Intermediate Data Visualization with Seaborn

May 4, 2020

Seaborn is an important Python library for data analysis and statistical visualization. Seaborn integrates with the Python data science landscape by leveraging matplotlib and integrating with pandas.

Codes are simple in Seaborn and their output gives strong statistical insights into data.

In this document I will be using various datasets and constrcuing almost all the plot types offered by Seaborn ranging from KDE, regression plots, and categorical plots like matrix plots, factor plots, lm plots, pair plots and joint plots, so on and so forth.

I will also be using various seaborn styles, colors and techniques. Going through the seaborn library in detail has made me well versed to statistically analyze any data visually using visualization techniues.

[37]:
```
# importing all modules
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[38]:
```
grant_file = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data␣
↪visualization with seaborn/Data/grant.csv'
```

[101]:
```
# Reading in the DataFrame
df = pd.read_csv(grant_file)
df.head()
```

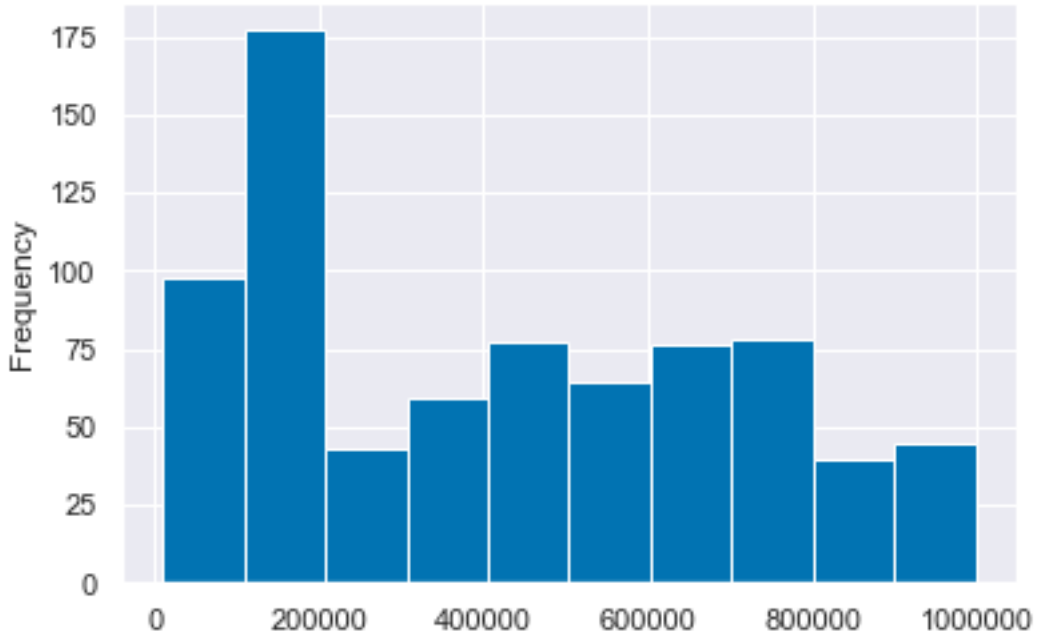
[101]:

	Unnamed: 0	School Name	City	State	\
0	0	HOGARTH KINGEEKUK MEMORIAL SCHOOL	SAVOONGA	AK	
1	1	AKIACHAK SCHOOL	AKIACHAK	AK	
2	2	GAMBELL SCHOOL	GAMBELL	AK	
3	3	BURCHELL HIGH SCHOOL	WASILLA	AK	
4	4	AKIAK SCHOOL	AKIAK	AK	

	District Name	Model Selected	Award_Amount	\
0	BERING STRAIT SCHOOL DISTRICT	Transformation	471014	
1	YUPIIT SCHOOL DISTRICT	Transformation	520579	
2	BERING STRAIT SCHOOL DISTRICT	Transformation	449592	
3	MATANUSKA-SUSITNA BOROUGH SCHOOL DISTRICT	Transformation	641184	
4	YUPIIT SCHOOL DISTRICT	Transformation	399686	

```
  Region
0   West
1   West
2   West
3   West
4   West
```

[94]:
```
# Using the pandas' plot.hist() function to plot a histogram of the
↪Award_Amount column.

# Displaying pandas histogram
df['Award_Amount'].plot.hist()
plt.show()

# Clearing out the pandas histogram
plt.clf()
```

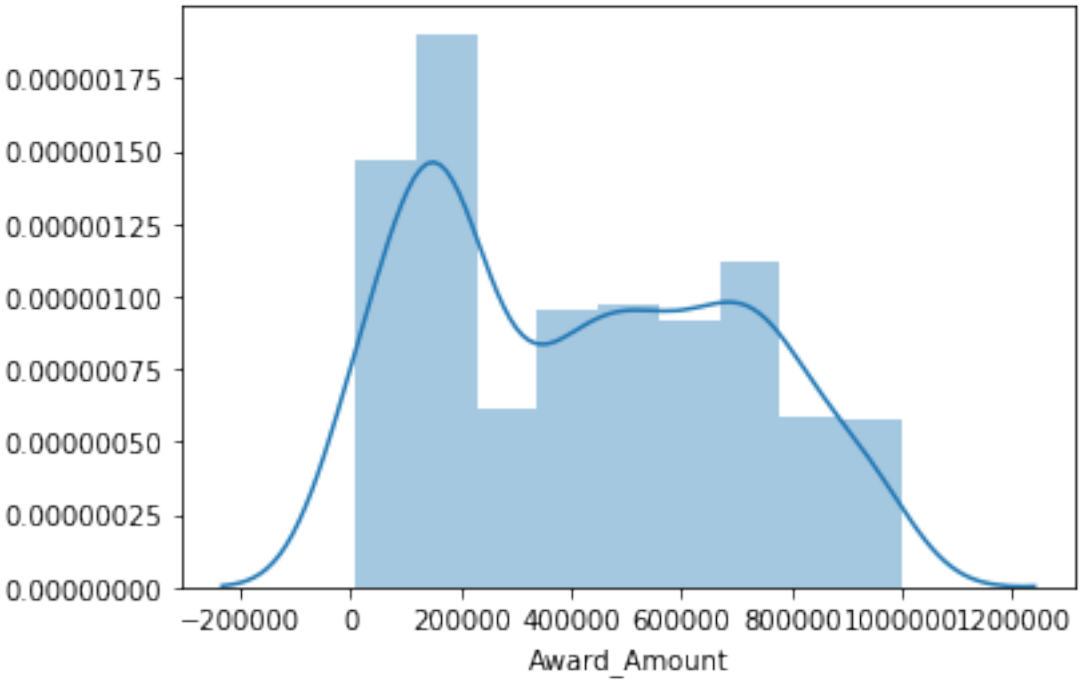



```
<Figure size 432x288 with 0 Axes>
```

By using highly customizable dist plot (univariate) in Seaborn we can understand how the values of a variable are distributed.

[41]:
```
# Using Seaborn's distplot() function to plot a distribution plot of the same
↪column.
```

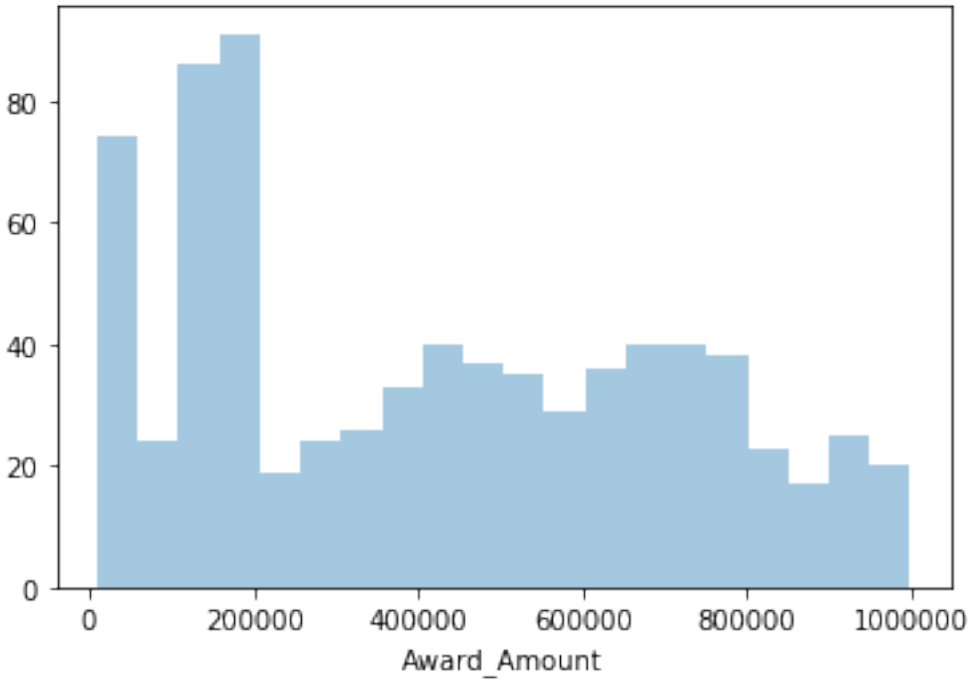
```
# Displaying a Seaborn distplot
sns.distplot(df['Award_Amount'])
plt.show()

# Clearing the distplot
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

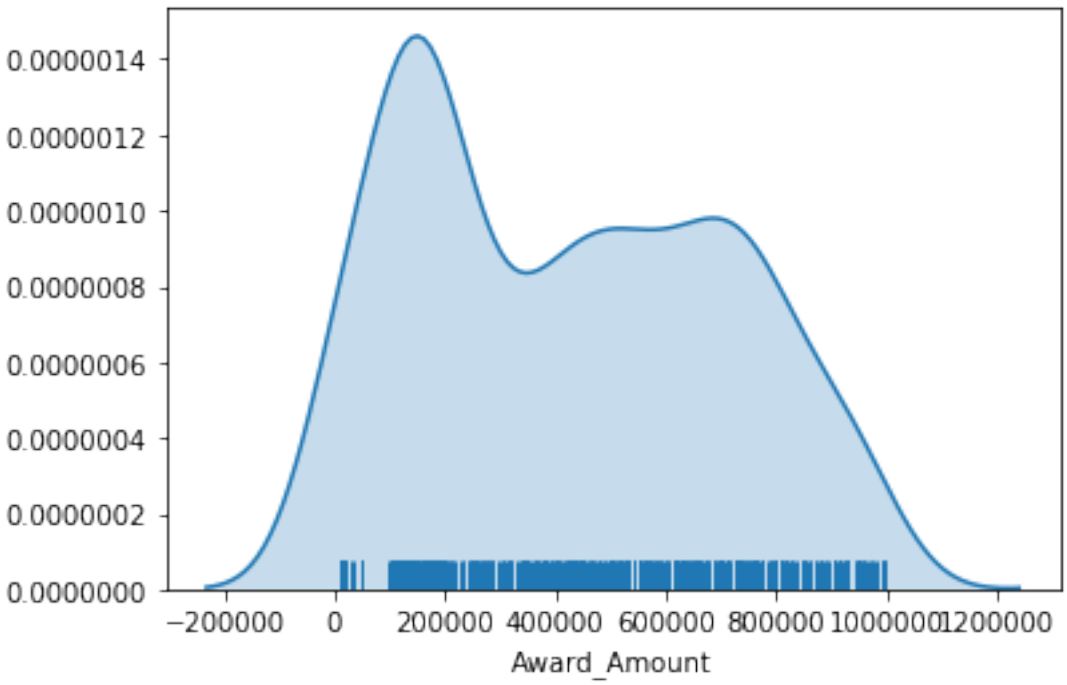
Notice how the pandas and Seaborn plots are complementary. They both show the distribution of data in different formats

[42]:
```
# Creating a distplot
sns.distplot(df['Award_Amount'],
             kde=False,
             bins=20)

# Displaying the plot
plt.show()
```



The distplot() function can be configured with multiple different arguments. By disabling the KDE output, you have created a histogram.

[43]:
```
# Creating a distplot of the Award Amount
sns.distplot(df['Award_Amount'],
             hist=False,
             rug=True,
             kde_kws={'shade':True})

# Plotting the results
plt.show()
```



There are a large group of award amounts < $400K.

[75]:
```
file_path = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data␣
↪visualization with seaborn/Data/insurance.csv'
```

[76]:
```
df1 = pd.read_csv(file_path)
df1.head()
```

[76]:

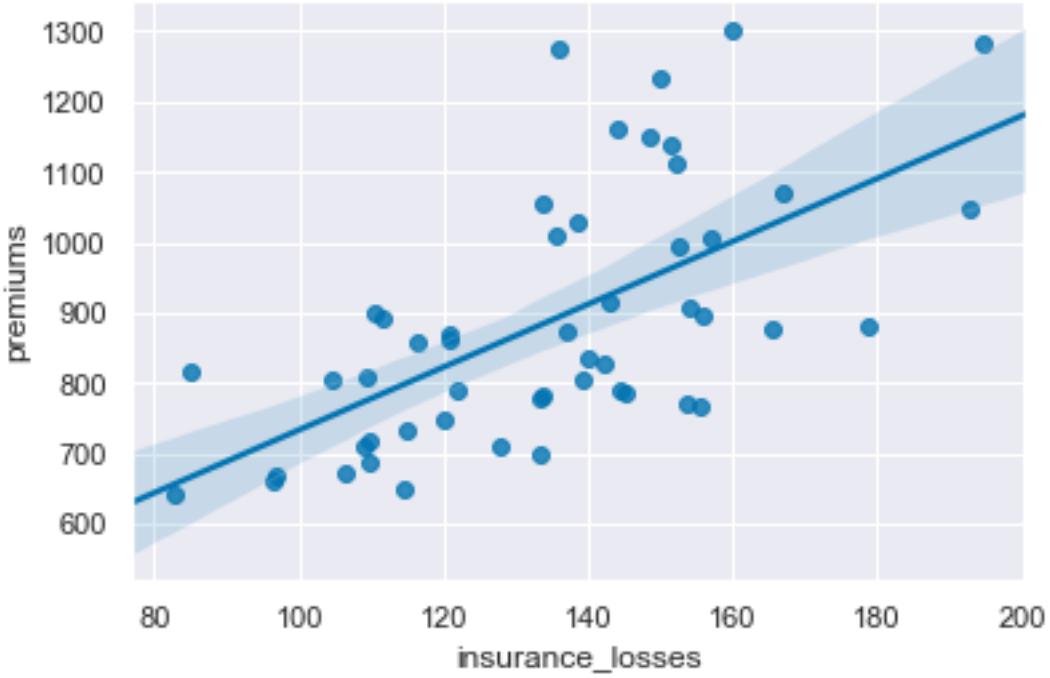
	State	fatal_collisions	fatal_collisions_speeding	fatal_collisions_alc	fatal_collisions_not_distracted	fatal_collisions_no_hist	premiums	insurance_losses	Region
0	Alabama	18.8	39	30	96	80	784.55	145.08	South
1	Alaska	18.1	41	25	90	94	1053.48	133.93	West
2	Arizona	18.6	35	28	84				
3	Arkansas	22.4	18	26	94				
4	California	12.0	35	28	91				

```
2                        96    899.47            110.35   West
3                        95    827.34            142.39  South
4                        89    878.41            165.63   West
```

Conducting Regression Analysis through visualization

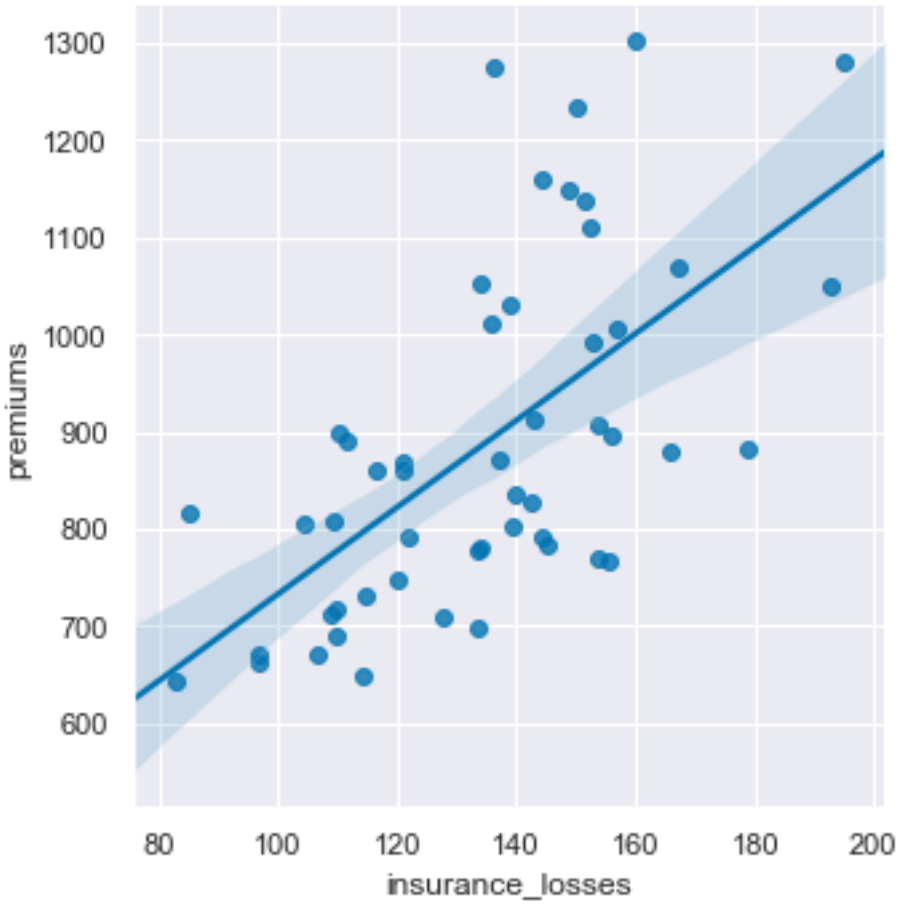
Regression analysis is bivariate because it looks at relationship between two variables.

[77]:
```
# Creating a regression plot of premiums vs. insurance_losses
sns.regplot(data=df1,
            x="insurance_losses",
            y="premiums")

# Displaying the plot
plt.show()
```



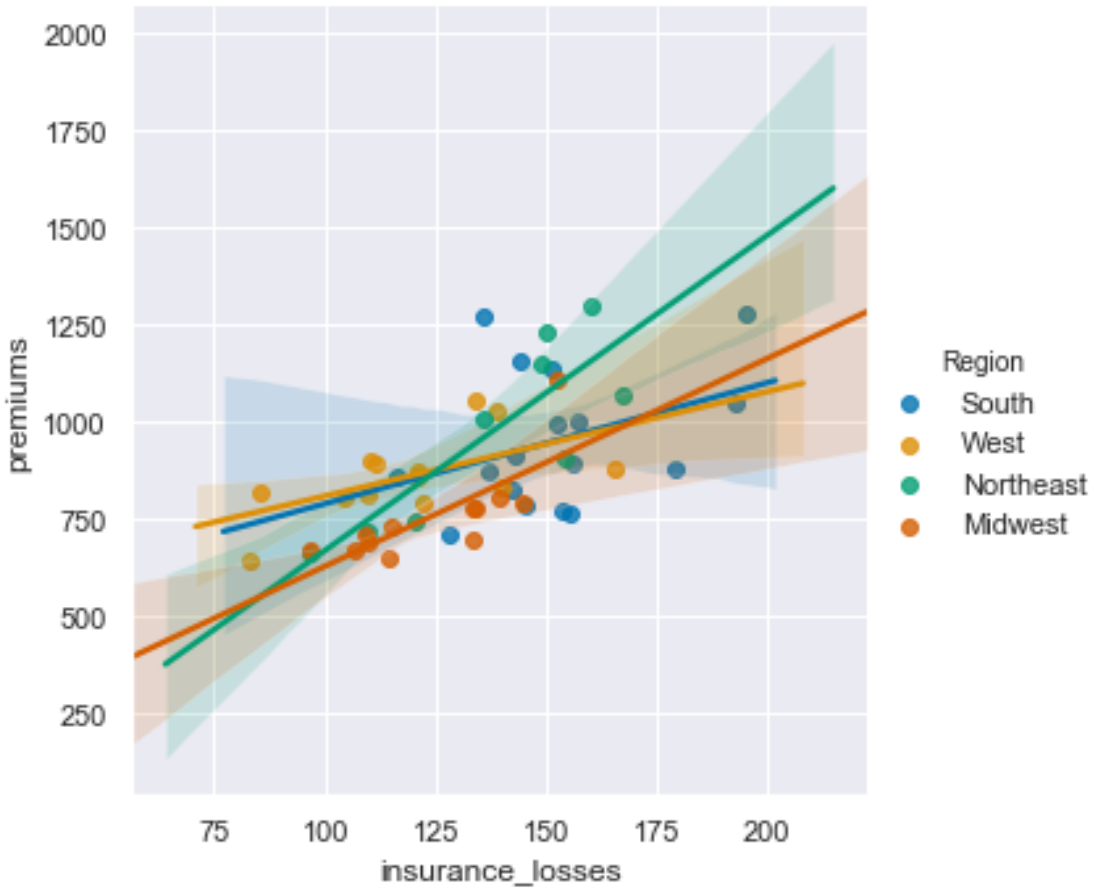
The regression line hints that there will be an increase in premiums as insurance losses increase. The lower level regplot() and higher level lmplot() are similar. They produce the similar output. However, the lmplot is much more powerful. The use of hue and columns is a powerful concept that is present throughout many of Seaborn's functions. The use of plotting multiple graphs while changing a single variable is often called faceting. Faceting can be accomplished by using lmplot() function. The base function is very similar to regplot() but it provides much more power by allowing us to add additional information using columns, colors or rows. Proceeding, I will be using the regplot() and lmplot() functions to analyze different datasets.

[78]:
```
# Creating an lmplot of premiums vs. insurance_losses
sns.lmplot(data=df1,
           x="insurance_losses",
           y="premiums")

# Displaying the second plot
plt.show()
```



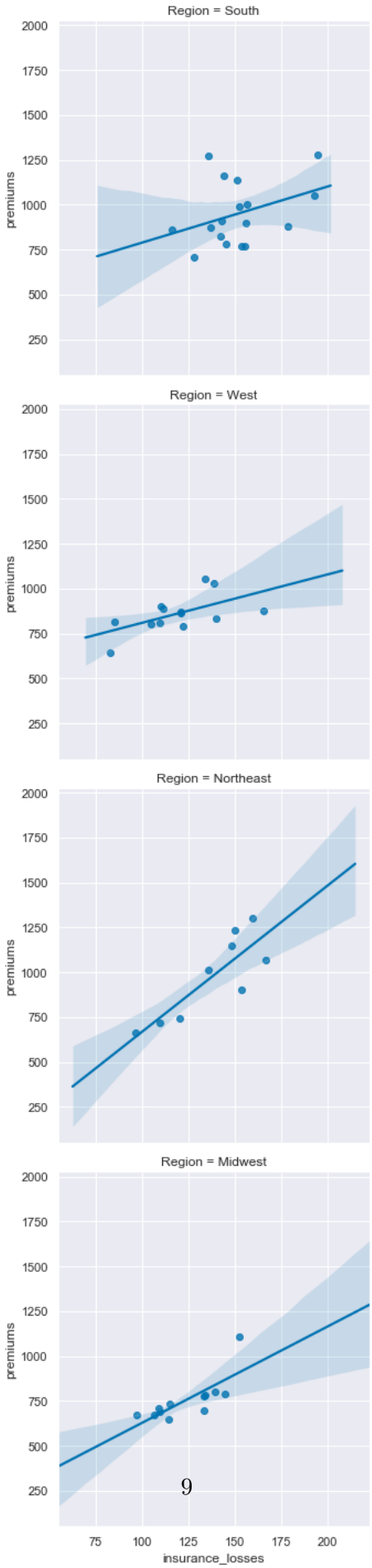
The output looks similar.

[79]:
```
# Creating a regression plot using hue
sns.lmplot(data=df1,
           x="insurance_losses",
           y="premiums",
           hue="Region")

# Showing the results
plt.show()
```



The results are a bit difficult to read. Maybe using multiple lines is not the best approach.

[80]:
```
# Create a regression plot with multiple rows
sns.lmplot(data=df1,
           x="insurance_losses",
           y="premiums",
           row="Region")

# Show the plot
plt.show()
```

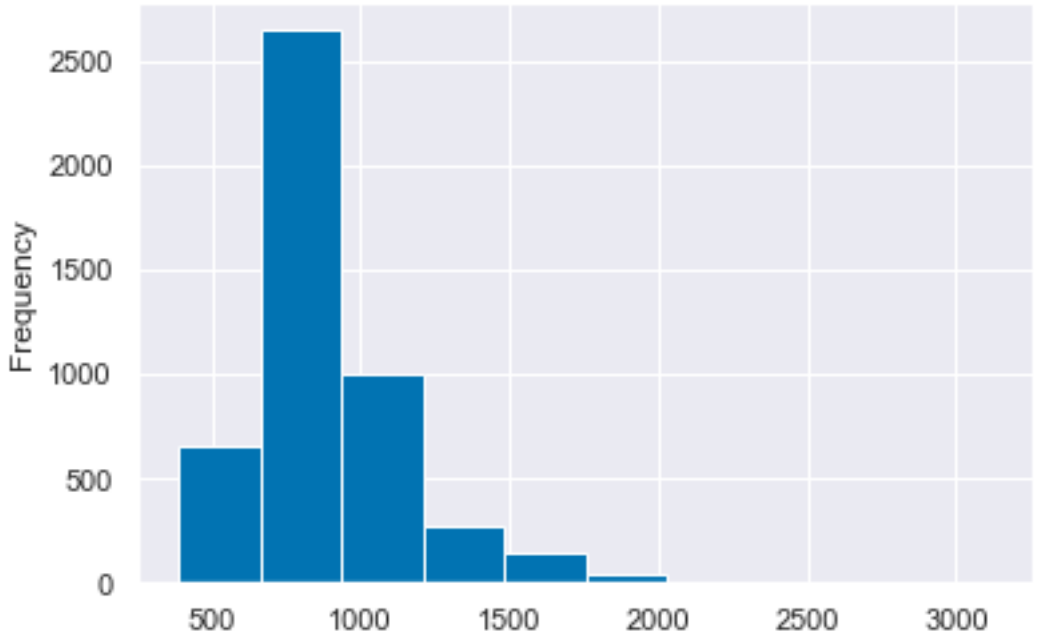

Region = South
Region = West
Region = Northeast
Region = Midwest
premiums
2000
1750
1500
1250
1000
750
500
250
75
100
125
150
175
200
insurance_losses

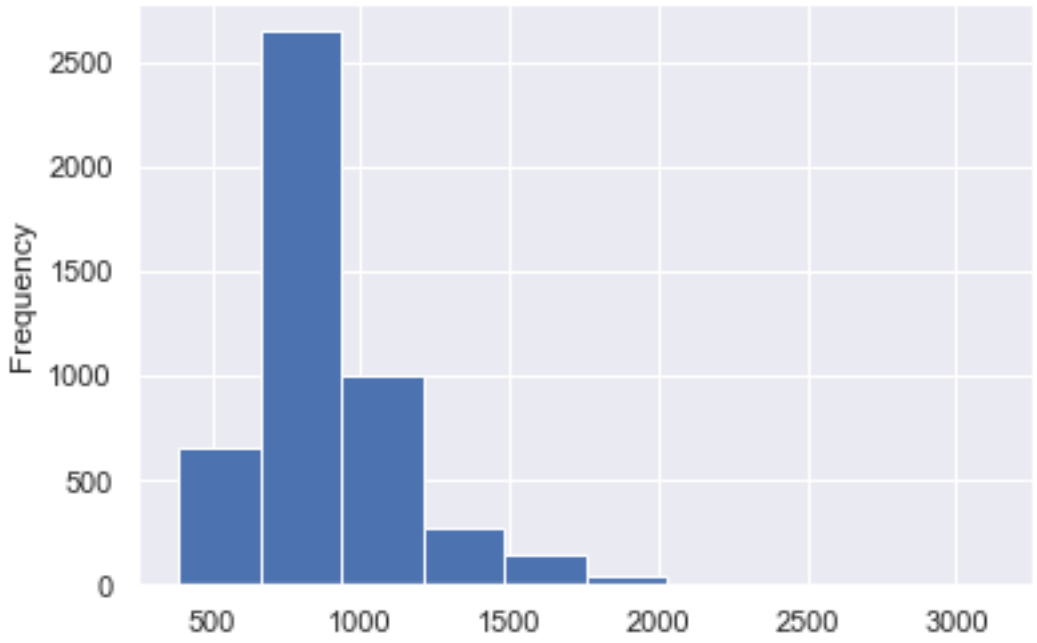
Faceting the data across multiple rows or columns can be a good way to see variable interactions in the data. The lmplot function supports plotting regression data by column, row and hue. This concept is used repeatedly throughout Seaborn.

[60]:
```
fp = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data visualization␣
 ↪with seaborn/Data/rent.csv'
```

[176]:
```
df2 = pd.read_csv(fp)
```

[82]:
```
# Plotting the pandas histogram
df2['fmr_2'].plot.hist()
plt.show()
plt.clf()

# Setting the default seaborn style
sns.set()

# Plotting the pandas histogram again
df2['fmr_2'].plot.hist()
plt.show()
plt.clf()
```



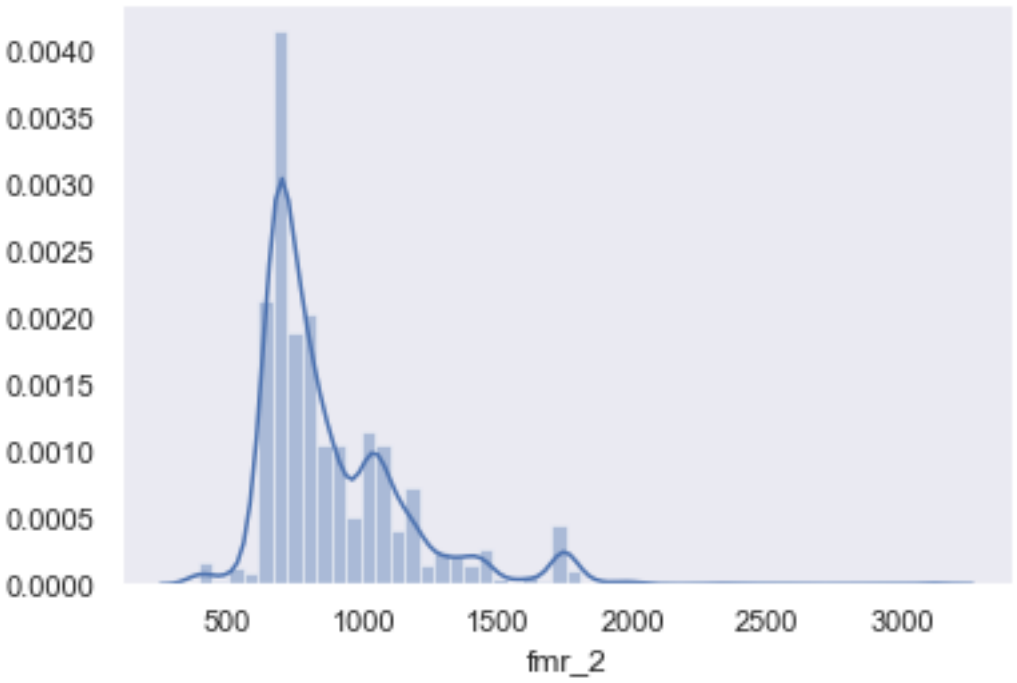


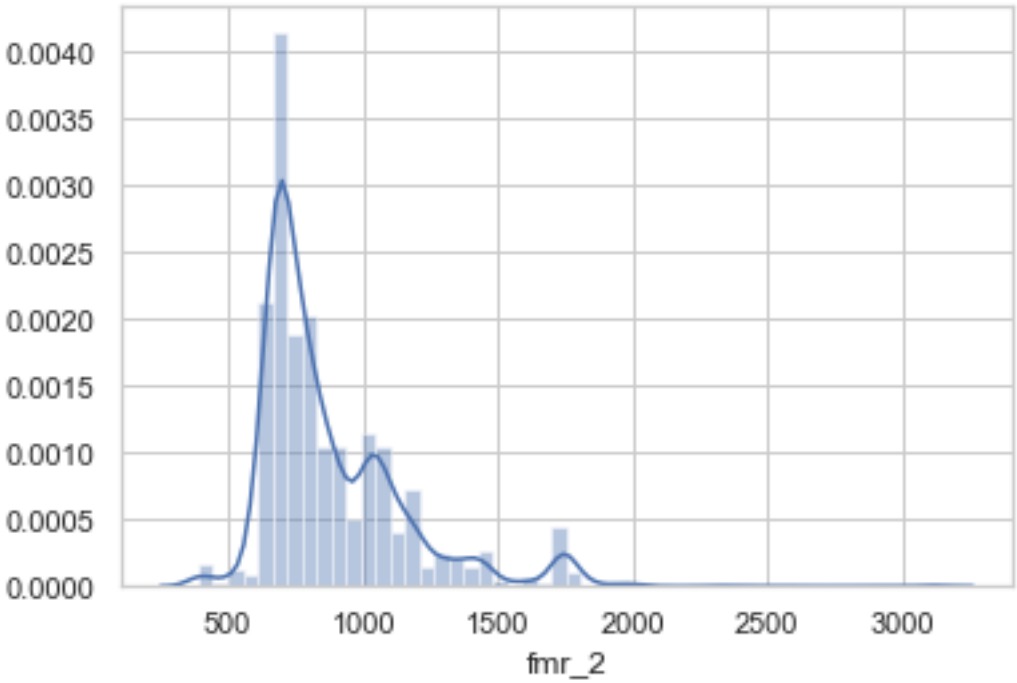

```
<Figure size 432x288 with 0 Axes>
```

Comparing styles

Seaborn supports setting different styles that can control the aesthetics of the final plot. I will plot the same data in two different styles in order to see how the styles change the output.

[83]:
```
# Plotting with a dark style
sns.set_style('dark')
sns.distplot(df2['fmr_2'])
plt.show()

# Clearing the figure
plt.clf()

# Plotting with a dark style
sns.set_style('whitegrid')
sns.distplot(df2['fmr_2'])
plt.show()

# Clearing the figure
plt.clf()
```





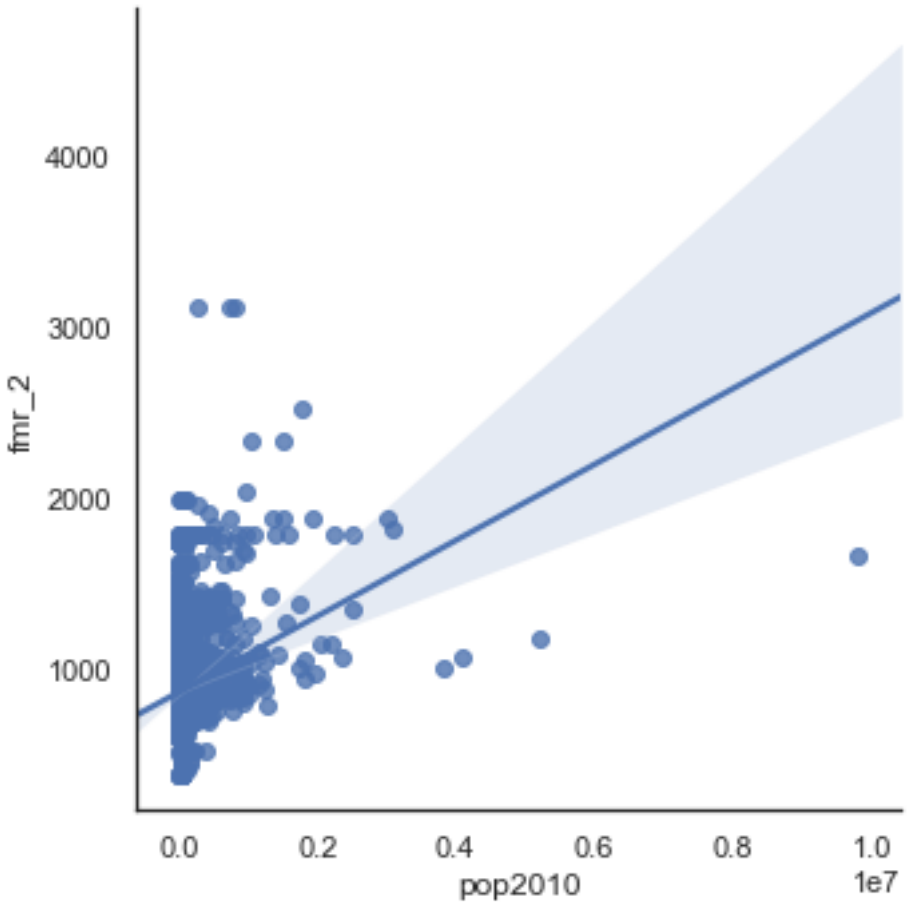

```
<Figure size 432x288 with 0 Axes>
```

Seaborn's styles provide a quick and easy way to alter the visualizations in a consistent and visually appealing manner.

Removing spines

In general, visualizations should minimize extraneous markings so that the data speaks for itself. Seaborn allows us to remove the lines on the top, bottom, left and right axis, which are often called spines.

[84]:
```
# Setting the style to white
sns.set_style('white')

# Creating a regression plot
sns.lmplot(data=df2,
           x='pop2010',
           y='fmr_2')

# Removing the spines
sns.despine(top=True, right=True)

# Showing the plot and clear the figure
plt.show()
plt.clf()
```



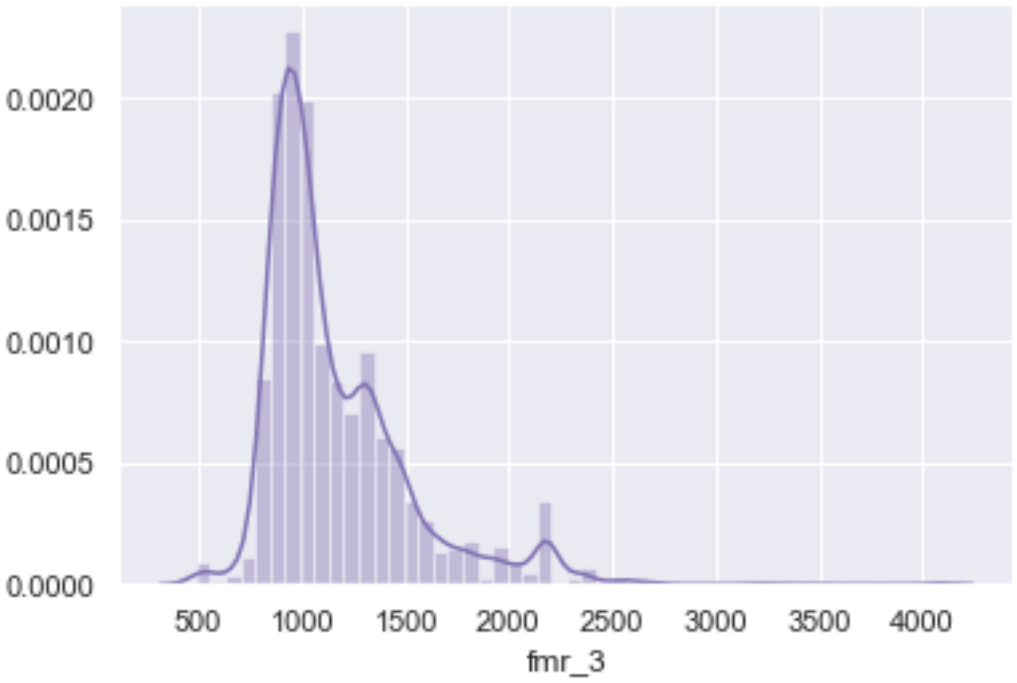

```
<Figure size 432x288 with 0 Axes>
```

Removing the spines lets the data speak for itself.

Matplotlib color codes

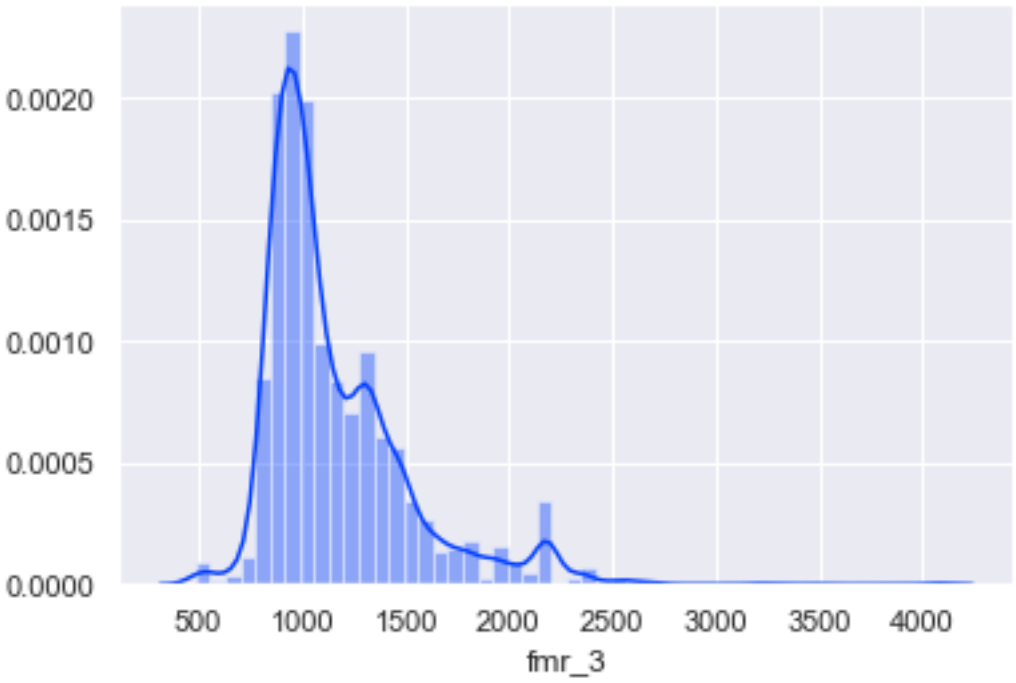
Seaborn offers several options for modifying the colors of your visualizations. The simplest approach is to explicitly state the color of the plot. A quick way to change colors is to use the standard matplotlib color codes.

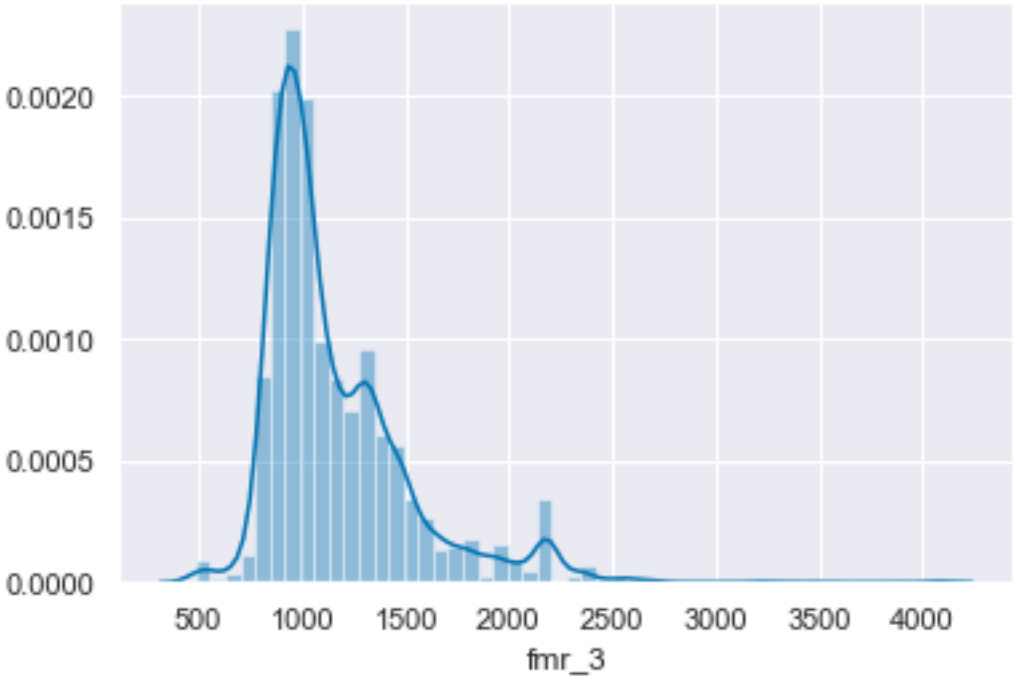
[85]:
```
# Setting style, enable color code, and create a magenta distplot
sns.set(color_codes=True)
sns.distplot(df2['fmr_3'], color='m')

# Showing the plot
plt.show()
```



This is a quick way to modify the colors of a plot.

[86]:
```
# Looping through differences between bright and colorblind palettes
for p in ['bright', 'colorblind']:
    sns.set_palette(p)
    sns.distplot(df2['fmr_3'])
    plt.show()

    # Clearing the plots
    plt.clf()
```





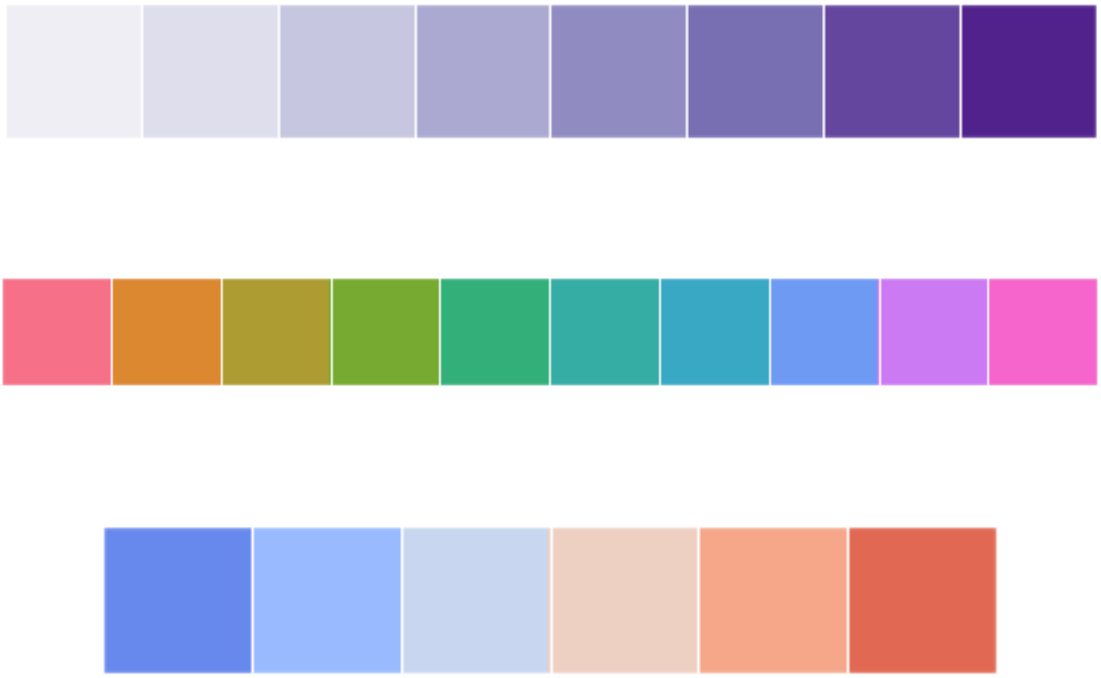

```
<Figure size 432x288 with 0 Axes>
```

Using the default color blind palette is a good option for making sure that the visualizations are easy to read for individuals with degrees of color blindness.

Creating Custom Palettes

Choosing a cohesive palette that works for the data can be time consuming. Fortunately, Seaborn has several functions that allows to create our own custom sequential, categorical, or diverging palettes. Seaborn also makes it easy to see the palettes by using the palplot() function.

[69]:
```
# Creating and displaying a Purples sequential palette containing 8 colors.
sns.palplot(sns.color_palette('Purples', 8))
plt.show()

# Creating and displaying a palette with 10 colors using the husl system.
sns.palplot(sns.color_palette('husl', 10))
plt.show()

# Creating and displaying a diverging palette with 6 colors coolwarm.
sns.palplot(sns.color_palette('coolwarm', 6))
plt.show()
```



Color palette possibilities are limitless. Using these helper functions allows to create unique and visually appealing plots that will stand out and get your points across.

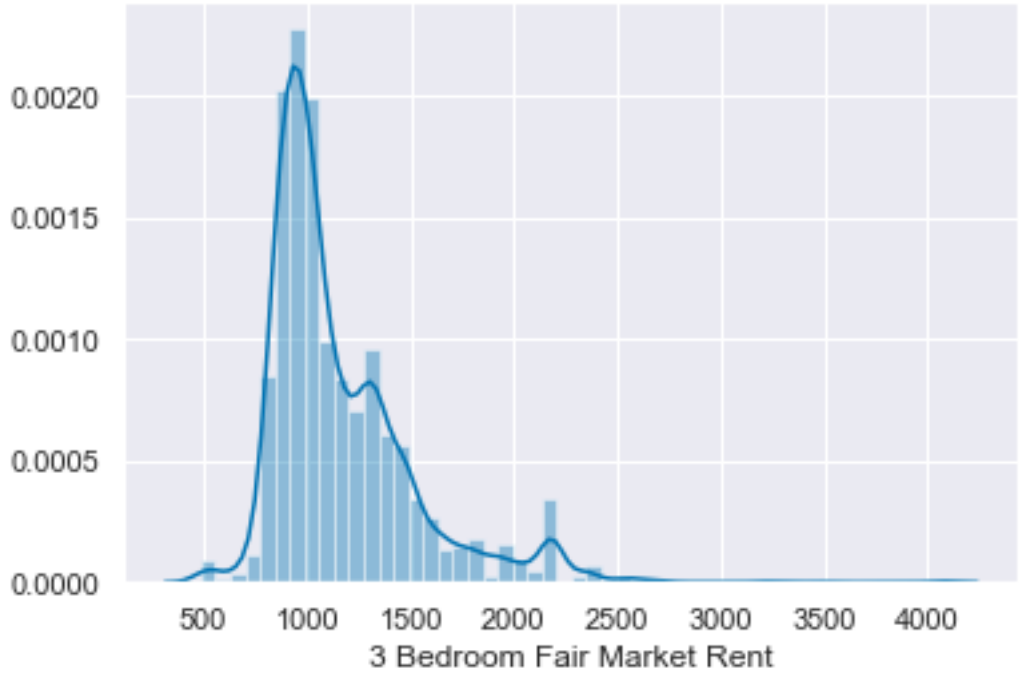
Using matplotlib axes

Seaborn uses matplotlib as the underlying library for creating plots. Most of the time, we can

use the Seaborn API to modify the visualizations but sometimes it is helpful to use matplotlib's functions to customize the plots. The most important object in this case is matplotlib's axes.

Once we have an axes object, we can perform a lot of customization of the plot.

[87]:
```
# Creating a figure and axes
fig, ax = plt.subplots()

# Plotting the distribution of data
sns.distplot(df2['fmr_3'], ax=ax)

# Creating a more descriptive x axis label
ax.set(xlabel="3 Bedroom Fair Market Rent")

# Showing the plot
plt.show()
```

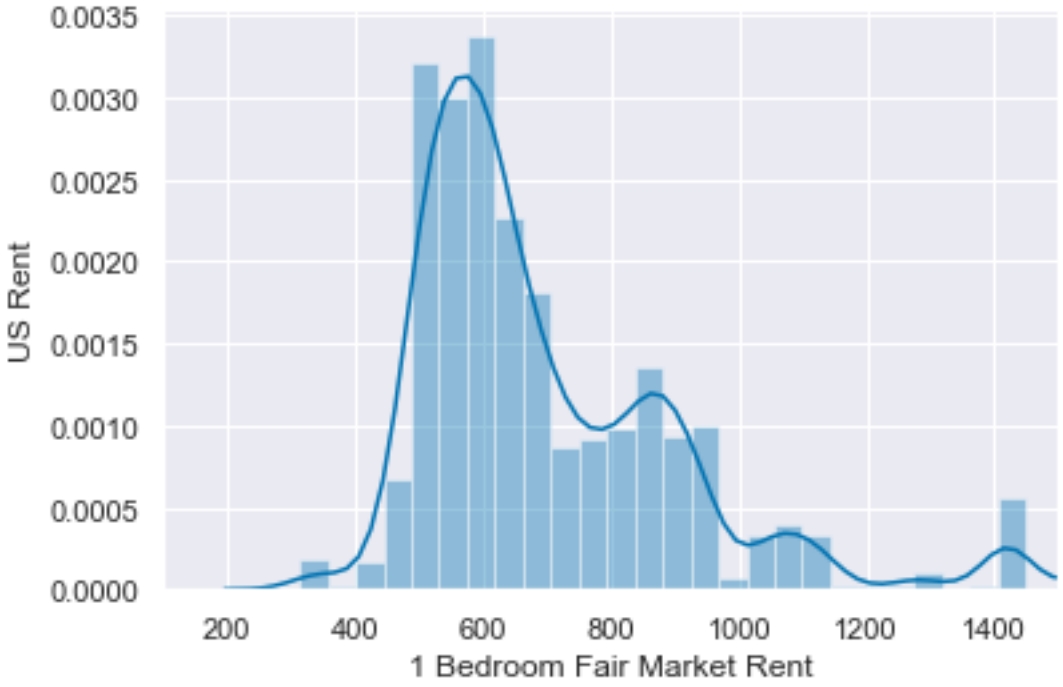


Additional plot customizations

The matplotlib API supports many common customizations such as labeling axes, adding titles, and setting limits. Let's do some more customization.

[88]:
```
# Creating a figure and axes
fig, ax = plt.subplots()

# Plotting the distribution of 1 bedroom rents
```

```
sns.distplot(df2['fmr_1'], ax=ax)

# Modifying the properties of the plot
ax.set(xlabel="1 Bedroom Fair Market Rent",
       xlim=(100,1500),
       ylabel="US Rent")

# Displaying the plot
plt.show()
```



Making these types of customizations to the plots can really improve their impact.

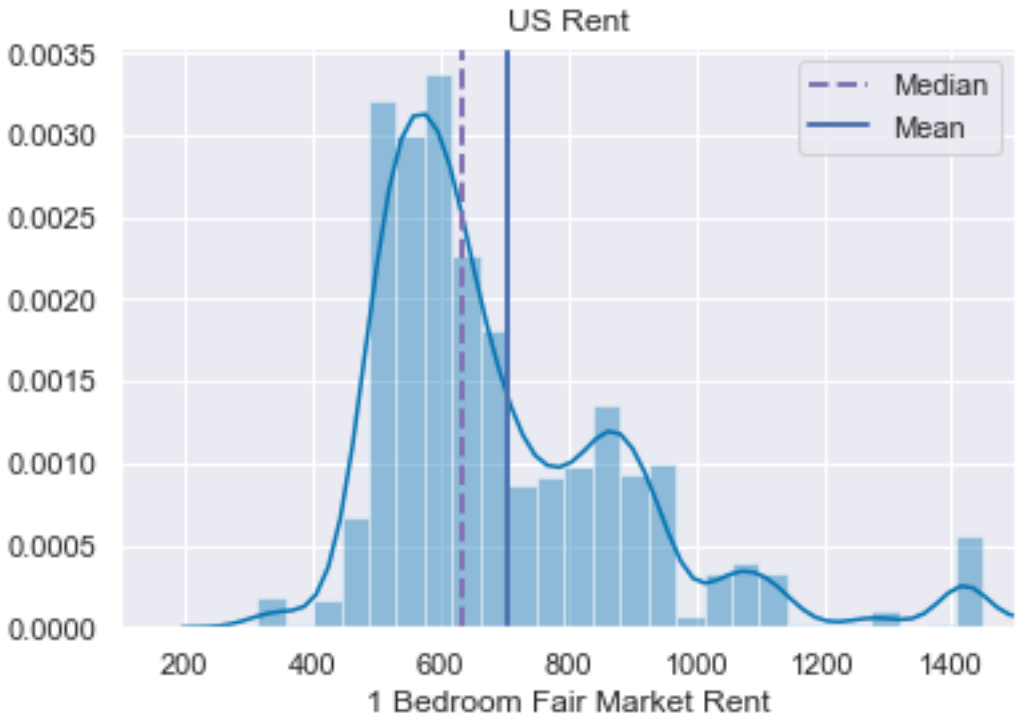
Adding annotations

Each of the enhancements we have covered can be combined together. In the following code I will annotate the distribution plot to include lines that show the mean and median rent prices.

For this example, the palette has been changed to bright using sns.set_palette()

[89]:
```
# Creating a figure and axes. Then plot the data
fig, ax = plt.subplots()
sns.distplot(df2['fmr_1'], ax=ax)

# Customizing the labels and limits
ax.set(xlabel="1 Bedroom Fair Market Rent", xlim=(100,1500), title="US Rent")
```

```
mean = df2['fmr_1'].mean()
median = df2['fmr_1'].median()

# Adding vertical lines for the median and mean
ax.axvline(x=median, color='m', label='Median', linestyle='--', linewidth=2)
ax.axvline(x=mean, color='b', label='Mean', linestyle='-', linewidth=2)

# Showing the legend and plot the data
ax.legend()
plt.show()
```

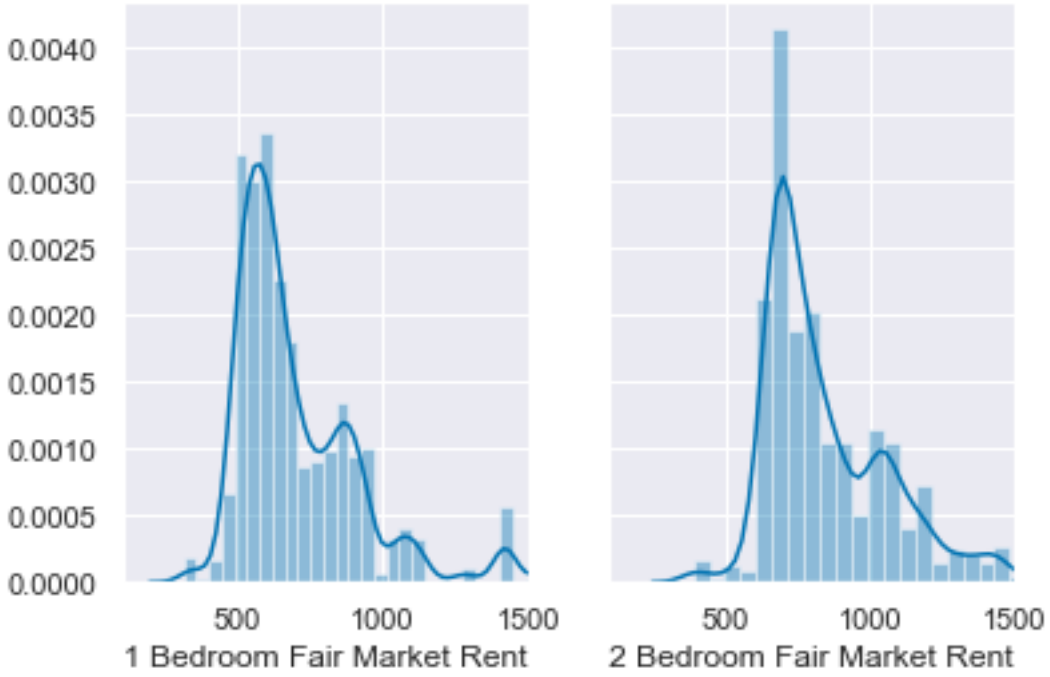


Multiple plots

Finally I will plot a comparison of the fair market rents for 1-bedroom and 2-bedroom apartments.

[90]:
```
# Creating a plot with 1 row and 2 columns that share the y axis label
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharey=True)

# Plotting the distribution of 1 bedroom apartments on ax0
sns.distplot(df2['fmr_1'], ax=ax0)
ax0.set(xlabel="1 Bedroom Fair Market Rent", xlim=(100,1500))

# Plotting the distribution of 2 bedroom apartments on ax1
sns.distplot(df2['fmr_2'], ax=ax1)
```

```
ax1.set(xlabel="2 Bedroom Fair Market Rent", xlim=(100,1500))

# Displaying the plot
plt.show()
```

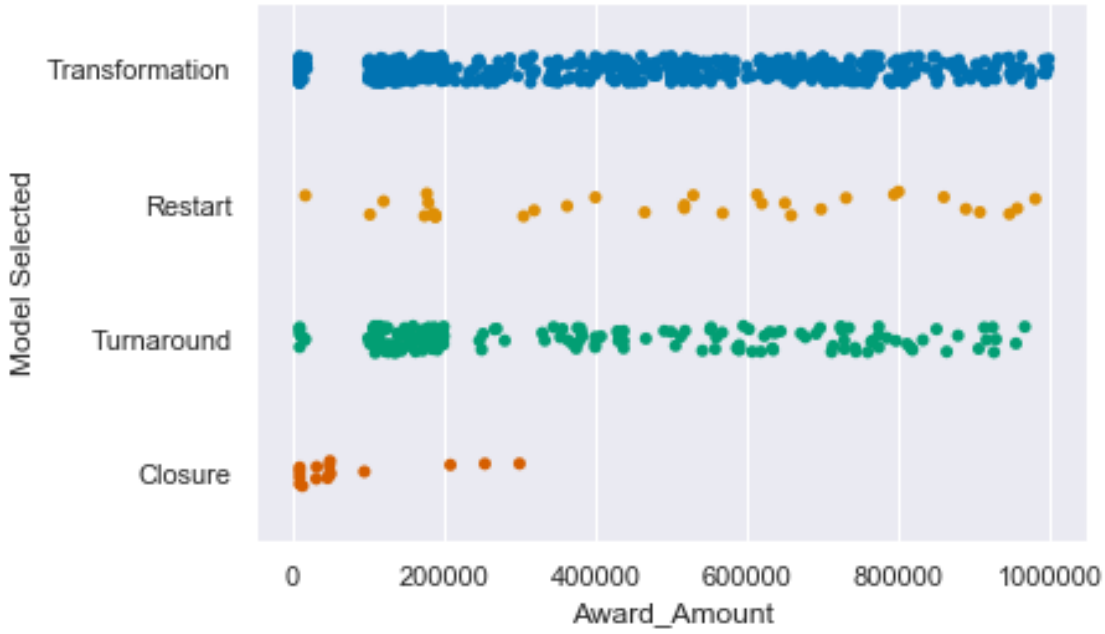


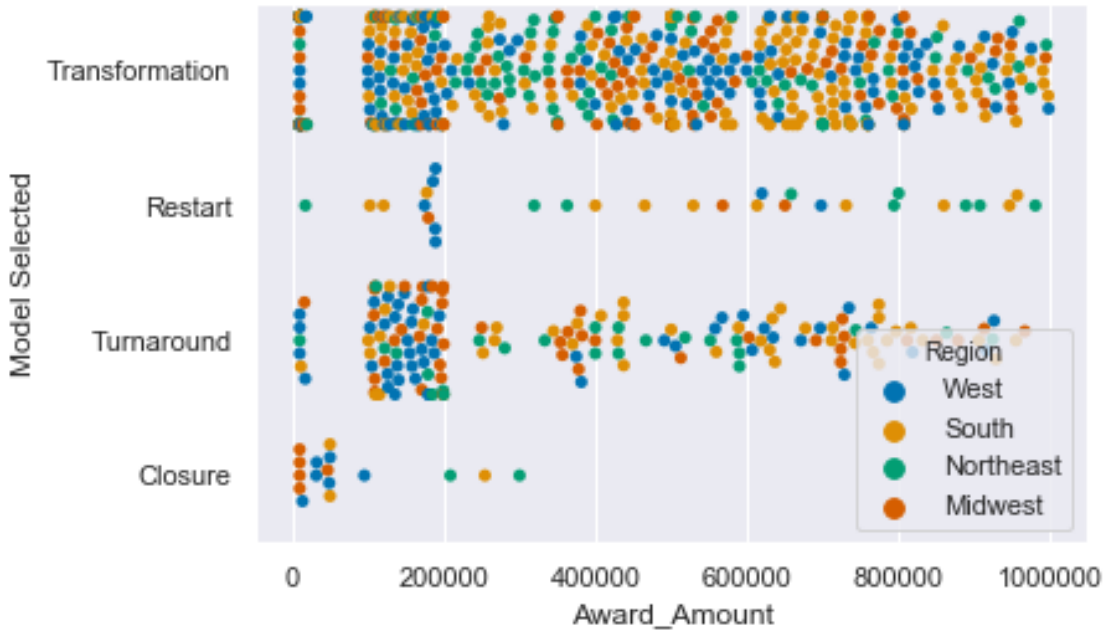
Stripplot() and swarmplot()

Many datasets have categorical data and Seaborn supports several useful plot types for this data. In the following example, I will continue to look at the 2010 School Improvement data and segment the data by the types of school improvement models used.

[95]:
```
# Creating a stripplot of the Award_Amount with the Model Selected on the y␣
↪axis with jitter enabled.

# Create the stripplot
sns.stripplot(data=df,
              x='Award_Amount',
              y='Model Selected',
              jitter=True)

plt.show()

# Creating a swarmplot() of the same data, but also include the hue by Region.

# Creating and display a swarmplot with hue set to the Region
```

```
sns.swarmplot(data=df,
          x='Award_Amount',
          y='Model Selected',
          hue='Region')

plt.show()
```





Categorical Plot Types

Categorical data is data which includes a limited or fixed number of values and is most useful when combined with numeric data. Seaborn breaks categorical data plots into three groups.

The first group includes the stripplot() and swarmplot(), which show all the individual observations on the plot.

The second category contains the familiar boxplot(), as well as the violinplot() and the lvplot(). These plots show an abstract representation of the categorical data.

The final group of plots show statistical estimates of the categorical variables.

The barplot() and pointplot() contain useful summaries of data.

The countplot() shows number of instances of each observation.

Stripplot() shows every observation in the dataset. In some cases, it can be difficult to see individual data points. We can use swarmplot() that uses a complex algorithm to place the observations in a manner where they do not overlap. The downside of this approach is that the swarmplot() does not scale well to large datasets.

The next category of plots show abstract representation of the data. A boxplot() is the most common of this type. This plot is used to show several measures related to the distribution of data, including the median, upper and lower quartiles, as well as outliers.

The violinplot() is a combination of kernel density plot and a box plot and can be suitable for providing an alternative view of the distribution of data. Because the plot uses a kernel density calculation it does not show all data points. This can be useful for displaying large datasets but it can be computationally intensive to create.

The final plot in the grouping is the lvplot() which stands for Letter Value plot. It scales more effectively to large datasets. The lvplot() is a hybrid between a boxplot() and violinplot() and is relatively quick to render and easy to interpret.

The final category of plots are statistical estimates of the data. The barplot() shows an estimate of the value as well as confidence interval.

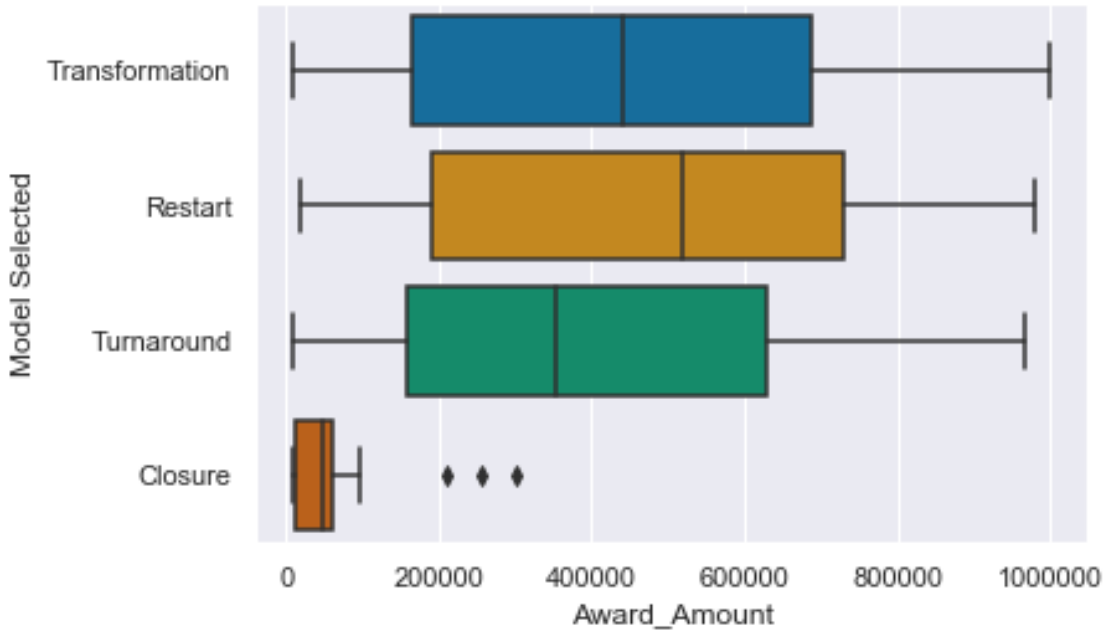
The pointplot() is similar to the barplot() in that it shows a summary measure and confidence interval. A pointplot() can be very useful for observing how values change across categorical values. The final categorical plot is the countplot() which displays the number of instances of each variable.

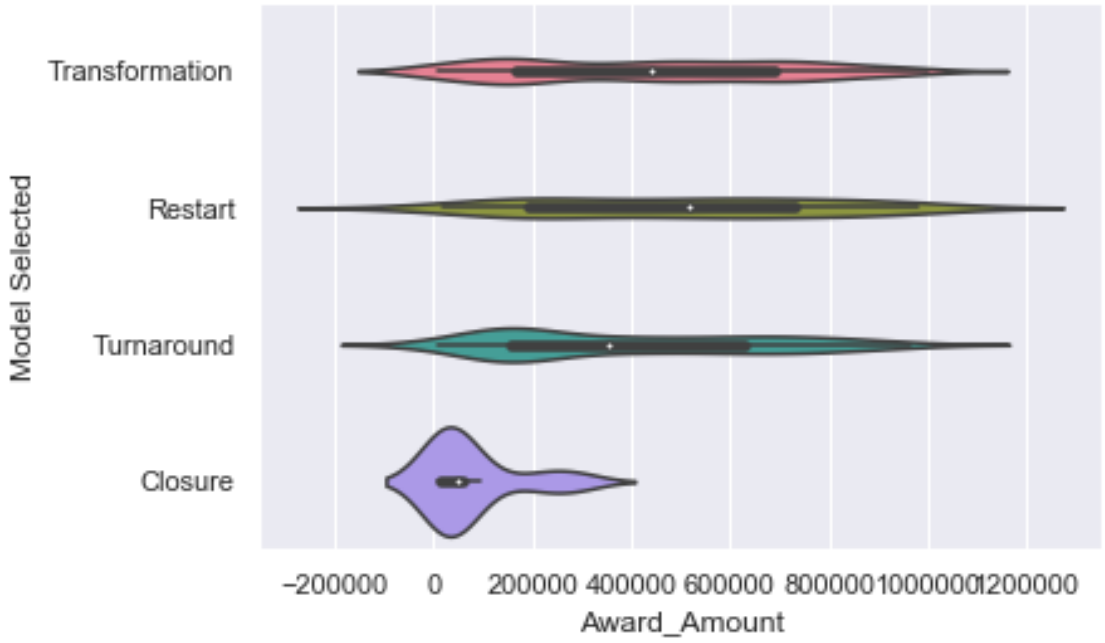
Boxplots, violinplots and lvplots

Seaborn's categorical plots also support several abstract representations of data. The API for each of these is the same so it is very convenient to try each plot and see if the data lends itself to one over the other.

I will use the color palette options to show how colors can easily be included in the plots.

[97]:
```
# Creating and displaying a boxplot of the data with Award_Amount on the x axis␣
 ↪and Model Selected on the y axis.

# Creating a boxplot
sns.boxplot(data=df,
            x='Award_Amount',
            y='Model Selected')

plt.show()
plt.clf()

# Creating and display a similar violinplot of the data, but use the husl␣
 ↪palette for colors.

# Creating a violinplot with the husl palette
sns.violinplot(data=df,
            x='Award_Amount',
            y='Model Selected',
            palette='husl')

plt.show()
plt.clf()

# Creating and display an lvplot using the Paired palette and the Region column␣
 ↪as the hue.
# Creating a lvplot with the Paired palette and the Region column as the hue
sns.lvplot(data=df,
            x='Award_Amount',
            y='Model Selected',
            palette='Paired',
            hue='Region')

plt.show()
plt.clf()
```

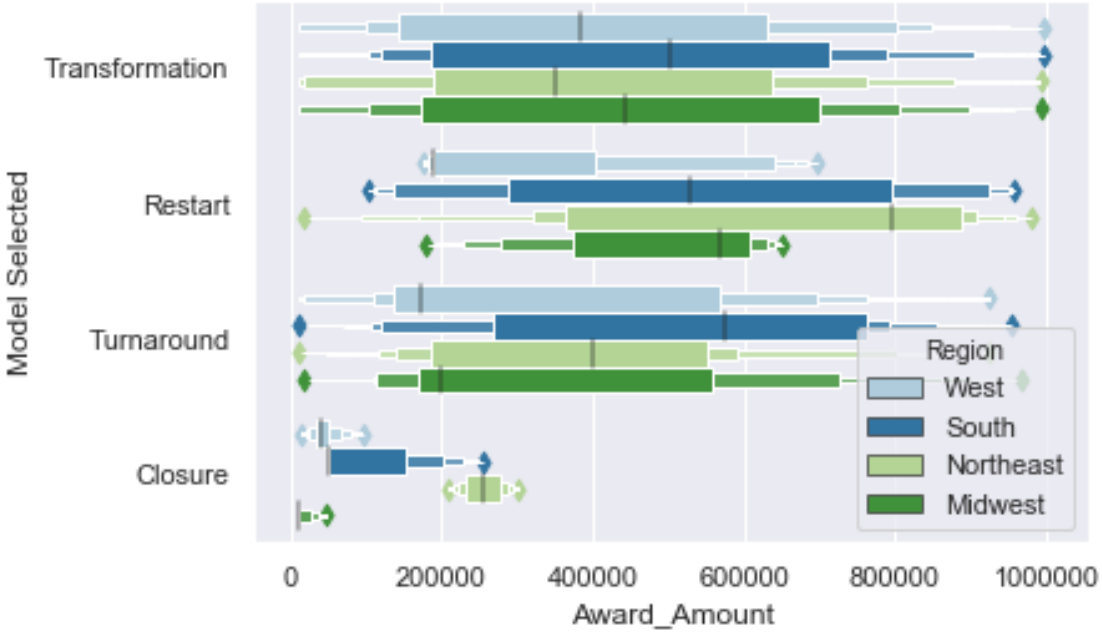





```
/opt/anaconda3/lib/python3.7/site-packages/seaborn/categorical.py:2612:
UserWarning: The `lvplot` function has been renamed to `boxenplot`. The original
name will be removed in a future release. Please update your code.
  warnings.warn(msg)
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
```

```
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
```

```
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
'x' & 'y'.  Please use a 2-D array with a single row if you really want to
specify the same RGB or RGBA value for all points.
```



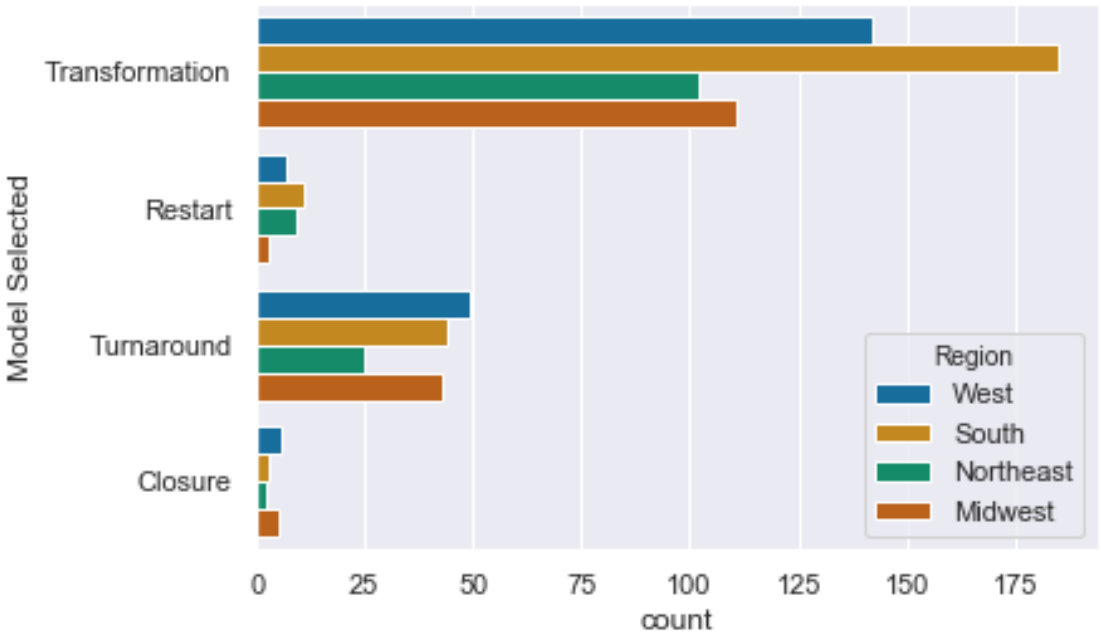

```
<Figure size 432x288 with 0 Axes>
```

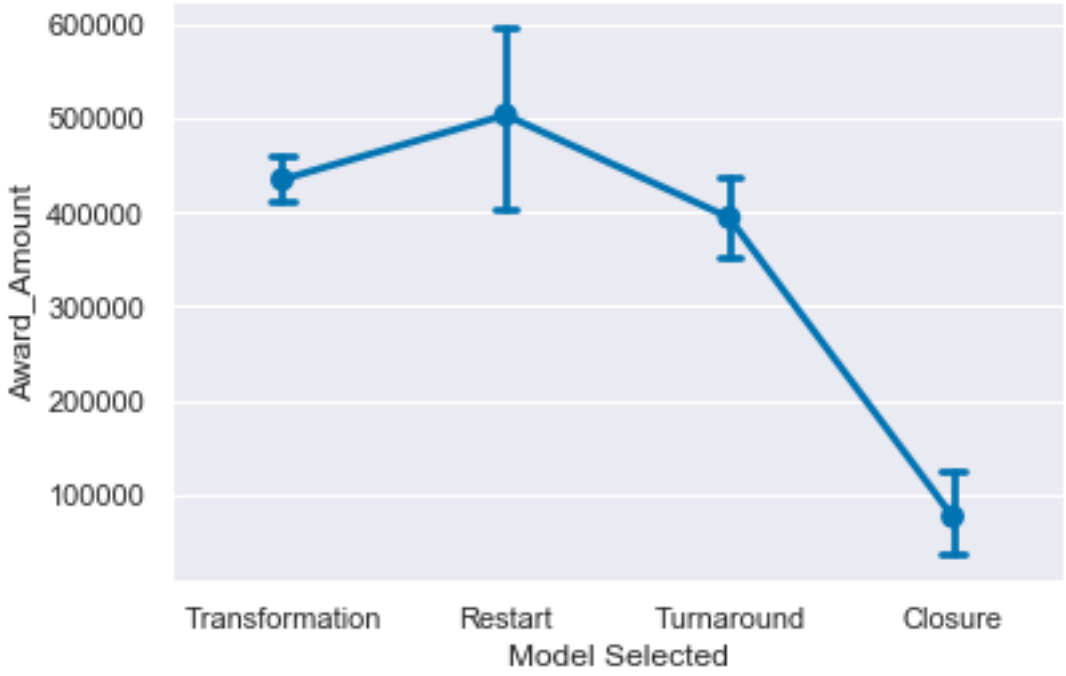
Seaborn's API makes it very simple to create multiple abstract representations of categorical data.

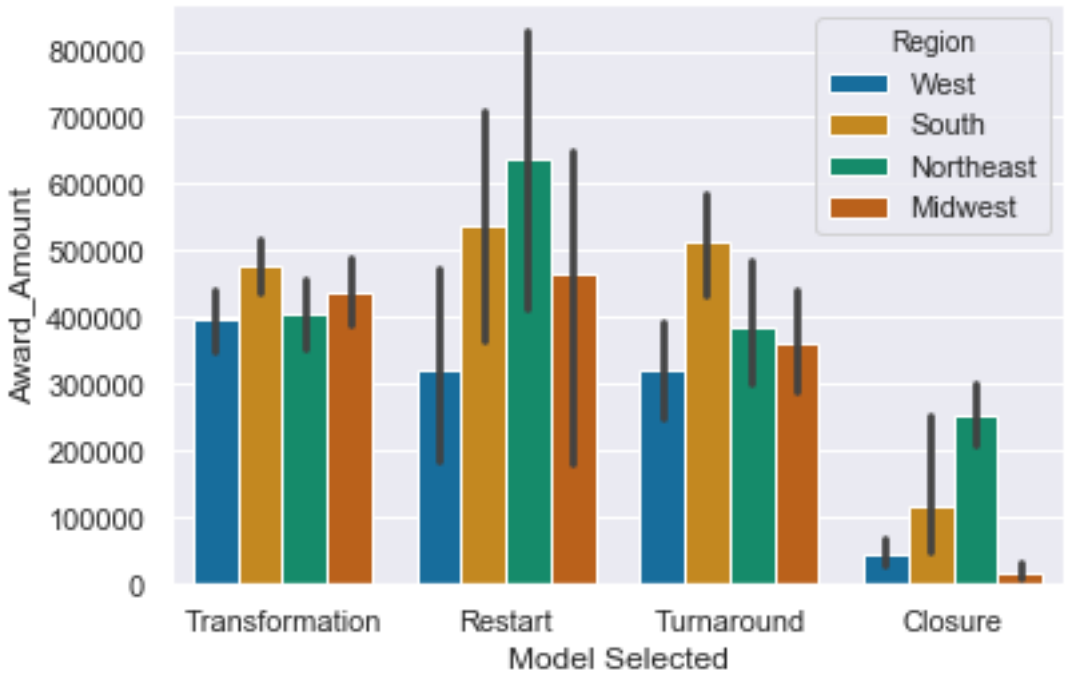
Barplots, pointplots and countplots

The final group of categorical plots are barplots, pointplots and countplot which create statistical summaries of the data. The plots follow a similar API as the other plots and allow further customization for the specific problem at hand.

[98]:
```
# Creating a countplot with the df dataframe and Model Selected on the y axis␣
 ↪and the color varying by Region.

# Showing a countplot with the number of models used with each region a␣
 ↪different color
sns.countplot(data=df,
              y="Model Selected",
              hue="Region")

plt.show()
plt.clf()


# Creating a pointplot with the df dataframe and Model Selected on the x-axis␣
 ↪and Award_Amount on the y-axis.
# Using a capsize in the pointplot in order to show the confidence interval.

# Creating a pointplot and include the capsize in order to show bars on the␣
 ↪confidence interval
sns.pointplot(data=df,
              y='Award_Amount',
              x='Model Selected',
              capsize=.1)

plt.show()
plt.clf()

# Creating a barplot with the same data on the x and y axis and change the␣
 ↪color of each bar based on the Region column.

# Creating a barplot with each Region shown as a different color
sns.barplot(data=df,
              y='Award_Amount',
              x='Model Selected',
              hue='Region')

plt.show()
plt.clf()
```


Transformation
Restart
Turnaround
Closure
Model Selected
0
25
50
75
100
125
150
175
count
Region
West
South
Northeast
Midwest


600000
500000
400000
300000
200000
100000
Award_Amount
Transformation
Restart
Turnaround
Closure
Model Selected




```
<Figure size 432x288 with 0 Axes>
```

The pointplot and barplot can be useful visualizations for understanding the variations of categorical data.

Regression Plots

Seaborn's regression plotting tools can be used to evaluate the data from multiple perspectives and look for relationships between these numeric variables. The function requires the definition of the data and the x and y variables.

The residual plot is a very useful plot for understanding the appropriateness of a regression model. Ideally the residual values in the plot should be plotted randomly across the horizontal line. If a value greater than 1 is passed to the order parameter of regplot() then Seaborn will attempt a polynomial fit using underlying NumPy functions. The residual plot can interpret the second order polynomial and plot the residual values. If the values are more randomly distributed, a second order equation is likely more appropriate for the problem.

Seaborn also supports regression plots with categorical variables. The jitter parameter makes it easier to see the individual distributions of the categorical variables. In some cases even with the jitter it is difficult to see any trends based on the values of the variables. Using an estimator for the x value can provide another helpful view of the data. When there are continuous variables, it can be helpful to break them into different bins.

Seaborn's regplot() function supports several parameters for creating highly customized regression plots.

[102]:
```
fp2 = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data visualization␣
↪with seaborn/Data/tuition.csv'
```

[162]:
```
df3 = pd.read_csv(fp2)
df3.head()
```

[162]:

	INSTNM	OPEID	REGION	SAT_AVG_ALL	PCTPELL	\
0	Alabama A & M University	100200	5	850.0	0.7249	
1	University of Alabama at Birmingham	105200	5	1147.0	0.3505	
2	Amridge University	2503400	5	NaN	0.7455	
3	University of Alabama in Huntsville	105500	5	1221.0	0.3179	
4	Alabama State University	100500	5	844.0	0.7567	

	PCTFLOAN	ADM_RATE_ALL	UG	AVGFACSAL	COMPL_RPY_5YR_RT	…	CONTROL	\
0	0.8159	0.653841	4380.0	7017.0	0.477631579	…	1	
1	0.5218	0.604275	10331.0	10221.0	0.673230442	…	1	
2	0.8781	NaN	98.0	3217.0	0.636363636	…	2	
3	0.4589	0.811971	5220.0	9514.0	0.762222222	…	1	
4	0.7692	0.463858	4348.0	7940.0	0.43006993	…	1	

	WOMENONLY	MENONLY	LOCALE	Tuition	Degree_Type	Ownership	\
0	0.0	0.0	12.0	13435.0	Graduate	Public	
1	0.0	0.0	12.0	16023.0	Graduate	Public	
2	0.0	0.0	12.0	8862.0	Graduate	Private non-profit	
3	0.0	0.0	12.0	18661.0	Graduate	Public	
4	0.0	0.0	12.0	7400.0	Graduate	Public	

	Regions	Locales	Locale_Short
0	South East	City: Midsize	City
1	South East	City: Midsize	City
2	South East	City: Midsize	City
3	South East	City: Midsize	City
4	South East	City: Midsize	City

[5 rows x 24 columns]

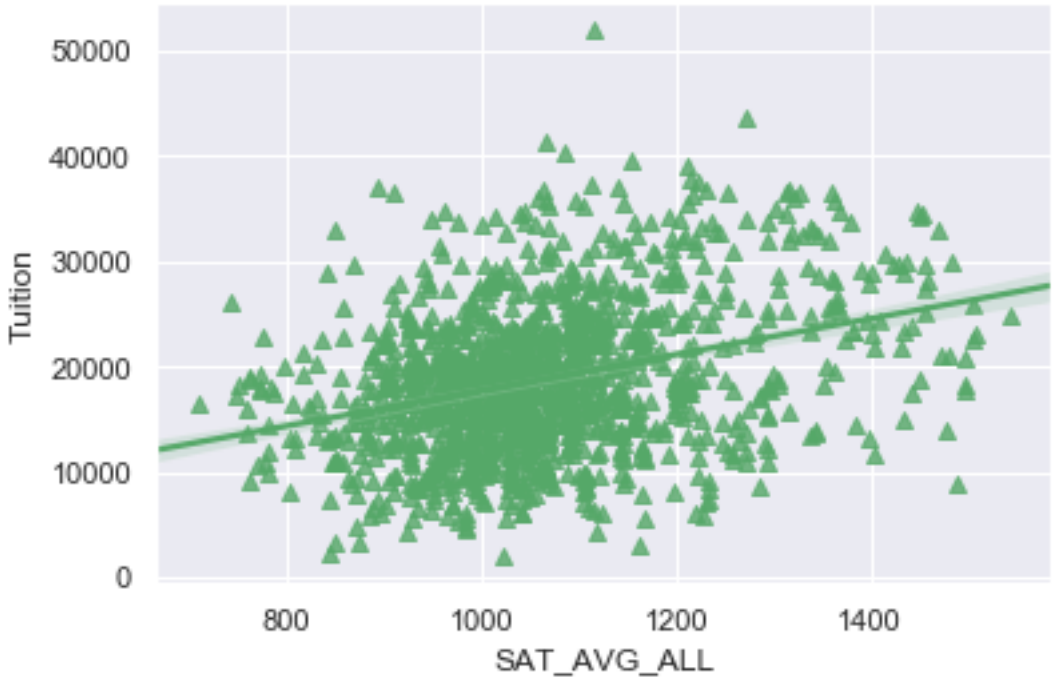
Regression and residual plots

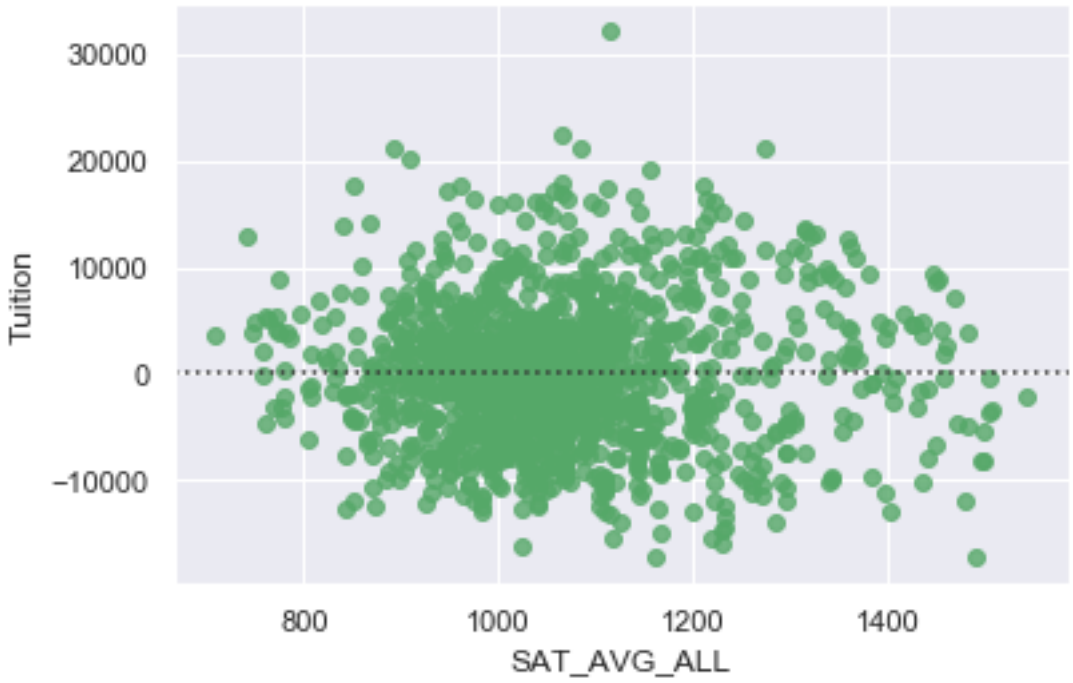
Linear regression is a useful tool for understanding the relationship between numerical variables. Seaborn has simple but powerful tools for examining these relationships.

I will look at some details from the US Department of Education on 4 year college tuition information and see if there are any interesting insights into which variables might help predict tuition costs.

[105]:
```
# Plotting a regression plot comparing Tuition and average SAT␣
↪scores(SAT_AVG_ALL).
```

```
# Displaying a regression plot for Tuition
sns.regplot(data=df3,
          y='Tuition',
          x="SAT_AVG_ALL",
          marker='^',
          color='g')

plt.show()
plt.clf()


# Using a residual plot to determine if the relationship looks linear.

# Displaying the residual plot
sns.residplot(data=df3,
           y='Tuition',
           x="SAT_AVG_ALL",
           color='g')

plt.show()
plt.clf()
```



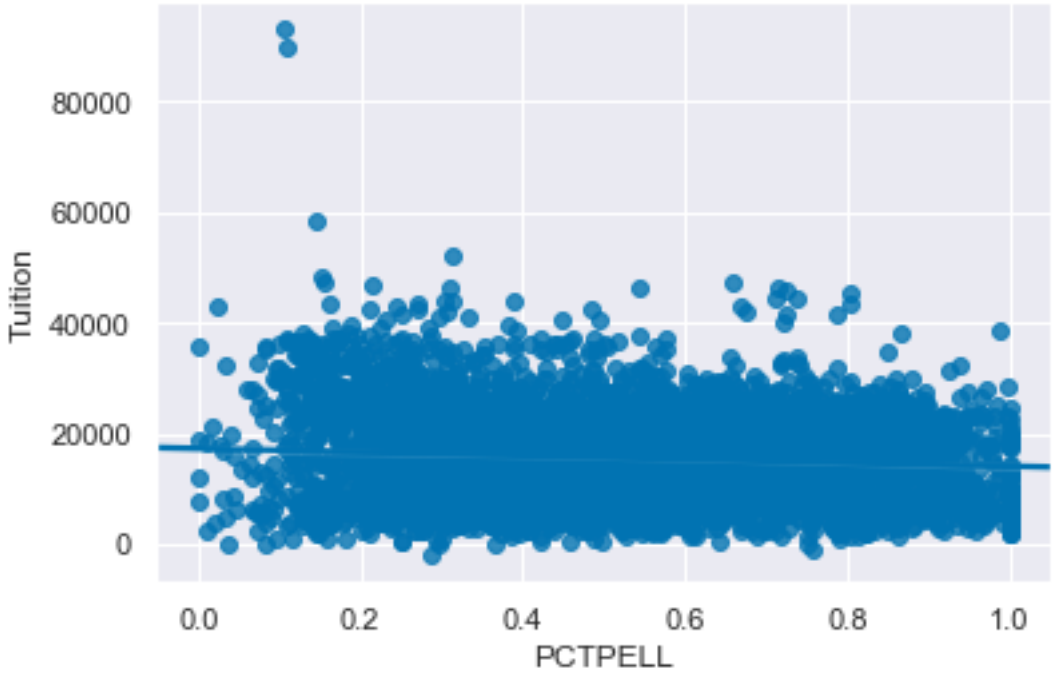


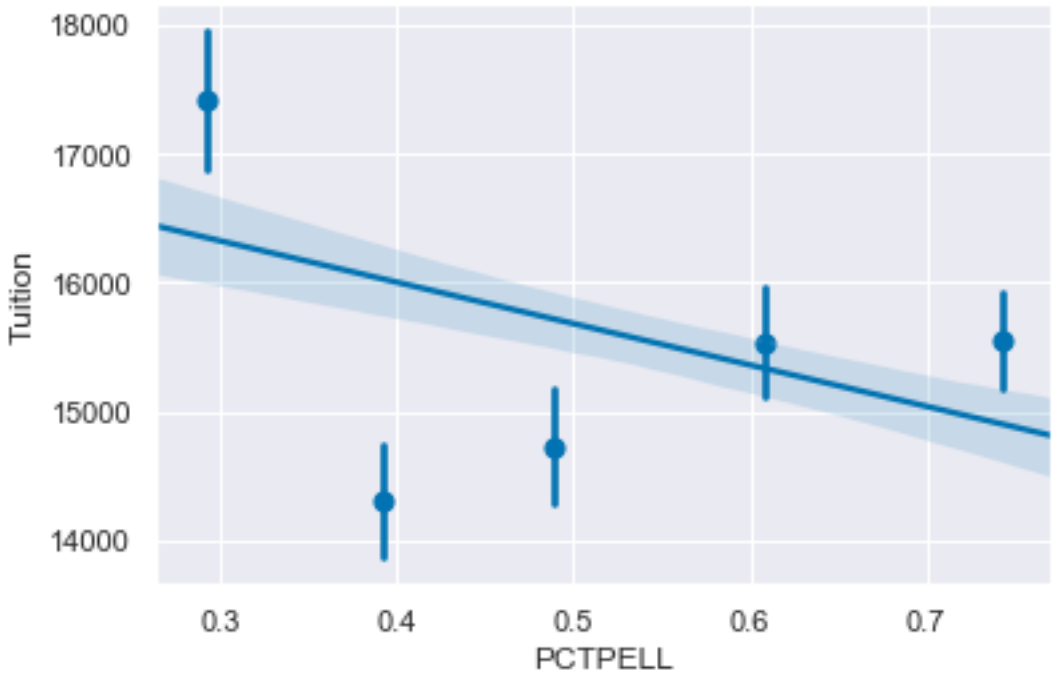

```
<Figure size 432x288 with 0 Axes>
```

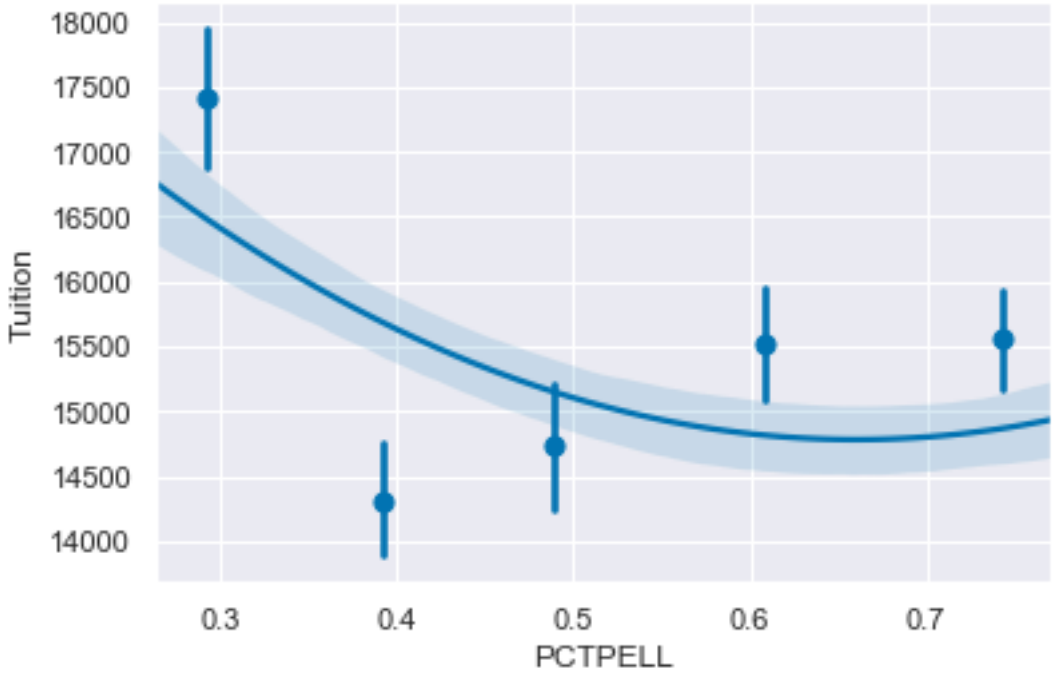
There does appear to be a linear relationship between tuition and SAT scores.

[107]:
```
# Plotting a regression plot of Tuition and PCTPELL.

# Plotting a regression plot of Tuition and the Percentage of Pell Grants
sns.regplot(data=df3,
            y='Tuition',
            x="PCTPELL")

plt.show()
plt.clf()

# Creating another plot that breaks the PCTPELL column into 5 different bins.

# Creating another plot that estimates the tuition by PCTPELL
sns.regplot(data=df3,
            y='Tuition',
            x="PCTPELL",
            x_bins=5)

plt.show()
plt.clf()
```

```
# Creating a final regression plot that includes a 2nd order polynomial␣
 ↪regression line.

# The final plot should include a line using a 2nd order polynomial
sns.regplot(data=df3,
            y='Tuition',
            x="PCTPELL",
            x_bins=5,
            order=2)

plt.show()
plt.clf()
```








```
<Figure size 432x288 with 0 Axes>
```

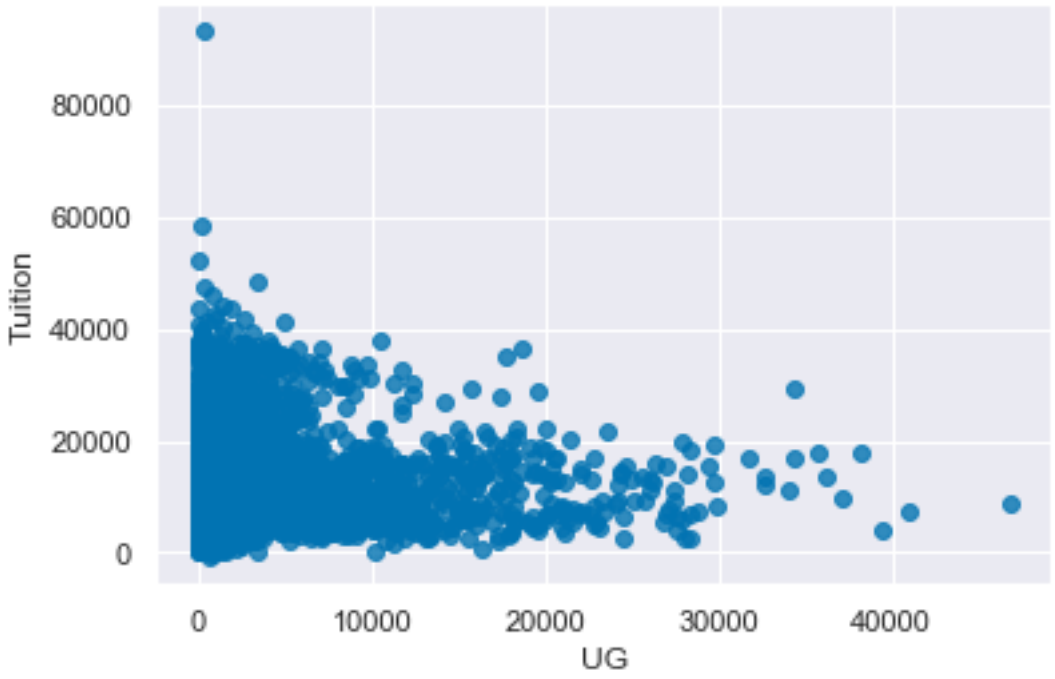
The regplot function is a very powerful tool for quickly analyzing data. However, we to be careful not to overfit the data!

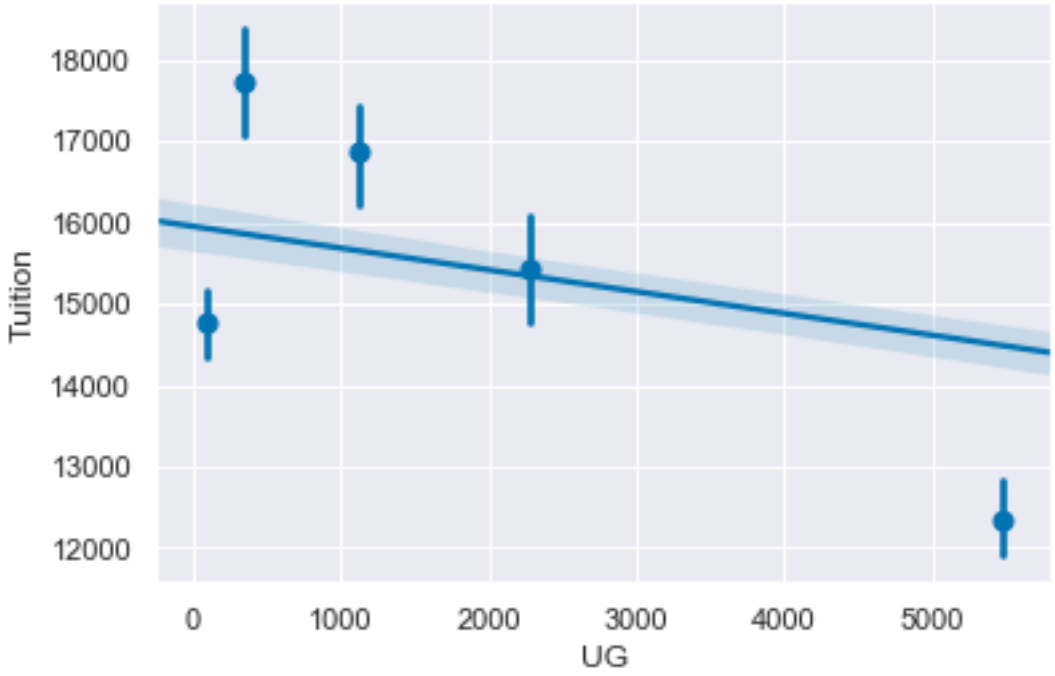
Binning data

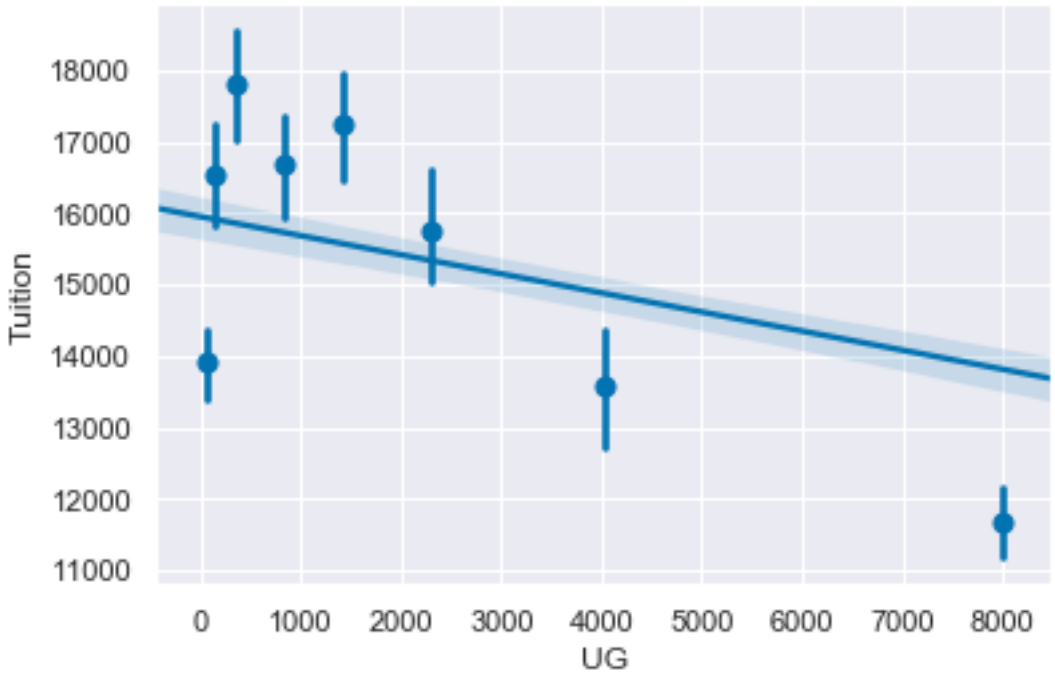
When the data on the x axis is a continuous value, it can be useful to break it into different bins in order to get a better visualization of the changes in the data.

I will look at the relationship between tuition and the Undergraduate population abbreviated as UG in this data. I will start by looking at a scatter plot of the data and examining the impact of different bin sizes on the visualization.

[108]:
```
# Creating a regplot of Tuition and UG and set the fit_reg parameter to False␣
↪to disable the regression line.

# Creating a scatter plot by disabling the regression line
sns.regplot(data=df3,
            y='Tuition',
            x="UG",
            fit_reg=False)

plt.show()
plt.clf()

# Creating another plot with the UG data divided into 5 bins.

# Creating a scatter plot and bin the data into 5 bins
sns.regplot(data=df3,
            y='Tuition',
            x="UG",
            x_bins=5)

plt.show()
plt.clf()

# Creating a regplot() with the data divided into 8 bins.

# Creating a regplot and bin the data into 8 bins
sns.regplot(data=df3,
         y='Tuition',
         x="UG",
         x_bins=8)

plt.show()
plt.clf()
```


80000
60000
40000
20000
0
Tuition
0
10000
20000
30000
40000
UG


18000
17000
16000
15000
14000
13000
12000
Tuition
0
1000
2000
3000
4000
5000
UG




```
<Figure size 432x288 with 0 Axes>
```

Matrix Plots

The heatmap is the most common type of matrix plot and can be easily created by Seaborn. These type of matrix plots can be useful for quickly seeing trends in a dataset. One common usage for heatmap is to visually represent the correlation between variables. Seaborn's heatmap() function expects the data to be in a matrix. The heatmap translates the numerical values in the matrix into a color coded grid. The colors get lighter as the numbers increase.

The display of the heatmap can be customized in multiple ways to present the most information as clearly as possible.

The plot shows the strength of correlation between different variables. The visualization can be useful to understand which variables you might want to further study using regplot().

[111]:
```
fp3 = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data visualization␣
↪with seaborn/Data/guests.csv'
```

[230]:
```
df4 = pd.read_csv(fp3)
df4.head()
```

[230]:

	YEAR	GoogleKnowlege_Occupation	Show	Group	Raw_Guest_List
0	1999	actor	1/11/99	Acting	Michael J. Fox
1	1999	Comedian	1/12/99	Comedy	Sandra Bernhard
2	1999	television actress	1/13/99	Acting	Tracey Ullman

```
3  1999              film actress  1/14/99  Acting  Gillian Anderson
4  1999                     actor  1/18/99  Acting  David Alan Grier
```

Creating heatmaps

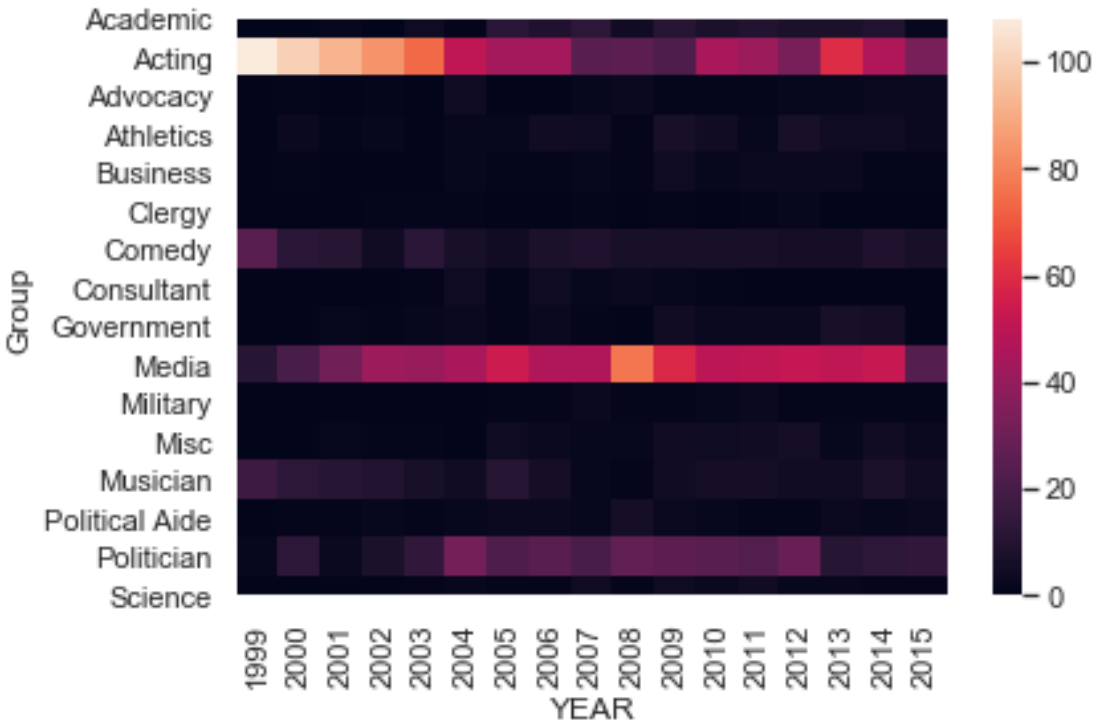
A heatmap is a common matrix plot that can be used to graphically summarize the relationship between two variables. I will start by looking at guests of the Daily Show from 1999 - 2015 and see how the occupations of the guests have changed over time.

The data includes the date of each guest appearance as well as their occupation. First I need to get the data into the right format for Seaborn's heatmap function to correctly plot the data.

[245]:
```
# Creating a crosstab table of the data
pd_crosstab = pd.crosstab(df4["Group"], df4["YEAR"])
print(pd_crosstab)

# Plotting a heatmap of the table
sns.heatmap(pd_crosstab)

# Rotating tick marks for visibility
plt.yticks(rotation=0)
plt.xticks(rotation=90)

plt.show()
```

```
YEAR           1999  2000  2001  2002  2003  2004  2005  2006  2007  2008  \
Group
Academic          0     0     2     0     4     1    12     9    13     5
Acting          108   100    92    84    74    51    44    44    25    26
Advocacy          0     1     0     1     0     4     0     0     2     3
Athletics         0     3     1     2     0     2     2     5     4     1
Business          0     1     0     0     0     2     1     1     2     1
Clergy            0     0     0     1     1     1     0     0     1     0
Comedy           25    12    11     5    12     7     5     8     9     7
Consultant        0     0     0     0     1     4     1     4     2     3
Government        0     0     2     1     2     3     1     3     1     0
Media            11    21    31    42    41    45    54    47    47    77
Military          0     0     0     0     0     0     1     1     3     1
Misc              0     0     2     1     1     0     4     3     2     2
Musician         17    13    11    10     7     5    11     6     2     1
Political Aide    0     1     1     2     1     2     3     3     2     6
Politician        2    13     3     8    14    32    22    25    21    27
Science           0     0     0     0     1     2     1     1     4     1

YEAR           2009  2010  2011  2012  2013  2014  2015
Group
Academic         11     8    10     8     8    10     2
Acting           22    45    42    33    60    47    33
Advocacy          1     1     1     2     2     3     3
```

Athletics	7	5	2	7	4	4	3
Business	4	2	3	3	3	1	1
Clergy	1	0	1	2	0	0	0
Comedy	7	7	7	6	6	9	7
Consultant	2	1	0	0	0	0	0
Government	5	3	3	3	7	6	0
Media	59	50	51	52	51	53	24
Military	1	2	3	1	1	1	1
Misc	5	4	5	6	2	5	3
Musician	5	6	6	5	5	8	5
Political Aide	3	2	1	1	3	2	3
Politician	26	25	23	29	11	13	14
Science	4	3	5	2	2	1	1



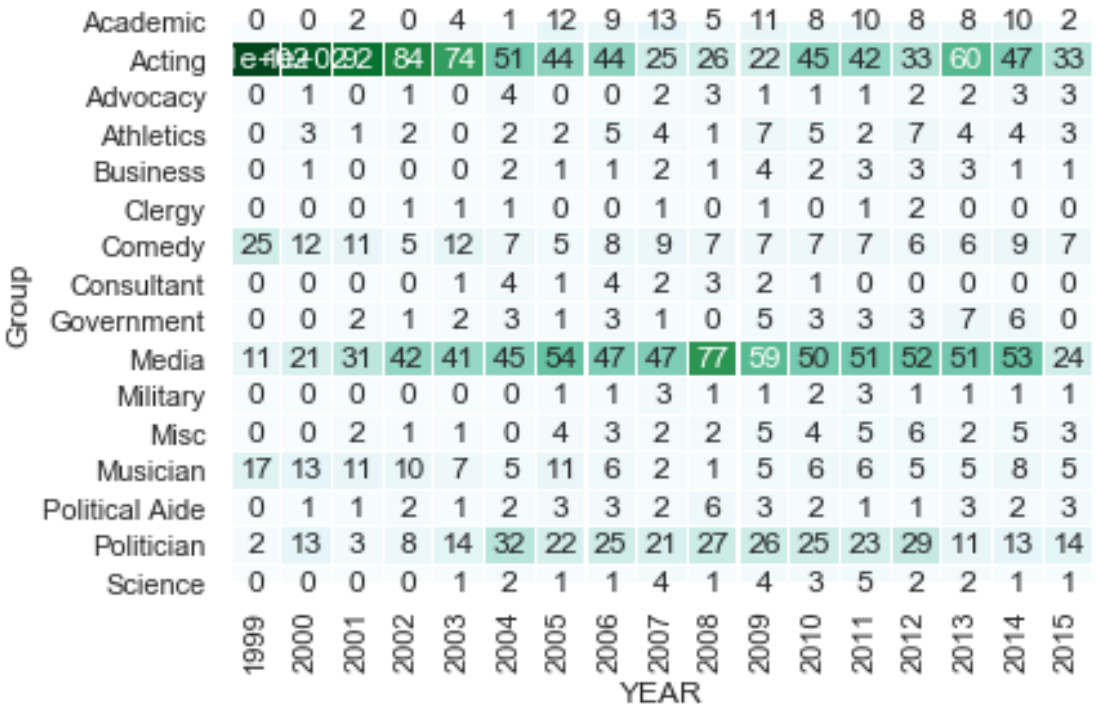
The lighter colors in above figure suggest higher frequency. Acting, media and politician are some of the main professions of the guests.

Customizing heatmaps

Seaborn supports several types of additional customizations to improve the output of a heatmap. I will continue to use the Daily Show data that is stored in the df4 variable but I will customize the output.

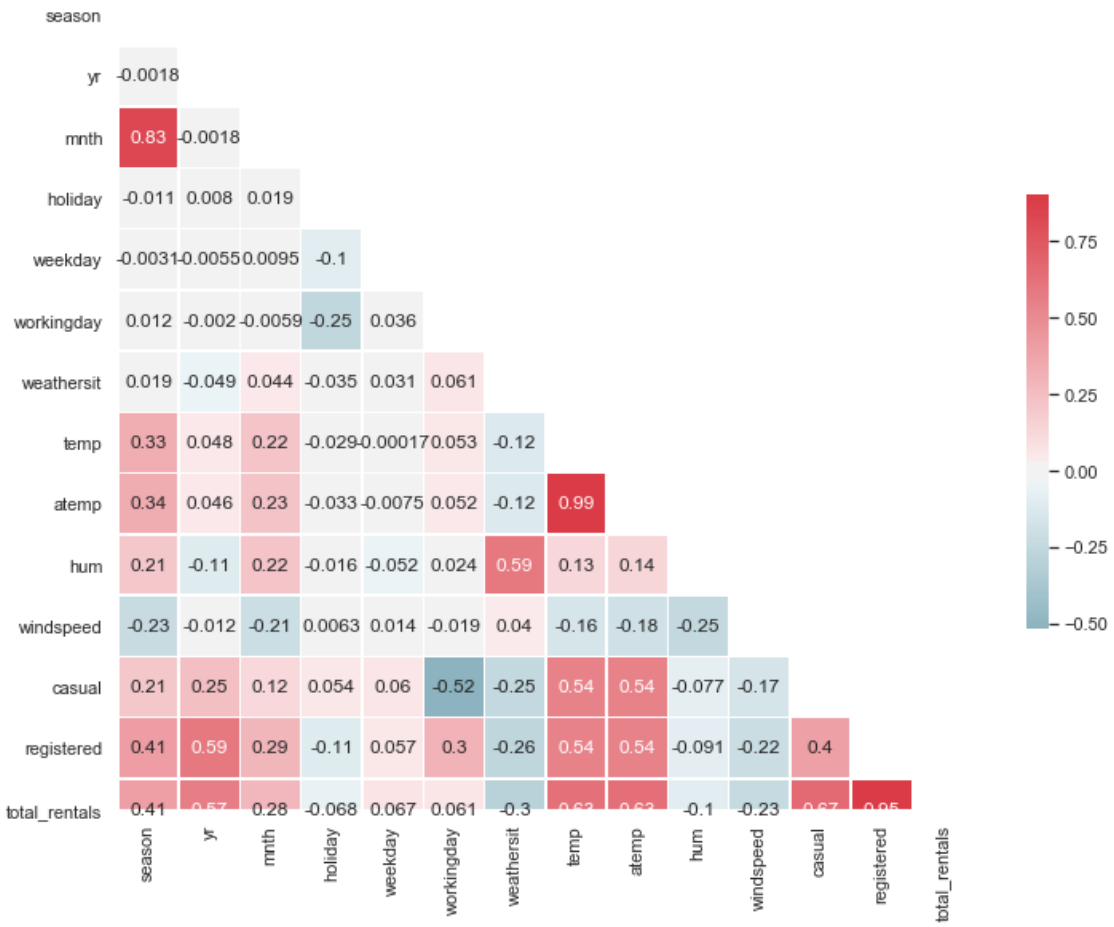
[228]:
```
# Creating the crosstab DataFrame
pd_crosstab = pd.crosstab(df4["Group"], df4["YEAR"])
```

```
# Plotting a heatmap of the table with no color bar and using the BuGn palette
sns.heatmap(pd_crosstab, cbar=False, cmap="BuGn", linewidths=0.2, annot=True)

# Rotating tick marks for visibility
plt.yticks(rotation=0)
plt.xticks(rotation=90)

#Showing the plot
plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

[240]:
```
import numpy as np
sns.set(style="white")
corr = df5.corr()
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
f, ax = plt.subplots(figsize=(12, 10))
cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.9, center=0, square=True,␣
 ↪linewidths=.5, annot=True,cbar_kws={"shrink": .5});
```



Darker colors in red suggest a strong positive and in blue a strong negative relationship.

Building a FacetGrid

Seaborn's FacetGrid is the foundation for building data-aware grids. A data-aware grid allows you to create a series of small plots that can be useful for understanding complex data relationships.

I will continue to look at the College Scorecard Data from the US Department of Education. This rich dataset has many interesting data elements that we can plot with Seaborn.

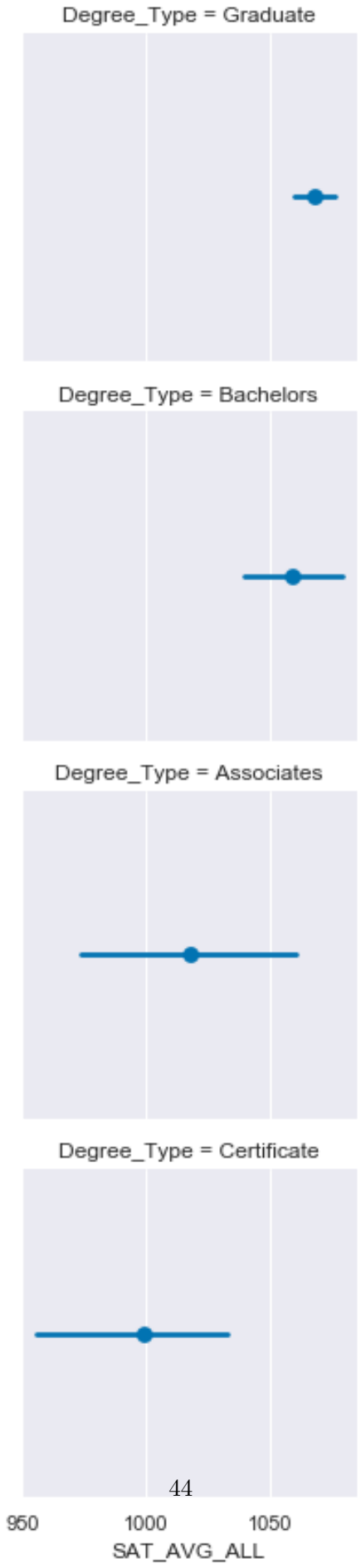
When building a FacetGrid, there are two steps:

- Creating a FacetGrid object with columns, rows, or hue.
- Mapping individual plots to the grid.

[116]:
```
# Creating FacetGrid with Degree_Type and specify the order of the rows using␣
↪row_order
g2 = sns.FacetGrid(df3,
                   row="Degree_Type",
                   row_order=['Graduate', 'Bachelors', 'Associates', 'Certificate'])
```

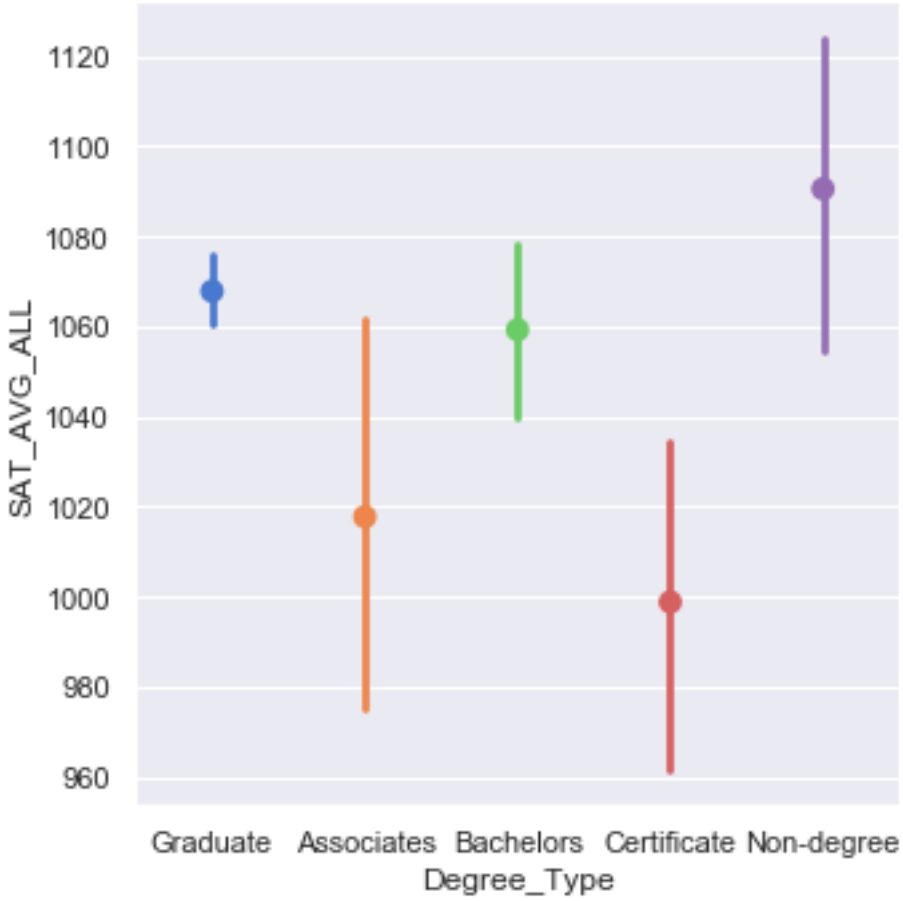
```
# Mapping a pointplot of SAT_AVG_ALL onto the grid
g2.map(sns.pointplot, 'SAT_AVG_ALL')

# Showing the plot
plt.show()
plt.clf()
```

```
/opt/anaconda3/lib/python3.7/site-packages/seaborn/axisgrid.py:715: UserWarning:
Using the pointplot function without specifying `order` is likely to produce an
incorrect plot.
  warnings.warn(warning)
```


Degree_Type = Graduate
Degree_Type = Bachelors
Degree_Type = Associates
Degree_Type = Certificate
950
1000
1050
SAT_AVG_ALL

```
<Figure size 432x288 with 0 Axes>
```

[137]:
```
# Setting up a factorplot
g = sns.factorplot("Degree_Type", "SAT_AVG_ALL", data=df3, kind="point",␣
 ↪palette="muted", legend=False)

# Showing plot
plt.show()
```

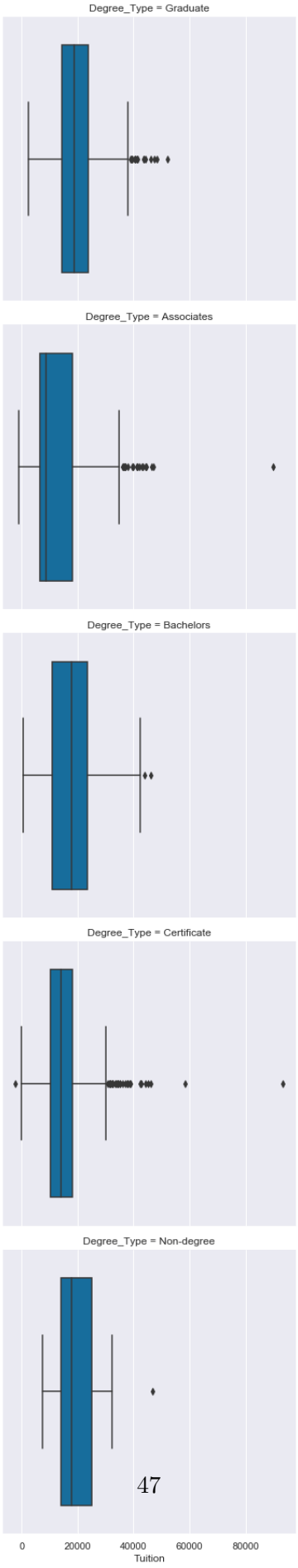


Using a factorplot

In many cases, Seaborn's factorplot() can be a simpler way to create a FacetGrid. Instead of creating a grid and mapping the plot, we can use the factorplot() to create a plot with one line of code. I will recreate one of the plots from above using factorplot() and show how to create a boxplot on a data-aware grid.

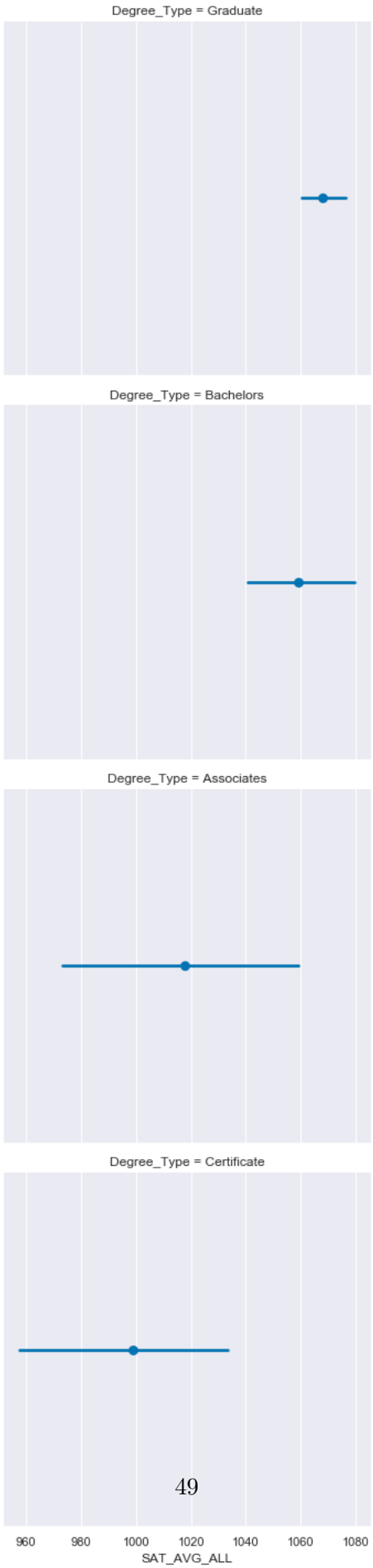
[119]:
```
# Creating a factorplot() that contains a boxplot (box) of Tuition values␣
 ↪varying by Degree_Type across rows.

# Creating a factor plot that contains boxplots of Tuition values
sns.factorplot(data=df3,
               x='Tuition',
               kind='box',
               row='Degree_Type')

plt.show()
plt.clf()


# Creating a factorplot() of SAT Averages (SAT_AVG_ALL) facetted across␣
 ↪Degree_Type that shows a pointplot (point).

# Creating a facetted pointplot of Average SAT_AVG_ALL scores facetted by␣
 ↪Degree Type
sns.factorplot(data=df3,
               x='SAT_AVG_ALL',
               kind='point',
               row='Degree_Type',
               row_order=['Graduate', 'Bachelors', 'Associates', 'Certificate'])

plt.show()
plt.clf()
```

```
/opt/anaconda3/lib/python3.7/site-packages/seaborn/categorical.py:3666:
UserWarning: The `factorplot` function has been renamed to `catplot`. The
original name will be removed in a future release. Please update your code. Note
that the default `kind` in `factorplot` (`'point'`) has changed `'strip'` in
`catplot`.
  warnings.warn(msg)
```


Degree_Type = Graduate
Degree_Type = Associates
Degree_Type = Bachelors
Degree_Type = Certificate
Degree_Type = Non-degree
0
20000
40000
60000
80000
Tuition

```
/opt/anaconda3/lib/python3.7/site-packages/seaborn/categorical.py:3666:
UserWarning: The `factorplot` function has been renamed to `catplot`. The
original name will be removed in a future release. Please update your code. Note
that the default `kind` in `factorplot` (`'point'`) has changed `'strip'` in
`catplot`.
  warnings.warn(msg)
```

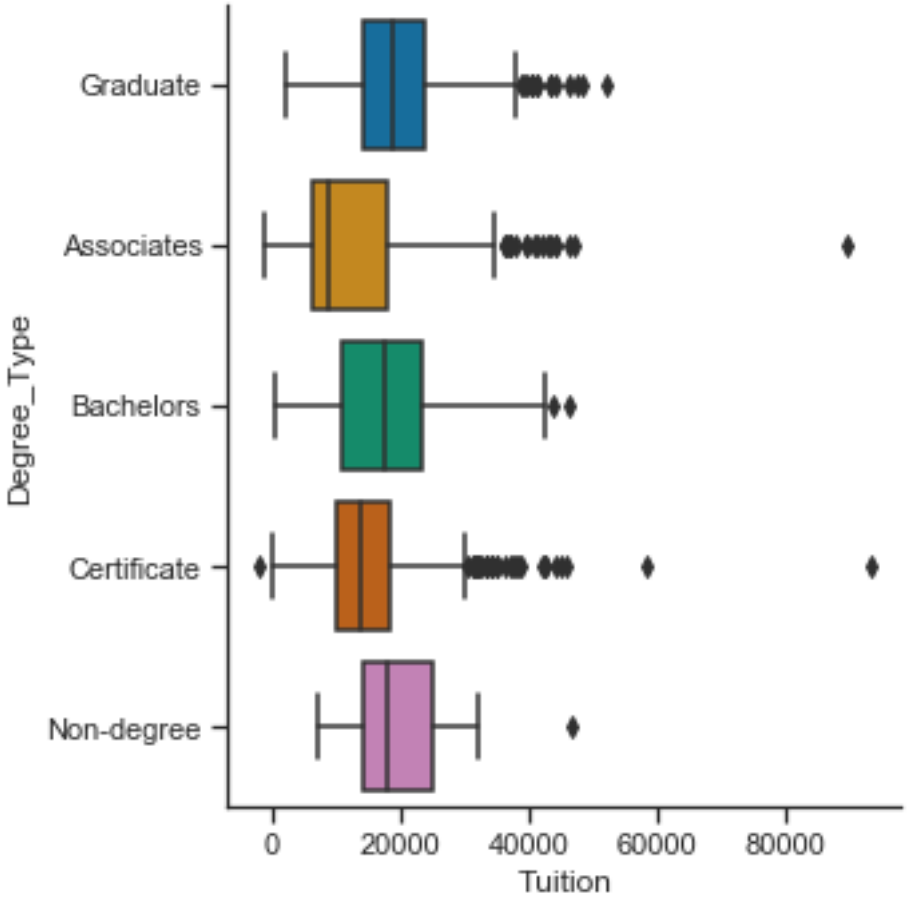
```
<Figure size 432x288 with 0 Axes>
```


Degree_Type = Graduate
Degree_Type = Bachelors
Degree_Type = Associates
Degree_Type = Certificate
960
980
1000
1020
1040
1060
1080
SAT_AVG_ALL

```
<Figure size 432x288 with 0 Axes>
```

The factorplot is often more convenient than using a FacetGrid for creating data aware grids.

[120]:
```
with sns.axes_style(style='ticks'):
    g = sns.factorplot("Tuition", "Degree_Type", data=df3, kind="box")
    g.set_axis_labels("Tuition", "Degree_Type");
```



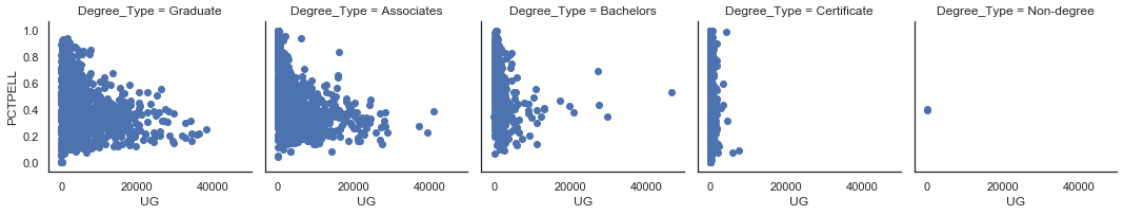
Using a lmplot

The lmplot is used to plot scatter plots with regression lines on FacetGrid objects. The API is similar to factorplot with the difference that the default behavior of lmplot is to plot regression lines.

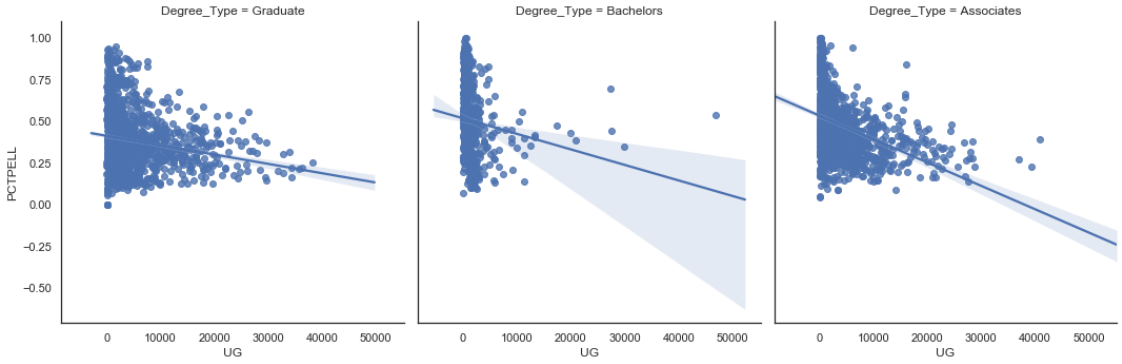
First I will look at the Undergraduate population (UG) and compare it to the percentage of students receiving Pell Grants (PCTPELL).

For the second lmplot I will look at the relationships between Average SAT scores and Tuition across the different degree types and public vs. non-profit schools.

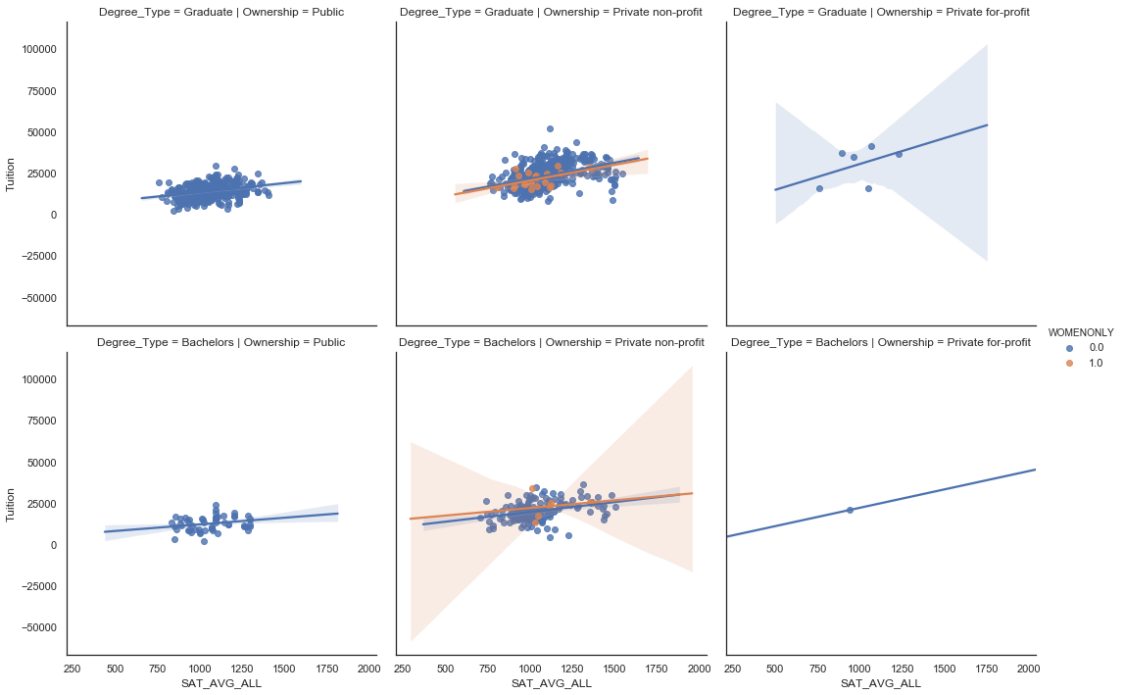
[236]:
```
# Creating a FacetGrid() with "Degree_Type" columns and scatter plot of "UG"␣
 ↪and "PCTPELL".

# Creating a FacetGrid varying by column and columns ordered with the␣
 ↪degree_order variable

g = sns.FacetGrid(df3, col="Degree_Type", row_order=['Graduate', 'Bachelors',␣
 ↪'Associates', 'Certificate'])

# Mapping a scatter plot of Undergrad Population compared to PCTPELL
g.map(plt.scatter, 'UG', 'PCTPELL')

plt.show()
plt.clf()

# Creating a lmplot() using the same values from the FacetGrid()

# Re-creating the plot above as an lmplot
sns.lmplot(data=df3,
        x='UG',
        y='PCTPELL',
        col="Degree_Type",
        col_order=['Graduate', 'Bachelors', 'Associates'])

plt.show()
plt.clf()

# Creating a facetted lmplot() comparing "SATAVGALL" to "Tuition" with columns␣
 ↪varying by "Ownership" and rows by "Degree_Type".
# In the lmplot() adding a hue for Women Only Universities.

# Creating an lmplot that has a column for Ownership, a row for Degree_Type and␣
 ↪hue based on the WOMENONLY column and columns defined by inst_order
sns.lmplot(data=df3,
           x='SAT_AVG_ALL',
           y='Tuition',
           col="Ownership",
           row='Degree_Type',
           row_order=['Graduate', 'Bachelors'],
           hue='WOMENONLY')

plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>



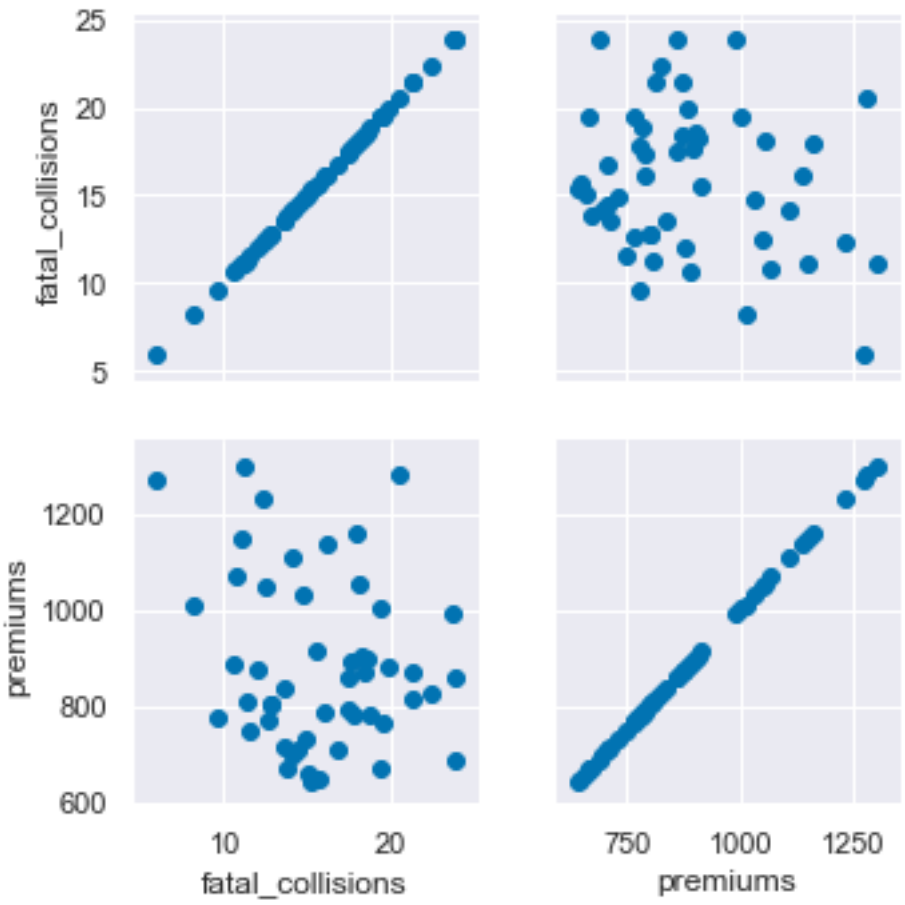
<Figure size 432x288 with 0 Axes>



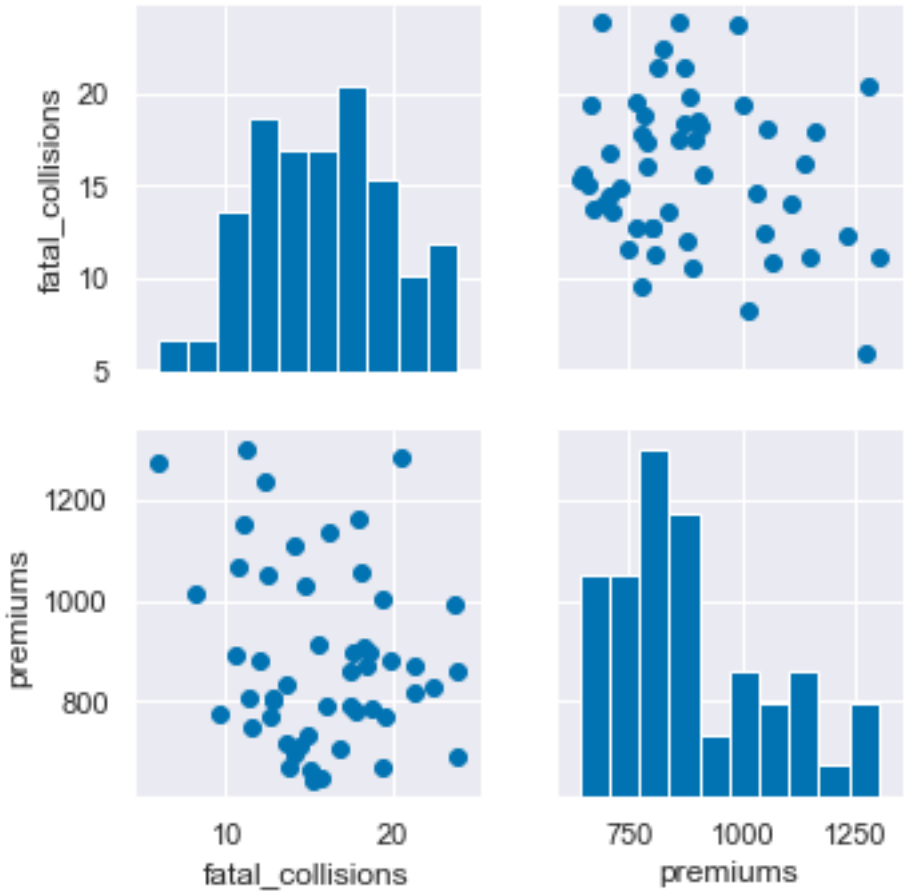
```
<Figure size 432x288 with 0 Axes>
```

Creating small multiples of plots is very useful for many types of analysis. With Seaborn, it is easy to use the plot types to quickly perform complex visualizations.

[165]:
```
# Comparing "fatal_collisions" to "premiums" by using a scatter plot mapped to␣
 ↪a PairGrid().

# Creating a PairGrid with a scatter plot for fatal_collisions and premiums
g = sns.PairGrid(df1, vars=["fatal_collisions", "premiums"])
g2 = g.map(plt.scatter)

plt.show()
plt.clf()

# Creating another PairGrid but plot a histogram on the diagonal and scatter␣
 ↪plot on the off diagonal.

# Creating the same PairGrid but map a histogram on the diag
g = sns.PairGrid(df1, vars=["fatal_collisions", "premiums"])
g2 = g.map_diag(plt.hist)
g3 = g2.map_offdiag(plt.scatter)

plt.show()
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

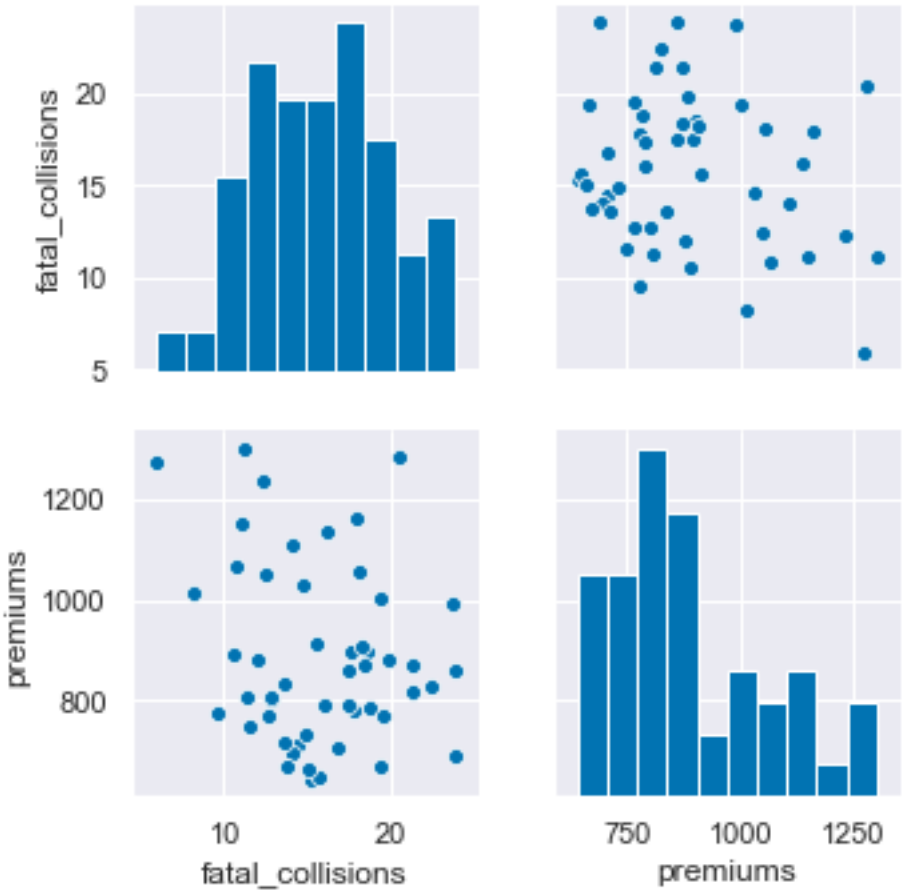



```
<Figure size 432x288 with 0 Axes>
```

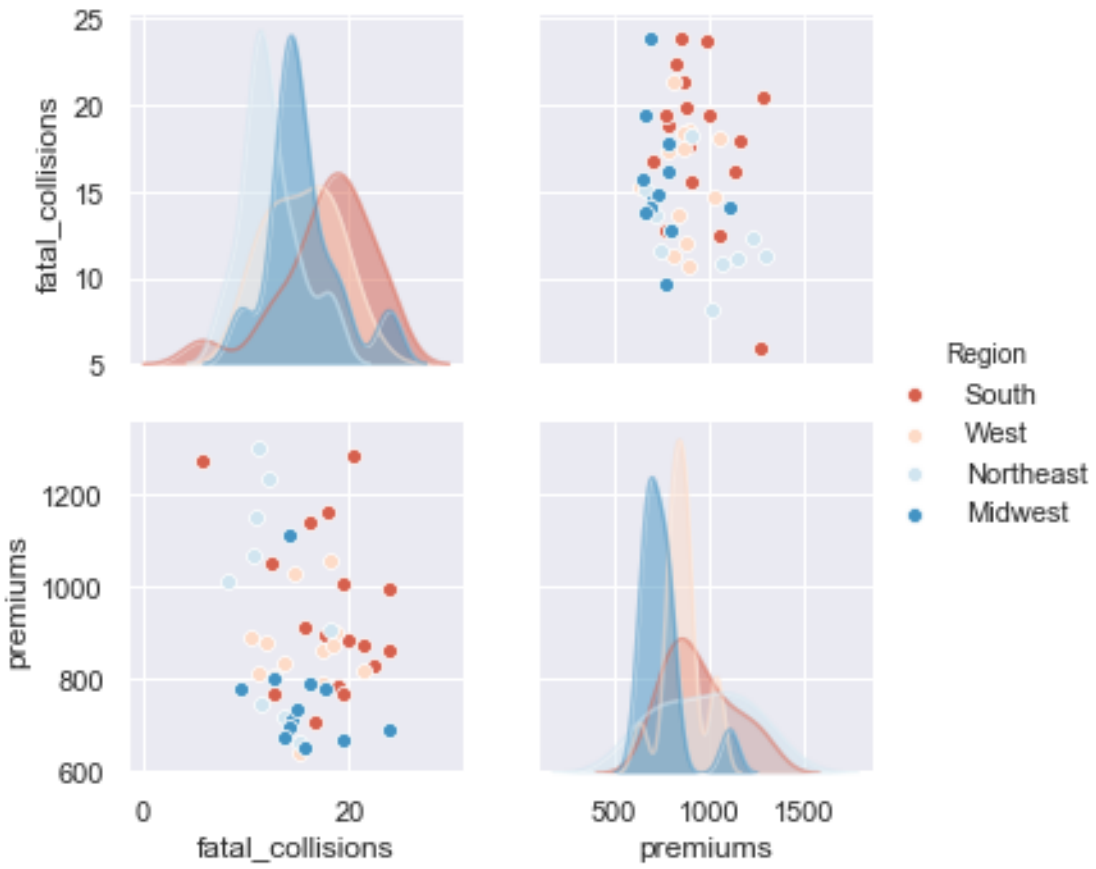
This analysis provides useful insight into the distribution of premium amounts as well as the limited relationships between fatal_collision and premiums.

[168]:
```
# Recreating the pairwise plot from the previous exercise using pairplot().

# Creating a pairwise plot of the variables using a scatter plot
sns.pairplot(data=df1,
        vars=["fatal_collisions", "premiums"],
        kind='scatter')

plt.show()
plt.clf()

# Creating another pairplot using the "Region" to color code the results.
# Using the RdBu palette to change the colors of the plot.
```

```
# Plotting the same data but use a different color palette and color code by␣
 ↪Region
sns.pairplot(data=df1,
        vars=["fatal_collisions", "premiums"],
        kind='scatter',
        hue='Region',
        palette='RdBu',
        diag_kws={'alpha':.5})

plt.show()
plt.clf()
```



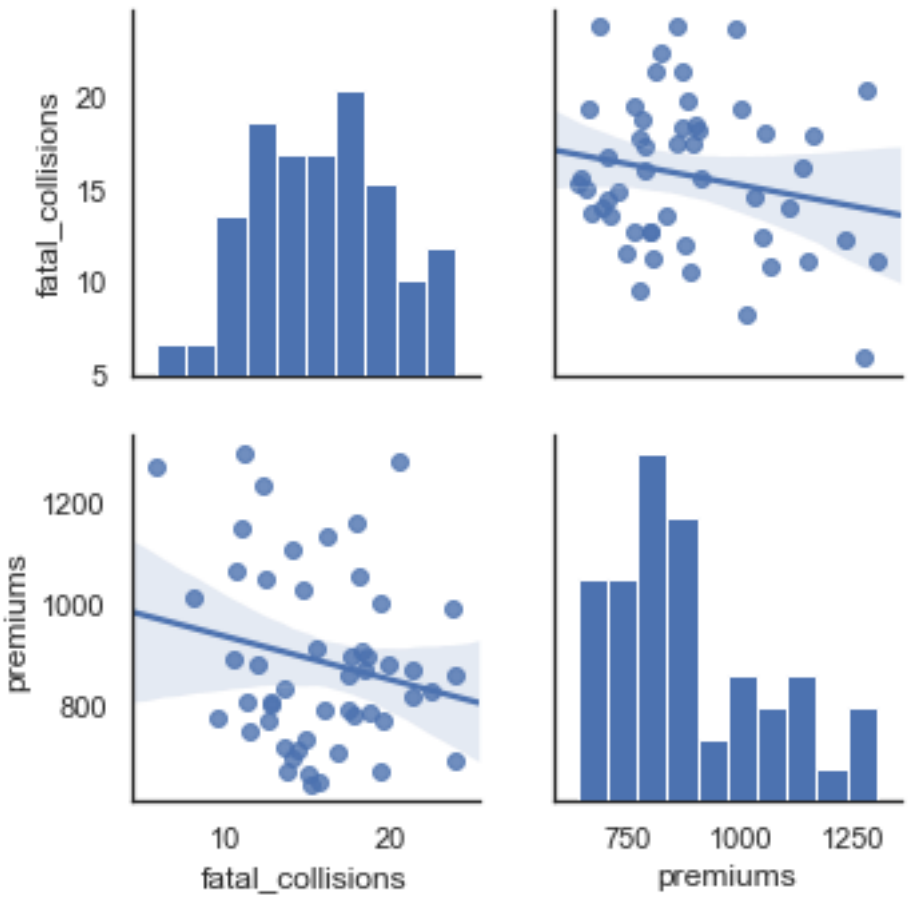

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

[238]:
```
sns.pairplot(df1, vars=["fatal_collisions", "premiums"],␣
↪kind='reg',diag_kind='hist')
```

[238]:
```
<seaborn.axisgrid.PairGrid at 0x1a197efdd0>
```



Additional pairplots

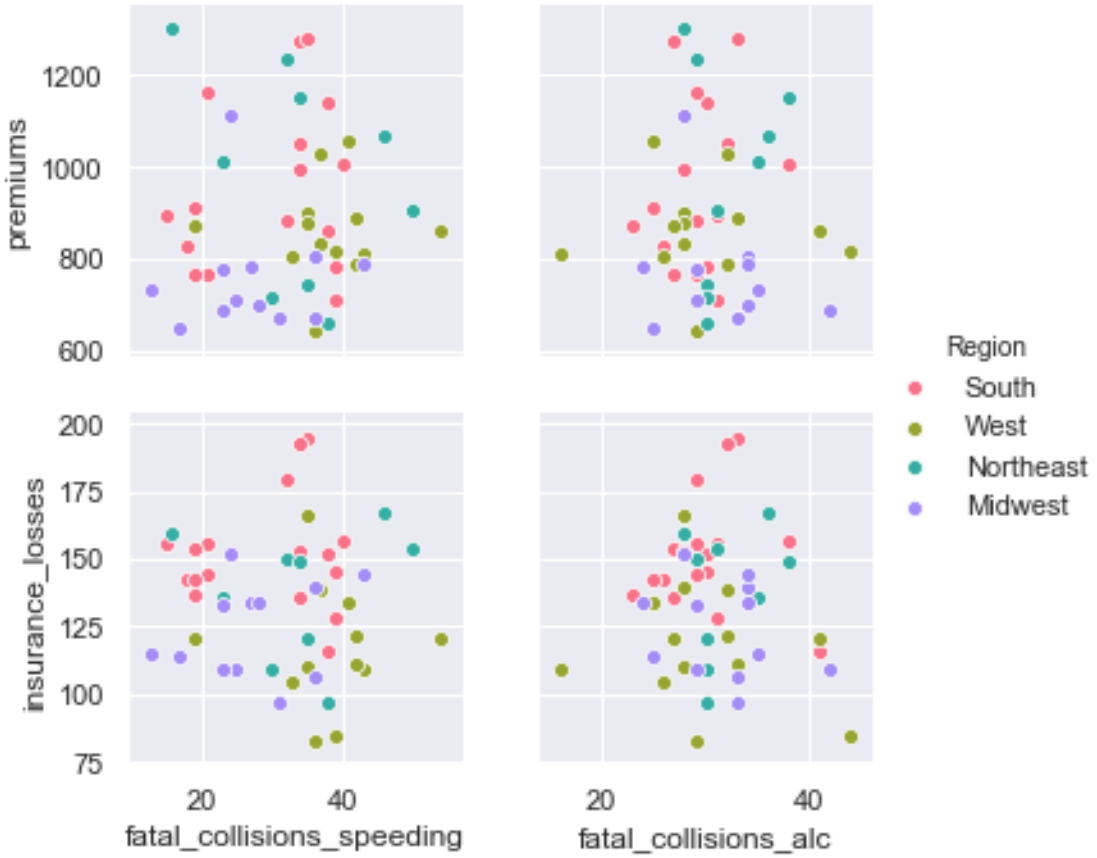
I am going through a couple of more examples of how the pairplot() can be customized for quickly analyzing data and determining areas of interest that might be worthy of additional analysis.

One area of customization that is useful is to explicitly defining the x_vars and y_vars that we wish to examine. Instead of examining all pairwise relationships, this capability allows us to look only at the specific interactions that may be of interest.

We have already looked at using kind to control the types of plots. We can also use diag_kind to control the types of plots shown on the diagonals. In the final example, we will include a regression and kde plot in the pairplot.

[169]:
```
# Creating a pair plot that examines fatal_collisions_speeding and␣
 ↪fatal_collisions_alc on the x axis and premiums and insurance_losses on the␣
 ↪y axis.
# Using the husl palette and color code the scatter plot by Region.

# Build a pairplot with different x and y variables
sns.pairplot(data=df1,
```

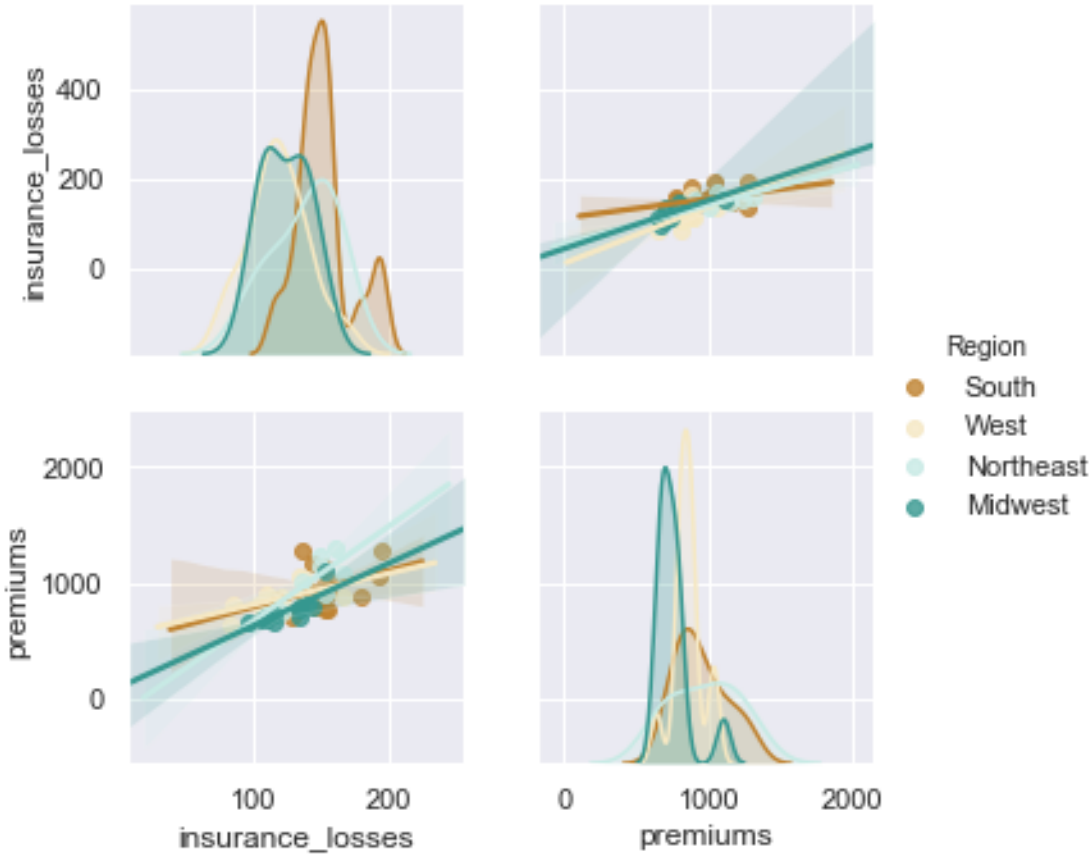
```
        x_vars=["fatal_collisions_speeding", "fatal_collisions_alc"],
        y_vars=['premiums', 'insurance_losses'],
        kind='scatter',
        hue='Region',
        palette='husl')

plt.show()
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

[170]:
```
# Building a pairplot() with kde plots along the diagonals. Include the␣
 ↪insurance_losses and premiums as the variables.
# Using a reg plot for the the non-diagonal plots.
# Using the BrBG palette for the final plot.
# plotting relationships between insurance_losses and premiums
sns.pairplot(data=df1,
             vars=["insurance_losses", "premiums"],
             kind='reg',
```

```
            palette='BrBG',
            diag_kind = 'kde',
            hue='Region')

plt.show()
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

JointGrid and Jointplot

A JointGrid() allows us to compare the distribution of data between two variables. A JointGrid() makes use of scatter plots, regression lines as well as histograms, distribution plots, and kernel density estimates to give us insight into our data. Jointgrid() takes a small number of inputs and creates an insightful visualization of the data. Jointgrid() depicts relationship between two variables. The centre of the plot contains a scatter plot of these two variables. The plots along the x and y-axis show the distribution of the data for each variable. The plot can be configured by specifying the type of joint plots as well as the marginal plots.

Hex plot is also a method to demonstrate the relationship between the two variables. Seaborn

automatically includes the Pearson r correlation in this plot.

The jointplot() supports simple creation of scatter, hex, residual, regression, and kde plots. It can also support adding overlay plots to enhance the final output. We can also set limit of a variable on x-axis. By including the plot_joint() function, a kde plot is overlaid on the scatter plot. This combination of plots is useful for understanding the areas where variables have natural groupings.

Building a JointGrid and jointplot

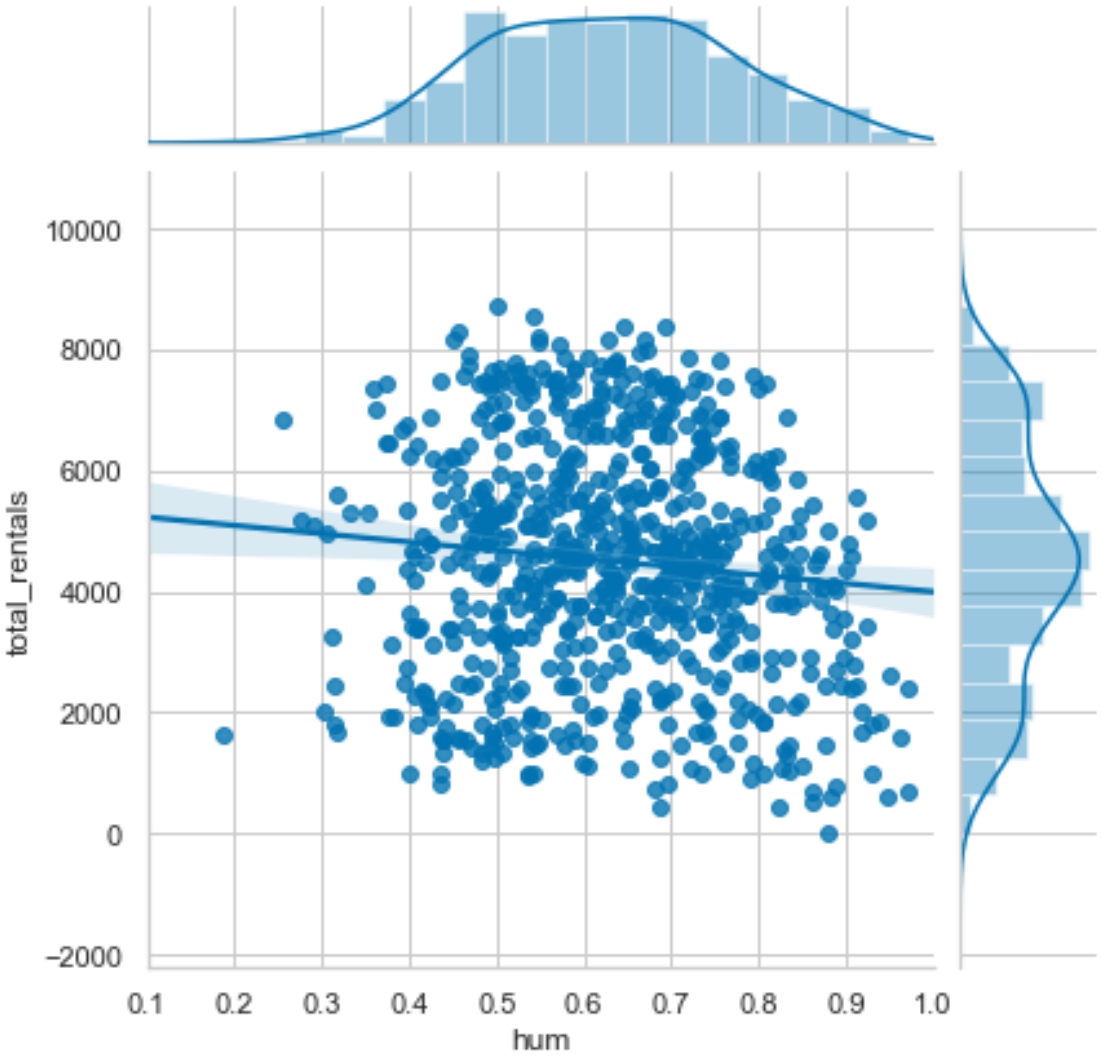
Seaborn's JointGrid combines univariate plots such as histograms, rug plots and kde plots with bivariate plots such as scatter and regression plots. These plots also demonstrate how Seaborn provides convenient functions to combine multiple plots together.

I will use the bike share data. I will look at the relationship between humidity levels and total rentals to see if there is an interesting relationship we might want to explore later.

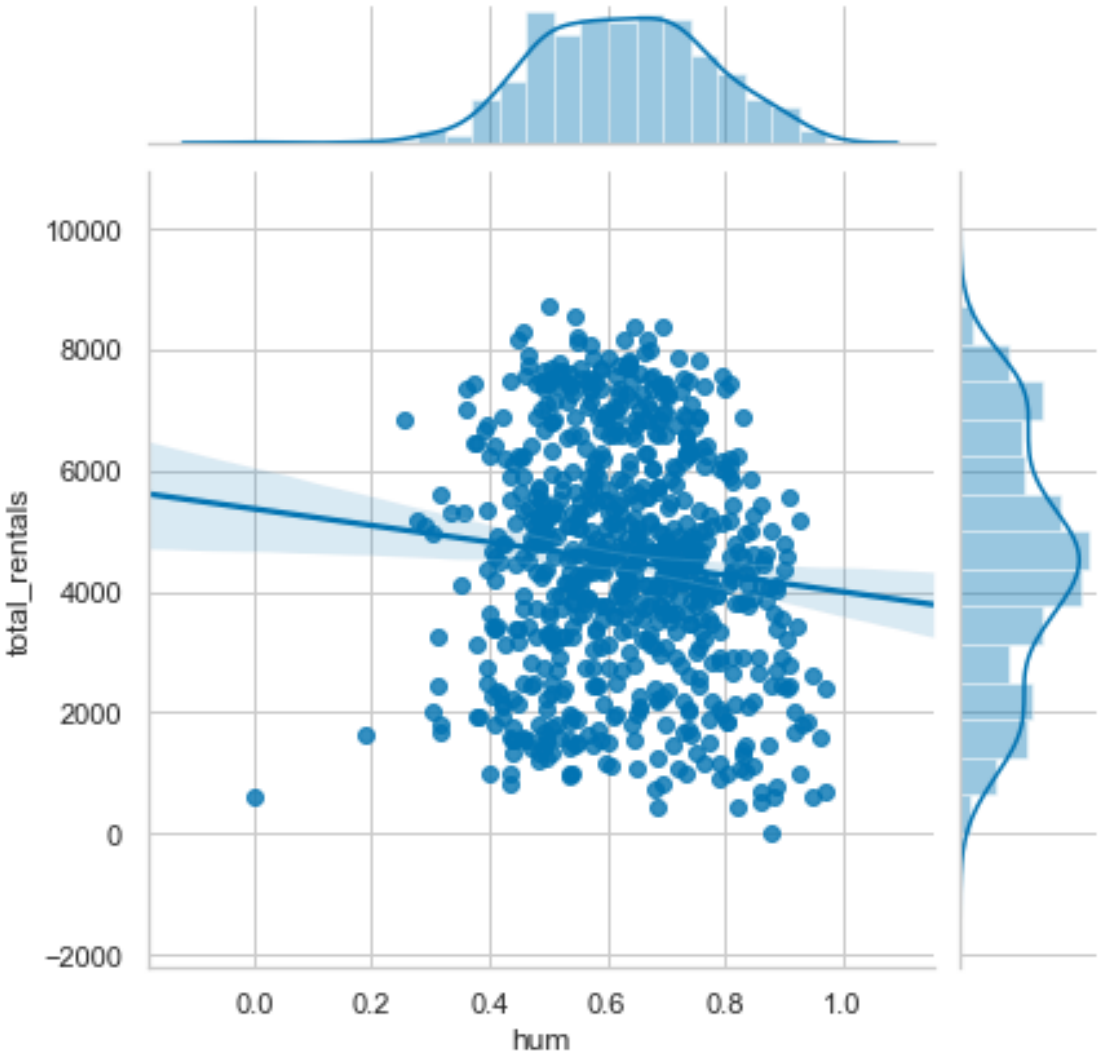
[177]:
```
fp4 = '/Users/MuhammadBilal/Desktop/Data Camp/Intermediate data visualization␣
↪with seaborn/Data/bike.csv'
```

[178]:
```
df5 = pd.read_csv(fp4)
```

[179]:
```
# Building a JointGrid comparing humidity and total_rentals
sns.set_style("whitegrid")
g = sns.JointGrid(x="hum",
                  y="total_rentals",
                  data=df5,
                  xlim=(0.1, 1.0))

g.plot(sns.regplot, sns.distplot)

plt.show()
plt.clf()

# Re-creating the plot using a jointplot().

# Creating a jointplot similar to the JointGrid
sns.jointplot(x="hum",
        y="total_rentals",
        kind='reg',
        data=df5)

plt.show()
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

These plots show that there is limited relationship between rental amounts and humidity levels.

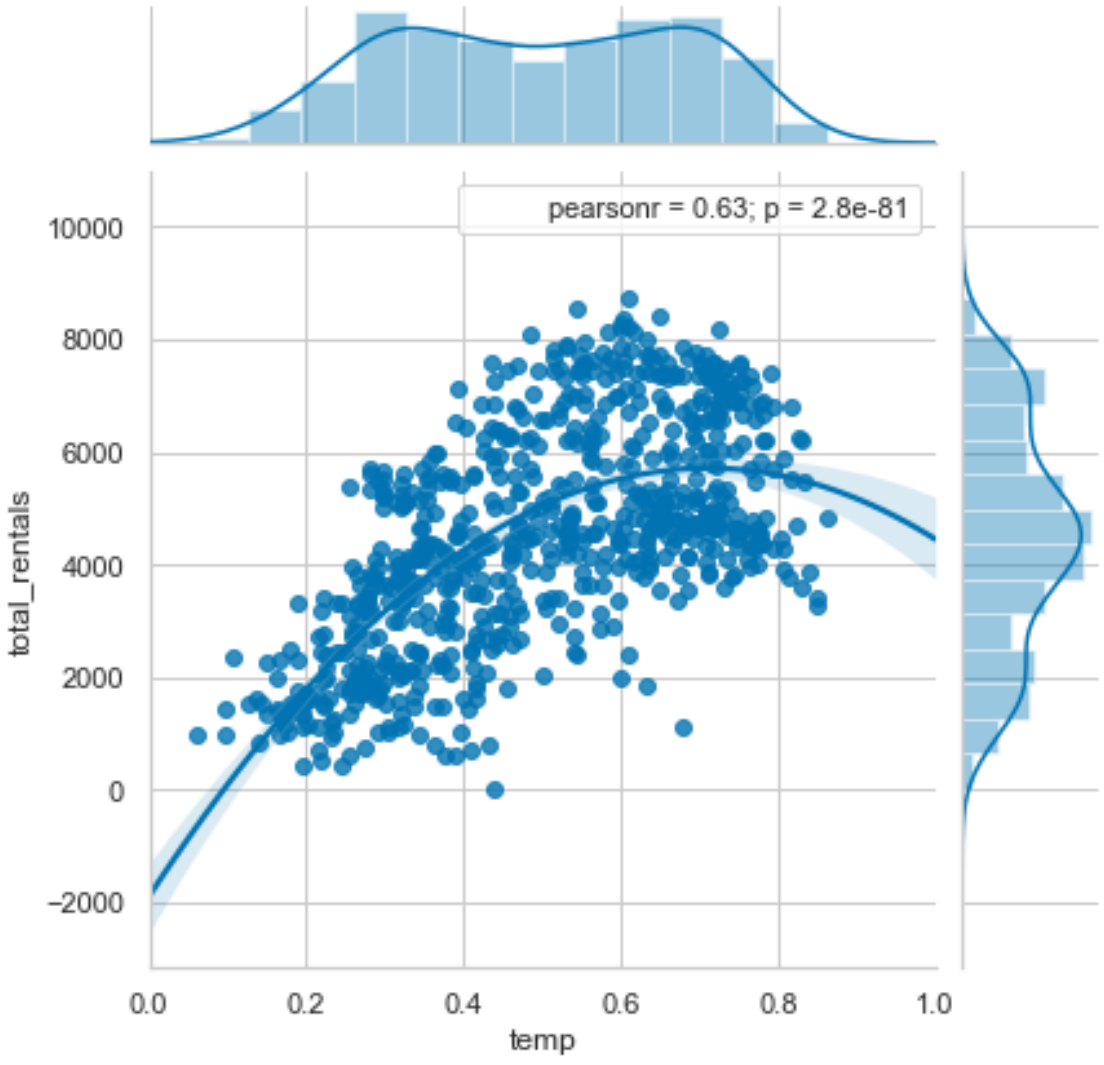
Jointplots and regression

Since the previous plot does not show a relationship between humidity and rental amounts, we can look at another variable that we reviewed earlier. Specifically, the relationship between temp and total_rentals.

[184]:
```
# Plotting temp vs. total_rentals as a regression plot
from scipy.stats import pearsonr
sns.jointplot(x="temp",
              y="total_rentals",
              kind='reg',
```

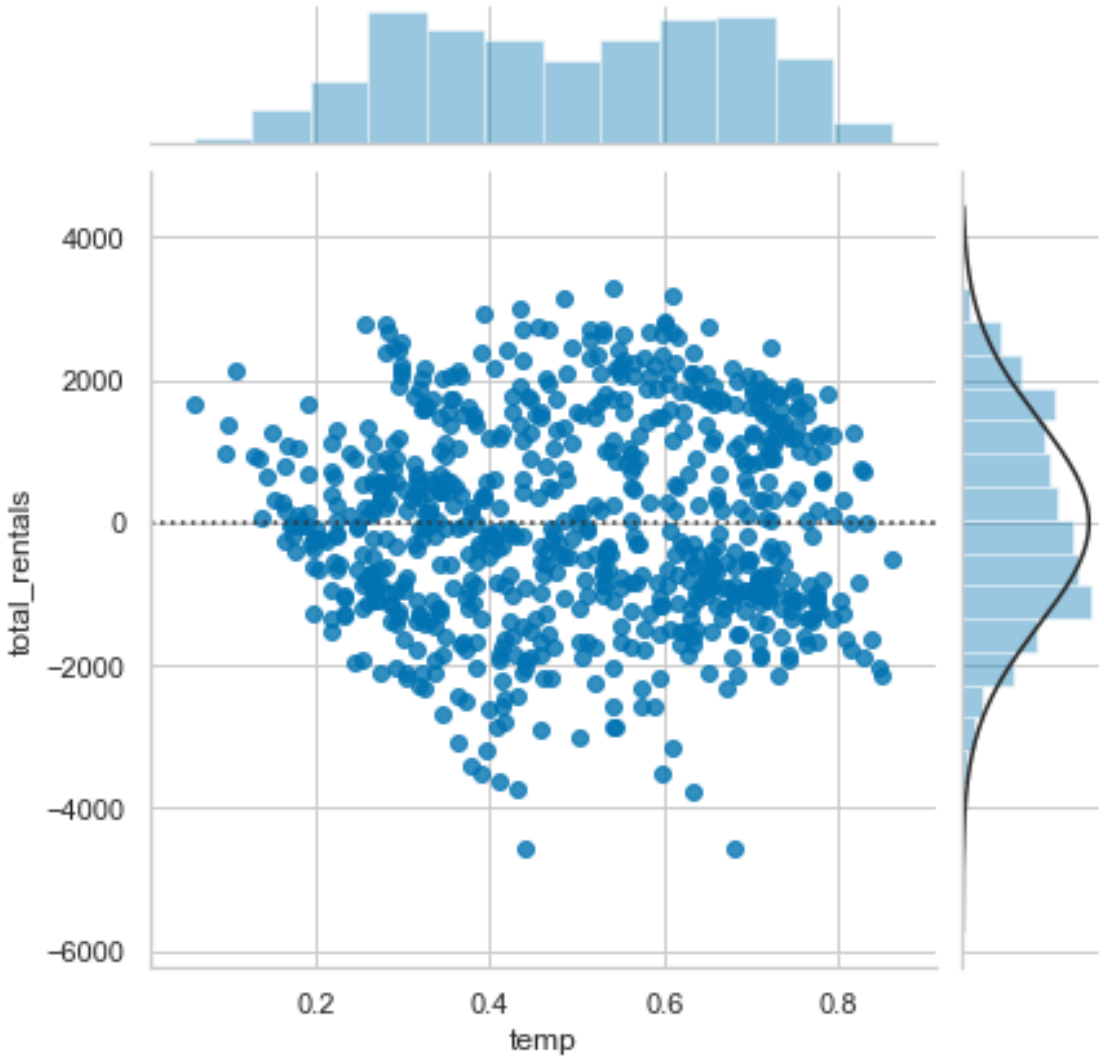
```
              data=df5,
              order=2,
              stat_func=pearsonr,
              xlim=(0, 1))

plt.show()
plt.clf()
```

```
/opt/anaconda3/lib/python3.7/site-packages/seaborn/axisgrid.py:1847:
UserWarning: JointGrid annotation is deprecated and will be removed in a future
release.
  warnings.warn(UserWarning(msg))
```




```
<Figure size 432x288 with 0 Axes>
```

[183]:
```
# Using a residual plot to check the appropriateness of the model.

# Plotting a jointplot showing the residuals
sns.jointplot(x="temp",
        y="total_rentals",
        kind='resid',
        data=df5,
        order=2)

plt.show()
plt.clf()
```



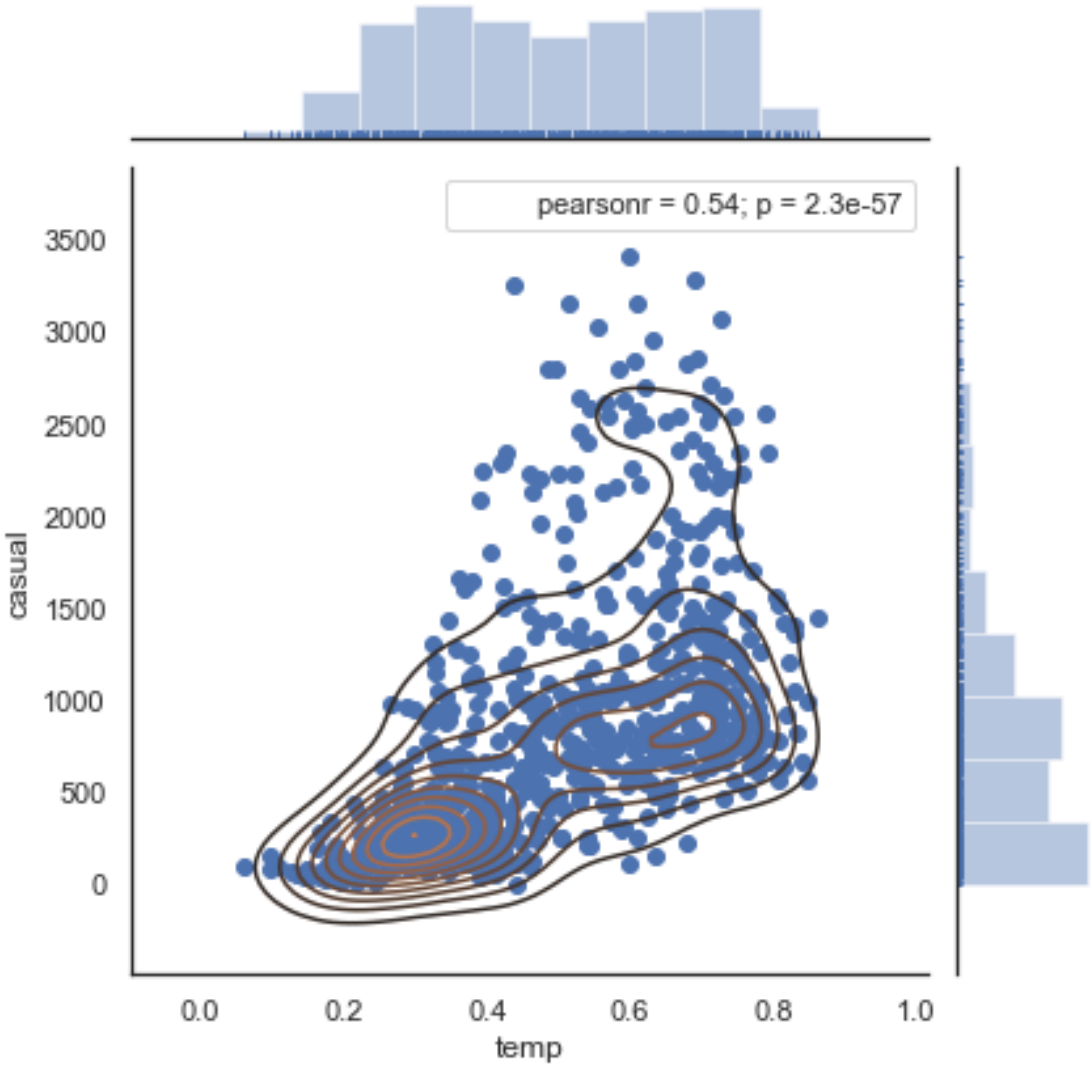

```
<Figure size 432x288 with 0 Axes>
```

Based on the residual plot and the pearson r value, there is a positive relationship between temperature and total_rentals.

Complex jointplots

The jointplot is a convenience wrapper around many of the JointGrid functions. However, it is possible to overlay some of the JointGrid plots on top of the standard jointplot. In this example, we can look at the different distributions for riders that are considered casual versus those that are registered.

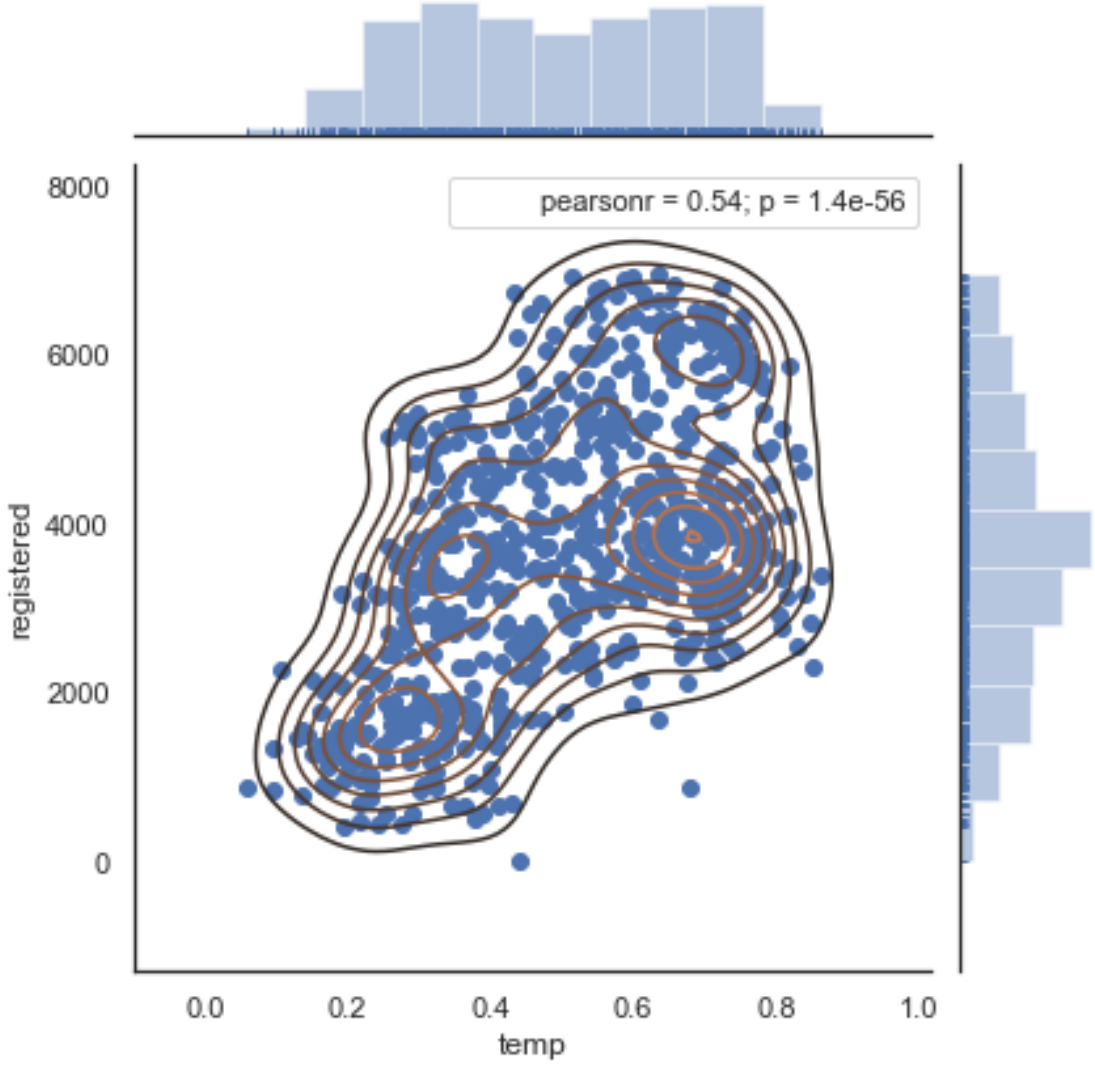
[218]:
```
# Creating a jointplot with a scatter plot comparing temp and casual riders.
# Overlaying a kdeplot on top of the scatter plot.
# Creating a jointplot of temp vs. casual riders
# Including a kdeplot over the scatter plot
g = (sns.jointplot(x="temp",
                   y="casual",
                   kind='scatter',
                   data=df5,
                   stat_func=pearsonr,
                   marginal_kws=dict(bins=10, rug=True))
    .plot_joint(sns.kdeplot))

plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

[219]:
```
# Building a similar plot for registered users.

# Replicating the above plot but only for registered riders
g = (sns.jointplot(x="temp",
                   y="registered",
                   kind='scatter',
                   data=df5,
                   stat_func=pearsonr,
                   marginal_kws=dict(bins=10, rug=True))
    .plot_joint(sns.kdeplot))
```

```
plt.show()
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

[ ]: