CAP 5768: Introduction to Data Science

Exploratory Data Analysis

What is exploratory data analysis?

Exploratory data analysis is the act of visualizing and transforming data to explore it in a systematic manner.

The process is an iterative cycle in which one

1. Generates questions about their data.
2. Searches for answers by visualizing, transforming, and modeling their data.
3. Uses what they've learned to refine their questions and/or generate novel questions.

Data cleaning is also an important component of exploratory data analysis.

Visualizing distributions

We have already seen several examples of ways to visualize distributions.

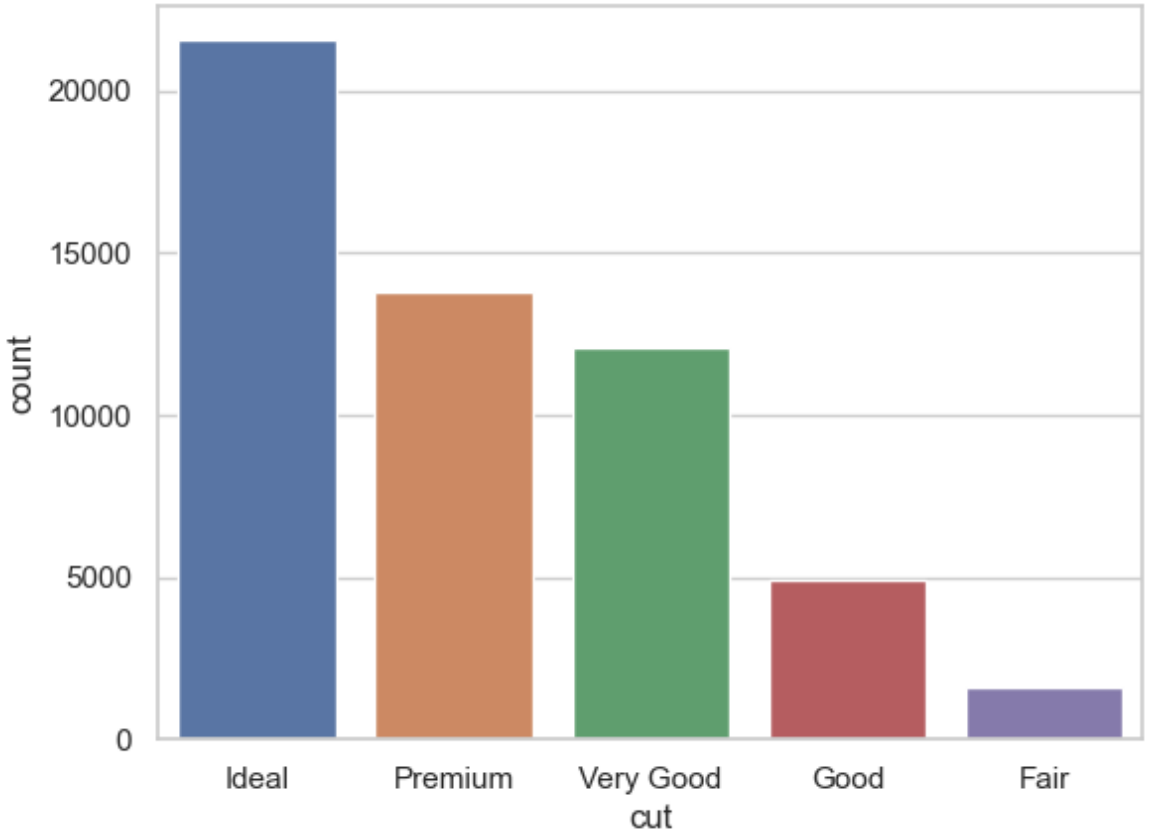
How one visualizes distributions will depend on whether a particular feature is categorical or continuous (quantitative).

Categorical features (variables) can take on a small set of possible values.

Continuous features (variables) can take on an infinite set of ordered values

Visualizing categorical distributions with bar plots

We have already seen that one way to visualize categorical features is with bar plots, which provide a summary of variation across categories, but not within categories.




```
sns.countplot(data=diamonds,
              x='cut',
              order = diamonds['cut'].value_counts().index)
```

Plotted values can be computed without visualization

We can compute the plotted values without visualizing by using some of the techniques we have learned about data transformation.

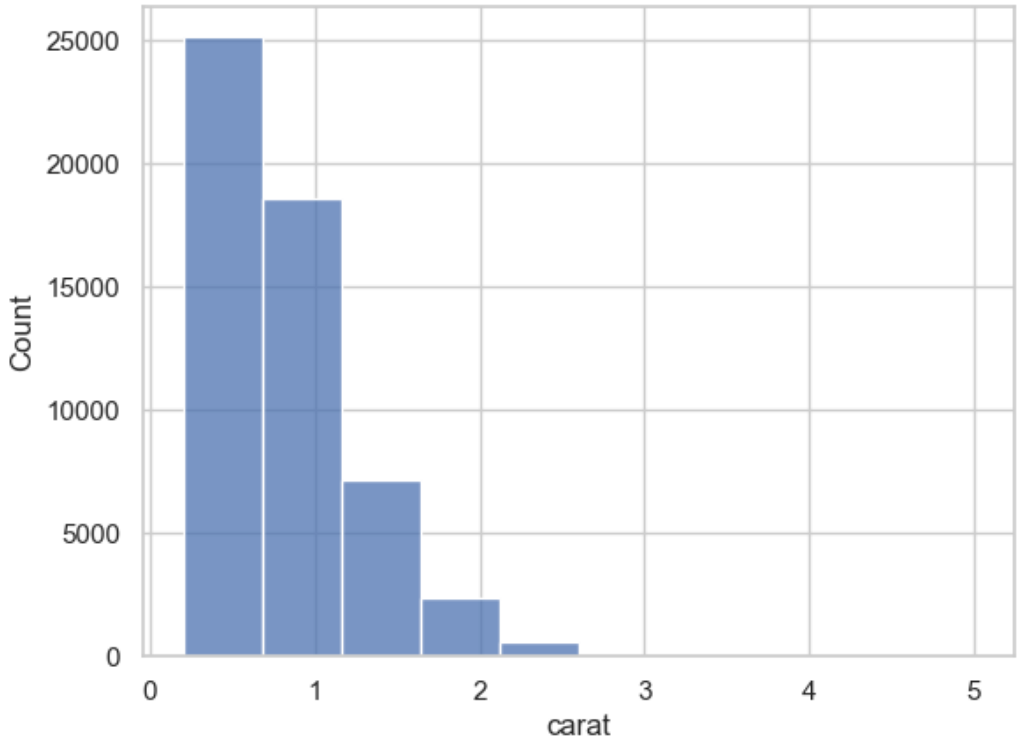
```
diamonds['cut'].value_counts()
```

```
Out[12]: Ideal        21551
         Premium      13791
         Very Good    12082
         Good          4906
         Fair          1610
         Name: cut, dtype: int64
```

Visualizing continuous distributions with histograms

We have also seen that a way to visualize continuous features is with histograms, which are generally good at visualizing distributions of all observations on a single feature.

```
sns.histplot(diamonds['carat'], bins=12)
```



Plotted values can be computed without visualization

We can compute the plotted values without visualizing by using some of the techniques we have learned about data transformation, and with a new function **`pd.cut()`**`()` that behaves the same as the **`bins`** argument of **`sns.histplot`**`()`, that bins values into discrete intervals.

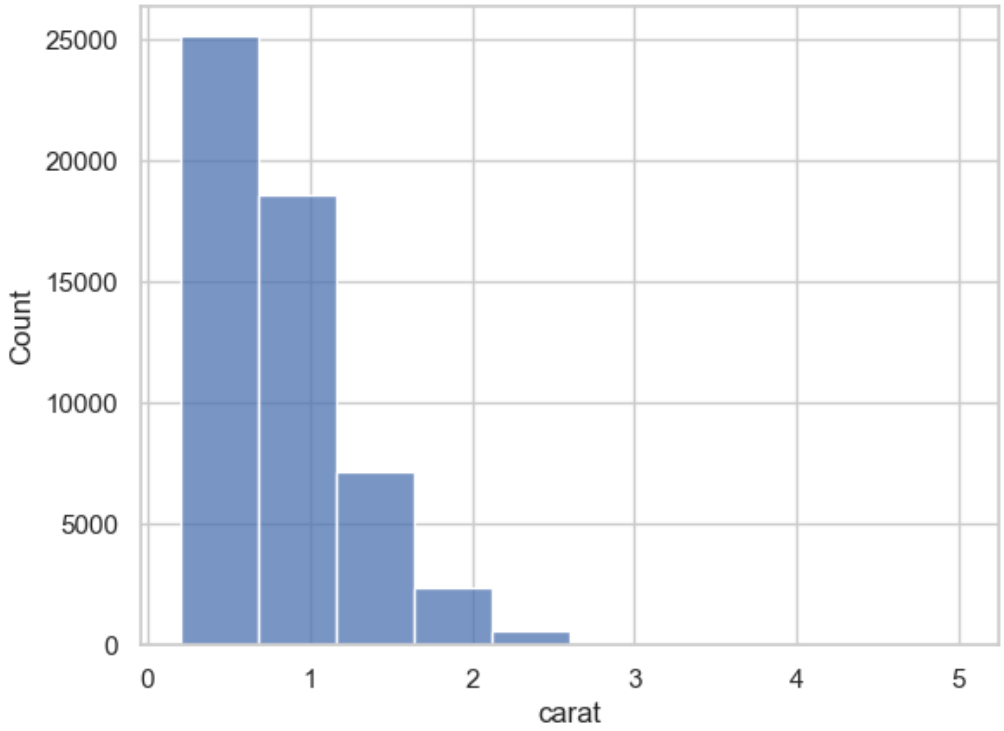
```
diamonds['carat_bin'] = pd.cut(diamonds['carat'], bins=12)
diamonds['carat_bin'].value_counts().sort_index()
```

```
Out[33]: (0.195, 0.601]    24448
         (0.601, 1.002]    11990
         (1.002, 1.402]    11093
         (1.402, 1.803]     4135
         (1.803, 2.204]     1812
         (2.204, 2.605]      395
         (2.605, 3.006]       35
         (3.006, 3.407]       22
         (3.407, 3.808]        4
         (3.808, 4.208]        4
         (4.208, 4.609]        1
         (4.609, 5.01]         1
         Name: carat_bin, dtype: int64
```

Exploring sub-regions of data and altering scale

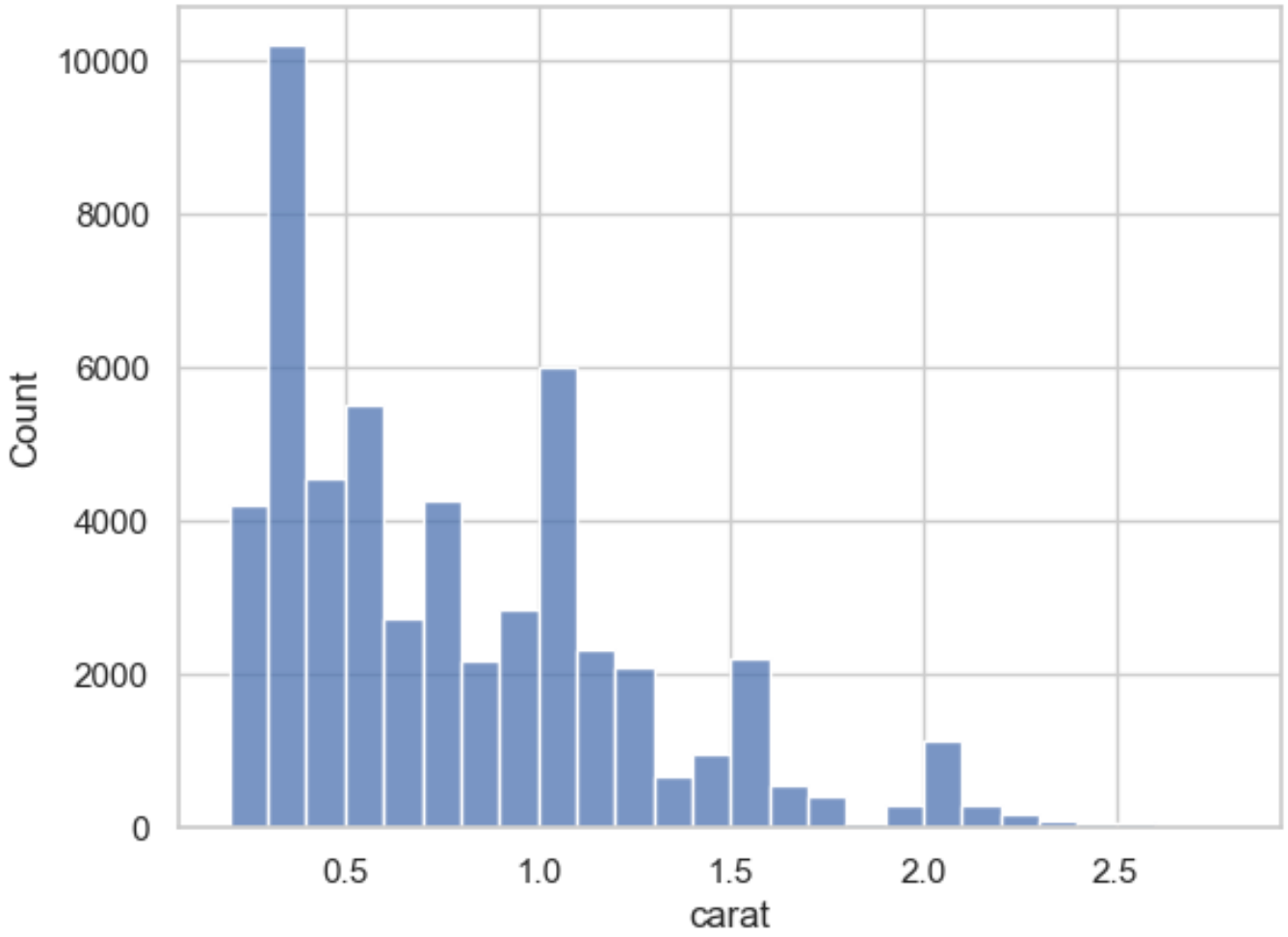
In the histogram below, we have a bin width that may be too large to identify any fine-scale patterns within the data.

Moreover, there are few observations greater than three carats, and so it may be helpful to filter the data and explore only a subset of observations on smaller diamonds.



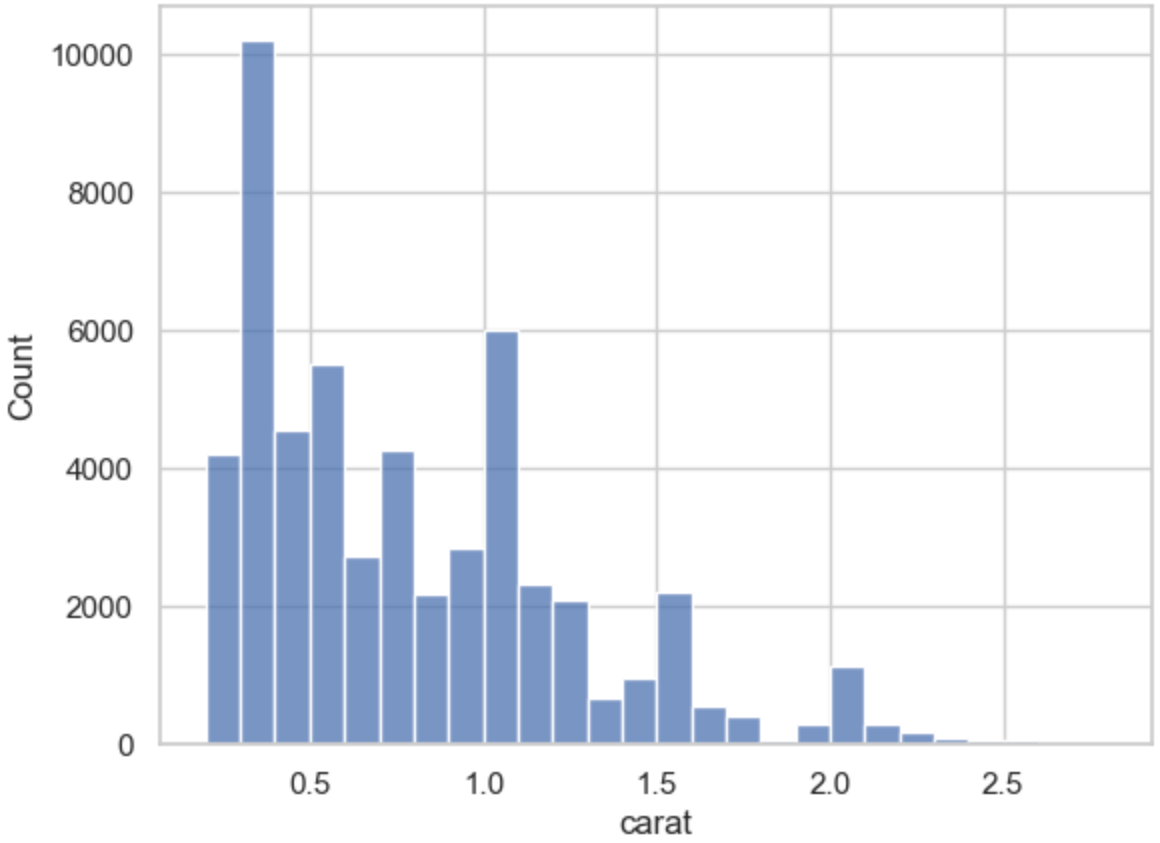
Filtering for small diamonds and focusing on small scale

```
daiamonds_f = diamonds[diamonds['carat']<3]
sns.histplot(daiamonds_f['carat'], binwidth=0.1)
```



Multiple modes (subgroups) of smaller diamonds

By zooming into the parameter space for which the majority of the observations reside (smaller diamonds weighing less than three carats) and by choosing a finer scale binning, we can see that there are multiple modes in the distribution, potentially corresponding to subgroups of diamonds



Plotting results of multiple different histograms

Before exploring this pattern in the data further, we briefly discuss ways of overlapping histograms.

Suppose we wished to make a separate histogram for each quality of diamond (**`cut`**) and depict each histogram with a different color.

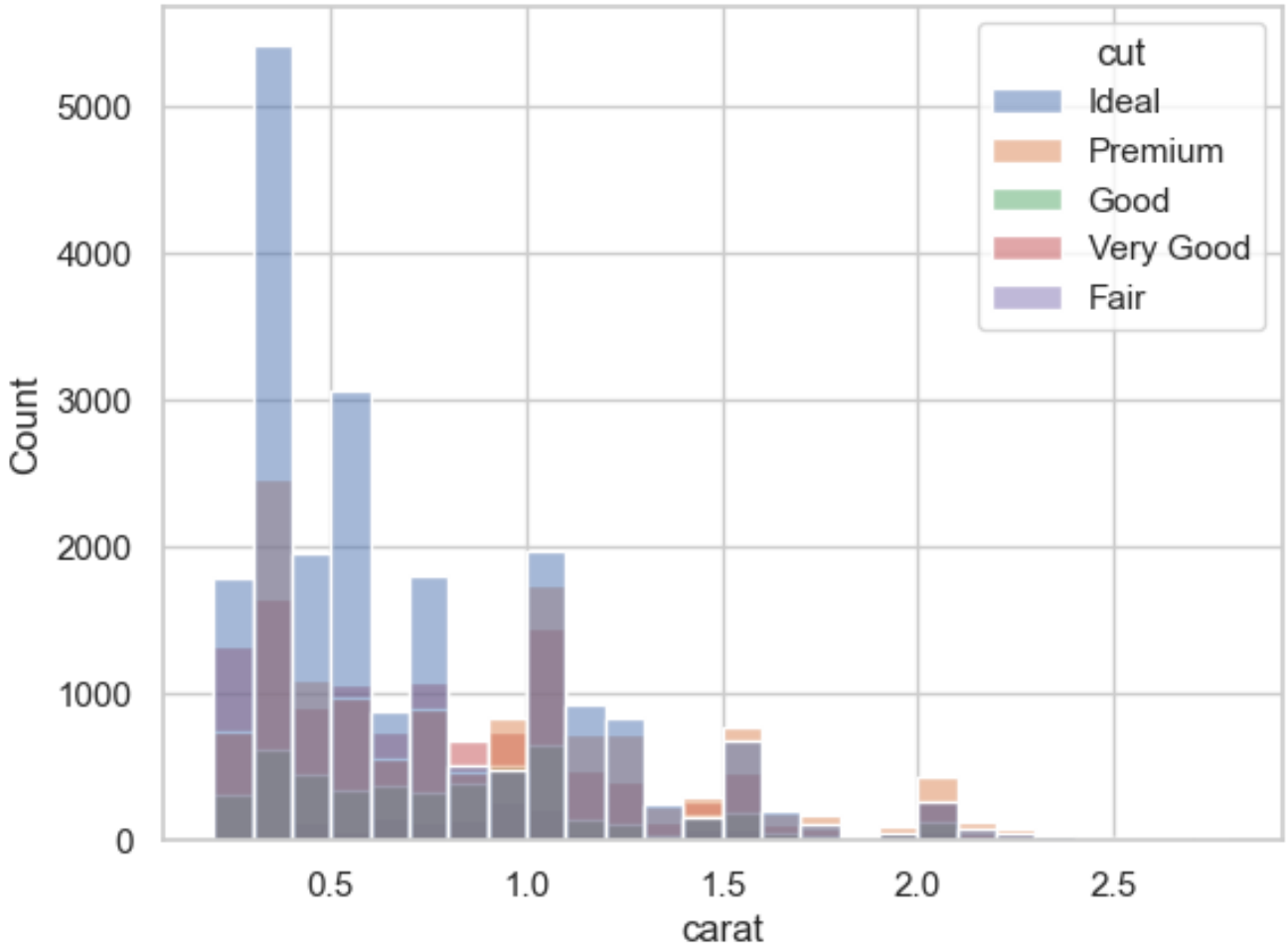
Naturally, we would gravitate to using the **`sns.histplot()`** function, and would potentially use the code

```
sns.histplot(daiamonds_f['carat'], binwidth=0.1)
```

However, the result of such code may not be what we envisioned.

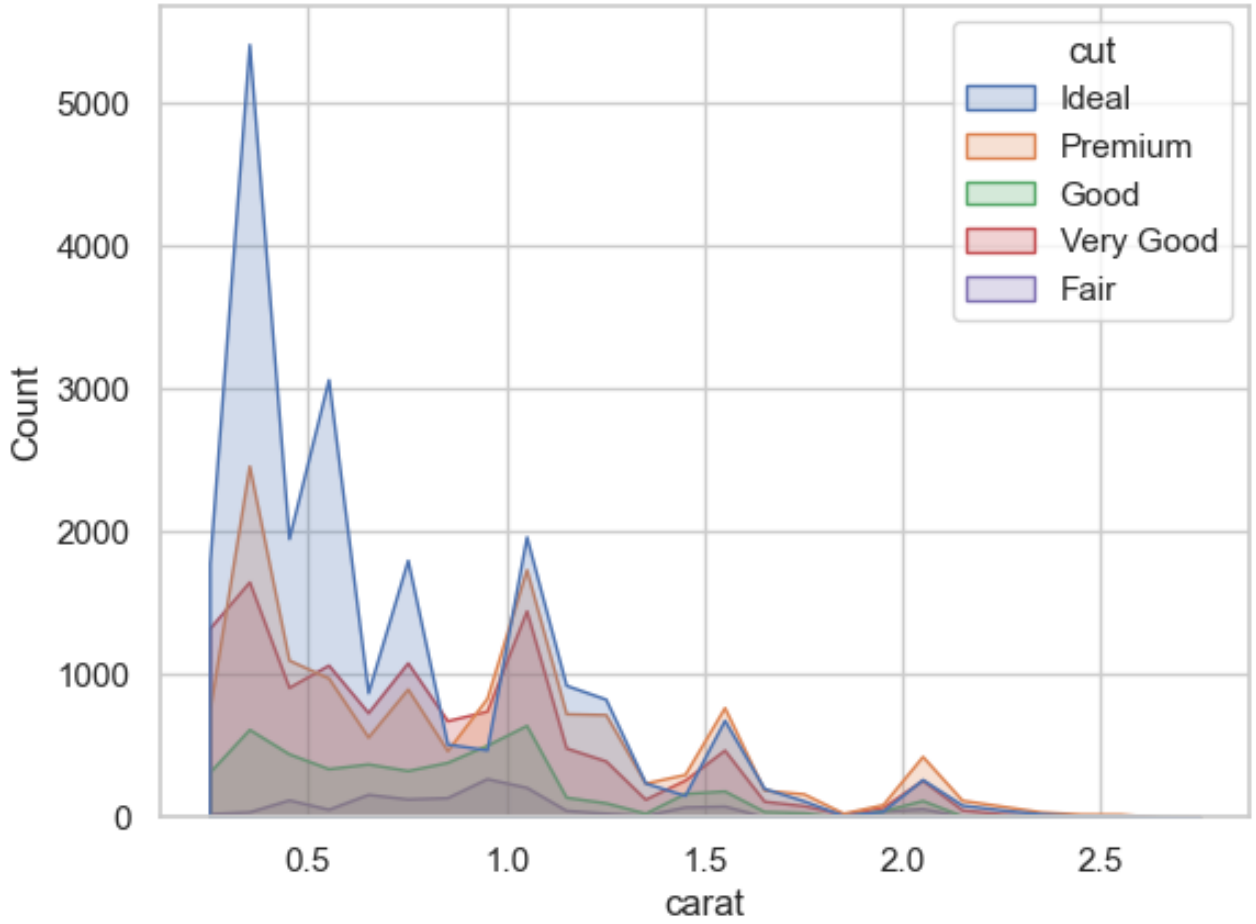
Using different colors as histograms not what hoped for

```
sns.histplot(data = daiamonds_f, x='carat',
             binwidth=0.1,
             hue='cut')
```



Use `element='poly'` to visualize multiple histograms

```
sns.histplot(data=daiamonds_f, x='carat',
             hue='cut',
             binwidth=0.1,
             element="poly")
```



Typical values

In bar charts and histograms, tall bars display common (or typical) values of a particular feature, with shorter bars being less common (or more atypical).

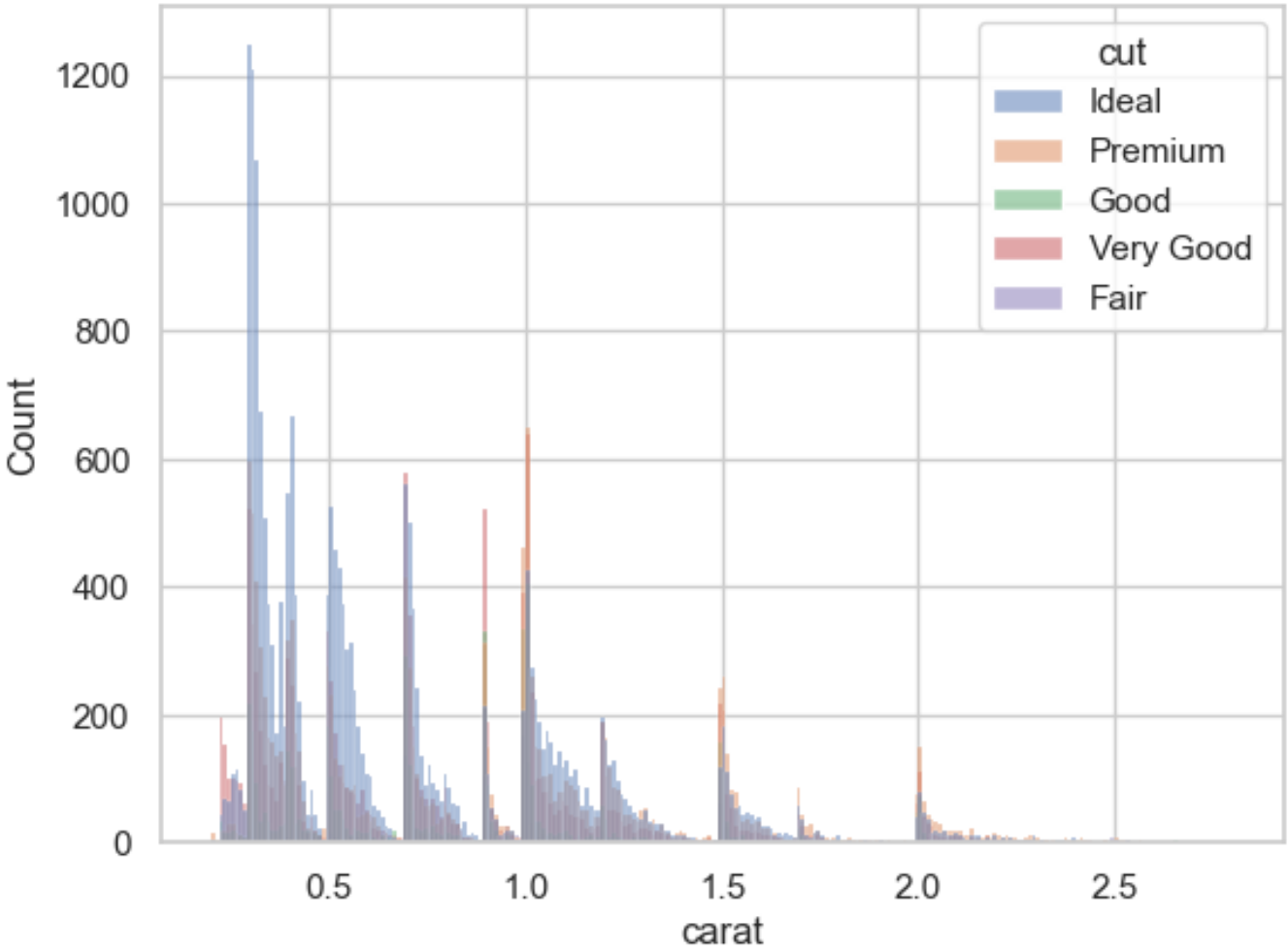
Regions without bars indicate that there were no observations for that particular value in the dataset.

These patterns can be turned into questions:

- ***Which values are the most common, and why?***
- ***Which value are rare, and why? Does this match our expectations?***
- ***Do we observe any unusual patterns? What might explain them?***

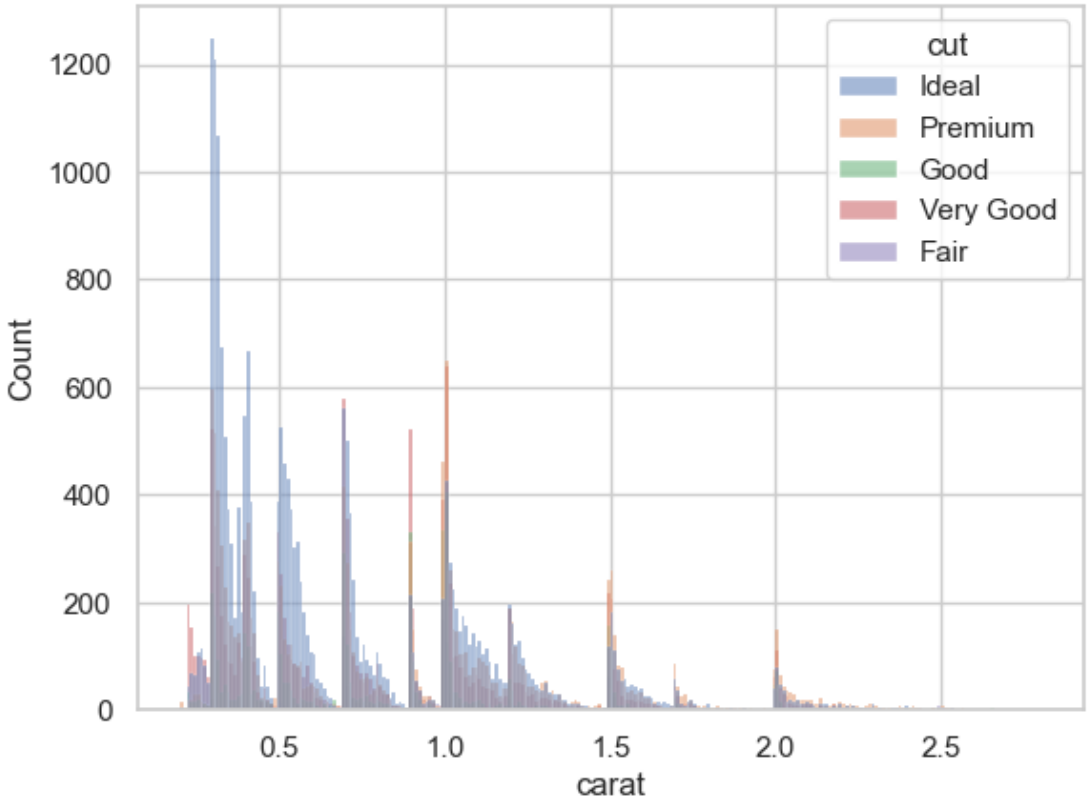
Consider diamond weight at finer scale

```
sns.histplot(data=daiamonds_f, x='carat',
             hue='cut',
             binwidth=0.01)
```



Questions to ask about this data

- Why are there more diamonds at whole carats and common fractions of carats?
- Why are there more diamonds slightly to the right of each peak than there are slightly to the left of each peak?
- Why are there no diamonds larger than three carats?



Clusters of similar values

Clusters of similar values suggest that subgroups exist in the data.

To understand the subgroups, ask:

- How are the observations within each cluster similar to each other?
- How are the observations in separate clusters different from each other?
- How can you explain or describe the clusters?
- Why might the appearance of clusters be misleading?

Unusual values

Outliers are observations that are unusual, or that do not seem to fit a particular pattern.

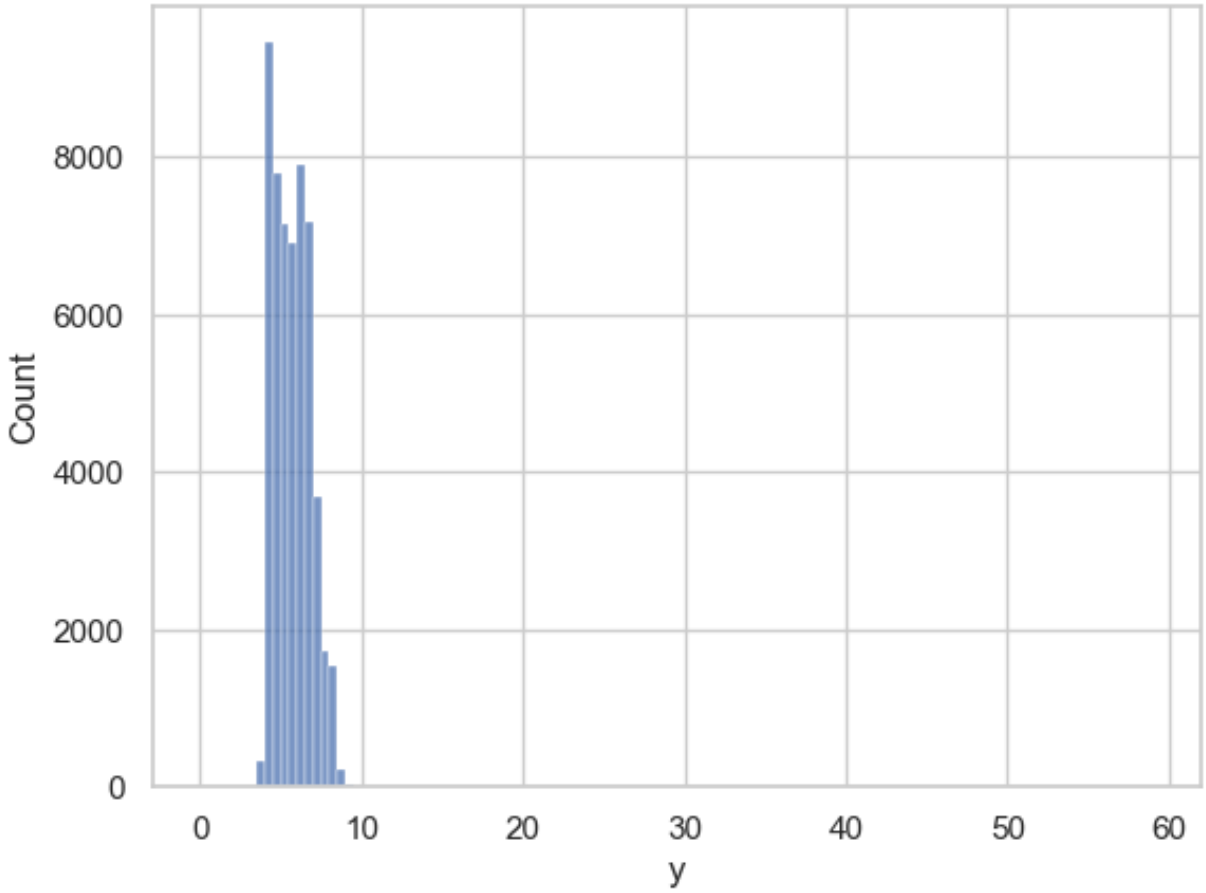
Sometimes outliers are data entry errors, whereas other times such outliers can suggest important new science.

With large datasets, outliers can be difficult to identify within a histogram.

For example, as suggested on the next slide, outliers within histograms will manifest as an unusually wide range on the x-axis.

Outliers observed as unusually large x-axis range

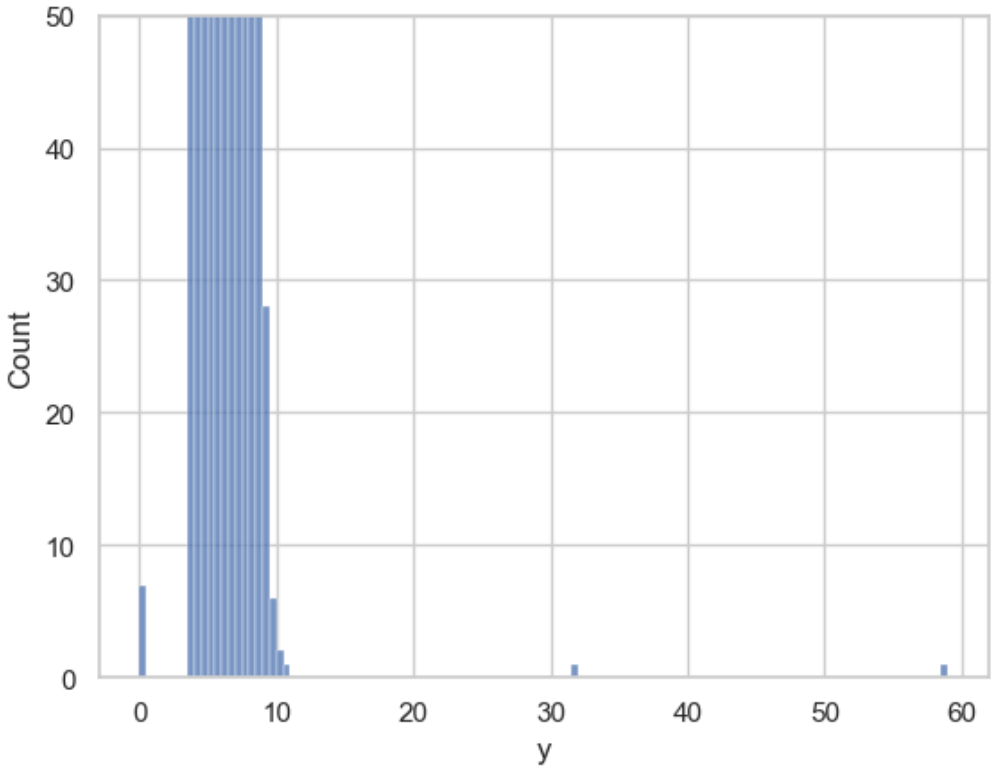
```
sns.histplot(data=diamonds, x='y', binwidth=0.5)
```



Here, there are so many observations in the common bins that the rare bins are so short and cannot be seen (for plotting purposes, they effectively have a 0 count on the y-axis scale).

Zoom into y-axis

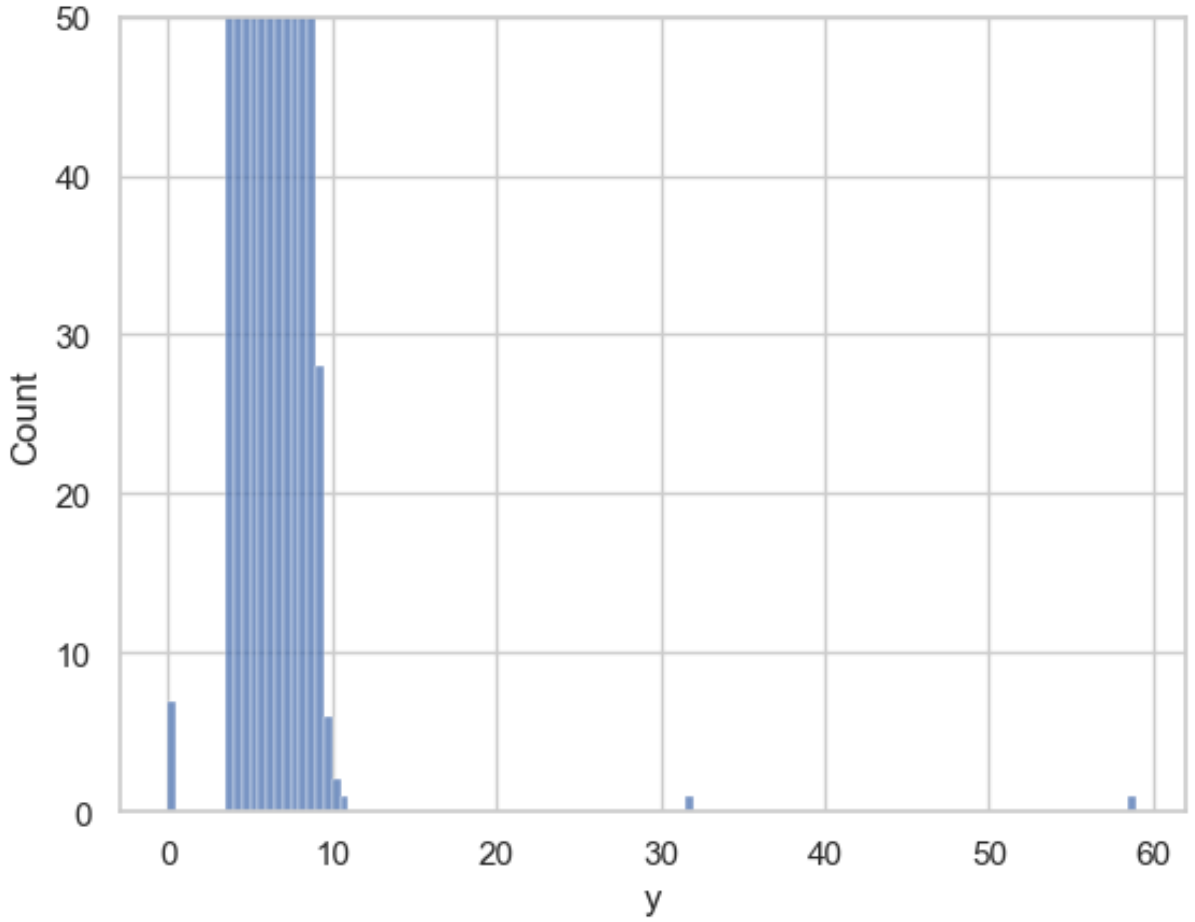
```
sns.histplot(data=diamonds, x='y', binwidth=0.5)
plt.ylim(0, 50)
```



The `plt.ylim()` function allows the user to change the plotted coordinate system, and to alter the range (limits) of the axes, with the above code setting the y-axis range to go from 0 to 50.

The x-axis scale can be similarly adjusted with `xlim()`.

Zoom into y-axis



This plot allows us to see that there are three unusual values: 0, approximately 30, and approximately 60.

We can use **`pandas`** to extract these values.

Using **pandas** to extract these unusual values

```
unusual = diamonds[-diamonds['y'].between(3,20)]
Unusual
```

Out[57]:

	carat	cut	color	clarity	depth	table	price	x	y	z	carat_bin
11963	1.00	Very Good	H	VS2	63.3	53.0	5139	0.00	0.0	0.00	(0.601, 1.002]
15951	1.14	Fair	G	VS1	57.5	67.0	6381	0.00	0.0	0.00	(1.002, 1.402]
24067	2.00	Premium	H	SI2	58.9	57.0	12210	8.09	58.9	8.06	(1.803, 2.204]
24520	1.56	Ideal	G	VS2	62.2	54.0	12800	0.00	0.0	0.00	(1.402, 1.803]
26243	1.20	Premium	D	VVS1	62.1	59.0	15686	0.00	0.0	0.00	(1.002, 1.402]
27429	2.25	Premium	H	SI2	62.8	59.0	18034	0.00	0.0	0.00	(2.204, 2.605]
49189	0.51	Ideal	E	VS1	61.8	55.0	2075	5.15	31.8	5.12	(0.195, 0.601]
49556	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.0	0.00	(0.601, 1.002]
49557	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.0	0.00	(0.601, 1.002]

Something is wrong with these observations

```
unusual[['x','y','z']]
```

Out[59]:

	x	y	z
11963	0.00	0.0	0.00
15951	0.00	0.0	0.00
24067	8.09	58.9	8.06
24520	0.00	0.0	0.00
26243	0.00	0.0	0.00
27429	0.00	0.0	0.00
49189	5.15	31.8	5.12
49556	0.00	0.0	0.00
49557	0.00	0.0	0.00

The feature **`y`** measures one of the three dimensions of a diamond in mm.

We know that diamonds cannot have a width of 0mm, suggesting that these data points with values of 0 are erroneous.

Something is wrong with these observations

```
unusual[['x','y','z', 'price']].sort_values('y')
```

Out[63]:

	x	y	z	price
11963	0.00	0.0	0.00	5139
15951	0.00	0.0	0.00	6381
24520	0.00	0.0	0.00	12800
26243	0.00	0.0	0.00	15686
27429	0.00	0.0	0.00	18034
49556	0.00	0.0	0.00	2130
49557	0.00	0.0	0.00	2130
49189	5.15	31.8	5.12	2075
24067	8.09	58.9	8.06	12210

Also, measurements of approximately 32mm and 59mm are likely implausible, as this would imply the diamonds are over an inch long.

However, diamonds that are over an inch long would not cost less than a hundred thousand dollars.

What to do with such unusual observations?

It is good practice to repeat analyses with and without outliers.

If the outliers have minimal effect on results, and one cannot figure out why they are in the dataset, then it is reasonable to replace them with missing values (`np.nan`).

However, if the outliers have substantial effect on the results, then they should not be dropped without justification.

The analyst will need to figure out what caused these unusual values (*e.g.*, a data entry error), and disclose that they were removed in whatever write-up or report is made based on the data.

If unusual values are encountered

If unusual values are encountered in the dataset and one wants to move on to the rest of the analysis, then there are two options:

1. Drop the entire row (observation) with strange values

```
diamonds2 = diamonds[diamonds['y'].between(3,20)]
```

Not necessarily recommended, as there is no reason to believe that just because one measurement is invalid that all other measurements on that observation are invalid.

If unusual values are encountered

If unusual values are encountered in the dataset and one wants to move on to the rest of their analysis, then there are two options:

2. Replace unusual values with missing values (**preferred solution**)

```
diamonds['y'] = diamonds['y'].where(
    (diamonds['y'] >= 3) & (diamonds['y'] <= 20),
    np.nan)
```

The `where(cond, other=nan)` function replaces values where the condition is False.

The first argument `cond` should be a logical expression.

Sometimes missing values are not errors

Recall that in the **`flights`** data frame, missing values in the **`dep_time`** feature indicate that the flight was canceled.

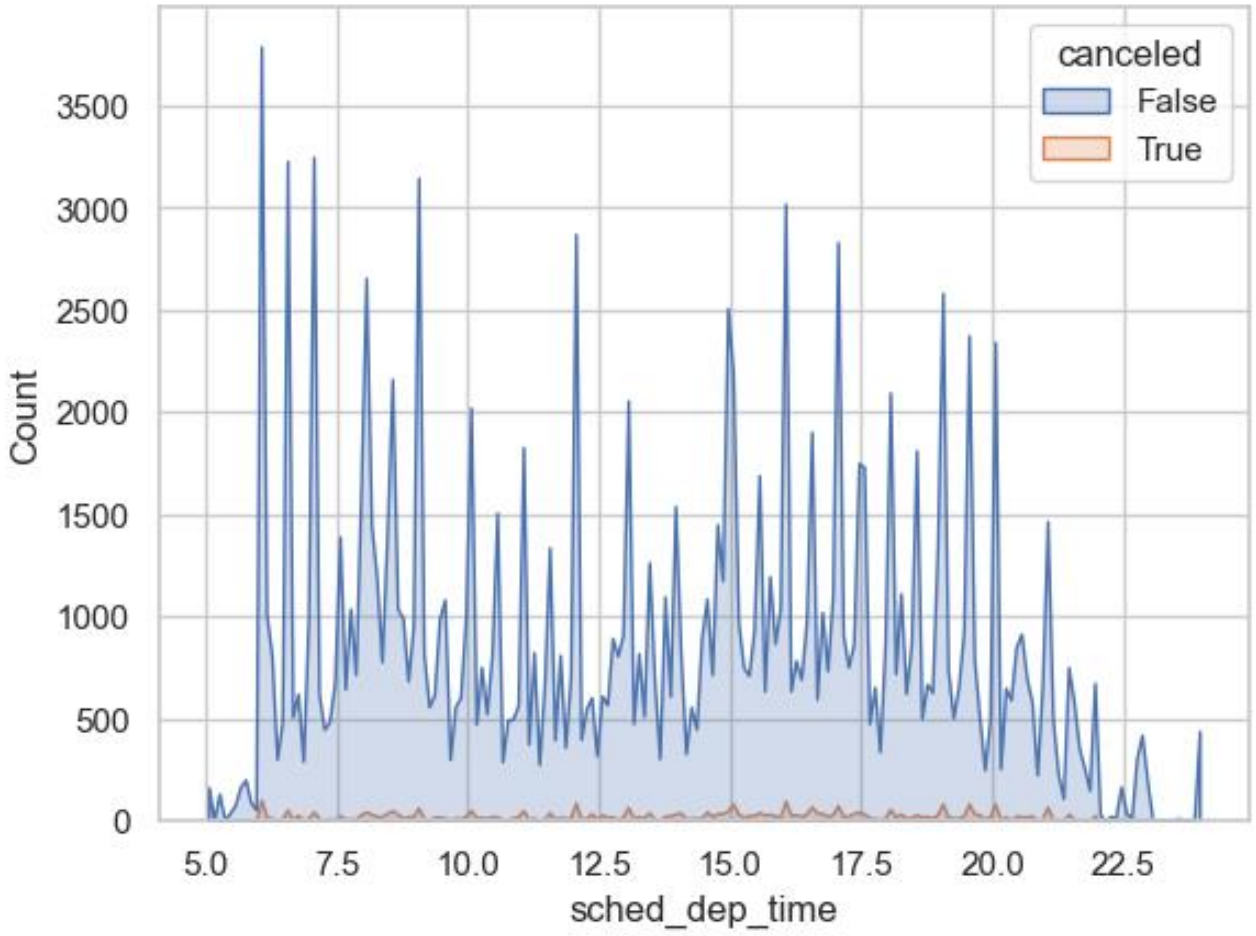
To compare the scheduled departure times for canceled and non-canceled times, a new variable can be created with **`is.na()`**.

```
# Add 'canceled' column indicating if 'dep_time' is missing
flights['canceled'] = flights['dep_time'].isna()

# Add 'sched_hour' and 'sched_min' by converting 'sched_dep_time' from HHMM to hours and minutes
flights['sched_hour'] = flights['sched_dep_time'] // 100
flights['sched_min'] = flights['sched_dep_time'] % 100

# Recalculate 'sched_dep_time' to be in hours
flights['sched_dep_time'] = flights['sched_hour'] + flights['sched_min'] / 60
```

We have introduced two new operators here: **`//`** and **`%`**, which respectively yield the **quotient** and the **remainder** from a division operation.

Departure times for canceled and non-canceled flights

```
sns.histplot(data=flights, x='sched_dep_time',
             hue='canceled',
             binwidth=0.1,
             element="poly")
```

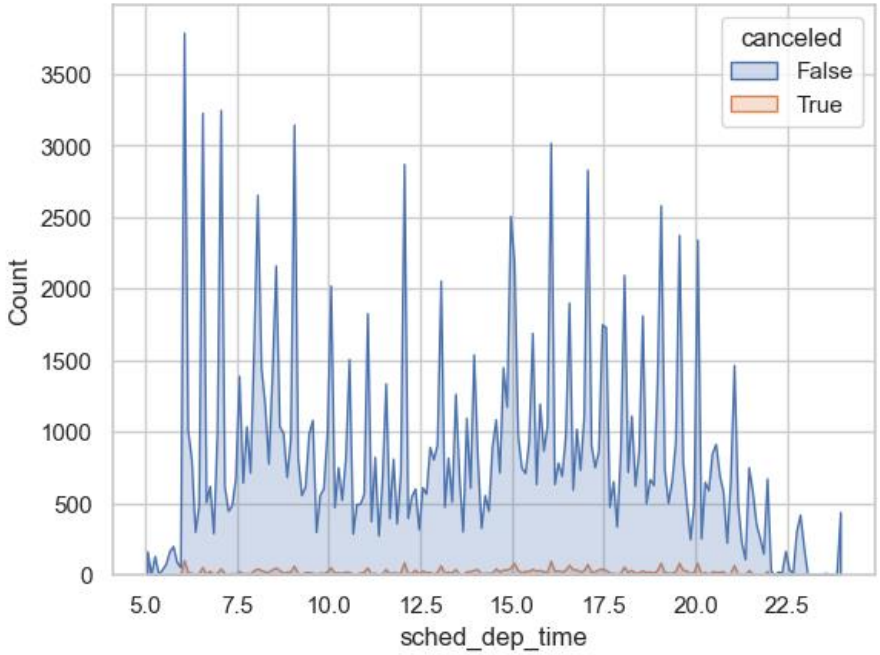


Graph not that informative, as on different scales

The graph depicted below is not that informative, as the scale on y-axes is different for the two categories.

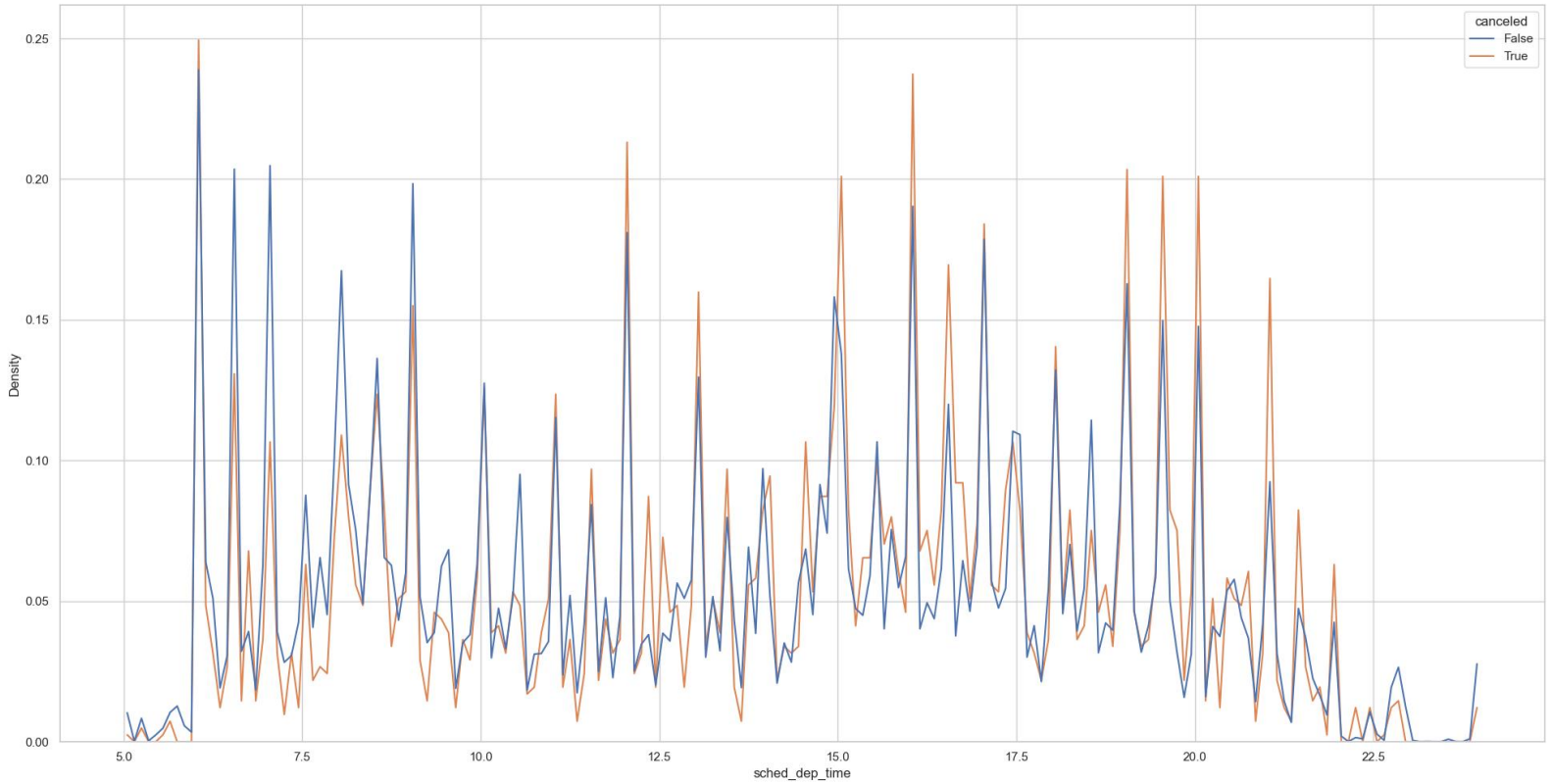
That is, the number of non-canceled flights is much greater than the number of canceled flights, and so no pattern can be discerned from the distribution of canceled flights.

We can instead scale the y-axis to plot the density rather than count distribution, such that the area under each curve sums to one.



Departure times for canceled and non-canceled flights

```
sns.histplot(data=flights, x='sched_dep_time',
             hue='canceled',
             binwidth=0.1,
             element="poly",
             stat='density',
             common_norm=False,
             fill=False)
```

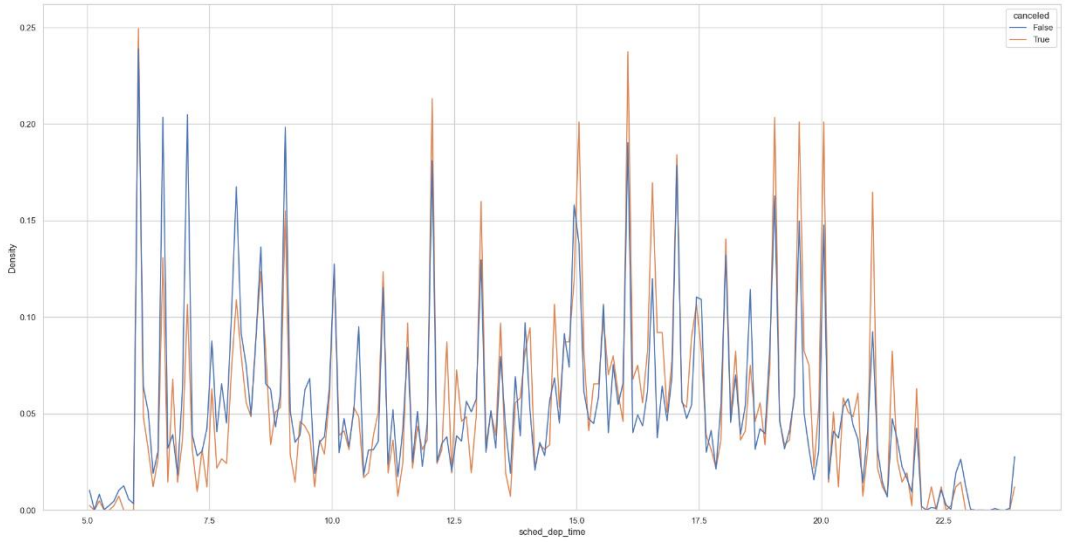


Flights may be more likely to be canceled later in day

A quick glance at the plotted graphs shows that the shapes of their distributions are similar.

However, a careful inspection suggests that there may be a key difference in the distributions, with canceled flights more likely to occur later in the day than at the beginning of the day.

Ultimately, this difference can be evaluated with a formal statistical test (model), but our exploration helped us arrive at this question.



Covariation

So far we have discussed variation within a single feature, which can be visualized across a categorical feature using bar plots and a continuous feature using histograms.

Covariation describes the variability between features, and is the tendency for values of two or more features to vary together in a particular way.

A way of identifying covariation is to visualize the relationship between two more features.

We will consider visualizations of three types of covariation:

- Categorical vs. continuous
- Categorical vs. categorical
- Continuous vs. continuous

Covariation between categorical & continuous features

It is common to explore the distribution of a continuous feature conditional on some categorical feature.

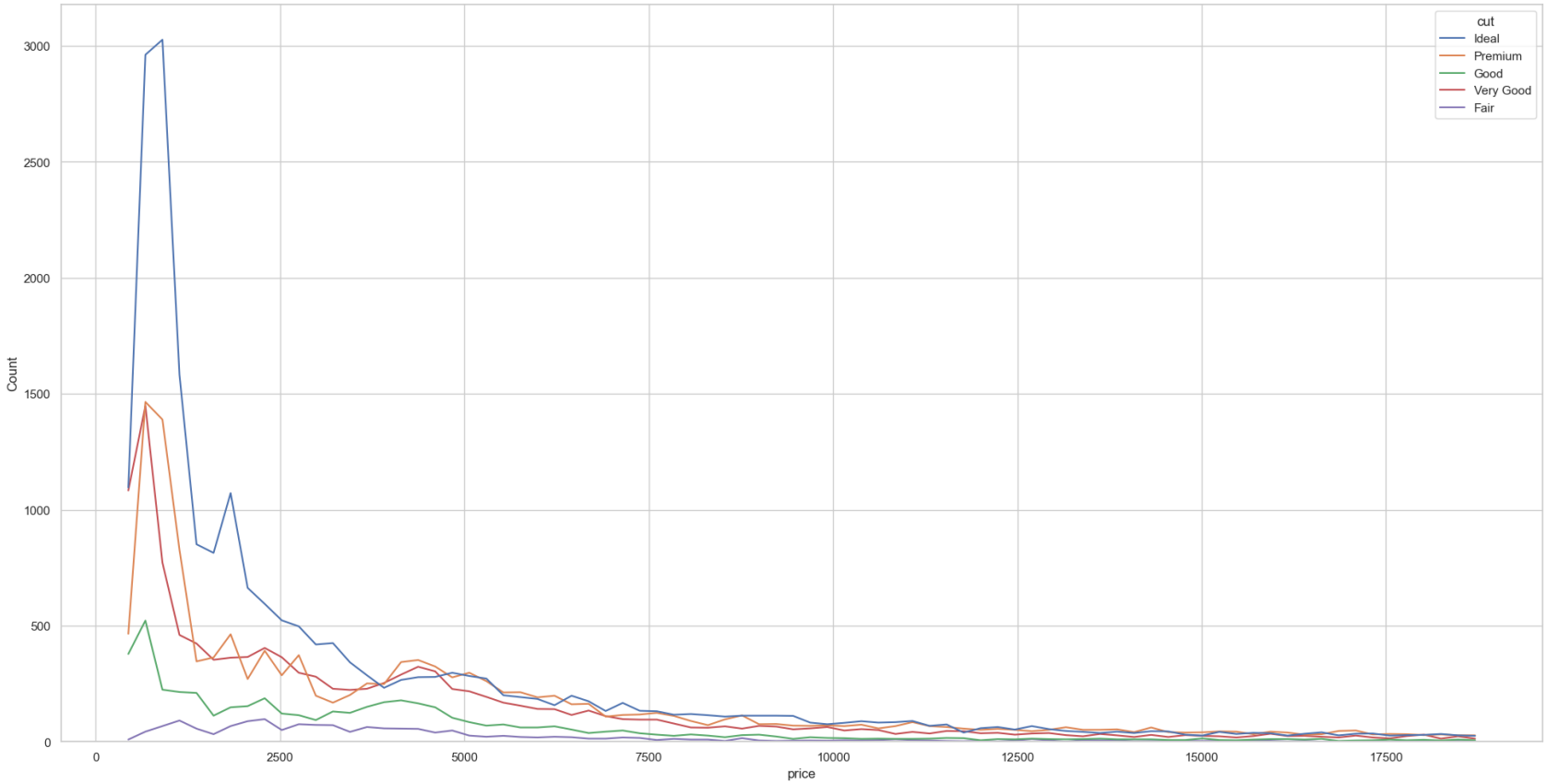
We have already observed two mechanisms of visualizing this, via frequency polygons `sns.histplot(element='poly')` and boxplots `sns.boxplot()`.

For frequency polygons, each distribution of a particularly continuous feature would be conditional on a particular categorical feature, and this distribution can be plotted with the density on the y-axis.

Box plots summarize the distribution of a continuous feature conditional on a particular categorical feature.

Frequency polygons visualize continuous vs. categorical

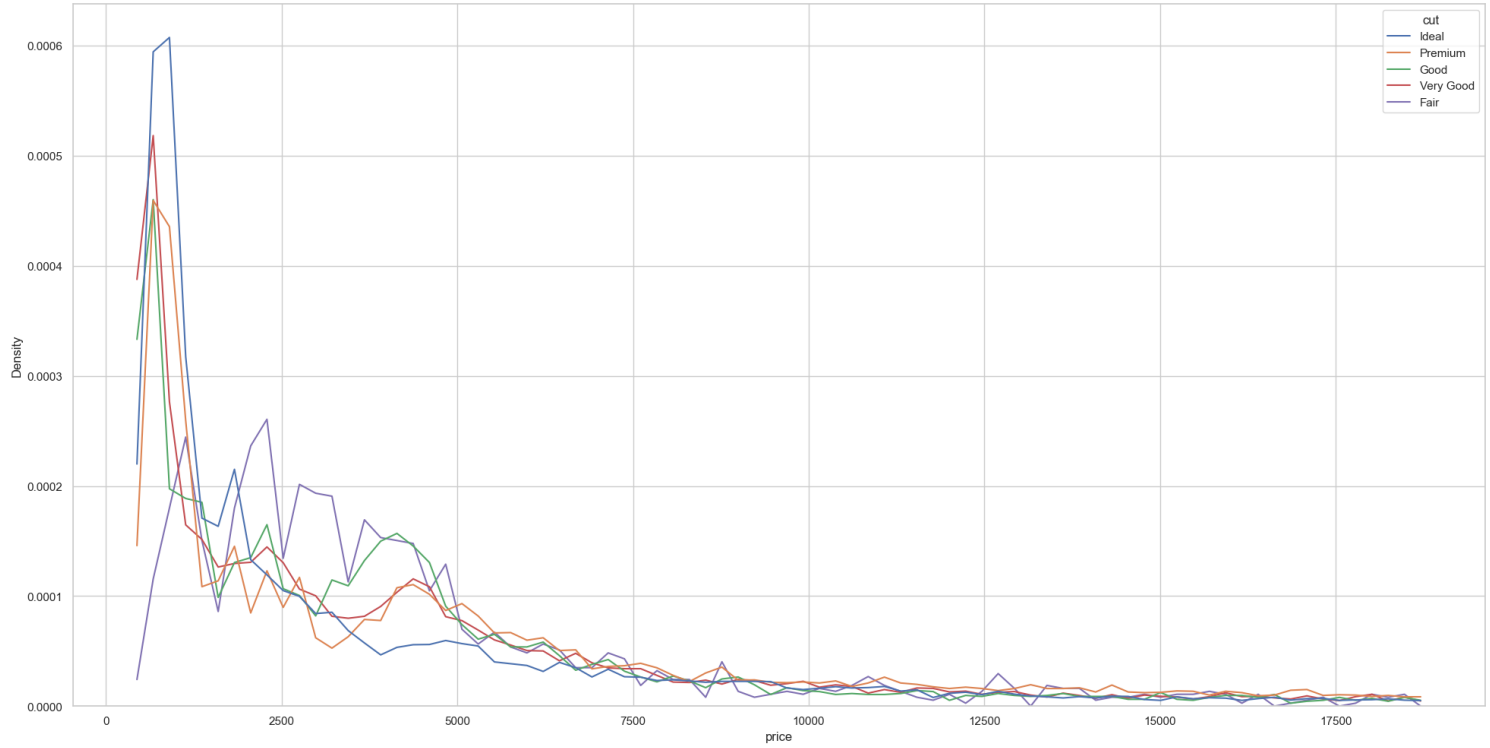
```
sns.histplot(data=diamonds, x='price',
             hue='cut',
             element="poly",
            common_norm=False,
            fill=False)
```



Unfortunately, in this example, it is difficult to compare the distributions of diamond price, as the different **`cut`** categories have highly different numbers of observations.

Frequency polygons visualize continuous vs. categorical

```
sns.histplot(data=diamonds, x='price', hue='cut',
             element="poly",
            common_norm=False,
             stat='density',
            fill=False)
```



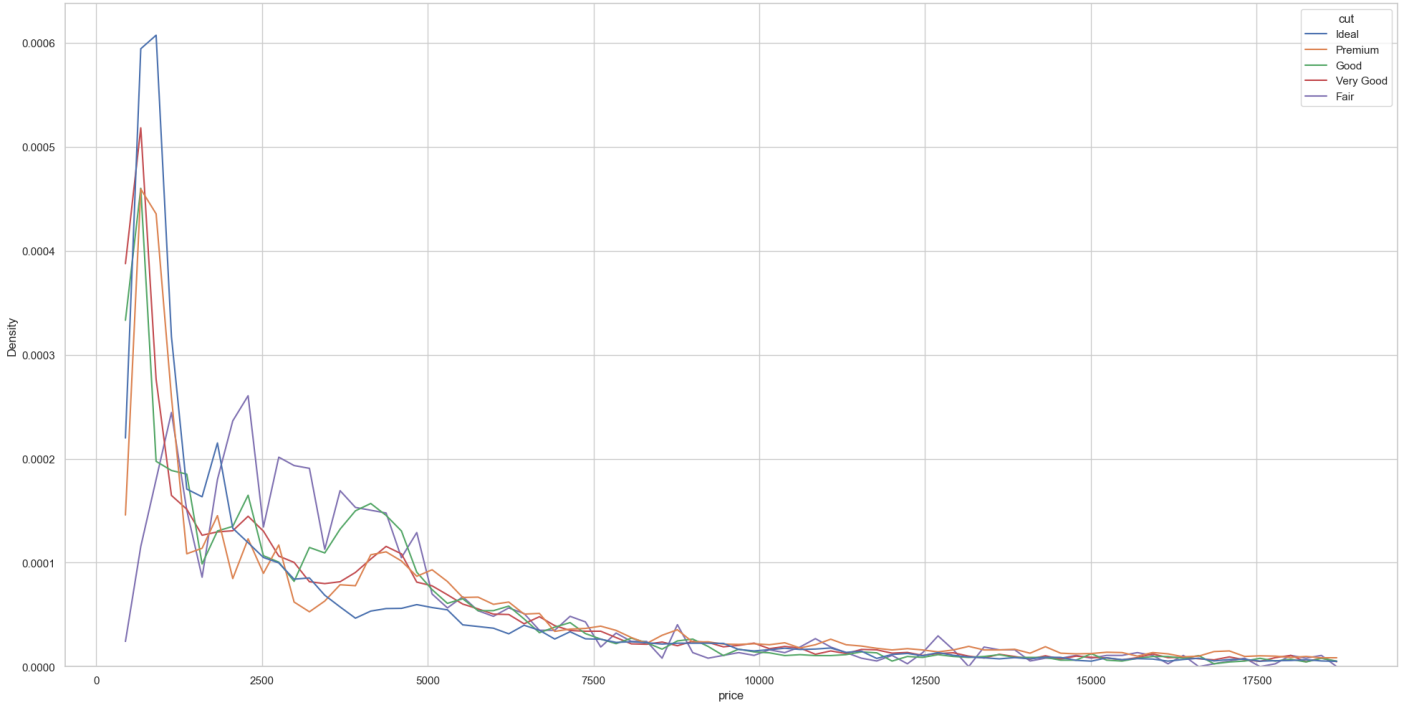
As an alternative, plot the frequency polygons so that the area under each distribution sums to one.

Fair diamonds have highest average price

This plot provided a somewhat surprising result, with the lowest quality (fair) diamonds have the highest average price.

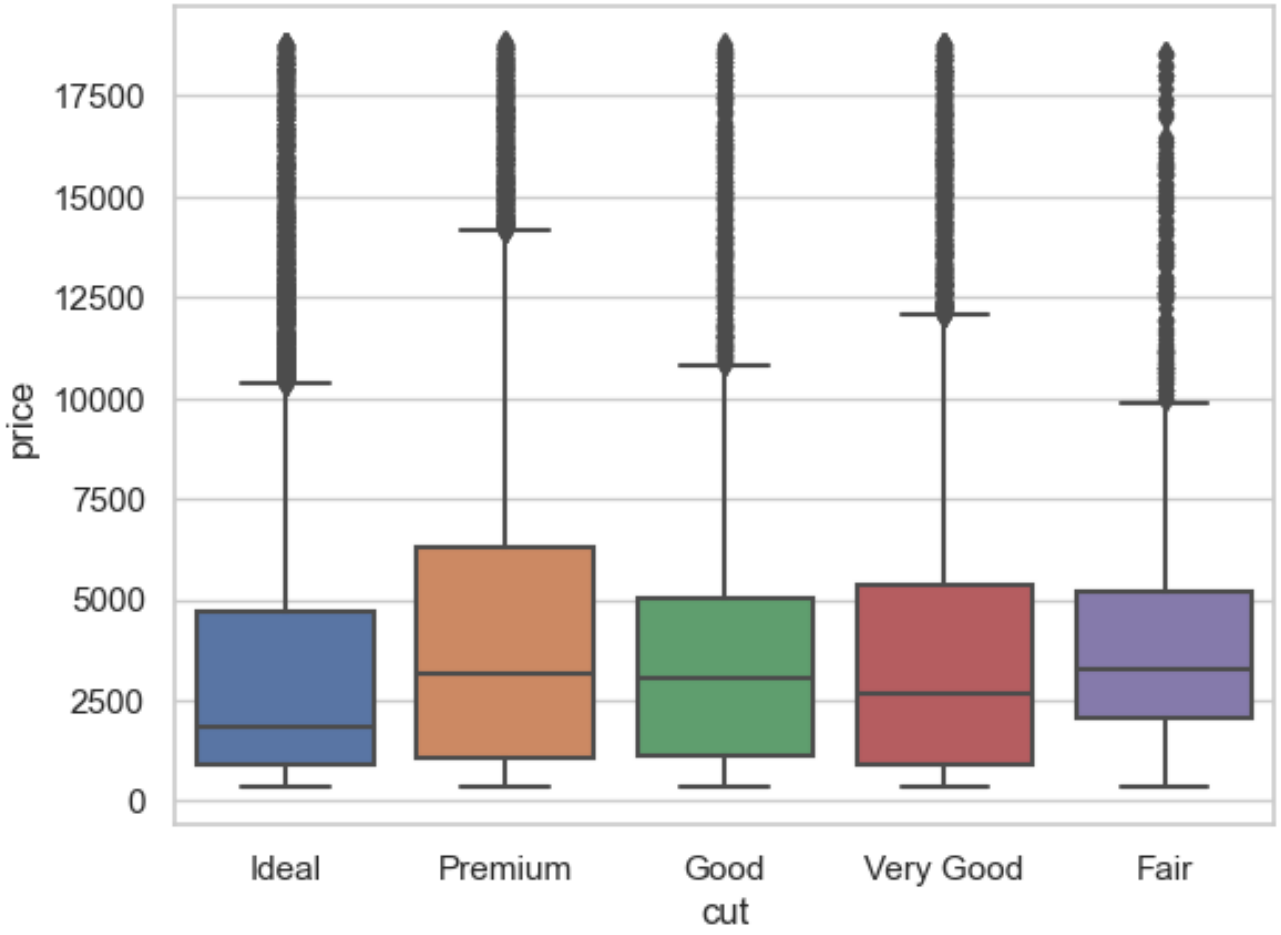
It is possible that we are just being mislead by looking at the distribution as frequency polygons, and maybe examining summaries of the distribution would solidify if this were the case.

We can examine such summaries using box plots.



Box plots to summarize continuous vs. categorical

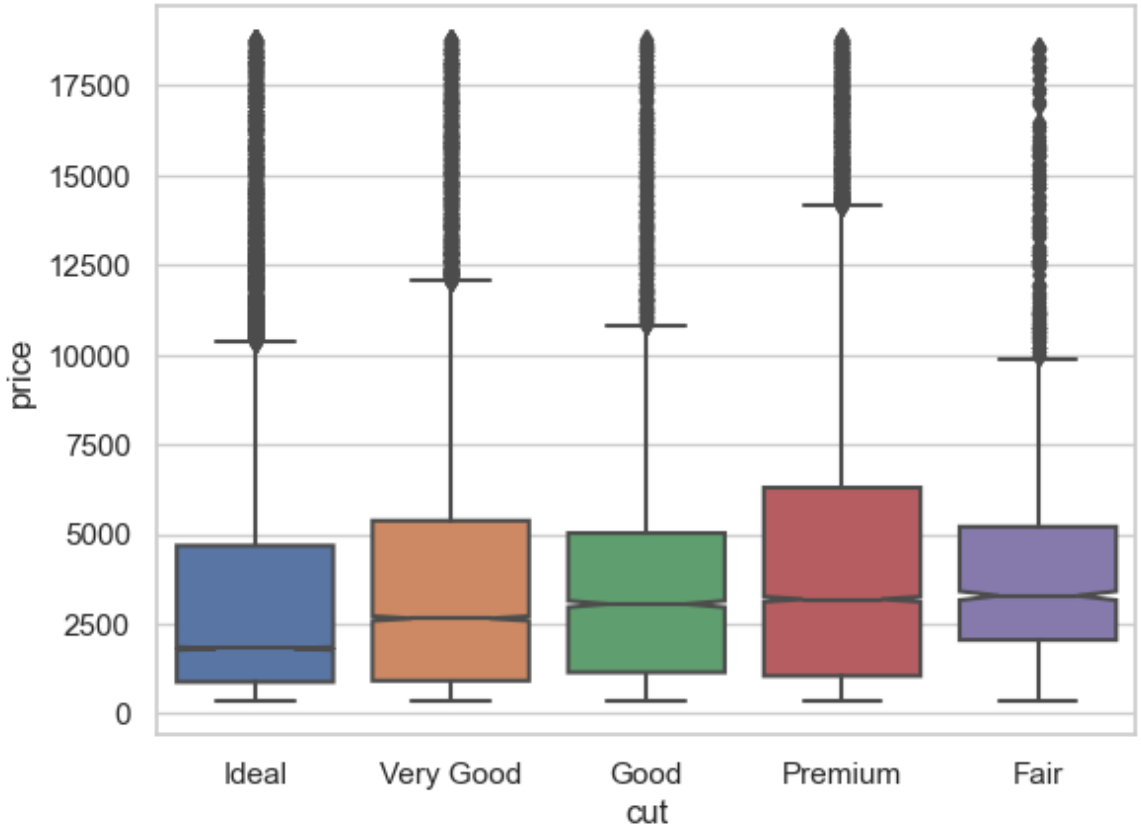
```
sns.boxplot(data=diamonds, x = 'cut', y = 'price')
```



It appears that in general, the distribution of values of lowest quality (fair) diamonds is shifted toward higher prices, and the median also seems be higher than other medians.

Notches suggest pattern may be statistically significant

```
sorted_cut = diamonds.groupby('cut')['price'].median().sort_values().index
sns.boxplot(data=diamonds, x='cut', y='price',
            notch=True,
           order=sorted cut)
```



Notches can be added to box plots using the **`notch`** argument, and represent a confidence interval around the median.

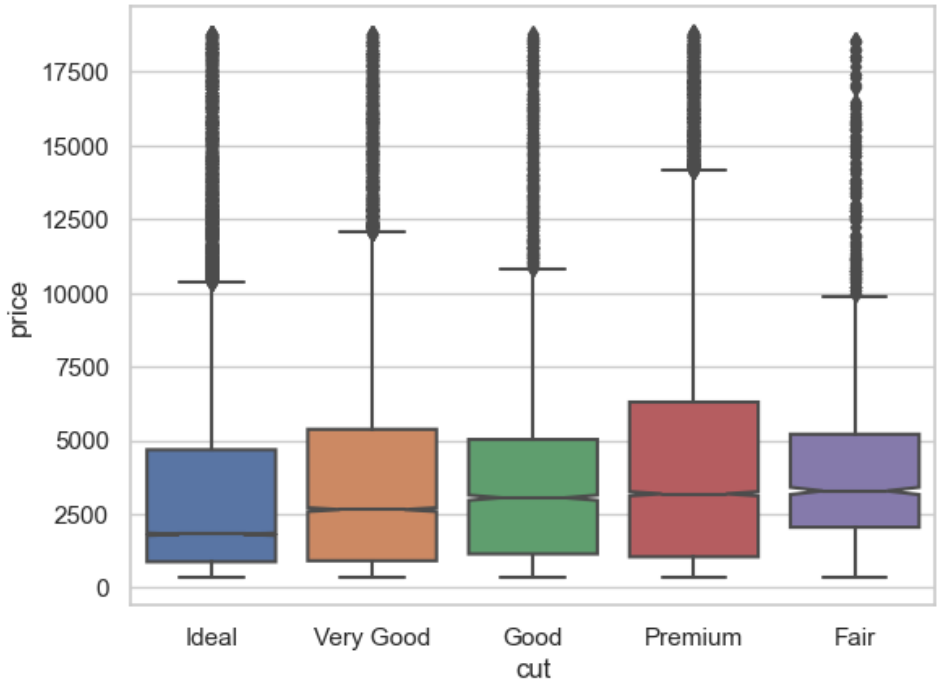
Non-overlapping notches suggest strong evidence that medians differ.

Why might fair diamonds have highest average price?

So why might fair diamonds have on average the highest price?

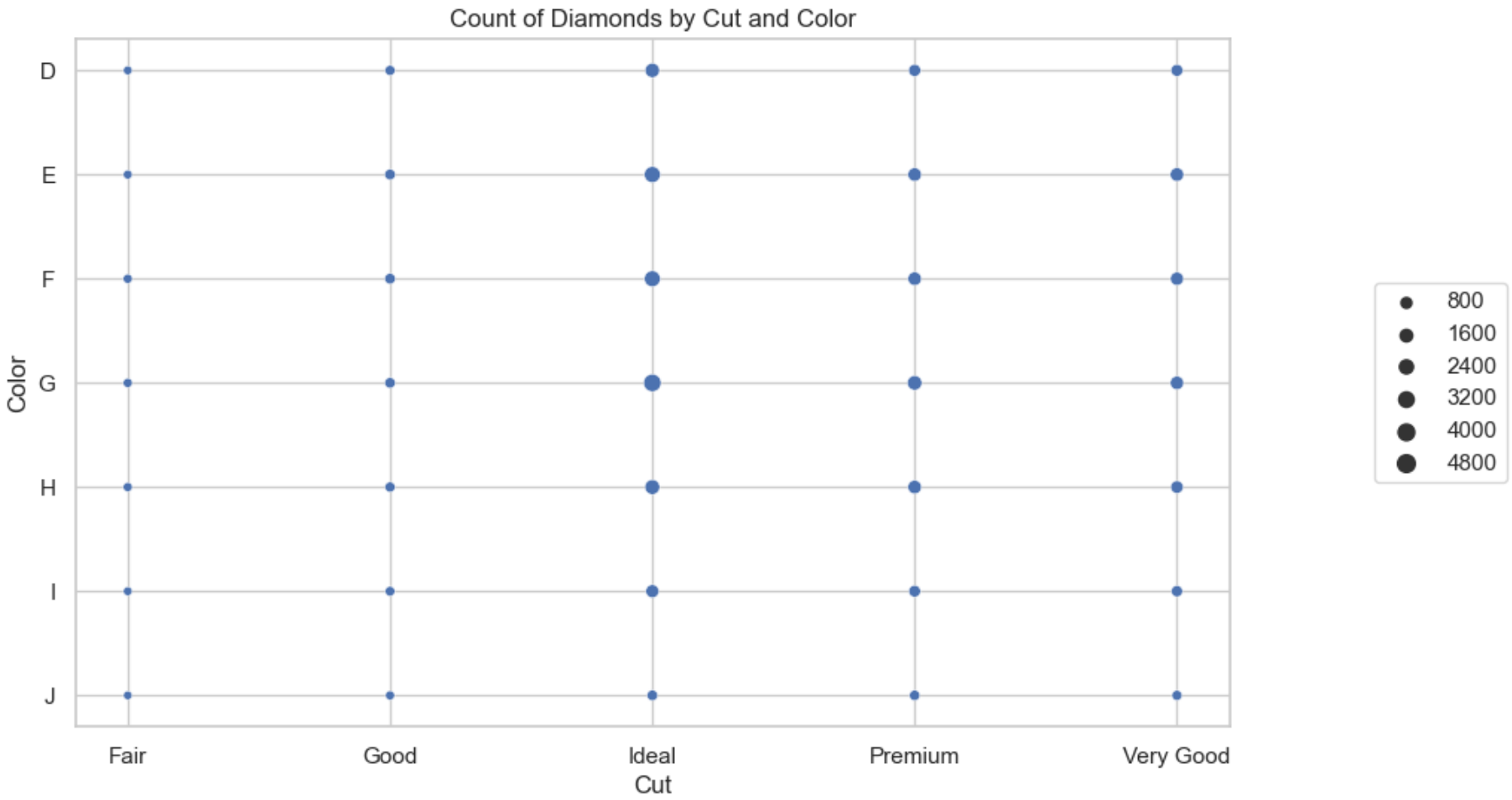
We will explore this question a little more when we discuss covariation between continuous features.

As a sneak peek, we will need to identify what feature has a strong relationship with diamond price.



Covariation between categorical features

```
counts = diamonds.groupby(['cut', 'color']).size().reset_index(name='counts')
sns.scatterplot(data=counts, x='cut', y='color', size='counts')
```



Plotted values can be computed without visualization

We can compute the plotted values without visualizing by using some of the techniques we have learned about data transformation.

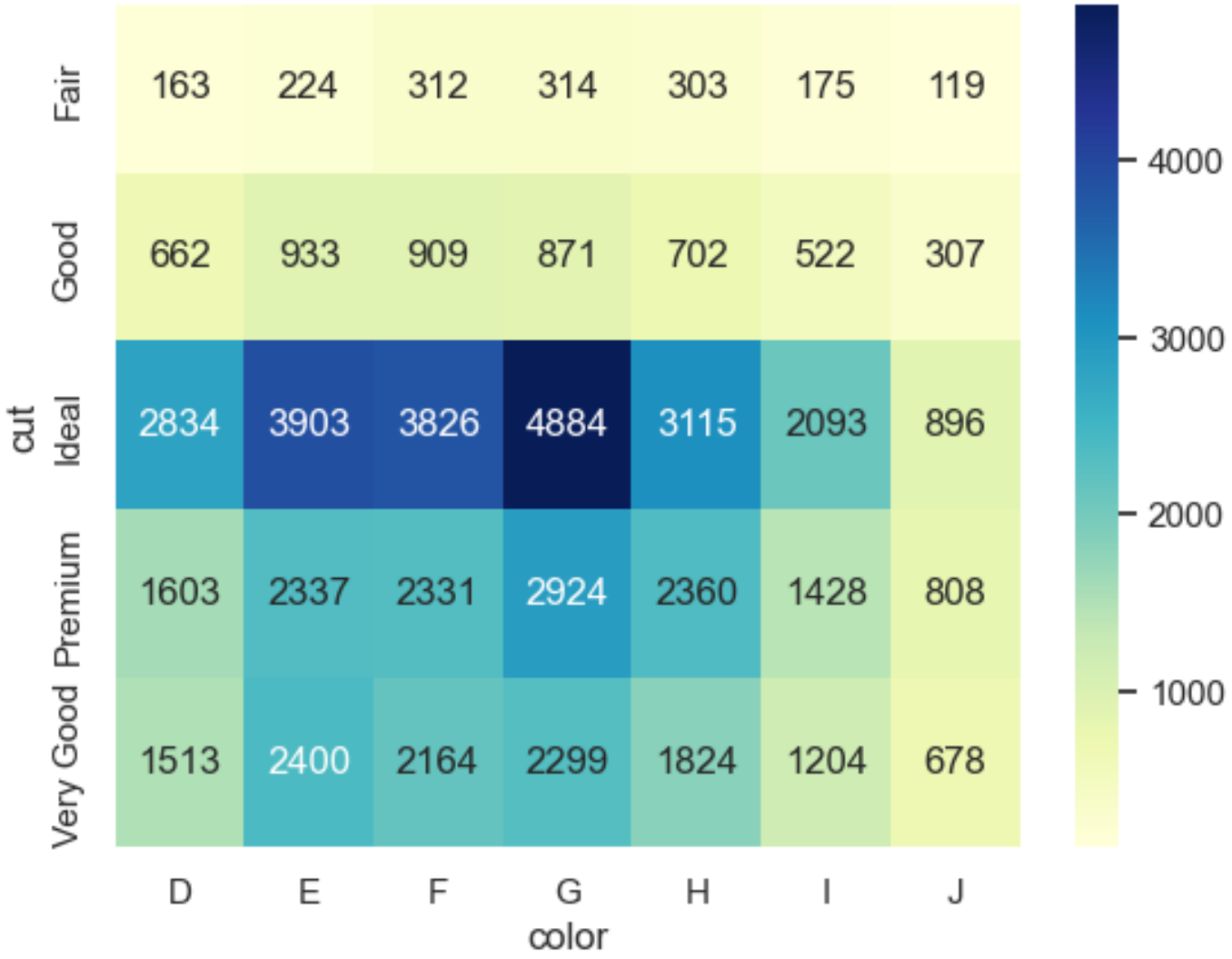
```
diamonds.groupby(['cut', 'color']).size().reset_index(name='counts')
```

Out[97]:

	cut	color	counts
0	Fair	D	163
1	Fair	E	224
2	Fair	F	312
3	Fair	G	314
4	Fair	H	303
5	Fair	I	175
6	Fair	J	119
7	Good	D	662
8	Good	E	933
9	Good	F	909
10	Good	G	871
11	Good	H	702
12	Good	I	522
13	Good	J	307
14	Ideal	D	2834
15	Ideal	E	3903
16	Ideal	F	3826
17	Ideal	G	4884
18	Ideal	H	3115
19	Ideal	I	2093
20	Ideal	J	896
21	Premium	D	1603
22	Premium	E	2337
23	Premium	F	2331

Instead create a heatmap of counts with `geom_tile()`

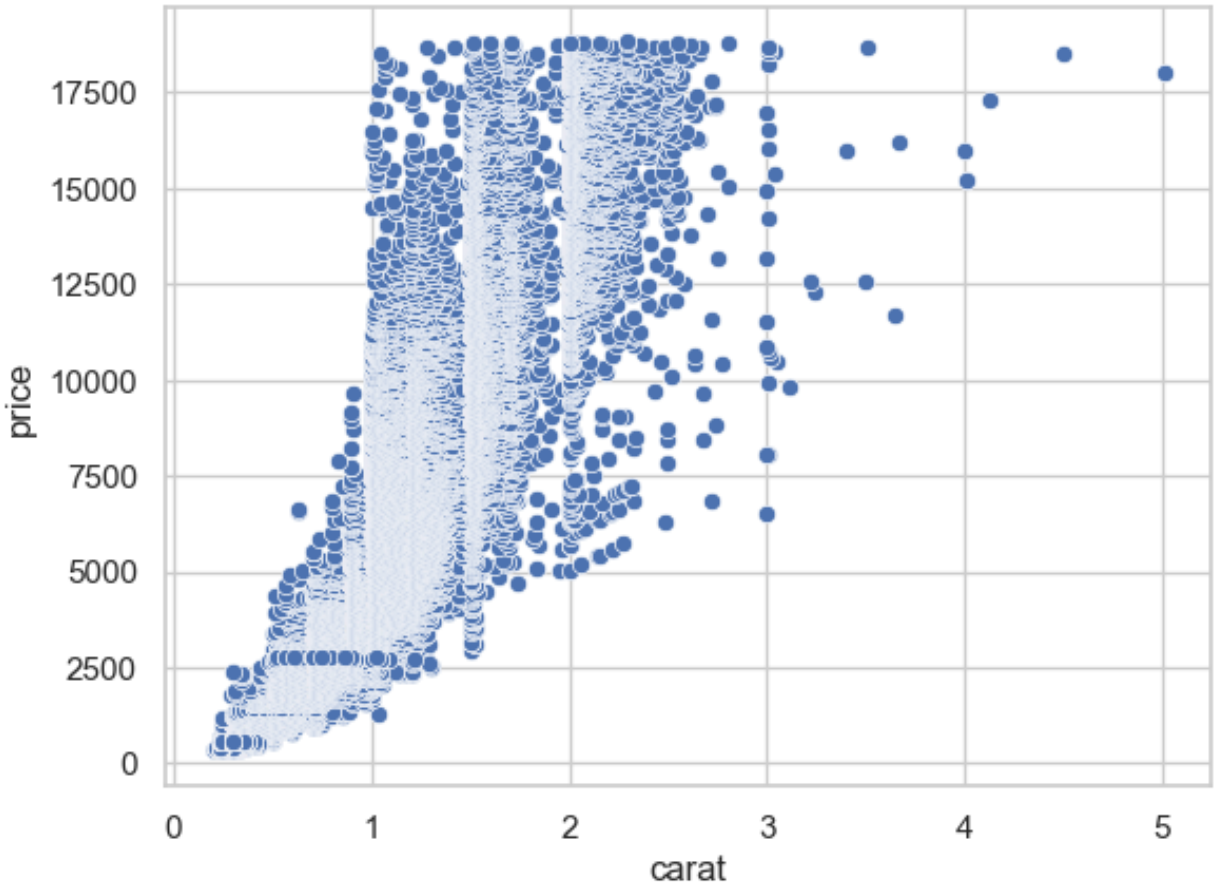
```
diamonds_count = diamonds.groupby(['color', 'cut']).size().reset_index(name='n')
sns.heatmap(heatmap_data, annot=True, fmt="d", cmap="YlGnBu")
```



Covariation between continuous features

We have already explored a way to examine covariation between continuous features, which was a scatter

```
sns.scatterplot(data=diamonds, x='carat', y='price')
```

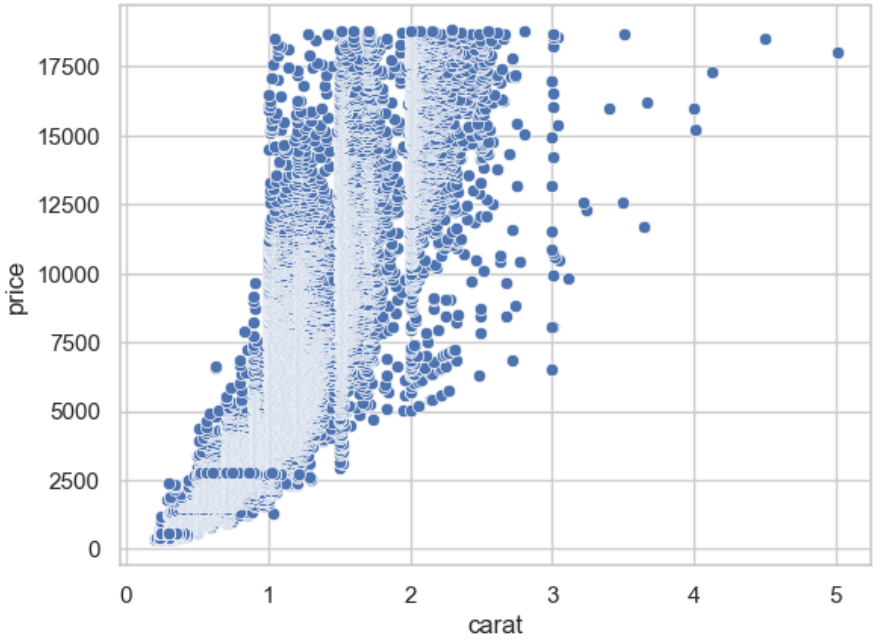


Why might fair diamonds have highest average price?

The plot below suggests that diamond **`price`** increases as diamond weight (**`carat`**) increases.

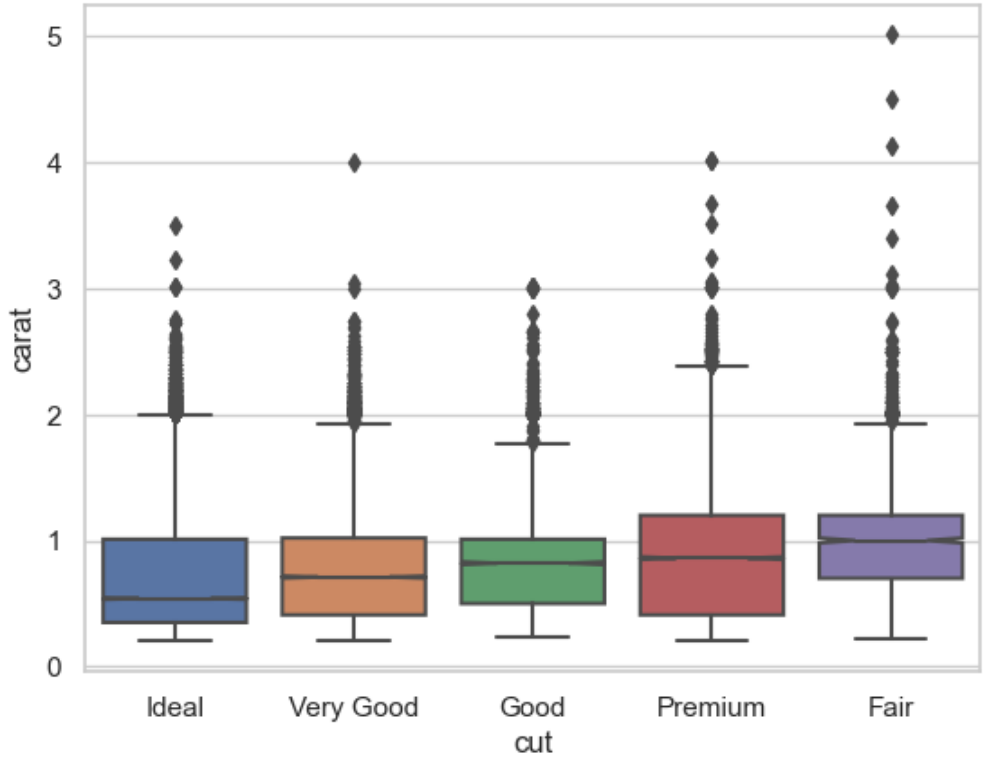
Therefore, maybe the reason for the higher average price of low-quality (fair) diamonds is that lower-quality diamonds tend to be heavier.

We can explore this hypothesis by creating a box plot to summarize the distribution of diamond weight as a function of diamond quality.



Fair diamonds are heaviest on average

```
sns.boxplot(data=diamonds, x='cut', y='carat',
            notch=True)
```



Supporting our hypothesis, fair diamonds have their weight distribution shifted toward larger values.

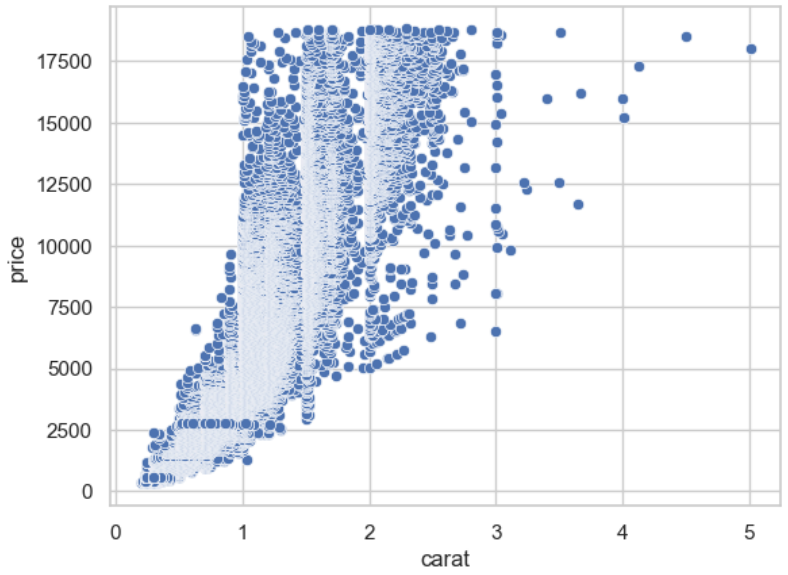
Also, notches do not overlap between fair and other qualities (**`cut`**).

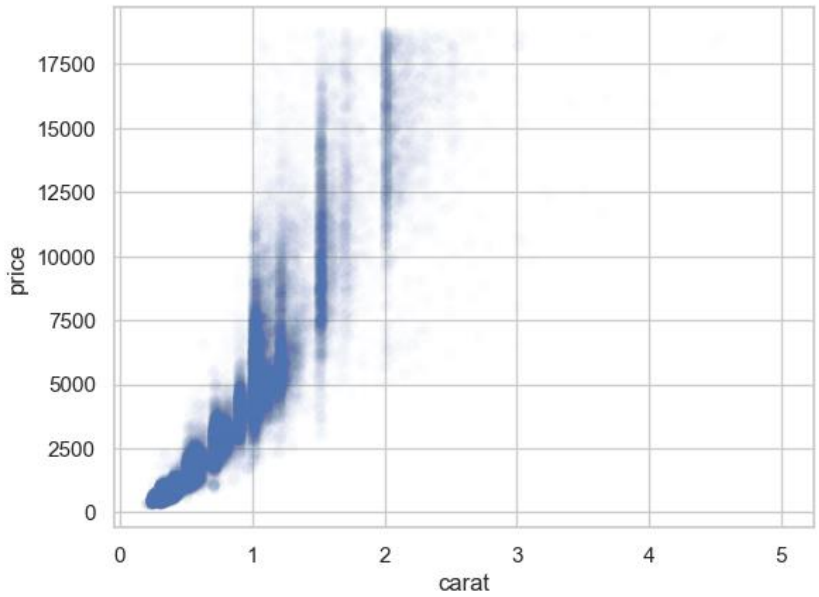
Visualizing distribution of points when they overlap

The plot to the left contains over 50,000 points, making it difficult to visualize where observations tend to be concentrated.

For such a plot, it can be beneficial to set the transparency of points such that overlaps can be observed (right), with the code

```
sns.scatterplot(data=diamonds, x='carat', y='price',
                alpha=1/100)
```



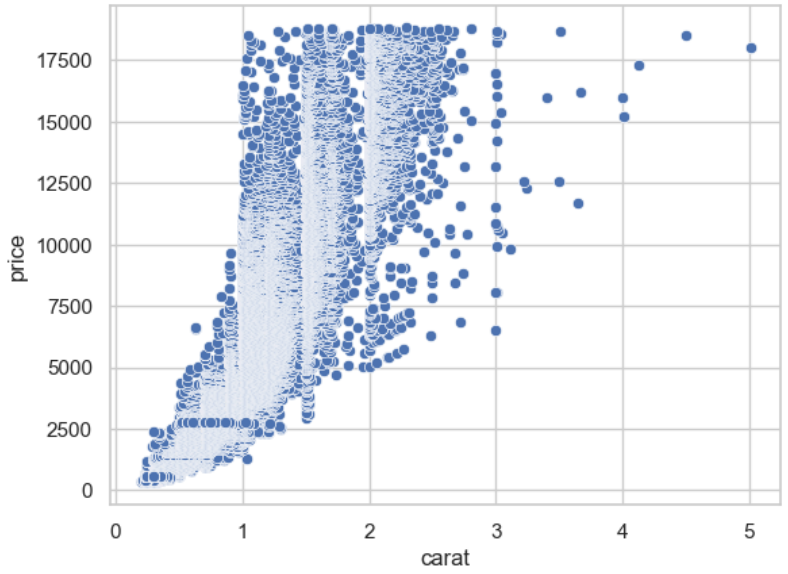


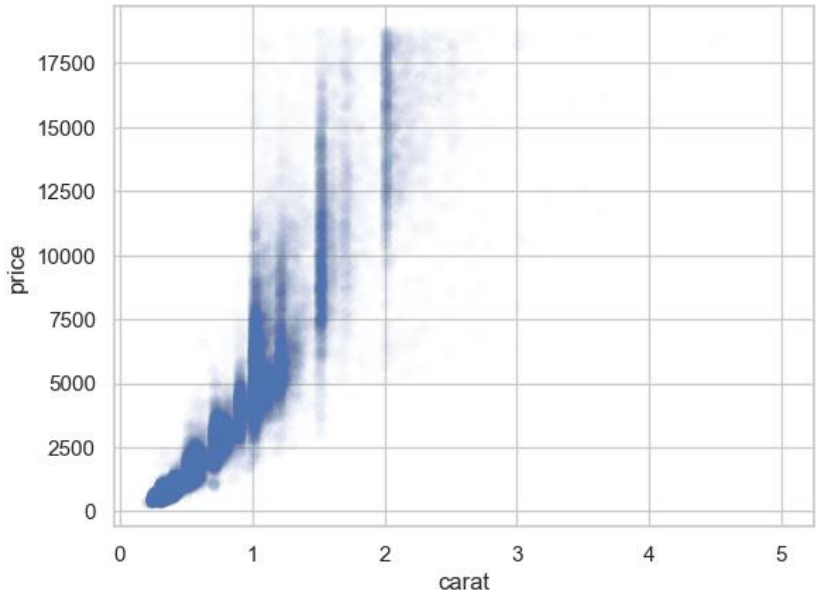
Using such transparency can be challenging though

Using such transparency can be challenging for large datasets, as all regions will look the same (blue) with a sufficient number of points.

A solution to this would be to make the points more transparent, but this makes them invisible in regions with too few points.

A solution is to plot overlaps as a heatmap using either the `geom_bin2d()` or the `geom_hex()` functions.



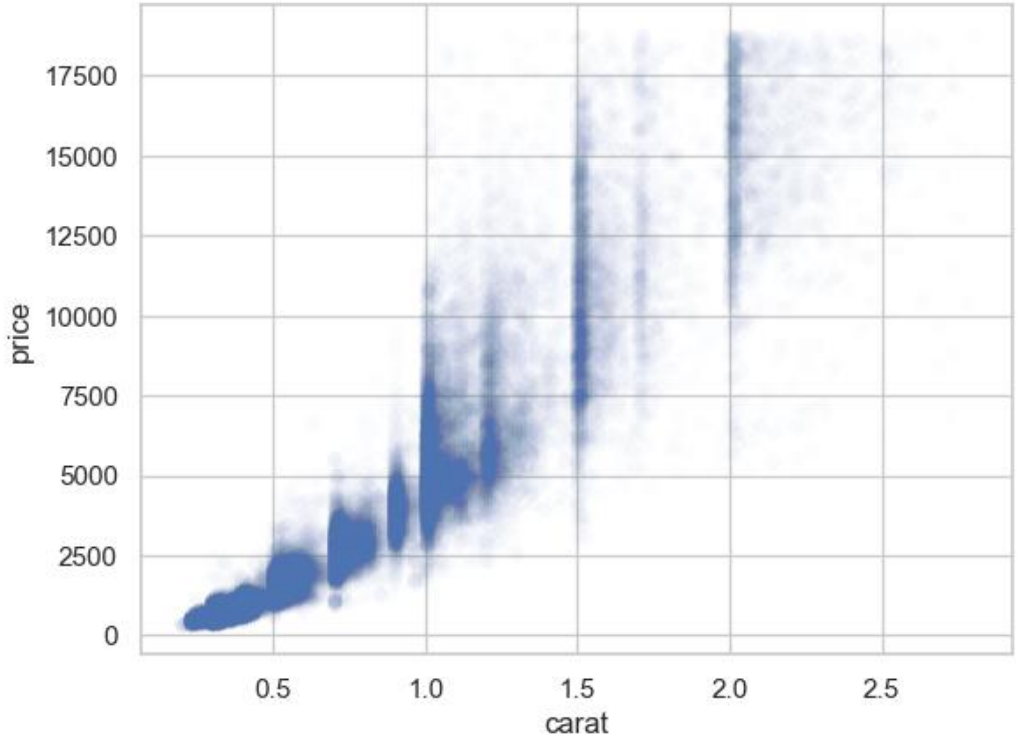


Illustrating scatterplot heatmaps

Let's reconsider the filtered dataset **`smaller`** we generated earlier, which was the set of observations weighing less than three carats.

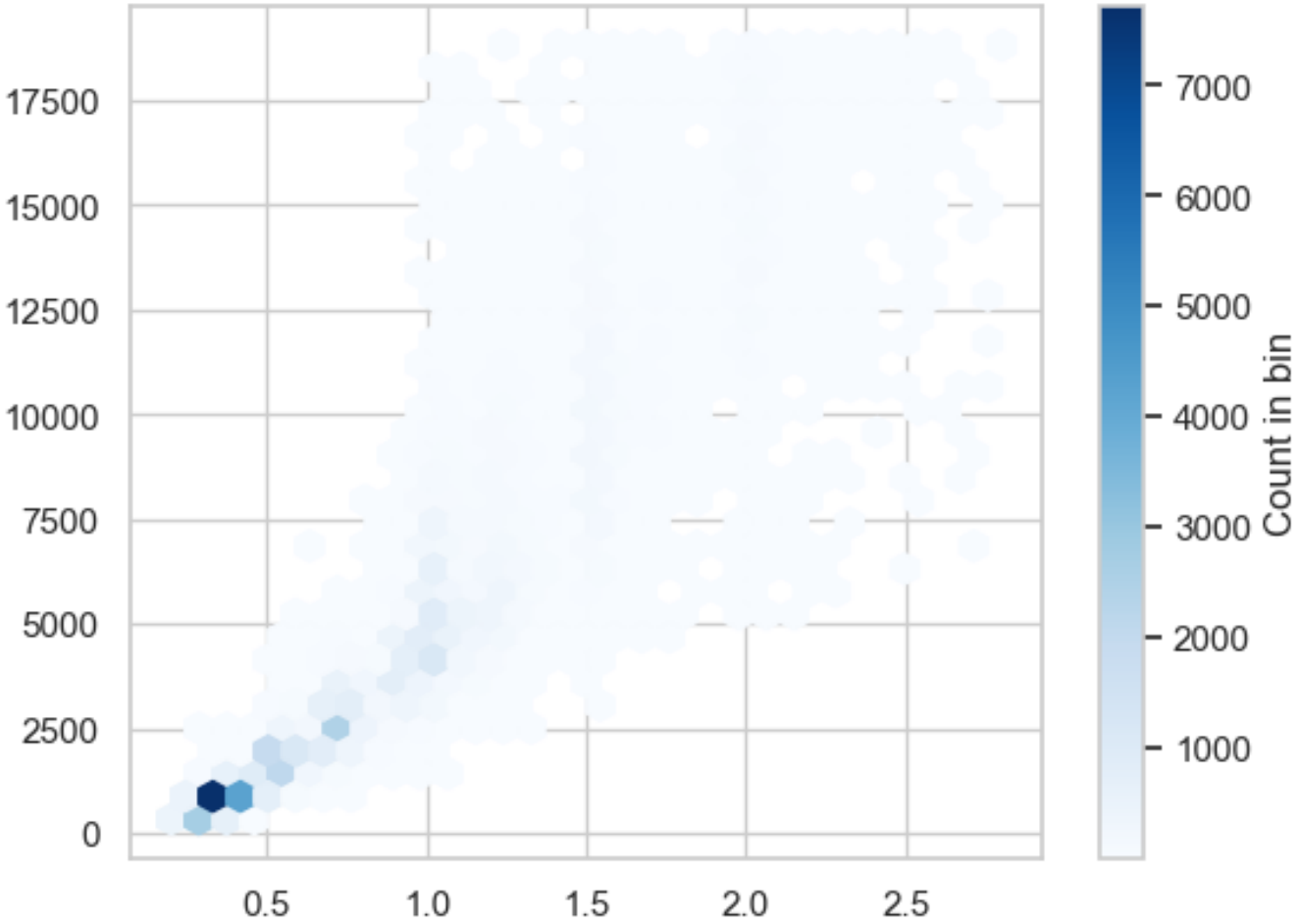
```
diamonds_f = diamonds[diamonds['carat']<3]
```

A scatter plot with transparency of this dataset is below.



Scatterplot heatmap with `plt.hexbin()`

```
plt.hexbin(daiamonds_f['carat'], daiamonds_f['price'],
           gridsize=30,
           cmap='Blues', mincnt=1)
plt.colorbar(label='Count in bin')
```



Discretizing one continuous feature through binning

It is possible to bin the space of values for one continuous feature into a discrete number of values, thereby converting a continuous feature into a categorical feature.

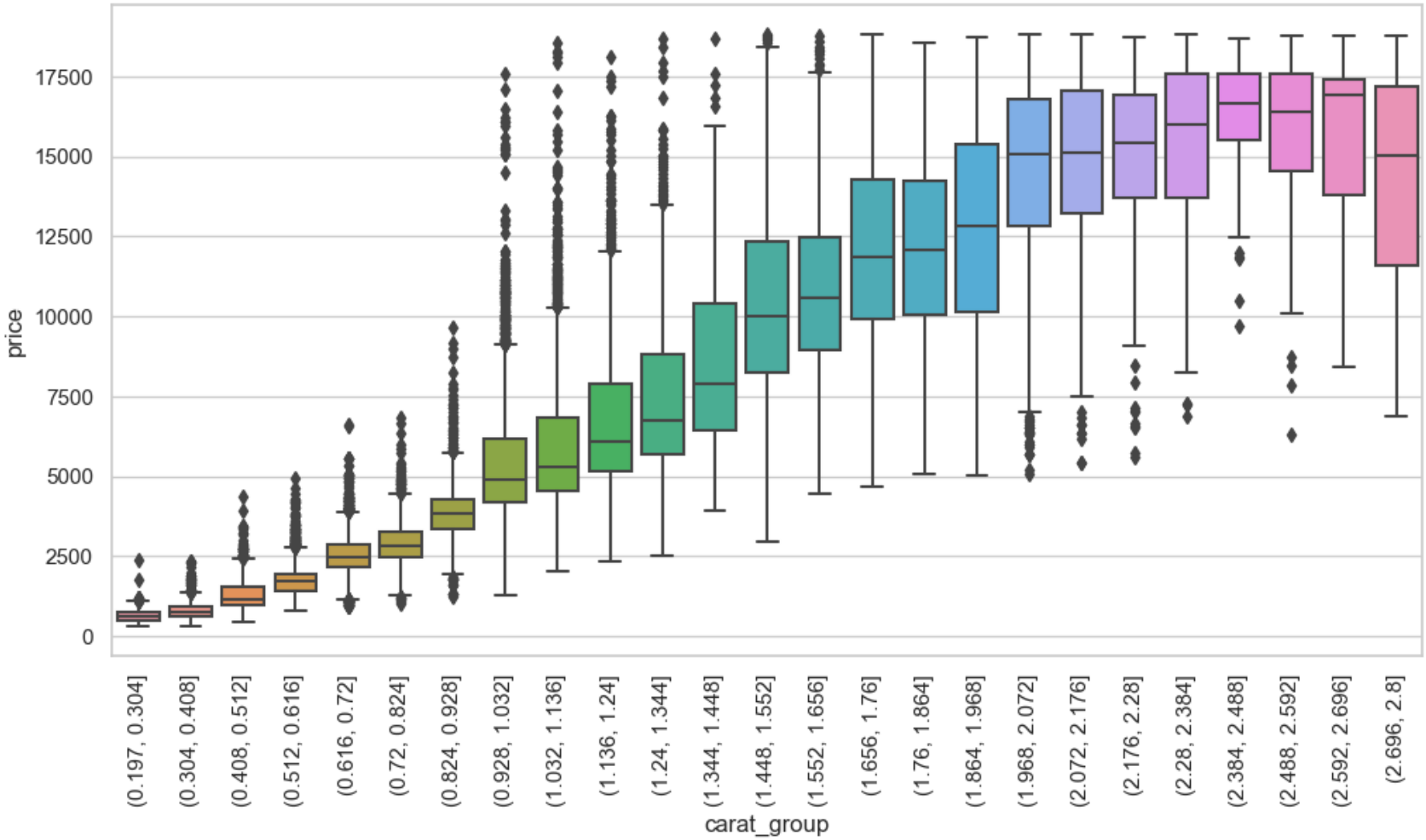
Such discretization would permit an analyst to employ one of the techniques (*e.g.*, box plots) for visualizing combinations of categorical and continuous features.

To perform this discretization, the analyst will choose a variable to partition (bin), and assign each of these bins as a separate group.

This space can be partitioned with either the function `cut()` for partitioning the space of values evenly according to a width argument, or with `bins` for partitioning the space into a specific number of bins with each bin having approximately the same number of observations.

Discretizing with `cut()`

```
daiamonds_f['carat_group'] = pd.cut(daiamonds_f['carat'], bins=25)
sns.boxplot(data=daiamonds_f, x='carat_group', y='price')
plt.xticks(rotation=90)
```



Patterns and models

Patterns in data provide clues about relationships, and if a systematic relationship exists between two features, then it will appear as a pattern in the data.

If one spots a pattern, then they should ask:

- Could this pattern be due to coincidence (*i.e.*, random chance)?
- How can one describe the relationship implied by the pattern?
- How strong is the relationships implied by the pattern?
- What other features might affect the relationship?
- Does the relationship change if you look at individual subgroups of data?

Patterns reveal covariation

Patterns are highly useful tools because they reveal covariation.

Variation is a phenomenon that **creates uncertainty**.

However, covariation is a phenomenon that **reduces uncertainty**.

If two features covary, then you can use the values of one feature to make better predictions about the values of the second feature.

If covariation is due to a causal relationship, which is a special case, then you can use the value of one feature to control the value of the second feature.

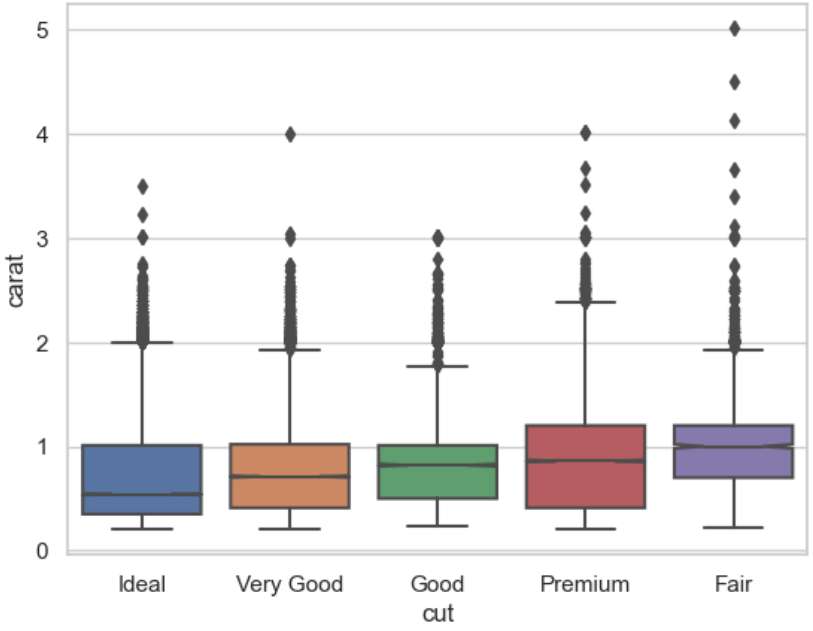
Models are tool for extracting patterns out of data

Models are tools for extracting patterns from data.

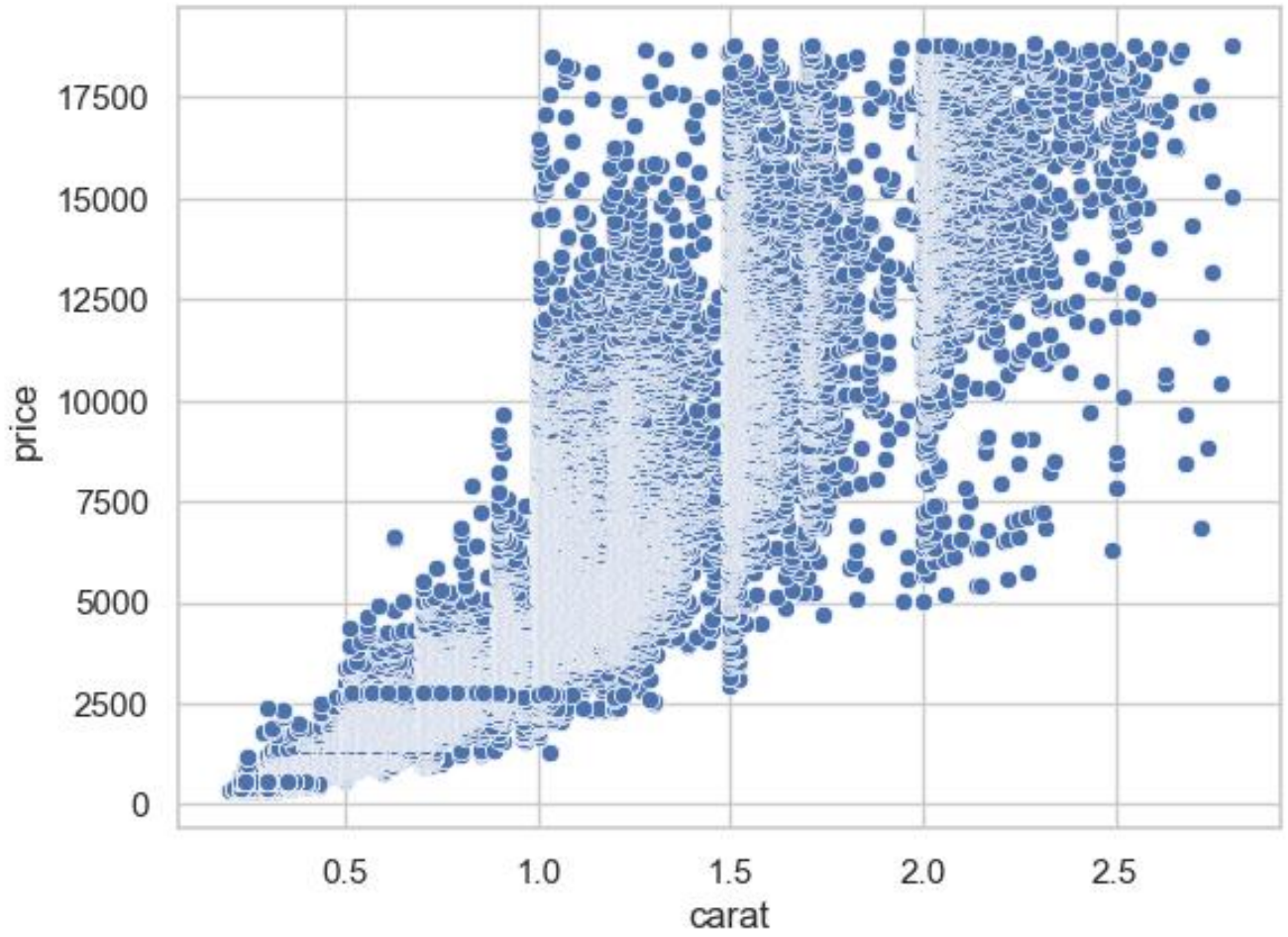
Consider again **`diamonds`**, where it was difficult to understand the relationship between diamond quality (**`cut`**) and **`price`**.

Reasoning: **`cut`** & **`carat`**, and **`carat`** & **`price`** are tightly related.

Use a model to remove the very strong relationship between **`price`** and **`carat`** to further explore the subtleties that remain.



The relationship between price and carat

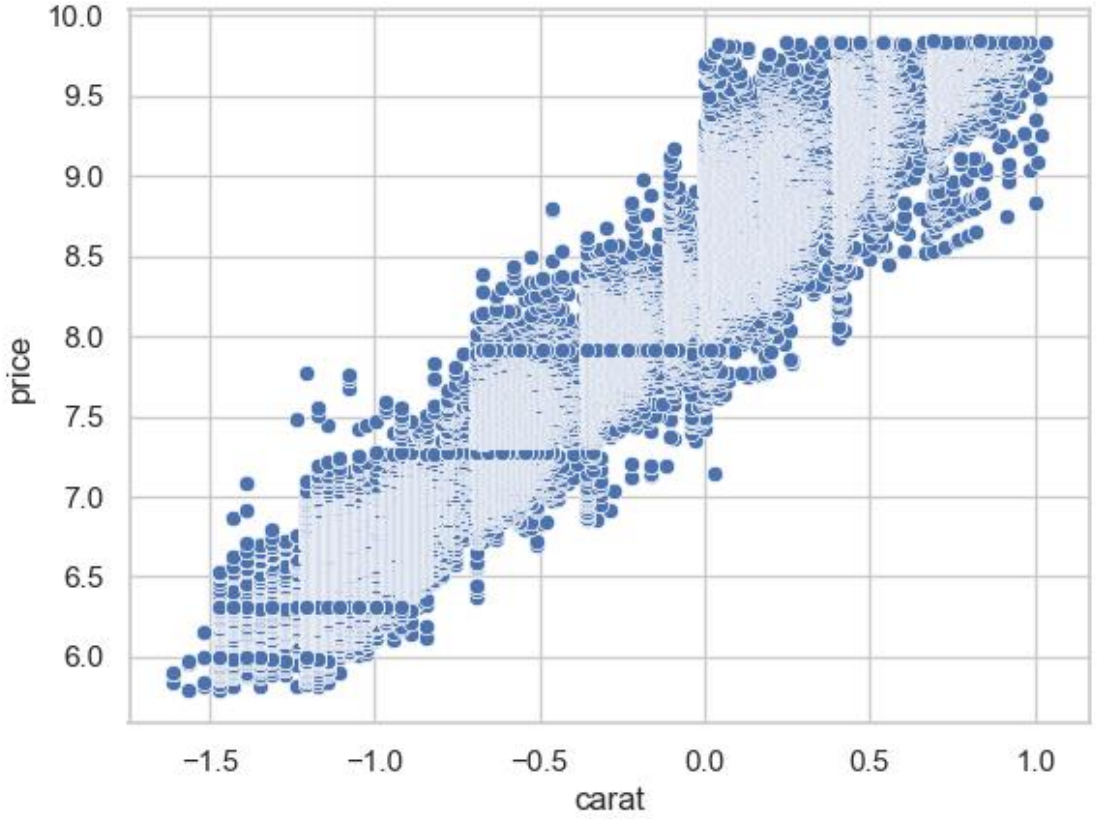


We wish model the relationship of **`price`** and **`carat`**.

However, the direct relationship of **`price`** and **`carat`** is not linear.

Performing a logarithmic transform to make more linear

```
sns.scatterplot(data=daiamonds_f,
                x=np.log(daiamonds_f.carat),
                y=np.log(daiamonds f.price))
```



After transforming both features with a natural logarithm, the relationship looks more linear, and so it would be appropriate to model the relationship with a linear function.

Modeling diamond price as a function of weight

We can use simple linear regression (a line) to predict the logarithm of diamond price from the logarithm of diamond weight.

This can be accomplished by fitting a **linear model** using the `statsmodels.OLS()` function.

```
import statsmodels.api as sm

# Prepare the data for modelin
X = diamonds['carat']
y = diamonds['price']

# Fit the linear regression mo
model = sm.OLS(y, X).fit()

# Print out the statistics
model.summary()
```

Out[145]:

OLS Regression Results

Dep. Variable:	price	R-squared (uncentered):	0.881
Model:	OLS	Adj. R-squared (uncentered):	0.881
Method:	Least Squares	F-statistic:	4.004e+05
Date:	Thu, 15 Feb 2024	Prob (F-statistic):	0.00
Time:	15:12:31	Log-Likelihood:	-4.8462e+05
No. Observations:	53940	AIC:	9.692e+05
Df Residuals:	53939	BIC:	9.692e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
carat	5666.2701	8.955	632.750	0.000	5648.718	5683.822

Omnibus:	26109.286	Durbin-Watson:	0.344
Prob(Omnibus):	0.000	Jarque-Bera (JB):	146424.223
Skew:	2.340	Prob(JB):	0.00
Kurtosis:	9.576	Cond. No.	1.00

Linear Model Fit