Visualizing Statistical Data Using Seaborn

by Janani Ravi

Course Overview

Course Overview

Hi! My name is Janani Ravi, and welcome to this course on Visualizing Statistical Data Using Seaborn. A little about myself. I have a master's degree in electrical engineering from Stanford and have worked at companies such as Microsoft, Google, and Flipkart. At Google, I was one of the first engineers working on real-time collaborative editing in Google Docs, and I hold four patents for its underlying technologies. I currently work on my own stuff at Loonycorn, a studio for high-quality video content. In this course, you will work with Seaborn, which has powerful libraries to visualize and explore your data. Seaborn works closely with the PyData stack. It is built on top of matplotlib and integrated with NumPy and StatsModels and other Python libraries used for data science. We start this course off by visualizing univariate and bivariate distributions. We'll build regression plots, KDE curves, and histograms to extract insights from data. We'll also study how we can detect correlations in our data using heat maps. We'll then use Seaborn to visualize pairwise relationships of high dimensionality using the facet grid and the pair grid. These are examples of trellis plots which are a precursor to building ML models. And we'll explore a bike sharing dataset which allows us to make decisions about the kind of machine learning model we want to build. Plots to fix color and style are important elements to making your visualizations memorable. We'll study the color palettes available in Seaborn and see how we can manipulate

specific plot elements in our graphs. At the end of this course, you'll be very comfortable using Seaborn libraries to build powerful, interesting, and vivid visualizations, an important precursor to using data in machine learning.

Visualizing Relationships and Distributions in Seaborn

Module Overview

Hi, and welcome to this course on Visualizing Statistical Data Using the Seaborn library in Python. We'll start off by visualizing relationships and distributions in Seaborn in this very first module. For any data analyst or data scientist, Seaborn is a very powerful tool in their tool clip. That's because Seaborn is a powerful visualization library which allows them to explore relationships that might exist in data and exploit these relationships when they build their machine learning models. If you've worked with data before, it's quite likely that you've used Matplotlib. Seaborn is very closely integrated with it. In fact, it's built on top of Matplotlib. In fact, Seaborn is very closely coupled with the PyData stack. The PyData stack is comprised of libraries for numerical and statistical analysis such as NumPy and pandas. An important Python library that we generally use for plotting figures and graphs is Matplotlib. Matplotlib seeks to make easy things easy and hard things possible. And Seaborn is a perfect complement to this. You can think of Seaborn as a higher-level abstraction which allows you to build very complex visualizations with very little code. Seaborn makes production ready plots.

Prerequisites and Course Overview

Here are some of the technologies that you need to know in order to make the most of this course, the course prereqs. This course assumes that you're very comfortable programming in Python. If you're not, here are some courses on Pluralsight that you can take before you come to Visualizing Data with Seaborn. Here are some skills that you need in order to be able to understand and work on the demos in this course. You need to be comfortable programming in Python, specifically Python 3 because that's what we use in this course. We assume that you already have Python 3 installed on your machine and you're using Jupyter notebooks. That's what we use to write all the code in this course. This is a course on using the Seaborn visualization

library in Python for statistical data, which means you need to be comfortable with basic high school level statistics. We'll start this course off by visualizing relationships in Seaborn. We'll see how we can work with both univariate and bivariate relationships. We'll study a wide variety of plots that can be used to model these relationships such as histograms, KDE curves, scatter plots, box plots, violin plots, etc. In the next module, we'll cover trellis plots in Seaborn. Trellis plots allow us to view panels of data in an n by n matrix. We'll specifically cover the facet grid for conditional relationships and the pair grid for pairwise relationships. In the module after that, we'll talk about plot aesthetics and style. We'll see the themes and color palettes that Seaborn makes available to us to set up our visualization exactly how we want it to be.

Installing Seaborn and Exploring the Wine Dataset

In this course, we'll be very, very hands-on. We'll work with a lot of different visualizations, and we'll tweak it to suit our needs. We'll start off by installing Seaborn. This course assumes that you already have Python 3 installed on your local machine and know how to work with Jupyter notebooks. After installing Seaborn, we'll explore our very first dataset that we use for visualizations, the wine dataset. Here is the directory that's going to contain all of the IPython notebooks that I'm going to be using for this course. IPython notebooks are basically another name for Jupyter notebooks. Jupyter notebooks are great to run Python because they provide an interactive Python shell on the browser. Within a Jupyter notebook, you can write your code, view the results right there on screen, and also include comments in Markdown. All the datasets that we'll use in this course will be in the form of CSV files and available within the datasets folder. Here are the datasets. We'll specifically work with the wine quality data, Facebook data, and bike rental data. I'm going to create a brand-new Python 3 notebook. All code in this course is written in Python 3. Here is the name of my first demo. I'm going to install Seaborn using pip and explore the wine dataset. The Seaborn library can be installed very easily by using the Python package manager, pip. You can use pip install seaborn prefixed by an exclamation point within the Jupyter notebook to get Seaborn. The pip command on my machine is mapped to pip 3 for python 3. Installing Seaborn will bring in all the other dependencies of Seaborn as well, all the packages in the PyData stack, NumPy, SciPy, Matplotlib, pandas, and so on. The Seaborn library is typically aliased as SNS when you import it into your Python code. The Seaborn version I'm using is 0. 8. 1, so if you're on a newer version of Seaborn, do check the documentation if you find any differences. It would be unlikely, though, because we are using pretty general components, which should be the same across versions. We'll also use DataFrames from the pandas library to work with our datasets. My pandas version is 0. 22. We'll be reading all of our data into a pandas

DataFrame using the pd. read_csv function. The white wine quality dataset along with all other datasets in this course will be in the datasets folder, which is under the current working directory. We assign the column names to our DataFrame. Here are various characteristics of our white wines--the acidity, the residual sugar, the chlorides, sulfur dioxide, and so on. And at the end, we have a quality score assigned to the wine. We've skipped the header row in our CSV file and specified our own column headers. Let's explore this data using the head method on our DataFrame. This will display the top five rows. Here we have the various columns which define the various characteristics of our wine. The very last column is a quality score that has been assigned to every wine. Wine quality ranges from 3 to 9. We can examine the last few records in our dataset by calling the tail method on our DataFrame. There are nearly 5, 000 records in this dataset, a good number to work with. The describe command on our DataFrame displays basic statistics across individual features in our data such as count, mean, standard deviation, and what values lie on the various percentiles. This is a great way to quickly get a sense of your data. If you eyeball the quality column here, you can see that wine quality scores range from 3 at the minimum, and a 9 is the maximum score.

Matplotlib and Seaborn

When you first starting working with visualizations in Python, it's quite likely that you'll use Matplotlib. Matplotlib tries to make easy things easy and hard things possible. Matplotlib allows you to generate a wide variety of graphical representations of your data such as plots, histograms, bar charts, error charts, and so on. All it needs are a few lines of code. The Seaborn visualization library takes the ease of visualizing data to the next level. It's built on top of Matplotlib. It's tightly integrated with the PyData stack. It has support for NumPy and pandas data structures and for statistical routines from SciPy and even StatsModels. Seaborn works directly with other statistical tools that you might be using when you're analyzing data. Let's take up a side-by-side comparison of Matplotlib and Seaborn to see how they're different. Matplotlib is a part of the PyData stack, the open data science stack available in Python. Seaborn is built on top of Matplotlib and tightly integrates with PyData. The whole idea behind Matplotlib is to make very difficult visualizations possible, which means you have very fine-grained control over what exactly you plot and how you depict your data. Seaborn tries to make things easier for the data analyst and the data scientist by providing high-level, easy-to-use abstractions for the most common use cases. You'll find that Seaborn simply takes the data and produces the result that you expect without you having to manually specify small details. If you've used Matplotlib, you're aware that it has two APIs--the Matplotlib API, which is very low level, and the Pyplot API, which

is much higher level and provides more abstraction. Seaborn takes things a step further. It's even higher level than Pyplot and, in fact, you'll use Pyplot and Seaborn side by side even in this course. When you're using visualizations in productions either on your production website or as a part of a presentation to accompany higher ups, you want to have production-level aesthetics. You want the graphs to be aesthetically pleasing and laid out exactly so. This is definitely possible in Matplotlib and requires use of the low-level API, which is a little more difficult and detail oriented. As far as Seaborn is concerned, though, production-level aesthetics are possible with abstractions without using very fine-grained API controls. Let's visually examine where Seaborn fits in with other Python libraries. Seaborn is a Python package which you'll install using pip install, and it's built on top of the PyData stack. Within the PyData stack, Seaborn directly works with Matplotlib. In fact, it's built using Matplotlib. Seaborn works directly with data that is represented as pandas DataFrames or as a NumPy array. The Matplotlib package comprises of higher-level APIs such as Pyplot and the Pylab modules. Both of these are built on top of the object-level APIs which give you very fine-grained control over your visualizations, the Matplotlib API. Notice that Seaborn lives on the very top of this stack. These are very high-level APIs. The Matplotlib API is much more granular and is used to power Seaborn. Seaborn is dependent on the entire PyData stack, and you'll find that different libraries have to be installed from within PyData before you can run Seaborn. Seaborn's tight integration with pandas and NumPy makes it very easy to feed in data for your visualization. You don't have to convert it to any other form. You'll be working with data in a pandas DataFrames. You'll just feed it indirectly. If you've ever worked with Matplotlib, you know that it's a very complex package. It has many modules within it. We have the low-level Matplotlib APIs, which offer fine-grained control over every object that is plotted onscreen. Typically people work with Matplotlib using the higher-level Pyplot API, which controls the state machine for visualizations. You might have also used Pylab, which is the convenience module that allows you to pull in different objects which make up a visualization into a single namespace. Plotting operations typically follow a hierarchy. We can either be very, very specific as depicted by the left side of this line or very, very general on the right side of this line. "Color this pixel red" is a very specific operation. "Contour this 2-D array" is a very general operation. Operations that are very granular or very low level act on specific plot elements, a particular axis, a particular pixel. High-level operations act on the figure or the plot as a whole. If you've studied Matplotlib in a structured manner, you'll see that this hierarchy is formalized in its codebase. Here is the general layout of the visualization hierarchy within Matplotlib. We start off with the figure at the very top. The figure is made up of the canvas and the axes. The axes can be X or Y, and each axis has ticks. All of these individual elements in Matplotlib are artists. And artists are arranged in a hierarchy. The artist is in fact an abstract base class within Matplotlib. The figure that you see at

the very top of this hierarchy is a container class, which contains the other artists. The figure is the top-level container, and the other artists are granular plot elements. Pyplot APIs and even Seaborn APIs operate at the higher levels--at canvas and axes levels or at figure levels. Matplotlib APIs operate at lower levels, at the level of the single axis or ticks within an axis.

The Kernel Density Estimation (KDE)

Our first Seaborn visualization will involve plotting univariate data or data represented using one variable. For this, we need to understand something called the kernel density function, which we'll do right now before we plunge into the visualizations. Now you can make a statement like "Michael Jordan is a once-in-a-lifetime player. " This statement is obviously true. Michael Jordan is an outlier in the world of basketball. If you plot basketball skill along a number line, you'll have people who are really bad at playing basketball at the very left, ordinary folks somewhere in the middle, NBA players at the very right, and Michael Jordan at the extreme right indicating he's an outlier. Outliers are players who are very far from the pack. The point is represented far away from the other. If you plot basketball skill as a histogram, this is the shape we generally get. The skill levels are along the X axis, and you'll find that most people have very ordinary levels of skill. They would be clustered around an average kind of at the center. The NBA players would be outliers. They would be at the very right of this histogram where there are fewer players at that skill level, and Michael Jordan would be an even greater outlier. He would be at the extreme right of this histogram. If you have any level of statistical knowledge, this shape should be very familiar to you. This is a chart that you use to tell us how common a specific level of skill is. The shape of this chart resembles a bell. This is called a bell curve or a normal probability distribution. Instead of representing this as a histogram of values, you'll often draw a small curve, a bell curve, or the Gaussian curve. This bell curve gives us a lot of information just at a glance. You can see that the average skill level is very common. Very high and very low skill levels are both unusual. This bell curve occurs everywhere in nature, not just for basketball skill levels. For height distributions, weight distributions, intelligence, everything. Now going back to our basketball distribution, you might ask yourself, What is the probability of any specific value X occurring in the data? And this answer you get using a probability distribution function, which tells you how the various points for that data are distributed. Here is some statistical analysis that is really common. You have different categories plotted along the X axis, and then you have a histogram where every bar represents a count or a frequency at a particular category. This represents your set of points. What you really want to do is to figure out a probability distribution of those points. You want to find the probability, P, with which a particular value, X, occurs in this set of points. Given that this

set of points represents your entire dataset, the area under the curve that you see onscreen must sum to 1, the total sum of probabilities of occurrence of individual data points in this set. And this is where the kernel density estimation, or the KDE, comes in. This is a standard technique to find the probability distribution of points in your dataset. This is a non-parametric technique, which means you have no parameters to represent assumptions in your data, and it's also a smoothing technique. The result will be a smooth curve. In order to calculate the kernel density estimation, or the KDE, curve, we assume that all the points in our dataset have the same distribution. They're independent identically distributed points, which means that you can draw a bell curve with the same standard deviation around every point in your dataset. Each of these points will have its own bell curve with the same identical distribution. So imagine all of these bell curves with the same standard deviation centered across each of your points in your dataset, so you have a stream of bell curves in this way. You'll now sum them all up. You'll use an integration to sum up all of these to get the resultant probability distribution function of your data. This result in probability distribution function is the kernel density estimation. The assumption here is that the individual points in our dataset are independent and identically distributed. They all follow the Gaussian probability distribution, which is defined by the mean mu and the standard deviation sigma.

Visualizing Univariate Distributions: Histograms, KDE Plots, Rugplots

We are now ready to study the first of our Seaborn visualizations. We'll see distributions KDE plots, joint plots, pair plots, and head maps. Remember, as a data analyst or a data scientist, your main job is to explore patterns within data by using visualizations. We start off by importing the various Python libraries that we need, the Seaborn libraries, pandas to read in your data's DataFrames, and Matplotlib. We'll work with the wine dataset that we explored in an earlier demo. We'll start off by viewing the most simple of distributions, univariate distribution where just one variable is involved. Here we use the distplot function, or the distribution plot function on Seaborn, to see the alcohol content across the various wines. The distplot function in Seaborn plots the histogram of values at the various alcohol content levels, which is represented on the X axis. Seaborn automatically calculates the right number of bins used to represent the alcohol content, and the histogram bars indicate what proportion of the wines in our dataset fall within a certain bar. Each bar corresponds to a range of alcohol content or a bin. You can observe from this histogram that around 40% of the wines in our dataset have alcohol content of about 9. 3%. The distplot made before plots the histogram, as well as the kernel density estimation of our data. Both of these are the most standard techniques to visualize univariate data. If you want your

graphical visualizations to be of a certain size, you'll use the Pyplot library within the Matplotlib package in order to specify a figure size. This should make it very clear that Seaborn basically uses Matplotlib under the hood. Now we are going to plot a distribution plot once again. But this time we do not want the kernel density estimation curve. So we specify KDE equals to False. This gives us a same histogram that we've seen before but this time on a larger figure size, and you can see that most of the wines have alcohol contents clustered in the 9-11% range. Now within this histogram, you can also view the individual data points that exist within every bin plotted as a stick, and Seaborn allows you to view your rug plot as a part of the histogram by specifying the rug parameter to True. KDE is still False here. And here in the result, we see the same histogram that we've seen before, but notice that along the X axis, individual data points for each wine record are plotted as sticks. You can see the correlation between the height of the histogram bars and the number of sticks or the number of data points within that bin or range. If you're not interested in the histogram, but you're only interested in the rug plot, you can use the rugplot function for the same data. Here the rug plot will be along the same axis with a height of 0. 5. And here is the rug plot visualization of our data. You can see that we have wines which range from 8% alcohol all the way to about 14. 5%. And you can see that most of the wines are clustered around the center between 11 and 12%. When we set up the distribution earlier, the histogram bins along the X axis were automatically calculated by Seaborn. To have more control over this, you can specify the number of bins you want to see. Here the bins parameter is 30. The resultant histogram bars are relatively thick. If you're interested in seeing more granular reveals of this information, you can simply increase the number of bins you want to view. For example, bins equal to 75 will allow us to see much more detail on the same histogram plot. The bins cover a much smaller range of alcohol content here, and you can see that some bins have no samples for example. There is no wine within our dataset that has around 9. 9% alcohol. There are many different ways in which you can view the same distribution using the distplot here if we don't want to see the histogram but we do want to see the rug plots. That means the result will be a kernel density estimation function and a rug plot. In order to shade the area covered by the kernel density function, you can set color_codes on Seaborn and then plot your data using a KDE plot. The KDE plot is used to show only the kernel density curve. We specify the parameters shade is equal to True and color is equal to "r" to get a red outline with a red shade on your resultant plot. In order to use color in this manner, you need to set color_codes is equal to True on Seaborn. Let's now plot multiple KDE curves on the same figure. Here we have around five different KDE plots, each of them with a different bandwidth. The bandwidth parameter of the KDE curve determines how tightly the estimate of the kernel density function fits the original points. Because we have multiple curves on the same plot, we now need to associate a label with each curve. This

will set up a legend. The resultant figure has different KDE plots, all of which are in the same figure. Each of these curves has been drawn in a different color to allow us to differentiate between them. Notice that for very low bandwidth, you can see that the KDE curve is much more accurate, but it crowds the data points. It's very hard to see general trends when we have such a non-smooth curve. For very high bandwidth values, you can see that the curve is too broad, and it doesn't represent the underlying data very well. You can see that the default value that Seaborn has chosen for our KDE curve is somewhere in between. This is a good representation of our data. This is the KDE curve that we saw in earlier plots.

Visualizing Bivariate Distributions: Jointplots, Hexbin Plots, KDE Plots

Visualizations are often used to depict relationships. Relationships require two or more variables. The jointplot function in Seaborn can be used to represent bivariate distributions and relationships between two variables. For every line in our dataset, we want to see the relationship between the free sulfur dioxide and the total sulfur dioxide. Notice we specify the X and the Y axes with different columns from our wine DataFrame. The jointplot function in Seaborn produces a scatter plot. The Y axis is the total sulfur dioxide, and the X axis is the free sulfur dioxide. The points representing individual wines are scattered in the XY axis. This is a scatter plot. The interesting thing here is that you get the distributions as well. Along the X and the Y axes, you get the individual univariate distributions in the form of a histogram. You can view this scatter plot better by specifying limits along the X and Y axes. We want the X axis to range from 0 to 100 and the Y axis from 0 to 250. This results in zooming in on our data. We can view the scatter plot more closely. If you want a specific visualization or figure to have a certain color palette or style, you can use the with keyword. Here we want the axes_style to be white. We want a plot on a white background. We'll study these themes and color palettes in much more detail in a later module of this course. We'll visualize the joint plot on the same data as before, but we'll make this visualization much more interesting by viewing it as a Hexbin plot. A Hexbin plot is basically a histogram representation of bivariate data. All the histograms being seen so far have been for a single variable. In the resulting visualization, we have a scatter plot exactly like we did before, but this time, there is a hue variation on individual points of this scatter plot. How light or dark a particular point is represents the height of the histogram at that point. If you look at the histograms that we have on the X and the Y axes, which represents the free sulfur dioxide and the total sulfur dioxide, you can see that the peaks of those histograms correspond to the darkest points on our joint plot. Every point is a hexagon with six sides. This is a Hexbin plot. Remember

that the histogram represents the frequency of points within a particular bin. Many of our wine examples have a free sulfur dioxide content of about 34 to 40. And, correspondingly, the total sulfur dioxide content is in the range of around 120. You can use the joint plot and have it plot KDE curves instead of scatter plots. Simply specify kind is equal to "KDE" when you call your joint plot. And here is our resultant KDE curve. For the bivariate distribution, it's along the XY plane. For the individual univariate distributions, it's along the X and the Y axes. KDE curves can also be drawn using the kdeplot function on Seaborn to which you can specify the clip parameter to limit your X and Y axes. The X and Y axes in your resultant KDE plot are limited to 250. If you want to limit each axis individually, you'll use the xlim and ylim functions on the Pyplot library. In the resulting visualization, you have the X axis limited to 100 and Y axis limited to 250. You can also plot multiple distributions on the same figure. We will set up a KDE plot for our bivariate distribution showing the relationship between free sulfur dioxide and total sulfur dioxide in our wine_data. We'll also have a runplot to mark individual points in our wine dataset. We'll represent the free sulfur dioxide content with sticks which are green in color. And we'll represent the total sulfur dioxide with sticks which are the default Seaborn color but on the vertical axis specified by vertical=True. We also indicate specific limits for the X and Y axes for this plot. Our KDE plot for the bivariate distribution is on the XY plane. The rug plots for the univariate data are on the specific axes, either X or Y. The X and Y ranges on the axes are constrained by the limits that we specified.

Pairwise Relationships Using Pairplot and Correlations Using Heatmap

Now our wine dataset has a number of different columns. And every pair of columns could have a different relationship. What is a quick way to see all of these relationships in one go? You can use a pair plot. A pair plot allows you to plot bivariate relationships between every pair of columns in a dataset. The result of a pair plot is a matrix grid of plots with every possible combination of columns on the X and the Y axes. There are 12 columns in our wine dataset giving us a 12 X 12 matrix. The column headings are the rows and columns of this matrix. When you zoom in here, you can see the column headings which form the rules. Notice that along the main diagonal, we have plots that are histograms. These are plots of the univariate variables. Along the main diagonal, the X and the Y variables are the same, which is why we get univariate plots. All the other plots which are not on the main diagonal represent bivariate relationships. The row categories form the Y variables, and the column categories form the X variables. The column values are printed at the very bottom here. You can see that the bottom left-most plot is a plot of

the quality score versus the fixed acidity of the wine. For the entire bottom-most row, the quality score is the Y variable, and the fixed acidity, volatile acidity, citric acid, residual sugars, etc., are the X variables. Now it's possible to get a more manageable pair plot. Let's say we're interested only in a few columns within our dataset. We specify only those within our pair plot. Within our wine_data, we want to see the pairwise relationships between the fixed acidity, volatile acidity, citric acid, and the residual sugar. And the result is a 4 X 4 pair grid matrix. Notice that the row values are the various variables, the four variables that we were interested in, as are the column values. The plots on the main diagonal are univariate plots, histograms, and the plots on the other grid cells are scatter plots by variate relationships. Let's say we want to view the pairwise plot of specific columns, but we want to diagnose that it's the univariate variables to be plotted as KDEs. We can use the pairplot function in Seaborn and specify that the diagonal kind should be KDE. Notice in the result that the off-diagonal elements are still scatter plots, but the mean diagonal, the univariate variables, are represented as KDEs, not histograms. Let's saw we want to see the pairwise relationships, we use a pair plot as before, but the scatter plots we want represented using a regression line fitted to our data. The kind input argument to our pairplot function should be equal to "reg" for regression. All our bivariate relationships are represented as scatter plots with the regression line fitted. We'll study more about regression plots and linear relationships later on in this module. So far, we've used the X and Y axes to model bivariate relationships, but Seaborn allows us to use another dimension to represent data, and that is the hue of the plot. We set up a pair plot to model pairwise relationships. We've set up a third dimension here. The pH value of our wines will be represented using the hue. This plots an extremely colorful matrix. The hue represented by our scatter plots and our histograms represent the pH value of the wine. The legend on the right indicates the full range of pH values. High pH values are represented by colors which tend towards pink and purple. Another way to view relationships between pairs of columns in your dataset is the heat map. The heat map is used to represent how correlated individual features in your dataset are. The input to a heat map is the correlation matrix between every pair of columns. This can be set up by calling the corr function on pandas. The correlation matrix is then passed in as an input to the heatmap function in Seaborn. We want the individual cells in the heat map to be represented as squares, and we want the heat map to be annotated so we can see what the correlations are. You can specify different color maps or themes for your heat map. We've chosen the summer theme. The heat map is a grid or a matrix that shows the correlation between every pair of features in your dataset. The individual features in your dataset are represented as rows and columns in your heat map. Along the main diagonal, you'll find that the features are perfectly correlated with a correlation value of 1. 0. That's because along the diagonal, we see the correlation of each feature with itself. Highly correlated features are

represented by yellows in our color scheme. You can see that within our wine dataset, there are other features that are correlated as well. For example, density is highly correlated with residual sugar. The heat map is an extremely useful tool when you explore your dataset. This will give you an idea of whether you need to perform dimensionality reduction on your data before you pass the data in to a machine learning module. If the features within your dataset are highly correlated, then dimensionality reduction makes sense and might improve your machine learning model.

Regression Plots Using Lmplot

In this demo, we'll continue modeling relationships between features. We'll see how we can use lmplots and regplots for linear relationships where we want to fit a regression line. We'll see how we can control the size and shape of plots and how we can use combination plots. We'll start off by visualizing linear relationships. We'll work with the same wine dataset that we've used so far. We'll read it into a pandas DataFrame. The lmplot function in Seaborn allows us to fit linear models on our data. In the last clip, we saw that residual sugar and the density of our wines were highly correlated. It could possibly mean that there exists a linear relationship between them. We can explore this using the lmplot. The size and the aspect input argument allow us to control the size of the plot. We'll study that in a little more detail in just a bit. We are modeling a bivariate relationship between density and residual sugar, which is why this is a scatter plot. Notice the regression line that has been drawn through the points of this plot. This regression line is also used to represent the confidence interval of our regression. At the very bottom left of this regression line, the confidence interval is narrow as the points are very close to the regression line. We have high confidence in the line here. At the top right here, you can see that the confidence interval in the regression line is much broader. The original points are much further away from the regression line here. The line at this point is a worse fit on the underlying data. So far, most of the relationships that we modeled have been with continuous data. We haven't worked with categorical variables much. Categorical variables are those which take on discrete values, not values within a continuous range. Days of the week, months of the year are all examples of categorical variables. Here our X variable is the quality of the wine, which takes on discrete scores. It's a categorical variable. This is a regression plot to see how the alcohol content varies by quality score. The resultant scatter plot in the lmplot has very clear clustering of data around the categorical values. You can see from this plot that it's very hard to see the individual data points for every quality score from 3 to 9. A common technique that is used to view the individual data points that are clustered around a categorical value is to artificially add some jitter which spreads the points when we visualize it on a plot. X_jitter spreads the points around the

variable values on the X axis. That's where our categorical variable is represented. It's important to understand that jitter is merely a visualization mechanism. The actual regression line that is fitted on the data points is not affected by jitter. For a better estimate of the wine alcohol content at every quality score, we can estimate the mean alcohol content as well. This we do using a NumPy function. NumPy is a scientific computing package in Python that supports large multidimensional arrays and matrices. We can calculate the mean at every discrete value of a categorical variable under X axis using the x_estimator. We now have a very interesting regression plot. For every quality score, we have the mean and the standard deviation represented in the plot. This particular visualization and regression line allows us to make the statement, "Higher quality wines tend to have higher alcohol content. " Let's use some interesting combination plots using lmplot. We'll start off with a very simple regression line here. We want to plot the pH values of the wines along the X axis and the acidity along the Y axis. This information is interesting, but this can be made more interesting by figuring out this relationship for every quality score--plot fixed acidity versus pH for different quality wines. This will plot a row of graphs where the columns in the row will represent quality. And here are the relationships between fixed acidity and pH value for each quality score. If you zoom in here, you can see that the column headers are quality score 3, 4, 5, and so on. Remember, any graph can represent data in three dimensions--the X axis, the Y axis, and the hue. We can represent the alcohol content in the wines by using the hue parameter. The result is the same plot as before but with an additional color dimension representing the alcohol content of the individual wines. Whenever you use the hue parameter, Seaborn is kind enough to give you the accompanying legend. Instead of having these plots laid out in a single row where the columns represent different quality scores, you can specify the quality scores along the row as well. This is the exact same plot as before, but each quality score is represented on a separate row.

Regression Plots Using Regplot

Seaborn offers a different function, an alternative to build regression plots called a regplot. The regplot accepts variables in a variety of different formats such as NumPy arrays, panda series, DataFrame variable references, and so on. The regplot is an axes-level function as opposed to lmplot, which is a figure-level function. Lmplot operates at a higher level of the Matplotlib hierarchy. The lmplot function that we studied earlier is much more powerful and offers much more granular control over your plots. A regplot has only a subset of the features specified by lmplot. You can see that the output by the regplot is essentially the same as what we got with the lmplot. The figure representations are a little different. The regplot can also accept its X and Y

variables, input in the form of a pandas series. We are accessing the individual series for alcohol content and density here. You can also access the individual column values within the DataFrame using the dot notation like you see here. Seaborn visualizations use the Pyplot APIs in order to control the size and shape of the plot. The axes that we have specified using Pyplot can be passed in as an input argument to the regplot function. A different want to control the shape of the plot is by specifying the size, as well as the aspect ratio of your visualization. The size refers to the height of the plot, and the aspect value multiplied by size gives us a width of the facet. That is the area where we view our plot. We studied the joint plot before for bivariate distributions. We can also specify that the joint plot draw a regression line by specifying kind is equal to "reg. " Here is the resultant scatter plot with the regression line fitted. Notice the univariate distributions on either axis, the histograms, as well as the KDE curve. You can also specify the pair plot with a regression line. The pair relationships that we want to view are specified as X and Y variables, and the kind of plot is regression. The result is a 1 X 3 matrix of plots, each a scatter plot fitted with a regression line.

Stripplots and Swarmplots for Categorical Data

In the last demo of this module, we'll try to cover a lot of ground. We'll study categorical plots, box plots. We'll see how we can perform statistical estimation within categories. We'll work with wide form data and factor plots as well. We'll continue working with the wine dataset here, read in and stored as a pandas DataFrame. We initialized the basic Seaborn style to wide grid. We'll study all of this in much more detail in the last module of this course. And we set color_codes to True. We also set up a random. seed for statistical analysis. When a particular variable that we want to plot is a categorical variable, rather than use a regular regression plot, we can use specific plots that are tailored to categorical values such as the strip plot. A strip plot is basically a scatter plot for categorical data. When you visualize data in a strip plot, you'll automatically have each categorical value plotted in a different color. In order to clearly view the individual data points at every categorical value, we can set the jitter argument to True in our strip plot. Jitter causes our individual data points to be spread out at every categorical value. This gives us a better idea of the density and the number of wines at each quality score. Adding jitter makes it very clear that there are very few wines in our dataset at the lowest and the highest quality score, 3 and 9 respectively. An even better way to view categorical data is by using swarm plots. Swarm plots spread out the points at every categorical value so that the overlap between the points is minimal. Swarm plots allow us to view every data point at each categorical value. The colors in a swarm plot are chosen at random. Notice that in spots of random jitter, every point is deliberately

spread out in a swarm plot. Just like with other plots that we've seen before, swarm plots can also use the additional hue parameter in order to view the relationship with another variable. Here the hue represents the pH value of the wine. You can see in the resultant swarm plot here that most of the wine data points have the greenish tinge or the greenish hue, which represent wines with a pH value of between 3 and 3. 2. You can also mix up your swarm plots a little bit and have a categorical variable be represented on the Y axis. The Y axis has the quality score, and the X access has the alcohol content of the wine. In the resulting swarm plot, you can see that the data points have been spread out along the width rather than along the height of the dimensions. With this setup, there is more room to maneuver along the width.

The Boxplot and the Violinplot

A very useful and extremely common way to visualize statistical data is by using box plots. Box plots serve to highlight the values at the various quartiles of any dataset. Box plots in Seaborn can be invoked using the boxplot function. Here is the box plot representation of the alcohol content of wines at every quality score. The rectangular boxes that you see in different colors represent the range between the 25th and the 75th percentile of data. The heights of our rectangular boxes represent the interquartile range of our values. These horizontal lines that you see at the top and bottom of your box plot represent the whiskers of the box plot. The whiskers extend two points that lie within 1. 5 times the interquartile range. Any individual data points which lie beyond the 1. 5 times the interquartile range or beyond the whiskers are represented individually. These are the outliers in our dataset. Another very common way to view statistical data is by using violin plots. Violin plots represent the same information that box plots do. The visualization is, of course, different and can be invoked using the violinplot function in Seaborn. The violin plot is so called because of the shape of the individual representations. Each is in the form of a violin. The curved boundaries, in fact, represent the kernel density function. This outer shape represents all possible points within a particular quality score. The thickness of the violin plot at any particular Y value represents how common a point is at that value. By using the KDE, violin plots represent the probability density of our data at different Y values. The whiskers of our plot, which are at a distance of 1. 5 times the interquartile range, are also part of the violin plot. The thickest portion of the violin at any quality score is the mode of our data, the mode of the alcohol content in wines. Seaborn also allows you to customize your violin plot in interesting ways. You can scale the width of the violin to be the number of observations in a particular bin by specifying scales equal to "count. " You can see the shape of the resultant violin plot is different because the thickness of the violin represents the counts of data at that value. Just like we did

when we used a rug plot, you can also add individual observations within a violin plot by specifying inner is equal to "stick. " We've also changed the representation of the violin plot by using a different palette. In the resultant visualization, you can see that the lines, the horizontal black lines are the closest together at wine quality 6, so there are many samples of data with quality 6 in our dataset. You can also combine multiple plots in the same figure. Here we have a violin plot and a swarm plot on the same data. They'll be combined together into one graphical representation. If you remember, the swarm plot spreads out the data points at each categorical value so that overlap is minimized. The swarm plots points are within the violin plot's area.

Statistical Estimation and Factorplots

In this clip, we'll see some examples of how we do statistical estimation within categories of our data. We'll start off by specifying a very simple bar plot, but we'll use it to represent relationships between three variables. On the X axis, we have the quality score. The Y axis is the pH content of our wines. And the hue is used to represent the alcohol content of wines. The result is actually a pretty complex visualization. Let's break this down. We have the quality categorical values on the X axis and the pH values on the Y axis. For every quality score, we have a separate bar plot, and the hue of the bar plot represents the alcohol content. In addition to the hue, certain bars have black tips. These are the outliers which lie outside the interquartile range for the pH value. Let's say we want to count the number of data points or the number of wines in a particular category. We can use the X axis to represent the category and use a count plot for this. Here the X axis is the alcohol content. We want the count of the number of wines at every alcohol level. The result gives us a bar graph representing a count of wines at every alcohol content level. Lower levels of alcohol are also represented in the darker green color. We can see that there are many wines in this range between 8 to 9. 5% alcohol. Notice that our X variables have been written out vertically for better readability. We rotated the xticks on our plot by 90 degrees. A point plot in Seaborn specifies an estimate of the central tendency of a variable and the confidence around this central tendency. Point plots can be much more useful than bar plots for comparing between different levels of one or more categorical values. Here we have the categorical data that is the quality score on the X axis. We also have the point estimates and the variation for the pH values for each quality score. Notice that the point estimates are connected across categorical values allowing us to compare the levels easily. We can plot a much more complex visualization by adding a third dimension--the hue represents the alcohol content of wines. The resultant visualization is far more complicated as you can see. Here are point estimates of the pH value for each quality score for each alcohol content percentage. The point estimates with the same hue are connected together.

The same hue represents the same alcohol content. It's hard to get much information from this visualization because it's too complicated. However, you can see how the alcohol content varies with the pH and the quality of wine by looking at the connections. Wide-form data is when repeated responses are represented in the same row and different responses are represented in different columns. Here we'll see a horizontally oriented box plot to represent wide-form data. Every characteristic of the wines in our dataset has been represented using a box plot. This is wide-form data because the different features form different rules. Along the columns, you see the variation in data for each feature. A factor plot can be used to represent categorical plots in a matrix format, a facet grid. We'll study facet grids in much more detail in a module which is coming up. A facet grid is basically a way to represent the same relationship between two variables under different conditions. This factor plot represents the same information as the point plot that we saw earlier. For each quality score, we represent the pH values of the wine, and the hue represents the alcohol content. The default is the point plot within the factor plot, but the factor plot can be used to plot other kinds of representations as well. Here is another example of a factor plot. On the X axis, we will plot the quality score of the wine. On the Y axis, the pH value. Hue is used to represent alcohol content, but this is a bar plot. Here is the same relationship that we saw earlier that we plotted using the barplot function. The same factorplot function can be used to visualize different categorical representations. Here is another factor plot example. This time we want to plot a box plot where the X values are the fixed acidity, Y values are the pH values of wine, and the columnar representation is the quality of the wine. We need to zoom in here to see the individual box plots for the various quality conditions. Here is another example of the factor plot once again, but this time we want to plot a swarm plot for the individual facets. The visualizations are kind of hard to see here. We zoom in, and we can see for different quality scores, we have swarm plots. And with this, we come to the very end of this module where we introduced the Seaborn visualization library. We saw that Seaborn is built on top of Matplotlib and uses several elements of Pyplot and other Matplotlib packages. Seaborn is tightly integrated with the PyData stack and, in fact, accepts input in the form of NumPy arrays or pandas DataFrames. Matplotlib allows very granular control over how data is visualized and represented. Seaborn is a higher level abstraction. In fact, it's a complement that can be used along with Matplotlib and Pyplot libraries. Seaborn abstractions make common use cases very simple and get production-ready plots with a single function call. In the next module, we'll study how we can build trellis plots in Seaborn, which allow us to represent data in a matrix format where every cell of the matrix represents a visualization of a relationship between different pairs of variables.

Building Trellis Plots in Seaborn

Module Overview

Hi, and welcome to this module on Building Trellis Plots in Seaborn. Trellis plots allow you to graph multivariate data, data with multiple variables across multiple plots which are displayed in a grid format. In this module, we'll work with two types of trellis plots. The first is the FacetGrid, which allows us to visualize relationships between multiple variables separately. FacetGrids are a way to view conditional relationships by setting up a matrix of plots. Every cell in a plot satisfies a certain condition, and you can view relationships between other data based on that condition. The PairGrid, on the other hand, is also a way to view relationships across multiple variables but in a pairwise manner. So you can specify a bunch of column pairs, and you can view all the relationships laid out in the form of a matrix.

Working with Facetgrids

You might have observed that certain plots that we saw in the last module were FacetGrids. FacetGrids are exclusive objects in Seaborn which allow you to link the various columns in a DataFrame to a Matplotlib figure. FacetGrids allow you to set up a matrix that is rows and columns of plots where every cell depicts a relationship within a certain condition. You assign different conditions to the rows and columns of this matrix grid, and within each plot, you can use color to depict variations. In this demo, we'll work on a Facebook dataset. You can imagine that you're the social media manager of an organization, and you're trying to figure out what kind of Facebook posts get the most response with your audience. As usual, we'll use pandas to read in the CSV file and store it as a DataFrame. Here is the information that you have on each Facebook post. You know the total number of likes the page has got, what kind of post it is, whether it's a status post, a post with a photo, a post with a video, or a link. You know which month it was posted, on what day of the week and hour of the day it was posted. You also know whether or not you paid to promote this post. A value of 1 indicates that you paid for this post. You also know what the reach of the post was, how many people engaged with it, how many impressions that post had. That's a lot of information that's used on the DataFrame in order to understand the basic structure of the data. We can see that we have 500 records in this DataFrame count of 500. By analyzing this Facebook data, one of the important things that you want to study is the

characteristics of a post based on its type, whether it's a photograph, whether it's a simple post, whether it has a link. You set up a FacetGrid for this. The FacetGrid will operate on the Facebook data. That's one of the input variables to the FacetGrid. The second variable is the column. We want the individual plots or the cells in the FacetGrid to represent some information about one type of post. And here is our FacetGrid. There are four types of posts, which is why we have four cells within this grid. The four types of posts are photos, status, links, and videos. We haven't specified what is the information we want to view within a cell, which is why each plot is empty. This time we instantiate the FacetGrid once again. In addition, we use the g. map function to specify what characteristics we want to view on every grid cell based on type. Every column in our FacetGrid is one type of post, a post which might contain a photo, a status, a link, or a video. For every kind of post, we want to view some univariate data, which is why we use a histogram. We want to plot a histogram of the total number of interactions that users have with that post. You can clearly see here that photos garner a very high response from our users, but the other kinds of posts, such as status, videos, etc., are not really that popular. In order to view the details a little better, we need to zoom in on individual plots. And this we can do by specifying limits for the X and Y axes. And here is the same FacetGrid once again, but each individual plot is much more zoomed in. There is much more detail that's visible here. The total interaction bins are specified on the X axis on this histogram, and you can see that photos garner a far more favorable response from our users than other kinds of posts.

Facetgrids with Regplots

In the last clip, we saw how we can use the FacetGrid in order to univariate data. You can also use the FacetGrid to plot bivariate data, that is, relationships between two variables. In this FacetGrid, we still want to break down our posts by type, but for each type of post, we'll use a scatter plot to depict the relationship between the total reach of a post across its lifetime and the lifetime engaged users on the post. Both of these are columns in our DataFrame. And here is our scatter plot with this information on each type of post. You can see at a glance by looking at the plot for photos that it has great reach, and some particular photos have lifetime engagement with users. An interesting thing here is you can see that some of the status posts may not have reached a lot of users, but they have good lifetime engagement. There is one more dimension on this data that we can view. We specified different values for the X and Y axes, but we haven't used the hue of the plot yet to get information. We'll now specify different hue values to represent whether a post is a paid post, which we paid to promote, or a free post. All you have to do is specify the hue argument to the FacetGrid indicating what column values need to be used within your

DataFrame. While setting up a scatter plot, we also specified an alpha value of 0. 7. This reduces the opacity of the points on the graph so that we can see overlap between points more clearly. Once we've got the hue or different colors representing the different pieces of information, it's useful to have a legend, which we can get by calling g. add_legend. This information on paid posts has actually proved very useful for us. You can see that there are some posts which are outliers. We've paid to have those posts promoted, and they've reached a huge number of people and have great lifetime engagement. So far, our FacetGrid has had just four columns depicting the various types of posts. We are going to add an additional row parameter here which will display the characteristics for a post type and a particular category. We'll continue to use the hue of the scatter plot to depict whether we paid to promote a particular post or not. Here we use a regression plot to depict that analysis. We want to see the relationship between the month a particular post went out and the total interactions with that post. We are using a regplot, but we do not want to fit a regression line on the data, which is why fit_reg is False. We'll add a legend and specify some limits on the Y axis so that we can view individual plots better. And the result is a matrix of plots. Every column in this matrix still refers to one type of post, whether it's a photo, a status, a link, or a video. However, we've used the rows to split the posts based on category. Category 1 is row 1, category 2 is row 2, and these titles on the right can be seen because we've set margin_titles to True. For every type and category of post, we are plotting the total interactions against the month in which the post was released. Just a glance at this matrix will tell you that photos are pretty evenly represented across the three categories. We don't know what exactly those categories are, but we can see that the other types of posts are kind of rare across category 2 and 3. It's pretty clear that photos clearly have much more information across the three categories of posts as compared with status posts, links, and videos. This means that we might want to customize the FacetGrid, which displays this information, such that a column which is devoted to the plots for photos occupy much more space, that is, have a wider width in the resultant matrix. This we can do within a FacetGrid by specifying the proportions for the various columns using the gridspec_kws parameter. In this FacetGrid, the individual columns or the width ratios are 5 is to 3 is to 2 is to 2. So you can see that photos and status posts have the most space to work with then within this grid. Within each plot, you can view the paid and the free posts in specific colors. The h variable indicates that free posts should be indicated in red, and paid posts in green. This is the color palette that we pass in as an input argument to the FacetGrid. The rest of the FacetGrid setup remains the same. We still want regression plots of the month in which a post was made against the total interactions of users with that post. You can immediately see that the look and feel of our FacetGrid changes when we use the red/green colors instead of the default hues that Seaborn uses. Instead of just the round dots, we can also use different shapes to

represent free and paid posts. Here we want the free posts to be red in color and paid posts to be green. We also want to represent the scatter plot points using triangular markers, the upward-facing triangle and the downward-facing triangle, which we specify by the caret and the v. Make sure you pass the marker and the hue information as input arguments to the FacetGrid. And here is the result, free and paid posts marked by triangles.

Facetgrids with Barplots

Let's analyze our Facebook posts based on the day of the week. We still want to view the individual types of posts in different plots. You can ignore this warning that you see onscreen. Basically, Seaborn is asking you to explicitly specify the order for the bar plot instead of inferring it from the data objects. However, for our demo purposes, you can simply have Seaborn infer the order for you. Here is a bar plot where the X axis is the weekday in which the post was posted to Facebook, and the Y axis is the total likes our page got. If you look at the very last plot on videos here, you can see that videos tend to get far more popular towards the weekend, whereas status posts, which are the second plot, remain evenly popular across the week. We can also customize the order in which we view these plots. Let's say we are the most interested in videos. We want to analyze that first. We can specify a column order with video first, then photo, and then status, then link. And the resultant display gives us the plots in the order that we want. Let's slice this data that we have along a different dimension. Instead of viewing the posts by type, let's review the posts by month. The column parameter that we now specified to the FacetGrid is the month in which we posted. Individual plots are bar graphs where the X axis is whether the post was on a particular weekday, and the Y axis the total number of likes that our page generated. You can see that we have 12 plots, 1 for each month. The information is kind of hard to view. We need to zoom in a bit. Within a month across weekdays, there doesn't seem to be much variation on how much users like our posts. However, as we move towards the end of the year towards month 12, you can see that our posts are more popular than they were in the earlier part of the year. A previous example should have made it very obvious to you that having the month laid out in a single row made things very hard to read. And this is why you can specify a column wrap (col_wrap) parameter in FacetGrid so that the output is laid out neatly. It's much easier to read now. We have just four plots in a single row.

Customizing Facetgrids

Now that we've understood in some detail how a FacetGrid works, let's see how we can customize the grid in order to get the layout and the information displayed in the way that we want. The import statements are the same as before, and we'll continue to work with the Facebook data. We'll pretend we are social media managers of our organization. We need to analyze Facebook posts. We want to slice our Facebook data across individual months of the year from month 1 to month 12, which is why our column is Post Month. We also want a plot display to be customized. We want just four months' of data in a single row. Each individual plot is a bar plot. On the X axis, we specify the weekday on which the post was made, and on the Y axis, we represent the total likes that our page got. We want the bar plot to be of a specific color, which we specify in the color parameter. Here is a hex value for our color. The edge color we want to be red, and we want the line widths to be of a specific size as well. And here are our purple bar plots with the red edge, one plot for every month of the year with just four plots to a single row. We've seen how easy it is to change the color of our plots, so we change it once again. Our color is now a different hex value. The edge color is white. In addition, though, we want the plots to be laid out more neatly. We want to adjust the whitespace between the plots. We add extra space between plots along the width, as well as along the height using wspace and hspace. These space adjustments are for every subplot within our FacetGrid figure. In the result that is displayed, you can see that we have the same plots as before, this time albeit in a different color. But we have additional space between plots both horizontally, as well as vertically. In this next example, we set up the same plot as before with different colors. But this time, we want the labels on the X and Y axes to be customized. We want the labels to be different from the column names in our DataFrame. The X label is Posts every week rather than Post Weekday, and the Y label is Total likes rather than Page total likes. In the resultant FacetGrid, we see that the Y labels have been customized. They now say Total likes. And the X labels have been customized as well. In this next example, we'll once again change the look and feel of our FacetGrid. We want only three columns in our resultant display, and this time we want very specific ticks to be displayed on the Y axis of individual plots. On the Y axis, the only tick values that we want displayed are 40, 000, 80, 000, and 120, 000. We have just three columns in the resultant FacetGrid, and the Y axis displays only the specified tick values. What if you want to zoom in to individual plots on this FacetGrid? You can do this by specifying X limits, as well as Y limits. In the resultant FacetGrid, all our bar plots have been cut off at 80, 000. That is our Y limit. We get no information. We've actually lost some information here. We've no idea whether the likes for a particular post went beyond 80, 000. Our X limit is from 0 to 6. The center of the first bar is basically at 0, and the center of the last bar is at limit 6.

Working with Pairgrids

PairGrids allow us to quickly summarize the different relationships between different pairs of data that exist within our dataset. When used with a pandas DataFrame, a PairGrid will give you every possible combination of columns in the resultant matrix. The import statements are the same as before, and we continue to work with the Facebook data for our organization. We're interested in how our users engage with our posts. This time we're specifically interested in the comments they made, the likes, and the shares. But we want to see these relationships in a pairwise manner. We want to see how comments relate to likes, likes with shares, share with comments, all pairwise relationships. Because each relationship in this PairGrid is a bivariate one, that is, every relationship is between two variables, we plot this paired grid as a scatter plot. There are three columns that we've specified that we're interested in. The resultant PairGrid is a 3 X 3 matrix. Notice the rows. Shares, likes, and comments are on the Y axis for every row. The same three variables--comments, likes, and shares--are also present along the columns or along the X axis of each plot. This is a PairGrid with three elements along the diagonal. Just two out of these three elements are shown and highlighted onscreen here. All three are univariate plots, comments plotted against comments, likes plotted against likes, and shares plotted against shares. All the other plots on this PairGrid which are not on the main diagonal are scatter plots. They are bivariate plots. Now if you think back to what we covered in the previous module, you'll know that having scatter plots to represent the data for a univariate variable makes no sense. It doesn't give us all the information we need. Ideally, we'd like the plots on the main diagonal to be histograms or KDE plots. In this example for our PairGrid, we'll specify that the off-diagonal plots should be scatter plots because they are bivariate, and the plots on the diagonal should be KDE plots. When you're working with Facebook data and using it to plot PairGrids and FacetGrids, you'll often find this kind of warning on your screen. These warnings are safe to ignore. That's because our dataset contains some missing values, and some values may not be formed perfectly, which is why Matplotlib (under the hood, that's what Seaborn uses) is not able to automatically detect handles and legends for this data. If you look at the resultant 3 X 3 matrix of our PairGrid, you'll see that this is indeed the case. All the plots on the main diagonal are KDE plots. You can view the kernel density estimation of comments, likes, and shares of our univariate plots. All the plots that are not on the main diagonal, just two are highlighted here out of six, are scatter plots. They represent bivariate relationships, and a scatter plot is a good way to model this. A PairGrid can be used to model three relationships, not just two. Every relationship is a pairwise relationship, in addition you can use the hue parameter for the third variable. Here the hue parameter will represent the type of Facebook post that was commented on, liked, or shared. Remember the four types--

photos, status, links, and videos. The off-diagonal plots, which model bivariate relationships, are scatter plots. The main diagonal for univariate data are KDE plots. And we also have a legend for the hue. As you can see, once again there's a warning here about some kind of invalid value. These are safe to ignore as far as this demo is concerned. If you're working with real-world data, you might want to clean up your dataset and see whether something is really wrong. And here is our resultant PairGrid, a 3 X 3 matrix with the colors showing the different types of posts. The legend for our hue is off to the right where photos are in blue, status in orange, link in green, and videos in red. In this next example, you'll see how you can fine-tune the PairGrid even further. You can have all the plots on the main diagonal be KDE plots for the univariate data. All plots which are on the upper side of the main diagonal off to the right will be scatter plots. And the plots on the lower side of the main diagonal will be regression plots. And here is the resultant PairGrid matrix. Notice the scatter plots on the upper side of the main diagonal. And notice the regression plots on the lower side of the main diagonal. In our previous example, we were interested in pairwise relationships between comments, likes, and shares. So we had the same data on the X axis, as well as on the Y axis, and all pairwise combinations of these. You can also use the PairGrid to model pairwise relationships between different data on the X and the Y axes. Here we want the X axis to have comments, likes, and shares. But the Y variables will be the month in which a post was posted, the weekday, and the hour. We'll continue to use the hue to represent the type of post. All the relationships are bivariate. We want scatter plots with a legend. The resultant 3 X 3 matrix have the Post Hour, Post Weekday, and Post Month as Y variables. The legend depicts the different colors, which represent different types of posts. And along the X axis, we have the comments, likes, and shares on individual posts.

Exploring the Bike Rental Dataset

So far, we've used a number of different kinds of plots to visualize data in Seaborn. Now let's say you had a machine learning dataset, and you wanted to use visualizations to explore this data. How would you go about it? We'll take a simple example here of a bike rental dataset. The bike rental dataset that we're going to use here contains information about how many bikes were rented for a particular day. This dataset has information for about two years. Now the number of bikes that were entered depends on a variety of factors such as the weather for that particular day, whether it was a working day or a holiday, a weekday or weekend, and so on. All of these factors are present in this dataset. The very first column here, the instant, represents one record for one day. We also have information for the particular year, the season, the month, and so on. The very last column here on the extreme right represents a count of the number of bikes that

were rented for that day. This is a very typical machine learning dataset which you'd use in a regression problem. You'd use this dataset to predict how many bikes will be rented on a particular day given the other weather features and holiday and weekday features. We're not going to be doing any machine learning here. We're only going to explore and visualize this dataset. This dataset has 731 records across two years roughly, one record for every day. The describe function on a pandas DataFrame will give us a lot of statistical information on the individual columns, which represent features in our dataset. Here is the average number of bikes that are rented in a day. You can see that it's roughly 4, 600 bikes on average. Let's set up a histogram for the count of bikes rented per day. This will give us an idea of how bike rentals are distributed. We're not interested in the KDE curve, but we do want the rug plot, the sticks to indicate how many data points at every count level. And the resultant histogram shows us that the demand for bikes is mostly in the range of 4, 000 to 5, 000. That's where the peaks are. That's where the rug plots are the closest together. If you look at the extreme ends of this histogram on the left, you can see that for some days the demand for bikes is as low as 100, and the maximum demand, the picture on the very right of this histogram plot, is over 8, 000 bikes a day. Let's see what else is interesting about this dataset. We'll use a joint plot to model some bivariate relationships. We have two years' worth of data. How did the number of rentals vary by day across these two years? On the X axis, we plot the instant, which represents individual days. And on the Y axis, we plot the count of bikes rented. Let's take a look and analyze this resultant graph. On the X axis, we have days over a period of two years going up to 731. There are two peaks in this data. The first peak represents the peak for the first year, typically in the summer months it looks like. There is a second peak in this scatter plot as well. You can see that bike rentals for the second year have gone up. The peak once again is in the middle of the year probably around the summer months. Instead of plotting days on the X axis, let's get information at a granularity that makes much more sense. We'll use a joint plot once again, but this time on the X axis, we plot the months. Our X axis represents categorical values here, the months of the year. Remember, the dataset is across two years. You can see that there is a distinct peak in the middle of the year, months from 6 to 8, which are the summer months. The fall season also seems to be a good one for bike rentals. There is another peak around months 9 and 10. We don't know yet how we are going to set up our machine learning problem. We are still exploring relationships in this data. Let's use a pair plot to model pairwise relationships in this dataset. Here we are going to see how holidays, weekdays, and working days affect the count of bikes rented. We'll use a regression plot here. We'll also use the hue to depict season. The resultant plot can give us a lot of information. Let's see if we can zoom in on some of these. Let's look at the center plot, which plots the weekdays against the count of bike rentals. You can see that bike rentals do not vary

much across weekdays. Let's consider this piece of information in combination with the information that we get from the first plot. Here we are mapping a holiday against bike rentals, and we can see that there is high demand on working days, days which are not holidays. These two bits of information seem to suggest that these bikes are being rented for people to get to work, maybe people rent bikes to go to train stations or to the subway. These bikes are typically not being rented for leisure activities. We've seen earlier when we plotted month data against bike rentals that seasons affect how many bikes are rented. We'll use a joint plot, specifically plot season on the X axis to see how seasonal bike rentals are. Seasons are categorical variables as well, which explains how this plot is. And you can see that bike rentals tend to be the highest in summer. Summer in this plot is season 2. Bike rentals are also pretty high in fall, in season 3. We can also use the pair plot to explore how the temperature and other weather conditions affect bike rentals. On the X axis, we'll consider the temperature, the atemp, or what the temperature actually feels like, the humidity, and the wind speed. And on the Y axis, we'll consider the count of bikes rented. In the resultant pair plot, we can see that the temperature and the a-temperature have very similar scatter plots. Also when the temperature tends to be higher, bike rentals tend to go up. People don't seem to like to rent bikes in colder weather. Interestingly enough, if you look at the humidity plot, the higher the humidity, more number of bikes are rented. The last plot represents information on the wind speed. When wind speeds are very high, people don't like to ride bikes. There are fewer rentals. We've explored a bunch of different data here. This gives us enough information to proceed with our machine learning problem, whether it's regression, clustering, or some other kind of prediction. And on this note, we've come to the very end of this module on Building Trellis plots in Seaborn. We've used trellis plots to explore multidimensional data. We've studied Seaborn's FacetGrid in detail, which allows us to visualize relationships between multiple variables separately. In fact, these can be thought of as conditional relationships. Every cell in the FacetGrid represents a different set of conditions. Within our trellis plots, we've used the X and Y axes and also the hue to show relationships. Another useful trellis plot that we studied is the PairGrid, which is used to plot relationships pairwise between two pairs of variables. In the next module, we'll see how we can improve the aesthetics of the figures that we plotted by using different themes and color palettes in our figures.

Controlling Plot Aesthetics and Style in Seaborn

Module Overview

Hi, and welcome to this module on Controlling Plot Aesthetics and Style in Seaborn. You have the visualization you need. How do you make it look good? That's what this module is all about. We'll start off by exploring the themes that are present in Seaborn which govern plot aesthetics. There are a number of built-in themes that you can use right out of the box. Seaborn also gives you a wide variety of color palettes to choose from. You can specify qualitative, sequential, and diverging color palettes. Each has its own use cases. You can also specify your own color palette based on your branding or design sense. In this module, we'll also study how we can override very granular style details in our plot to get the perfect look and feel.

Themes and Figure Styles

We'll start off by exploring the themes and figure styles that are available in Seaborn. For whatever examples we need data, we'll use the same Facebook dataset that we are familiar with. We'll read it into the fb_data DataFrame. We've seen this data before. It contains a bunch of information on the social media posts made by our organization on Facebook. The different types and categories of posts, the month in which the posts were made, the user engagement with these posts, and so on. Let's say that you specify no styles within Seaborn, and you set up a histogram plot plotting a univariate distribution, the lifetime reach of all the posts that your organization made on Facebook. This is what your probability distribution looks like by default. Now you can use the sns. set method to restore the default visual style in Seaborn. It's possible that you have some other styles set because of some other plots that you set up earlier. The sns. set method will restore the default style. Let's view the same distribution once again this time with the default style. Notice the grid in the background and the default colors. Seaborn has five preset themes that you can use--whitegrid, darkgrid, dark, white, and ticks. Let's explore all of these starting with whitegrid. Notice that the background color of our visualization is white. There is a grid in the background which allows us to see the values on the X and the Y axes, and the default color of the histogram and the curve is some kind of blue. The darkgrid theme can be thought of as the opposite of whitegrid. Notice that the background is some kind of darkish grayish hue with a white grid. The other colors that are used to plot the histogram and the curve are the same. Another preset theme that Seaborn uses is the dark style. The dark style is very similar to the darkgrid except that there is no grid in the background. One thing you need to note with Seaborn is once you set up the style for one particular plot, all subsequent plots will follow the same style unless you explicitly change it. The preset theme white is very similar to whitegrid. The background is white, but there are no grid lines in the background. All of the values that are

specified on the X and Y axes so far did not have any ticks. When you use the preset theme ticks, you're axes will automatically have the ticks specified. Notice that the rest of the theme is white because that's the theme we set up last. All subsequent plots will follow the same theme unless explicitly changed. If you want to change the theme or the plot style only for the current plot and not affect all plots that are going to follow, you can use the with keyword in Python. Here we want to use the darkgrid theme for the axes for this particular plot, but we do not want to affect other plots that follow. The next plot example that we see here continues to be with the white preset theme. That was the theme that was last specified without the with keyword. You can use the despine method on Seaborn in order to remove the spines from your plot. Despine automatically removes the spines at the top and the right. If you want more control over how you want to despine your plot, you can pass in input arguments to the despine method. If you want a specific axis to be despined, specify True as the input argument. Here we've despined the left and the bottom axes, but we haven't despined the right and the top axes. The despine method in Seaborn lends itself to more finer-grained control. Offset is equal to 15 moves the X axis 15 pixels to the right. Trim is equal to True despines those 15 pixels.

Qualitative Color Palettes

In this demo, we'll see how we can use the various color palettes that Seaborn provides. The import statements here are the same as before. We use the matplotlib inline command to display our Matplotlib plots inline within the Jupyter notebook. Once again, we work with Facebook data. Really any dataset will do here. We aren't really doing any special analysis with the data. We are just using it to see how our graphs look. Instead of using the preset themes, we can explicitly specify color palettes that our Seaborn graphs are to use by calling the sns. set palette method. Greens_d uses a dark green palette. And our histogram and our curve are both plotted with dark green colors. We can use a preset theme along with a specific palette. Here is the whitegrid theme with the color palette which is set to Blues. Let's reset our Seaborn visualization library to the default theme and palette using the sns. set method. Seaborn has a special function called the palplot, which allows us to view the colors in any palette. Displayed onscreen is the current palette that Seaborn is using. Seaborn has color palettes for different kinds of data. What you see onscreen here is a qualitative color palette which is used typically with categorical data, data which has no inherent ordering. Notice that every color is very different from the other. We've seen an example of this qualitative color palette before when you set up multiple KDE plots on the same graph. It uses the colors from the qualitative color palette. The first thing you might notice is the set_context("poster"). What exactly the poster context is we'll cover later on in this

module. I'm going to skip over it for now. Here are the different KDE plots at the different bandwidths. And notice that each of them have been drawn with a different color from the qualitative color palette. If you want a color palette which has more than six colors, you can specify an HLS color palette with any number of colors that you're interested in. HLS stands for hue, lightness, and saturation. Hue refers to the color. Lightness refers to the intensity, how bright or dark the color is. Saturation refers to how much gray there is in the color. When we need more colors, Seaborn chooses from a circular color space where the hue changes based on an overlap of colors. We can customize the HLS palette that we use by specifying our own lightness value and saturation value. All the colors in this palette that're highlighted onscreen are of the same intensity. Our eyes just perceive them differently. Seaborn allows us to view what colors are available within a certain palette and what palettes are available within Seaborn using something called a Color Brewer palette. Here we can choose the Color Brewer palette for qualitative colors, and then you can change the name of the theme that you're interested in. Here we are viewing the palette for Set1. Let's choose another value here. Let's go ahead and view the palette for the accent. We can interactively see what colors will be added when we change the number of colors in this palette. You can also modify the saturation levels of your palette to get exactly the shades that you're interested in. This interactive widget is extremely useful in order to help you get set up with the right colors for you. I'm going to set up a pair plot here on the Facebook dataset which uses the Paired color palette. This is one of the palettes that Seaborn provides. Setting up this pair plot results in an error. This is because of invalid values in our dataset and can be ignored for this demo. And here are all the colors that you can see on this Paired palette. If you go back up to the Color Brewer palette and view the Paired palette, you'll see that the same colors are as shown here. Seaborn allows very fine-grained control over the colors that you show within your graph. You can explicitly specify every color that you want within your palette. This is extremely useful when you want to use the color palette that is specific to your organization. You don't always have to specify colors using RGB values. You can also use named colors which Seaborn identifies. Pale red, medium green, and denim blue, these are some of the named colors from the xkcd color survey. And here is our graph visualized with those colors.

Sequential and Cubehelix Palettes

So far we've studied qualitative color palettes, which are used for categorical data. Sequential color palettes are very useful for plots where data has a wide range. But the low values are uninteresting, and the high values are interesting. If you want to emphasize the high values, you tend to use a sequential color palette. Here is an example of one with blues. Here is another

example with the Blues color palette, but this time the colors are reversed. They go from the dark blue to the light blue. Similarly, Blues_d is the same Blues palette but with darker blues. Or if you want a light palette with a green shade, you could use the sns. light_palette function with green as an input argument. And here is an example of a KDE plot which uses the greens light palette. We specify a cmap using the sns. light_palette function and pass the cmap into our KDE plot. Similar to the light palette, we can also specify dark palettes. Here is a dark purple palette with the colors reversed. Just like we did with qualitative palettes earlier, we can use the Seaborn Color Brewer to view sequential palettes as well. You can specify the color that you're interested in. For example, here is a sequential palette in the purple color. You can use the n to vary the number of colors in your sequential palette, and you can use the desat to vary the saturation level of your palette. You can use the palette in reverse by choosing the reverse variant. Cubehelix palettes are a special variation of sequential palettes in Seaborn. The Matplotlib version of the cubehelix palette is on top, and the Seaborn version is at the bottom. Just like with sequential palettes, cubehelix palettes will have important values in darker colors. So what's the difference between a cubehelix palette and a sequential palette? The cubehelix sequential palettes have a linear increase or decrease of brightness, as well as hue variation. This basically means that when you use a cube helix palette, even if you print out the image in black and white, you'll still see the variations in data. A black and white representation of the graphs that we plot using a regular sequential palette may not be differentiable. However, the cubehelix palette will ensure that you'll still see variations.

Diverging Color Palettes

In addition to qualitative and sequential color palettes, Seaborn also offers diverging color palettes. Here the low intensities are at the center. Diverging color palettes are typically used where both the low values, as well as the high values are of interest, and we want to draw attention to both. What you see onscreen is an example of a diverging color palette from the Seaborn Color Brewer. There is a Matplotlib equivalent as well. The coolwarm palette is a diverging color palette from Matplotlib. We can once again use the Seaborn Color Brewer to view what diverging color palettes are available. Here onscreen is the red/blue diverging palette. We also have the brown/green diverging palette. You can change n to get more colors on this palette. You can also change the desat value as before. Let's choose the red/yellow/blue diverging color palette with nine colors, and assign this palette to the h variable. We can now set up a PairGrid, which uses the palette specified by h. And notice that all the colors are the red and yellow shades from our color palette. You can scroll back up in your Jupyter notebook and change the color

palette that you assigned to the h variable, and then execute the graph once again to view the same plot in a different color palette. You can also set up the Seaborn PairGrid to use the Matplotlib coolwarm color palette. And you can see that all the data is specified in the coolwarm shades of blue and red.

Figure Aesthetics

In this very last demo in this module and of this course, we'll see how we can override very granular styles in Seaborn. For this demo, the import statements are the same, and we use the same Facebook data as earlier. You can call the Seaborn function axes_style in order to see the current style that we your plot axes use. This contains very granular information on fonts, colors, ticks, and so on. Each of these individual styles can be overridden within a plot by specifying a dictionary of the styles that we want. Here is the default visualization in Seaborn where we haven't specified any explicit styles. And here we have a dictionary of very specific styles for the ticks in our plot. As you can see, the resultant plot looks very different indeed. Let's see the specific styles that we've overridden. We've specified that the ytick color should be green. We want our xticks to be in the blue color. That's what we've specified as well. We also want our yticks to be facing inward towards the graph. The ytick direction is in. In addition, our dictionary also specifies a color for the background, the size of the ticks, and so on. It's pretty clear from this granular representation that we have complete control over how our plot is displayed. Let's reset our Seaborn style to the default, and let's view the various preset contexts that are available in Seaborn. Depending on where you want to use your visualization, you can specify a context for your plot. For example, this plot that you see here onscreen is in the paper context. We're expecting to view this plot on paper, which is why you can see that the axes fonts are a little small. These are fonts that we can view on paper very easily. However, if you're planning to use this visualization as a part of a talk or a presentation, you'll use the talk context. Notice in the talk context, the font sizes are much larger. These are font sizes that can be viewed by our audience. Or if you're planning to print this graph out on some kind of poster, you'll use the poster context. The poster context also has larger fonts. And the last preset context available in Seaborn is the notebook context. Font sizes are smaller here.

Summary and Further Study

And on this note, we come to the very end of this module and to the end of this course on Seaborn. We've seen the Seaborn visualization library allows us to specify themes to govern plot

aesthetics. Seaborn allows us to use a number of preset themes that are available out of the box. In this module, we also looked at how we can use the different kinds of color palettes available--the qualitative, sequential, and divergent color palettes. We also saw sequential color palette variations such as the cubehelix palette. Seaborn along with Matplotlib gives us very fine-grained control over our plot elements. We can override very specific styles within our plot. And with this, we come to the end of this course. If you're interested in data visualization, here are some Pluralsight courses that you can watch for further study. Data Visualization for Developers is an extremely popular course. If you're interested in data visualization and exploration as a precursor to machine learning, How to Think About Machine Learning Algorithms and Understanding Machine Learning with Python are also good courses that you can use to get started in this field. That's all from me today. Thank you for listening.

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

Level	Beginner
Rating	★★★★½ (14)
My rating	★★★★★
Duration	1h 43m
Released	7 Jun 2018

Share course