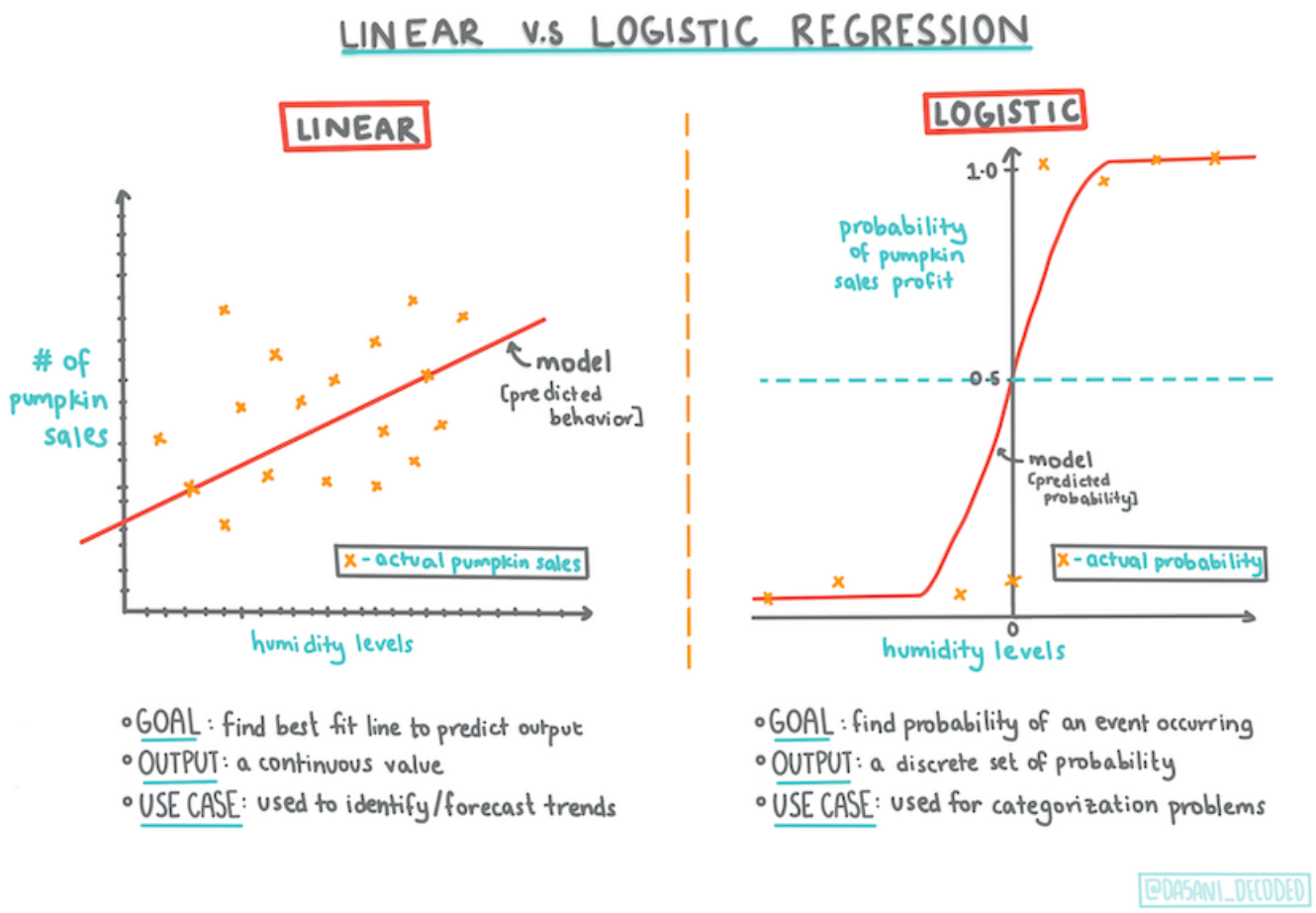
Logistic regression to predict categories



> Infographic by Dasani Madipalli

Prerequisite

Having worked with the pumpkin data, we are now familiar enough with it to realize that there's one binary category that we can work with: `Color`.

Let's build a logistic regression model to predict that, given some variables, *what color a given pumpkin is likely to be* (orange 🎃 or white 👻).

> Why are we talking about binary classification in a lesson grouping about regression? Only for linguistic convenience, as logistic regression is really a classification method, albeit a linear-based one. Learn about other ways to classify data in the next lesson group.

Define the question

For our purposes, we will express this as a binary: 'Orange' or 'Not Orange'. There is also a 'striped' category in our dataset but there are few instances of it, so we will not use it. It disappears once we remove null values from the dataset, anyway.

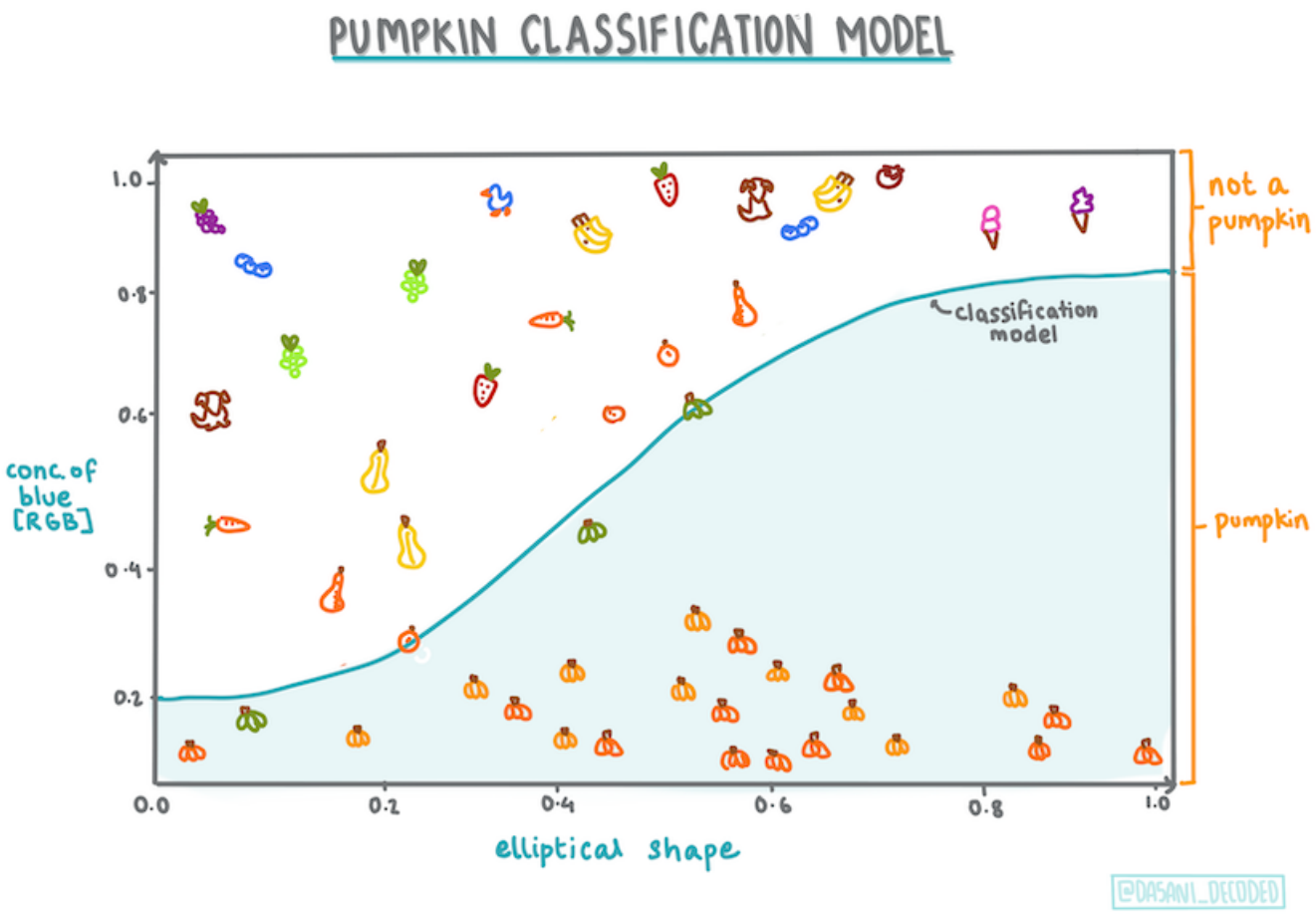
> 🎃 Fun fact, we sometimes call white pumpkins 'ghost' pumpkins. They aren't very easy to carve, so they aren't as popular as the orange ones but they are cool looking!

About logistic regression

Logistic regression differs from linear regression, which you learned about previously, in a few important ways.

Binary classification

Logistic regression does not offer the same features as linear regression. The former offers a prediction about a binary category ("orange or not orange") whereas the latter is capable of predicting continual values, for example given the origin of a pumpkin and the time of harvest, *how much its price will rise*.

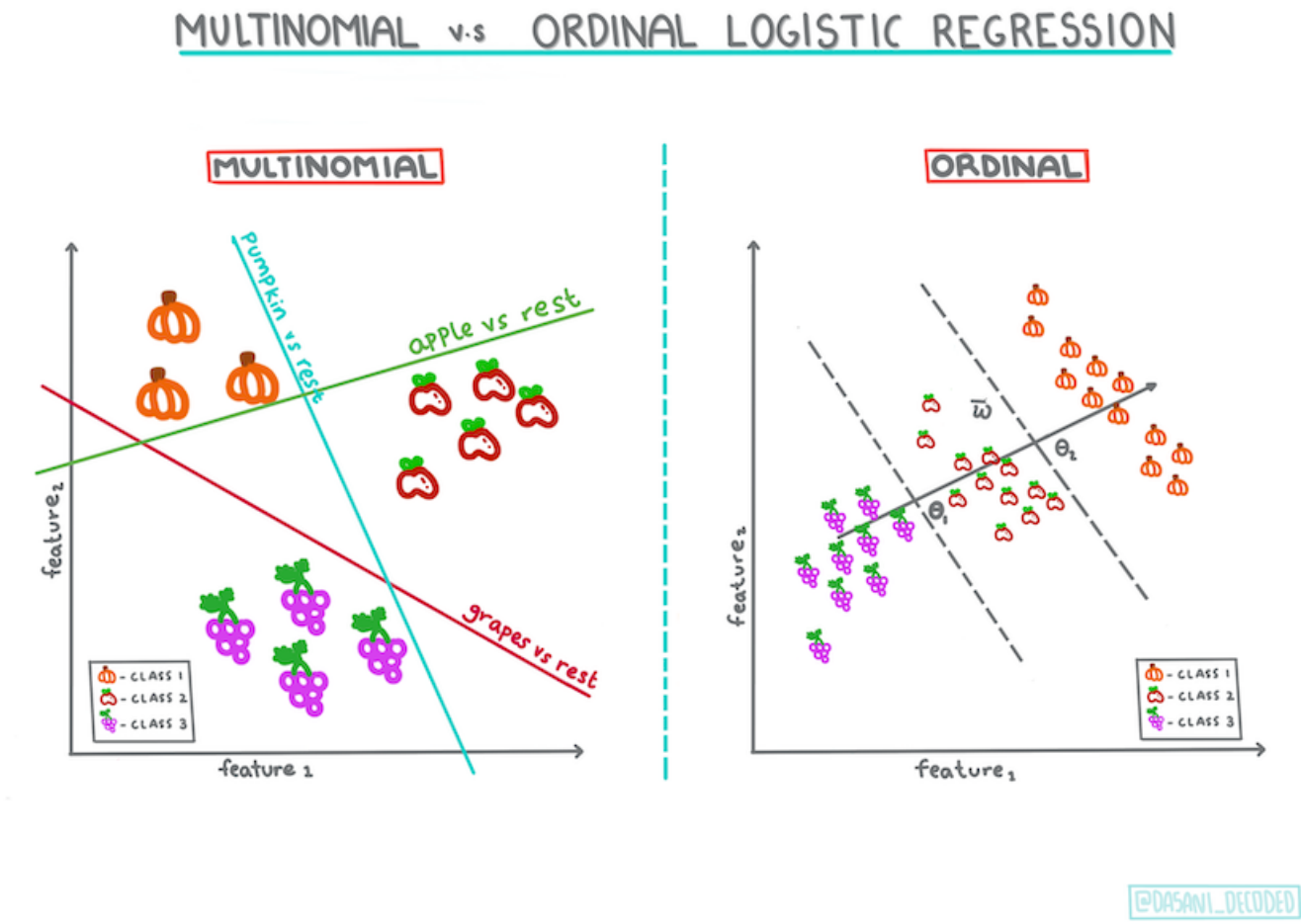


> Infographic by Dasani Madipalli

Other classifications

There are other types of logistic regression, including multinomial and ordinal:

- **Multinomial**, which involves having more than one category - "Orange, White, and Striped".
- **Ordinal**, which involves ordered categories, useful if we wanted to order our outcomes logically, like our pumpkins that are ordered by a finite number of sizes (mini,sm,med,lg,xl,xxl).



> Infographic by Dasani Madipalli

It's still linear

Even though this type of Regression is all about 'category predictions', it still works best when there is a clear linear relationship between the dependent variable (color) and the other independent variables (the rest of the dataset, like city name and size). It's good to get an idea of whether there is any linearity dividing these variables or not.

Variables DO NOT have to correlate

Remember how linear regression worked better with more correlated variables? Logistic regression is the opposite - the variables don't have to align. That works for this data which has somewhat weak correlations.

You need a lot of clean data

Logistic regression will give more accurate results if you use more data; our small dataset is not optimal for this task, so keep that in mind.

 Think about the types of data that would lend themselves well to logistic regression

Exercise - tidy the data

First, clean the data a bit, dropping null values and selecting only some of the columns:

1. Add the following code:

```
from sklearn.preprocessing import LabelEncoder

new_columns = ['Color','Origin','Item Size','Variety','City
Name','Package']

new_pumpkins = pumpkins.drop([c for c in pumpkins.columns if c not in
new_columns], axis=1)

new_pumpkins.dropna(inplace=True)

new_pumpkins = new_pumpkins.apply(LabelEncoder().fit_transform)
```

You can always take a peek at your new dataframe:

```
new_pumpkins.info
```

Visualization - side-by-side grid

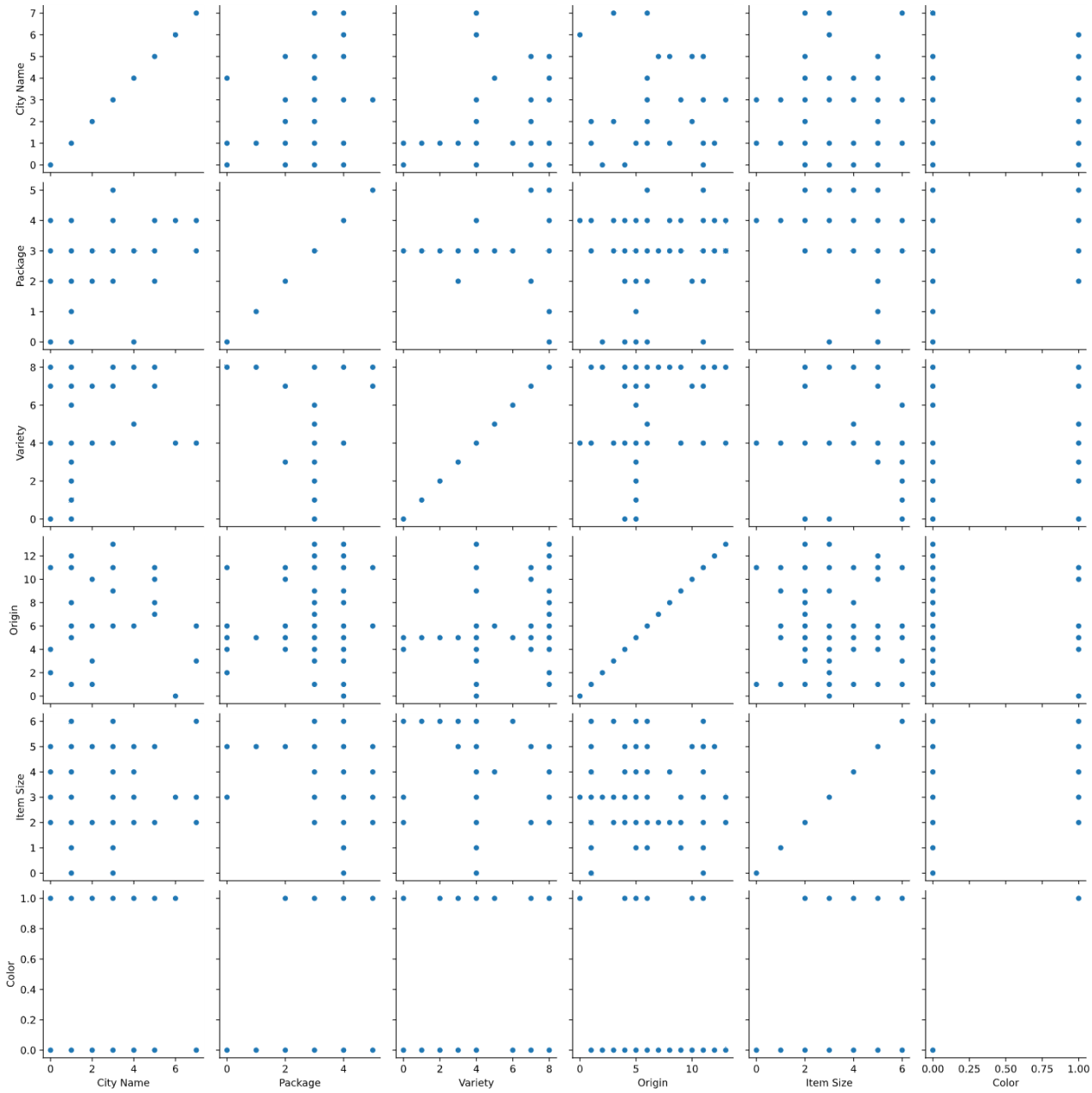
By now you have loaded up the starter notebook with pumpkin data once again and cleaned it so as to preserve a dataset containing a few variables, including `Color`. Let's visualize the dataframe in the notebook using a different library: Seaborn, which is built on Matplotlib which we used earlier.

Seaborn offers some neat ways to visualize your data. For example, you can compare distributions of the data for each point in a side-by-side grid.

1. Create such a grid by instantiating a `PairGrid`, using our pumpkin data `new_pumpkins`, followed by calling `map()`:

```
import seaborn as sns

g = sns.PairGrid(new_pumpkins)
g.map(sns.scatterplot)
```



By observing data side-by-side, you can see how the Color data relates to the other columns.

Given this scatterplot grid, what are some interesting explorations you can envision?

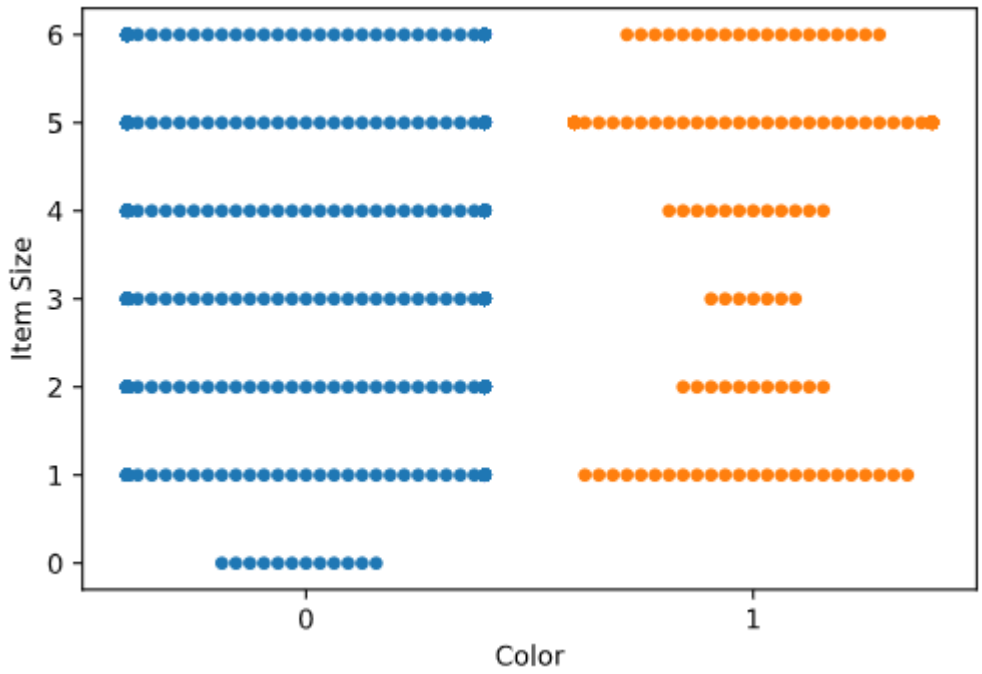
Use a swarm plot

Since Color is a binary category (Orange or Not), it's called 'categorical data' and needs 'a more specialized approach to visualization'. There are other ways to visualize the relationship of this category with other variables.

You can visualize variables side-by-side with Seaborn plots.

1. Try a 'swarm' plot to show the distribution of values:

```
sns.swarmplot(x="Color", y="Item Size", data=new_pumpkins)
```

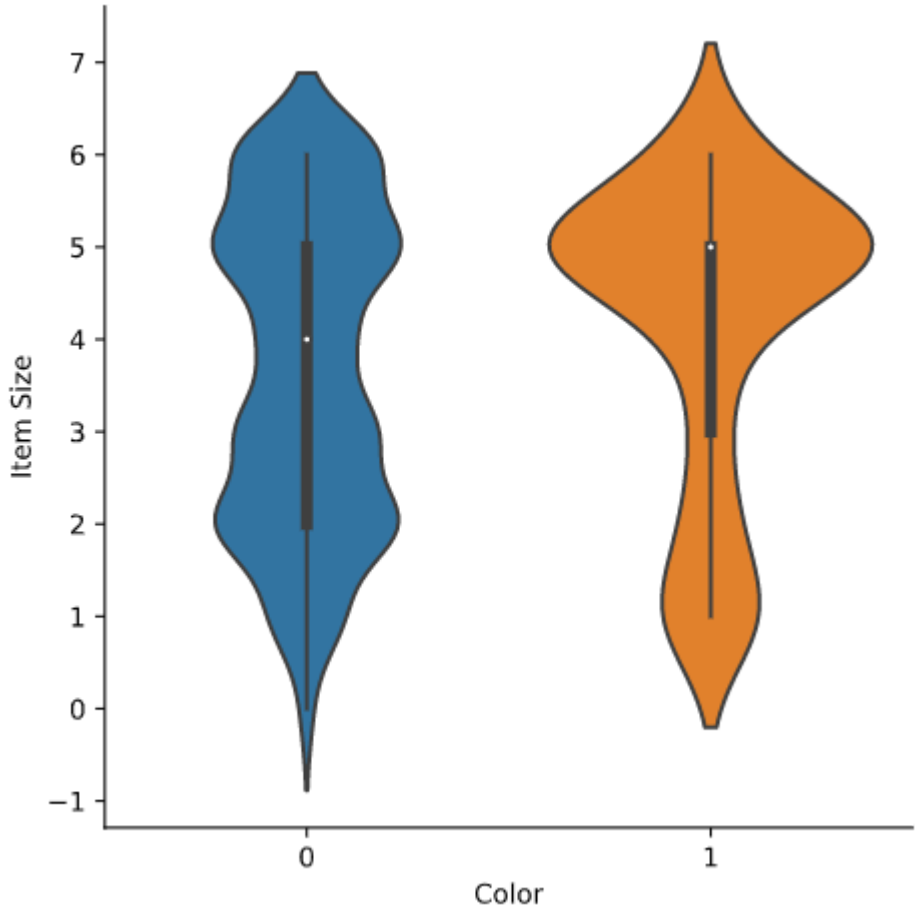


Violin plot

A 'violin' type plot is useful as you can easily visualize the way that data in the two categories is distributed. Violin plots don't work so well with smaller datasets as the distribution is displayed more 'smoothly'.

1. As parameters `x=Color`, `kind="violin"` and call `catplot()`:

```
sns.catplot(x="Color", y="Item Size",
            kind="violin", data=new_pumpkins)
```



 Try creating this plot, and other Seaborn plots, using other variables.

Now that we have an idea of the relationship between the binary categories of color and the larger group of sizes, let's explore logistic regression to determine a given pumpkin's likely color.

> **Show Me The Math**
>
> Remember how linear regression often used ordinary least squares to arrive at a value? Logistic regression relies on the concept of 'maximum likelihood' using sigmoid functions. A 'Sigmoid Function' on a plot looks like an 'S' shape. It takes a value and maps it to somewhere between 0 and 1. Its curve is also called a 'logistic curve'. Its formula looks like this:
>
> $$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$
>
> where the sigmoid's midpoint finds itself at x's 0 point, L is the curve's maximum value, and k is the curve's steepness. If the outcome of the function is more than 0.5, the label in question will be given the class '1' of the binary choice. If not, it will be classified as '0'.

Build your model

Building a model to find these binary classification is surprisingly straightforward in Scikit-learn.

1. Select the variables you want to use in your classification model and split the training and test sets calling `train_test_split()`:

```
from sklearn.model_selection import train_test_split

Selected_features = ['Origin','Item Size','Variety','City
Name','Package']

X = new_pumpkins[Selected_features]
y = new_pumpkins['Color']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)
```

2. Now you can train your model, by calling `fit()` with your training data, and print out its result:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)

print(classification_report(y_test, predictions))
print('Predicted labels: ', predictions)
print('Accuracy: ', accuracy_score(y_test, predictions))
```

Take a look at your model's scoreboard. It's not too bad, considering you have only about 1000 rows of data:

```
                precision    recall  f1-score   support

           0       0.85      0.95      0.90       166
           1       0.38      0.15      0.22        33

    accuracy                           0.82       199
   macro avg       0.62      0.55      0.56       199
weighted avg       0.77      0.82      0.78       199

Predicted labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
```

```
0 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
 0 0 0 1 0 1 0 0 1 0 0 0 1 0]
```

Better comprehension via a confusion matrix

While you can get a scoreboard report terms by printing out the items above, you might be able to understand your model more easily by using a confusion matrix to help us understand how the model is performing.

> 🎓 A 'confusion matrix' (or 'error matrix') is a table that expresses your model's true vs. false positives and negatives, thus gauging the accuracy of predictions.

1. To use a confusion metrics, call `confusion_matrix()`:

   ```
   from sklearn.metrics import confusion_matrix
   confusion_matrix(y_test, predictions)
   ```

   Take a look at your model's confusion matrix:

   ```
   array([[162,   4],
          [ 33,   0]])
   ```

In Scikit-learn, confusion matrices Rows (axis 0) are actual labels and columns (axis 1) are predicted labels.

	0	**1**
0	TN	FP
1	FN	TP

What's going on here? Let's say our model is asked to classify pumpkins between two binary categories, category 'orange' and category 'not-orange'.

- If your model predicts a pumpkin as not orange and it belongs to category 'not-orange' in reality we call it a true negative, shown by the top left number.
- If your model predicts a pumpkin as orange and it belongs to category 'not-orange' in reality we call it a false negative, shown by the bottom left number.
- If your model predicts a pumpkin as not orange and it belongs to category 'orange' in reality we call it a false positive, shown by the top right number.
- If your model predicts a pumpkin as orange and it belongs to category 'orange' in reality we call it a true positive, shown by the bottom right number.

As you might have guessed it's preferable to have a larger number of true positives and true negatives and a lower number of false positives and false negatives, which implies that the model performs better.

How does the confusion matrix relate to precision and recall? Remember, the classification report printed above showed precision (0.83) and recall (0.98).

Precision = tp / (tp + fp) = 162 / (162 + 33) = 0.8307692307692308

Recall = tp / (tp + fn) = 162 / (162 + 4) = 0.9759036144578314

 Q: According to the confusion matrix, how did the model do? A: Not too bad; there are a good number of true negatives but also several false negatives.

Let's revisit the terms we saw earlier with the help of the confusion matrix's mapping of TP/TN and FP/FN:

🎓 Precision: TP/(TP + FP) The fraction of relevant instances among the retrieved instances (e.g. which labels were well-labeled)

🎓 Recall: TP/(TP + FN) The fraction of relevant instances that were retrieved, whether well-labeled or not

🎓 f1-score: (2 * precision * recall)/(precision + recall) A weighted average of the precision and recall, with best being 1 and worst being 0

🎓 Support: The number of occurrences of each label retrieved

🎓 Accuracy: (TP + TN)/(TP + TN + FP + FN) The percentage of labels predicted accurately for a sample.

🎓 Macro Avg: The calculation of the unweighted mean metrics for each label, not taking label imbalance into account.

🎓 Weighted Avg: The calculation of the mean metrics for each label, taking label imbalance into account by weighting them by their support (the number of true instances for each label).

 Can you think which metric you should watch if you want your model to reduce the number of false negatives?

Visualize the ROC curve of this model

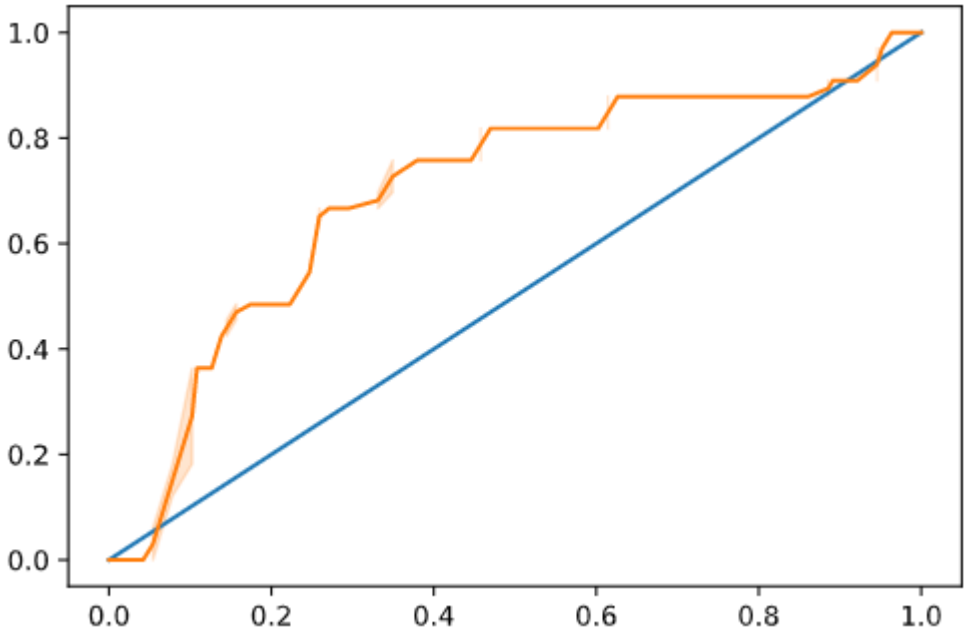
This is not a bad model; its accuracy is in the 80% range so ideally you could use it to predict the color of a pumpkin given a set of variables.

Let's do one more visualization to see the so-called 'ROC' score:

```
from sklearn.metrics import roc_curve, roc_auc_score

y_scores = model.predict_proba(X_test)
# calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores[:,1])
sns.lineplot([0, 1], [0, 1])
sns.lineplot(fpr, tpr)
```

Using Seaborn again, plot the model's Receiving Operating Characteristic or ROC. ROC curves are often used to get a view of the output of a classifier in terms of its true vs. false positives. "ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis." Thus, the steepness of the curve and the space between the midpoint line and the curve matter: you want a curve that quickly heads up and over the line. In our case, there are false positives to start with, and then the line heads up and over properly:



Finally, use Scikit-learn's `roc_auc_score` API to compute the actual 'Area Under the Curve' (AUC):

```
auc = roc_auc_score(y_test,y_scores[:,1])
print(auc)
```

The result is `0.6976998904709748`. Given that the AUC ranges from 0 to 1, you want a big score, since a model that is 100% correct in its predictions will have an AUC of 1; in this case, the model is *pretty good*.

In future lessons on classifications, you will learn how to iterate to improve your model's scores. But for now, congratulations! You've completed these regression lessons!
