | 0.70 | 0.71 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  ‘509904120'' |
| 12 | Movement vs. no movement (fidget included) | SVM | 60000 | 0.74 | 0.74 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 14 | Movement vs no movement | SVM (optical flow and wheel) | 60000 | 0.72 | 0.73 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 15 | Fidget vs movement | SVM  (wheel  only) | 60000 | 0.89 | 0.94 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 16 | Fidget vs. movement | SVM (  Wheel and optical flow) | 60000 |  |  |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '509904120' |
| 17 | Fidget and movement vs. neither | SVM | 60000 | 0.93 | 0.92 |  | Train:  '501560436', '501021421', '500860585', '501004031'  Test:  '500860585' |
| 18 | Movement vs. neither and fidget,  Then neither vs fidget | SVM | 40000 | 0.91 | 0.90 |  | Train:  '501560436'  '509904120'  ‘502741583’  Test:  '500860585'  frame 10000 to 20000 |
| 19 | Movement and fidget vs neither,  Then movement vs. fidget | SVM | 80000 | 0.80 | 0.83 |  | Train:  '501560436'  '509904120'  ‘502741583’  Test:  '500860585'  frame 60000 to 80000 |

70 – 80, 0436, 1583, 4120 test on 0585

10-20 0436, 0401, 4120 test on 1583

50-70 0436, 0401 test on 1583

20-60 4120 test 1583

HDDA:

feature processing finished

Models C th BIC

M1 3 0.01 -410181126.394

M2 3 0.01 -410181104.39

M3 3 0.01 -410181104.39

M4 3 0.01 -410181082.386

M5 3 0.01 -410630173.335

M6 3 0.01 -410630151.33

M7 3 0.01 -410630151.33

M8 3 0.01 -410630129.326

Best model is M4

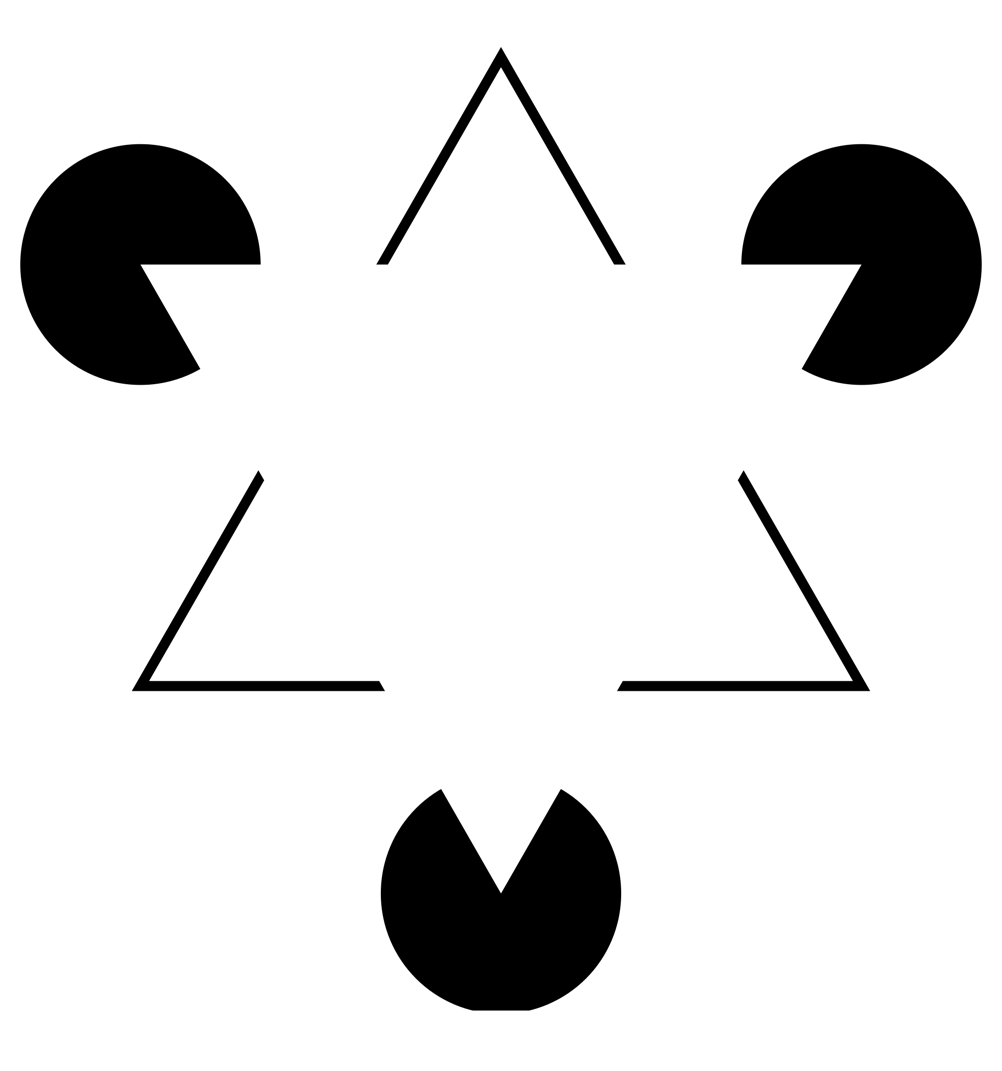
## Features

One of the most important aspects of machine learning is feature extraction. The desired feature vector not only will give you the clustering separation (in n-dimensions) you need to consistently classify your data, but also will be concise and devoid of any redundancy. As with most machine learning problems, we are constrained by how much memory and computing power we have access to.

If we take just the raw pixels, optical flow and optical angle values, we would end with a vector that is 3 x (540\*240) long, with turns out to be 388,800 values per frame. Considering that each video contains around 100,000 frames, the training or test data for a whole video would be close to 360 Gigabytes. Not only is 360 GB a considerable amount of storage on a hard disk, it would be nearly impossible to get access to a computer with an equivalent amount of RAM memory.

I took inspiration from our visual cortex to build the model currently described above in my code. As scientists began exploring the working of vision and our visual cortex, it became clear that the brain does not process the raw input our world provides to our eyes due to several observations. First, the information provided by the trillions of photons our eyes receive each second must be reduced to the temporal and spatial resolution of our optic nerve, which contains only around one million nerve fibers firing at a maximal rate of around 1000 hertz.

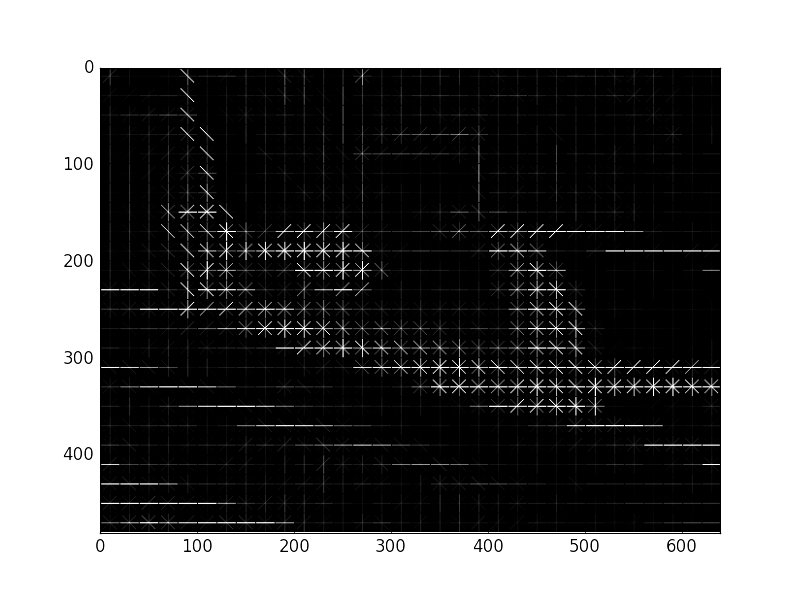
Due to the physical limitations of our physiology, and the need for near real-time and extremely quick decision making based on our visual input, the brain must consider only the most crucial aspects of our world. The seamless and complete quality of reality we experience is merely due to the fact that our brain fills in these gaps of information. That is why we perceive a visual existence where our blind spot should be for example. The brain does this guessing based on the statistics of the visual scene. Sometime this guessing is not an accurate representation of reality, such as the optical illustration below wherein your brain gives you the illusion of the presence of a triangle.



Similarly, my model does not take an input of the raw features, but rather considers a statistical representation of the visual scene as described by the expectation histogram calculated over each window. Not only does this reduce the data size needed by a factor of 90, a statistical representation has been shown to a more robust and reliable input to computer vision problems.

### Pixel information:

Pixel information is crucial to the computer vision model employed, as it describes important information on mouse form and structure, and filters out unnecessary pixel information. Below is a visualization of the histogram of oriented gradients employed in the model:



### Optical flow:

One of the defining features of behavior for a biological organism is movement. Thus, it comes to no surprise that a measurement of movement is crucial in identifying movement in behaviors such as walking.

### Optical angle:

Along with an absolute measure of movement, it is also very important to consider the direction of movement, where in different behaviors of similar movement magnitude might differ in their direction or spatial pattern. Below is a visualization of optical flow, where in the intensity of the pixels indicates movement magnitude, and coloring indicates the direction of movement based on the 0 -180 HSV color scale.

