tu14_DataWranglingII

March 2, 2023

1 Wraggling, Summarizing, and Plotting

1.1 Preliminaries

You don't need to import numpy to run pandas, but numpy comes in handy so often, we generally import it as well.

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

Let's re-use our function to load and clean up data from last time (but with a new name).

[2]:
```
def bcd_load_clean():
    bcd = pd.read_csv('./data/breast_cancer_data.csv')
    bcd['patient_id'] = bcd['patient_id'].astype('string')
    bcd['doctor_name'] = bcd['doctor_name'].str.split().str[1]
    bcd['bare_nuclei'] = bcd['bare_nuclei'].replace('?', '')
    bcd['bare_nuclei'] = pd.to_numeric(bcd['bare_nuclei'])

    return bcd
```

Load our data:

[3]:
```
bcd = bcd_load_clean()
```

[5]:
```
bcd.head()
```

[5]:
```
   patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
0     1000025              5.0                   1.0                      1
1     1002945              5.0                   4.0                      4
2     1015425              3.0                   1.0                      1
3     1016277              6.0                   8.0                      8
4     1017023              4.0                   1.0                      1

   marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
0                  1                    2          1.0              3.0
1                  5                    7         10.0              3.0
2                  1                    2          2.0              3.0
```

```
3                  1                    3          4.0              3.0
4                  3                    2          1.0              3.0

   normal_nucleoli  mitoses   class doctor_name
0              1.0        1  benign         Doe
1              2.0        1  benign       Smith
2              1.0        1  benign         Lee
3              7.0        1  benign       Smith
4              1.0        1  benign        Wong
```

Now, for convenience, let's make a smaller data set to play with. We'll do this by dropping some of the columns.

We can do this in one of two ways. We can either `.drop` the columns we don't want, or `.copy` the columns we do. Here's the first method:

[4]:
```
bcd2 = bcd.drop(labels = ['patient_id', 'cell_size_uniformity',␣
↪'cell_shape_uniformity',
                          'cell_shape_uniformity', 'marginal_adhesion',␣
↪'single_ep_cell_size',
                          'bare_nuclei', 'normal_nucleoli', 'mitoses'],
                axis = 1,             # we're selecting column - default is rows
                inplace = False)      # we could modify bcd itself with True
```

[6]:
```
bcd2.head()
```

[6]:
```
   clump_thickness  bland_chromatin   class doctor_name
0              5.0              3.0  benign         Doe
1              5.0              3.0  benign       Smith
2              3.0              3.0  benign         Lee
3              6.0              3.0  benign       Smith
4              4.0              3.0  benign        Wong
```

In the cell below, make the same new data frame using column indexing and the `.copy()` method.

[11]:
```
 # make new bcd2 using .copy()
bc3 = bcd2.copy()
```

[12]:
```
 # look at new bcd2
bc3
```

[12]:
```
     clump_thickness  bland_chromatin      class doctor_name
0                5.0              3.0     benign         Doe
1                5.0              3.0     benign       Smith
2                3.0              3.0     benign         Lee
```

```
3                6.0              3.0     benign       Smith
4                4.0              3.0     benign        Wong
..               …               …        …          …
694              3.0              1.0     benign         Lee
695              2.0              1.0     benign       Smith
696              5.0              8.0  malignant         Lee
697              4.0             10.0  malignant         Lee
698              4.0             10.0  malignant        Wong

[699 rows x 4 columns]
```

1.2 What might we want from this data set?

The main thing that comes to mind is whether any of the measures are related to the kind of tumor. To do this, we can

- group the data by the "class" column
- perform some operation, like computing the mean, separately for the groups.

We might also want to see if the doctors are behaving consistently with respect to one another.

1.2.1 The split-apply-combine workflow

Much of data wrangling can be thought of "split-apply-combine". This is where we

- *split* the data into groups
- do ("*apply*") some function or manipulation on a per-group basis
- *combine* the results back into a data frame, series, etc.

Happily, the "combine" step is often handled for you by the methods that do the "apply" step.

Splitting - the `groupby()` method Grouping the data is easy using the `groupby()` method. We just provide the name of a grouping variable. Since the main question at hand is how the measurements might relate to the type of tumor, Let's group by tumor "class".

[13]:
```
grpd = bcd2.groupby('class')
```

If we try to look at it:

[15]:
```
grpd.head()
```

[15]:
```
    clump_thickness  bland_chromatin      class doctor_name
0               5.0              3.0     benign         Doe
1               5.0              3.0     benign       Smith
2               3.0              3.0     benign         Lee
3               6.0              3.0     benign       Smith
4               4.0              3.0     benign        Wong
5               8.0              9.0  malignant       Smith
```

```
12              NaN              4.0  malignant       Smith
14              8.0              5.0  malignant         Doe
15              7.0              4.0  malignant         Lee
18             10.0              4.0  malignant       Smith
```

we see that the output of '.groupby() isn't a regular data frame, but rather a `DataFrameGroupBy` object. To interegate it, well need to use its methods or look at its attributes.

In the cell below, use the <TAB> key trick to browse the methods and properties that `grpd` has.

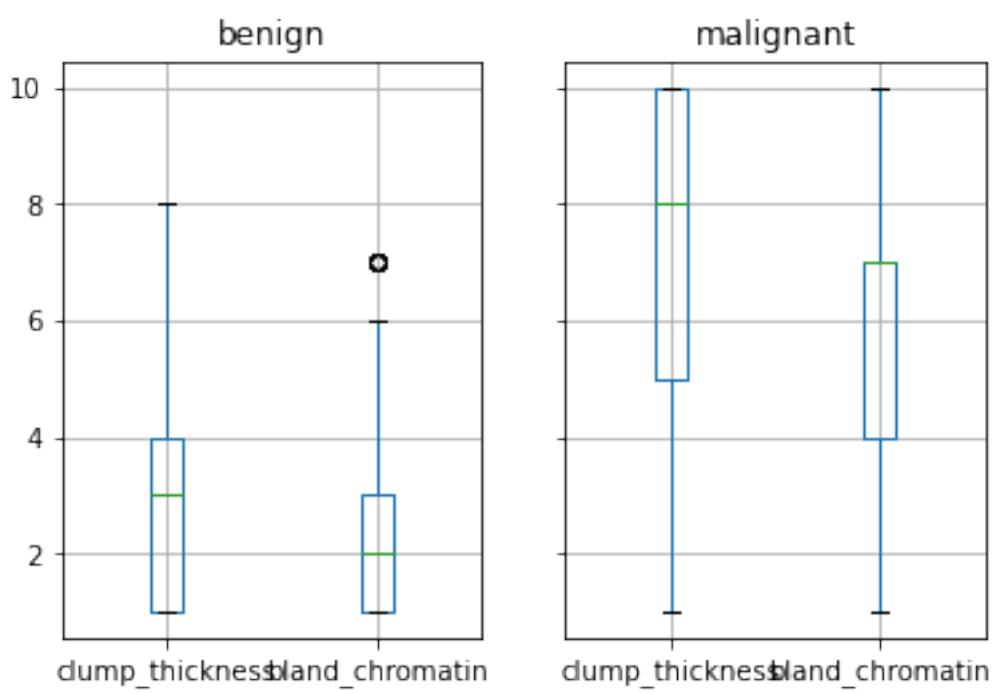
[17]:
```
grpd
```

[17]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fa22f7aa580>
```

One of the early methods on the list is `.boxplot` - see what that does in the cell below!

[22]:
```
grpd.boxplot()
```

[22]:
```
benign            AxesSubplot(0.1,0.15;0.363636x0.75)
malignant    AxesSubplot(0.536364,0.15;0.363636x0.75)
dtype: object
```



While not the prettiest plot in the world, it does give us a hint that both of these variables might be related to tumor size.

(Make sure you remember or remind yourself what a box shows you.)

Now let's see if we can `.describe` the grouped data using the cell below.

[23]:
```
# some useful summary numbers
grpd.describe()
```

[23]:
```
          clump_thickness                                                  \
                    count      mean       std  min  25%  50%   75%   max
class
benign              458.0  2.956332  1.674318  1.0  1.0  3.0   4.0   8.0
malignant           240.0  7.204167  2.429763  1.0  5.0  8.0  10.0  10.0

          bland_chromatin
                    count      mean       std  min  25%  50%  75%   max
class
benign              455.0  2.105495  1.081417  1.0  1.0  2.0  3.0   7.0
malignant           240.0  5.991667  2.270406  1.0  4.0  7.0  7.0  10.0
```

What is the approximate mean difference between the groups for each of the two measures?

- `The mean of benign is 2.96, but the mean of malignant is 7.20. The mean differences between the two is 4.24, which could be a lot when considering the mean of the benign is only 2.96.`

Do a very rough guesstimate of Student's t for the clump thickness (e.g., just use the larger `std` and smaller `countn`). You can use the cell below as a calculator if you like.

[30]:
```
import math
```

[27]:
```
grpd.std()
```

[27]:
```
           clump_thickness  bland_chromatin
class
benign            1.674318         1.081417
malignant         2.429763         2.270406
```

[31]:
```
(7.20 - 2.96)/(2.43/math.sqrt(240))
```

[31]:
```
27.031192408097855
```

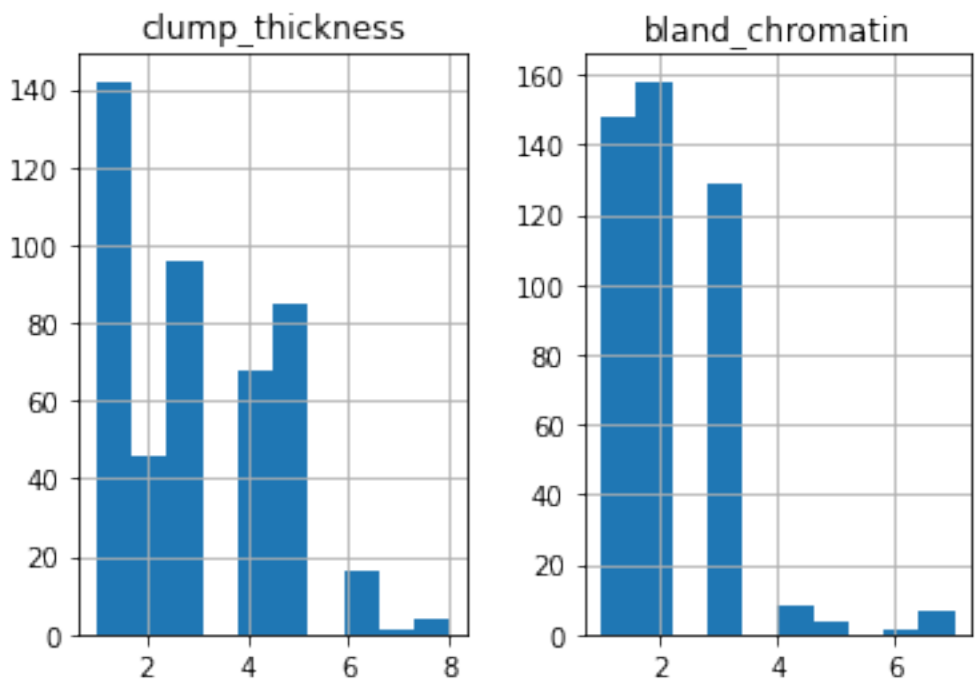
What does that tell you?

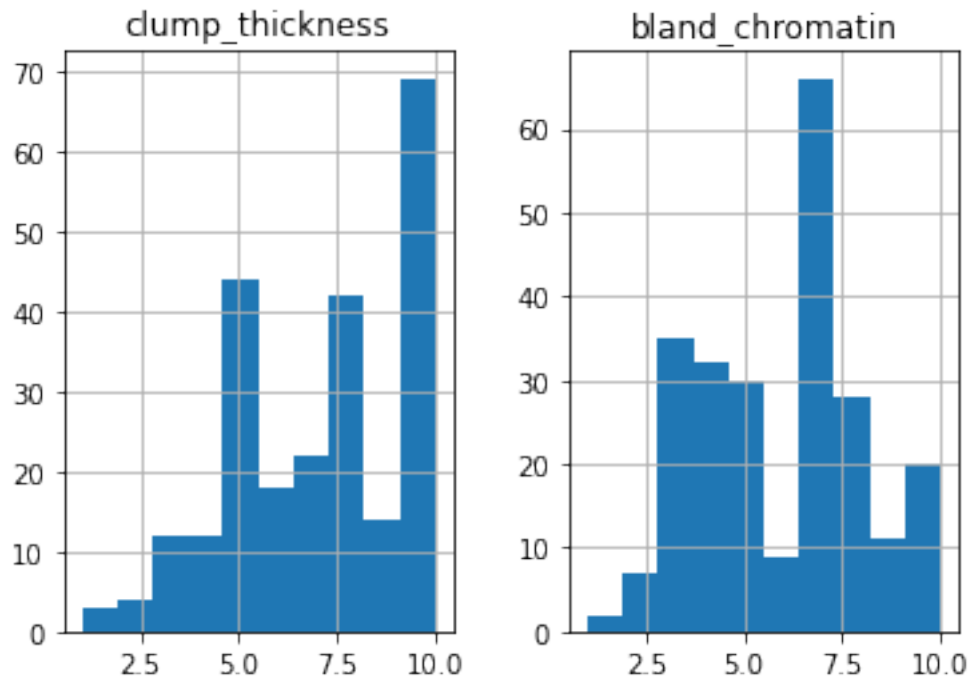
- The t-score is quite large. So, I am assuming that there is a significant diffrence between the mean of clump thickness and bland chromatin.

Now let's make a histogram of grouped data using the appropriate method.

[32]:
```
grpd.hist()
```

[32]:
```
class
benign       [[AxesSubplot(0.125,0.125;0.336957x0.755), Axe…
malignant    [[AxesSubplot(0.125,0.125;0.336957x0.755), Axe…
dtype: object
```





Again, not the prettiest plot in the world. By comparison with the boxplot above, we can see that the top row of these histograms correspond to the benign tumors. We can also see that, consistent with the box plots, there is quite a bit overlap in the data values across groups. Is this consistent with your calculation of t?

- `Yes, it is consistent with my calculated t because it suggests the difference between the two.`

We can also get the data for a specific group out of the grouped object. This will return a regular data frame the same width as the original, but only containing the requested group's data.

[33]:
```
grpd.get_group('benign')
```

[33]:
```
     clump_thickness  bland_chromatin   class doctor_name
0                5.0              3.0  benign         Doe
1                5.0              3.0  benign       Smith
2                3.0              3.0  benign         Lee
3                6.0              3.0  benign       Smith
4                4.0              3.0  benign        Wong
..               …                …      …          …
690              1.0              1.0  benign         Doe
692              3.0              1.0  benign        Wong
693              3.0              2.0  benign         Lee
694              3.0              1.0  benign         Lee
```

```
695              2.0              1.0  benign       Smith

[458 rows x 4 columns]
```

In the cell below, confirm that the returned object is indeed a pandas `DataFrame`.

[34]:
```
grpd
```

[34]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fa22f7aa580>
```

Applying - doing things to the data within groups Once we have grouped data, we can easily caluculate things per group. Using the <TAB> trick, we can see that these objects produced by `groupby()` have methods for all the common statistical summaries.

Simple calculations We can compute the mean for each measure *by group*

[35]:
```
my_means = grpd.mean(numeric_only = True)
my_means
```

[35]:
```
           clump_thickness  bland_chromatin
class
benign            2.956332         2.105495
malignant         7.204167         5.991667
```

In the cells below, compute and show the

standard deviations:

[39]:
```
my_sds =  grpd.std()
my_sds
```

[39]:
```
           clump_thickness  bland_chromatin
class
benign            1.674318         1.081417
malignant         2.429763         2.270406
```

and the counts

[41]:
```
my_count = grpd.count()
my_count
```

[41]:
```
           clump_thickness  bland_chromatin  doctor_name
class
benign                 458              455          458
```

```
malignant              240              240          241
```

We can easily do simple maths on data frames of a compatible size. Here's a comparison of how many z-scores above zero each of the means are:

[42]:
```
my_zeds = my_means/my_sds
my_zeds
```

[42]:
```
           clump_thickness  bland_chromatin
class
benign            1.765693         1.946977
malignant         2.964967         2.639029
```

We can look at the difference between the z-scores for each measure using the `diff()` method, which takes the first difference down the rows.

[43]:
```
my_zeds.diff()
```

[43]:
```
           clump_thickness  bland_chromatin
class
benign                 NaN              NaN
malignant         1.199274         0.692052
```

This difference in z-scores – how far apart two means are in terms of the standard deviation of the data – is roughly what statisticians call "effect size".

Why do the NaNs appear in the first row?

- `My hypothesis that the difference between clump and bland in benign class are not large enough to compute.`

In the cell below, extract just the row with the numbers from `my_zeds` into a new series. (hint: using `.loc` is probably easiest)

[47]:
```
malig_z = my_zeds.loc['malignant']
malig_z
```

[47]:
```
clump_thickness    2.964967
bland_chromatin    2.639029
Name: malignant, dtype: float64
```

We could also compute the difference by using `.loc[]` row indexing and simple maths.

[48]:
```
my_zeds.loc['malignant'] - my_zeds.loc['benign']
```

[48]:
```
clump_thickness    1.199274
bland_chromatin    0.692052
dtype: float64
```

Mulitiple caclulations with `agg()` We can do multiple calculation at once by placing function names inside the `agg()`) or `aggregate()`) methods (they are synonyms). Here's where importing numpy comes in handy.

[53]:
```
grpd.agg([np.mean, np.std])
```

```
/var/folders/yq/3rc62cqs3nn_n_c8mm6k56jw0000gn/T/ipykernel_781/1757925766.py:1:
FutureWarning: ['doctor_name'] did not aggregate successfully. If any error is
raised this will raise in a future version of pandas. Drop these columns/ops to
avoid this warning.
  grpd.agg([np.mean, np.std])
```

[53]:

	clump_thickness		bland_chromatin	
	mean	std	mean	std
class				
benign	2.956332	1.674318	2.105495	1.081417
malignant	7.204167	2.429763	5.991667	2.270406

That worked, but pandas still complained to us because `grpd` has the doctors' names in it, and we obviously can't compute the means and standard deviations of those!

In the cell below, repeat the above calculation without triggering the warning.

[79]:
```
grpd2 = grpd[['clump_thickness','bland_chromatin']]
grpd2.agg([np.mean, np.std])
```

[79]:

	clump_thickness		bland_chromatin	
	mean	std	mean	std
class				
benign	2.956332	1.674318	2.105495	1.081417
malignant	7.204167	2.429763	5.991667	2.270406

We can use the pandas versions of functions by placing them in quotes. This is handy because, for example, pandas has a `count` and numpy doesn't.

[80]:
```
grpd[['clump_thickness', 'bland_chromatin']].agg([np.mean, 'std', 'count'])
```

[80]:

	clump_thickness			bland_chromatin		
	mean	std	count	mean	std	count
class						
benign	2.956332	1.674318	458	2.105495	1.081417	455

```
malignant        7.204167  2.429763   240        5.991667  2.270406   240
```

MultiIndexing - getting at our summary data Let's store our summary table little summary table above in its own data frame. This is going to complete our *split-apply-combine* by creating and naming a `DataFrame` object.

[81]:
```
my_summary = grpd[['clump_thickness', 'bland_chromatin']].agg([np.mean, 'std',␣
↪'count'])
my_summary
```

[81]:
```
          clump_thickness                 bland_chromatin
                     mean       std count            mean       std count
class
benign           2.956332  1.674318   458        2.105495  1.081417   455
malignant        7.204167  2.429763   240        5.991667  2.270406   240
```

Notice that this data frame has hierachical column labels. In other words, there is a "clump_thickness" meta-column that contains three colums of its own, and that these subcolumns have the same names as those in the other meta-column. Thus saying "look at the mean column" would be ambiguous because the meta-column wasn't specified.

In pandas, this is known at "multiIndexing".

Getting a meta-column is easy - it's just like getting a regular column from a data frame.

[82]:
```
my_summary['clump_thickness']
```

[82]:
```
               mean       std  count
class
benign     2.956332  1.674318    458
malignant  7.204167  2.429763    240
```

To get a subcolumn, we can index the meta-column, and then index the subcolumn from that.

[83]:
```
my_summary['clump_thickness']['mean']
```

[83]:
```
class
benign       2.956332
malignant    7.204167
Name: mean, dtype: float64
```

If this looks confusing, consider the same thing broken up into two steps:

[84]:
```
meta_c = my_summary['clump_thickness']
meta_c
```

[84]:
```
               mean       std  count
class
benign     2.956332  1.674318    458
```

```
malignant  7.204167  2.429763    240
```

[85]:
```
meta_c['mean']
```

[85]:
```
class
benign       2.956332
malignant    7.204167
Name: mean, dtype: float64
```

In the cell below, extract the mean and std of "bland_chromatin" in one go.

[91]:
```
my_summary['bland_chromatin'][['mean','std']]
```

[91]:
```
               mean       std
class
benign     2.105495  1.081417
malignant  5.991667  2.270406
```

If we want values from a row, we need to get a bit more fancy and use `.loc`.

We can get a whole row using `df.loc[row_index(s)]`

[92]:
```
my_summary.loc['benign']
```

[92]:
```
clump_thickness  mean       2.956332
                 std        1.674318
                 count    458.000000
bland_chromatin  mean       2.105495
                 std        1.081417
                 count    455.000000
Name: benign, dtype: float64
```

(note: that this gave us a hierarchical index!)

We can get a row and and a particular column with `df.loc[row_index(s), (metacolumn_index, subcolumn_index)]`

[93]:
```
my_summary.loc['benign', ('clump_thickness', 'mean')]
```

[93]:
```
2.9563318777292578
```

We can also get bigger slices of the data with the colon `:` operator:

[94]:
```
my_summary.loc['benign', ('clump_thickness', 'mean'):('bland_chromatin', 'std')]
```

[94]:
```
clump_thickness  mean       2.956332
                 std        1.674318
                 count    458.000000
bland_chromatin  mean       2.105495
                 std        1.081417
Name: benign, dtype: float64
```

In the cell below, extract the mean and std of the bland chromatin meta-column.

[101]:
```
my_summary.loc['benign', ('bland_chromatin', 'mean'):('bland_chromatin', 'std')]
```

[101]:
```
bland_chromatin  mean    2.105495
                 std     1.081417
Name: benign, dtype: float64
```

Simple caculations with pivot tables Pivot tables are summary data with the levels of one variable running down the row names (the index), the levels of another running across the column names, and values populating the interior. This should be made concrete by making one with the `pivot_table()` method:

[102]:
```
bcd2.pivot_table(index = 'doctor_name', columns = 'class', values =␣
 ↪'bland_chromatin')
```

[102]:
```
class          benign  malignant
doctor_name
Doe          2.000000   5.456140
Lee          2.067227   6.150000
Smith        1.980392   6.459459
Wong         2.388889   5.714286
```

By default, `pivot_table()` computes the group (row x column) means, but we can compute any of the standard summary statistics we wish. We just specify it using the `aggfunc` argument:

[103]:
```
bcd2.pivot_table(index = 'doctor_name',
                 columns = 'class',
                 values = 'bland_chromatin',
                 aggfunc = 'std')
```

[103]:
```
class          benign  malignant
doctor_name
Doe          1.003992   2.260453
Lee          1.014564   2.121920
Smith        0.943769   2.330202
Wong         1.303004   2.263846
```

These are called "pivot tables" because their implementation makes it easy to pivot our view of the data summary.

In the cell below, "pivot" our view of the means so we have "class" down the rows, doctor name across the columns, and the means of clump thickness inside the table.

[104]:
```
bcd2.pivot_table(index = 'class',
                 columns = 'doctor_name',
                 values = 'clump_thickness',
                 aggfunc = 'mean')
```

[104]:
```
doctor_name       Doe       Lee     Smith      Wong
class
benign       2.637795  2.983471  3.098039  3.166667
malignant    7.586207  6.600000  7.356164  7.265306
```

Because clumb thickness and bland chromatin only have the values 1 to 10, we could use either one as a grouping variable in a pivot table:

[105]:
```
bcd2.pivot_table(index = 'clump_thickness', columns = 'class', values =␣
 ↪'bland_chromatin')
```

[105]:
```
class              benign  malignant
clump_thickness
1.0              1.978723   5.666667
2.0              2.090909   5.000000
3.0              2.145833   5.500000
4.0              2.117647   7.916667
5.0              2.011765   5.818182
6.0              3.062500   6.666667
7.0              2.000000   5.818182
8.0              3.750000   5.761905
9.0                   NaN   5.142857
10.0                  NaN   6.147059
```

Notice the NaNs – no benign tumors have a thickness of 9 or 10.

1.3 Grouped plotting

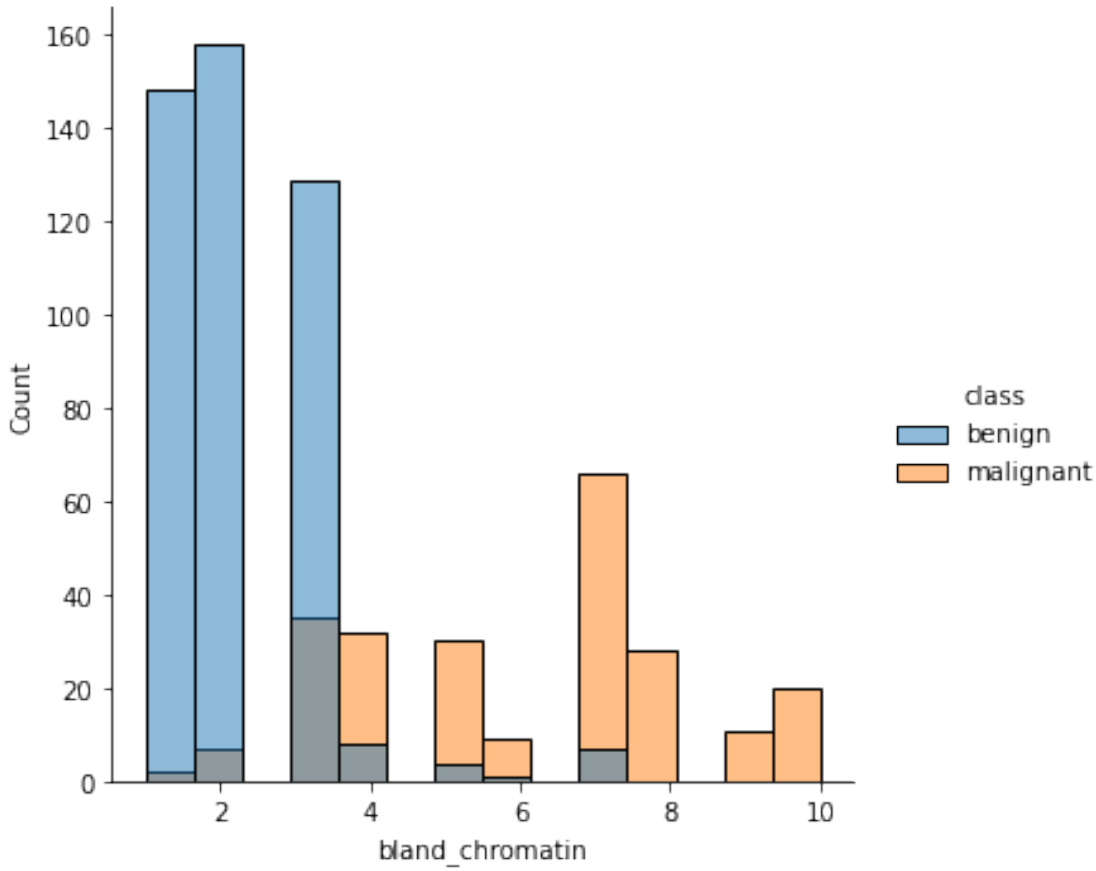
As we have already seen, the seaborn package can take care of grouping for us – we just need to assign a grouping variable to color ('hue'), style, etc. And we already know how to do all this!

[106]:
```
import seaborn as sns
```

Make a distribution plot (like a histogram) of bland chromatin values grouped by tumor type.

[116]:
```
sns.displot(data = bcd2, x = 'bland_chromatin', hue = 'class')
```

[116]: <seaborn.axisgrid.FacetGrid at 0x7fa21c6e0c40>



Make a joint plot grouped by tumor type:

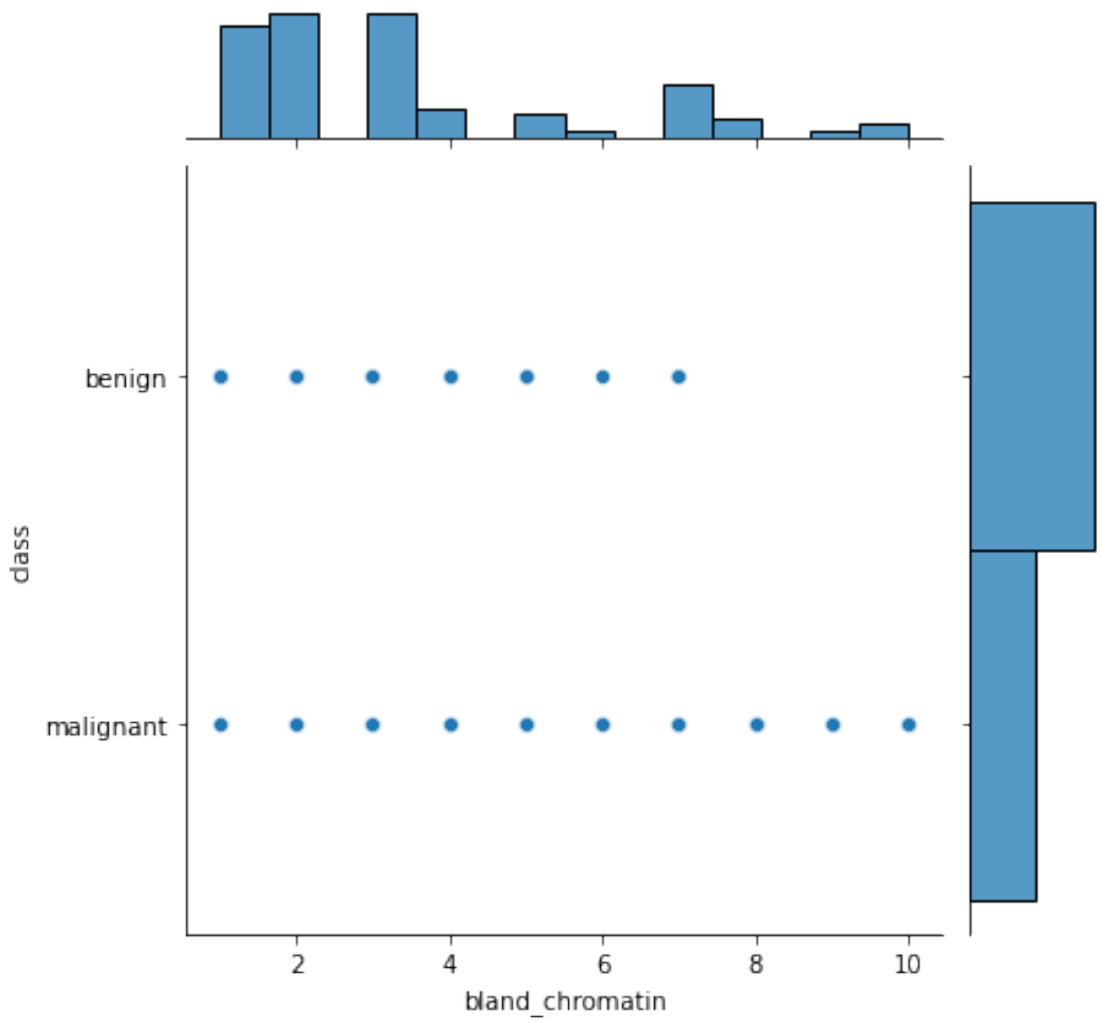
[122]:
```
bcd2.head()
```

[122]:

	clump_thickness	bland_chromatin	class	doctor_name
0	5.0	3.0	benign	Doe
1	5.0	3.0	benign	Smith
2	3.0	3.0	benign	Lee
3	6.0	3.0	benign	Smith
4	4.0	3.0	benign	Wong

[125]:
```
sns.jointplot(data = bcd2, x = 'bland_chromatin', y = 'class')
```

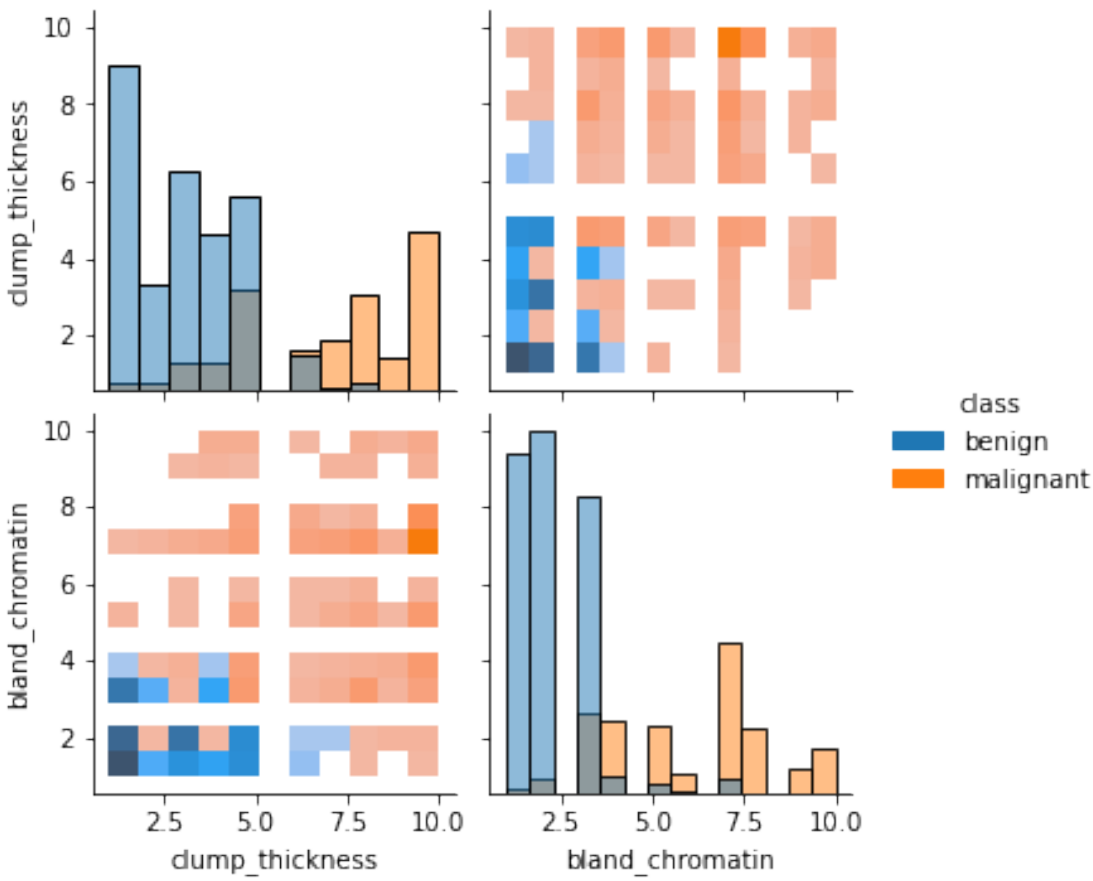
[125]: <seaborn.axisgrid.JointGrid at 0x7fa21eab0460>



Make a pairplot of our two measurement variables grouped by tumor type.

[142]:
```
sns.pairplot(data = bcd2, hue = 'class', kind = 'hist')
```

[142]: <seaborn.axisgrid.PairGrid at 0x7fa2230bc4f0>

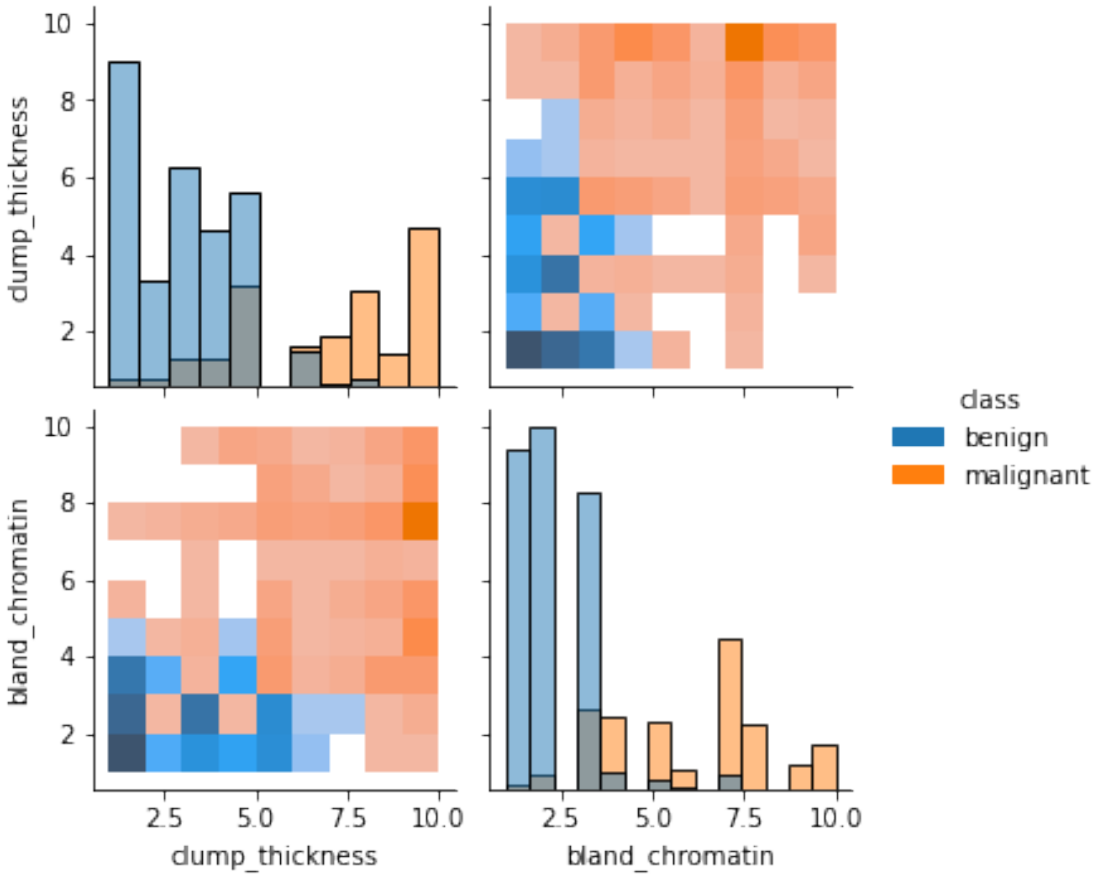


The plot above is okay except for the white space in the diagonal plots that aren't actually missing data. To fix this, we need to set our binwidth to 1, so that it matches the data (which are integers from 1 to 10).

Use the cell below to remake the plot with a binwidth of 1 for the diagonal plot. Hint: use the `plot_kws` argument to adjust this.

[144]:
```
sns.pairplot(data = bcd2, hue = 'class', kind = 'hist',plot_kws=dict(binwidth =␣
↪1))
```

[144]: <seaborn.axisgrid.PairGrid at 0x7fa223ca6820>



1.4 Your conclusions

In the cell below, briefly state your conclusions from our analysis above. Are either or both of the measurements related to tumor type?

- `Yes, at least for bland chromatin display the correlation to stay lower. Meanwhile, the clump chromatin tend to have a higher value.`

1.5 Summary

In this tutorial, we learned to analyze data by group:

- the split-apply-combine concept
- grouping using `groupby()`
- doing simple grouped calculations
- doing multiple calculations with `agg()`
- multiIndexing
- simple summaries with pivot tables