KET-VR5 Image Data Wrangling

January 29, 2024

1 Ketamine VR5 - SINGLE LABELED DATA ONLY

Jonathan Ramos 1/26/2024 I'm glad these data came just as I finished the sleep dep set so some of code is still fresh in my brain. For these data, the format of the csvs is quite different (due to the difference in the way PIPSQUEAK vs POLYGON spit out csvs). Col names are different and some label names need to be changed; in particular, some stain type names are simply called "hand drawn" if the user added ROIs that were not detected by the polygon algorithm. This causes probems because all hand drawn ROIs of any stain type are all called "hand drawn." This has been an on going issue with polygon, but we have a work around.

In the filename col, all files follow a consistent naming scheme: - *_2.tif : PV - *_3.tif : cFos - *_4.tif : Npas4 - *_5.tif : WFA

Additionally, since there is no subject ID col, we can construct it from informatively named filenames instead. For this project, filenames follow the following format:

(rat number)_(brain region)_(bregma)_(n).tif

In this notebook I will wrangle all the data into one spot (data is distributed over ~600 small csvs), clean things up, normalize intensity and count mean cell ns.

1.1 Cleaning, Wrangling Data

1.1.1 Loading data, stitching sets together

[1]:
```
import numpy as np
import pandas as pd
import glob

# load cohort key (there's a few empty rows at the end)
df_key = pd.read_csv('KETAMINE_COHORT KEY.csv').dropna()

# load data; getting it all in one spot
df_full = pd.concat([pd.read_csv(f) for f in glob.glob('CSV_FILES/*/*.csv')])

# col names begin with a whitespace char; let's remove all ' ' chars from col␣
↪names
df_full.columns = [col.replace(' ', '') for col in df_full.columns]

# let's take a look
```

```
df_full
```

```
/var/folders/b2/3h2lpxx14kgb12pp_7pltxnc0000gn/T/ipykernel_52455/1229073174.py:2
: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of
pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better
interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
```

[1]:

	cell_number	roi_id	roi_source	roi_type	CoM_x	CoM_y	\
0	1	000-00000	Parvalbumin	OVAL	120.64	307.06	
1	2	000-00001	Parvalbumin	OVAL	403.99	379.52	
2	3	000-00002	Parvalbumin	OVAL	363.44	463.29	
3	4	000-00003	Parvalbumin	OVAL	68.43	322.91	
4	5	000-00004	Parvalbumin	OVAL	386.72	251.18	
..	…	…	…	…	…	…	
73	74	FFF-00073	hand-drawn	OVAL	491.39	60.52	
74	75	FFF-00074	hand-drawn	OVAL	497.35	15.91	
75	76	FFF-00075	hand-drawn	OVAL	437.04	376.98	
76	77	FFF-00076	hand-drawn	OVAL	472.11	432.2	
77	78	FFF-00077	hand-drawn	OVAL	352.86	214.76	

	pixel_area	background	mean_intensity	median_intensity	…	feret_angle	\
0	665.0	285.8023	636.1711	677.3578	…	0.0	
1	468.0	285.8023	412.1381	395.8835	…	90.0	
2	399.0	285.8023	369.5932	366.851	…	0.0	
3	550.0	285.8023	518.8725	540.6805	…	0.0	
4	524.0	285.8023	832.9809	920.6865	…	0.0	
..	…	…	…	…	…	…	
73	130.0	541.3023	435.1857	415.7151	…	0.0	
74	65.0	541.3023	413.2948	392.1778	…	0.0	
75	130.0	541.3023	493.2017	462.339	…	0.0	
76	104.0	541.3023	472.384	433.4397	…	0.0	
77	96.0	541.3023	422.3084	387.4282	…	0.0	

	feret_min	circularity	aspect_ratio	roundness	solidity	skewness	kurtosis	\
0	28.0	0.9387	1.1447	0.8205	0.9419	-0.4228	-0.6905	
1	24.0	0.8789	1.1555	0.7613	0.8797	0.4731	-0.5155	
2	20.0	0.8698	1.4266	0.6299	0.9027	0.3051	-0.1709	
3	22.0	0.8089	1.6476	0.535	0.8814	-0.4078	-0.4258	
4	26.0	0.9146	1.0654	0.8539	0.9066	-0.6879	-0.6308	
..	…	…	…	…	…	…	…	

```
73       28.0      0.1835       1.1447    0.1604    0.1841   1.3612    1.5127
74       28.0      0.0918       1.0617    0.0933    0.0921   3.3638   15.8851
75       28.0      0.1835       1.1447    0.1604    0.1841   1.4819    2.5275
76       28.0      0.1468       1.1447    0.1283    0.1473   1.6255    3.4102
77       28.0      0.1355       1.1447    0.1184    0.1360   1.8271    3.7074

                     filename                  analysis_date
0     PE-11-7_PFC_3.9_A_2.tif   Thu Jan 25 15:09:00 PST 2024
1     PE-11-7_PFC_3.9_A_2.tif   Thu Jan 25 15:09:00 PST 2024
2     PE-11-7_PFC_3.9_A_2.tif   Thu Jan 25 15:09:00 PST 2024
3     PE-11-7_PFC_3.9_A_2.tif   Thu Jan 25 15:09:00 PST 2024
4     PE-11-7_PFC_3.9_A_2.tif   Thu Jan 25 15:09:00 PST 2024
..                        …                         …
73   KET-10-4_PFC_3.5_C_4.tif   Mon Jan 22 16:14:32 PST 2024
74   KET-10-4_PFC_3.5_C_4.tif   Mon Jan 22 16:14:32 PST 2024
75   KET-10-4_PFC_3.5_C_4.tif   Mon Jan 22 16:14:32 PST 2024
76   KET-10-4_PFC_3.5_C_4.tif   Mon Jan 22 16:14:32 PST 2024
77   KET-10-4_PFC_3.5_C_4.tif   Mon Jan 22 16:14:32 PST 2024

[24613 rows x 30 columns]
```

1.1.2 Relabeling incorrect data

[2]:
```
# there were some issues with the naming/cohort key

# PE-12-7 was incorrectly labeled as PE-12-3; this was confirmed by JR and AG␣
 ↪by checking slide books/hard copies of behavior data
df_full['filename'] = df_full.filename.replace({'PE-12-3': 'PE-12-7'},␣
 ↪regex=True)

# similarly KET-8-2 was incorrectly labeled as KET-8-5; this was confirmed by␣
 ↪JR and AG  by checking slide books. this wouldn't really
# change anything since they received the same treatment but let's just change␣
 ↪it to the correct label anyway
df_full['filename'] = df_full.filename.replace({'KET-8-5': 'KET-8-2'},␣
 ↪regex=True)

# check result
assert df_full.filename.str.contains('PE-12-3').sum() == 0
assert df_full.filename.str.contains('PE-12-7').sum() != 0
assert df_full.filename.str.contains('KET-8-5').sum() == 0
assert df_full.filename.str.contains('KET-8-2').sum() != 0
```

1.1.3 Building the necessary cols

In particular we will need a rat_n (sid) col, stain_type col, and a treatment col. the filename col functions as the image name (iid) col.

We need the following cols - rat_n (sid) - treatment - filename (fid) - imagename (iid) - stain_type - CoM_x - CoM_y - mean-background

[3]:
```
# creating a new rat_n col
df_full['rat_n'] = df_full.filename.apply(lambda x: x.split('_')[0])\
    .replace({' ': ''}, regex=True) # for some reason, we have more leading␣
 ↪whitespace chars

# some checks. we want be sure that the structure of all our rat_n labels is␣
 ↪consistent
# in particular, we expect something of the form 'PE-12-7', that is we have␣
 ↪exactly
# two dashes '-' separating some letters, followed by two numbers
assert df_full.rat_n.apply(lambda x: len(x.split('-')) == 3).sum() ==␣
 ↪len(df_full)
assert df_full.rat_n.apply(lambda x: x.split('-')[0].isalpha()).sum() ==␣
 ↪len(df_full)
assert df_full.rat_n.apply(lambda x: x.split('-')[1].isnumeric()).sum() ==␣
 ↪len(df_full)
assert df_full.rat_n.apply(lambda x: x.split('-')[2].isnumeric()).sum() ==␣
 ↪len(df_full)

# building a cohort key dictionary from df_key
treatment = dict(zip(df_key.Subject, df_key.TX.replace({' ': '_'}, regex=True)))

# creating new treatment col by mapping from cohort key dict
df_full['treatment'] = df_full.rat_n.map(treatment)

# creating new stain_type col from filename
stains = {
    '.*_2.tif$' : 'PV',
    '.*_3.tif$' : 'cFos',
    '.*_4.tif$' : 'Npas4',
    '.*_5.tif$' : 'WFA'
}
df_full['stain_type'] = df_full.filename.replace(stains, regex=True)

# check that stain_type col contains the appropriate labels
assert set(df_full.stain_type.unique()) == set(stains.values())

# building image name (iid) from file name (fid) col
df_full['image_name'] = df_full.filename.replace({'_[0-9]\.tif': ''},␣
 ↪regex=True)

df_subset = df_full[['rat_n', 'treatment', 'stain_type', 'filename',␣
 ↪'image_name', 'CoM_x', 'CoM_y', 'mean-background']]
```

```
# let's take a look
df_subset
```

[3]:
```
       rat_n treatment stain_type                    filename  \
0    PE-11-7   VR5_KET         PV    PE-11-7_PFC_3.9_A_2.tif
1    PE-11-7   VR5_KET         PV    PE-11-7_PFC_3.9_A_2.tif
2    PE-11-7   VR5_KET         PV    PE-11-7_PFC_3.9_A_2.tif
3    PE-11-7   VR5_KET         PV    PE-11-7_PFC_3.9_A_2.tif
4    PE-11-7   VR5_KET         PV    PE-11-7_PFC_3.9_A_2.tif
..       …       …        …                          …
73  KET-10-4   VR5_SAL      Npas4   KET-10-4_PFC_3.5_C_4.tif
74  KET-10-4   VR5_SAL      Npas4   KET-10-4_PFC_3.5_C_4.tif
75  KET-10-4   VR5_SAL      Npas4   KET-10-4_PFC_3.5_C_4.tif
76  KET-10-4   VR5_SAL      Npas4   KET-10-4_PFC_3.5_C_4.tif
77  KET-10-4   VR5_SAL      Npas4   KET-10-4_PFC_3.5_C_4.tif

              image_name   CoM_x   CoM_y mean-background
0     PE-11-7_PFC_3.9_A  120.64  307.06         354.873
1     PE-11-7_PFC_3.9_A  403.99  379.52        127.1692
2     PE-11-7_PFC_3.9_A  363.44  463.29         86.6384
3     PE-11-7_PFC_3.9_A   68.43  322.91        233.9562
4     PE-11-7_PFC_3.9_A  386.72  251.18        547.2813
..                  …     …     …               …
73   KET-10-4_PFC_3.5_C  491.39   60.52       -106.9829
74   KET-10-4_PFC_3.5_C  497.35   15.91       -120.9047
75   KET-10-4_PFC_3.5_C  437.04  376.98        -49.6622
76   KET-10-4_PFC_3.5_C  472.11   432.2        -67.7978
77   KET-10-4_PFC_3.5_C  352.86  214.76       -119.9755

[24613 rows x 8 columns]
```

1.1.4 Dropping nans, duplicates

[4]:
```
# which cols have nans, how many?
print('Nan per col:')
print(df_subset.isna().sum())
# it looks like we have no nans! nothing to drop here.

# how many duplicated rows do we have?
print('\nTotal n of duplicated rows:')
print(df_subset.duplicated().sum())

# it looks like we have 15 duplicated rows. let's take a look
df_full[df_subset.duplicated()].head(15)

# I'm not concerned about dropping these duplicates, so let's just toss em
df_cleaned = df_subset[~df_subset.duplicated()]
```

```
assert df_cleaned.duplicated().sum() == 0
```

```
Nan per col:
rat_n              0
treatment          0
stain_type         0
filename           0
image_name         0
CoM_x              0
CoM_y              0
mean-background    0
dtype: int64

Total n of duplicated rows:
15
```

1.2 Normalizing Intensity

all parameterized functions will get set aside into module for future use and standardization.

[5]:
```
def normalize_intensity(df, norm_condition):
    '''
    computes the mean of rows of the norm_condition and divides mean-background␣
↪by this mean,
    normalizing all data to the mean of the norm_condition. sets normalized␣
↪value into new
    column called "norm mean-background" and returns new dataframe containing␣
↪normalized intensity.
    '''
    df_norm = df[df.treatment == norm_condition]
    norm_mean = df_norm['mean-background'].astype('f').mean()

    df_norm = df.copy(deep=True)
    df_norm['norm mean-background'] = df['mean-background'].astype('f') /␣
↪norm_mean

    # quickly check that the mean of the norm condition is set to about 1.00000
    # this is never exatly 1 due to small rounding errors from floating point␣
↪operations
    assert round(df_norm[df_norm.treatment == norm_condition]['norm␣
↪mean-background'].mean(), 5) == 1

    return df_norm

def prism_reorg(df, group):
    '''
```

```
    Takes just the norm_mean-background intensity col per rat, groups by␣
↪treatment
    and
    '''
    treatments = np.unique(df.treatment)
    reorg = []

    for t in treatments:
        df_treat = df[df.treatment == t]
        norm_int_ratn = []
        treatment_ratns = np.unique(df_treat.rat_n)

        for rat in treatment_ratns:
            norm_int = df_treat[df_treat.rat_n == rat]['norm mean-background']
            df_normint = pd.DataFrame({t: norm_int}).reset_index(drop=True)
            norm_int_ratn.append(df_normint)

        # concat "vertically"
        df_ratn_cols = pd.concat(norm_int_ratn, axis=0).reset_index(drop=True)

        # write csv to disk
        reorg.append(df_ratn_cols)

    # concat "horizontally"
    df_prism_reorg = pd.concat(reorg, axis=1)


    # write csv to disk
    df_prism_reorg.to_csv(f'{group}_{np.unique(df.stain_type).item()}_{t}_PRISM.
↪csv')

    return df_prism_reorg

def prism_reorg(df, group):
    '''
    Takes just the norm_mean-background intensity col per rat, groups by␣
↪treatment
    and
    '''
    treatments = np.unique(df.treatment)
    reorg = []

    for t in treatments:
        df_treat = df[df.treatment == t]
        norm_int_ratn = []
        treatment_ratns = np.unique(df_treat.rat_n)
```

```
        for rat in treatment_ratns:
            norm_int = df_treat[df_treat.rat_n == rat]['norm mean-background']
            df_normint = pd.DataFrame({t: norm_int}).reset_index(drop=True)
            norm_int_ratn.append(df_normint)

        # concat "vertically"
        df_ratn_cols = pd.concat(norm_int_ratn, axis=0).reset_index(drop=True)

        # write csv to disk
        reorg.append(df_ratn_cols)

    # concat "horizontally"
    df_prism_reorg = pd.concat(reorg, axis=1)

    return df_prism_reorg
```

1.3 Counting Mean Cell Ns

Again, all parameterized functions will get set aside into module for future use and standardization

[6]:
```
def count_imgs(df, sid, iid):
    '''
    takes a dataframe and counts the number of unique strings that occur in the
    "image_name" col for each rat in "rat_n" col
    args:
        df: pd.core.frame.DataFrame(n, m)
            n: the number of rows,
            m: the number of features
        sid: str, denoting the name of the col containing unique subject ids
        iid: str, denoting the name of the col containing unique image ids
    return:
        df_imgn: pd.core.frame.DataFrame(n=|sid|), m=2)
            n: the number of rows, equal to the cardinality of the sid set
            (the number of unique ID strings in sid)
            this df contains 2 cols: a sid col, and an iid col containing counts
    '''
    assert iid in df.columns

    df_imgn = df.groupby([sid])[[sid, iid]]\
        .apply(lambda x: len(np.unique(x[iid])))\
        .reset_index(name='image_n')

    return df_imgn

def count_cells(df, cols):
```

```
    '''
    takes a df and counts the number of instances each distinct row
    (created by unique combinations of labels from columns indicated
    by cols arg); counts are reported in a new col called "cell_counts"
    args:
        df: pd.core.frame.DataFrame(N, M); N: the number of rows, M: the
            number of cols (assumed to have already been split by stain_type)
        cols: list(n), n: the number of cols over which to count distinct rows
    return:
        df_counts: pd.core.frame.DataFrame(N,M+1)
    '''
    df_counts = df.value_counts(cols)\
        .reset_index(name='cell_counts')\
        .sort_values(by=cols)

    return df_counts

def sum_cells(df, cols, iid):
    '''
    takes cell count df, groups by cols denoted in cols list and computes sum
    of cell_counts col for each group. Adds new column "cell_count_sums"
    containing sums.
    args:
        df: pd.core.frame.DataFrame(N, M), N: the number of rows (N=|id_col|),
            M: the number of cols, must contain col called "cell_counts"
        cols: list(M-2), list containing col name strings that define each␣
 ↪group
            for group by and reduction (in this case summing)
        iid: str, denotes
    return:
        df_sums: pd.core.frame.DataFrame; dataframe containing summed cell
            counts per subject id.
    '''
    # remove image id col (we want to sum counts across all images per rat)
    reduce_cols = list(filter(lambda x: x != iid, cols))

    if 'scaled_counts' in df.columns:
            # group by, reduce
        df_sums = df.groupby(by=reduce_cols)[cols]\
            .apply(lambda x: np.sum(x.scaled_counts))\
            .reset_index(name='cell_count_sums')

    else:
        # group by, reduce
        df_sums = df.groupby(by=reduce_cols)[df.columns]\
            .apply(lambda x: np.sum(x.cell_counts))\
            .reset_index(name='cell_count_sums')
```

```
    return df_sums

def average_counts(df_sums, df_ns, cols, sid, iid):
    '''
    takes df of cell count sums and df of image ns, and computes the mean cell
    n (divides cell count sums by the number of images) for each subject.
    args:
        df_sums: pd.core.frame.DataFrame(ni, mi), ni: the number of rows
            (ni=|sid|), mi: the number of cols (mi = |cols|); must
            contain a col "cell_count_sums".
        df_ns: pd.core.frame.DataFrame(nj, mj), nj: the number of rows
            (nj=|sid|), mj: the number of cols (mj=2); must contain a col
            "image_n"
        cols: list(n), n: the number of cols (contains all cols necessary to
            create every unique group combination)
        sid: str, denoting the name of the col containing unique subject ids
        iid: str, denoting the name of the col containing unique image ids
    return:
        mean_cell_ns: pd.core.frame.DataFrame(N,M), N: the number of rows (N=
        |sid|), M: the number of cols (M=|cols|+2)

    '''
    # list of cols with out image id, since it was removed during the reduction␣
 ↪step
    reduce_cols = list(filter(lambda x: x != iid, cols))

    # compute mean cell n
    mean_cell_ns = df_sums.join(df_ns.set_index(sid), on=sid, how='inner')\
        .sort_values(by=reduce_cols)
    mean_cell_ns['mean_cell_n'] = mean_cell_ns.cell_count_sums / mean_cell_ns.
 ↪image_n

    # reorder so that subject id is the first col
    col_reorder = [sid] + list(filter(lambda x: x != sid, list(mean_cell_ns.
 ↪columns)))
    mean_cell_ns = mean_cell_ns[col_reorder]

    return mean_cell_ns

def mean_cell_n(df_stain, df_full, cols, sid, iid, return_counts=False):
    '''
    wrapper function to compute mean cell ns; magnification/zoom factor
    is assuemd to be equal across all images. NOTE that we count total image
    ns based on full cleaned dataset: it may be the case the not every image
    contains every stain type combination, and we must still count images
    with 0 cells of a particular stain type towards the total number of images.
```

```
    args:
        df_stain: pd.core.frame.DataFrame; df containing data for a given stain␣
↪type
        df_full: pd.core.frame.DataFrame; df containing data for full (cleaned)␣
↪set
        cols: list, contains str denoting col names for grouping
        sid: str, col name denoting col containing unique subject ids
        iid: str, col name denoting col containing unique image ids
        return_counts: bool, flag for added utility during debugging
    return:
        mean_cell_ns: pd.core.frame.DataFrame; df containing final mean cell ns
        cell_counts: pd.core.frame. DataFram; df containing cell counts per
            image (for debugging)

    '''
    # count n of unique image names per subject
    img_ns = count_imgs(df_full, sid, iid)

    # count n of cells per image for each subject
    cell_counts = count_cells(df_stain, cols)

    # sum cell counts across all images for each subject
    cell_sums = sum_cells(cell_counts, cols, iid)

    # compute mean cell count per image for each subject
    mean_cell_ns = average_counts(cell_sums, img_ns, cols, sid, iid)

    if not return_counts:
        return mean_cell_ns

    return (cell_counts, mean_cell_ns)
```

1.4 Time to run it!

Normalize Intensity, write to disk

[7]:
```
grp = 'VR5_KET'
for stain in df_subset.stain_type.unique():

    # split by stain
    df_stain = df_subset[df_subset.stain_type == stain]

    # normalize to FR1_SAL
    df_norm = normalize_intensity(df_stain, norm_condition='FR1_SAL')
    df_norm.to_csv(f'{grp}_{stain}_single_NORM.csv')

    # reorganize into cols for prism
```

```
    df_prism = prism_reorg(df_norm, grp)
    df_prism.to_csv(f'{grp}_{stain}_PRISM.csv')

# let's take a look at one of our final output dataframes, organized for entry␣
 ↪into prism
print(stain)
df_prism
```

```
WFA
```

[7]:

	FR1_KET	FR1_SAL	VR5_KET	VR5_SAL
0	3.573138	5.740431	-1.730680	-0.584066
1	-0.228130	0.071669	-1.073475	-0.080524
2	0.373468	0.019555	-0.500109	-0.956723
3	-1.247351	3.060095	1.072946	-0.567922
4	0.126912	3.312895	-1.526467	-0.273813
..	…	…	…	…
354	NaN	NaN	-1.250855	NaN
355	NaN	NaN	1.005051	NaN
356	NaN	NaN	-1.339845	NaN
357	NaN	NaN	-1.134914	NaN
358	NaN	NaN	-0.498651	NaN

[359 rows x 4 columns]

Count mean cell ns, write to disk

[8]:
```
# count n of unique image names per subject
sid = 'rat_n'
iid = 'image_name'
cols = ['treatment', 'stain_type', sid, iid]

# wrapper fn calls
for stain in df_cleaned.stain_type.unique():

    # split by stain type
    df_stain = df_cleaned[df_cleaned.stain_type == stain]

    # compute mean cell ns
    df_means = mean_cell_n(df_stain, df_cleaned, cols, sid, iid)

    # write to disk
    df_means.to_csv(f'{grp}_{stain}_mean_cell_ns.csv')

# let's take a look at one of our final output dataframes
print(stain)
df_means
```

WFA

[8]:

	rat_n	treatment	stain_type	cell_count_sums	image_n	mean_cell_n
0	KET-10-12	FR1_KET	WFA	32	5	6.40
1	KET-9-1	FR1_KET	WFA	29	4	7.25
2	PE-11-1	FR1_KET	WFA	20	5	4.00
3	PE-11-2	FR1_KET	WFA	25	5	5.00
4	PE-11-3	FR1_KET	WFA	46	5	9.20
5	PE-12-1	FR1_KET	WFA	26	5	5.20
6	PE-12-2	FR1_KET	WFA	39	5	7.80
7	PE-12-7	FR1_KET	WFA	47	5	9.40
8	KET-10-1	FR1_SAL	WFA	35	5	7.00
9	KET-10-5	FR1_SAL	WFA	32	5	6.40
10	KET-8-2	FR1_SAL	WFA	33	5	6.60
11	KET-9-2	FR1_SAL	WFA	48	5	9.60
12	KET-9-4	FR1_SAL	WFA	46	5	9.20
13	KET-9-5	FR1_SAL	WFA	45	5	9.00
14	KET-9-6	FR1_SAL	WFA	40	5	8.00
15	KET-10-14	VR5_KET	WFA	38	5	7.60
16	KET-8-7	VR5_KET	WFA	32	4	8.00
17	PE-11-4	VR5_KET	WFA	35	5	7.00
18	PE-11-5	VR5_KET	WFA	30	5	6.00
19	PE-11-6	VR5_KET	WFA	52	5	10.40
20	PE-11-7	VR5_KET	WFA	38	5	7.60
21	PE-13-2	VR5_KET	WFA	39	5	7.80
22	PE-13-3	VR5_KET	WFA	40	5	8.00
23	PE-13-6	VR5_KET	WFA	55	5	11.00
24	KET-10-2	VR5_SAL	WFA	31	5	6.20
25	KET-10-3	VR5_SAL	WFA	35	5	7.00
26	KET-10-4	VR5_SAL	WFA	26	5	5.20
27	PE-13-1	VR5_SAL	WFA	66	5	13.20
28	PE-13-11	VR5_SAL	WFA	52	5	10.40
29	PE-13-9	VR5_SAL	WFA	48	5	9.60

2 Negative Intensity?

I noticed that in the test output of the normalized intensities reshaped for prism, we were getting some negative values for normalized intensity. Negative values don't really make sense here. This would mean that the observed effect of the treatment results in cells actually being **dimmer** than background.

The only way for negative numbers to arise here is if the mean intensity before background subtraction was **less than** the background at the time the image was captured. If the average signal in a selected region was less than (or not different from) background, that is its signal to noise ratio (SNR) is less than 1, it is unclear to me why we would consider this selection as an ROI.

This means that either, selections were made where cells actually showed less fluorescence than background (did the stain work?), or background regions were poorly selected (did the background

selection include ROIs?).

2.0.1 Distribution of dim selections across stain types

[9]:
```
df_full = df_full[~df_full.duplicated()]
df_full['mean_intensity'] = df_full.mean_intensity.astype('f')

df_lt = df_full.query('mean_intensity < background')

print('total number of cells where mean intensity < background, per stain:')
print(df_lt.stain_type.value_counts())

print('\npercent of cells where mean intensity < background, per stain:')
print(df_lt.stain_type.value_counts() / df_full.stain_type.value_counts() * 100)
```

```
total number of cells where mean intensity < background, per stain:
stain_type
Npas4    11324
cFos      4828
WFA        644
PV         466
Name: count, dtype: int64

percent of cells where mean intensity < background, per stain:
stain_type
Npas4    91.736876
PV       27.622999
WFA      55.517241
cFos     51.241775
Name: count, dtype: float64
```

2.0.2 Distribution of dim selections across rats

[10]:
```
print('\npercent cells where mean intensity < background, per rat:')
print(df_lt.rat_n.value_counts() / df_full.rat_n.value_counts() * 100)
```

```
percent cells where mean intensity < background, per rat:
rat_n
KET-10-1     65.707434
KET-10-12    68.518519
KET-10-14    63.389831
KET-10-2     53.207547
KET-10-3     60.782443
KET-10-4     68.148148
KET-10-5     63.955638
KET-8-2      62.178218
KET-8-7      69.841270
```

```
KET-9-1      72.389791
KET-9-2      74.835526
KET-9-4      81.145251
KET-9-5      69.240506
KET-9-6      59.211823
PE-11-1      70.720000
PE-11-2      75.552050
PE-11-3      75.396825
PE-11-4      71.587302
PE-11-5      82.324841
PE-11-6      79.788839
PE-11-7      81.153305
PE-12-1      69.776119
PE-12-2      71.132765
PE-12-7      68.951194
PE-13-1      80.444965
PE-13-11     66.795367
PE-13-2      71.444824
PE-13-3      79.567308
PE-13-6      79.648241
PE-13-9      78.071334
Name: count, dtype: float64
```

[ ]: