Springboard Regression Case Study, Unit 8 - The Red Wine Dataset - Tier 1

Welcome to the regression case study! Please note: this is *Tier 1* of the case study.

This case study was designed for you to use Python to apply the knowledge you've acquired in reading *The Art of Statistics* (hereinafter *AoS*) by Professor Spiegelhalter. Specifically, the case study will get you doing regression analysis; a method discussed in Chapter 5 on p.121. It might be useful to have the book open at that page when doing the case study to remind you of what it is we're up to (but bear in mind that other statistical concepts, such as training and testing, will be applied, so you might have to glance at other chapters too).

The aim is to *use exploratory data analysis (EDA) and regression to predict alcohol levels in wine with a model that's as accurate as possible*.

We'll try a *univariate* analysis (one involving a single explanatory variable) as well as a *multivariate* one (involving multiple explanatory variables), and we'll iterate together towards a decent model by the end of the notebook. The main thing is for you to see how regression analysis looks in Python and jupyter, and to get some practice implementing this analysis.

Throughout this case study, questions will be asked in the markdown cells. Try to answer these yourself in a simple text file when they come up. Most of the time, the answers will become clear as you progress through the notebook. Some of the answers may require a little research with Google and other basic resources available to every data scientist.

For this notebook, we're going to use the red wine dataset, wineQualityReds.csv. Make sure it's downloaded and sitting in your working directory. This is a very common dataset for practicing regression analysis and is actually freely available on Kaggle, here.

You're pretty familiar with the data science pipeline at this point. This project will have the following structure: 1. Sourcing and loading

- **Import relevant libraries**
- **Load the data**
- **Exploring the data**
- **Choosing a dependent variable**

2. Cleaning, transforming, and visualizing

- **Visualizing correlations**

3. Modeling

- **Train/Test split**
- **Making a Linear regression model: your first model**
- **Making a Linear regression model: your second model: Ordinary Least Squares (OLS)**
- **Making a Linear regression model: your third model: multiple linear regression**
- **Making a Linear regression model: your fourth model: avoiding redundancy**

4. Evaluating and concluding

- **Reflection**
- **Which model was best?**
- **Other regression algorithms**

1. Sourcing and loading

1a. Import relevant libraries

In [1]:

```
# Import relevant libraries and packages.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns # For all our visualization needs.
import statsmodels.api as sm # What does this do? Find out and type here.
from statsmodels.graphics.api import abline_plot # For visualling evaluating predictions.
from sklearn.metrics import mean_squared_error, r2_score # What does this do? Find out an
d type here.
from sklearn.model_selection import train_test_split # For splitting the data.
from sklearn import linear_model, preprocessing # What does this do? Find out and type he
re.
import warnings # For handling error messages.
# Don't worry about the following two instructions: they just suppress warnings that coul
d occur later.
warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.filterwarnings(action="ignore", module="scipy", message="^internal gelsd")
```

1b. Load the data

In [2]:

```
# Load the data. We'll set the parameter index_col to 0, because the first column contain
s no useful data.
wine = pd.read_csv("wineQualityReds.csv", index_col=0)
```

1c. Exploring the data

In [3]:

```
# The first thing we do after importing data - call a .head() on it to check out its appe
arance.
wine.head()
```

Out[3]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dioxide	density	pH	sulpha
1	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
2	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
3	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
4	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
5	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

In [4]:

```
# Another very useful method to call on a recently imported dataset is .info(). Call it h
ere to get a good
# overview of the data:
# Examine the data types of our dataset
wine.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1599 entries, 1 to 1599
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed.acidity         1599 non-null   float64
 1   volatile.acidity      1599 non-null   float64
 2   citric.acid           1599 non-null   float64
 3   residual.sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free.sulfur.dioxide   1599 non-null   float64
 6   total.sulfur.dioxide  1599 non-null   float64
```

```
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 162.4 KB
```

What can you infer about the nature of these variables, as output by the info() method?

Which variables might be suitable for regression analysis, and why? For those variables that aren't suitable for regression analysis, is there another type of statistical modeling for which they are suitable?

In [5]:

```
# We should also look more closely at the dimensions of the dataset with .shape().
# Remember: parameters to print() are separated by commas.
print("There are:", wine.shape[0], 'rows.')
print("There are:", wine.shape[1], 'columns.')
```

```
There are: 1599 rows.
There are: 12 columns.
```

1d. Choosing a dependent variable

We now need to pick a dependent variable for our regression analysis: a variable whose values we will predict.

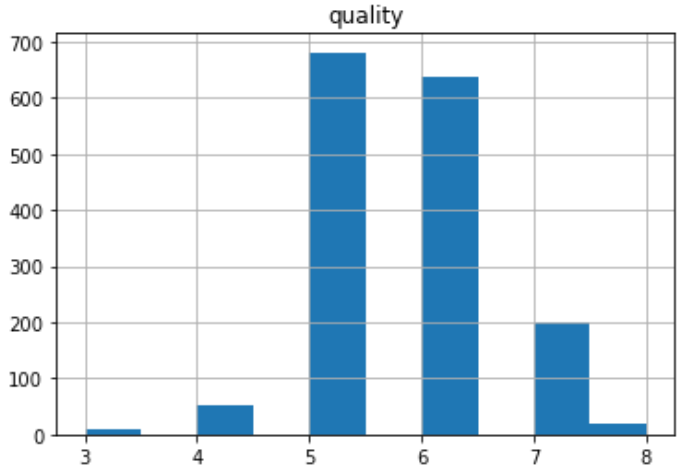
'Quality' seems to be as good a candidate as any. Let's check it out. One of the quickest and most informative ways to understand a variable is to make a histogram of it. This gives us an idea of both the center and spread of its values.

In [6]:

```
# Making a histogram of the quality variable.
wine.hist(column="quality")
```

Out[6]:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000029AA7398580>]],
      dtype=object)
```



We can see so much about the quality variable just from this simple visualization. Answer yourself: what value do most wines have for quality? What is the minimum quality value below, and the maximum quality value? What is the range? Remind yourself of these summary statistical concepts by looking at p.49 of the *AoS*.

But can you think of a problem with making this variable the dependent variable of regression analysis? Remember the example in *AoS* on p.122 of predicting the heights of children from the heights of parents? Take a moment here to think about potential problems before reading on.

The issue is this: quality is a *discrete* variable, in that its values are integers (whole numbers) rather than floating point numbers. Thus, quality is not a *continuous* variable. But this means that it's actually not the best target for

regression analysis.

Before we dismiss the quality variable, however, let's verify that it is indeed a discrete variable with some further exploration.

In [7]:

```
# A great way to get a basic statistical summary of a variable is to call the describe()
method on the relevant field.
wine["quality"].describe()

# What do you notice from this summary?
```

Out[7]:

```
count    1599.000000
mean        5.636023
std         0.807569
min         3.000000
25%         5.000000
50%         6.000000
75%         6.000000
max         8.000000
Name: quality, dtype: float64
```

In [8]:

```
# Calling .value_counts() on the quality field with the parameter dropna=False,
# get a list of the values of the quality variable, and the number of occurrences of each
.
# Do you know why we're calling value_counts() with the parameter dropna=False? Take a mo
ment to research the
# answer if you're not sure.
wine["quality"].value_counts(dropna=False)
```

Out[8]:

```
5    681
6    638
7    199
4     53
8     18
3     10
Name: quality, dtype: int64
```

The outputs of the describe() and value_counts() methods are consistent with our histogram, and since there are just as many values as there are rows in the dataset, we can infer that there are no NAs for the quality variable.

But scroll up again to when we called info() on our wine dataset. We could have seen there, already, that the quality variable had int64 as its type. As a result, we had sufficient information, already, to know that the quality variable was not appropriate for regression analysis. Did you figure this out yourself? If so, kudos to you!

The quality variable would, however, conduce to proper classification analysis. This is because, while the values for the quality variable are numeric, those numeric discrete values represent *categories*; and the prediction of category-placement is most often best done by classification algorithms. You saw the decision tree output by running a classification algorithm on the Titanic dataset on p.168 of Chapter 6 of *AoS*. For now, we'll continue with our regression analysis, and continue our search for a suitable dependent variable.

Now, since the rest of the variables of our wine dataset are continuous, we could — in theory — pick any of them. But that does not mean that they are all equally sutiable choices. What counts as a suitable dependent variable for regression analysis is determined not just by *intrinsic* features of the dataset (such as data types, number of NAs etc) but by *extrinsic* features, such as, simply, which variables are the most interesting or useful to predict, given our aims and values in the context we're in. Almost always, we can only determine which variables are sensible choices for dependent variables with some domain knowledge.

Not all of you might be wine buffs, but one very important and interesting quality in wine is acidity. As the Waterhouse Lab at the University of California explains, 'acids impart the sourness or tartness that is a fundamental feature in wine taste. Wines lacking in acid are "flat." Chemically the acids influence titrable acidity which affects taste and pH which affects color, stability to oxidation, and consequantly the overall lifespan of a

wine.'

If we cannot predict quality, then it seems like **fixed acidity** might be a great option for a dependent variable. Let's go for that.

So if we're going for fixed acidity as our dependent variable, what we now want to get is an idea of *which variables are related interestingly to that dependent variable*.

We can call the .corr() method on our wine data to look at all the correlations between our variables. As the documentation shows, the default correlation coefficient is the Pearson correlation coefficient (p.58 and p.396 of the *AoS*); but other coefficients can be plugged in as parameters. Remember, the Pearson correlation coefficient shows us how close to a straight line the data-points fall, and is a number between -1 and 1.

In [10]:

```
# Call the .corr() method on the wine dataset
wine.corr()
```

Out[10]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.dioxide	de
fixed.acidity	1.000000	-0.256131	0.671703	0.114777	0.093705	-0.153794	-0.113181	0.66
volatile.acidity	-0.256131	1.000000	-0.552496	0.001918	0.061298	-0.010504	0.076470	0.02
citric.acid	0.671703	-0.552496	1.000000	0.143577	0.203823	-0.060978	0.035533	0.36
residual.sugar	0.114777	0.001918	0.143577	1.000000	0.055610	0.187049	0.203028	0.35
chlorides	0.093705	0.061298	0.203823	0.055610	1.000000	0.005562	0.047400	0.20
free.sulfur.dioxide	-0.153794	-0.010504	-0.060978	0.187049	0.005562	1.000000	0.667666	0.02
total.sulfur.dioxide	-0.113181	0.076470	0.035533	0.203028	0.047400	0.667666	1.000000	0.07
density	0.668047	0.022026	0.364947	0.355283	0.200632	-0.021946	0.071269	1.00
pH	-0.682978	0.234937	-0.541904	-0.085652	-0.265026	0.070377	-0.066495	0.34
sulphates	0.183006	-0.260987	0.312770	0.005527	0.371260	0.051658	0.042947	0.14
alcohol	-0.061668	-0.202288	0.109903	0.042075	-0.221141	-0.069408	-0.205654	0.49
quality	0.124052	-0.390558	0.226373	0.013732	-0.128907	-0.050656	-0.185100	0.17

Ok - you might be thinking, but wouldn't it be nice if we visualized these relationships? It's hard to get a picture of the correlations between the variables without anything visual.

Very true, and this brings us to the next section.

2. Cleaning, Transforming, and Visualizing

2a. Visualizing correlations

The heading of this stage of the data science pipeline ('Cleaning, Transforming, and Visualizing') doesn't imply that we have to do all of those operations in *that order*. Sometimes (and this is a case in point) our data is already relatively clean, and the priority is to do some visualization. Normally, however, our data is less sterile,

and we have to do some cleaning and transforming first prior to visualizing.

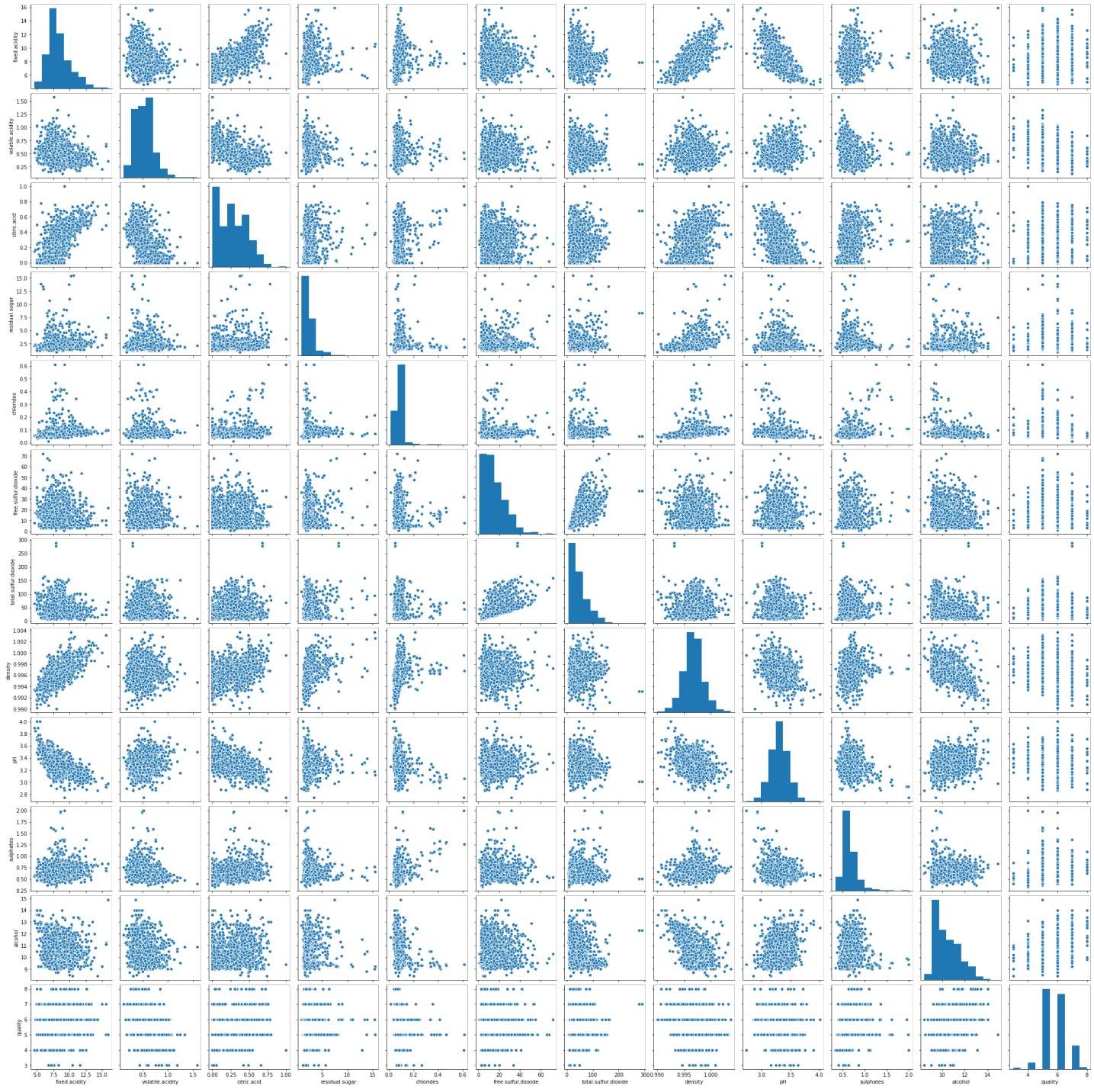
Now that we've chosen alcohol level as our dependent variable for regression analysis, we can begin by plotting the pairwise relationships in the dataset, to check out how our variables relate to one another.

In [11]:

```
# Call the .pairplot() method on our Seaborn object 'sns', and plug in our wine data as a parameter.
# Nb: this instruction will take a long time to execute. It's doing a lot of operations!
sns.pairplot(wine)
```

Out[11]:

```
<seaborn.axisgrid.PairGrid at 0x29aa7abdf70>
```



If you've never executed your own Seaborn pairplot before, just take a moment to look at the output. They certainly output a lot of information at once. What can you infer from it? What can you *not* justifiably infer from it?

... All done?

Here's a couple things you might have noticed:

- a given cell value represents the correlation that exists between two variables
- on the diagonal, you can see a bunch of histograms. This is because pairplotting the variables with themselves would be pointless, so the pairplot() method instead makes histograms to show the distributions of those variables' values. This allows us to quickly see the shape of each variable's values.
- the plots for the quality variable form horizontal bands, due to the fact that it's a discrete variable. We were certainly right in not pursuing a regression analysis of this variable.
- Notice that some of the nice plots invite a line of best fit, such as alcohol vs density. Others, such as citric acid vs alcohol, are more inscrutable.

So we now have called the .corr() method, and the .pairplot() Seaborn method, on our wine data. Both have flaws. Happily, we can get the best of both worlds with a heatmap.

In [13]:

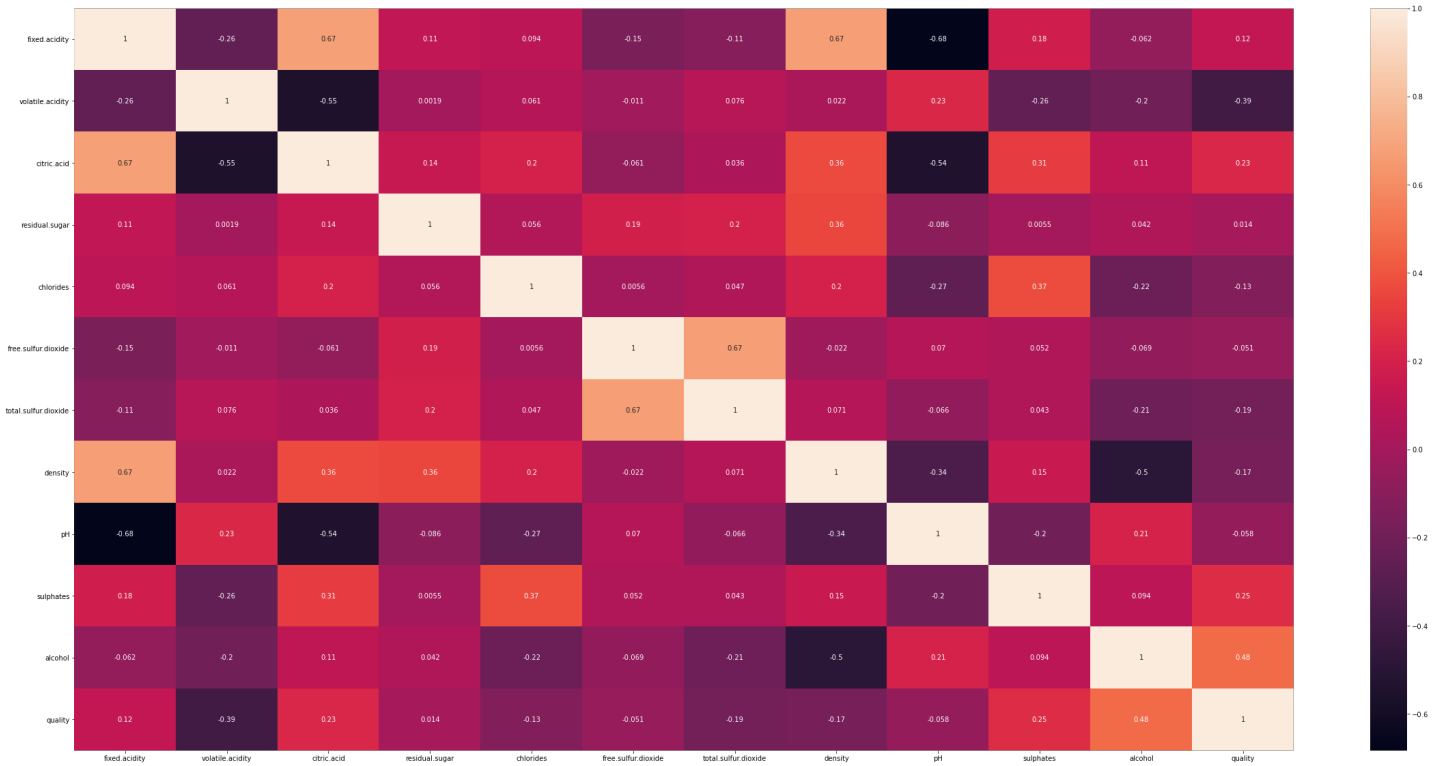
```
# We need to do some preliminary work, and ensure that the Matplotlib plot is big enough.

# Call .figure() on plt, and plug in the parameter figsize=(40,20) (or similar suitably l
arge dimensions)
plt.figure(figsize=(40,20))

# To create an annotated heatmap of the correlations, we call the heatmap() method on our
sns object.
# Ensure to plug in, as first parameter, wine.corr(), and as second parameter, annot=True
(so the graph is annotated)
sns.heatmap(wine.corr(), annot=True)
```

Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x29aad8842e0>
```



Take a moment to think about the following questions:

- How does color relate to extent of correlation?
- How might we use the plot to show us interesting relationships worth investigating?
- More precisely, what does the heatmap show us about the fixed acidity variable's relationship to the density variable?

There is a relatively strong correlation between the density and fixed acidity variables respectively. In the next code block, call the scatterplot() method on our sns object. Make the x-axis parameter 'density', the y-axis parameter 'fixed.acidity', and the third parameter specify our wine dataset.

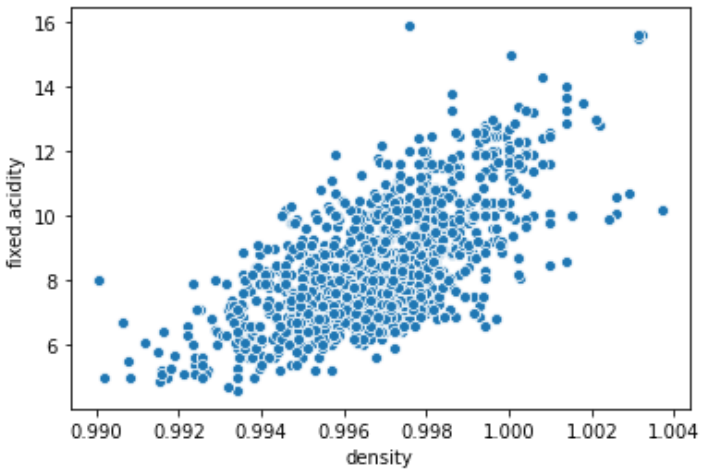
In [15]:

In [15]:

```
# Plot density against alcohol
sns.scatterplot(x="density", y="fixed.acidity", data=wine)
```

Out[15]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x29aa7ac3d90>
```



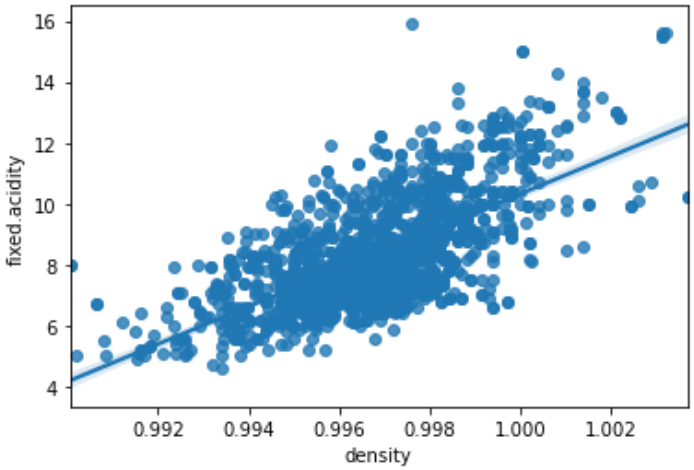
We can see a positive correlation, and quite a steep one. There are some outliers, but as a whole, there is a steep looking line that looks like it ought to be drawn.

In [16]:

```
# Call the regplot() method on your sns object, with parameters: x = 'density', y = 'fixe
d.acidity',
# and data=wine, to make this correlation more clear
sns.regplot(x="density", y="fixed.acidity", data=wine)
```

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x29aad33b0d0>
```



The line of best fit matches the overall shape of the data, but it's clear that there are some points that deviate from the line, rather than all clustering close.

Let's see if we can predict fixed acidity based on density using linear regression.

3. Modeling

3a. Train/Test Split

While this dataset is super clean, and hence doesn't require much for analysis, we still need to split our dataset into a test set and a training set.

You'll recall from p.158 of *AoS* that such a split is important good practice when evaluating statistical models. On p.158, Professor Spiegelhalter was evaluating a classification tree, but the same applies when we're doing regression. Normally, we train with 75% of the data and test on the remaining 25%.

To be sure, for our first model, we're only going to focus on two variables: fixed acidity as our dependent variable, and density as our sole independent predictor variable.

We'll be using sklearn here. Don't worry if not all of the syntax makes sense; just follow the rationale for what we're doing.

In [18]:

```
# Subsetting our data into our dependent and independent variables.
# Create a variable called 'X' and assign it the density field of wine.
# Create a variable called 'y' (that's right, lower case) and assign it the fixed.acidity
field of wine.
# Using double brackets allows us to use the column headings.
X = wine[["density"]]
y = wine[["fixed.acidity"]]

# Split the data. This line uses the sklearn function train_test_split().
# The test_size parameter means we can train with 75% of the data, and test on 25%.
# The random_state parameter allows our work to be checked and replicated by other data s
cientists
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state
= 123)
```

In [19]:

```
# We now want to check the shape of the X train, y_train, X_test and y_test to make sure
the proportions are right.
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(1199, 1) (1199, 1)
(400, 1) (400, 1)
```

3b. Making a Linear Regression model: our first model

Sklearn has a LinearRegression() function built into the linear_model module. We'll be using that to make our regression model.

In [20]:

```
# Create the model: make a variable called rModel, and assign it linear_model.LinearRegre
ssion(normalize=True).
# Note: the normalize=True parameter enables the handling of different scales of our vari
ables.
rModel = linear_model.LinearRegression(normalize=True)
```

In [21]:

```
# We now want to train the model on our test data.
# Call the .fit() method of rModel, and plug in X-train, y_train as parameters, in that o
rder.
rModel.fit(X_train, y_train)
```

Out[21]:

```
LinearRegression(normalize=True)
```

In [22]:

```
# Evaluate the model by printing the result of calling .score() on rModel, with parameter
s X_train, y_train.
print(rModel.score(X_train, y_train))
```

```
0.45487824100681673
```

The above score is called R-Squared coefficient, or the "coefficient of determination". It's basically a measure of

how successfully our model predicts the variations in the data away from the mean: 1 would mean a perfect model that explains 100% of the variation. At the moment, our model explains only about 23% of the variation from the mean. There's more work to do!

In [23]:

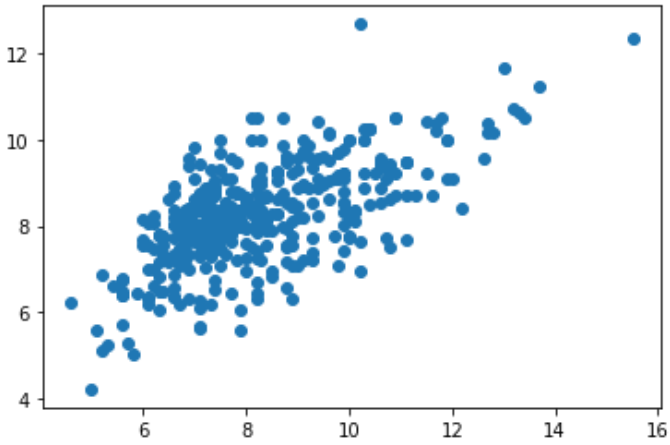
```
# Use the model to make predictions about our test data
# Make a variable called y_pred, and assign it the result of calling the predict() method
on rModel. Plug X_test into that method.
y_pred = rModel.predict(X_test)
```

In [24]:

```
# Let's plot the predictions against the actual result. Use scatter()
plt.scatter(y_test,y_pred)
```

Out[24]:

```
<matplotlib.collections.PathCollection at 0x29aad7a9190>
```



The above scatterplot represents how well the predictions match the actual results.

Along the x-axis, we have the actual fixed acidity, and along the y-axis we have the predicted value for the fixed acidity.

There is a visible positive correlation, as the model has not been totally unsuccesful, but it's clear that it is not maximally accurate: wines with an actual fixed acidity of just over 10 have been predicted as having acidity levels from about 6.3 to 13.

Let's build a similar model using a different package, to see if we get a better result that way.

3c. Making a Linear Regression model: our second model: Ordinary Least Squares (OLS)

In [39]:

```
# Create the test and train sets. Here, we do things slightly differently.
# We make the explanatory variable X as before.
X = wine[["density"]]

# But here, reassign X the value of adding a constant to it. This is required for Ordinar
y Least Squares Regression.
# Further explanation of this can be found here:
# https://www.statsmodels.org/devel/generated/statsmodels.regression.linear_model.OLS.htm
l
X = sm.add_constant(X)
```

In [40]:

```
# The rest of the preparation is as before.
y =  wine[["fixed.acidity"]]

# Split the data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state
= 123)
```

In [41]:

```
# Create the model
rModel2 = sm.OLS(y_train, X_train)
# Fit the model with fit()
rModel2_results = rModel2.fit()
```

In [42]:

```
# Evaluate the model with .summary()
rModel2_results.summary()
```

Out[42]:

OLS Regression Results

Dep. Variable:	fixed.acidity	**R-squared:**	0.455
Model:	OLS	**Adj. R-squared:**	0.454
Method:	Least Squares	**F-statistic:**	998.8
Date:	Thu, 10 Sep 2020	**Prob (F-statistic):**	6.68e-160
Time:	11:34:51	**Log-Likelihood:**	-2011.0
No. Observations:	1199	**AIC:**	4026.
Df Residuals:	1197	**BIC:**	4036.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-615.7316	19.746	-31.182	0.000	-654.473	-576.990
density	626.0927	19.810	31.604	0.000	587.226	664.959

Omnibus:	94.056	**Durbin-Watson:**	1.985
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	122.229
Skew:	0.668	**Prob(JB):**	2.87e-27
Kurtosis:	3.812	**Cond. No.**	1.06e+03

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.06e+03. This might indicate that there are strong multicollinearity or other numerical problems.

One of the great things about Statsmodels (sm) is that you get so much information from the summary() method.

There are lots of values here, whose meanings you can explore at your leisure, but here's one of the most important: the R-squared score is 0.455, the same as what it was with the previous model. This makes perfect sense, right? It's the same value as the score from sklearn, because they've both used the same algorithm on the same data.

Here's a useful link you can check out if you have the time: https://www.theanalysisfactor.com/assessing-the-fit-of-regression-models/

In [43]:

```
# Let's use our new model to make predictions of the dependent variable y. Use predict(),
and plug in X_test as the parameter
y_pred = rModel2_results.predict(X_test)
```

In [45]:

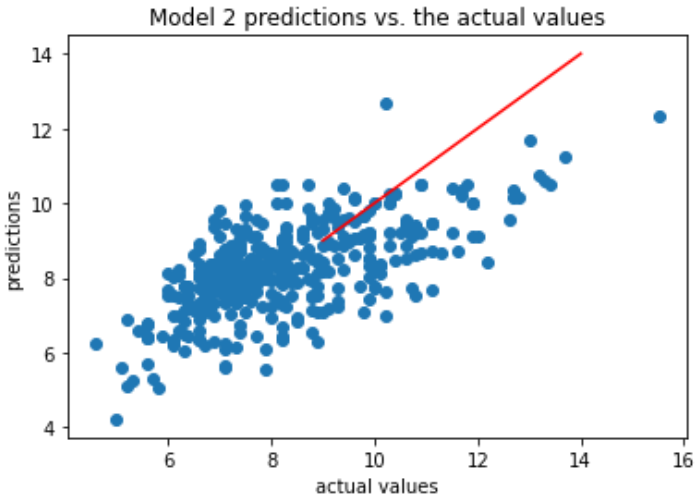
```
# Plot the predictions
# Build a scatterplot
plt.scatter(y_test, y_pred)

# Add a line for perfect correlation. Can you see what this line is doing?
plt.plot([x for x in range(9,15)],[x for x in range(9,15)], color='red')

# Label it nicely
plt.title("Model 2 predictions vs. the actual values")
plt.xlabel("actual values")
plt.ylabel("predictions")
```

Out[45]:

```
Text(0, 0.5, 'predictions')
```



The red line shows a theoretically perfect correlation between our actual and predicted values - the line that would exist if every prediction was completely correct. It's clear that while our points have a generally similar direction, they don't match the red line at all; we still have more work to do.

To get a better predictive model, we should use more than one variable.

3d. Making a Linear Regression model: our third model: multiple linear regression

Remember, as Professor Spiegelhalter explains on p.132 of *AoS*, including more than one explanatory variable into a linear regression analysis is known as *multiple linear regression*.

In [46]:

```
wine.columns
```

Out[46]:

```
Index(['fixed.acidity', 'volatile.acidity', 'citric.acid', 'residual.sugar',
       'chlorides', 'free.sulfur.dioxide', 'total.sulfur.dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [50]:

```
# Create test and train datasets
# This is again very similar, but now we include more columns in the predictors
# Include all columns from data in the explanatory variables X except fixed.acidity and q
uality (which was an integer)
X = wine.drop(["fixed.acidity", "quality"],axis=1)
# Create constants for X, so the model knows its bounds
X = sm.add_constant(X)
y = wine[["fixed.acidity"]]

# Split the data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state
= 123)
```

In [51]:

```
# We can use almost identical code to create the third model, because it is the same algo
rithm, just different inputs
# Create the model
rModel3 = sm.OLS(y_train, X_train)
# Fit the model
rModel3_results = rModel3.fit()
```

In [52]:

```
# Evaluate the model
rModel3_results.summary()
```

Out[52]:

OLS Regression Results

Dep. Variable:	fixed.acidity	**R-squared:**	0.871
Model:	OLS	**Adj. R-squared:**	0.870
Method:	Least Squares	**F-statistic:**	804.4
Date:	Thu, 10 Sep 2020	**Prob (F-statistic):**	0.00
Time:	11:44:20	**Log-Likelihood:**	-1145.6
No. Observations:	1199	**AIC:**	2313.
Df Residuals:	1188	**BIC:**	2369.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-648.2418	15.246	-42.518	0.000	-678.154	-618.329
volatile.acidity	0.1313	0.135	0.971	0.332	-0.134	0.397
citric.acid	1.8651	0.155	11.995	0.000	1.560	2.170
residual.sugar	-0.2485	0.015	-16.589	0.000	-0.278	-0.219
chlorides	-3.6575	0.443	-8.263	0.000	-4.526	-2.789
free.sulfur.dioxide	0.0068	0.002	2.859	0.004	0.002	0.012
total.sulfur.dioxide	-0.0064	0.001	-7.966	0.000	-0.008	-0.005
density	671.0968	15.184	44.198	0.000	641.306	700.887
pH	-5.1954	0.149	-34.792	0.000	-5.488	-4.902
sulphates	-0.8038	0.130	-6.193	0.000	-1.058	-0.549
alcohol	0.5713	0.025	22.753	0.000	0.522	0.621

Omnibus:	155.238	**Durbin-Watson:**	2.052
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	562.315
Skew:	0.595	**Prob(JB):**	7.85e-123
Kurtosis:	6.137	**Cond. No.**	7.23e+04

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 7.23e+04. This might indicate that there are strong multicollinearity or other numerical problems.

The R-Squared score shows a big improvement - our first model predicted only around 45% of the variation, but

now we are predicting 87%!

In [53]:

```
# Use our new model to make predictions
y_pred = rModel3_results.predict(X_test)
```

In [54]:

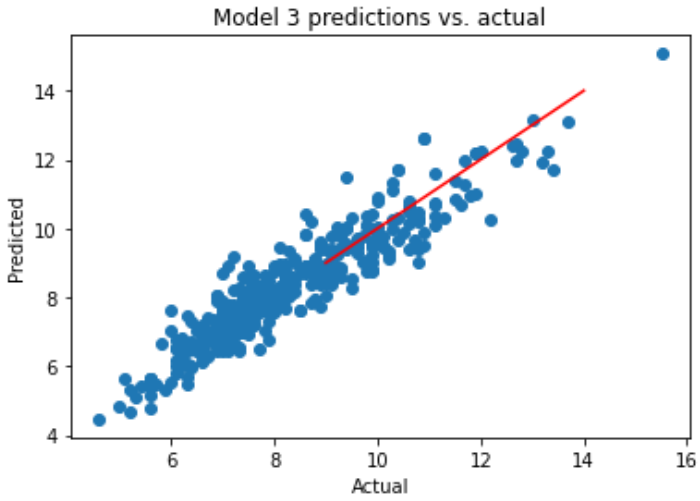
```
# Plot the predictions
# Build a scatterplot
plt.scatter(y_test, y_pred)

# Add a line for perfect correlation
plt.plot([x for x in range(9,15)],[x for x in range(9,15)], color='red')

# Label it nicely
plt.title("Model 3 predictions vs. actual")
plt.xlabel("Actual")
plt.ylabel("Predicted")
```

Out[54]:

```
Text(0, 0.5, 'Predicted')
```



We've now got a much closer match between our data and our predictions, and we can see that the shape of the data points is much more similar to the red line.

We can check another metric as well - the RMSE (Root Mean Squared Error). The MSE is defined by Professor Spiegelhalter on p.393 of *AoS*, and the RMSE is just the square root of that value. This is a measure of the accuracy of a regression model. Very simply put, it's formed by finding the average difference between predictions and actual values. Check out p. 163 of *AoS* for a reminder of how this works.

In [55]:

```
# Define a function to check the RMSE. Remember the def keyword needed to make functions?

def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
```

In [56]:

```
# Get predictions from rModel3
y_pred = rModel3_results.predict(X_test)

# Put the predictions & actual values into a dataframe
matches = pd.DataFrame(y_test)
matches.rename(columns = {'fixed.acidity':'actual'}, inplace=True)
matches["predicted"] = y_pred

rmse(matches["actual"], matches["predicted"])
```

Out[56]:

```
0.6163194678948751
```

The RMSE tells us how far, on average, our predictions were mistaken. An RMSE of 0 would mean we were making perfect predictions. 0.6 signifies that we are, on average, about 0.6 of a unit of fixed acidity away from the correct answer. That's not bad at all.

3e. Making a Linear Regression model: our fourth model: avoiding redundancy

We can also see from our early heat map that volatile.acidity and citric.acid are both correlated with pH. We can make a model that ignores those two variables and just uses pH, in an attempt to remove redundancy from our model.

In [58]:

```
# Create test and train datasets
# Include the remaining six columns as predictors
X = wine[["residual.sugar","chlorides","total.sulfur.dioxide","density","pH","sulphates"
]]

# Create constants for X, so the model knows its bounds
X = sm.add_constant(X)

y = wine[["fixed.acidity"]]

# Split the data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state
= 123)
```

In [59]:

```
# Create the fifth model
rModel4 = sm.OLS(y_train, X_train)
# Fit the model
rModel4_results = rModel4.fit()
# Evaluate the model
rModel4_results.summary()
```

Out[59]:

OLS Regression Results

Dep. Variable:	fixed.acidity	**R-squared:**	0.742
Model:	OLS	**Adj. R-squared:**	0.741
Method:	Least Squares	**F-statistic:**	571.8
Date:	Thu, 10 Sep 2020	**Prob (F-statistic):**	0.00
Time:	11:51:38	**Log-Likelihood:**	-1562.3
No. Observations:	1199	**AIC:**	3139.
Df Residuals:	1192	**BIC:**	3174.
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-485.6576	16.010	-30.335	0.000	-517.068	-454.247
residual.sugar	-0.1078	0.020	-5.481	0.000	-0.146	-0.069
chlorides	-6.3544	0.578	-10.990	0.000	-7.489	-5.220
total.sulfur.dioxide	-0.0094	0.001	-11.799	0.000	-0.011	-0.008
density	516.4441	15.894	32.492	0.000	485.260	547.628

pH	-6.0430	0.184	-32.766	0.000	-6.405	-5.681
sulphates	0.7540	0.165	4.559	0.000	0.430	1.078

Omnibus:	105.987	**Durbin-Watson:**	2.002
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	206.837
Skew:	0.572	**Prob(JB):**	1.22e-45
Kurtosis:	4.683	**Cond. No.**	5.08e+04

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.08e+04. This might indicate that there are strong multicollinearity or other numerical problems.

The R-squared score has reduced, showing us that actually, the removed columns were important.

Conclusions & next steps

Congratulations on getting through this implementation of regression and good data science practice in Python!

Take a moment to reflect on which model was the best, before reading on.

. . .

Here's one conclusion that seems right. While our most predictively powerful model was rModel3, this model had explanatory variables that were correlated with one another, which made some redundancy. Our most elegant and economical model was rModel4 - it used just a few predictors to get a good result.

All of our models in this notebook have used the OLS algorithm - Ordinary Least Squares. There are many other regression algorithms, and if you have time, it would be good to investigate them. You can find some examples here. Be sure to make a note of what you find, and chat through it with your mentor at your next call.