НИУ ВШЭ НН. Факультет ИМиКН. Методы анализа данных. Charge de cours: В. А. Калягин

Домашнее задание 2: алгоритмы кластерного анализа. **Вариант 14**. Выполнил: **Игорь Рухович**

Импортируем необходимые библиотеки

In [ ]:
```
from matplotlib import pyplot as plt
from mst_clustering import MSTClustering
import networkx as nx
from networkx.algorithms.community import modularity
import numpy as np
import pandas as pd
from scipy.spatial import distance_matrix
from sklearn.cluster import DBSCAN, KMeans, SpectralClustering
from sklearn.metrics import davies_bouldin_score
from sklearn.mixture import GaussianMixture
import seaborn as sns
```

Код для чтения данных + общие переменные

In [ ]:
```
data_path = "../data/xls/"
filename = lambda dtype, number: f"{dtype}/{dtype.upper()}_4_{number}.xlsx"
variants = range(1, 16)
dtypes = ["x", "y"]
my_variant = 14
random_state = 123

def read_specific_variant(dtype: str, number: int) -> pd.DataFrame:
    if dtype not in dtypes:
        raise RuntimeError("Wrong dtype")
    if number not in variants:
        raise RuntimeError("Wrong variant number")
    return pd.read_excel(data_path + filename(dtype, number), header=None)
```

Данные. Первичная обработка.

0.1 Рассмотрим данные

Рассмотрим первые строчки датасетов:

In [ ]:
```
df = {}

df[dtypes[0]] = read_specific_variant(dtypes[0], my_variant)
df["x"].head()
```

Out[ ]:

	0	1	2	3
0	5.092168	1.034724	7.080165	-1.987996
1	4.846424	0.968371	6.787735	-1.941311
2	5.297292	1.205368	6.978480	-1.681188
3	3.729736	0.846779	4.919136	-1.189400
4	5.003340	1.094723	6.722511	-1.719171

In [ ]:
```
df[dtypes[1]] = read_specific_variant(dtypes[1], my_variant)
df["y"].head()
```

Out[ ]:

	0	1	2	3
0	0.906940	1.006914	0.963778	0.824834
1	2.029085	0.799312	1.623610	0.952911
2	1.978421	1.102946	1.458704	0.978444
3	2.275134	0.829438	1.964464	1.040498
4	1.276684	0.056152	-0.198731	1.130742

Видим, что в обоих случаях мы имеем дело с векторами действительных чисел размерности 4.

Соберём некоторые статистики: среднее, стандартное отклонение и каждый 25-й процентиль:

In [ ]:
```
df["x"].describe()
```

Out[ ]:

	0	1	2	3
count	200.000000	200.000000	200.000000	200.000000
mean	2.999880	0.978973	3.062841	-0.062961
std	2.665400	1.190919	2.636290	1.659201
min	-1.770611	-1.098890	-1.686545	-3.001601
25%	0.967169	0.094967	1.037855	-1.337723
50%	3.043348	1.013192	3.001980	-0.003096
75%	5.053307	1.939336	5.064147	1.147939
max	7.684823	3.085918	8.097899	3.110166

In [ ]:
```
df["y"].describe()
```

Out[ ]:

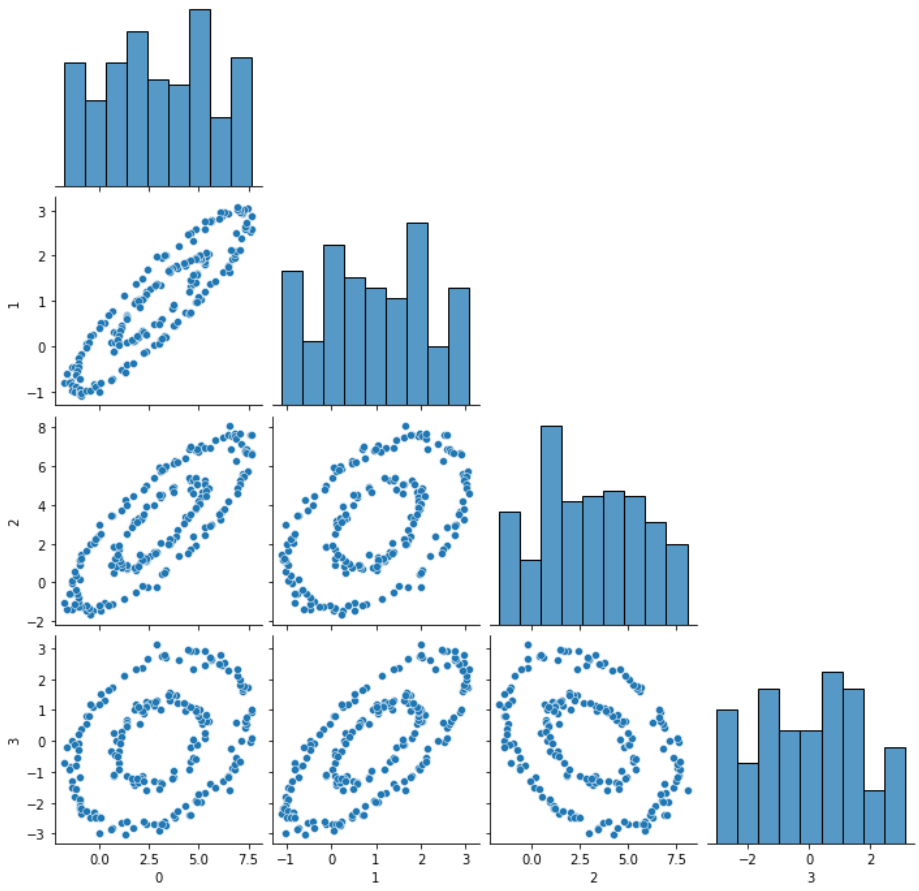
	0	1	2	3
count	200.000000	200.000000	200.000000	200.000000
mean	1.384176	0.678902	1.068702	0.975327
std	0.514553	0.497740	0.875862	0.148078
min	0.627841	-0.832312	-0.307301	0.543181
25%	0.942615	0.177061	0.076568	0.894277
50%	1.121562	0.919683	1.059251	0.978145
75%	1.967040	1.036559	1.925251	1.050821
max	2.275134	1.349070	2.520664	1.482438

Оба набора данных состоят из 200 действительных векторов. По собранным статистикам зависимости между наборами или колонками не прослеживаются.

Чтобы лучше понять устройство данных, построим гистограммы распределения всех прихнаков, а так же их попарные распределения. **Сверху** распределения **набора X**, а **снизу** - **набора Y**.

In [ ]:
```
sns.pairplot(df["x"], corner=True)
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9183140d0>
```



В первом наборе данных распределение признаков по отдельности ни о чем нам не говорит, но можно заметить схожесть гистограмм разных признаков. 3 из 4 распределений имеют по 4 пика на тех же местах, значения плотно сгруппированы в узких диапазонах. В то же время, попарные распределения формируют очень интересную картину - мы видим, что данные в каждом попарном распределении представляют собой два эллипса с шумом, вписанные друг в друга. Можем сделать вывод, что данные - это две

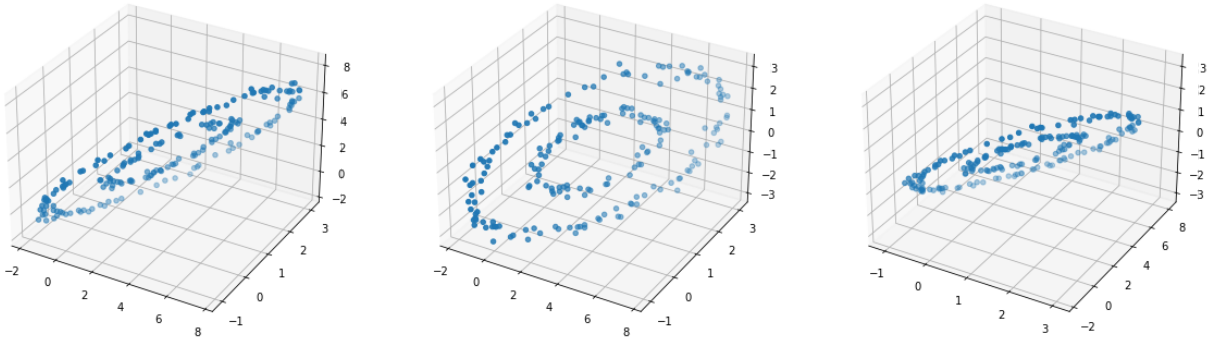
двумерных окружности (или эллипса) с общим центром в 4-мерном пространстве. **Есть вероятность, что этот набор возможно вписать в 2 координаты** без значительных потерь.

Проверим на 3-мерном рисунке:

In [ ]:
```
fig, axs = plt.subplots(1, 3, subplot_kw=dict(projection='3d'))
fig.set_size_inches(20,10)

axs[0].scatter(df["x"][0], df["x"][1], df["x"][2])
axs[1].scatter(df["x"][0], df["x"][1], df["x"][3])
axs[2].scatter(df["x"][1], df["x"][2], df["x"][3])
```

Out[ ]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fb8f91c3eb0>
```



Проверим наше предположение с помощью сингулярного разложения:

In [ ]:
```
u, s, vt = np.linalg.svd(df["x"], full_matrices=False)
print(f"Сингулярные числа матрицы:\n{s.round(4)}")
tmp_s = np.append(s, 0)
print(f"Абсолютная погрешность аппроксимации в спектральной матричной норме:\n{tmp_s[1:].round(4)}")
abs_errors_f = (tmp_s**2)[::-1].cumsum()[::-1][1:]
print(f"Абсолютная погрешность аппроксимации в норме Фробениуса:\n{abs_errors_f.round(4)}")
print(f"Относительная погрешность аппроксимации в норме Фробениуса, %:\n{(abs_errors_f/(tmp_s**2).sum()*100).round(4)}")
```

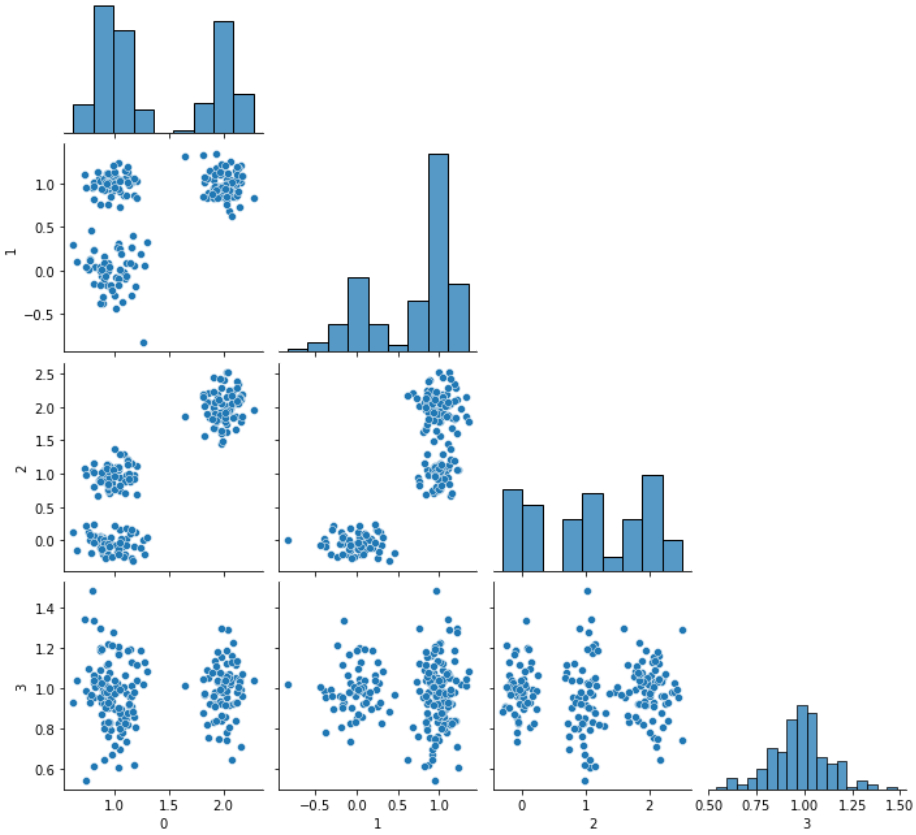
```
Сингулярные числа матрицы:
[80.8877 30.8643  0.      0.    ]
Абсолютная погрешность аппроксимации в спектральной матричной норме:
[30.8643  0.      0.      0.    ]
Абсолютная погрешность аппроксимации в норме Фробениуса:
[952.6072   0.       0.       0.    ]
Относительная погрешность аппроксимации в норме Фробениуса, %:
[12.7092  0.      0.      0.    ]
```

Действительно, матрица **имеет ровно 2 ненулевых сингулярных числа**, а значит данные можно поместить в Евклидово пространство размерности 2 без потерь! (но мы в работе, разумеется, делать этого не будем)

Перейдём ко второму набору:

In [ ]:
```
sns.pairplot(df["y"], corner=True)
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8f9653820>
```



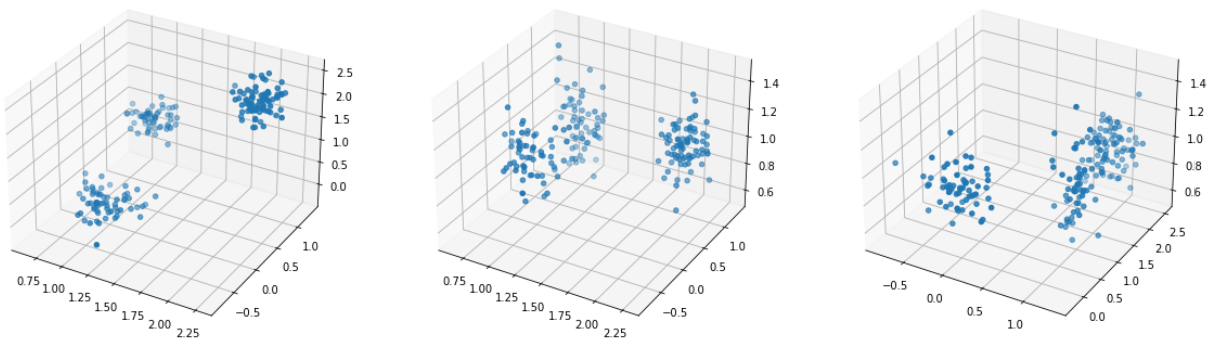
По одиночным распределениям видим, что признаки из набора похожи на одно (признак 3) или несколько (признаки 0, 1) нормальных распределений. Попарные распределения явно делят данные на "плотные" множества, с похожим на нормальное с шумом распределение.

Посмотрим на трехмерные картинки:

In [ ]:
```
fig, axs = plt.subplots(1, 3, subplot_kw=dict(projection='3d'))
fig.set_size_inches(20,10)

axs[0].scatter(df["y"][0], df["y"][1], df["y"][2])
axs[1].scatter(df["y"][0], df["y"][1], df["y"][3])
axs[2].scatter(df["y"][1], df["y"][2], df["y"][3])
```

Out[ ]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fb8faa71b50>
```



"Кучная" структура сохраняется, видим 3 разделяемых подмножества.

Снова применим сингулярное разложение:

In [ ]:
```
u, s, vt = np.linalg.svd(df["y"], full_matrices=False)
print(f"Сингулярные числа матрицы:\n{s.round(4)}")
tmp_s = np.append(s, 0)
print(f"Абсолютная погрешность аппроксимации в спектральной матричной норме:\n{tmp_s[1:].round(4)}")
abs_errors_f = (tmp_s**2)[::-1].cumsum()[::-1][1:]
print(f"Абсолютная погрешность аппроксимации в норме Фробениуса:\n{abs_errors_f.round(4)}")
print(f"Относительная погрешность аппроксимации в норме Фробениуса, %:\n{(abs_errors_f/(tmp_s**2).sum()*100).round(4)}")
```

```
Сингулярные числа матрицы:
[32.5016  8.4099  4.6414  2.1061]
Абсолютная погрешность аппроксимации в спектральной матричной норме:
[8.4099 4.6414 2.1061 0.    ]
Абсолютная погрешность аппроксимации в норме Фробениуса:
[96.7043 25.9782  4.4356  0.    ]
Относительная погрешность аппроксимации в норме Фробениуса, %:
[8.3868 2.253  0.3847 0.    ]
```

Видим, что данные без больших потерь возможно поместить в Евклидово пространство размерности 2 и 3, но сохраним в данной работе полноценную структуру.

0.2 Подготовка данных

В качестве дистанции выберем Евклидово расстояние между точками, поскольку на обеих картинках выше мы используем Евклидову метрику и уже видим хорошую кластерную структуру. Нет смысла усложнять этот шаг. Матрицу близости (adjacency matrix) возьмём как матрицу расстояний, вычтенную из её наибольшего значения.

In [ ]:
```
dist_x = distance_matrix(df["x"], df["x"], p=2)
adj_x = dist_x.max() - dist_x
adj_x
```

Out[ ]:
```
array([[13.37497493, 12.98447807, 12.95584968, ..., 12.56958097,
        13.07251206, 12.66696944],
       [12.98447807, 13.37497493, 12.77206687, ..., 12.74660732,
        13.2822014 , 12.75819323],
       [12.95584968, 12.77206687, 13.37497493, ..., 12.19863341,
        12.81294539, 12.26975961],
       ...,
       [12.56958097, 12.74660732, 12.19863341, ..., 13.37497493,
        12.74654112, 13.20472724],
       [13.07251206, 13.2822014 , 12.81294539, ..., 12.74654112,
        13.37497493, 12.78326362],
       [12.66696944, 12.75819323, 12.26975961, ..., 13.20472724,
        12.78326362, 13.37497493]])
```

In [ ]:
```
dist_y = distance_matrix(df["y"], df["y"], p=2)
adj_y = dist_y.max() - dist_y
adj_y
```

Out[ ]:
```
array([[3.29791481, 1.97349415, 2.1038289 , ..., 3.1387171 , 1.96430166,
        1.47992442],
       [1.97349415, 3.29791481, 2.94776274, ..., 2.04471137, 3.02423831,
        2.42449095],
       [2.1038289 , 2.94776274, 3.29791481, ..., 2.16211617, 2.85622037,
        2.32885691],
       ...,
       [3.1387171 , 2.04471137, 2.16211617, ..., 3.29791481, 2.00983162,
        1.51321064],
       [1.96430166, 3.02423831, 2.85622037, ..., 2.00983162, 3.29791481,
        2.62051559],
       [1.47992442, 2.42449095, 2.32885691, ..., 1.51321064, 2.62051559,
        3.29791481]])
```

В таком виде матрицу расстояний можно также назвать матрицей смежности для полного графа, вершинами которого являются точки из набора данных, а расстояние между точками определяет вес ребра. Поскольку все точки имеют конечные координаты, существует ребро между любой парой точек, а значит граф полный. В координатах i, i находятся нули - граф не содержит петель. Евклидово расстояние коммутативно - граф неориентированный:

In [ ]:
```
dist_x
```

Out[ ]:
```
array([[0.        , 0.39049687, 0.41912526, ..., 0.80539397, 0.30246287,
        0.70800549],
       [0.39049687, 0.        , 0.60290806, ..., 0.62836761, 0.09277354,
        0.61678171],
       [0.41912526, 0.60290806, 0.        , ..., 1.17634152, 0.56202955,
        1.10521532],
       ...,
       [0.80539397, 0.62836761, 1.17634152, ..., 0.        , 0.62843381,
        0.17024769],
       [0.30246287, 0.09277354, 0.56202955, ..., 0.62843381, 0.        ,
        0.59171132],
       [0.70800549, 0.61678171, 1.10521532, ..., 0.17024769, 0.59171132,
        0.        ]])
```

Создадим графовые структуры с помощью модуля `networkx`:

In [ ]:
```
def create_graph(distances):
    G = nx.Graph()
    for i in range(distances.shape[0]):
        for j in range(distances.shape[1]):
            G.add_edge(i, j, weight=distances[i, j])
    return G

G_x = create_graph(adj_x)
G_y = create_graph(adj_y)
```

1. Кластеризация на 2 кластера

Попробуем использовать некоторые алгоритмы кластеризации над данными.

Будем отдельно хранить метки кластеров, для дальнейших подсчетов:

In [ ]:
```
labels = {"x": {}, "y": {}}
```

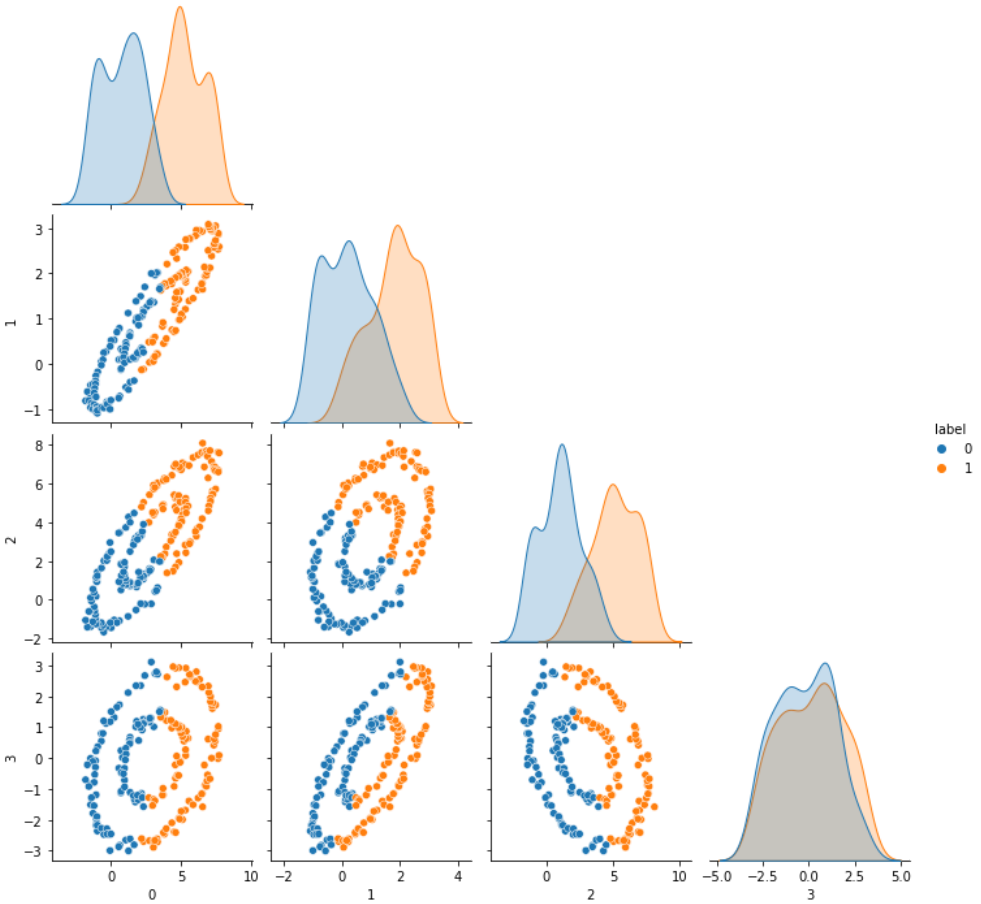
1.1 Находим разбиение наборов

K-means clustering

In [ ]:
```
kmeans = KMeans(n_clusters=2, init="k-means++", n_init="auto",
                random_state=random_state, algorithm="lloyd").fit(df["x"])
labels["x"]["kmeans"] = kmeans.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["kmeans"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8faaa5d60>
```



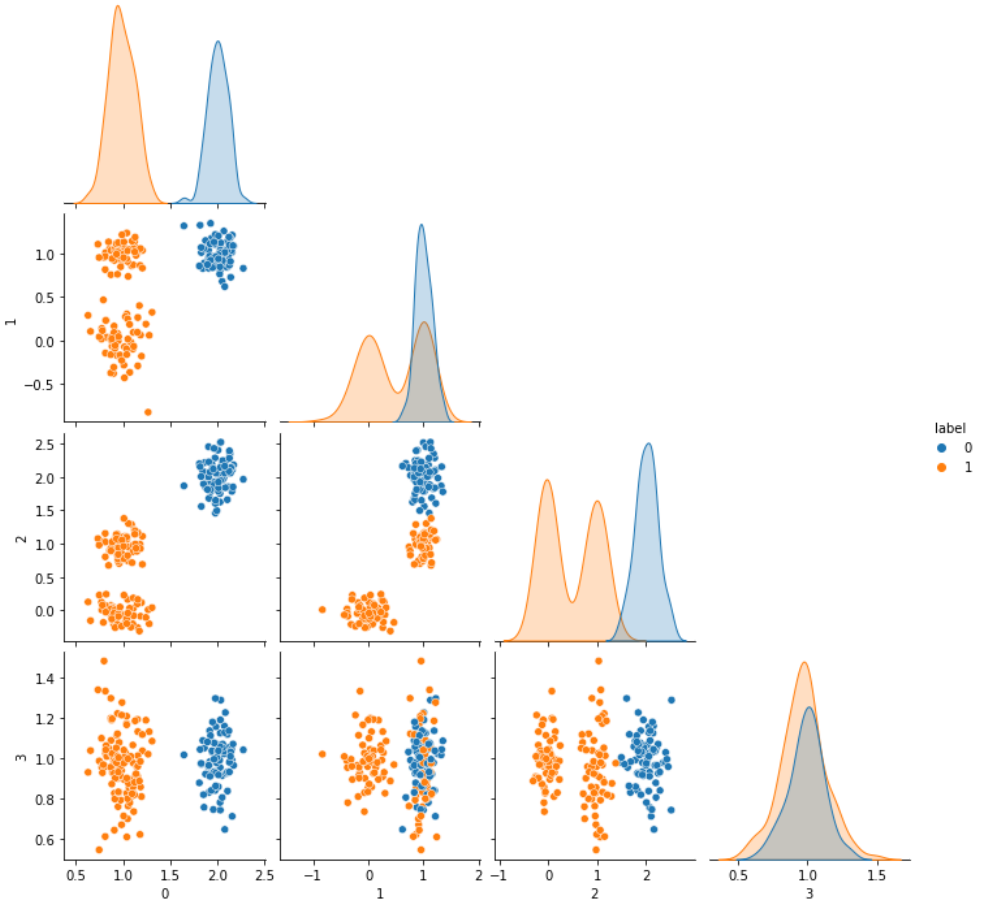
Мы использовали оптимизированный алгоритм `greedy k-means++`. Его преимущество в том, что начальные центроиды выбираются с помощью специального алгоритма, основанного на инерции. Таким образом, нам не требуется делать много итераци, чтобы получить хороший результат.

Тем не менее видим, что k-means со своей задачей на этом наборе справился плохо - что ожидаемо, поскольку алгоритм основан на близости точек, а мы имеем дело с вписанными эллипсами.

In [ ]:
```
kmeans = KMeans(n_clusters=2, init="k-means++", n_init="auto",
                random_state=random_state, algorithm="lloyd").fit(df["y"])
labels["y"]["kmeans"] = kmeans.labels_

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["kmeans"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb918a8fe80>
```



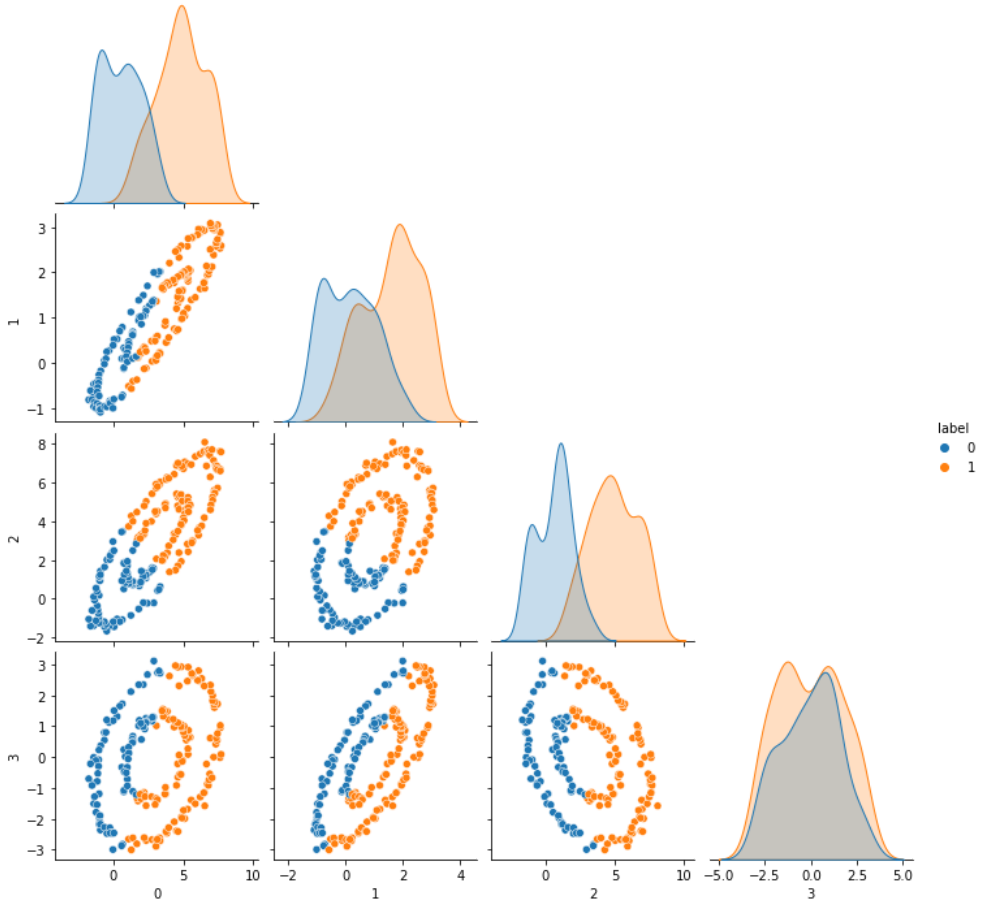
Что касается второго набора данных - k-means показывает себя отлично. Он явно отделил одно из множеств точек в отдельный кластер (остальные смешаны, поскольку имеем 2 кластера)

EM-алгоритм (Gaussian mixture)

In [ ]:
```
labels["x"]["em"] = GaussianMixture(n_components=2, init_params="k-means++",
                                    random_state=random_state).fit_predict(df["x"])

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["em"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8fcd67e50>
```



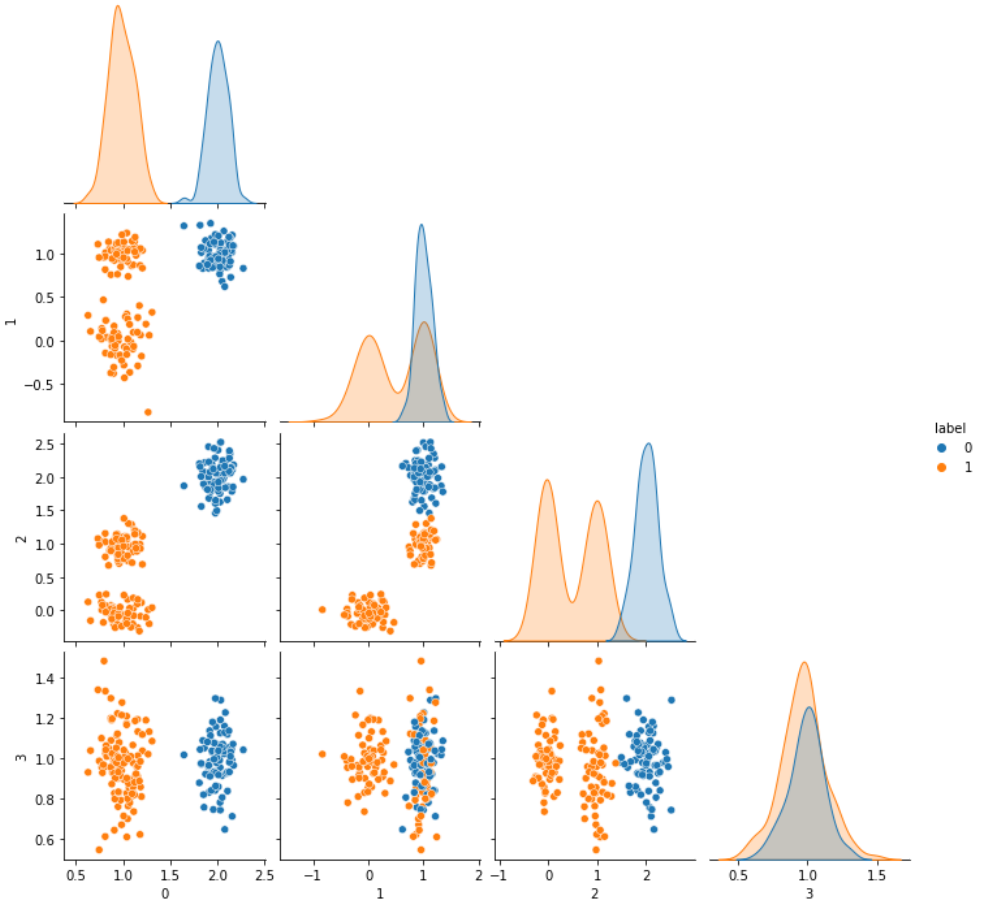
EM-алгоритм справился с датасетом X не лучше k-means, что не удивительно, ведь алгоритмы довольно похожи. В качестве инициализации EM брали уже упомянутый `k-means++`, помогающий лучше выбрать стартовые значения.

Проверим этот алгоритм на Y:

In [ ]:
```
labels["y"]["em"] = GaussianMixture(n_components=2, init_params="k-means++",
                                    random_state=random_state).fit_predict(df["y"])

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["em"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8faed03a0>
```



На втором наборе EM-алгоритм тоже смог отделить один из кластеров и сработал очень похоже на k-means

MST method

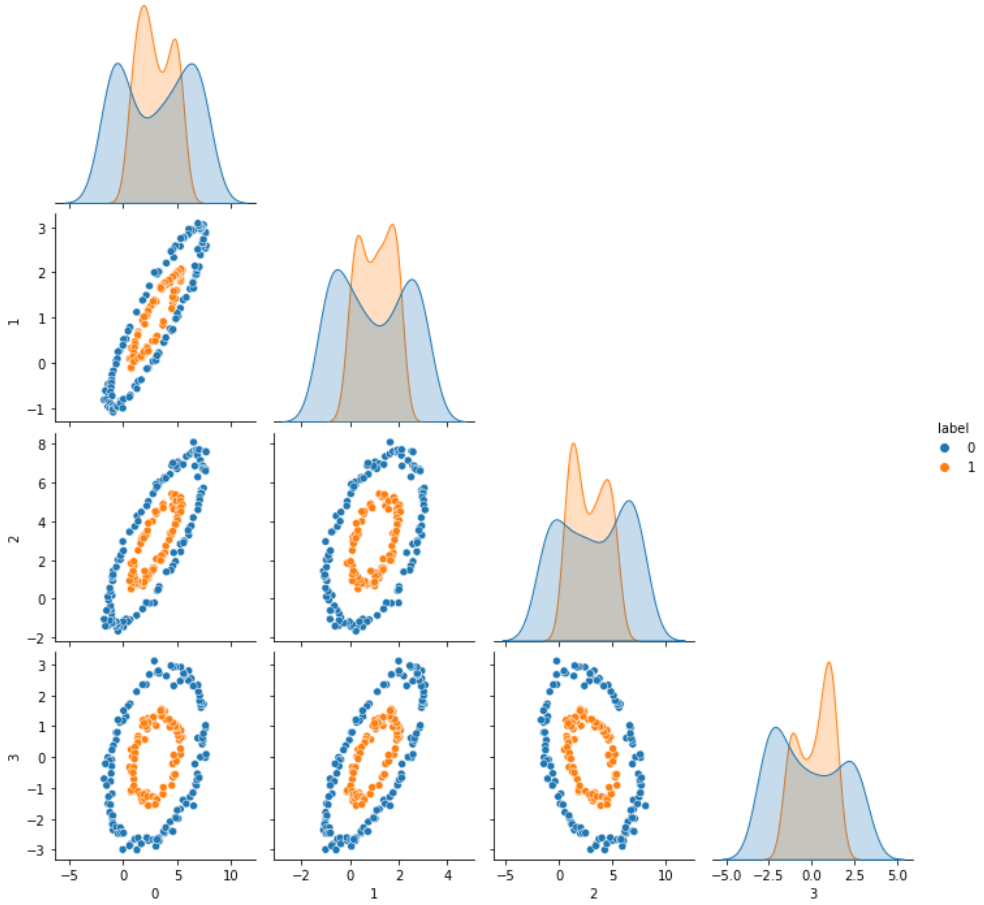
Для MST алгоритма используется сторонняя реализация: https://github.com/jakevdp/mst_clustering

В параметрах данного алгоритма выбираем, сколько рёбер нам оставить (`cutoff=1` - нужно получить 2 кластера). В качестве дистанции в этой реализации используется Евклидово расстояние между точками.

In [ ]:
```
labels["x"]["mst"] = MSTClustering(cutoff=1, approximate=False).fit_predict(df["x"])

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["mst"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8fe19d9a0>
```

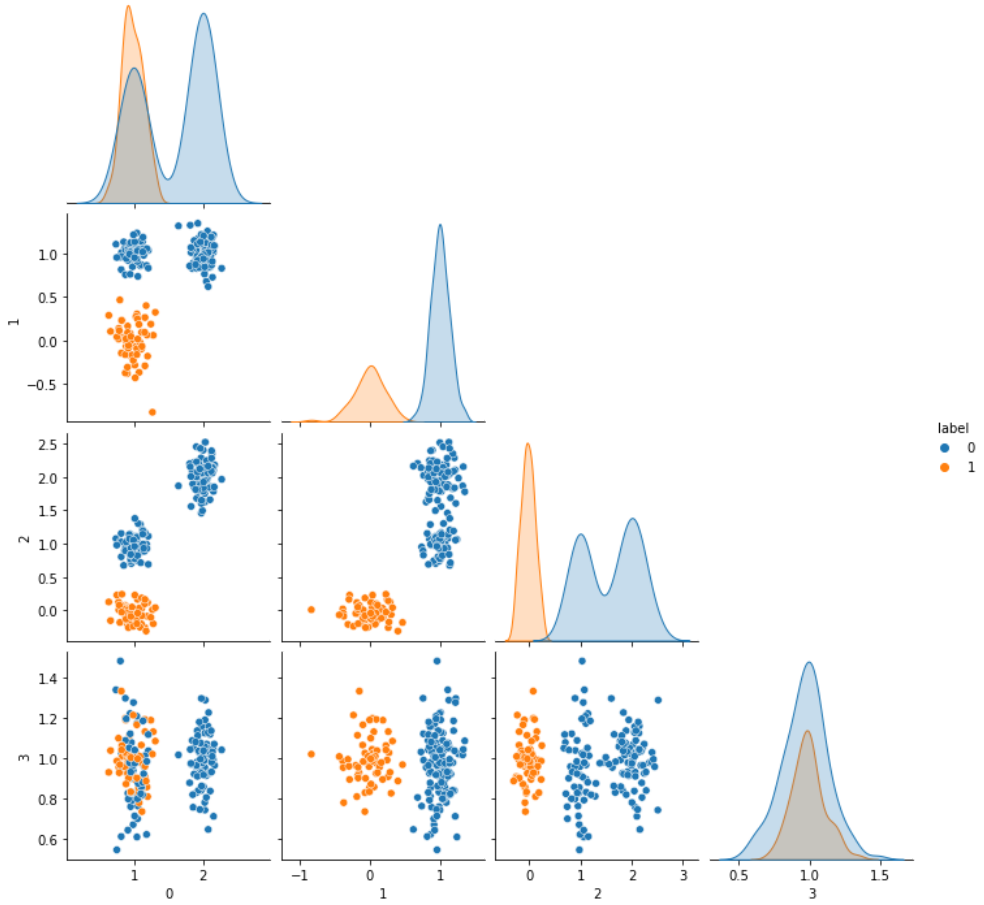


Как мы видим, алгоритм отлично справился с первым набором, четко разделив окружности. Попробуем на втором:

In [ ]:
```
labels["y"]["mst"] = MSTClustering(cutoff=1, approximate=False).fit_predict(df["y"])

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["mst"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8ffa8ff40>
```



Как и предыдущие алгоритмы, MST нашёл среди данных "плотное" множество и выделил его в кластер - отлично!

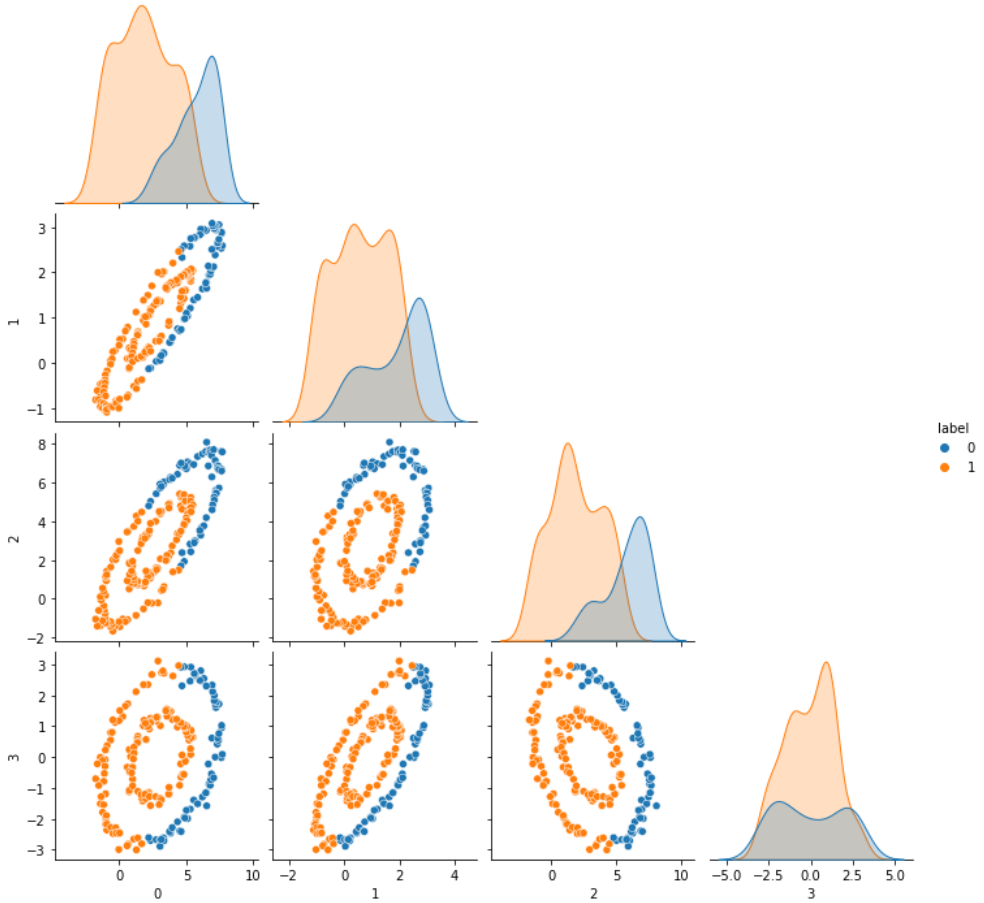
Spectral clustering

Попробуем кластеризовать наборы с помощью спектрального алгоритма:

In [ ]:
```
spectral = SpectralClustering(n_clusters=2, affinity="nearest_neighbors", random_state=random_state).fit(df["x"])
labels["x"]["spectral"] = spectral.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8ff8acaf0>
```



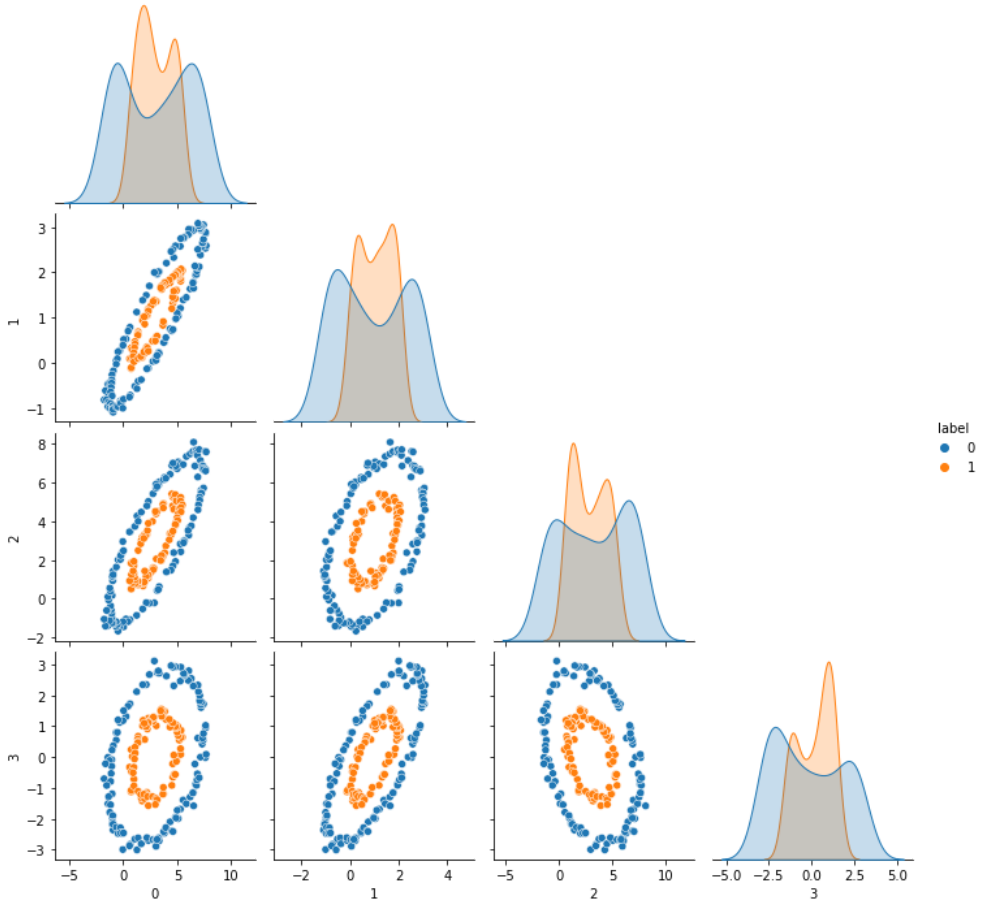
Ничего не получилось - и это не ожидаемо. Спектральный алгоритм должен хорошо работать на таких данных. Видим, что он не взял центральное кольцо в синий кластер, тем не менее, внешнее кольцо он выделил только наполовину.

Возможно, всё дело в избыточной размерности. Попробуем перед кластеризацией применить PCA и оставить 2 компоненты (знаем, что этот набор можно без потерь поместить в двумерное Евклидово пространство):

In [ ]:
```
u, s, vt = np.linalg.svd(df["x"], full_matrices=False)
spectral = SpectralClustering(n_clusters=2, affinity="nearest_neighbors", random_state=random_state).fit(u[:,:2])
labels["x"]["spectral"] = spectral.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8ffaaaee0>
```



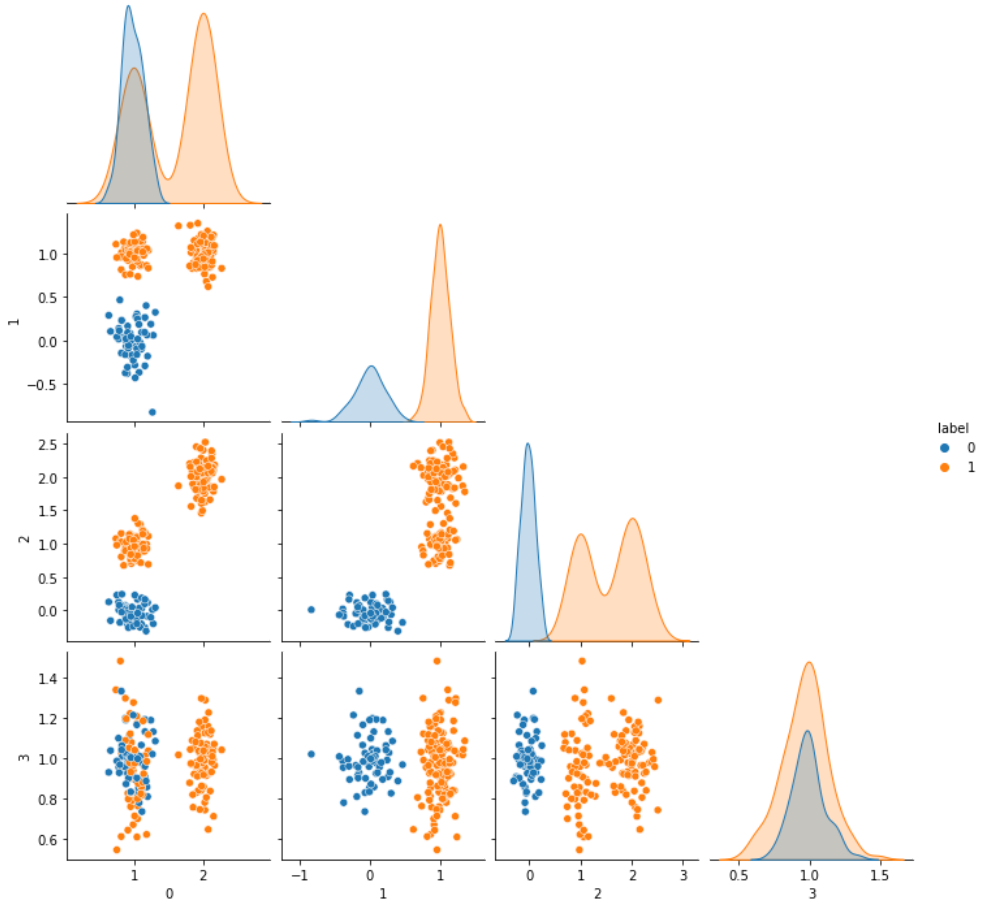
Да, действительно! При удалении линейных зависимостей в признаках спектральный алгоритм стал правильно выполнять кластеризацию!

Посмотрим на наборе Y:

In [ ]:
```
spectral = SpectralClustering(n_clusters=2, random_state=random_state).fit(df["y"])
labels["y"]["spectral"] = spectral.labels_

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9017108b0>
```



Спектральный алгоритм справился с этой задачей. Один кластер выделен.

1.2, 1.3 RAND index и модулярность

RAND index - индекс похожести двух разбиений, похожий на метрику `accuracy`. Реализуем его подсчет самостоятельно:

In [ ]:
```
def compute_rand_index(labels_x, labels_y):
    if labels_x.shape[0] != labels_y.shape[0]:
        raise RuntimeError("Shapes must be equal")
    ab, cd = 0, 0
    for i in range(labels_x.shape[0]):
        for j in range(i+1, labels_x.shape[0]):
            if labels_x[i] == labels_x[j]:
                if labels_y[i] == labels_y[j]:
                    ab += 1
                else:
                    cd += 1
            else:
                if labels_y[i] == labels_y[j]:
                    cd += 1
                else:
                    ab += 1

    return (ab) / (ab + cd)
```

In [ ]:
```
print(f'RAND index для X, K-means и EM: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["em"]):.3f}')
print(f'RAND index для Y, K-means и EM: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["em"]):.3f}\n')

print(f'RAND index для X, K-means и MST: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["mst"]):.3f}')
print(f'RAND index для Y, K-means и MST: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["mst"]):.3f}\n')

print(f'RAND index для X, K-means и Spectral: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, K-means и Spectral: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["spectral"]):.3f}\n')

print(f'RAND index для X, EM и MST: \
{compute_rand_index(labels["x"]["em"], labels["x"]["mst"]):.3f}')
print(f'RAND index для Y, EM и MST: \
{compute_rand_index(labels["y"]["em"], labels["y"]["mst"]):.3f}\n')

print(f'RAND index для X, EM и Spectral: \
{compute_rand_index(labels["x"]["em"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, EM и Spectral: \
```

```
{compute_rand_index(labels["y"]["em"], labels["y"]["spectral"]):.3f}\n')

print(f'RAND index для X, MST и Spectral: \
{compute_rand_index(labels["x"]["mst"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, MST и Spectral: \
{compute_rand_index(labels["y"]["mst"], labels["y"]["spectral"]):.3f}\n')
```

```
RAND index для X, K-means и EM: 0.861
RAND index для Y, K-means и EM: 1.000

RAND index для X, K-means и MST: 0.502
RAND index для Y, K-means и MST: 0.586

RAND index для X, K-means и Spectral: 0.502
RAND index для Y, K-means и Spectral: 0.586

RAND index для X, EM и MST: 0.498
RAND index для Y, EM и MST: 0.586

RAND index для X, EM и Spectral: 0.498
RAND index для Y, EM и Spectral: 0.586

RAND index для X, MST и Spectral: 1.000
RAND index для Y, MST и Spectral: 1.000
```

Видим, как и ожадалось, что K-means и EM алгоритм дают похожие разбиения, а также MST и Спектральный алгоритм работают в данном случае идентично. Между собой разбиения, полученные этими алгоритмами похожи не так сильно.

Перейдём к модулярности. Будем рассчитывать её с помощью встроенного в модуль `networkx` функционала. Уже имеем графовые структуры на основе матриц близости в этом фреймворке. Посчитаем модулярности разбиений:

In [ ]:
```
def get_lables(dtype, algo, n_clusters):
    if dtype not in dtypes:
        raise RuntimeError("Dtype must be x or y")
    if algo not in labels[dtype]:
        raise RuntimeError("Lables are not calculated yet")
    s0 = set(np.where(labels[dtype][algo]==0)[0])
    s1 = set(np.where(labels[dtype][algo]==1)[0])
    s2 = set(np.where(labels[dtype][algo]==2)[0])
    if n_clusters == 2:
        return [s0, s1]
    elif n_clusters == 3:
        return [s0, s1, s2]
    else:
        raise RuntimeError("n_clusters must be in [2, 3]")
```

In [ ]:
```
############### THIS CODE WILL BE USED LATER (3.2) ###############

db_score_2 = f"""Подсчет Davies-Bouldin score для набора X (2 кластера):
K-means: {davies_bouldin_score(df["x"], labels["x"]["kmeans"]):.3f}
EM: {davies_bouldin_score(df["x"], labels["x"]["em"]):.3f}
MST: {davies_bouldin_score(df["x"], labels["x"]["mst"]):.3f}
Spectral: {davies_bouldin_score(df["x"], labels["x"]["spectral"]):.3f}

Подсчет Davies-Bouldin score для набора Y (2 кластера):
K-means: {davies_bouldin_score(df["y"], labels["y"]["kmeans"]):.3f}
EM: {davies_bouldin_score(df["y"], labels["y"]["em"]):.3f}
MST: {davies_bouldin_score(df["y"], labels["y"]["mst"]):.3f}
Spectral: {davies_bouldin_score(df["y"], labels["y"]["spectral"]):.3f}
"""

####################################################################
```

In [ ]:
```
print("X dataset:")
print(f'Modularity value for K-means clustering: \
{modularity(G_x, get_lables("x", "kmeans", 2)):.3f}')
print(f'Modularity value for EM clustering: \
{modularity(G_x, get_lables("x", "em", 2)):.3f}')
print(f'Modularity value for MST clustering: \
{modularity(G_x, get_lables("x", "mst", 2)):.3f}')
print(f'Modularity value for Spectral clustering: \
{modularity(G_x, get_lables("x", "spectral", 2)):.3f}')

print("\nY dataset:")
print(f'Modularity value for K-means clustering: \
{modularity(G_x, get_lables("y", "kmeans", 2)):.3f}')
print(f'Modularity value for EM clustering: \
{modularity(G_x, get_lables("y", "em", 2)):.3f}')
print(f'Modularity value for MST clustering: \
{modularity(G_x, get_lables("y", "mst", 2)):.3f}')
print(f'Modularity value for Spectral clustering: \
{modularity(G_x, get_lables("y", "spectral", 2)):.3f}')
```

```
X dataset:
Modularity value for K-means clustering: 0.112
Modularity value for EM clustering: 0.105
Modularity value for MST clustering: 0.017
Modularity value for Spectral clustering: 0.017

Y dataset:
Modularity value for K-means clustering: 0.006
Modularity value for EM clustering: 0.006
Modularity value for MST clustering: 0.006
Modularity value for Spectral clustering: 0.006
```

Модулярность показывает, что разбиения, сделанные K-means и EM лучше MST и Spectral для обоих наборов данных. В то же время мы видим, что, напротив, правильно кластеризовали первый набор только MST и Спектральный алгоритм. Что касается набора Y - все алгоритмы показали себя одинаково хорошо, хотя модулярность и не говорит об этом.

1.4 Выводы

Набор X, состоящий из двух необычно расположенных множеств, был правильно кластеризован алгоритмами MST и Спектральным (при понижении размерности). EM и K-means не нашли никакой интересной кластерной структуры.

В то же время набор Y, состоящий из 3-х плотно расположенных групп точек, был правильно кластеризован всеми алгоритмами - каждый правильно выделил по одному обособленному множеству.

Таким образом, MST-алгоритм объявляется победителем в данном эксперименте. Для Спектрального делаем вывод - алгоритм хорош, но нужно избавляться от линейной зависимости.

2. Кластеризация на 3 кластера

Выполним все те же действия, но на этот раз будем искать в данных 3 кластера:

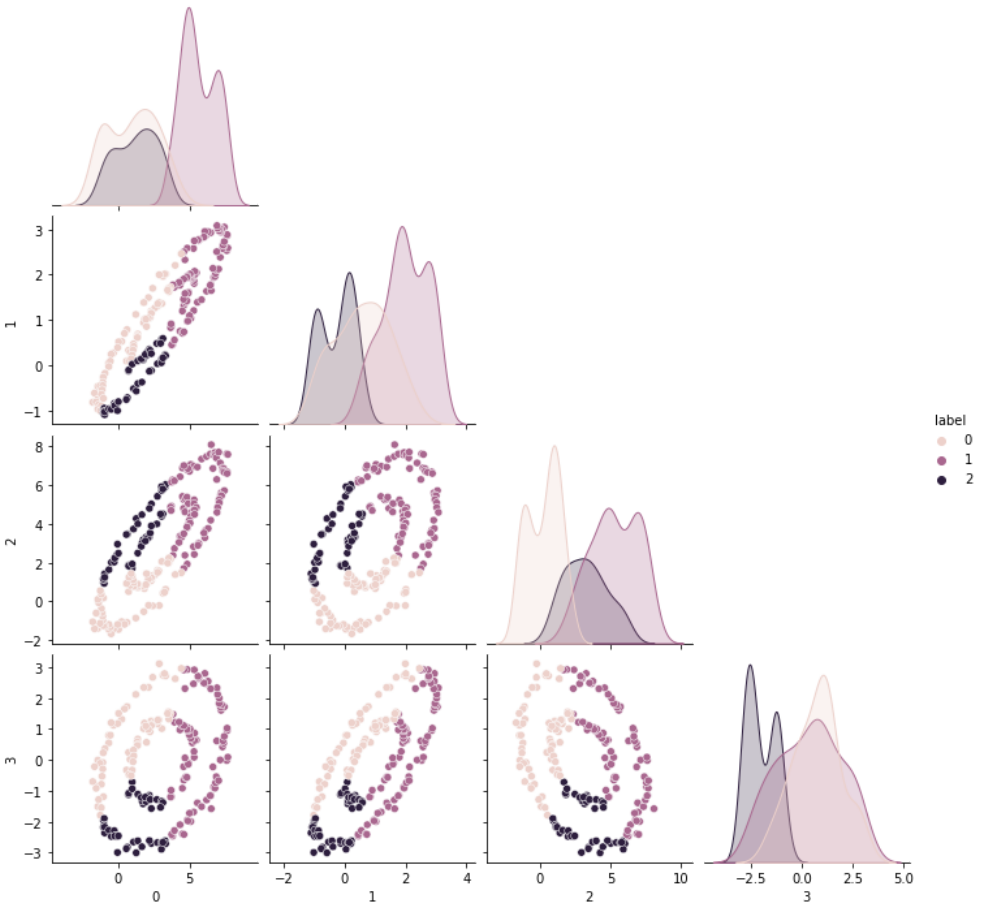
2.1 Находим разбиение наборов

K-means clustering

In [ ]:
```
kmeans = KMeans(n_clusters=3, init="k-means++", n_init="auto",
                random_state=random_state, algorithm="lloyd").fit(df["x"])
labels["x"]["kmeans"] = kmeans.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["kmeans"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9028345e0>
```

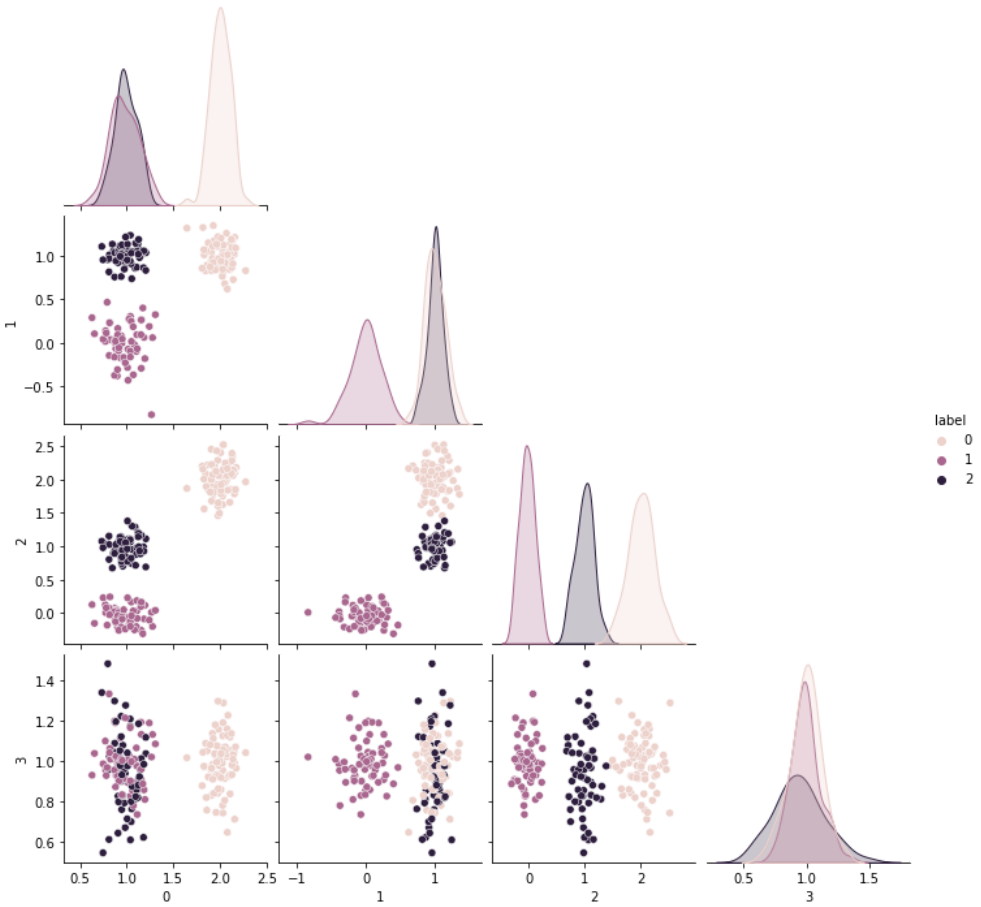


K-means снова не смог установить кластерную структуру в наборе X.

In [ ]:
```
kmeans = KMeans(n_clusters=3, init="k-means++", n_init="auto",
                random_state=random_state, algorithm="lloyd").fit(df["y"])
labels["y"]["kmeans"] = kmeans.labels_

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["kmeans"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8ffab5400>
```



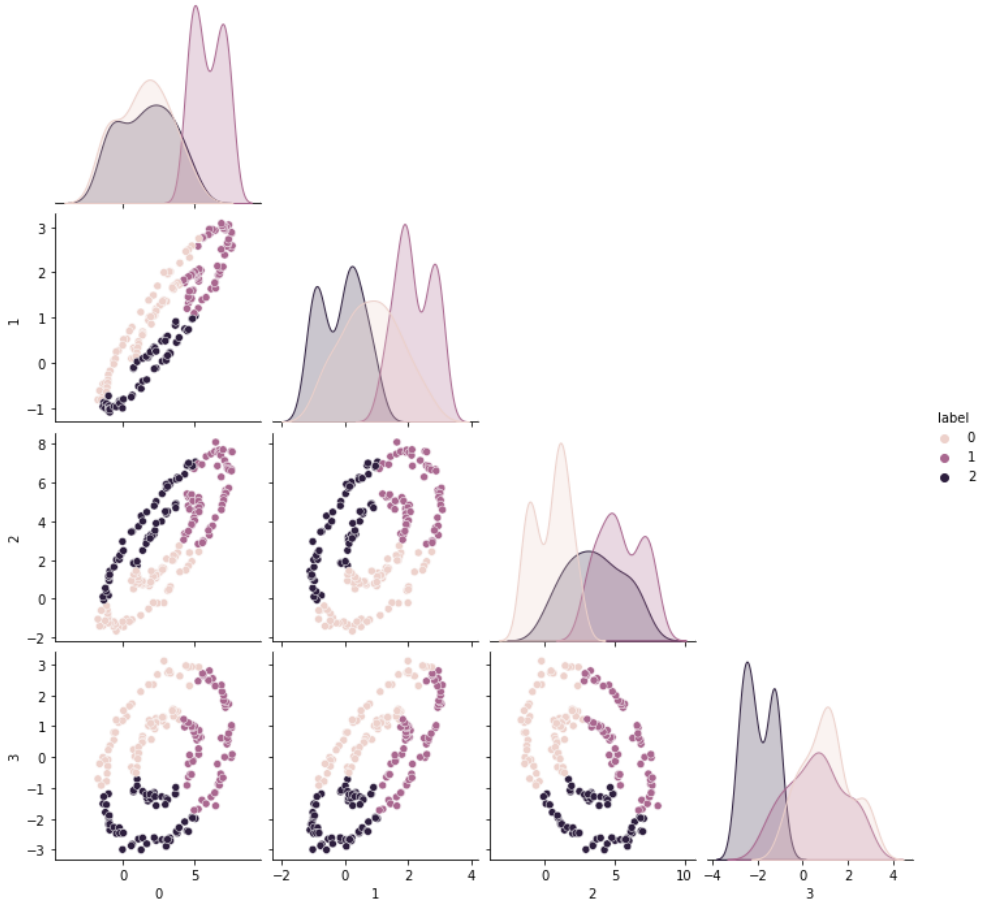
Набор Y разделился идеально!

EM-алгоритм (Gaussian mixture)

In [ ]:
```
labels["x"]["em"] = GaussianMixture(n_components=3, init_params="k-means++",
                                    random_state=random_state).fit_predict(df["x"])

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["em"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

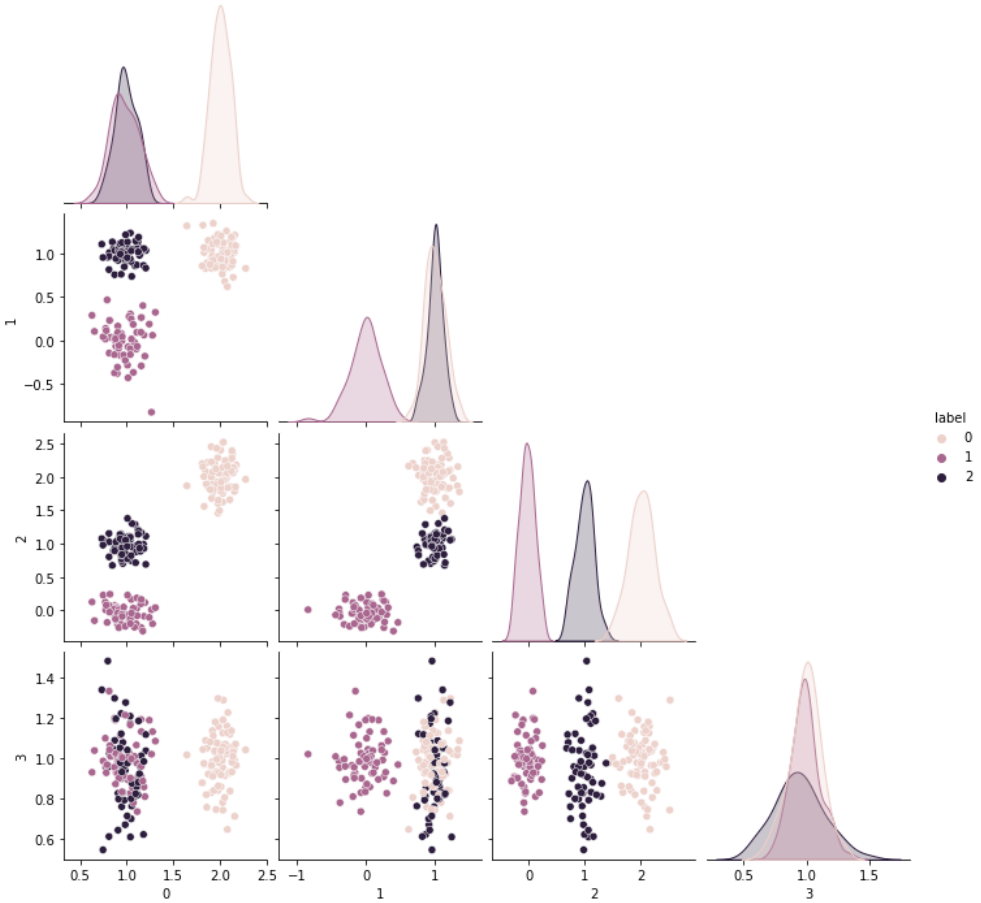
Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9039c7640>
```



In [ ]:
```
labels["y"]["em"] = GaussianMixture(n_components=3, init_params="k-means++",
                                    random_state=random_state).fit_predict(df["y"])

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["em"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9059a7e80>
```



Похожий алгоритм EM сработал совершенно аналогично. Визуально разбиения кажутся одинаковыми - точнее узнаем, когда сосчитаем метрики.

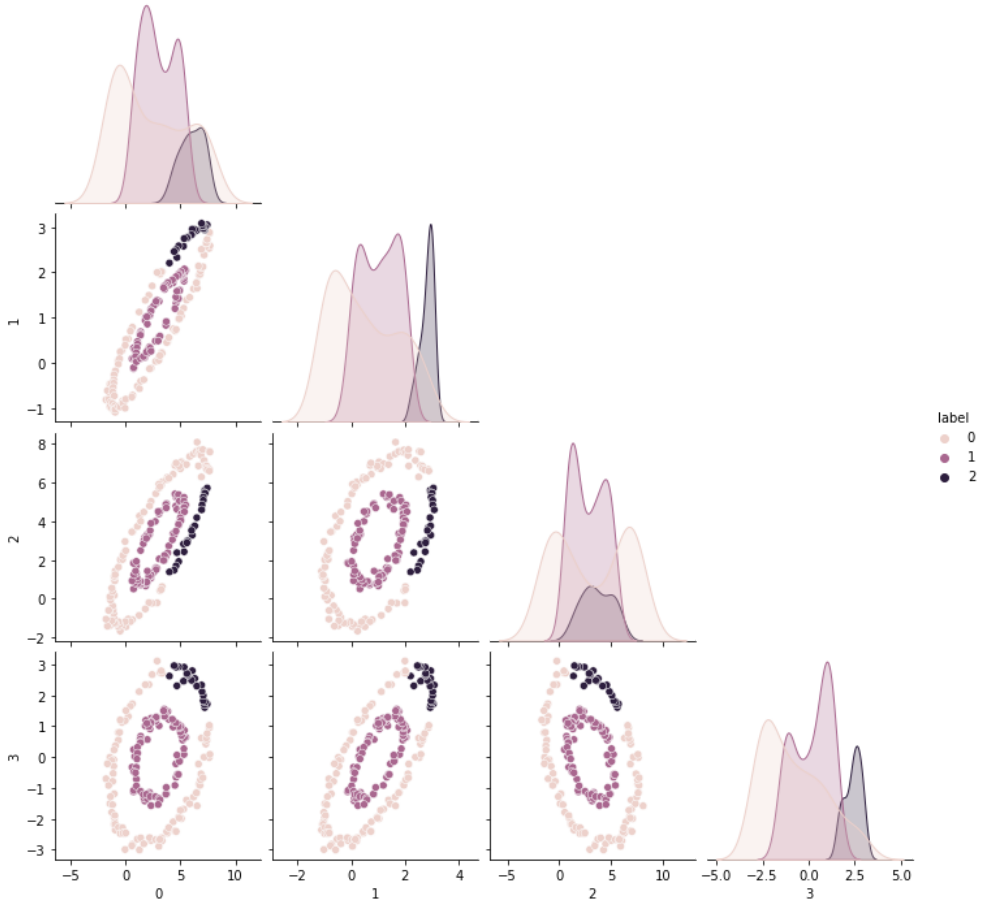
MST method

На этот раз ставим `cutoff=2` - нам необходимо 3 кластера, а значит 2 ребра

In [ ]:
```
labels["x"]["mst"] = MSTClustering(cutoff=2, approximate=False).fit_predict(df["x"])

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["mst"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb906419040>
```



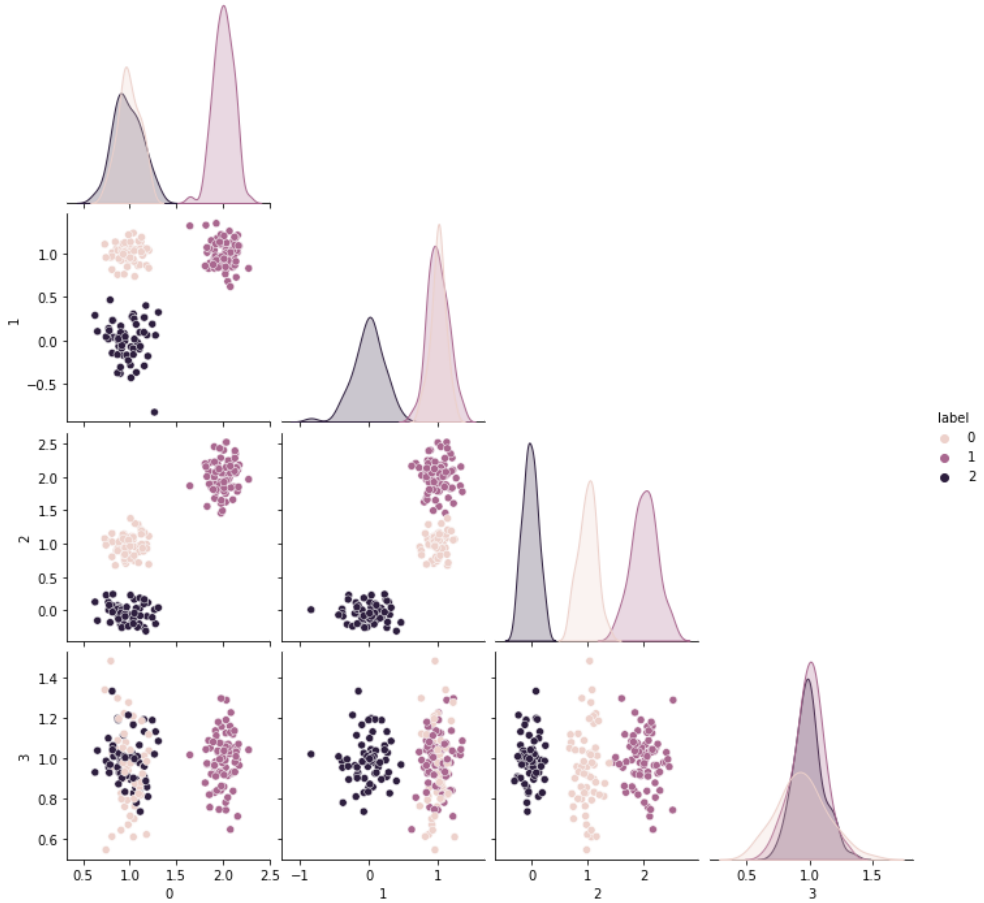
В двухкластерной структуре алгоритм правильно выделил кольца, но, из-за требования найти 3 кластера, ему пришлось оставить лишнюю связь, тем самым разбив внешнее кольцо на 2.

Смотрим на Y:

In [ ]:
```
labels["y"]["mst"] = MSTClustering(cutoff=2, approximate=False).fit_predict(df["y"])

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["mst"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb9063abb80>
```



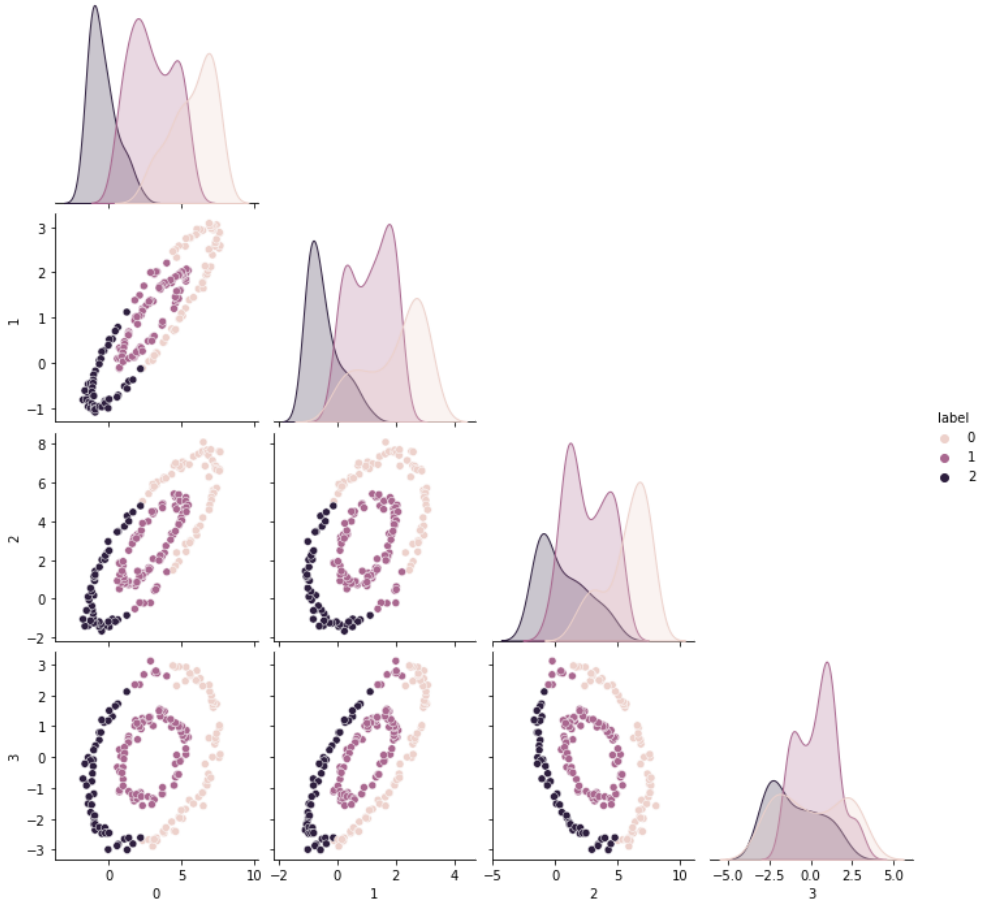
Идеально! С этим набором пока справляются все модели.

Spectral clustering

In [ ]:
```
spectral = SpectralClustering(n_clusters=3, affinity="nearest_neighbors",
                              random_state=random_state).fit(df["x"])
labels["x"]["spectral"] = spectral.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb90774d6a0>
```

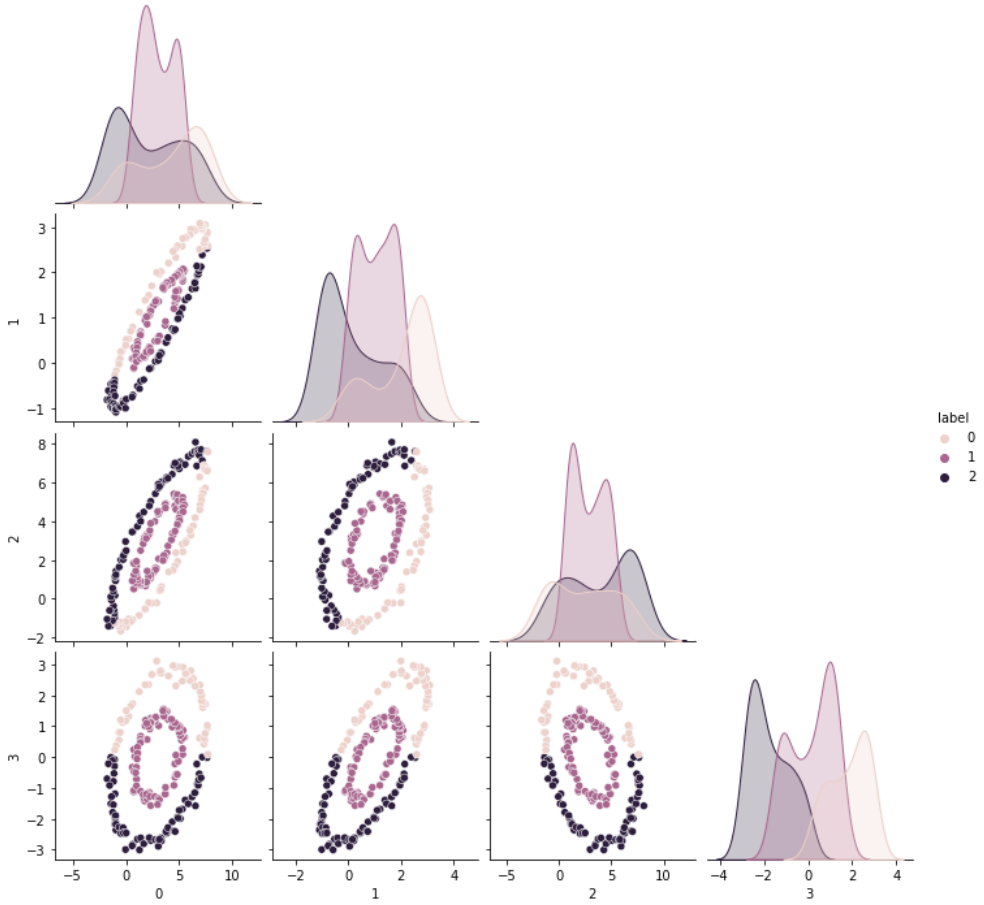


Снова неудача - спектральный алгоритм не смог выделить внутреннее кольцо. Попробуем понизить размерность:

In [ ]:
```
u, s, vt = np.linalg.svd(df["x"], full_matrices=False)
spectral = SpectralClustering(n_clusters=3, affinity="nearest_neighbors", random_state=random_state).fit(u[:,:2])
labels["x"]["spectral"] = spectral.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb906454df0>
```



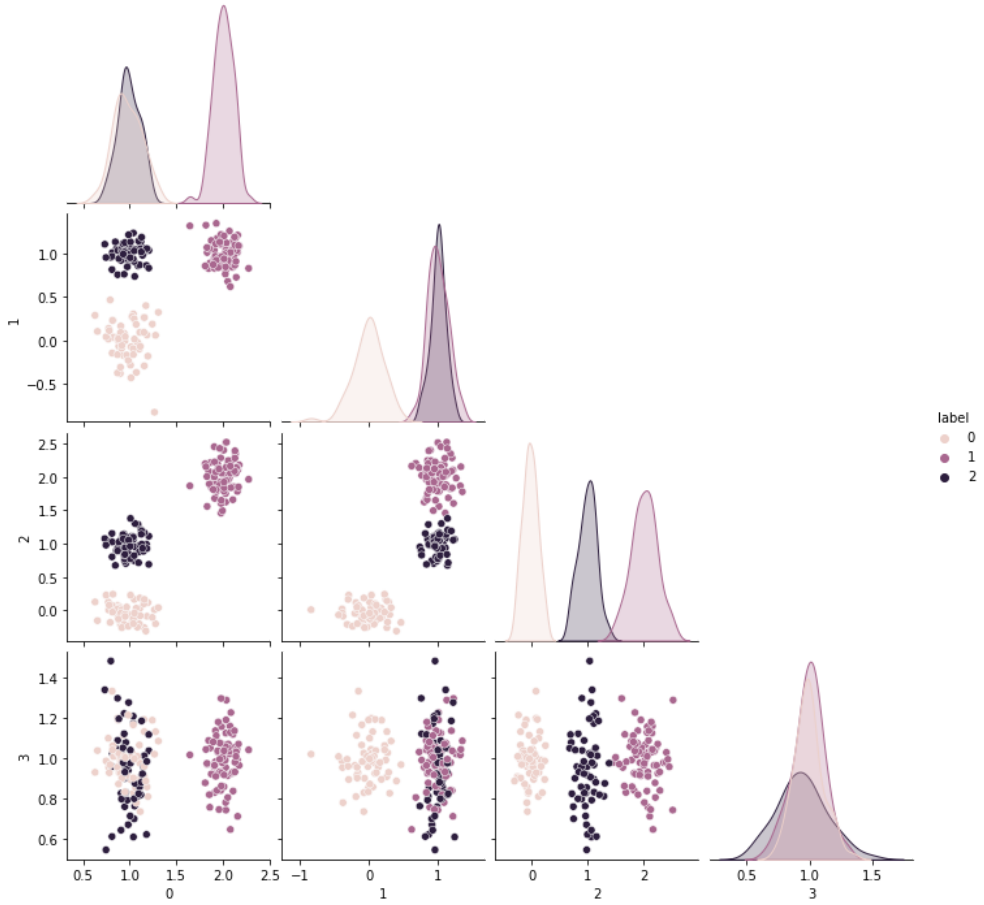
Здорово! Внутреннее кольцо найдено правильно, внешнее разбито на 2 равных части.

Перейдём к Y:

In [ ]:
```
spectral = SpectralClustering(n_clusters=3, random_state=random_state).fit(df["y"])
labels["y"]["spectral"] = spectral.labels_

plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["spectral"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8e99af5e0>
```



Отлично! Кластеризация удалась.

2.2, 2.3 RAND index и модулярность

Для более точных выводов снова посчитаем метрики:

In [ ]:
```
print(f'RAND index для X, K-means и EM: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["em"]):.3f}')
print(f'RAND index для Y, K-means и EM: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["em"]):.3f}\n')

print(f'RAND index для X, K-means и MST: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["mst"]):.3f}')
print(f'RAND index для Y, K-means и MST: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["mst"]):.3f}\n')

print(f'RAND index для X, K-means и Spectral: \
{compute_rand_index(labels["x"]["kmeans"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, K-means и Spectral: \
{compute_rand_index(labels["y"]["kmeans"], labels["y"]["spectral"]):.3f}\n')

print(f'RAND index для X, EM и MST: \
{compute_rand_index(labels["x"]["em"], labels["x"]["mst"]):.3f}')
print(f'RAND index для Y, EM и MST: \
{compute_rand_index(labels["y"]["em"], labels["y"]["mst"]):.3f}\n')

print(f'RAND index для X, EM и Spectral: \
{compute_rand_index(labels["x"]["em"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, EM и Spectral: \
{compute_rand_index(labels["y"]["em"], labels["y"]["spectral"]):.3f}\n')

print(f'RAND index для X, MST и Spectral: \
{compute_rand_index(labels["x"]["mst"], labels["x"]["spectral"]):.3f}')
print(f'RAND index для Y, MST и Spectral: \
{compute_rand_index(labels["y"]["mst"], labels["y"]["spectral"]):.3f}\n')
```

```
RAND index для X, K-means и EM: 0.858
RAND index для Y, K-means и EM: 1.000

RAND index для X, K-means и MST: 0.531
RAND index для Y, K-means и MST: 1.000

RAND index для X, K-means и Spectral: 0.565
RAND index для Y, K-means и Spectral: 1.000

RAND index для X, EM и MST: 0.549
RAND index для Y, EM и MST: 1.000

RAND index для X, EM и Spectral: 0.605
RAND index для Y, EM и Spectral: 1.000

RAND index для X, MST и Spectral: 0.872
RAND index для Y, MST и Spectral: 1.000
```

Интересный факт: набор Y был кластеризован всеми моделями **абсолютно одинаково**. Для X снова видим похожесть между EM и K-means, а также между MST и спектральным алгоритмом.

Найдём модулярности:

In [ ]:
```
print("X dataset:")
print(f'Modularity value for K-means clustering: \
{modularity(G_x, get_lables("x", "kmeans", 3)):.3f}')
print(f'Modularity value for EM clustering: \
{modularity(G_x, get_lables("x", "em", 3)):.3f}')
print(f'Modularity value for MST clustering: \
{modularity(G_x, get_lables("x", "mst", 3)):.3f}')
print(f'Modularity value for Spectral clustering: \
{modularity(G_x, get_lables("x", "spectral", 3)):.3f}')

print("\nY dataset:")
print(f'Modularity value for K-means clustering: \
{modularity(G_x, get_lables("y", "kmeans", 3)):.3f}')
print(f'Modularity value for EM clustering: \
{modularity(G_x, get_lables("y", "em", 3)):.3f}')
print(f'Modularity value for MST clustering: \
{modularity(G_x, get_lables("y", "mst", 3)):.3f}')
print(f'Modularity value for Spectral clustering: \
{modularity(G_x, get_lables("y", "spectral", 3)):.3f}')
```

```
X dataset:
Modularity value for K-means clustering: 0.104
Modularity value for EM clustering: 0.095
Modularity value for MST clustering: 0.031
Modularity value for Spectral clustering: 0.039

Y dataset:
Modularity value for K-means clustering: 0.008
Modularity value for EM clustering: 0.008
Modularity value for MST clustering: 0.008
Modularity value for Spectral clustering: 0.008
```

Снова видим аналогичную картину - модулярности набора Y ожидаемо совпадают, а для X эта метрика отдает предпочтение K-means и EM, хотя их кластерная структура кажется менее информативной.

2.4 Выводы

Все из использованных в эксперименте моделей одинаково хорошо показали себя на наборе Y. 3 кластера были идеально и эквивалентно найдены.

На двухкластерном наборе X хуже всех отработали модели K-means и EM - никакой полезной структуры не нашлось. Метод, основанный на минимальном остове смог выделить одно из колец и разделить внешнее. Спектральный алгоритм на исходных данных отработал плохо - внутреннее кольцо выделено, но вместе с ним в кластер попала и часть внешнего. Зато после понижения размерности до исходной картинка получилась самой интересной - центральное кольцо и две половинки.

Общий вывод по экспериментам

Первое место в эксперименте занимает спектральный алгоритм, при условии, что отсутствует линейная зависимость между признаками. Второе место с небольшим отрывом - алгоритм MST, хорошо работающий "из коробки". Более простые алгоритмы K-means и EM справились хуже всех, тем не менее они правильно разделили простой набор Y.

3. Общий случай

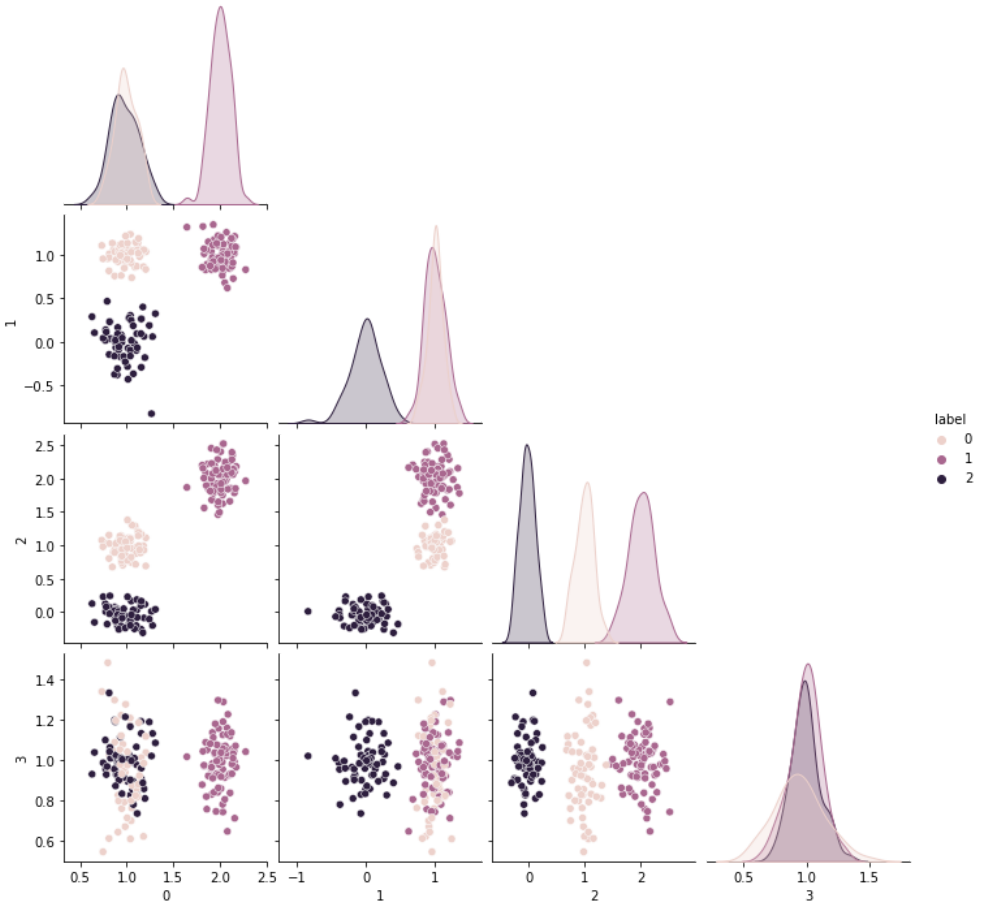
Алгоритм DBSCAN

DBSCAN - алгоритм поиска кластеров, основанный на плотности. Является очень популярным в кругу исследователей, в том числе потому, что самостоятельно пытается определить число кластеров в данных. Разработчики утверждают, что алгоритм хорошо работает, когда все кластеры имеют примерно одинаковую плотность распределения (чему как раз соответствует набор Y). Проверим этот алгоритм в реализации `scikit-learn` на наших данных:

In [ ]:
```
dbscan = DBSCAN(n_jobs=-1).fit(df["y"])
labels["y"]["dbscan"] = dbscan.labels_
```

```
plotting_df = df["y"].copy()
plotting_df["label"] = labels["y"]["dbscan"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]: <seaborn.axisgrid.PairGrid at 0x7fb906ac5ee0>

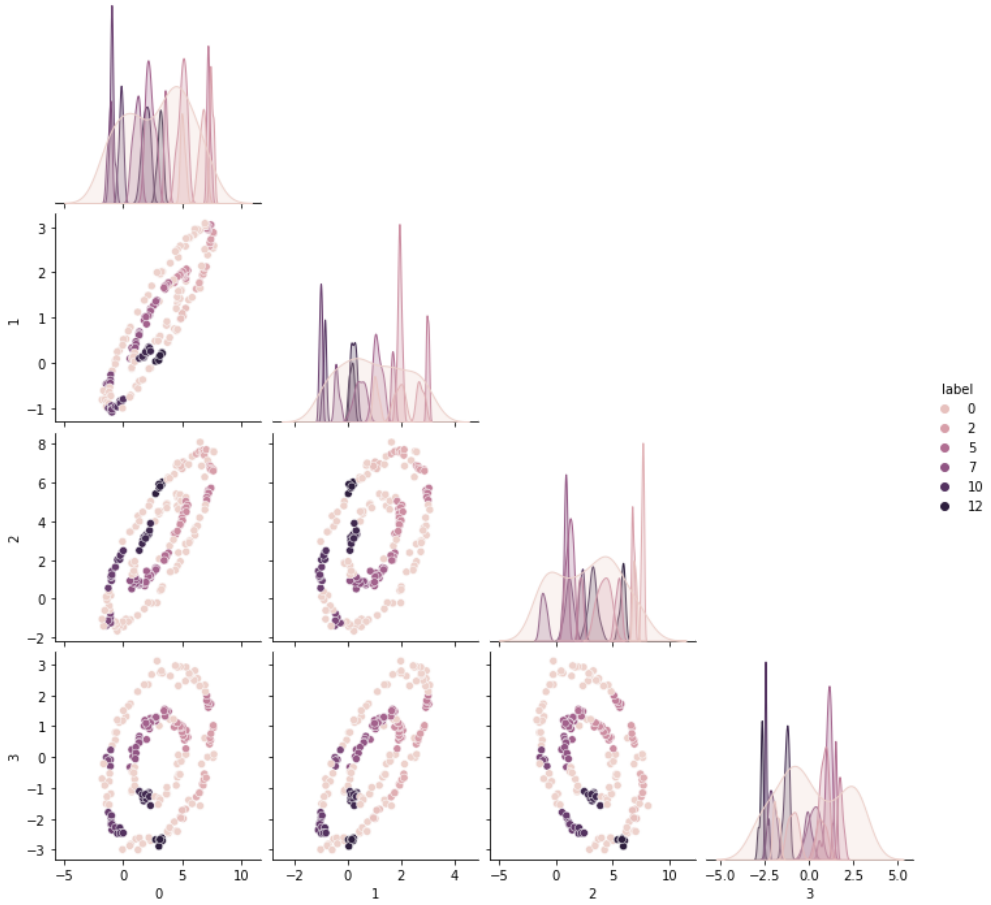


Датасет Y был правильно и успешно разделён на 3 кластера без дополнительных настроек, "из коробки".

In [ ]:
```
dbscan = DBSCAN(n_jobs=-1).fit(df["x"])
labels["x"]["dbscan"] = dbscan.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["dbscan"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]: <seaborn.axisgrid.PairGrid at 0x7fb8eb41df40>



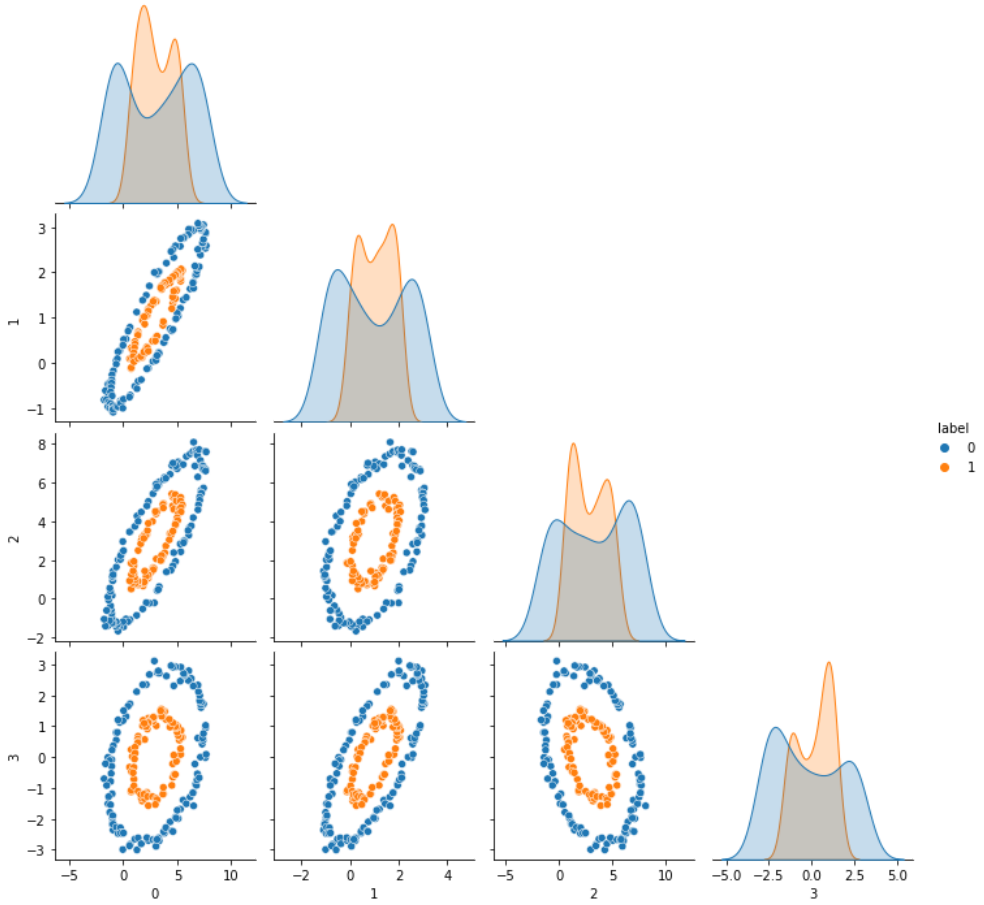
Без дополнительных настроек в наборе X было найдено целых 13 кластеров - не самое интересное разбиение.

Немного изменив основной параметр алгоритма (увеличиваем длину интервала, в котором для точки ищем соседей), получаем идеальное разбиение на 2 кластера:

In [ ]:
```
dbscan = DBSCAN(eps=1.25, n_jobs=-1).fit(df["x"])
labels["x"]["dbscan"] = dbscan.labels_

plotting_df = df["x"].copy()
plotting_df["label"] = labels["x"]["dbscan"]
sns.pairplot(plotting_df, corner=True, hue="label")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7fb8eb3ecbb0>
```



Индекс Дэвиса-Болдуина (Davies-Bouldin score)

Данная метрика является очень популярной при оценке кластеризации. Меньшее значение означает лучшее разбиение.

In [ ]:
```
print(db_score_2)
```

```
Подсчет Davies-Bouldin score для набора X (2 кластера):
K-means: 0.842
EM: 0.859
MST: 13.289
Spectral: 13.289

Подсчет Davies-Bouldin score для набора Y (2 кластера):
K-means: 0.549
EM: 0.549
MST: 0.523
Spectral: 0.523
```

In [ ]:
```
print("Подсчет Davies-Bouldin score для набора X (3 кластера):")
print(f'K-means: \
{davies_bouldin_score(df["x"], labels["x"]["kmeans"]):.3f}')
print(f'EM: \
{davies_bouldin_score(df["x"], labels["x"]["em"]):.3f}')
print(f'MST: \
{davies_bouldin_score(df["x"], labels["x"]["mst"]):.3f}')
print(f'Spectral: \
{davies_bouldin_score(df["x"], labels["x"]["spectral"]):.3f}')
print(f'\nDBSCAN: \
{davies_bouldin_score(df["x"], labels["x"]["dbscan"]):.3f}\n\
! Стоит отнести к верхней таблице, поскольку алгоритм находит 2 кластера !')

print("\nПодсчет Davies-Bouldin score для набора Y (3 кластера):")
print(f'K-means: \
{davies_bouldin_score(df["y"], labels["y"]["kmeans"]):.3f}')
print(f'EM: \
{davies_bouldin_score(df["y"], labels["y"]["em"]):.3f}')
print(f'MST: \
{davies_bouldin_score(df["y"], labels["y"]["mst"]):.3f}')
print(f'Spectral: \
{davies_bouldin_score(df["y"], labels["y"]["spectral"]):.3f}')
print(f'DBSCAN: \
{davies_bouldin_score(df["y"], labels["y"]["dbscan"]):.3f}')
```

```
Подсчет Davies-Bouldin score для набора X (3 кластера):
K-means: 0.958
EM: 1.022
MST: 4.056
Spectral: 2.752

DBSCAN: 13.289
! Стоит отнести к верхней таблице, поскольку алгоритм находит 2 кластера !

Подсчет Davies-Bouldin score для набора Y (3 кластера):
K-means: 0.395
EM: 0.395
MST: 0.395
Spectral: 0.395
DBSCAN: 0.395
```

И снова метрика нас подвела. На наборе Y она не представляет большого интереса - все алгоритмы делали примерно одинаковые по качеству (визуально) разбиения. А на наборе X метрика снова сделала выбор в пользу алгоритмов выбирающих "близкие группы".

Единственный плюс данной метрики - на наборе X при делении на 3 кластера она оценила спектральный алгоритм выше MST, что кажется визуально более правильным.

Общие выводы

- Было исследовано 5 различных методов кластеризации на двух наборах данных.
- Кластеризация - очень относительное и визуальное понятие. Не всегда можно точно сказать, как правильно делить на кластеры тот или иной набор данных.
- На простых и легко разделимых датасетах хорошо работают любые алгоритмы кластеризации.
- На более сложных данных замысловатой структуры лучше работают более сложные и современные алгоритмы.
- Спектральный метод кластеризации не всегда хорош "из коробки" и требует преобразования над данными.
- Тяжело найти хорошую метрику правильности кластеризации. Обе использованных в работе метрики дают неоднозначные результаты на "сложных" данных