Clustern

Joern Ploennigs · AI4SC

> A city, far from being a cluster of buildings, is actually a sequence of spaces enclosed and defined by buildings.
>
> — Ieoh Ming Pei

Die wichtigste Aufgabe des Unsupervised Learning ist das *Clustern*. *Clustern* ist eine Technik des maschinellen Lernens, mit der Datenpunkte in Gruppen (Cluster) mit ähnlichen Eigenschaften zusammengefasst werden. Typische Anwendungen für Clustering sind zum Beispiel die Kundensegmentierung, die Betrugserkennung und die Bildverarbeitung.

Es gibt unterschiedliche Typen von Clustering-Verfahren, die je nach Zielsetzung und Eigenschaften der Daten geeignet sind. Einige davon werden wir im Folgenden näher betrachten:

- *Partitionierende Verfahren* teilen die Daten in eine feste Anzahl von Clustern und versuchen, die Varianz innerhalb der Cluster zu minimieren.
- *Hierarchische Verfahren* bauen eine Clusterhierarchie auf und sind flexibler, benötigen jedoch mehr Berechnungsressourcen.
- *Dichtebasierte Verfahren* identifizieren Cluster basierend auf der Dichte der Datenpunkte und sind besonders gut für Daten mit Rauschen und Clustern unregelmäßiger Form geeignet.
- *Gitterbasierte Verfahren* teilen den Datenraum in eine endliche Anzahl von Zellen (Gitter) und führen das Clustering auf diesen Zellen durch.
- *Graphbasierte Verfahren* nutzen Graphstrukturen, um die Ähnlichkeiten zwischen den Datenpunkten darzustellen und das Clustering durchzuführen.

Möchtest du z.B. verschiedene Gebäudetypen anhand ihres Energieverbrauchs clustern, kannst du mit einem partitionierenden Verfahren wie k-Means eine feste Anzahl von Verbrauchsmustern identifizieren. Wenn hingegen Sensoren im Untergrund punktuell Werte melden, können dichte-

basierte Verfahren helfen, Cluster instabiler Zonen (z. B. bei Rutschgefahr) zu erkennen – auch bei unregelmäßiger Datenverteilung.

Partitionierende Clustering: k-Means

Methode

k-Means ist ein weit verbreiteter Algorithmus, um Datenpunkte in eine vordefinierte Anzahl k von Clustern aufzuteilen. Der Algorithmus ist sehr beliebt, weil er einfach zu verstehen ist und auch bei großen Datensätzen sehr schnell rechnet.

Der Algorithmus versucht, die Summe der quadratischen Abstände zwischen den Datenpunkten und ihren jeweiligen Clusterzentren zu minimieren:

$$J = \sum_{j=1}^{k} \sum_{x_i \in C_j} \| x_i - \mu_j \|^2$$

Dabei geht der Algorithmus wie folgt vor:

1. *Initialisierung*: Wähle k Initialwerte für die Clusterzentren.
2. *Zuordnung*: Weise jeden Datenpunkt x_i dem nächstgelegenen Clusterzentrum μ_j zu:

$$C_j = \left\{ x_i : \| x_i - \mu_j \|^2 \leq \| x_i - \mu_z \|^2 \quad \forall z, 1 \leq z \leq k \right\}$$

3. *Aktualisierung*: Berechne die neuen Clusterzentren als Mittelwert der zugewiesenen Punkte:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

4. *Wiederholung*: Wiederhole die Schritte 2 und 3, bis die Clusterzentren konvergieren (d.h., sie ändern sich nicht mehr signifikant).

k-Means eignet sich am besten für Cluster mit sphärischer Form. In anderen Fällen kann die Qualität der Cluster suboptimal sein. Da die zufälligen Startwerte der Clusterzentren das Ergebnis beeinflussen können, lohnt es sich, k-Means mehrfach auszuführen oder direkt robuste Initialisierungen wie k-means++ zu verwenden.

Einfaches Beispiel

Clustering wird am besten in einem 2-D-Rahmen verstanden, in dem wir die Daten und Cluster leicht visualisieren können. Das haben wir schon bei der Klassifikation mit dem Yin-Yang Datensatz gemacht. Der eignet sich für das Clustering nicht, da beide Klassen nicht klar getrennt sind und somit durch Clustering nicht zu identifizieren wären.

Deshalb erzeugen wir uns zuerst ein einfaches Beispiel mit 5 Clustern. Als ersten Schritt erstellen wir zufällig die entsprechenden Clusterzentren x_{1c} und x_{2c} aus einer Normalverteilung mit einer Standardabweichung (`scale`) von 5.

```
import numpy as np
import pandas as pd
import plotly.express as px

nc = 5  # number of clusters
np.random.seed(1)  # make the results replicable
## create cluster centers
x1c = np.random.normal(scale=5, size=nc)
x2c = np.random.normal(scale=5, size=nc)
```

Als nächstes erstellen wir 100 Datenpunkte. Wir weisen jedem Datenpunkt einem Cluster zu und platzieren ihn dann in einer zufälligen Position in der Nähe des entsprechenden Clusterzentrums. Die Cluster könnten sinnvolle IDs haben, aber hier weisen wir ihnen einfach ein numerisches Label $0, 1, \ldots, 4$ zu.

```
n = 100  # number of data points
cl = np.random.choice(nc, n)  # cluster membership label for each dot
## create clusters around centers
x1 = x1c[cl] + np.random.normal(size=n)
x2 = x2c[cl] + np.random.normal(size=n)
df = pd.DataFrame({"x1":x1, "x2":x2, "cluster":cl})
```

Die Variable `cl` ist die ID (numerisches Label) des entsprechenden Clusters, die zufällig mit `np.random.choice` ausgewählt wird. Beachten Sie, wie die Punkte (x_1, x_2) in der Nähe des entsprechenden Clusterzentrums (x_{1c}, x_{2c}) platziert sind, während zusätzliches normales Rauschen hinzugefügt wird. Die Standardabweichung dieses Rauschens beträgt 1. Daher sind die Datenpunkte im Vergleich zu den Zentren selbst deutlich enger um das jeweilige Clusterzentrum verteilt.

Es gibt fünf verschiedene Cluster, die hier je nach entsprechender Cluster-ID unterschiedlich eingefärbt sind. Der vertikale und horizontale Maßstab sind gleich, damit das menschliche Auge die Entfernung richtig einschätzen kann, die wir unten verwenden.

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Der Einsatz unüberwachter Methoden in *sklearn* ist in vielerlei Hinsicht ähnlich wie die Verwendung überwachter Methoden, mit der Ausnahme, dass das Anpassen des Modells ohne das Ziel y erfolgt. Wir importieren `KMeans` aus `sklearn.cluster` wie gewohnt. Danach richten wir das Modell ein, indem wir einfach `KMeans()` aufrufen. Das wichtigste Argument ist die Anzahl der Cluster k. Es gibt auch weitere Optionen für die Iterationen (`max_iter`) oder den genauen Algorithmus (`algorithm`). Als Nächstes erstellen wir die Designmatrix $\bar{X}$, die wir unten für die Anpassung benötigen. Danach passen wir das Modell mit `fit` an. Beachten Sie, dass wir für das

unüberwachte Modell nur die Designmatrix `X` und keine Zielvariable `y` angeben, da es sich um ein unüberwachtes Modell handelt:

```
from sklearn.cluster import KMeans

X = df[['x1','x2']]

m = KMeans(n_clusters=5)
m.fit(X)
```

	n_clusters n_clusters: int, default=8 The number of clusters to form as well as the number of centroids to generate. For an example of how to choose an optimal value for `n_clusters` refer to :ref:`sphx_glr_auto_examples_cluster_plot_kmeans_silhouette_analysis.py`.	5
	init init: {'k-means++', 'random'}, callable or array-like of shape (n_clusters, n_features), default='k-means++' Method for initialization: * 'k-means++' : selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contribution to the overall inertia. This technique speeds up convergence. The algorithm implemented is "greedy k-means++". It differs from the vanilla k-means++ by making several trials at each sampling step and choosing the best centroid	'k-means++'

	among them. * 'random': choose `n_clusters` observations (rows) at random from data for the initial centroids. * If an array is passed, it should be of shape (n_clusters, n_features) and gives the initial centers. * If a callable is passed, it should take arguments X, n_clusters and a random state and return an initialization. For an example of how to use the different `init` strategies, see :ref:`sphx_glr_auto_examples_cluster_plot_kmeans_digits.py`. For an evaluation of the impact of initialization, see the example :ref:`sphx_glr_auto_examples_cluster_plot_kmeans_stability_low_dim_dense.py`.	
	n_init n_init: 'auto' or int, default='auto' Number of times the k-means algorithm is run with different centroid seeds. The final results is the best output of `n_init` consecutive runs in terms of inertia. Several runs are recommended for sparse high-dimensional problems (see :ref:`kmeans_sparse_high_dim`). When `n_init='auto'`, the	'auto'

	number of runs depends on the value of init: 10 if using `init='random'` or `init` is a callable; 1 if using `init='k-means++'` or `init` is an array-like. .. versionadded:: 1.2 Added 'auto' option for `n_init`. .. versionchanged:: 1.4 Default value for `n_init` changed to `'auto'`.	
	max_iter max_iter: int, default=300 Maximum number of iterations of the k-means algorithm for a single run.	300
	tol tol: float, default=1e-4 Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.	0.0001
	verbose verbose: int, default=0 Verbosity mode.	0
	random_state random_state: int, RandomState instance or None, default=None Determines random number generation for centroid initia-	None

	lization. Use an int to make the randomness deterministic. See :term:`Glossary `.	
	copy_x copy_x: bool, default=True When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean. Note that if the original data is not C-contiguous, a copy will be made even if copy_x is False. If the original data is sparse, but not in CSR format, a copy will be made even if copy_x is False.	True
	algorithm algorithm: {"lloyd", "elkan"}, default="lloyd" K-means algorithm to use. The classical EM-style algorithm is `"lloyd"`. The `"elkan"` variation can be more efficient on some datasets with well-defined clusters, by using the triangle inequality.	'lloyd'

	However it's more memory intensive due to the allocation of an extra array of shape `(n_samples, n_clusters)`. .. versionchanged:: 0.18 Added Elkan algorithm .. versionchanged:: 1.1 Renamed "full" to "lloyd", and deprecated "auto" and "full". Changed "auto" to use "lloyd" instead of "elkan".	

Dies erstellt das angepasste Modell, das wir in den nächsten Schritten zur Vorhersage von Cluster-Labels mit `predict()` verwenden können.

```
df['pred'] = m.predict(X)  # predicted cluster membership for each dot
df['pred'].head()
```

```
0    3
1    2
2    4
3    4
4    1
Name: pred, dtype: int32
```

Die Methode `predict` berechnet das vorhergesagte *Cluster-Label* für jede Beobachtung (in `X`), wobei der mögliche Label-Bereich von 0 bis $k - 1$ reicht. Die identifizierte *Cluster-Zentren* können mit dem Attribut `cluster_centers_` abgefragt werden:

```
hatc = m.cluster_centers_  # centroids for each cluster
print("centers:\n", hatc)
```

```
centers:
 [[ -5.41322852   1.54188806]
 [  8.0292849  -11.62947644]
 [  4.20444648  -1.50785831]
 [ -2.64264202  -3.99768911]
 [ -3.11293253   8.42900307]]
```

Dies gibt k (hier 5) Vektoren zurück (als Zeilen der Matrix). Jeder Vektor entspricht den Schwerpunkten (Centroid) der entsprechenden Cluster, in der gleichen Reihenfolge wie die Labels. Die erste Zeile der Matrix ist also der Schwerpunkt für Cluster „0“. Da die Daten hier 2-dimensional sind, enthalten die Schwerpunkte zwei Komponenten.

Es ist ziemlich einfach, die Ergebnisse von k-means zu visualisieren:

```
Unable to display output for mime type(s): text/html
```

Wahl der Anzahl der Cluster k

Die Herausforderung beim k-Means-Clustering ist es, den Parameter k gut zu wählen. Hierbei möchte man zum einen sicherstellen, dass die Cluster gut separiert sind, aber auch nicht so groß, dass zu viele kaum separierte Untercluster entstehen. Vergleichen wir dazu einmal die Ergebnisse für 3 und 10 Cluster. Achten Sie dabei darauf, dass bei $k = 3$ mehrere natürliche Gruppen zusammengelegt werden, während bei $k = 10$ einzelne Gruppen künstlich aufgespalten werden.

```
m = KMeans(n_clusters=3).fit(X)
hatc2 = m.cluster_centers_
df['pred3'] = m.predict(X)

fig = px.scatter(df, x="x1", y="x2", color="pred3", opacity=0.5, width=600,
height=600)
fig.add_scatter(x=hatc2[:,0], y=hatc2[:,1], mode="markers", marker =
dict(symbol = 'circle-open-dot', size=20), name="center")
fig.update_coloraxes(showscale=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
m = KMeans(n_clusters=10).fit(X)
hatc2 = m.cluster_centers_
df['pred10'] = m.predict(X)

fig = px.scatter(df, x="x1", y="x2", color="pred10", opacity=0.5, width=600,
height=600)
fig.add_scatter(x=hatc2[:,0], y=hatc2[:,1], mode="markers", marker =
dict(symbol = 'circle-open-dot', size=20), name="center")
fig.update_coloraxes(showscale=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Um ein geeignetes k zu identifizieren, kann man sogenannte *Ellbogen-Diagramme* erzeugen, indem man mehrere Cluster mit unterschiedlichen k-Werten trainiert und jeweils den quadrati-

schen Fehler (`inertia_`) speichert. Wenn man diese Fehler als Liniendiagramm darstellt, ergibt sich eine abfallende Funktion, die einem Ellbogen ähnlich ist. Gute k-Werte liegen dort, wo die Kurve deutlich abflacht. In diesem Beispiel ist das nicht überraschend bei 5 der Fall.

```
Unable to display output for mime type(s): text/html
```

Das Attribut `inertia_` des angepassten Modells enthält den entsprechenden Verlustfunktionswert. Hier passen wir k-Means-Modelle für $k = 2, 3, \ldots, 9$ an, speichern den jeweiligen Verlust und plotten ihn.

Achten Sie auf den Punkt, an dem die Steigung der Kurve stark abnimmt: Dieser „Ellbogen" zeigt, dass weitere Erhöhungen von k nur noch einen geringen Zusatznutzen bringen. In diesem Beispiel liegt dieser Bereich ungefähr bei $k = 5$, was auch zu den oben visualisierten Gruppen passt.

Beispiel Baustellen

Prüfen wir den Ansatz auf einem echten Datensatz, der Liste an Baustellen in Rostock. Wir wollen die Baustellen nach ihrem Ort clustern, also nach der Latitude und Longitude.

```
baust  = pd.read_csv("../data/Baustellen/baustellen_flat.csv")
baust = baust.dropna(subset=["latitude", "longitude","sparte"])
baust.head(3)
```

	latitude	longitude … strasse_schluessel (overlapping headers)	sparte	vonnach	baubeginn	bauende	verkehrsbeeintraechtigungen	baumassnahme	dauer
0	54.0892821069429 … Ros- 13003 … db25-45af-9fba-8d63b287910 … Hanse- und Universitätsstadt … Hanse- und Universitätsstadt … 13003000100 … Budapester Str. (overlapping values)	Fernwärmeleitung	NaN	2024-03-04 07:00:00	2024-04-12 17:00:00	Sicherungsmaßnahmen entlang der Straße, Verkeh…	FW-Leitung i.a. der Hanseatic …	39.375000

	latitude	longitude	[illegible]	[illegible]	[illegible]	[illegible]	strasse_name	sparte	von	nach	baubeginn	bauende	verkehrsbeeinträchtigungen	baumassnahme	dauer
1	54.06[illegible]	12.07[illegible]	[illegible]	Rostock	Rostock, Hanse- und Universitätsstadt	Rostock, Hanse- und Universitätsstadt	Satower Str.	Grünpflege	56	65	2024-[illegible] 07:00[illegible]	2024-[illegible]	Halbseitige Sperrung, Sicherungsmaßnahmen entl…	Baumpflanzung	24.291667
2	54.17[illegible]	12.03[illegible]	[illegible] c35e-4074-8491-7d142045a8bb	Rostock	Rostock, Hanse- und Universitätsstadt	Rostock, Hanse- und Universitätsstadt	Am Markt	Wasserleitung	6	NaN	2024-[illegible] 07:00[illegible]	2024-03-28 18:00[illegible]	Sicherungsmaßnahmen entlang der Straße, Sicher…	Einbau Trinkwasserschieber	22.458333

Bilden wir als Erstes die Designmatrix und visualisieren das Ellbogendiagramm, um eine geeignete Anzahl an Clustern zu identifizieren. Dabei ist zu beachten, dass Breiten- und Längengrade keine euklidischen Meterkoordinaten sind. Für eine präzise räumliche Analyse sollte man die Daten daher vorher projizieren oder einen geographischen Distanzbegriff verwenden.

```
Xb = baust[['latitude','longitude']]
k_range = range(2, 30) # Clusteranzahl
loss = [KMeans(k).fit(Xb).inertia_ for k in k_range]

px.line(x=k_range, y=loss)
```

```
Unable to display output for mime type(s): text/html
```

Das ist ein typisches Ellbogendiagramm, bei dem das beste k nicht klar erkennbar ist. Allerdings ist der Abfall für Werte unter 10 noch deutlich, während die Kurve ab etwa 15 merklich abflacht. Daher kann man $k = 15$ hier als pragmatische Grenze wählen.

```
kmeans7 = KMeans(n_clusters=7)
baust['pred_km7']=kmeans7.fit_predict(Xb)
cc_km7 = kmeans7.cluster_centers_  # centroids for each cluster

fig = px.scatter_map(baust, lat="latitude", lon="longitude", color="pred_km7",
hover_name="sparte", zoom=9, width=600, height=600)
fig.add_scattermapbox(lat=cc_km7[:,0], lon=cc_km7[:,1], mode="markers",
name="center")
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

```
kmeans15 = KMeans(n_clusters=15)
baust['pred_km15']=kmeans15.fit_predict(Xb)
cc_km15 = kmeans15.cluster_centers_  # centroids for each cluster

fig = px.scatter_map(baust, lat="latitude", lon="longitude",
color="pred_km15", hover_name="sparte", zoom=9, width=600, height=600)
fig.add_scattermapbox(lat=cc_km15[:,0], lon=cc_km15[:,1], mode="markers",
name="center")
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

Hierarchisches Clustering

Agglomeratives Hierarchisches Clustern

Hierarchisches Clustern ist eine Methode zur Gruppierung von Datenpunkten, die eine hierarchische Struktur von Clustern erstellt. Es gibt zwei Hauptansätze: agglomerative (bottom-up) und divisive (top-down) Methoden.

Agglomeratives Hierarchisches Clustering beginnt mit jedem Datenpunkt als eigenständigem Cluster und fusioniert iterativ die nächsten Paare von Clustern, bis nur noch ein Cluster übrigbleibt. Dabei sind die Schritte des agglomerativen Clusterings:

1. *Initialisierung*: Initialisiere n Cluster, wobei jeder Cluster einen Datenpunkt enthält.
2. *Entfernungsmessung*: Berechne die Distanz zwischen allen Paaren von Clustern unter Nutzung einer der untenstehenden Abstandsmaße.
3. *Fusionierung*: Identifiziere die zwei nächsten Cluster unter Nutzung einer der untenstehenden Fusionierungsstrategien. Fasse diese beiden Cluster zu einem neuen Cluster zusammen. Aktualisiere die Distanzen zwischen den Clustern.
4. *Wiederholung*: Wiederhole Schritt 2 und 3, bis nur noch die gewünschte Anzahl von Clustern übrig ist.

Hierbei werden unterschiedliche *Abstandsmaße* verwendet:

- **Jaccard**: Die Jaccard-Distanz wird hauptsächlich für binäre oder kategorische Daten (Text) verwendet. Sie misst die Unähnlichkeit zwischen endlichen Mengen und ist besonders nützlich, wenn es um die Anwesenheit oder Abwesenheit von Merkmalen geht.

$$L_J = 1 - \frac{|A \cap B|}{|A \cup B|}$$

- **Euklidisch**: Die Euklidische Distanz ist das am häufigsten verwendete Maß für den Abstand zwischen zwei Punkten in einem n-dimensionalen Raum. Sie ist besonders nützlich, wenn die Datenpunkte metrisch sind.

$$L_E = \sqrt{\sum_{k=1}^{p} \left(x_{ik} - x_{jk}\right)^2}$$

- **Pearson**: Die Pearson-Korrelation misst die lineare Korrelation zwischen zwei Variablen und liegt zwischen −1 und 1. Als daraus abgeleitete Distanz kann man zum Beispiel $1 - r$ verwenden, wobei r die Pearson-Korrelation ist.

$$L_P = 1 - r\left(x_i, x_j\right)$$

- **Manhattan**: Die Manhattan-Distanz, auch City Block Distanz genannt, summiert die absoluten Unterschiede der Koordinaten zweier Punkte. Sie ist besonders nützlich, wenn die Daten eine natürliche Gitterstruktur haben.

$$L_M = \sum_{k=1}^{p} \left|x_{ik} - x_{jk}\right|$$

und unterschiedliche Fusionierungsstrategien benutzt:

- **Single Linkage** misst die minimale Distanz zwischen Punkten in zwei Clustern. Es neigt dazu, längliche und kettenartige Cluster zu erstellen.

$$D_{\text{sl}}(A,B) := \min_{a\in A, b\in B}\{d(a,b)\}$$

- **Complete Linkage** misst die maximale Distanz zwischen Punkten in zwei Clustern. Es erzeugt kompakte und sphärische Cluster.

$$D_{\text{cl}}(A,B) := \max_{a\in A, b\in B}\{d(a,b)\}$$

- **Average Linkage** misst die durchschnittliche Distanz zwischen Punkten in zwei Clustern. Es balanciert zwischen den Extremen von Single und Complete Linkage und ist nützlich, wenn Cluster ähnliche Varianz haben.

$$D_{\text{al}}(A,B) := \frac{1}{|A||B|}\sum_{a\in A, b\in B} d(a,b)$$

- **Centroid-Linkage** misst die Distanz zwischen den Schwerpunkten (Centroide) der Cluster. Es ist nützlich, wenn der Schwerpunkt eines Clusters von Interesse ist. Es kann aber auch zu Problemen führen, wenn Schwerpunkte sich in leere Räume bewegen (Inversionsproblem).

$$D_{\text{cd}}(A,B) := d\left(\bar{a},\bar{b}\right)$$

- **Ward's Method** minimiert die Varianz innerhalb der Cluster. Es ist eine der besten Methoden und fördert die Bildung von kompakten, kugelförmigen Clustern.

$$D_{\text{wd}}(A,B) := \frac{d\left(\bar{a},\bar{b}\right)^2}{1/|A| + 1/|B|}$$

Die Verwendung der Modelle ist wie gehabt.

```
from sklearn.cluster import AgglomerativeClustering

mAHC = AgglomerativeClustering(n_clusters = 5)
df["pred_AHC5"] = mAHC.fit_predict(X)
```

Es gibt jedoch einen deutlichen Unterschied: Das Modell `AgglomerativeClustering` hat *keine* Methode `.predict`. Man sollte stattdessen `.fit_predict` verwenden, da dies sowohl das Anpassen als auch das Vorhersagen in einem Schritt durchführt.

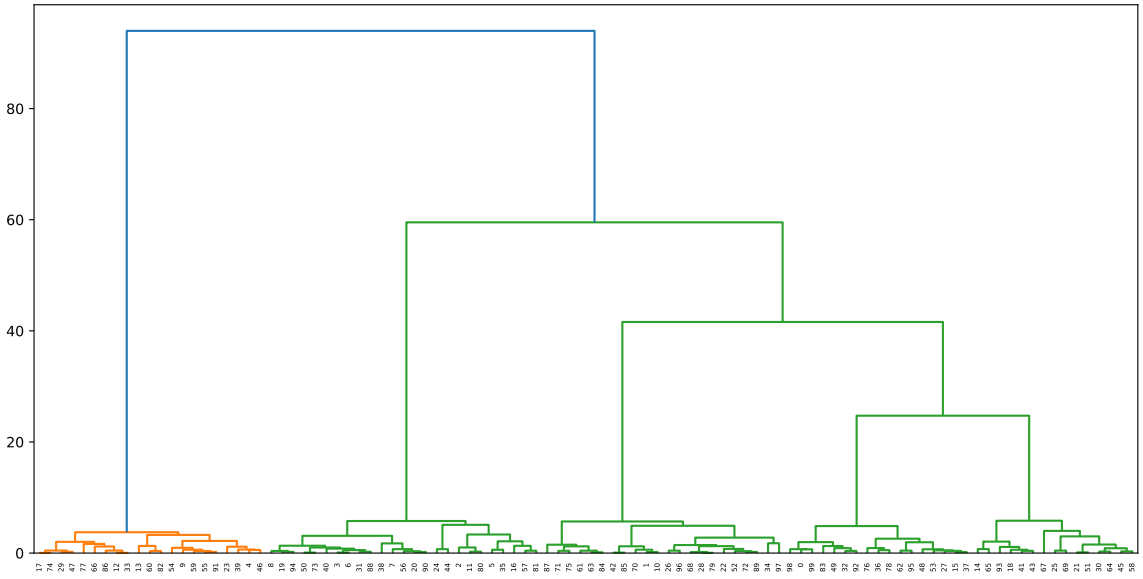
```
fig=px.scatter(df, x="x1", y="x2", color="pred_AHC5", opacity=0.5, width=600,
height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

Basierend auf diesen Daten sind die hierarchischen Cluster genau die gleichen wie beim k-means (nur ihre Labels können sich unterscheiden). Das liegt daran, dass die Daten so sauber in

verschiedene Cluster aufgeteilt sind, und daher alle Clustering-Algorithmen die gleiche Struktur erkennen werden.

Allerdings ermöglicht hierarchisches Clustering die Anzeige von Dendrogrammen. Dies ist eine Baumdarstellung des hierarchischen Clusterings. Wichtig ist dabei, dass die Höhe der Äste zwischen Knoten die Distanz beider Cluster anzeigt. Je länger der Ast, desto größer ist die Distanz zwischen den vereinigten Clustern. Dadurch lässt sich auch gut eine passende Anzahl n an Clustern abschätzen.



Versuchen wir den Clusteringansatz auf dem Baustellendatensatz:

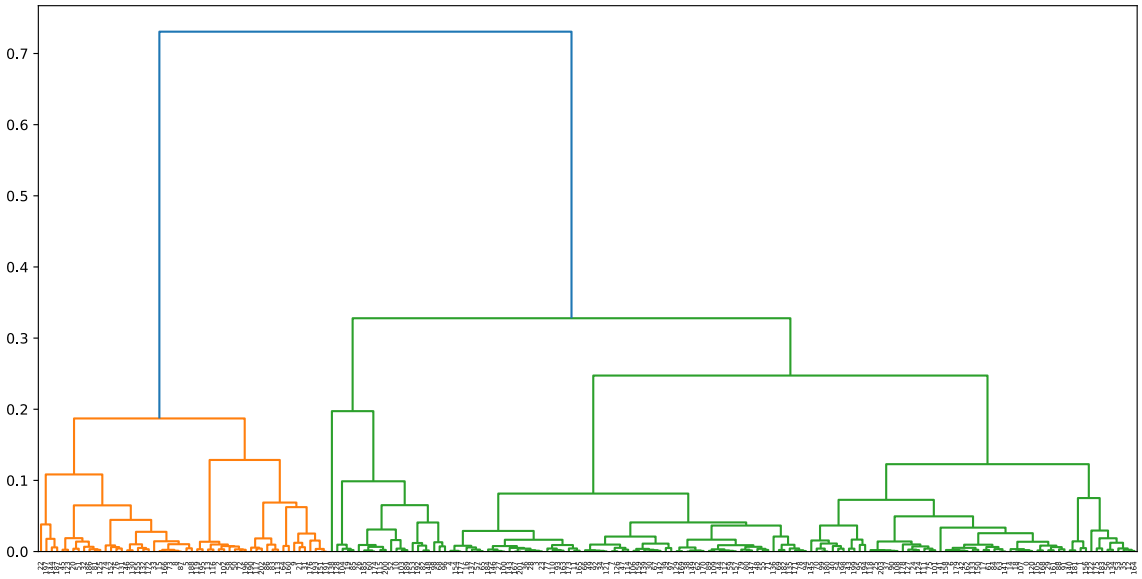
```
mbAHC = AgglomerativeClustering(n_clusters = 15)
baust["pred_AHC15"] = mbAHC.fit_predict(Xb)

fig = px.scatter_map(baust, lat="latitude", lon="longitude",
color="pred_AHC15", hover_name="sparte", zoom=9, width=600, height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

```
mAHC = AgglomerativeClustering(n_clusters = 5)
baust["pred_AHC5"] = mAHC.fit_predict(Xb)

fig = px.scatter_map(baust, lat="latitude", lon="longitude",
color="pred_AHC5", hover_name="sparte", zoom=9, width=600, height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

```
plt.figure(figsize=(14, 7))
dendrogram = sch.dendrogram((sch.linkage(Xb, method='ward')))
plt.show()
```



Divisives Clustering

Der Algorithmus für divisives Clustering kann wie folgt beschrieben werden:

1. **Initialisierung**: Initialisiere einen einzelnen Cluster, der alle Datenpunkte enthält.
2. **Evaluierung**: Bewerte, ob der aktuelle Cluster weiter aufgeteilt werden soll. Dies kann z. B. durch ein statistisches Kriterium erfolgen, z. B. anhand der Intra-Cluster-Varianz.
3. **Aufspaltung**: Wenn eine Aufspaltung erforderlich ist, teile den Cluster in zwei neue Cluster. Die Aufteilung kann z. B. durch eine K-Means-Initialisierung erfolgen.
4. **Wiederholung**: Wiederhole Schritt 2 und 3, bis die gewünschte Anzahl von Clustern erreicht ist.

Das Problem des Ansatzes ist, dass divisive Verfahren eine Heuristik brauchen, mit der die Datensätze gespalten werden. Dabei werden häufig ähnliche Kriterien wie beim agglomerativen Clustering genutzt. Da der top-down-Ansatz meist aufwendiger ist und in vielen Fällen keine klaren Vorteile bietet, findet er in der Praxis weniger Anwendung und ist in `sklearn` auch nicht direkt implementiert.

Dichtebasiertes Clustering

DBSCAN - Density-Based Spatial Clustering of Applications with Noise

DBSCAN ist ein Algorithmus für das dichtebasierte räumliche Clustering, der Datenpunkte in Cluster gruppiert, basierend auf ihrer Dichte im Raum. Im Gegensatz zu anderen Clustering-Algorithmen, bei denen die Anzahl der Cluster im Voraus festgelegt werden muss, kann DBSCAN automatisch die Anzahl und Form der Cluster in den Daten erkennen.

DBSCAN basiert auf der Einteilung des Datensatzes in Kernpunkte, Randpunkte und Rauschen. *Kernpunkte* bilden die zentralen Punkte der Cluster. Es sind Datenpunkte x_i mit einer hohen Dichte und mit mindestens MinPts Nachbarn in einem Radius ϵ. Punkte in der Nachbarschaft eines Kernpunktes können entweder selbst wieder Kernpunkte oder *Randpunkte* sein. Randpunkte gehören zu einem Cluster, erfüllen aber selbst das Kernpunkt-Kriterium nicht. Alle anderen Datenpunkte haben eine geringe Dichte und werden als *Rauschen* bezeichnet.

Der Algorithmus durchläuft die Datenpunkte und identifiziert die Kernpunkte und Randpunkte wie folgt:

1. Initialisierung: Markiere alle Datenpunkte als nicht klassifiziert.
2. Distanzberechnung: Berechne die Distanz zwischen allen Punkten (z.B. Euklidisch).
3. Für jeden unbesuchten Punkt x_i:
   a. Markiere x_i als besucht.
   b. Bestimme alle Nachbarn im Radius ϵ via $N_\epsilon(x_i) = \{x_j \in X \mid d(x_i, x_j) \leq \epsilon\}$.
   c. Wenn $|N_\epsilon(x_i)| < \text{MinPts}$: Markiere x_i als Rauschen.
   d. Sonst: Erstelle einen neuen Cluster C_{x_i} und füge x_i als Kernpunkt hinzu. Erweitere den Cluster um alle Nachbarn x_j in $N_\epsilon(x_i)$. Punkte, die selbst das Kernpunkt-Kriterium nicht erfüllen, werden dabei als Randpunkte behandelt.

Die Anwendung von DBSCAN auf unseren Testdatensatz läuft analog zu den bisherigen Beispielen. Hier kann man die Cluster-Labels über das Attribut `labels_` auslesen.

```
from sklearn.cluster import DBSCAN

mDB = DBSCAN(eps=3, min_samples=2).fit(X)
df['pred_DBS'] = mDB.labels_

fig = px.scatter(df, x="x1", y="x2", color="pred_DBS", opacity=0.5, width=600,
height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

Der Ansatz erkennt alle fünf Cluster gut, was nicht überrascht, da es klare Dichteunterschiede gibt. Beim Baustellendatensatz sehen wir allerdings Unterschiede. Wo die anderen Clusteransätze die Baustelle in Graal-Müritz als Ausreißer in einem separaten Cluster gepackt haben, landet diese bei DBSCAN im Label `-1`. Dieses Label steht für Rauschen bzw. Ausreißer und zählt nicht als eigener Cluster. Der Ansatz ist somit weniger anfällig für solche Ausreißer.

```
mbDB = DBSCAN(eps=0.01, min_samples=4).fit(Xb)
baust['pred_DBS'] = mbDB.labels_

fig = px.scatter_map(baust, lat="latitude", lon="longitude", color="pred_DBS",
hover_name="sparte", zoom=9, width=600, height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

OPTICS - Ordering Points To Identify the Clustering Structure

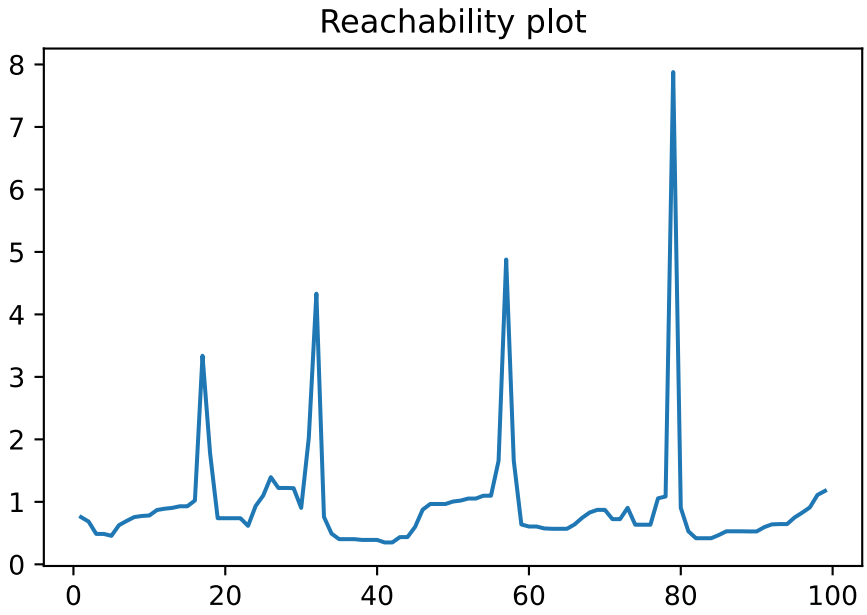
Obwohl man bei DBSCAN keine Clusteranzahl angeben muss, ist die Wahl von ϵ und minPts entscheidend für die Qualität der Cluster und häufig schwierig, da sie unintuitiver sind als die Clusteranzahl k. Dies versucht der OPTICS Algorithmus zu verbessern, in dem verschieden dichte Cluster erkannt werden indem eine hierarchische Darstellung (Erreichbarkeitsdiagramm/ Reachability Plot) der Clusterstruktur erzeugt wird. Dadurch können Cluster auf verschiedenen Granularitätsebenen extrahiert werden.

```
from sklearn.cluster import OPTICS

mOPT = OPTICS().fit(X)
df['pred_OPT'] = mOPT.labels_

fig = px.scatter(df, x="x1", y="x2", color="pred_OPT", opacity=0.5, width=600,
height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

OPTICS sortiert intern die Cluster in ihrer Dichte, bzw. der Distanz in der Dichte. Das kann in Form eines *Erreichbarkeitsdiagramm* (*Reachability Plot*) dargestellt werden, welche sich ähnlich wie Dendrogramme gut eignen die Trennbarkeit von Datensätzen zu untersuchen. In dem Plot werden Cluster als Täler repräsentiert. Ein tiefes *Tal* bedeutet, dass die Punkte in diesem Bereich eng beieinander liegen, was auf eine höhere Dichte hinweist. *Spitzen* zwischen den Tälern deuten auf die Trennung zwischen Clustern hin. Hohe Spitzen bedeuten, dass es einen größeren Abstand gibt, um von einem Cluster zum nächsten zu gelangen, was auf eine geringe Dichte oder Rauschen hinweist. Punkte, die isoliert sind, können als Rauschen betrachtet werden.

```
# Generate reachability plot
reachability = mOPT.reachability_[mOPT.ordering_]
plt.plot(reachability)
plt.title('Reachability plot')
plt.show()
```



Hier erkennt man gut die fünf Cluster, die gut separiert sind.

Beim Baustellendatensatz ergeben sich andere Cluster als bei DBSCAN. Die Behandlung von Ausreißern, wie die Baustelle in Graal-Müritz ist jedoch gleich.

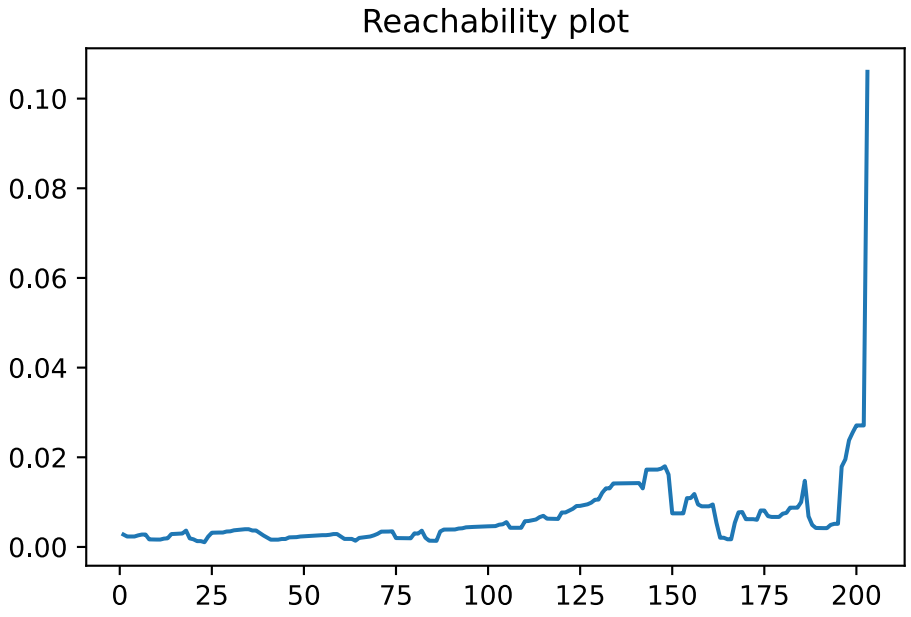
```
mbOPT = OPTICS().fit(Xb)
baust['pred_OPT'] = mbOPT.labels_

fig = px.scatter_map(baust, lat="latitude", lon="longitude", color="pred_OPT",
hover_name="sparte", zoom=9, width=600, height=600)
fig.update_coloraxes(showscale=False).show()
```

```
Unable to display output for mime type(s): text/html
```

Das Erreichbarkeitsdiagramm ist in diesem Fall auch wesentlich uneindeutiger, was auf eine schlechtere Trennbarkeit der Cluster hindeutet.

```
# Generate reachability plot
reachability = mbOPT.reachability_[mbOPT.ordering_]
plt.plot(reachability)
```

```
plt.title('Reachability plot')
plt.show()
```



Referenzen

Bibliographie