Clustering

Using Python to Group Data Based on Similarities

Aniket Adhikari

2023-11-15

On this page

Supervised Learning vs. Unsupervised Learning **1**

What is Clustering? **2**

How Does Clustering Work? **2**

Clustering Use Cases **3**

K-Means Clustering **3**
- Process of K-Means Clustering? . . . 4
- Choosing the Number of Clusters k . . . 4
- Example of K-Means Clustering . . . 4

Gaussian Mixture Models (GMMs) **10**
- How Does GMMs Work . . . 10
- Example of GMM . . . 10

DBSCAN **12**
- How Does It Work? . . . 12
- Example of DBSCAN . . . 13

Supervised Learning vs. Unsupervised Learning

Before we talk about clustering, we must address some integral concepts related to it: supervised and unsupervised learning.

Supervised learning is a machine learning technique that is used to train/teach a machine using **labeled** data. Labeled data implies that the data is already tagged with the correct answer. Teaching the machine on labeled data allows for future data to be correctly predicted. Under the umbrella of supervised learning, there are 2 categories of algorithms:

- **Classification**: Output variable is a category, so we are looking to categorize the output
- **Regression**: Output variable is a real value, so we are looking to predict the value of the output

Unsupervised Learning is a machine learning technique that uses used to train/teach a machine using that isn't labeled or classified. The machine is now responsible for grouping the data according to similarities in characteristics without prior training. This is much harder because there is no "teacher" here, meaning the machine is tasked with finding the hidden structure in the unlabeled data. Under the umbrella of unsupervised learning, there are 2 categories of algorithms:

- **Clustering**: Grouping of cata to find similarities
- **Association**: Discover riles that describe large portions of data

Table 1: Comparison of Supervised and Unsupervised Learning

	Supervised Learning	Unsupervised Learning
Input	Labeled Data	Unlabeled, uncategorized
Accuracy	Highly accurate	Less acurate
Output	Categorized or real values	Groupings

What is Clustering?

As mentioned, clustering is a type of unsupervised learning. Here, we are grouping data so that each group, or cluster, exhibits similar qualities. Ultimately, the goal of clustering to uncover intrinsic patterns and structures within data that can be used for analysis.

How Does Clustering Work?

Algorithms that are focused on clustering measure similarity between data points across a set of features. Features should be continuous variables but can be categorical. However, categorical data needs special encoding. Data points in the cluster that appear close to each

other based on the features are grouped together, which the data points that are far away are separated into different clusters. There are several approaches to clustering, including:

- **K-Means**: Grouping of data points in K clusters by minimizing the intra-cluster sum-of-squares. This requires setting the number of clusters up front, as we'll see in the application section.
- **Hierarchial**: Hierarchy of clusters are built iteratively
- **DBSCAN**: groups dense regions of points and considers the sparse areas as outliers. Intuitively detects arbitrary cluster shapes.
- **Gaussian Mixture Models**: Fits data as a mixture of Gaussian distributions where clusters are modeled using mean and covariance parameters.

Clustering Use Cases

So when is it a good time to use clustering? As mentioned, the best time to use clustering would be when we have data that is unlabeled. The following are more specific reasons to use clustering

1. **Exploratory Data Analysis**: Clustering can help to reveal intrinsic groups and patterns in data without prior knowledge. It can open the door for further analysis by uncovering segements that were previously unknown
2. **Customer Segementation**: Cluster customers based on certain attributes like demographics, purchasing beahvior, and more to achieve targeted marketing
3. **Social Network Analysis**: Identify communities within a social network by clustering nodes based on connectivity and usage patterns
4. **Anomaly Detection**: Detect anomalous data points that might not fit into a cluster to detect potential fraud or network attacks.

K-Means Clustering

K-Means is one of the most popular clustering algorithms that is used for discovering intrinsic groups in unlabeled data. The

K-Means partitions a dataset into a predefined number of clusters, k. It does this by minimizing the sum of distances between each data point and its assigned centroid, also known as the within-cluster sum of squares (WCSS).

Process of K-Means Clustering?

1. Select k initial centroids for the clusters
2. Assign each data point to its closest centroid point based on Euclidean distance
3. Recompute the centroids as the mean of all data points assigned to that cluster
4. Repeat steps 2-3 until convergence, meaning the centrods no longer change between iterations

Choosing the Number of Clusters k

When choosing the number of clusters for K-Means it's important to choose the right number of clusters because it affects the quality of clustering.

You could simply try different integer values to represent k through a process of guessing-and-checking. Then, you could examine which one of k values produces clustering visuals that makes the most sense for your data.

Another way to select k is the *elbow method*, which involves plotting the WCSS vs. k. The part of the curve that is shaped like an elbow will contain what is likely a good choice for the k

There is also the *silhouette method*, which measures how well each data point fits into its assigned cluster compared to other clusters and selects the k with the highest average silhouette score.

Example of K-Means Clustering

We can actually apply K-Means clustering in Python. Similar to the last post, I want to demonstrate how that can be done in code using a dataset of Pokemon from Generations 1-7.

First we can start off by importing the libraries that are needed for the code. In addition to the same libraries from the previous post, `pandas` and `matplotlib`, we are going to import `numpy` as well as `KMeans` from `sklearn.cluster`.

`numpy` will simply be used for transforming a python array of colors into something that can be used for designating clusters to be of certain colors.

`KMeans` will be used for the actual clustering.

```
from sklearn.cluster import KMeans
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

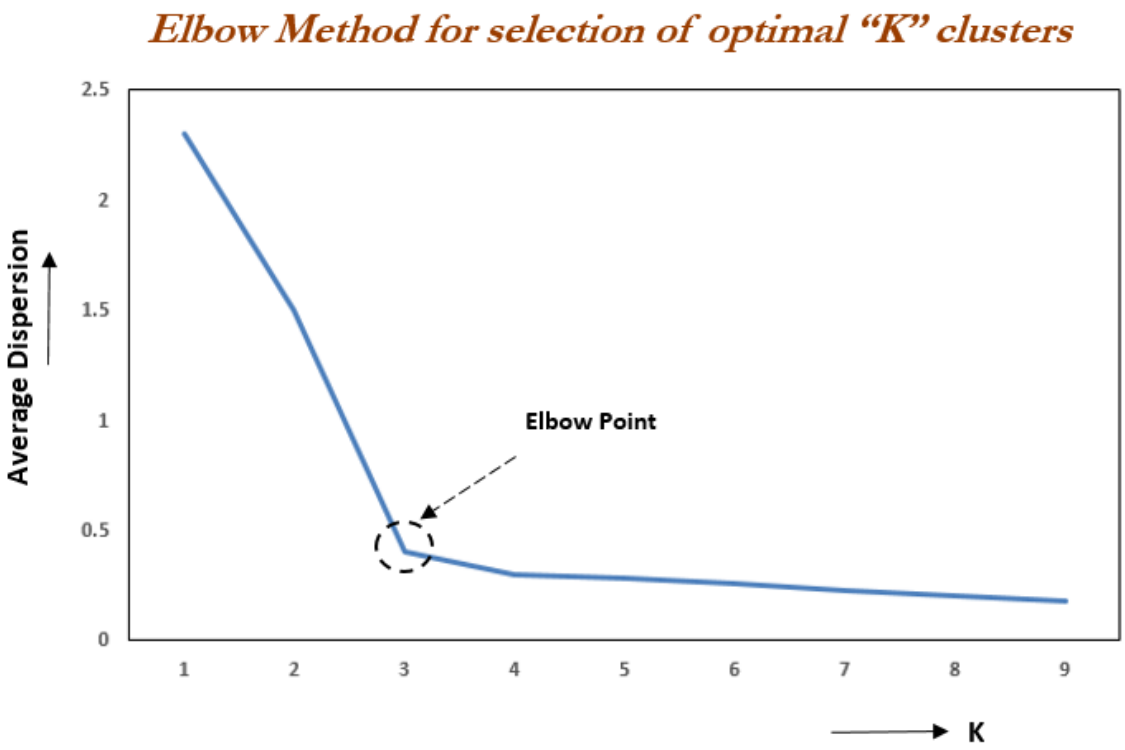


Figure 1: Elbow Method from O'Reilly

We are then going to want to load our data. Again, it's from the Pokemon dataset and it's comma-separated.

```
data = pd.read_csv("../../datasets/pokemon.csv", sep=",")
data
```

```
/Users/aniketadhikari/anaconda3/envs/homl3/lib/python3.10/site-packages/IPython/core/formatte
  return method()
```

	attack	base_egg_steps	base_happiness	base_total	capture_rate	defense	experien
0	49	5120	70	318	45	49	
1	62	5120	70	405	45	63	
2	100	5120	70	625	45	123	
3	52	5120	70	309	45	43	
4	64	5120	70	405	45	58	
5	104	5120	70	634	45	78	
6	48	5120	70	314	45	65	
7	63	5120	70	405	45	80	
8	103	5120	70	630	45	120	
9	30	3840	70	195	255	35	
10	20	3840	70	205	120	55	
11	45	3840	70	395	45	50	
12	35	3840	70	195	255	30	
13	25	3840	70	205	120	50	
14	150	3840	70	495	45	40	
15	45	3840	70	251	255	40	
16	60	3840	70	349	120	55	
17	80	3840	70	579	45	80	
18	56	3840	70	253	255	35	
19	71	3840	70	413	127	70	
20	60	3840	70	262	255	30	
21	90	3840	70	442	90	65	
22	60	5120	70	288	255	44	
23	95	5120	70	448	90	69	
24	55	2560	70	320	190	40	
25	85	2560	70	485	75	50	
26	75	5120	70	300	255	90	
27	100	5120	70	450	90	120	
28	47	5120	70	275	235	52	
29	62	5120	70	365	120	67	
30	92	5120	70	505	45	87	
31	57	5120	70	273	235	40	
32	72	5120	70	365	120	57	
33	102	5120	70	505	45	77	
34	45	2560	140	323	150	48	
35	70	2560	140	483	25	73	
36	41	5120	70	299	190	40	
37	67	5120	70	505	75	75	
38	45	2560	70	270	170	20	
39	70	2560	70	435	50	45	
40	45	3840	70	245	255	35	
41	80	3840	70	455	90	70	
42	50	5120	70	320	255	55	
43	65	5120	70	395	120	70	
44	80	5120	7 70	490	45	85	
45	70	5120	70	285	190	55	
46	95	5120	70	405	75	80	
47	55	5120	70	305	190	50	
48	65	5120	70	450	75	60	
49	55	5120	70	265	255	30	
50	100	5120	70	425	50	60	
51	35	5120	70	290	255	35	

Recall that clustering requires features to be **continuous variables**. In this dataset, we have a couple of different continuous variables, such as:

- `attack`
- `defense`
- `base_total`
- `sp_attack`
- `sp_defense`
- `height_m`
- `percentage_male`
- `speed`
- `weight_kg`

Some of these variables are kind of useless though because they aren't going to give us anything interesting. Instead we want to focus on the stats associated with each Pokemon. More specifically, we want to focus on

- `attack`
- `defense`
- `sp_attack`
- `sp_defense`

Honestly, I had trouble trying to decide on whether I wanted to cluster based on `attack` and `defense` or `sp_attack` and `sp_defense`. As a result, I ended up combining them in a way so that `attack` and `sp_attack` add up to `total_attack` and `defense` and `sp_defense` add up to `total_defense`.

```
data['Total Attack'] = data['attack'] + data['sp_attack']
data['Total Defense'] = data['defense'] + data['sp_defense']
```

Here we are storing data for `total_attack` and `total_defense` into variable `X`

```
x_val = 'Total Attack'
y_val = 'Total Defense'
values = [x_val, y_val]
X = data[values]
```

After we have our data, we need to determine the number of clusters, or k. This is up to you, but I wanted to cluster it by 3 so we can have clusters of **weak**, **intermediate**, and **strong**.

```
kmeans = KMeans(n_clusters=3)
```

We then need to compute the K-Means clusters by using the `fit()` function. Here we supply the method with the data that was gathered earlier.

```
kmeans.fit(X)
```

```
KMeans(n_clusters=3)
```

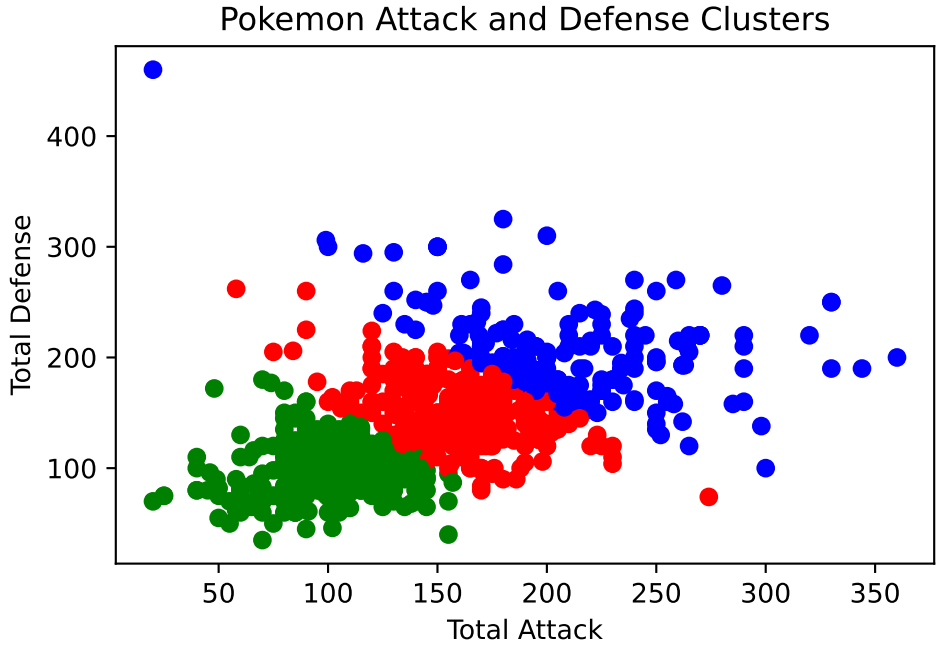
We can then store the labels of the cluster into `labels` so that it can be used for applying colors to each cluster with the `colormap` array

```
labels = kmeans.labels_
colormap = np.array(['red', 'green', 'blue'])
```

Afterwards, we are going to actually plot a scatter graph of the clustered data, using `total_attack` and `total_defense` as features

```
plt.scatter(X['Total Attack'], X['Total Defense'], c=colormap[labels])
plt.title('Pokemon Attack and Defense Clusters')
plt.xlabel('Total Attack')
plt.ylabel('Total Defense')

plt.show()
```



In the above clustering, data points labeled in red are considered "weak", green are considered "intermediate", and blue is considered "strong.

I'm going to be replaying Pokemon games over the winter break so I'm hoping this scatter will give me an indication as to what Pokemon I should catch!

Gaussian Mixture Models (GMMs)

Gausian Mixture Models (GMMs) are probabilistic models that assumes data points are generated from a mixture of Gaussian distributions. GMMs are a popular clustering algorithm with applications such as density estimation, data compression, and more.

How Does GMMs Work

A GMM models each cluster as a Gaussian distribuion, characterized by a mean and covariance matrix. The complete model is a weighted sum of the component Gaussian densities.

The GMM fitting process determines the parameters of each Gaussian as well as the mixture weights. This is done through iterative expectation-maximiation algorithm that converges to find the maximum likelihood estimates.

The number of Gaussian components corresponds to the number of clusters. Choosing the right number of clusters is important to avoid underfitting or overfitting.

Example of GMM

Here we are importing a couple of different libraries that have been used before. We are importing `GaussianMixture` in order to create Gaussian Mixture Models.

```
1 import pandas as pd
2 from sklearn.mixture import GaussianMixture
3 import matplotlib.pyplot as plt
```

This time, I wanted to a different dataset because the one with Pokemon didn't seem to be creating clusters very well. The dataset being imported consists of customer data, including information such as

- `Gender`
- `Age`
- `Annual Income ($)`
- `Spending Score (1-100)`

- Profession
- Work Experience
- Family Size

```
customer_data = pd.read_csv('../../datasets/Customers.csv')
```

I am particularly interested in using variables that are continuous, which are `Age`, `Annual Income`, and `Work Experience`. We are using these variables as features for segmentation

```
selected_features = ['Age', 'Annual Income ($)', 'Work Experience']
```

Here we are preprocessing by extracting and scaling the selected features

```
customer_features = customer_data[selected_features]
customer_features = (customer_features - customer_features.mean()) / customer_features.std()
```

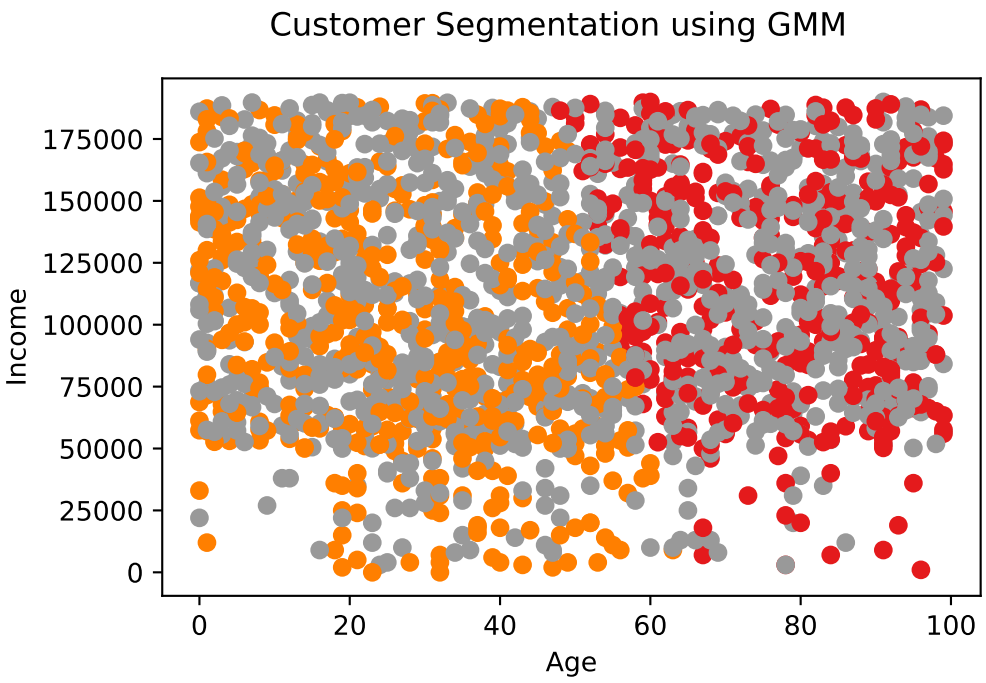
We can create a GMM by creating 3 clusters of the data and then fitting it based on the preprocessed data.

We are then predicting the labels and then assigning it to a column called `Cluster`

```
num_clusters = 3
gmm = GaussianMixture(n_components=num_clusters)
gmm.fit(customer_features)
cluster_labels = gmm.predict(customer_features)
customer_data['Cluster'] = cluster_labels
```

Finally, we plot the data on a scatterplot using `Age` and `Annual Income ($)`.

```
plt.scatter(x=customer_data["Age"], y=customer_data["Annual Income ($)"], c=cluster_labels, cmap='viridis')
plt.xlabel("Age")
plt.ylabel("Income")
plt.suptitle('Customer Segmentation using GMM')
plt.show()
```



DBSCAN

DBSCAN, or Denity-Based Spaital Clustering of Applications with Noise, is a type of clustering that is density-based. So instead of assuming that clusters are spherical like K-Means does, DBSCAN can identify clusters of arbitrary shapes. DBSCAN is good at finding arbitrary shaped clusters, identify points of noise, and handling outliers well.

How Does It Work?

DBSCAN views clusters are areas of high density separated by areas of low density.

1. DBSCAN starts by picking an arbitrary unseen point in the dataset
2. It then finds all the points connected to the starting point that are within the specified radius which is desginated as the EPS
3. If there are a ton of points in the EPS radius, a cluster is initialized. Otherwise, the point is labeled as an outlier or noise.
4. DBSCAN then iteratively grows the cluster by finding all the points connected to existing points in the cluster that are within the EPS radius
5. If the point isn't reachable by any of the clusters, then the point is marked as noise

Before running DBSCAN, we can specify a minimum number of points that to form a cluster and then the EPS radius.

Example of DBSCAN

Here we start with doing the same stuff as before, importing the relevant libraries and loading the Pokemon data

```
import pandas as pd
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
df = pd.read_csv('../../datasets/pokemon.csv')
```

We then create 2 columns `Total Attack` and `Total Defense`, which will be combinations of `attack` and `sp_attack` and `defense` and `sp_defense`

```
df['Total Attack'] = df['attack'] + df['sp_attack']
df['Total Defense'] = df['defense'] + df['sp_defense']
```

We then want to extract these 2 columns from the overall dataset.

```
# Extract features to cluster on
X = df[['Total Attack', 'Total Defense']]
```

with `StandardScaler()`, the features are standardized by removing mean and scaling to unit variable.

```
scaler = StandardScaler()
pokemon_stats_scaled = scaler.fit_transform(X)
pokemon_stats_scaled
```

```
array([[-0.65942537, -0.5835015 ],
       [-0.13433866, -0.01794378],
       [ 1.36590908,  1.93225527],
       ...,
       [ 0.91583476, -0.73951743],
       [ 1.59094624,  0.89864978],
       [ 1.42216837,  1.6787294 ]])
```

We can then proceed to create a DBSCAN object. We set the `eps` variable to 0.25, which has neighors within a 0.25 radius clustered. Additionally, `min_samples` is set to 1, which is the number of data points in a neighborhood needed for a point to be considered a core point.

```
dbscan = DBSCAN(eps=0.25, min_samples=1)
```

Afterwards, we fit the data to the DBSCAN object we just created.

```
clusters = dbscan.fit_predict(pokemon_stats_scaled)
```

Finally, we create a scatterplot that visualizes `Total Attack` and `Total Defense`.

We can see that most of the data is dense towards the area of red. Data points that aren't red are considered outliers, meaning they're Pokemon that are really weak or really strong in terms of `Total Attack` and `Total Defense`

```
plt.scatter(X['Total Attack'], X['Total Defense'], c=clusters, cmap='Set1', s=50, alpha=0.7)
plt.title('DBSCAN Clustering of Pokemon Stats')
plt.xlabel('Attack')
plt.ylabel('Defense')
plt.show()
```