Ketamine VR5 - COLOCALIZED DATA ONLY

Jonathan Ramos 2/2/2024

From the previous set of notebooks, we developed a new method to deal with negative intensity values. Rather than subtracting out the background to get an intenisty metric that takes into account the background fluorescence, I tried taking the ratio of the mean ROI intensity to the background intensity, generating an SNR value that (unlike the previous metric) cannot be negative. Next, visualizations of the distributions of intensity SNRs per stain type, along with visual inspection of actual image data (the "worst offenders," with the most negative intensity values) an acceptable SNR threshold was determined that removed ROIs whose SNR is too low to consider as a legitimate observation but still retains those dim ROIs that may still reasonably be considered as true observations of real cells (confirmed by AG, JR, and BS). Below are the plots of the SNR distributions, annotated with the SNR threshold.

For each stain type we determined the following SNR thresholds:

- PV : 0.8
- cFos : 0
- Npas4 : 0.8
- WFA : 0.85

As before, col names are different than older verions of these csvs, 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. we have the following file naming scheme denoting stain types:

- *_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 [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import glob
```

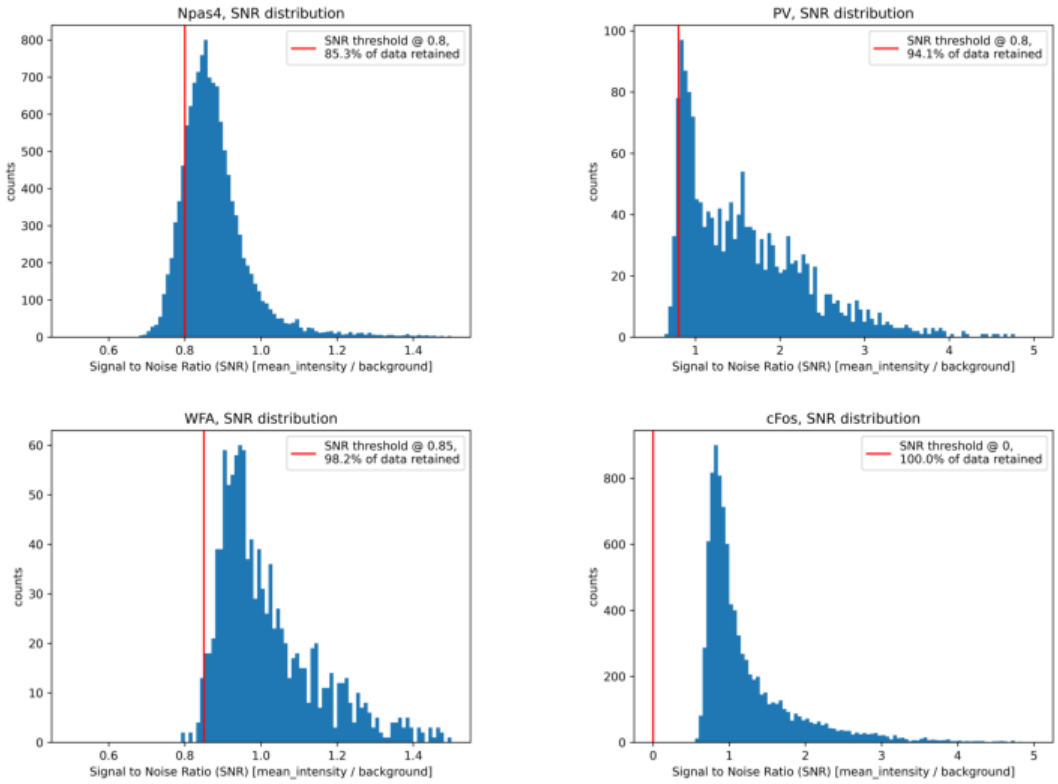
```
/var/folders/b2/3h2lpxx14kgb12pp_7pltxnc0000gn/T/ipykernel_98745/4078894385.py:2: DeprecationWarnin
g:
Pyarrow will become a required dependency of pandas in the next major release of pandas (pandas 3.
0),
(to allow more performant data types, such as the Arrow string type, and better interoperability wi
th 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
```

SNR distributions and thresholds

In [2]:
```
plots = glob.glob('SNR_distributions/*.png')
plots.sort()
f, ax = plt.subplots(nrows=2, ncols=2, figsize=(10,7))
ax = ax.flatten()
for i, p in enumerate(plots):
    img = mpimg.imread(p)
    ax[i].axis('off')
    ax[i].imshow(img)

plt.subplots_adjust(wspace=0, hspace=0)
```



Cleaning, Wrangling Data

loading data, stitching sets together

loading re-exported set (previously contained weird cells?)

In [226]:
```
# load cohort key (there's a few empty rows at the end)
df_key = pd.read_csv('KETAMINE_COHORT KEY.csv').dropna()

df_rexp = [pd.read_csv(f) for f in glob.glob('Re-exported Colocalized Results/*.csv') if not 'batch
_export' in f]

# because these data contain colocalization for stain type combinations for 2, 3, and 4 stains,
# there are a variable number of cols based on the number stains used for colocalization
print(np.unique([len(df.columns) for df in df_rexp]))

# separating out colocalization types (double/triple/quad) by number of cols
double_rexp = [df for df in df_rexp if len(df.columns) == 30]
triple_rexp = [df for df in df_rexp if len(df.columns) == 31]
quad_rexp = [df for df in df_rexp if len(df.columns) == 32]

def preprocessing(coloc_dfs):
    '''
    small fn to automate some of the preprocessing steps that must be done
    on each df before we can concat: removing unncessary whitespaces,
    dropping rows without an intensity measurement
    '''
    # remove leading whitespace from col names
    for df in coloc_dfs:
        df.columns = [col.replace(' ', '') for col in df.columns]

    # drop rows without intensity measurement
    coloc_dfs = [df[~df['mean-background'].isna()] for df in coloc_dfs]

    # remove leading whitespace from filenames (note access via .loc to avoid setting with copy)
    for df in coloc_dfs:
        df.loc[:, 'filename'] = df.filename.str.replace(' ', '')

    return coloc_dfs

double_rexp = preprocessing(double_rexp)
triple_rexp = preprocessing(triple_rexp)
quad_rexp = preprocessing(quad_rexp)

# we have our data still split across hundreds of small dfs; let's take a look at one of them
triple_rexp[0]
```

```
[30 31 32]
```

Out[226]:

	stain	colocw/Parvalbumin	colocw/hand-drawn	colocw/hand-drawn.1	roi_id	CoM_x	CoM_y	pixel_area	background	mean_intensity	...	fe
0	Parvalbumin	NaN	0-FFF-00069	0-FFF-00003	0-000-00000	147.05	391.43	522.0	141.5172	212.0318	...	
1	Parvalbumin hand-drawn	-	-	-	0-FFF-00013	167.38	333.88	152.0	141.5172	121.8087	...	
2	Parvalbumin hand-drawn	-	-	-	0-FFF-00012	27.46	131.04	82.0	141.5172	104.7599	...	
3	Parvalbumin hand-drawn	-	-	-	0-FFF-00011	41.58	140.37	120.0	141.5172	103.0452	...	
4	Parvalbumin hand-drawn	-	-	-	0-FFF-00010	106.88	327.00	251.0	141.5172	117.2991	...	
...	...	...	...	...	...	...	...	...	...	...	...	
87	WFA hand-drawn	0-000-00000	0-FFF-00069	NaN	0-FFF-00003	152.59	391.10	774.0	110.0857	103.5261	...	
88	WFA hand-drawn	-	-	-	0-FFF-00006	213.53	9.59	279.0	110.0857	105.9867	...	
89	WFA hand-drawn	-	-	-	0-FFF-00004	332.66	432.75	1015.0	110.0857	124.8189	...	
90	WFA hand-drawn	-	-	-	0-FFF-00002	157.33	326.14	691.0	110.0857	109.3734	...	
91	WFA hand-drawn	-	-	-	0-FFF-00001	252.83	36.06	660.0	110.0857	107.7370	...	

92 rows × 31 columns

loading original set

In [227]:
```
df_orig = [pd.read_csv(f) for f in glob.glob('CSV_FILES/*/*.csv')]

# separating out colocalization types (double/triple/quad) by number of cols
double_orig = [df for df in df_orig if len(df.columns) == 30]
triple_orig = [df for df in df_orig if len(df.columns) == 31]
quad_orig = [df for df in df_orig if len(df.columns) == 32]

double_orig = preprocessing(double_orig)
triple_orig = preprocessing(triple_orig)
quad_orig = preprocessing(quad_orig)
```

pulling out coloc data

In [228]:
```
# we don't wont rows where all coloc cols are '-'
# (indicates this cell was not colocalized with any of the other given stain types)
def drop_dash(df, coloc_type):
    '''
    takes a dataframe and drops all rows where all coloc rows
    are '-'.
    args:
        pd.core.frame.DataFrame(N,M), N: the number of rows, M: the number of cols;
            coloc cols must contain 'coloc' in col name.
    '''
    if not coloc_type in {'double', 'triple', 'quad'}:
        raise ValueError('coloc_type must be either "double", "triple", or "quad"')

    cols = df.columns
    n_coloc_cols = np.array(['coloc' in f for f in list(cols)]).sum()

    # check that cols 1 and 2 are coloc cols
    assert 'coloc' in cols[1]
    assert 'coloc' in cols[2]
    if coloc_type == 'double':
        # check that the number of coloc cols is exactly 2
        assert n_coloc_cols == 2

        df_dropped = df.replace('-', np.NaN)\
            .dropna(subset=[cols[1], cols[2]], how='all')

    if coloc_type == 'triple':
        # check that col 3 is coloc col
        assert 'coloc' in cols[3]

        # check that there are exactly 3 coloc cols
        assert n_coloc_cols == 3

        df_dropped = df.replace('-', np.NaN)\
            .dropna(subset=[cols[1], cols[2], cols[3]], how='all')

    if coloc_type == 'quad':
        # check that cols 3 and 4 are coloc cols
        assert 'coloc' in cols[3]
        assert 'coloc' in cols[4]

        # check that there are exactly 4 coloc cols
        assert n_coloc_cols == 4

        df_dropped = df.replace('-', np.NaN)\
            .dropna(subset=[cols[1], cols[2], cols[3], cols[4]], how='all')

    return df_dropped.reset_index()

# rexp
double_rexp = [drop_dash(df, coloc_type='double') for df in double_rexp]
triple_rexp = [drop_dash(df, coloc_type='triple') for df in triple_rexp]
quad_rexp = [drop_dash(df, coloc_type='quad') for df in quad_rexp]

# repeat for orig
double_orig = [drop_dash(df, coloc_type='double') for df in double_orig]
triple_orig = [drop_dash(df, coloc_type='triple') for df in triple_orig]
quad_orig = [drop_dash(df, coloc_type='quad') for df in quad_orig]

# let's check
triple_rexp[0]
```

```
/var/folders/b2/3h2lpxx14kgb12pp_7pltxnc0000gn/T/ipykernel_98745/2233548078.py:24: FutureWarning: D
owncasting behavior in `replace` is deprecated and will be removed in a future version. To retain t
he old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavi
or, set `pd.set_option('future.no_silent_downcasting', True)`
  df_dropped = df.replace('-', np.NaN)\
```

Out[228]:

	index	stain	colocw/Parvalbumin	colocw/hand-drawn	colocw/hand-drawn.1	roi_id	CoM_x	CoM_y	pixel_area	background	...	feret_angle
0	0	Parvalbumin	NaN	0-FFF-00069	0-FFF-00003	0-000-00000	147.05	391.43	522.0	141.5172	...	0.0000
1	13	General Segmentation v1 hand-drawn	0-000-00000	NaN	0-FFF-00003	0-FFF-00069	146.15	391.78	133.0	535.9423	...	0.0000
2	87	WFA hand-drawn	0-000-00000	0-FFF-00069	NaN	0-FFF-00003	152.59	391.10	774.0	110.0857	...	147.6927

3 rows × 32 columns

Relabeling incorrect data

In [229]:
```
# there were some issues with the naming/cohort key
def relabel_filename(df, dct):
    df['filename'] = df.filename.replace(dct, regex=True)
    return df

# PE-12-7 was incorrectly labeled as PE-12-3; this was confirmed by JR and AG by checking slide boo
ks/hard copies of behavior data
# similarly KET-8-2 was incorrectly labeled as KET-8-5; this was confirmed by JR and AG  by checkin
g slide books. this wouldn't really
# change anything since they received the same treatment but let's just change it to the correct la
bel anyway

relabel = {
    'PE-12-3': 'PE-12-7',
    'KET-8-5': 'KET-8-2'
}

double_rexp = [relabel_filename(df, relabel) for df in double_rexp]
triple_rexp = [relabel_filename(df, relabel) for df in triple_rexp]
quad_rexp = [relabel_filename(df, relabel) for df in quad_rexp]

double_orig = [relabel_filename(df, relabel) for df in double_orig]
triple_orig = [relabel_filename(df, relabel) for df in triple_orig]
quad_orig = [relabel_filename(df, relabel) for df in quad_orig]
```

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_intensity
- background

In [230]:
```
def get_ratn(df):
    '''
    '''
    df['rat_n'] = df.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.rat_n.apply(lambda x: len(x.split('-')) == 3).sum() == len(df)
    assert df.rat_n.apply(lambda x: x.split('-')[0].isalpha()).sum() == len(df)
    assert df.rat_n.apply(lambda x: x.split('-')[1].isnumeric()).sum() == len(df)
    assert df.rat_n.apply(lambda x: x.split('-')[2].isnumeric()).sum() == len(df)

    return df

def get_treatment(df, treatment):
    '''
    '''
    # creating new treatment col by mapping from cohort key dict
    df['treatment'] = df.rat_n.map(treatment)

    # check that all rat_ns were accounted for
    assert df.treatment.isna().sum() == 0

    return df


def get_staintype(df, stains):
    '''
    '''
    # creating new stain_type col from filename
    df['stain_type'] = df.filename.replace(stains, regex=True)

    # check that all filenames were accounted for
    assert df.stain_type.isna().sum() == 0

    return df

def get_imagename(df):
    '''
    '''
    df['image_name'] = df.filename.replace({'_[0-9]\.tif': ''}, regex=True)

    return df

def col_wrapper(df, treatment, stains):
    '''
    '''

    df_ratn = get_ratn(df)
    df_treatment = get_treatment(df_ratn, treatment)
    df_staintype = get_staintype(df_treatment, stains)
    df_imgname = get_imagename(df_staintype)

    return df_imgname
```

building coloc stain type col from single stain type col

we know that for a given coloc csv, the only stain types represented in the set must be the ones used for colocalization. Because of the way these smaller csvs are grouped (by image and importantly, by combination) we don't even need to look at all those ugly "coloc w/" cols. Instead since we already know if a given dataframe represents double, triple, or quad labeled data, we can simply take the single labeled stain_type label and add on to it the other stain_types coming from the same dataframe.

In [231]:
```
def coloc_staintype(df, coloc):
    '''
    '''
    stains = df.stain_type.unique()
    set_stains = set(stains)

    # for any n number stains, we need any given stain and its complement (all others except that o
ne)
    stain_combinations = [(stain, tuple(sorted(list(set_stains.difference({stain}))))) for stain in
stains]

    # construct string, build dict
    # double label
    if coloc == 'double':
        # check that we have exactly two stains
        if len(stains) == 2:
            # build dictionary for replace
            coloc_comb = dict([(stain, f'{stain}_coloc_w_{comb[0]}') for stain, comb in stain_combi
nations])

        elif len(stains) < 2:
            coloc_comb = dict([(stain, f'lonely_{stain}') for stain in stains])


    # triple label
    if coloc == 'triple':
        # check that we have exactly three stains
        if len(stains) == 3:
            # build dictionary for replace
            coloc_comb = dict([(stain, f'{stain}_coloc_w_{comb[0]},{comb[1]}') for stain, comb in s
tain_combinations])

        elif len(stains) < 3:
            coloc_comb = dict([(stain, f'lonely_{stain}') for stain in stains])

    # quad
    if coloc == 'quad':
        # check that we have exatly four stains
        if len(stains) == 4:
            # build dictionary for replace
            coloc_comb = dict([(stain, f'quad_{stain}') for stain in stains])

        elif len(stains) < 4:
            coloc_comb = dict([(stain, f'lonely_{stain}') for stain in stains])

    # toss coloc stain type strings into col
    df['coloc_stain_type'] = df.stain_type.replace(coloc_comb)

    return df
```

In [232]:
```
# building a cohort key dictionary from df_key
treatment = dict(zip(df_key.Subject, df_key.TX.replace({' ': '_'}, regex=True)))

stains = {
    '.*_2.tif$' : 'PV',
    '.*_3.tif$' : 'cFos',
    '.*_4.tif$' : 'Npas4',
    '.*_5.tif$' : 'WFA'
}

# rexp
double_rexp = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='double') for df in double
_rexp]
triple_rexp = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='triple') for df in triple
_rexp]
quad_rexp = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='quad') for df in quad_rexp]

# repeat for orig
double_orig = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='double') for df in double
_orig]
triple_orig = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='triple') for df in triple
_orig]
quad_orig = [coloc_staintype(col_wrapper(df, treatment, stains), coloc='quad') for df in quad_orig]
```

SNR Threshold

For each stain type we determined the following SNR thresholds:

- PV : 0.8
- cFos : 0
- Npas4 : 0.8
- WFA : 0.85

In [233]:
```
def get_snr(df):
    '''
    '''
    df['snr'] = df['mean_intensity'].astype('f') / df['background'].astype('f')

    return df

def filter_snr(df, snr_cutoff):
    '''
    '''
    df_threshold = pd.concat([df.query(f'stain_type == "{stain}"').query(f'snr > {snr_cutoff[stai
n]}') for stain in df.stain_type.unique()])

    return df_threshold

# build dict of snr thresholds
snr_threshold = {
    'PV': 0.8,
    'cFos': 0,
    'Npas4': 0.8,
    'WFA': 0.85
}

# rexp
double_rexp = [filter_snr(get_snr(df), snr_threshold) for df in double_rexp]
triple_rexp = [filter_snr(get_snr(df), snr_threshold) for df in triple_rexp]
quad_rexp = [filter_snr(get_snr(df), snr_threshold) for df in quad_rexp]

# repeat for orig
double_orig = [filter_snr(get_snr(df), snr_threshold) for df in double_orig]
triple_orig = [filter_snr(get_snr(df), snr_threshold) for df in triple_orig]
quad_orig = [filter_snr(get_snr(df), snr_threshold) for df in quad_orig]
```

Retaining only full coloc cell groupings

The SNR threshold may have caused some colocalized cells to become "incomplete." That is, say we had a PV cell that was colocalized to a c-Fos cell, resulting in two entries in our double labeled set: a PV_coloc_w/c-Fos cell, and a c-Fos_coloc_w/_PV cell. It may be the case that this particular PV cell was removed by the SNR threshold, but the c-Fos cell was not. If the PV cell was removed, we would still have that left over c-Fos_coloc_w/_PV cell in our set.

The kernel of the issue is that we set a threshold based on one stain type, but we want to remove a cell of a different stain type; that is, we want to maintain full coloc groupings among our remaining colocalized cells.

My approach here is to create groupings (sets) of coloc cells roi ids and remove cells who do not meat the following criteria:

- each set must contain exactly the number of roi ids required to create the coloc stain type combinations (2 for double, 3 for triple, and 4 for quad)
- within each image, each unique set must appear the appropriate number of times (2 for double, 3 for triple, and 4 for quad)

However, before creating groupings the following issues must be addressed:

- colocw/ titles are not consistent or accurately named
- roi id strings are not unique across stain types or across images

NOTE because not only are roi id strings reused across stain types, they are reset for each unique image, we must carry out this process on an image by image basis. We may only concat our distributed dfs after subset selection and data replacement (when cols are finally standardized across images and stain type combinations).

In [264]:
```
def rename_colocw_cols(df):
    '''
    '''
    coloc_cols = [col for col in df.columns if 'colocw/' in col]
    coloc_rename = []
    for col in coloc_cols:
        # for a given colocw/ col, get unique (single) stain_type (there should be exactly 1)
        single_stain_type = df[df[col].isna()].stain_type.unique()
        assert len(single_stain_type) == 1

        # append in (key, val) format
        coloc_rename.append((col, f'coloc_w/_{single_stain_type.item()}'))

    d_coloc_rename = dict(coloc_rename)
    df = df.rename(columns=d_coloc_rename)

    return df

def rename_roi_ids(df):
    '''
    '''
    coloc_cols = [col for col in df.columns if 'coloc_w/' in col]
    for col in coloc_cols:
        stain = col.split('_')[-1]
        # add stain string to end of roi id if not nan
        df[col] = df[col].apply(lambda x: x+f'_{stain}' if pd.notnull(x) else x)

    df['roi_id'] = df.apply(lambda x: x.roi_id + '_' + x.stain_type, axis=1)

    return df

def count_coloc_groupings(df):
    '''
    '''
    roi_id_cols = [col for col in df.columns if 'coloc_w/_' in col] + ['roi_id']
    groups = df.apply(lambda x: tuple(sorted([y for y in list(x[roi_id_cols]) if pd.notnull(y)])),
axis=1)
    df['coloc_roi_id_grouping'] = groups
    groups, counts = np.unique(df.coloc_roi_id_grouping, return_counts=True)
    d_group_counts = dict(zip(groups, counts))

    df['roi_id_grouping_counts'] = df.coloc_roi_id_grouping.map(d_group_counts)

    return df

def paired(df, coloc_type):
    '''
    '''
    # remove unnecesary index col
    if 'index' in df.columns:
        df = df.drop('index', axis=1)

    # enforce coloc_type
    if not coloc_type in {'double', 'triple', 'quad'}:
        raise ValueError('coloc_type must be either "double", "triple", or "quad"')

    cols = df.columns
    n_coloc_cols = np.array(['colocw/' in f for f in list(cols)]).sum()

    if coloc_type == 'double':
        # check that the number of coloc cols is exactly 2, and
        # that there are exactly 2 unique (single) stain_types in df
        assert n_coloc_cols == 2
        assert len(df.stain_type.unique()) == 2

        # rename coloc_w/ cols; rename roi_id strings
        df_rename = rename_roi_ids(rename_colocw_cols(df))

        # create roi id coloc groupings, count groupings
        df_grouped = count_coloc_groupings(df_rename)

        # only return rows that were fully paired; for double we expect exactly 2
        df_paired = df_grouped[df_grouped.roi_id_grouping_counts == 2]

    if coloc_type == 'triple':
        # check that there are exactly 3 coloc cols, and that
```

```
        # there are exactly 3 unique (single) stain_types in df
        assert n_coloc_cols == 3
        assert len(df.stain_type.unique()) == 3

        # rename coloc_w/ cols; rename roi_id strings
        df_rename = rename_roi_ids(rename_colocw_cols(df))

        # create roi id coloc groupings, count groupings
        df_grouped = count_coloc_groupings(df_rename)

        # only return rows that were fully paired; for triple we expect exactly 3
        df_paired = df_grouped[df_grouped.roi_id_grouping_counts == 3]

    if coloc_type == 'quad':
        # check that there are exactly 4 coloc cols, and that
        # there are exactly 3 unique (single) stain_types in df
        assert n_coloc_cols == 4
        assert len(df.stain_type.unique()) == 4

        # rename coloc_w/ cols; rename roi_id strings
        df_rename = rename_roi_ids(rename_colocw_cols(df))

        # create roi id coloc groupings, count groupings
        df_grouped = count_coloc_groupings(df_rename)

        # only return rows that were fully paired; for quad we expect exactly 4
        df_paired = df_grouped[df_grouped.roi_id_grouping_counts == 4]

    return df_paired


# rexp
# additional minimum length constraints because for double label, we require df with at
# least 2 unique stain_types. similarly, for triple label we require df with at least 3
# unique stain_types and for quad label we requiredf with at least 4 unique stain_types
double_rexp_paired = [paired(df, coloc_type='double') for df in double_rexp if len(df.stain_type.un
ique()) >= 2]
triple_rexp_paired = [paired(df, coloc_type='triple') for df in triple_rexp if len(df.stain_type.un
ique()) >= 3]
quad_rexp_paired = [paired(df, coloc_type='quad') for df in quad_rexp if len(df.stain_type.unique
()) >= 4]

# repeat for orig
double_orig_paired = [paired(df, coloc_type='double') for df in double_orig if len(df.stain_type.un
ique()) >= 2]
triple_orig_paired = [paired(df, coloc_type='triple') for df in triple_orig if len(df.stain_type.un
ique()) >= 3]
quad_orig_paired = [paired(df, coloc_type='quad') for df in quad_orig if len(df.stain_type.unique
()) >= 4]
```

Replace old data

subset selection

In [266]:
```
# taking out only the col subset we need
cols = ['rat_n', 'treatment', 'stain_type', 'coloc_stain_type', 'filename', 'image_name', 'CoM_x',
'CoM_y', 'mean_intensity', 'background', 'snr', 'coloc_roi_id_grouping', 'roi_id_grouping_counts']

# rexp
df_double_rexp = pd.concat([df[cols] for df in double_rexp_paired])
df_triple_rexp = pd.concat([df[cols] for df in triple_rexp_paired])
df_quad_rexp = pd.concat([df[cols] for df in quad_rexp_paired])

# repeat for orig
df_double_orig = pd.concat([df[cols] for df in double_orig_paired])
df_triple_orig = pd.concat([df[cols] for df in triple_orig_paired])
df_quad_orig = pd.concat([df[cols] for df in quad_orig_paired])
```

Replacing original data with new data

In [268]:
```
def replace_im_subset(df_orig, df_rexp):
    '''
    '''
    # Check that all the images in the rexp set were also in the in orig set
    orig_imset = set(df_orig.image_name)     # note the switch from filename to image_name; there w
ere some
    rexp_imset = set(df_rexp.image_name)     # images in the rexported set that were not previously
in the orig set
    # assert rexp_imset.issubset(orig_imset)

    # toss imnames into list
    orig_imnames = sorted(list(orig_imset))
    rexp_imnames = sorted(list(rexp_imset))

    # build dicts from imnames where key is imname and val is df for that image
    d_orig = dict(zip(orig_imnames, [df_orig.query(f'image_name == "{imname}"') for imname in orig_
imnames]))
    d_rexp = dict(zip(rexp_imnames, [df_rexp.query(f'image_name == "{imname}"') for imname in rexp_
imnames]))

    # build new df from orig imnames unless rexp fname
    df_full_repl = []
    for imname in orig_imnames:

        # if imname not in rexp set, take from orig set
        if not imname in rexp_imset:
            df_full_repl.append(d_orig[imname])

        # else if imname in rexp, take from rexp set
        elif imname in rexp_imset:
            df_full_repl.append(d_rexp[imname])

        else:
            print('check your imnames')

    # now we can FINALLY concat; number of cols and col names match after subset selection
    # which we couldn't do until stain types were dealt with and old image data was replaced
    df_full_repl = pd.concat(df_full_repl)

    return df_full_repl

df_double = replace_im_subset(df_double_orig, df_double_rexp)
df_triple = replace_im_subset(df_triple_orig, df_triple_rexp)
df_quad = replace_im_subset(df_quad_orig, df_quad_rexp)

# final concat
df_coloc = pd.concat([df_double, df_triple, df_quad])

# let's take a look
print(df_coloc.shape)
df_coloc.head()
```

```
(18514, 13)
```

Out[268]:

	rat_n	treatment	stain_type	coloc_stain_type	filename	image_name	CoM_x	CoM_y	mean_intensity	background	
0	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	306.85	306.78	485.9018	573.9955	0.846
1	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	170.95	312.57	503.089	573.9955	0.876
2	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	299.44	422.83	544.4831	573.9955	0.948
3	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	154.15	476.33	524.3108	573.9955	0.913
4	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	36.62	458.4	471.7593	573.9955	0.82

check for lonely stain types

from above, while building the coloc stain type col, I encountered a weird edge case where some PV cells were colocalized Npas4 cells, but in the same image no Npas4 cells were reported to be colocalized with PV. I caught this edge case because it failed the assertion that in a double stained csv, there were two stain types. In this case, I called the cell with potentially false positive colocalization "lonely_PV" when constructing the label string. Upon further inspection of the CoM_X and CoM_Y coordinates, the cells that were apparently colocalized were not even near each other and so I find it is therefore appropriate to remove these types of cells from the set.

In [269]:
```
df_coloc = df_coloc[~df_coloc.coloc_stain_type.str.contains('lonely')].reset_index()
print(df_coloc.shape)
df_coloc.head()

# it was only a difference of two cells anyway. The edge case I observed must have been the only on
e.
# all lonely stain types would have been removed in the pairing process anyway
```

```
(18514, 14)
```

Out[269]:

	index	rat_n	treatment	stain_type	coloc_stain_type	filename	image_name	CoM_x	CoM_y	mean_intensity	backgroun
0	0	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	306.85	306.78	485.9018	573.995
1	1	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	170.95	312.57	503.089	573.995
2	2	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	299.44	422.83	544.4831	573.995
3	3	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	154.15	476.33	524.3108	573.995
4	4	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	36.62	458.4	471.7593	573.995

Dropping nans, duplicates

In [270]:
```
# which cols have nans, how many?
print('Nan per col:')
print(df_coloc.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_coloc.duplicated().sum())

# looks like we've cleaned up pretty good so far, no need to drop anything.
```

```
Nan per col:
index                     0
rat_n                     0
treatment                 0
stain_type                0
coloc_stain_type          0
filename                  0
image_name                0
CoM_x                     0
CoM_y                     0
mean_intensity            0
background                0
snr                       0
coloc_roi_id_grouping     0
roi_id_grouping_counts    0
dtype: int64

Total n of duplicated rows:
0
```

Adjusting Mean-Background

In [272]:
```
def adjust_mmbg(df):
    '''
    '''
    # remove unnecesary index col
    if 'index' in df.columns:
        df = df.drop('index', axis=1)

    # add minimum observed intensity
    adjusted_mmbg = []
    for stain in df.stain_type.unique():
        # separate by stain
        df_stain = df.query(f'stain_type == "{stain}"').reset_index()

        # compute new mean-background col
        df_stain['mean-background'] = df_stain.loc[: ,'mean_intensity'].astype('f') - df_stain.loc
[:, 'background'].astype('f')

        # get min mean-background (some negative number)
        min_mmbg = df_stain['mean-background'].min()

        # add the absolute value of min to all other cells of the same stain
        df_stain['adjusted_mean-background'] = df_stain['mean-background'] + abs(min_mmbg)

        # toss back into list
        adjusted_mmbg.append(df_stain)

    df_adjusted = pd.concat(adjusted_mmbg)

    return df_adjusted

df_adjusted = adjust_mmbg(df_coloc)
print(df_adjusted.shape)
df_adjusted.head()
```

```
(18514, 16)
```

Out[272]:

	index	rat_n	treatment	stain_type	coloc_stain_type	filename	image_name	CoM_x	CoM_y	mean_intensity	backgroun
0	0	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	306.85	306.78	485.9018	573.995
1	1	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	170.95	312.57	503.089	573.995
2	2	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	299.44	422.83	544.4831	573.995
3	3	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	154.15	476.33	524.3108	573.995
4	4	KET-10-12	FR1_KET	Npas4	Npas4_coloc_w_WFA	KET-10-12_PFC_3.7_A_4.tif	KET-10-12_PFC_3.7_A	36.62	458.4	471.7593	573.995

Normalizing SNR, Counting Mean Cell Ns

Normalizing SNR

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

In [273]:
```
def normalize_intensity(df, norm_condition, col='mean-background'):
    '''
    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[col].astype('f').mean()

    df_norm = df.copy(deep=True)
    df_norm[f'norm_{col}'] = df[col].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][f'norm_{col}'].mean(), 5) == 1

    return df_norm

def prism_reorg(df, col='norm_mean-background'):
    '''
    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][col]
            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
```

Counting Mean Cell Ns

In [274]:
```
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)
```

Time to run it!

Normalize SNR, write to disk

In [276]:
```
group = 'KET-VR5'
for stain in df_adjusted.coloc_stain_type.unique():

    # split by stain
    df_stain = df_adjusted.query(f'coloc_stain_type == "{stain}"')

    # normalize to FR1_SAL
    df_norm = normalize_intensity(df_stain, norm_condition='FR1_SAL', col='adjusted_mean-backgroun
d')
    df_norm.to_csv(f'{group}_{stain}_NORM.csv')

    # reorganize into cols for prism
    df_prism = prism_reorg(df_norm, col='norm_adjusted_mean-background')
    df_prism.to_csv(f'{group}_{stain}_PRISM.csv')

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

```
quad_PV
```

Out[276]:

	FR1_KET	FR1_SAL	VR5_KET	VR5_SAL
0	1.478083	0.304439	2.061987	1.352451
1	0.546212	0.769373	2.150633	0.704056
2	1.506044	1.215628	1.241221	1.147529
3	1.049369	1.878789	1.275500	1.769058
4	1.348888	1.322184	0.707405	0.622042
5	1.757156	1.736948	0.537656	1.491575
6	1.008237	1.634087	1.013584	0.275652
7	1.323251	1.446699	0.907289	1.254814
8	0.960671	1.550145	0.288422	0.521585
9	2.098650	0.843862	0.416335	0.948158
10	0.994067	1.153271	1.050569	1.313911
11	0.511631	1.211506	1.170627	1.696379
12	0.469674	0.644638	2.386011	1.555641
13	0.519146	0.398056	0.455354	1.456941
14	1.177153	1.252357	0.995490	2.204717
15	0.722278	0.606160	2.117426	0.502566
16	0.985480	0.564660	2.070061	1.108755
17	1.693207	0.824340	1.094848	1.372309
18	1.399020	0.825511	2.459456	1.178585
19	1.194942	0.715921	1.318507	2.093364
20	1.081531	1.130234	1.858122	0.591180
21	1.708587	0.729050	2.385293	0.461702
22	1.441974	0.898688	1.874710	0.909288
23	0.567222	0.493724	1.569139	0.941738
24	1.041453	1.923315	1.548087	1.081923
25	0.851077	0.772085	2.109693	0.893340
26	0.527120	1.146393	1.743039	0.240296
27	0.807345	0.493258	0.418018	1.203486
28	0.919813	0.543802	0.511499	1.346587
29	0.653105	1.012938	0.556632	1.090450
30	0.679475	0.762404	2.280709	0.415805
31	0.787108	0.477269	3.154561	0.332632
32	0.724328	1.373971	1.474316	0.723325
33	1.505377	2.459485	1.742101	NaN
34	0.675881	0.305906	1.083603	NaN
35	0.374505	1.434091	0.802821	NaN
36	0.669511	0.217966	1.183620	NaN
37	0.493354	0.810036	0.540935	NaN
38	1.054022	1.004659	0.211901	NaN
39	1.347492	0.638693	0.625251	NaN
40	NaN	1.745882	2.234151	NaN
41	NaN	0.727575	1.266667	NaN
42	NaN	NaN	1.539985	NaN
43	NaN	NaN	0.369338	NaN

Count mean cell ns, write to disk

In [278]:
```
# count n of unique image names per subject
sid = 'rat_n'
iid = 'image_name'
cols = ['treatment', 'coloc_stain_type', sid, iid]
group = 'KET-VR5'

# wrapper fn calls
for stain in df_coloc.coloc_stain_type.unique():

    # split by stain type
    df_stain = df_coloc[df_coloc.coloc_stain_type == stain]

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

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

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

```
quad_WFA
```

Out[278]:

	rat_n	treatment	coloc_stain_type	cell_count_sums	image_n	mean_cell_n
0	KET-10-12	FR1_KET	quad_WFA	8	5	1.60
1	KET-9-1	FR1_KET	quad_WFA	4	4	1.00
2	PE-11-1	FR1_KET	quad_WFA	3	5	0.60
3	PE-11-2	FR1_KET	quad_WFA	4	5	0.80
4	PE-11-3	FR1_KET	quad_WFA	1	5	0.20
5	PE-12-1	FR1_KET	quad_WFA	3	5	0.60
6	PE-12-2	FR1_KET	quad_WFA	7	5	1.40
7	PE-12-7	FR1_KET	quad_WFA	10	5	2.00
8	KET-10-1	FR1_SAL	quad_WFA	4	5	0.80
9	KET-10-5	FR1_SAL	quad_WFA	4	5	0.80
10	KET-8-2	FR1_SAL	quad_WFA	4	5	0.80
11	KET-9-2	FR1_SAL	quad_WFA	12	5	2.40
12	KET-9-4	FR1_SAL	quad_WFA	6	5	1.20
13	KET-9-5	FR1_SAL	quad_WFA	5	5	1.00
14	KET-9-6	FR1_SAL	quad_WFA	7	5	1.40
15	KET-10-14	VR5_KET	quad_WFA	8	5	1.60
16	KET-8-7	VR5_KET	quad_WFA	7	4	1.75
17	PE-11-4	VR5_KET	quad_WFA	5	5	1.00
18	PE-11-5	VR5_KET	quad_WFA	5	5	1.00
19	PE-11-6	VR5_KET	quad_WFA	4	5	0.80
20	PE-11-7	VR5_KET	quad_WFA	3	5	0.60
21	PE-13-2	VR5_KET	quad_WFA	1	5	0.20
22	PE-13-3	VR5_KET	quad_WFA	5	5	1.00
23	PE-13-6	VR5_KET	quad_WFA	6	5	1.20
24	KET-10-2	VR5_SAL	quad_WFA	5	5	1.00
25	KET-10-3	VR5_SAL	quad_WFA	3	5	0.60
26	KET-10-4	VR5_SAL	quad_WFA	10	5	2.00
27	PE-13-1	VR5_SAL	quad_WFA	9	5	1.80
28	PE-13-11	VR5_SAL	quad_WFA	2	5	0.40
29	PE-13-9	VR5_SAL	quad_WFA	4	5	0.80

In [ ]: