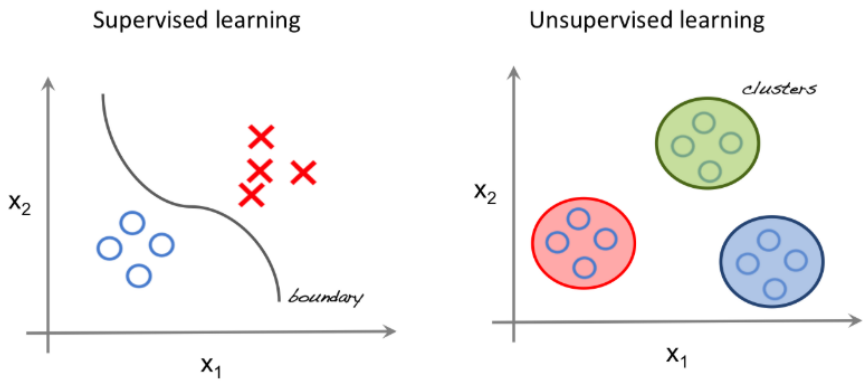
Python и кластерный анализ

1. Обучение без учителя: 3 метода кластеризации данных на Python

Обучение без учителя (unsupervised learning, неконтролируемое обучение) – класс методов машинного обучения для поиска шаблонов в наборе данных. Данные, получаемые на вход таких алгоритмов, обычно не размечены, то есть передаются только входные переменные X без соответствующих меток y. Если в контролируемом обучении (обучении с учителем, supervised learning) система пытается извлечь уроки из предыдущих примеров, то в обучении без учителя – система старается самостоятельно найти шаблоны непосредственно из приведенного примера.



На левой части изображения представлен пример контролируемого обучения: здесь для того, чтобы найти лучшую функцию, соответствующую представленным точкам, используется метод регрессии. В то же время при неконтролируемом обучении входные данные разделяются на основе представленных характеристик, а предсказание свойств основывается на том, какому кластеру принадлежит пример.

Кластеризация (clustering) является задачей разбиения набора данных на группы, называемые кластерами.

Цель – разделить данные таким образом, чтобы точки, находящие в одном и том же кластере, были очень схожи друг с другом, а точки, находящиеся в разных кластерах, отличались друг от друга. Как и алгоритмы

классификации, алгоритмы кластеризации присваивают (или прогнозируют) каждой точке данных номер кластера, которому она принадлежит.

2. Метод *k*-средних

Метод *k*-средних – один из самых простых и наиболее часто используемых алгоритмов кластеризации. Это итеративный алгоритм кластеризации, основанный на минимизации суммарных квадратичных отклонений точек кластеров от центроидов (средних координат) этих кластеров.

Сначала выбирается число кластеров *k*. После выбора значения *k* алгоритм *k*-средних случайным образом отбирает точки, которые будут представлять центры кластеров (cluster centers). Затем для каждой точки данных вычисляется его евклидово расстояние до каждого центра кластера. Каждая точка назначается ближайшему центру кластера. Алгоритм вычисляет центроиды (centroids) – центры тяжести кластеров. Каждый центроид – это вектор, элементы которого представляют собой средние значения характеристик, вычисленные по всем точкам кластера. Центр кластера смещается в его центроид. Точки заново назначаются ближайшему центру кластера. Этапы изменения центров кластеров и переназначения точек итеративно повторяются до тех пор, пока границы кластеров и расположение центроидов не перестанут изменяться, т.е. на каждой итерации в каждый кластер будут попадать одни и те же точки данных. Рисунок 1 иллюстрирует работу алгоритма на синтетическом наборе данных.

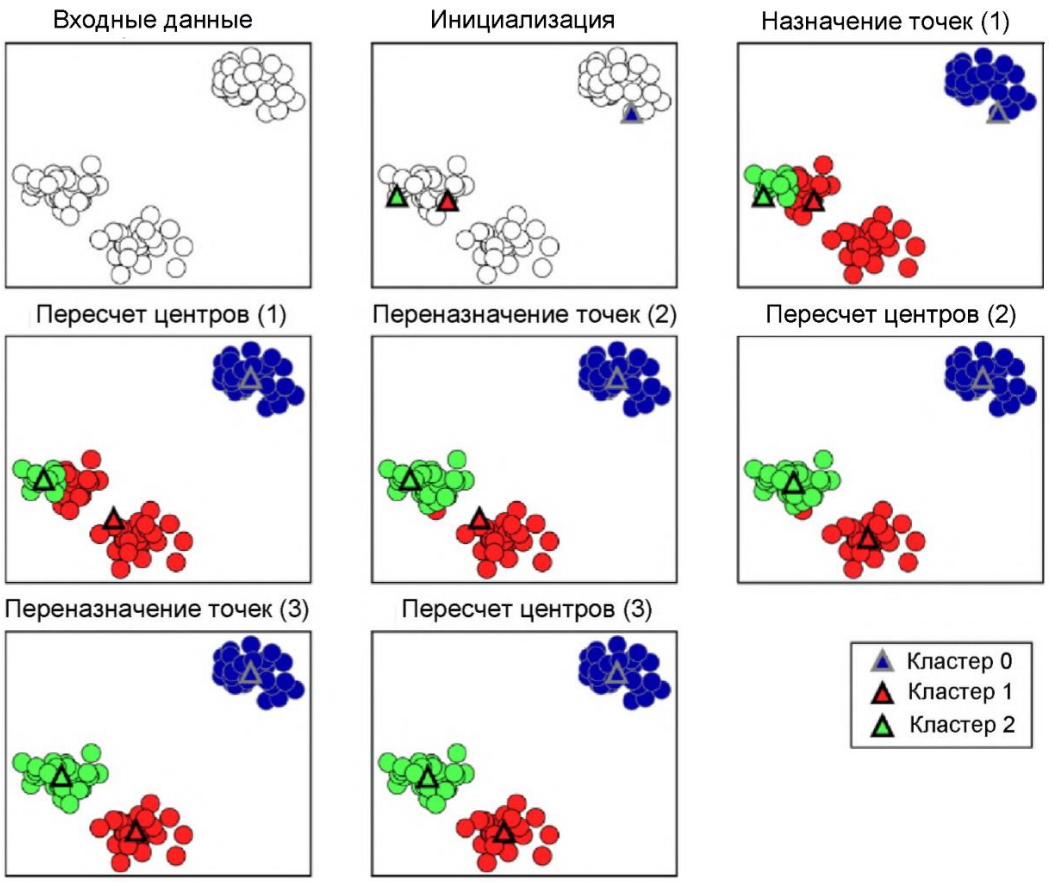


Рис. 1. Исходные данные и этапы алгоритма *k*-средних

Центры кластеров представлены в виде треугольников, в то время как точки данных отображаются в виде окружностей. Цвета указывают принадлежность к кластеру. Было указано, что ищем три кластера, поэтому алгоритм был инициализирован с помощью случайного выбора трех точек данных в качестве центров кластеров (см. «**Инициализация**»). Затем запускается итерационный алгоритм. Во-первых, каждая точка данных назначается ближайшему центру кластера (см. «**Назначение точек** (1)»). Затем центры кластеров переносятся в центры тяжести кластеров (см. «**Пересчет центров** (1)»). Затем процесс повторяется еще два раза. После третьей итерации принадлежность точек кластерным центрам не изменилась, поэтому алгоритм останавливается. Получив новые точки данных, алгоритм *k*-средних будет присваивать каждую точку данных ближайшему центру кластера. На рисунке 2 показаны границы центров кластеров, процесс вычисления которых был приведен на рисунке 1.

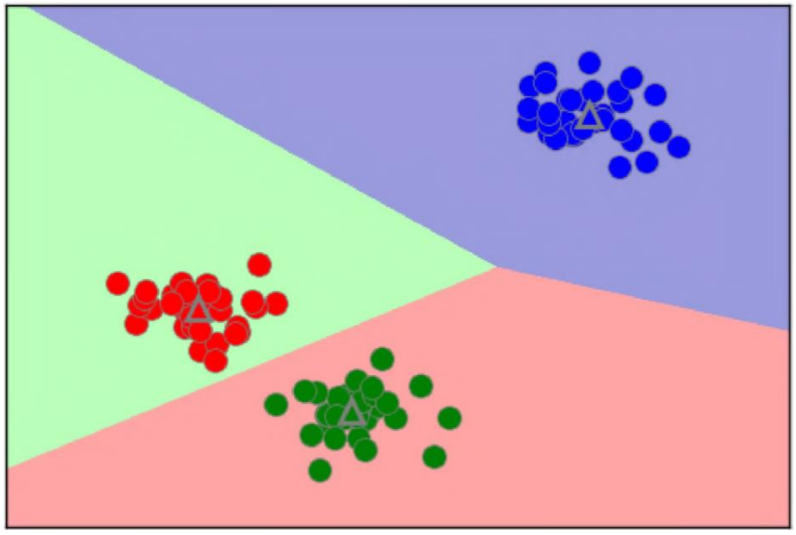


Рис. 2. Центры и границы кластеров, найденные с помощью алгоритма *k*-средних

Применим алгоритм *k*-средних к синтетическим данным, которые использовали для построения предыдущих графиков, воспользовавшись библиотекой `scikit-learn`:

– создаем экземпляр класса `KMeans`;

– задаем количество выделяемых кластеров (если не задать количество выделяемых кластеров, то значение `n_clusters` по умолчанию будет равно 8);

– вызываем метод `fit` и передаем ему в качестве аргумента данные:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
# генерируем синтетические двумерные данные
X, y = make_blobs(random_state=1)
# строим модель кластеризации
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
print("Принадлежность к кластерам:\n{}".format(kmeans.labels_))
```

```
Принадлежность к кластерам:
[1 0 0 0 2 2 2 0 1 1 0 0 2 1 2 2 2 1 0 0 2 0 2 1 0 2 2 1 1 2 1 1 2 1 0 2
0
 0 0 2 2 0 1 0 0 2 1 1 1 1 0 2 2 2 1 2 0 0 1 1 0 2 2 0 0 2 1 2 1 0 0 0 2
1
 1 0 2 2 1 0 1 0 0 2 1 1 1 1 0 1 2 1 1 0 0 2 2 1 2 1]
```

Во время работы алгоритма каждой точке обучающих данных `X` присваивается метка кластера. Найти эти метки можно в атрибуте `kmeans.labels_`.

Поскольку мы задали три кластера, кластеры пронумерованы от 0 до 2.

Кроме того, вы можете присвоить метки кластеров новым точкам с помощью метода `predict`. В ходе прогнозирования каждая новая точка назначается ближайшему центру кластера, но существующая модель не меняется. Запуск метода `predict` на обучающем наборе возвращает тот же самый результат, что содержится в атрибуте `labels_`:

```
print(kmeans.predict(X))
```

```
[1 0 0 0 2 2 2 0 1 1 0 0 2 1 2 2 2 1 0 0 2 0 2 1 0 2 2 1 1 2 1 1 2 1 0 2
0
 0 0 2 2 0 1 0 0 2 1 1 1 1 0 2 2 2 1 2 0 0 1 1 0 2 2 0 0 2 1 2 1 0 0 0 2
1
 1 0 2 2 1 0 1 0 0 2 1 1 1 1 0 1 2 1 1 0 0 2 2 1 2 1]
```

Можно заметить, что *кластеризация* немного похожа на *классификацию* в том плане, что каждый элемент получает метку. Однако нет никаких оснований утверждать, что данная метка является истинной и поэтому сами по себе метки не несут никакого априорного смысла.

В случае с кластеризацией, которую мы только что построили для двумерного синтетического набора данных, это означает, что мы не должны придавать значения тому факту, что одной группе был присвоен 0, а другой – 1. Повторный запуск алгоритма может привести к совершенно иной нумерации кластеров в силу случайного характера инициализации. Ниже приводится новый график для тех же самых данных (рис. 3). Центры кластеров записаны в атрибуте `cluster_centers_` и мы нанесли их на график в виде треугольников.

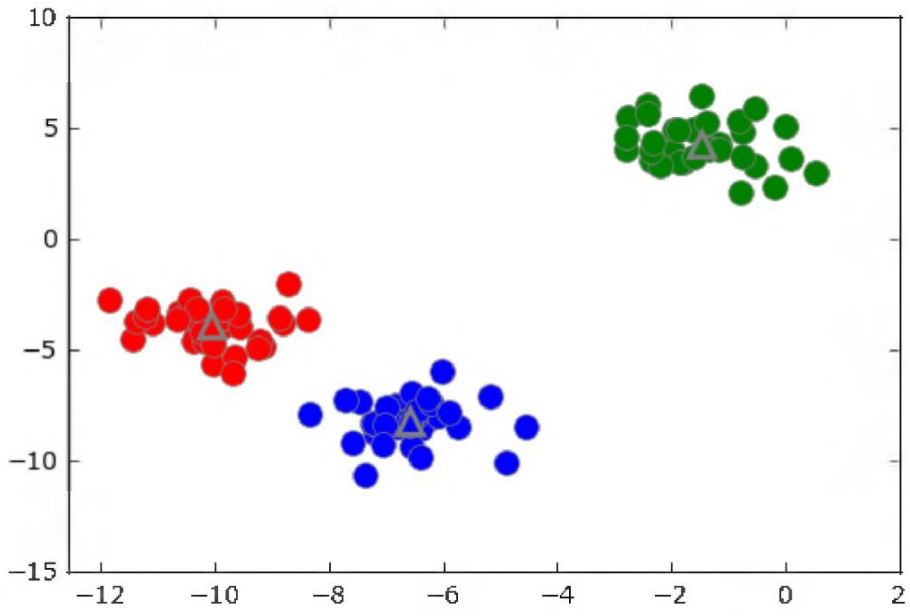


Рис. 3. Принадлежность к кластерам и центры кластеров, найденные с помощью алгоритма *k*-средних, *k*=3

Кроме того, мы можем увеличить или уменьшить количество центров кластеров (рис. 4):

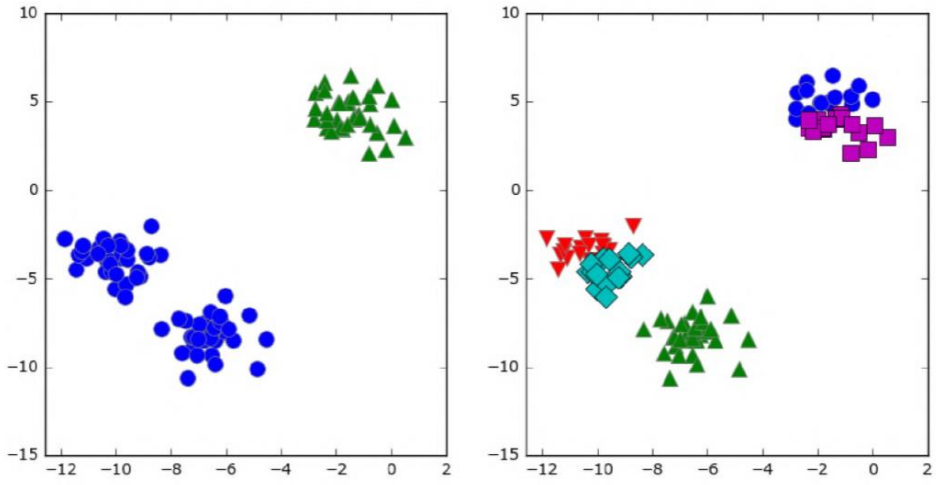


Рис. 4. Принадлежность к кластерам, найденная с помощью алгоритма *k*-средних, *k*=3 (слева) и *k*=5 (справа)

Недостатки алгоритма *k*-средних

Даже если вы знаете «правильное» количество кластеров для конкретного набора данных, алгоритм *k*-средних не всегда может выделить их. Каждый кластер определяется исключительно его центром, это означает, что каждый кластер имеет выпуклую форму. В результате этого алгоритм *k*-

средних может описать относительно простые формы. Кроме того, алгоритм *k*-средних предполагает, что все кластеры в определенном смысле имеют одинаковый «диаметр», он всегда проводит границу между кластерами так, чтобы она проходила точно посередине между центрами кластеров. Это иногда может привести к неожиданным результатам, как показано на рис. 5:

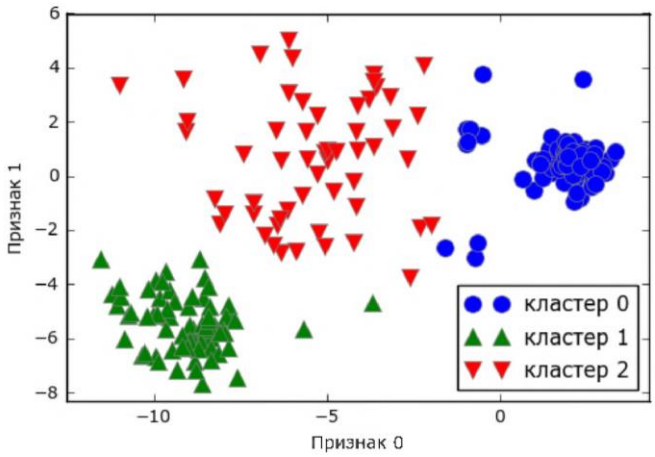


Рис. 5. Принадлежность к кластерам, найденная с помощью алгоритма *k*-средних, при этом кластеры имеют разные плотности

Можно было бы ожидать плотную область в нижнем левом углу, которая рассматривалась бы в качестве первого кластера, плотную область в верхнем правом углу в качестве второго кластера и менее плотную область в центре в качестве третьего кластера. Вместо этого, у кластера 0 и кластера 1 есть несколько точек, которые сильно удалены от всех остальных точек этих кластеров, «тянущихся» к центру. Кроме того, алгоритм *k*-средних предполагает, что все направления одинаково важны для каждого кластера. Следующий график (рис. 6) показывает двумерный набор данных с тремя четко обособленными группами данных. Однако эти группы вытянуты по диагонали. Поскольку алгоритм *k*-средних учитывает лишь расстояние до ближайшего центра кластера, он не может обработать данные такого рода:

```
# генерируем случайным образом данные для кластеризации
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)
# преобразуем данные так, чтобы они были вытянуты по диагонали
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)
# группируем данные в три кластера
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
# строим график принадлежности к кластерам и центров кластеров
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.scatter(kmeans.cluster_centers_[:, 0],
            kmeans.cluster_centers_[:, 1], marker='^',
            c=[0, 1, 2], s=100, linewidth=2)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

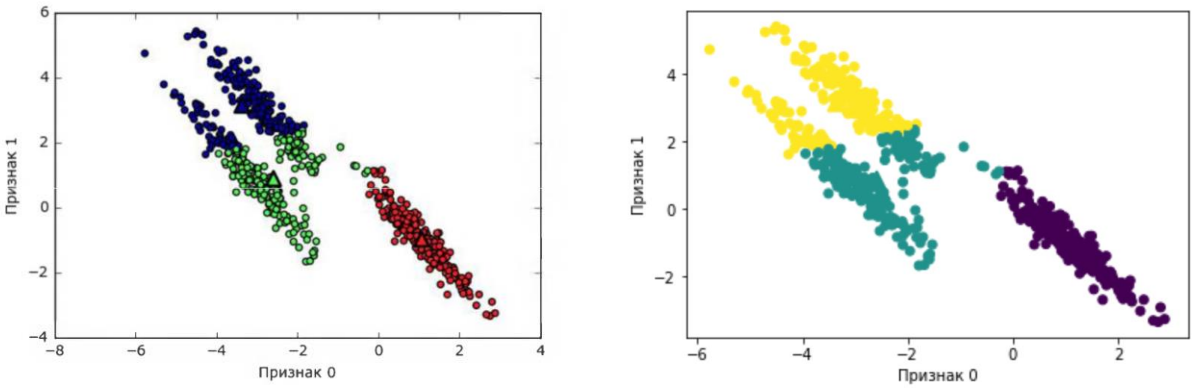


Рис. 6. Алгоритм *k*-средних не позволяет выявить несферические кластеры

Кроме того, алгоритм *k*-средних плохо работает, когда кластеры имеют более сложную форму, как в случае со следующими данными (`two_moons`):

```
# генерируем синтетические данные two_moons (на этот раз с меньшим количест
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state= 0)
# группируем данные в два кластера
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)
# строим график принадлежности к кластерам и центров кластеров
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', s=100, linewidth=2)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

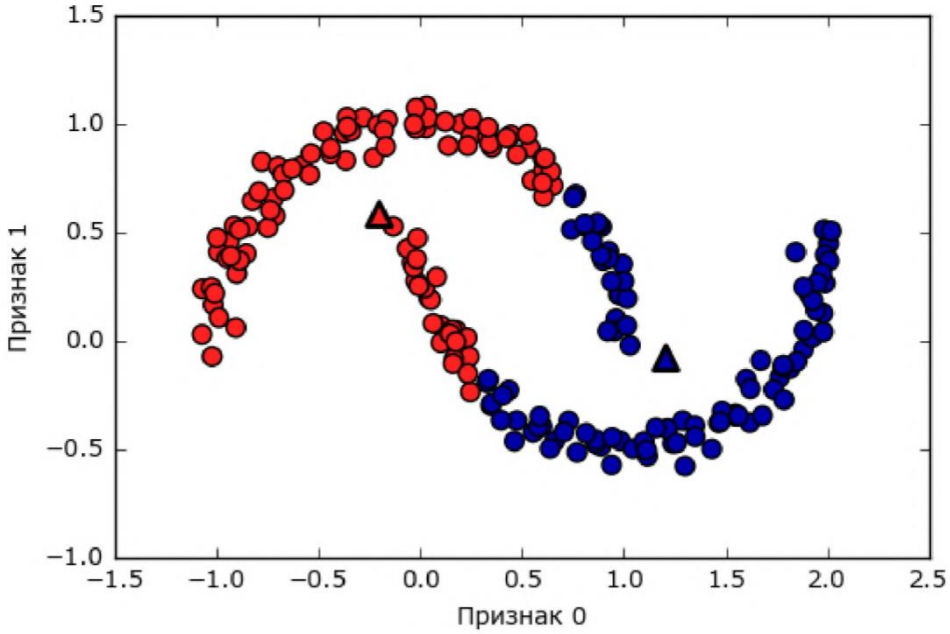


Рис. 7. Алгоритм *k*-средних не позволяет выявить кластеры более сложной формы

В данном случае мы понадеялись на то, что алгоритм кластеризации сможет обнаружить два кластера в форме полумесяцев. Однако определить их с помощью алгоритма *k*-средних не представляется возможным.

3. Иерархическая кластеризация и дендрограммы

Иерархическая кластеризация представляет собой алгоритм, который строит иерархию кластеров. Алгоритм начинает работу с того, что каждому экземпляру данных сопоставляется свой собственный кластер. Затем два ближайших кластера объединяются в один и так далее, пока не будет образован один общий кластер.

Инструмент для визуализации результатов иерархической кластеризации называется *дендрограммой* (dendrogram). Он позволяет обрабатывать многомерные массивы данных. К сожалению, на данный момент в `scikit-learn` нет инструментов, позволяющих рисовать дендрограммы. Однако их легко можете создать их с помощью `SciPy`. По сравнению с алгоритмами кластеризации `scikit-learn` алгоритмы кластеризации `SciPy` имеют немного другой интерфейс. В `SciPy`

используется функция, которая принимает массив данных `X` в качестве аргумента и вычисляет массив связей (`linkage array`) с записанными сходствами между кластерами. Затем с помощью функции `SciPy dendrogram` по данному массиву можем построить дендрограмму (рис. 8):

```
# импортируем функцию dendrogram и функцию кластеризации ward из SciPy
from scipy.cluster.hierarchy import dendrogram, ward
X, y = make_blobs(random_state=0, n_samples=12)
# применяем кластеризацию ward к массиву данных X
# функция SciPy ward возвращает массив с расстояниями
# вычисленными в ходе выполнения агломеративной кластеризации
linkage_array = ward(X)
# теперь строим дендрограмму для массива связей, содержащего расстояния
# между кластерами
dendrogram(linkage_array)
# делаем отметки на дереве, соответствующие двум или трем кластерам
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')
ax.text(bounds[1], 7.25, 'два кластера', va='center', fontdict={'size': 15}
ax.text(bounds[1], 4, 'три кластера', va='center', fontdict={'size': 15})
plt.xlabel("Индекс наблюдения")
plt.ylabel("Кластерное расстояние")
```

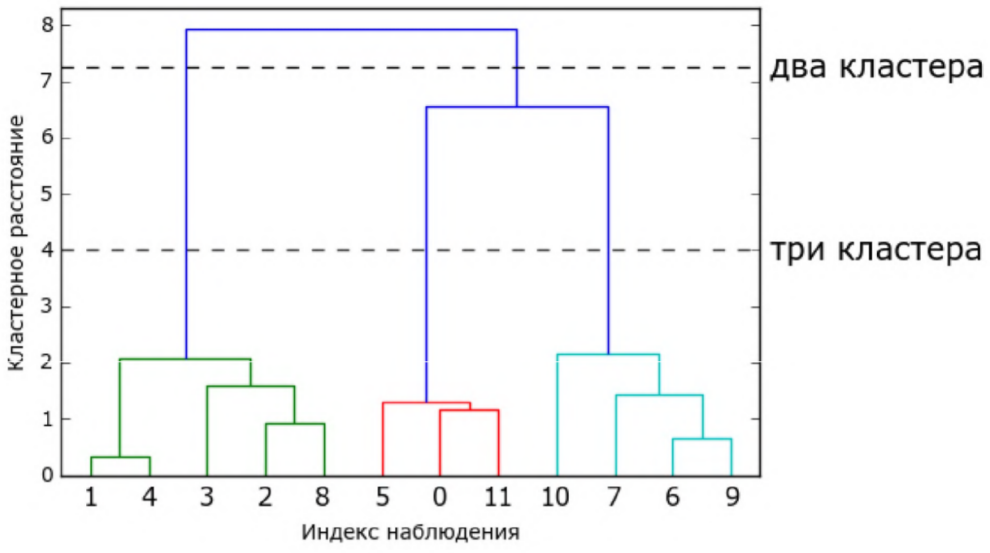


Рис. 8. Дендрограмма для кластеризации: линии обозначают расщепления на два и три кластера

4. Алгоритм кластеризации DBSCAN

DBSCAN – отличный инструмент для кластеризации данных, в которых встречаются сгустки произвольной формы (рис. 9).

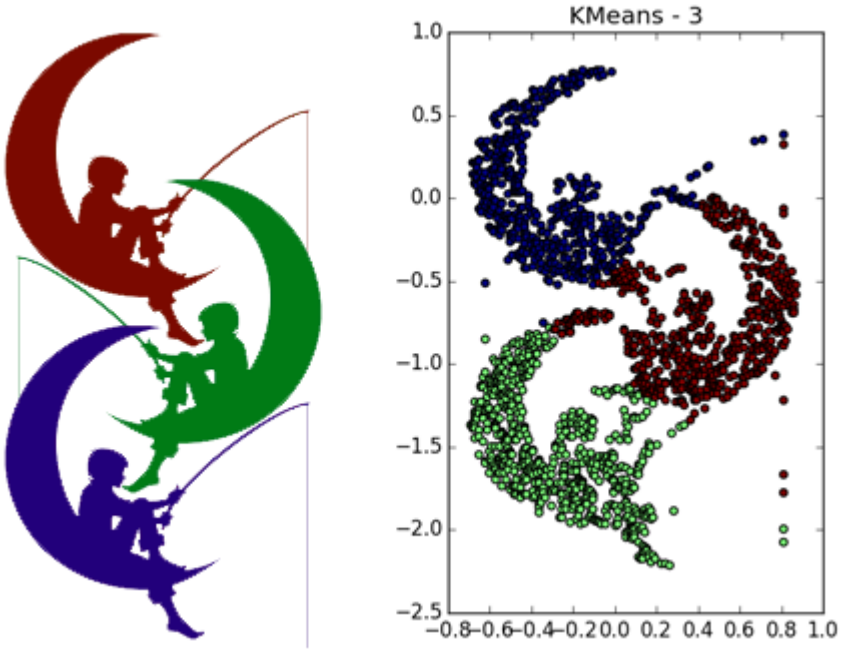


Рис. 9

DBSCAN (Density-based spatial clustering of applications with noise, плотностной алгоритм пространственной кластеризации с присутствием шума), как следует из названия, оперирует плотностью данных. На вход он просит матрицу близости и два параметра – радиус ε-окрестности и количество соседей.

Рассмотрим **пример**. В огромном зале толпа людей справляет чей-то день рождения. Кто-то слоняется один, но большинство – с товарищами. Некоторые компании просто толпятся гурьбой, некоторые – водят хороводы или танцуют ламбаду. Мы хотим разбить людей в зале на группы. Но как выделить группы столь разной формы, да ещё и не забыть про одиночек? Попробуем оценить плотность толпы вокруг каждого человека. Наверное, если плотность толпы между двумя людьми выше определённого порога, то они принадлежат одной компании.

Будем говорить, что рядом с некоторым человеком собралась толпа, если близко к нему стоят несколько других человек. Отсюда следует, что нужно задать два параметра:

– параметр, характеризующий понятие «близко». Например, если люди могут дотронуться до голов друг друга, то они находятся близко (пусть, на расстоянии метра);

– параметр, характеризующий понятие «несколько других человек». Допустим, три человека.

Пусть каждый подсчитает, сколько человек стоят в радиусе метра от него. Все, у кого есть хотя бы три соседа, берут в руки зелёные флажки. Теперь они коренные элементы, именно они формируют группы.

Обратимся к людям, у которых меньше трёх соседей. Выберем тех, у которых, по крайней мере, один соседей держит зелёный флаг, и вручим им жёлтые флаги. Скажем, что они находятся на границе групп.

Остались одиночки, у которых мало того что нет трёх соседей, так ещё и ни один из них не держит зелёный флаг. Раздадим им красные флаги. Будем считать, что они не принадлежат ни одной группе.

Таким образом, если от одного человека до другого можно создать цепочку «зелёных» людей, то эти два человека принадлежат одной группе. Очевидно, что все подобные скопища разделены либо пустым пространством, либо людьми с жёлтыми флагами. Можно их пронумеровать: каждый в группе №1 может достичь по цепочке рук каждого другого в группе №1, но никого в №2, №3 и так далее. То же для остальных групп.

Если рядом с человеком с жёлтым флажком есть только один «зелёный» сосед, то он будет принадлежать той группе, к которой принадлежит его сосед. Если таких соседей несколько, и у них разные группы, то придётся выбирать. Тут можно воспользоваться разными методами – посмотреть, кто из соседей ближайший, например. Придётся как-то обходить краевые случаи.

Иллюстрация примера приведена на рисунке 10.

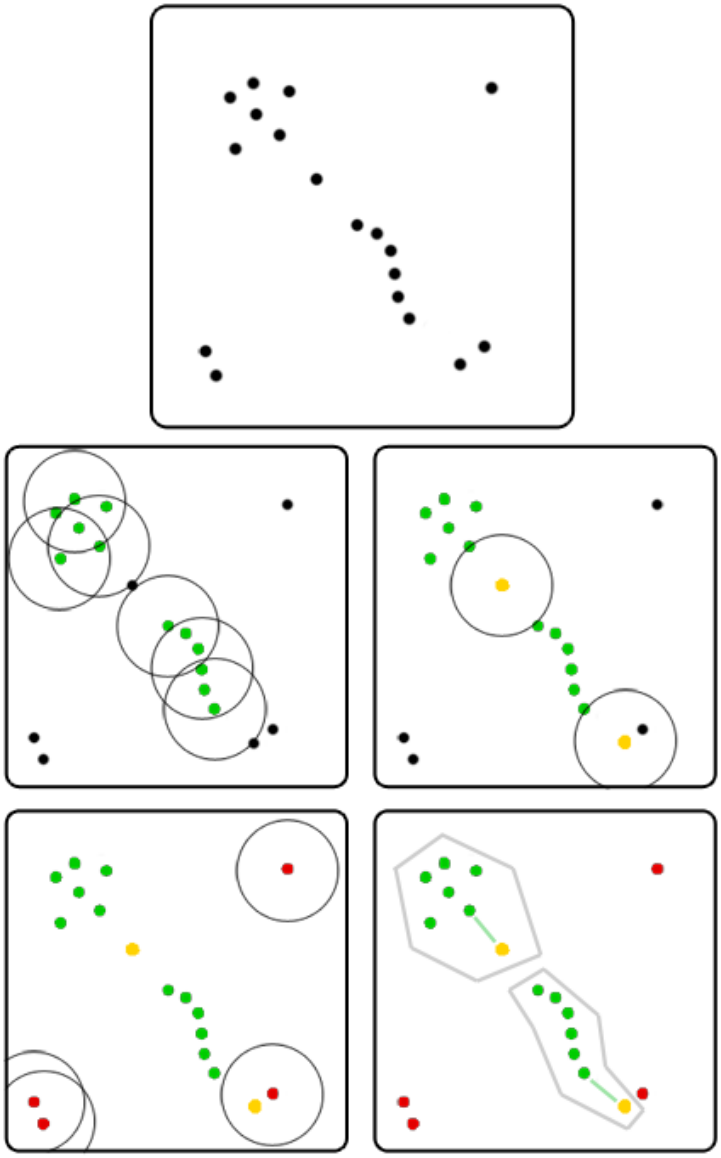


Рис. 10

Как правило, нет смысла помечать всех коренных элементов толпы сразу. Раз от каждого коренного элемента группы можно провести цепочку до каждого другого, то всё равно с какого начинать обход – рано или поздно найдёшь всех. Тут лучше подходит итеративный вариант:

1. Подходим к случайному человеку из толпы.
2. Если рядом с ним меньше трёх человек, переносим его в список возможных отшельников и выбираем кого-нибудь другого.
3. Иначе:

- Исключаем его из списка людей, которых надо обойти.
- Вручаем этому человеку зелёный флажок и создаём новую группу, в которой он пока что единственный обитатель.
- Обходим всех его соседей. Если его сосед уже в списке потенциальных одиночек или рядом с ним мало других людей, то перед нами край толпы. Для простоты можно сразу пометить его жёлтым флагом, присоединить к группе и продолжить обход. Если сосед тоже оказывается «зелёным», то он не стартует новую группу, а присоединяется к уже созданной; кроме того, мы добавляем в список обхода соседей соседа. Повторяем этот пункт, пока список обхода не окажется пуст.

4. Повторяем шаги 1–3, пока так или иначе не обойдём всех людей.
5. Разбираемся со списком отшельников. Если на шаге 3 мы уже раскидали всех краевых, то в нём остались только выбросы-одиночки – можно сразу закончить. Если нет, то нужно как-нибудь распределить людей, оставшихся в списке.

Формальный подход

Введём несколько определений. Пусть задана некоторая симметричная функция расстояния $\rho(x, y)$ и константы ε и m. Тогда

1. Назовём область $E(x)$, для которой $\forall y$: $\rho(x, y) \le \varepsilon$, ε-окрестностью объекта x.
2. Корневым объектом или ядерным объектом степени m называется объект, ε-окрестность которого содержит не менее m объектов: $|E(x)| \ge m$.
3. Объект p непосредственно плотно-достижим из объекта q, если $p \in E(q)$ и q – корневой объект.
4. Объект p плотно-достижим из объекта q, если $\exists p_1, p_2, \ldots, p_n$, $p_1=q$, $p_n=p$, такие что $\forall i \in 1, \ldots, n-1$: p_i+1 непосредственно плотно-достижим из p_i.

Выберем какой-нибудь корневой объект *p* из датасета, пометим его и поместим всех его непосредственно плотно-достижимых соседей в список обхода. Теперь для каждой *q* из списка: пометим эту точку, и, если она тоже корневая, добавим всех её соседей в список обхода. Тривиально доказывается, что кластеры помеченных точек, сформированные в ходе этого алгоритма максимальны (т.е. их нельзя расширить ещё одной точкой, чтобы удовлетворялись условия) и связны в смысле плотно-достижимости. Отсюда следует, что если мы обошли не все точки, можно перезапустить обход из какого-нибудь другого корневого объекта, и новый кластер не поглотит предыдущий.

Представим простенькую реализацию алгоритма на Python (рис. 11):

```
from itertools import cycle
from math import hypot
from numpy import random
import matplotlib.pyplot as plt
def dbscan_naive(P, eps, m, distance):
    NOISE = 0
    C = 0
    visited_points = set()
    clustered_points = set()
    clusters = {NOISE: []}
    def region_query(p):
        return [q for q in P if distance(p, q) < eps]
    def expand_cluster(p, neighbours):
        if C not in clusters:
            clusters[C] = []
        clusters[C].append(p)
        clustered_points.add(p)
        while neighbours:
            q = neighbours.pop()
            if q not in visited_points:
                visited_points.add(q)
                neighbourz = region_query(q)
                if len(neighbourz) > m:
                    neighbours.extend(neighbourz)
            if q not in clustered_points:
                clustered_points.add(q)
                clusters[C].append(q)
                if q in clusters[NOISE]:
                    clusters[NOISE].remove(q)
```

```
    for p in P:
        if p in visited_points:
            continue
        visited_points.add(p)
        neighbours = region_query(p)
        if len(neighbours) < m:
            clusters[NOISE].append(p)
        else:
            C += 1
            expand_cluster(p, neighbours)
    return clusters
if __name__ == "__main__":
    P = [(random.randn()/6, random.randn()/6) for i in range(150)]
    P.extend([(random.randn()/4 + 2.5, random.randn()/5) for i in range(150
    P.extend([(random.randn()/5 + 1, random.randn()/2 + 1) for i in range(1
    P.extend([(i/25 - 1, + random.randn()/20 - 1) for i in range(100)])
    P.extend([(i/25 - 2.5, 3 - (i/50 - 2)**2 + random.randn()/20) for i in
    clusters = dbscan_naive(P, 0.2, 4, lambda x, y: hypot(x[0] - y[0], x[1]
    for c, points in zip(cycle('bgrcmykgrcmykgrcmykgrcmykgrcmykgrcmyk'), cl
        X = [p[0] for p in points]
        Y = [p[1] for p in points]
        plt.scatter(X, Y, c=c)
    plt.show()
```

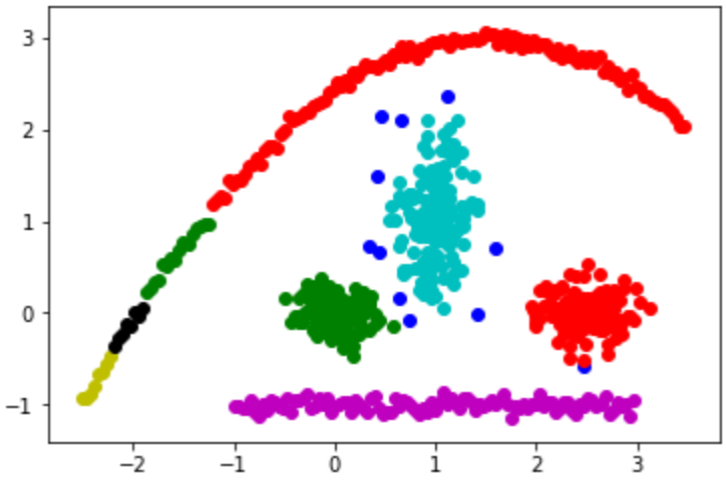


Рис. 11

Пример более корректной реализации DBSCAN на Python можно найти в пакете `sklearn`.

Нюансы применения

DBSCAN не вычисляет самостоятельно центры кластеров, особенно учитывая произвольную форму кластеров. Зато DBSCAN автоматически определяет выбросы.

Соотношение $m \epsilon n$, где n – размерность пространства, можно интуитивно рассматривать как пороговую плотность точек данных в области пространства. Ожидаемо, что при одинаковом соотношении $m \epsilon n$, и результаты будут примерно одинаковы. Иногда это действительно так, но есть причина, почему алгоритму нужно задать два параметра, а не один.

Во-первых, типичное расстояние между точками в разных датасетах разное – явно задавать радиус приходится всегда.

Во-вторых, играют роль неоднородности датасета. Чем больше m и ε, тем больше алгоритм склонен «прощать» вариации плотности в кластерах. С одной стороны, это может быть полезно: неприятно увидеть в кластере «дырки», где просто не хватило данных. С другой стороны, это вредно, когда между кластерами нет чёткой границы или шум создаёт «мост» между скоплениями. Тогда DBSCAN запросто соединит две разные группы. В балансе этих параметров и кроется сложность применения DBSCAN: реальные наборы данных содержат кластеры разной плотности с границами разной степени размытости. В условиях, когда плотность некоторых границ между кластерами больше или равна плотности каких-то обособленных кластеров, приходится чем-то жертвовать.

Существуют варианты DBSCAN, способные смягчить эту проблему. Идея состоит в подстраивании ε в разных областях по ходу работы алгоритма. К сожалению, возрастает количество параметров алгоритма.

Существуют эвристики для выбора m и ε. Чаще всего применяется следующий метод и его вариации:

1. Выберите m. Обычно используются значения от 3 до 9, чем более неоднородный ожидается датасет, и чем больше уровень шума, тем большим следует взять m.
2. Вычислите среднее расстояние по m ближайшим соседям для каждой точки, т.е. если m=3, нужно выбрать трёх ближайших соседей, сложить расстояния до них и поделить на три.

3. Сортируем полученные значения по возрастанию и выводим на экран.
4. Видим что-то вроде такого резко возрастающего графика (рис. 12). Следует взять ε где-нибудь в полосе, где происходит самый сильный перегиб. Чем больше ε, тем больше получатся кластеры, и тем меньше их будет.

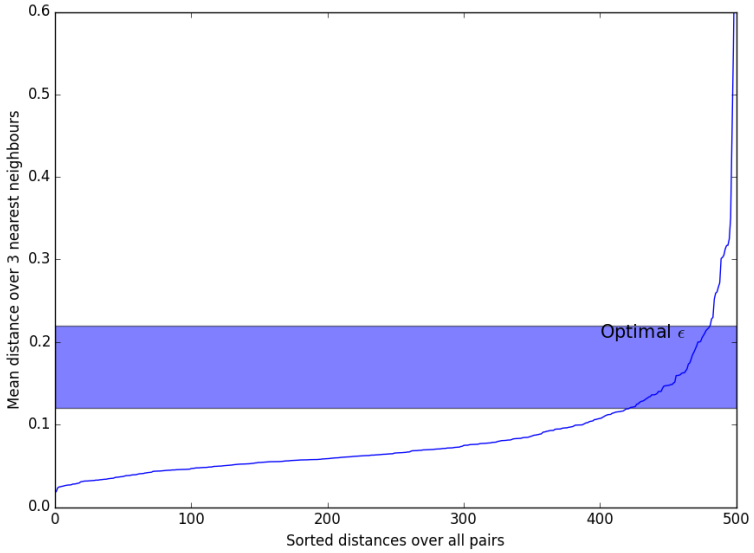


Рис. 12

В любом случае, главные недостатки DBSCAN – неспособность соединять кластеры через проёмы, и, наоборот, способность связывать явно различные кластеры через плотно населённые перемычки. Отчасти поэтому при увеличении размерности данных n коварный удар наносит проклятие размерности: чем больше n, тем больше мест, где могут случайно возникнуть проёмы или мосты. Напомним, что адекватное количество точек данных N возрастает экспоненциально с увеличением n.

Заключение

Используйте DBSCAN, когда:

- У вас в меру большой датасет.
- Заранее известна функция близости, симметричная, желательно, не очень сложная.

- Вы ожидаете увидеть сгустки данных экзотической формы: вложенные и аномальные кластеры, складки малой размерности.
- Плотность границ между сгустками меньше плотности наименее плотного кластера. Лучше, если кластеры вовсе отделены друг от друга.
- Сложность элементов датасета значения не имеет. Однако их должно быть достаточно, чтобы не возникало сильных разрывов в плотности (см. предыдущий пункт).
- Количество элементов в кластере может варьироваться сколь угодно.
- Количество выбросов значения не имеет (в разумных пределах), если они рассеяны по большому объёму.
- Количество кластеров значения не имеет.

DBSCAN успешных применяется в задачах:

- обнаружения социальных сообществ,
- сегментирования изображений,
- моделирования поведения пользователей веб-сайтов,
- предварительной обработки в задаче предсказания погоды.