Explore unsupervised learning and K-means

Video: Welcome to week 3
2 min

Video: Introduction to K-means
4 min

Reading: More about K-means
20 min

Reading: Follow-along instructions: Use K-means for color compression with Python
10 min

Lab: Annotated follow-along guide: Use K-means for color compression with Python
20 min

Video: Use K-means for color compression with Python
7 min

Reading: Clustering beyond K-means
10 min

Practice Quiz: Test your knowledge: Explore unsupervised learning and K-means
3 questions

Evaluate a K-means model

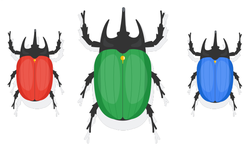
Review: Unsupervised learning techniques

More about K-means

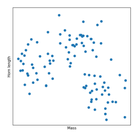
Previously, you learned about the K-means algorithm, an unsupervised learning technique used to cluster unlabeled data. You have a foundational understanding of its underpinning theory. In this reading, you will examine this methodology in greater depth by studying how the algorithm behaves when different *k* values are used to cluster two-dimensional data. You will gain a deeper understanding of K-means by comparing cases where it works well, and those where it works poorly. Being aware of the strengths and limitations of this methodology will enable you to use it appropriately and effectively as a data professional.

Consider the data

Suppose you're presented with data for 100 samples selected from three species of horned beetles.



The data compares total body mass to horn length. Because it's two-dimensional, you can plot it and examine the results to identify patterns or trends.



What do you notice about the data in the scatter plot above? Does it fall into groups? Can you tell that three species are represented in this data?

What if we color the same data by species?

Hopefully you were able to discern similar clusters on your own. Notice that the points in this scatter plot are loosely grouped into three clusters. In this case, each color represents a different species of beetle. You can use K-means to cluster this data.

Cluster with K-means

Before continuing, it's important to note that in this illustrative case:

1. You know there are three species of beetles.
2. By using a scatter plot, you can verify that they are clustered into three groups.

Remember, you will not always know how many clusters you *should*have, and you probably won't be able to visualize your data so easily because it will likely have more than three features (i.e., dimensions). We use this example because it is effective for teaching concepts to learners.

Modeling: *k* = 3

Since you know there are three clusters, you know to set *k* equal to three when you build your model. Recall the steps of the K-means algorithm:

1. Randomly place centroids in the data space.
2. Assign each point to its nearest centroid.
3. Update the location of each centroid to the mean position of all the points assigned to it.
4. Repeat steps 2 and 3 until the model converges (i.e., all centroid locations remain unchanged with successive iterations).

Now, consider these steps as the K-means algorithm iterates over the data.

Iteration 0: The centroids are placed at random, and the points are assigned to their nearest centroid. (**Note:** Centroids are indicated by "X".)

Iteration 1: Update centroid locations, reassign points to their nearest centroid.

Iteration 2: Repeat.

Iteration 3: Repeat.

Iteration 4: Repeat.

Stop.

Why stop here? Well, notice that there is no meaningful difference between the results of iteration 3 and iteration 4. The algorithm has converged (the centroids have stopped moving and cluster assignment has stabilized). The three points marked with squares were assigned to the wrong clusters by the model. Except for these points, the cluster assignments of the data were correct after just two iterations, and the centroid locations converged after three. *K*-means is both effective and efficient in clustering this data.

Note that in the case above, the centroids may still be moving after iteration 4, but if so, all movement is below a convergence tolerance that can be set by the modeler. The default in scikit-learn is 0.0001. If all centroid locations move less than this distance, the model is declared converged.

Modeling: *k* = 2

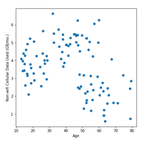
Suppose you didn't know how best to cluster your data, and you decided to set the value of *k* to two. Note how the clustering converges in this case:

Not only did it take six iterations to converge, but the data in the top cluster was split, resulting in class assignment that, in this case, you know to be incorrect (because there are three species of beetle). This is an example of what can happen when you don't use the best value for *k*. If you were to set *k* to a value greater than three, the algorithm could similarly split the data into unintuitive clusters.

When there are no correct answers

In the previous examples, you knew that there were three different kinds of beeties, and therefore to set *k* to three. The purpose of using this labeled data for an unsupervised learning demonstration was to illustrate that K-means can effectively group data. However, more often, you won't have any class labels to determine how "good" your model is.

Suppose you have the same data points, but they represent something else—in this new case, mobile phone owners plotted by age and cellular (non-wifi) data usage.



You may want to perform a market segmentation analysis on this data. In this scenario, there is no "correct" answer. You cannot verify your model's clustering against a baseline truth. It's possible that you could make a case to use two, three, four, or more clusters. You might not have any idea how many to choose. There are tools to help you make this decision when you cannot visualize your data or don't otherwise know how many clusters you should have, which you'll learn later in this lesson.

A note on K-means++

Earlier in this course, you learned the importance of running K-means multiple times with different initial positions of the centroids to help avoid using a model that gets stuck in local minima. Fortunately, most machine learning packages have improved implementations of K-means that make it easier for you by removing this requirement.

In scikit-learn, this implementation is called **K-means++**. K-means++ still randomly initializes centroids in the data, but it does so based on a probability calibration. Basically, it randomly chooses one point within the data to be the first centroid, then it uses other data points as centroids, selecting them pseudo-randomly. The probability that a point will be selected as a centroid increases the farther it is from other centroids. This helps to ensure that centroids aren't initially placed very close together, which is when convergence in local minima is most likely to occur.

K-means++ is the default implementation when you instantiate K-means in scikit-learn. If you don't want to use this implementation and would prefer to start with truly random centroids, you can change this by setting the "init" parameter to "random", but rarely would you want to do this.

Key takeaways

When using K-means to model your data, it's important to understand how the methodology works to cluster data. If you know the most common ways by which K-means can arrive at both good and bad clustering results, it will help you understand how to choose the best value for *k* and ultimately to make better decisions when building models as a data professional.

Resources for more information

- For further reading on the K-means implementation in scikit-learn, refer to the documentation.
- For the theory behind K-means++, see
  Arthur, D.; Vassilvitskii, S. (2007). "k-means++: the advantages of careful seeding" (PDF). Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 1027–1035
  .

Mark as completed

Like Dislike Report an issue