17.3.1

Overview of Logistic Regression

Jill now believes that you are ready to try your hand at solving a classification problem with machine learning. The first model you will use is logistic regression, a popular classification model. She explains that despite its name, logistic regression is actually not a regression model. It is a classification model. With logistic regression, it is possible to try to answer questions such as whether a credit card holder is likely to miss a payment in the next month.

Let's walk through an example of building a logistic regression model in Python. First, download the following file.

Download 17-3-1-logistic_regression_demo.zip (https://2u-data-curriculum-team.s3.amazonaws.com/dataviz-online/module_17/17-3-1-logistic_regression_demo.zip)

In the previous linear regression example, the model attempted to predict a person's salary based on that person's years of work experience. In this model, is salary a categorical or a continuous variable?

- ○ A categorical variable, which can take on one of a limited, and usually fixed, number of possible values.
- ○ A continuous variable, which can take on any value, usually in a specified range.

Check Answer

Finish ▶

Logistic regression predicts binary outcomes, meaning that there are only two possible outcomes. An example of logistic regression might be to decide, based on personal information, whether to approve a credit card application. Multiple variables, such as an applicant's age and income, are assessed to arrive at one of two answers: to approve or to deny the application.

In other words, a logistic regression model analyzes the available data, and when presented with a new sample, mathematically determines its probability of belonging to a class. If the probability is above a certain cutoff point, the sample is assigned to that class. If the probability is less than the cutoff point, the sample is assigned to the other class.

In this section, we'll first walk through an example of logistic regression. Then we'll learn how the algorithm works in greater detail.

Practice Logistic Regression

Open `logistic_regression.ipynb`. In the first cell, Matplotlib and Pandas libraries are imported. In the second cell, a synthetic dataset is generated with Scikit-learn's `make_blobs` module. You do not need to focus on the details; the sole purpose here is to create two clusters of data named X and y:

```
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.datasets import make_blobs
X, y = make_blobs(centers=2, random_state=42)

print(f"Labels: {y[:10]}")
print(f"Data: {X[:10]}")
```

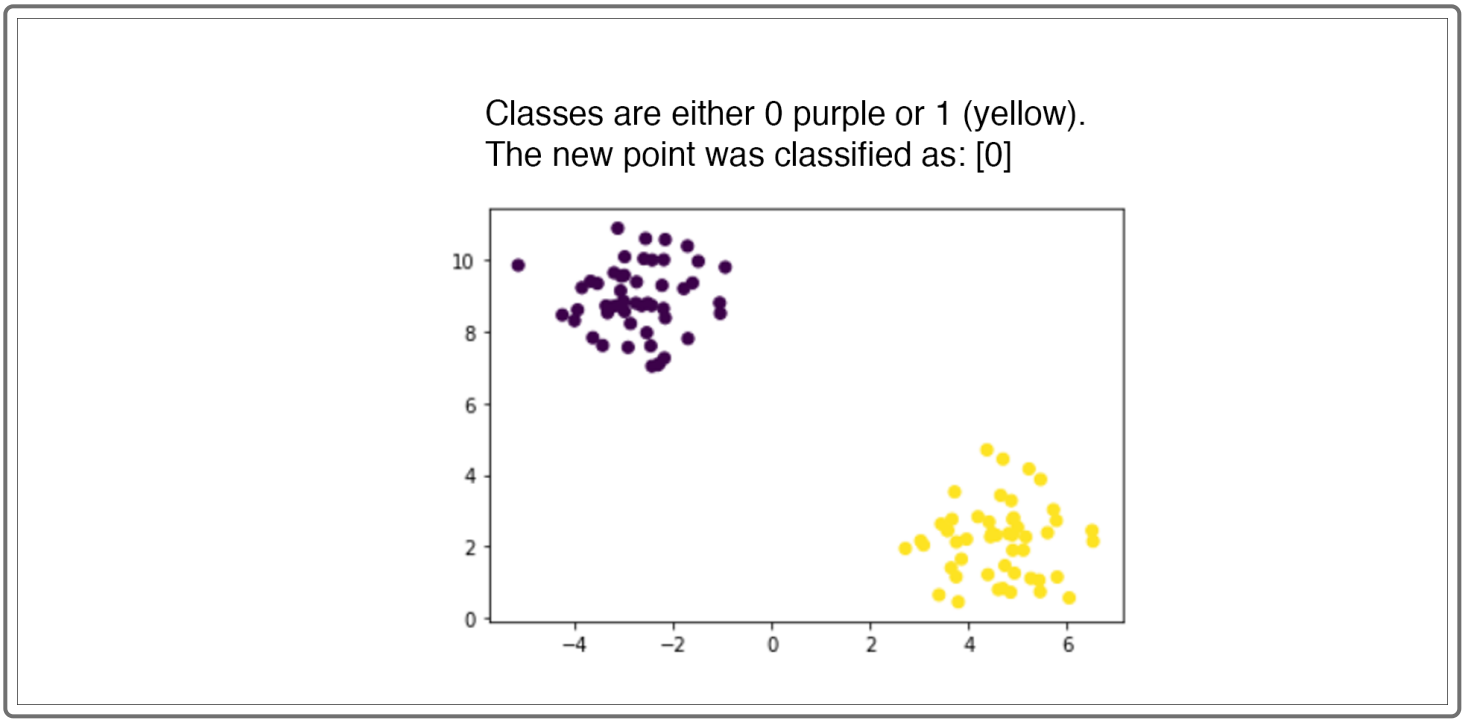
```
print(f"Labels: {y[:10]}")
print(f"Data: {X[:10]}")
   [ 5.72293008  3.02697174]
   [-3.05358035  9.12520872]
   [ 5.461939    3.86996267]
   [ 4.86733877  3.28031244]
   [-2.14780202 10.55232269]
   [ 4.91656964  2.80035293]
   [ 3.08921541  2.04173266]
   [-2.90130578  7.55077118]
   [-3.34841515  8.70507375]
```

The `centers` argument specifies the number of clusters in the dataset; in this case there are two clusters. The `random_state` ensures reproducibility of this dataset: even though the numbers in this dataset are generated pseudo-randomly, specifying 42 as the `random_state` argument will generate the identical dataset in the future.

Notice also that the dataset is split into X and y arrays. X contains the coordinates of each data point, and y contains information on the class of each data point. That is, each point belongs to either class 0 or 1. Purple dots belong to class 0, while yellow dots belong to class 1. The term "labels" is used here as a synonym for target variable.

In the next cell, the dataset is visualized. We see a scatter plot with two clearly separated groups of data points. The data points to the left of 0 on the x-axis are purple, while those to its right are colored yellow. The logistic regression model will be trained on this data, and it will be able to categorize a new data point as one of the two classes (yellow or purple):

```
plt.scatter(X[:, 0], X[:, 1], c=y)
```



Split the Dataset into Train and Test Sets

In the next cell, we split the dataset into two: train and test datasets. Imagine that you are studying for a major exam and have a test bank of 100 questions available. A good strategy might be to use 75 of the questions as you study for the exam. These questions will help pinpoint important topics and patterns of information you might encounter on the actual exam.

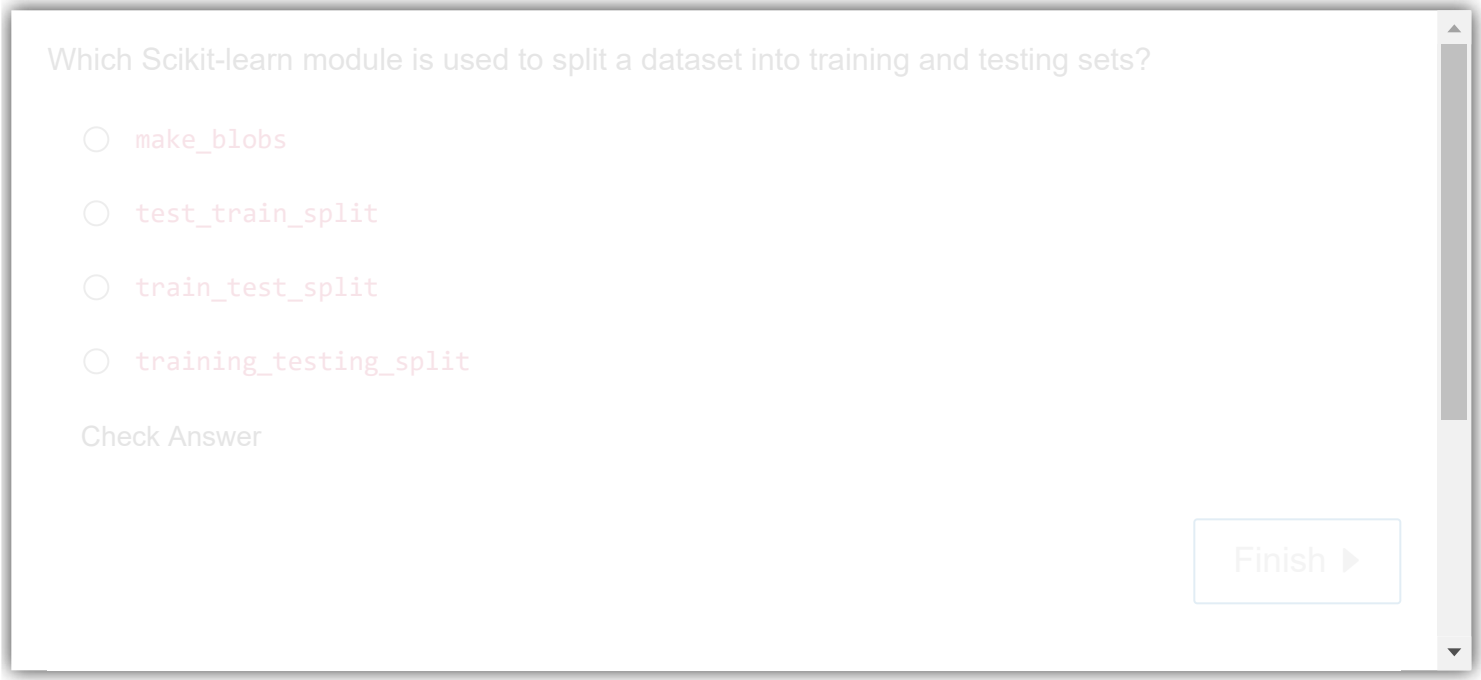
Then, a few days before the exam, you sit down to answer the remaining 25 questions in a mock test. Since you haven't seen these questions before, how well you perform on them will give you a good idea of how you will do on the actual exam.

Similarly, a dataset is split into training and testing sets in supervised learning. The model uses the **training dataset** to learn from it. It then uses the **testing dataset** to assess its performance. If you use your entire dataset to train the model, you won't know how well the model will perform when it encounters unseen data. That is why it's important to set aside a portion of your dataset to evaluate your model.

In the next cell, the dataset is split into training and testing sets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=1, stratify=y)
```

Recall that X is the input, and y is the output, or what we wish to predict. Scikit-learn's `train_test_split` module takes X and y as arguments and splits each into training and test sets. So in all, we end up with four sets. X is split into `X_train` and `X_test` sets, and y is split into `y_train` and `y_test` sets.

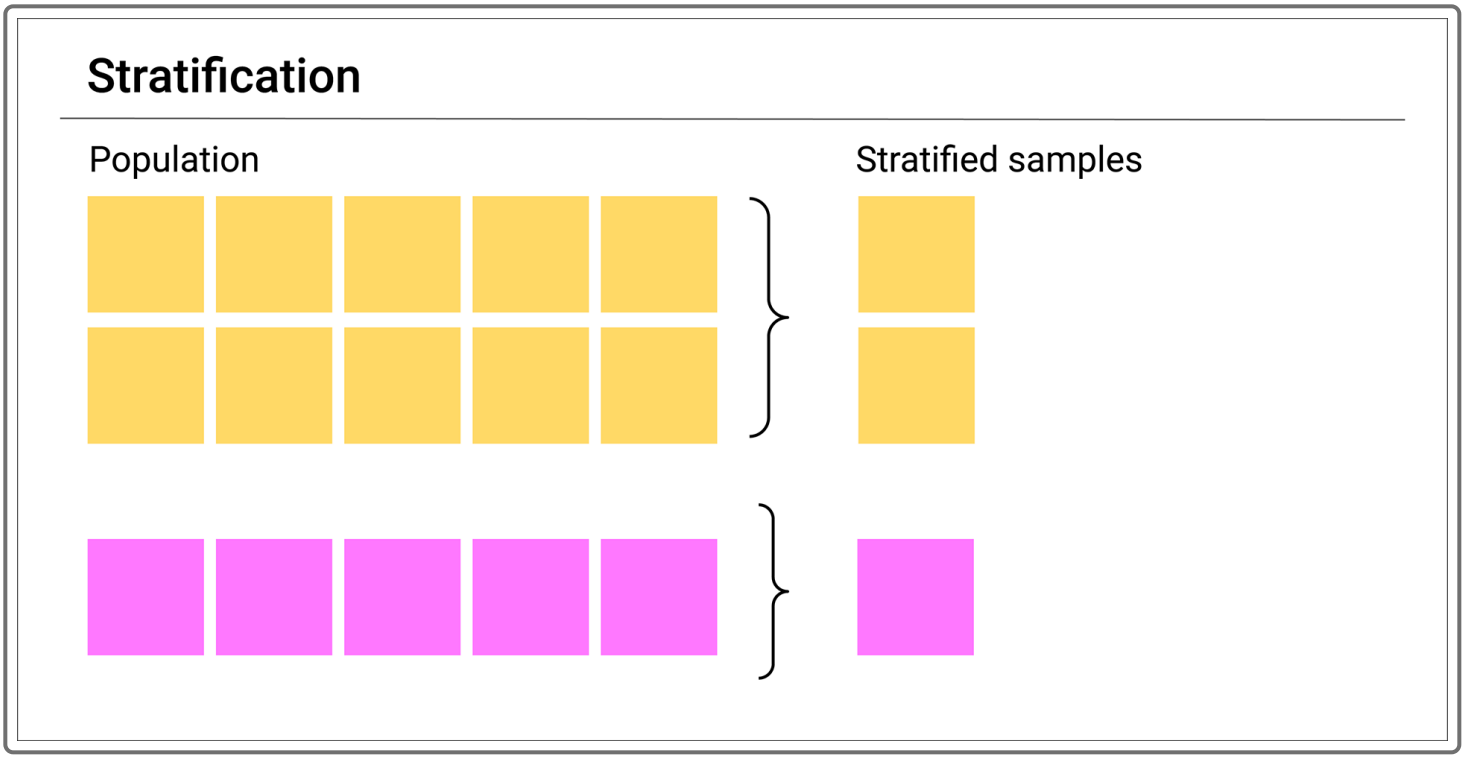


The `train_test_split()` function has four arguments here. The first two, as discussed, are X and y, which are split into train and test sets. We saw previously that `random_state` is used to make the data reproducible. Specifying a random state of 1 ensures that the same rows are assigned to train and test sets, respectively. A different random state number would distribute different rows of data to the train and test sets. Note that the `random_state` argument is used here only for pedagogical purposes, so that you will obtain the same results shown here when you run the notebook. You will not need to use it when you create your own analyses.

NOTE

Consult the **sklearn documentation** (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) for additional information about the `train_test_split()` function.

The last argument, `stratify`, also deserves discussion. Stratifying a dataset divides it proportionally. For example, say that 60% of a dataset belongs to the yellow class, and 40% of it belongs to the purple class. Stratifying it ensures that when the entire dataset is split into training and testing sets, 60% of both will belong to the yellow class, and 40% will belong to the purple class. Without specifying stratification, samples will be assigned randomly, so that it is possible to end up with a training set with a 68-32 split and a testing set with a 65-35 split, for example:



In our current dataset, both the purple and yellow clusters have roughly equal sizes, and a stratified split will not make a significant difference in the outcome.

However, imagine a scenario in which one class is significantly larger than the other. For example, let's say that we're analyzing 10,000 credit card transactions, and only 40 of them are flagged as fraudulent. We might allocate 75% of the dataset to the training set, and 25% to the testing set, so we'd expect the training set to contain 75% of the fraudulent transactions (30), and the testing set to contain 25% of the fraudulent transactions (10). Without stratification, it's possible for the fraudulent transactions to be distributed disproportionately—for example, 20 to the

training set and 20 to the testing set. And as the model trains on the unrepresentative data, it can reach wrong conclusions. It's therefore important to consider stratifying the data, especially when the classes are severely unbalanced, or when the dataset is small.

IMPORTANT

Stratification is especially important when there's a class imbalance.

Instantiate a Logistic Regression Model

In the next cell, we create a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(solver='lbfgs', random_state=1)
classifier
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
   intercept_scaling=1, max_iter=100, multi_class='warn', penalty='12',
   random_state=1, solver='lbfgs' tol=0.0001, warm_start=False)
```

Let's break down this code:

- We first import `LogisticRegression` from the Scikit-learn library, and then instantiate the model.
- The `solver` argument is set to `'lbfgs'`, which is the default setting, but we are adding it to point out this is an algorithm for learning and optimization. The particular solver isn't very important in this example, but note that a number of optimizers exist.
- Once again, the `random_state` is specified so that you'll be able to reproduce the same results as you run this notebook.

Train the Logistic Regression Model

The next step is to train the model with the training set data. We use the `fit()` method to train the model:

```
classifier.fit(X_train, y_train)
```

Validate the Logistic Regression Model

The next step is to create predictions and assemble the results into a Pandas DataFrame:

```
predictions = classifier.predict(X_test)
pd.DataFrame({"Prediction": predictions, "Actual": y_test})
```

Let's break down this code:

- The first line of code uses the `predict()` method to create predictions based on `X_test`.
- The second line creates a DataFrame of predicted values and actual values.

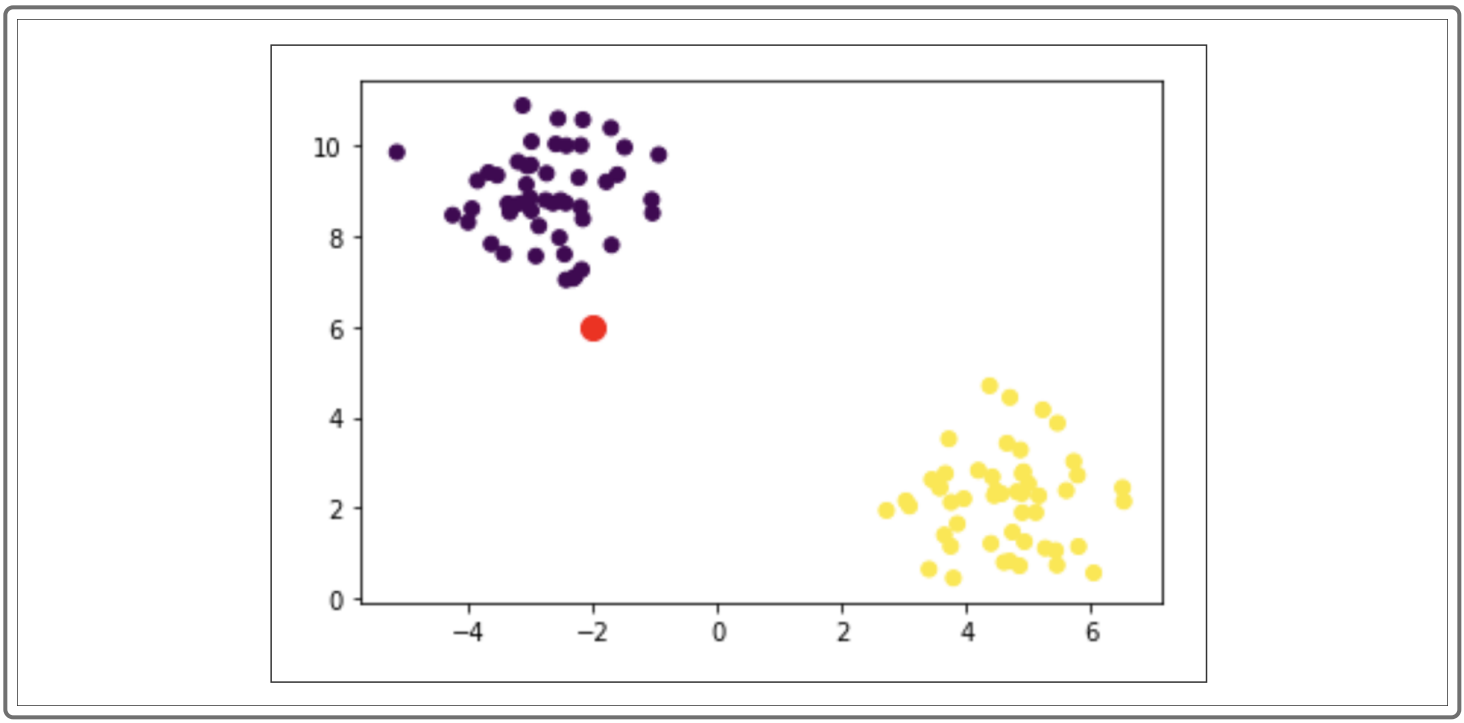
Next, we validate the model, or evaluate its performance. You can create a model and use it to make predictions, but you won't know how good the model is unless you assess its performance:

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

The model achieved an accuracy score of 1.0. This means that every single observation in the testing set was predicted correctly by the model. All samples belonging to class 1 (yellow) were correctly predicted, and all samples belonging to class 0 (purple) were likewise correctly predicted by the model. Although perfect accuracy was achieved in this example, it is rare in actual practice. Moreover, an extremely high metric should raise your suspicion of overfitting. Overfitting refers to an instance in which the patterns picked up by a model are too specific to a specific dataset. We will discuss overfitting in greater detail later in the module.

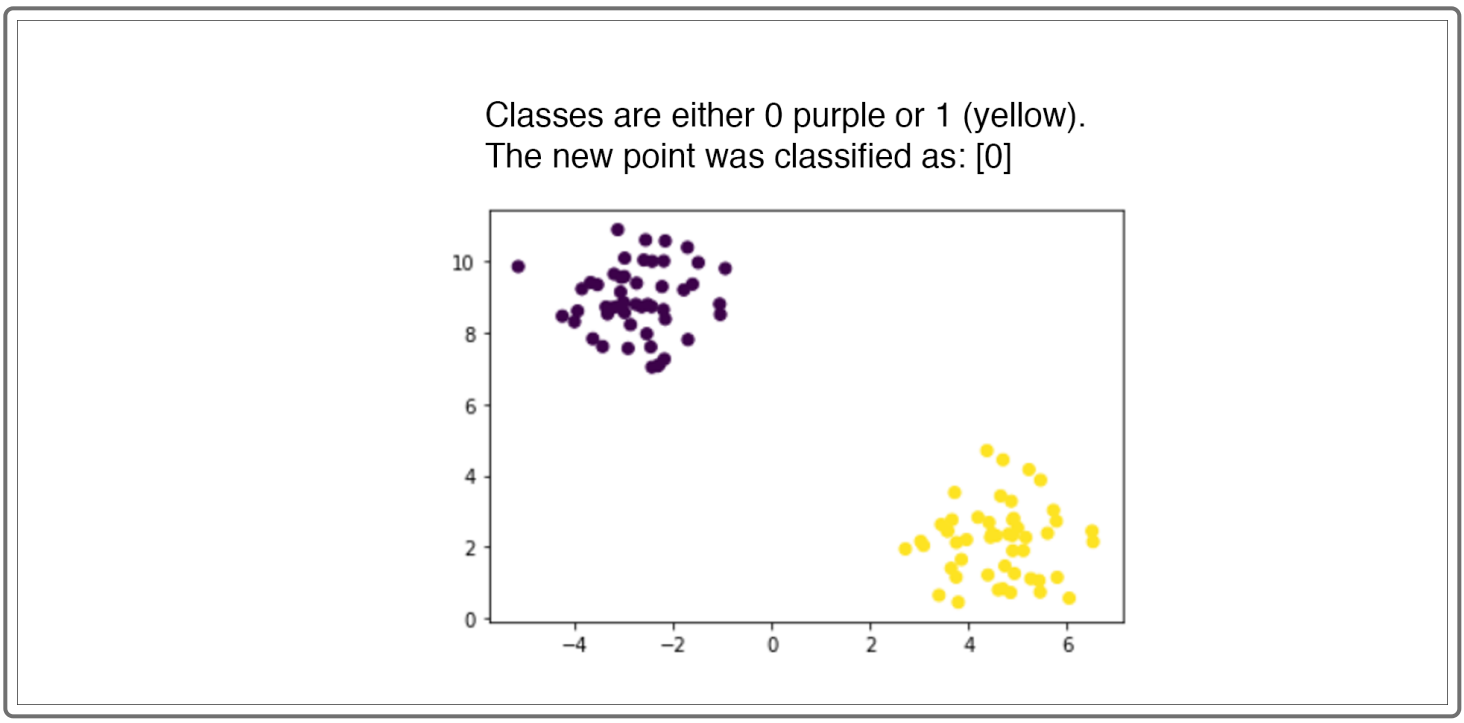
The sole purpose of the next cell is to create a new data point, which shows up as a red dot on the new plot. The logistic regression model will then classify this sample as belonging to class 0 (purple) or class 1 (yellow):

```
import numpy as np
new_data = np.array([[-2, 6]])
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.scatter(new_data[0, 0], new_data[0, 1], c="r", marker="o", s=100)
plt.show()
```



Once again, we use `predict()` to predict which class the new data point belongs to. It turns out, as expected, that the red dot belongs to the purple cluster:

```
predictions = classifier.predict(new_data)
print("Classes are either 0 (purple) or 1 (yellow)")
print(f"The new point was classified as: {predictions}")
```



Let's summarize the steps we took to use a logistic regression model:

1. Create a model with `LogisticRegression()`.
2. Train the model with `model.fit()`.
3. Make predictions with `model.predict()`.
4. Validate the model with `accuracy_score()`.