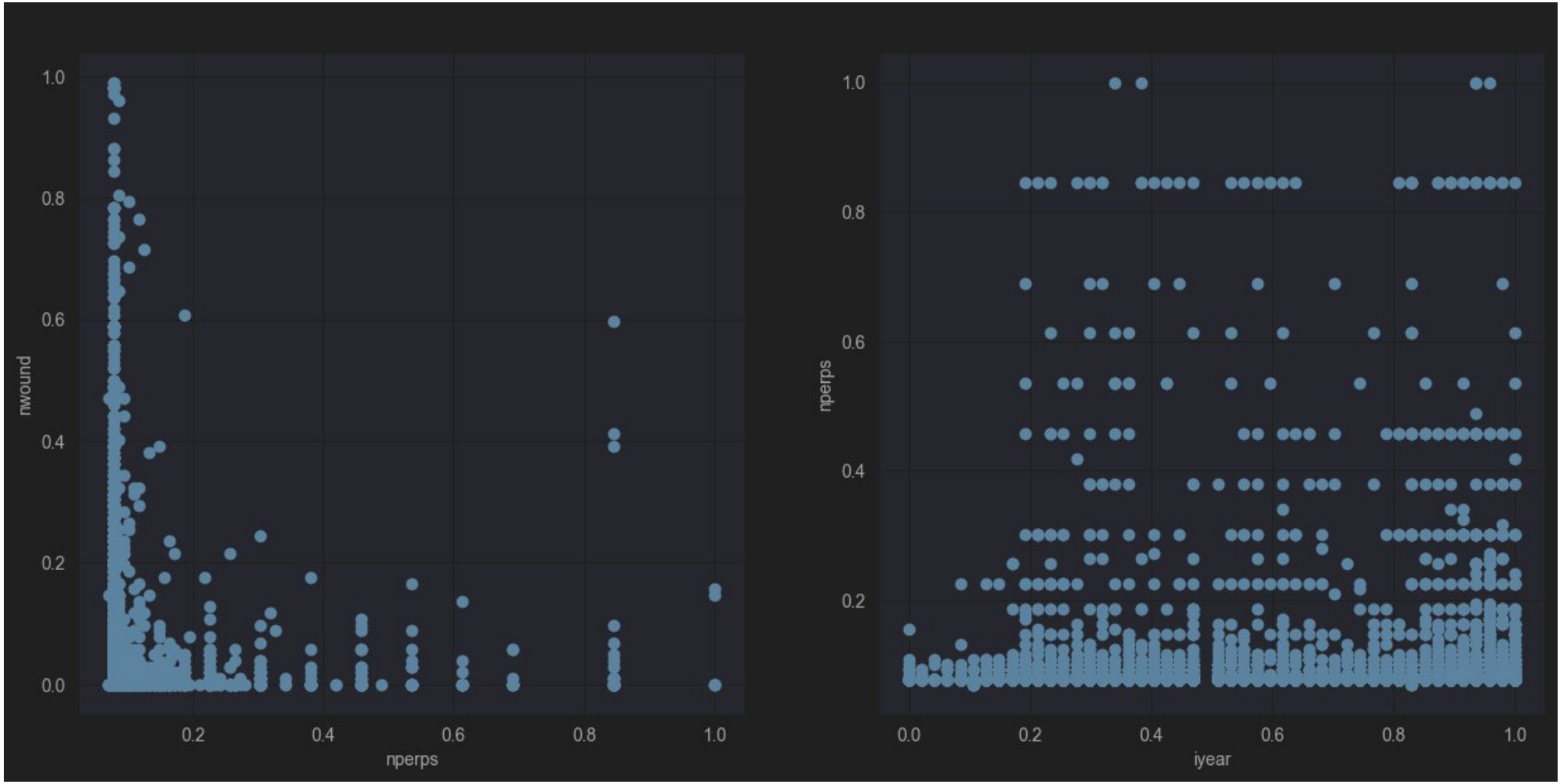
Кластеризация

Ссылка на код

Пименов Глеб

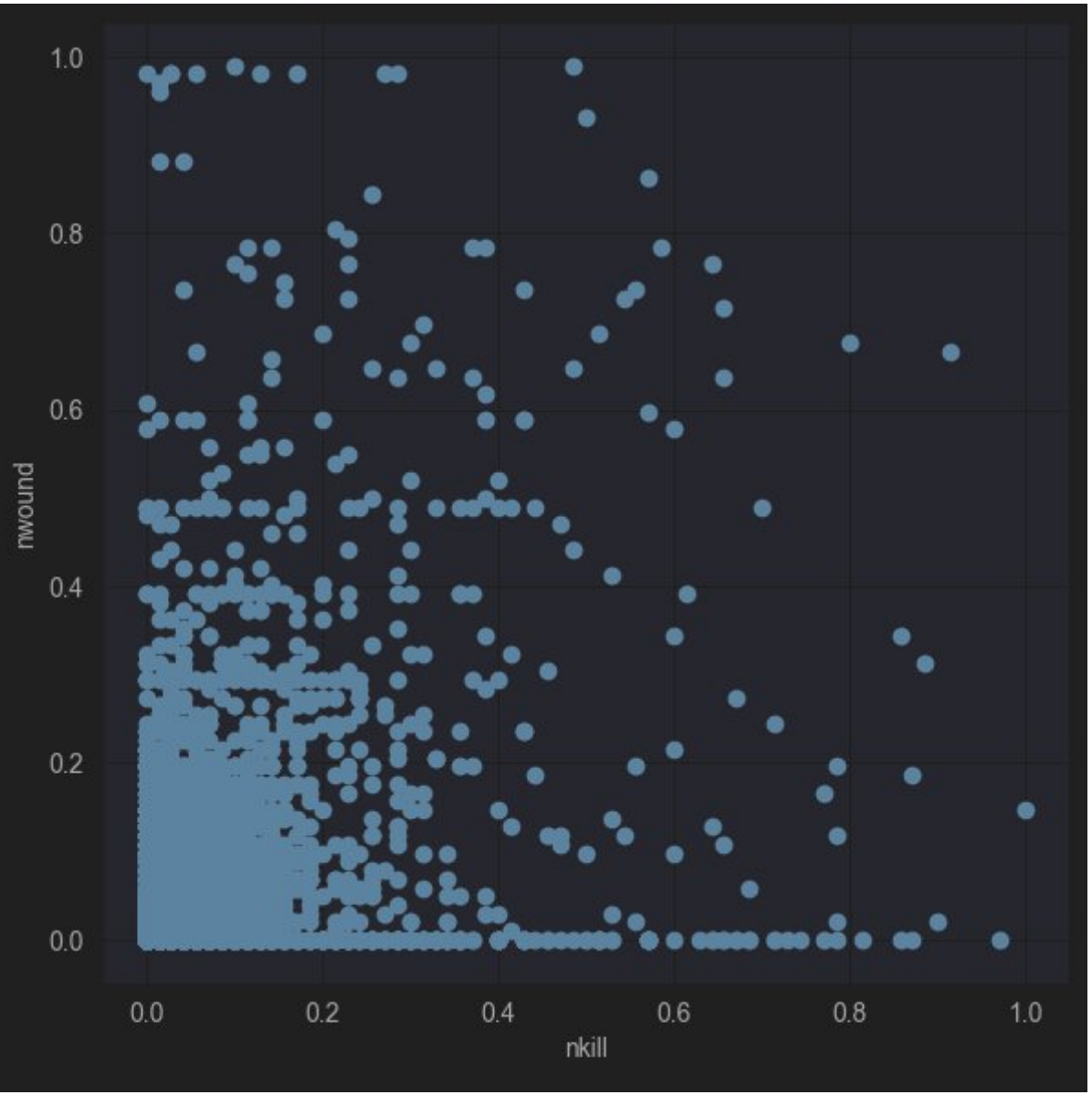
Санкт-Петербург
2023

Первый этап – предобработка данных



- Оставьте только несколько параметров, заранее посмотрите на категориальные классы в выборке
- Можно оставить только 2 параметра для визуализации, в случае успеха – расширить их число


1.0
0.8
0.6
0.4
0.2
nperps
0.0
0.2
0.4
0.6
0.8
1.0
nkill


1.0
0.8
0.6
0.4
0.2
0.0
nwound
0.0
0.2
0.4
0.6
0.8
1.0
nkill

Код для графиков рассеяния

```python
fig, ax = plt.subplots(2, 2, figsize=(15, 15))
# Первый график рассеяния: жертвы и террористы
ax[0][0].scatter(df["nkill"], df["nperps"])
ax[0][0].set_xlabel("nkill")
ax[0][0].set_ylabel("nperps")

# Второй график рассеяния: жертвы и раненые
ax[0][1].scatter(df["nkill"], df["nwound"])
ax[0][1].set_xlabel("nkill")
ax[0][1].set_ylabel("nwound")

# Второй график рассеяния: террористы и раненые
ax[1][0].scatter(df["nperps"], df["nwound"])
ax[1][0].set_xlabel("nperps")
ax[1][0].set_ylabel("nwound")

# Второй график рассеяния: года и террористы
ax[1][1].scatter(df["iyear"], df["nperps"])
ax[1][1].set_xlabel("iyear")
ax[1][1].set_ylabel("nperps")
plt.show()
```

Executed at 2023.11.10 02:05:48 in 1s 444ms

- Упрощение данных

```
df_two = df[["nkill", "nperps"]]
```
Executed at 2023.11.10 02:05:49 in 640ms

10% от выборки для ускорения работы

- Изначальные данные

```
df = pd.read_csv('../Data/global_preprocessed_without_onehot.csv', encoding='ISO-8859-1').sample(frac=0.1)
df.head()
```
Executed at 2023.11.10 02:05:47 in 252ms

5 rows | 5 rows × 11 columns pd.DataFrame | CSV

	iyear	extended	region	nkill	nwound	attacktype1	nperps	suicide	success	gname	targtype1
100665	0.893617	0	10	0.014286	0.039216	2	0.077519	0	1	2	14
138046	0.957447	0	6	0.000000	0.019608	3	0.077519	0	1	1972	3
92459	0.851064	0	6	0.000000	0.000000	3	0.077519	0	1	2	2
1225	0.042553	0	1	0.000000	0.009804	3	0.077519	0	1	42	1
13946	0.234043	0	2	0.128571	0.000000	2	0.077519	0	1	2	14

Второй этап – обучение моделей и подбор параметров

- Алгоритм KMeans

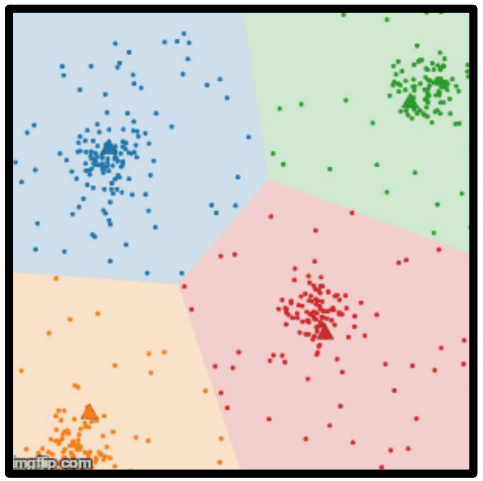
```
kmeans = KMeans(n_clusters=3, max_iter=300, tol=1e-4, n_init="auto")
kmeans.fit(df_two)
Executed at 2023.11.10 02:05:49 in 635ms
```

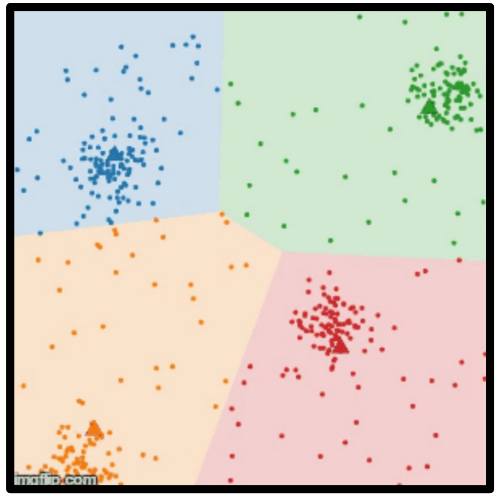
В контексте k-средних или других подобных алгоритмов, tol встречается в контексте сходимости центров кластеров. Если изменение местоположения центров кластеров после новой итерации ниже заданного уровня tol, алгоритм считается сбежавшимся и оптимальные центры кластеров найдены

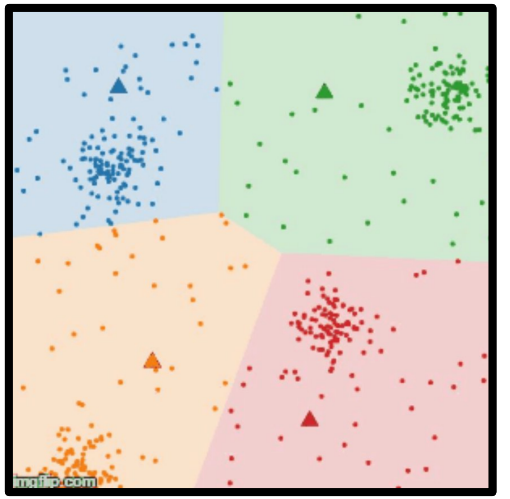
n_clusters - кол-во кластеров

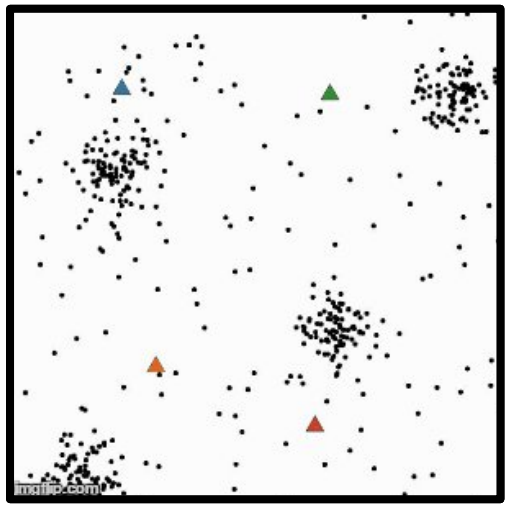
max_iter - максимальное кол-во итераций за один проход

n_init - кол-во запусков с разными центрами в начале









1. Начальное размещение центров кластеров произвольным образом.
2. Для каждого объекта данных вычисляется расстояние до каждого центра кластера.
3. Объекты данных присваиваются к ближайшим центрам кластеров.
4. После назначения всех объектов данных кластерам пересчитываются центры путем вычисления среднего значения координат объектов данных в каждом кластере.
5. Повторение последних двух шагов до тех пор, пока не будет достигнута сходимость или заданное количество итераций.

Parameters:

n_clusters : *int, default=8*

The number of clusters to form as well as the number of centroids to generate.

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 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` strategy, see the example entitled A demo of K-Means clustering on the handwritten digits data.

n_init : *'auto' or int, default=10*

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 Clustering sparse data with k-means).

When `n_init='auto'`, the 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.

New in version 1.2: Added 'auto' option for `n_init`.

max_iter : *int, default=300*

Maximum number of iterations of the k-means algorithm for a single run.

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.

verbose : *int, default=0*

Verbosity mode.

random_state : *int, RandomState instance or None, default=None*

Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See Glossary.

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.

Немного оптимизированная версия

algorithm : *{"lloyd", "elkan", "auto", "full"}, default="lloyd"*

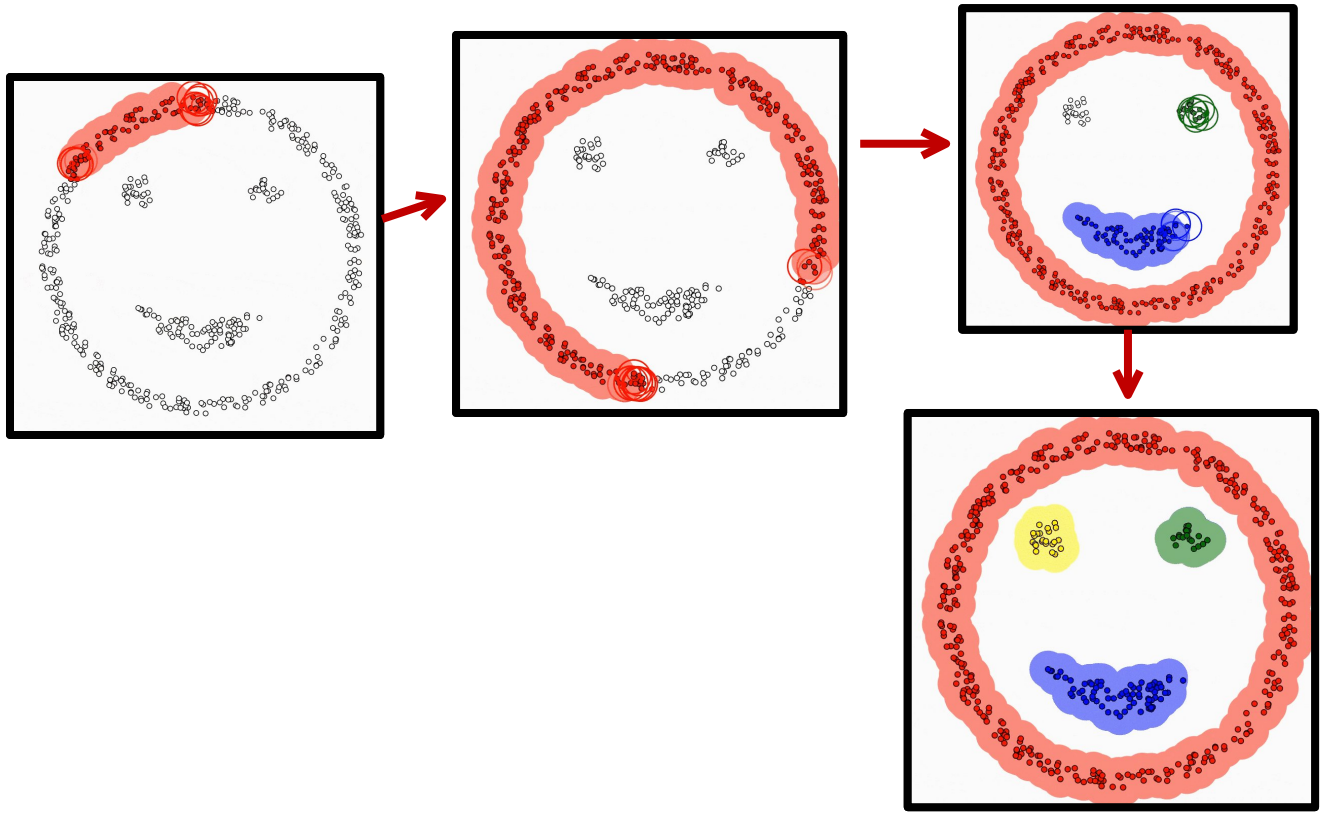
K-means algorithm to use. The classical EM-style algorithm is `"lloyd"`. The `"elkan"` variation can be more efficient on some datasets with well-defined clusters, by using the triangle inequality. However it's more memory intensive due to the allocation of an extra array of shape `(n_samples, n_clusters)`.

`"auto"` and `"full"` are deprecated and they will be removed in Scikit-Learn 1.3. They are both aliases for `"lloyd"`.

- Алгоритм DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

```
dbscan = DBSCAN(eps=0.1, min_samples=3, n_jobs=-1)
dbscan.fit(df_two)
```

n_jobs = -1 - это задействование всех процессоров, применимо для некоторых алгоритмов



1. Задается два параметра: радиус окрестности (eps) и минимальное количество точек в окрестности (min_samples).
2. Выбирается произвольная неразмеченная точка данных, которая еще не была рассмотрена.
3. Вычисляется расстояние между выбранной точкой и всеми другими точками в наборе данных.
4. Если число точек в окрестности данной точки (расстояние между ними меньше eps) больше или равно min_samples, то создается новый кластер, и все точки, находящиеся в его окрестности, добавляются в кластер. (если есть кластер, добавляются в существующий)
5. Если число точек в окрестности недостаточно велико, тогда рассматриваемая точка помечается как выброс (шум) и не добавляется в кластеры.
6. Процесс повторяется пока все точки не будут рассмотрены.
7. Если есть некоторые точки, которые остались нерассмотренными, они могут быть добавлены в кластеры, если они находятся в окрестности других точек, принадлежащих к кластеру.

Parameters:

eps : *float, default=0.5*

The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples : *int, default=5*

The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself. If `min_samples` is set to a higher value, DBSCAN will find denser clusters, whereas if it is set to a lower value, the found clusters will be more sparse.

metric : *str, or callable, default='euclidean'*

The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by `sklearn.metrics.pairwise_distances` for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

New in version 0.17: metric *precomputed* to accept precomputed sparse matrix.

metric_params : *dict, default=None*

Additional keyword arguments for the metric function.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size : *int, default=30*

Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : *float, default=None*

The power of the Minkowski metric to be used to calculate distance between points. If None, then `p=2` (equivalent to the Euclidean distance).

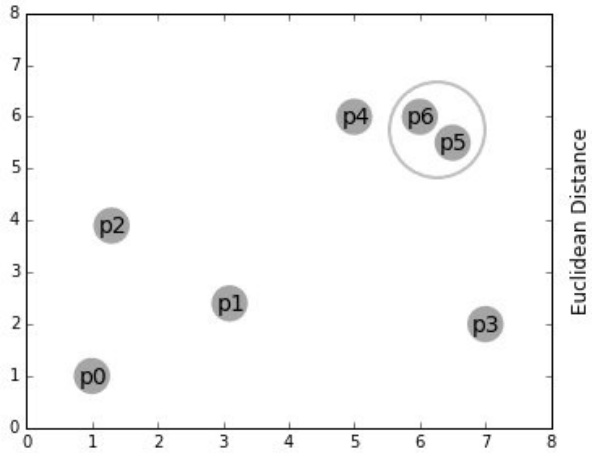
n_jobs : *int, default=None*

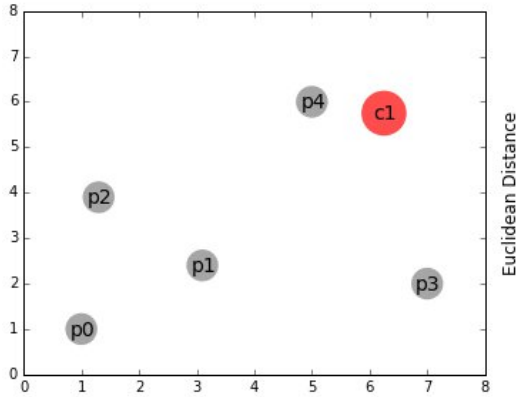
The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

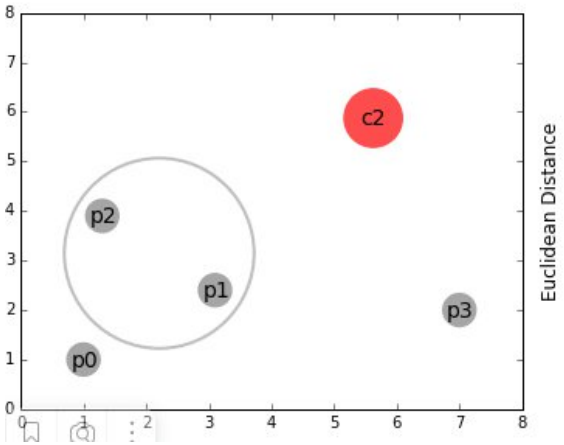
- Иерархическая кластеризация

```
h_clustering = AgglomerativeClustering(n_clusters=3)
h_clustering.fit(df_two)
```

connectivity=True/False - создаётся специальная матрица, чтобы кластера были непрерывными







1. Начало: Исходно каждый объект считается отдельным кластером.

2. Расстояние: Определите расстояние или схожесть между каждой парой объектов в выборке.

3. Объединение: Объедините два "ближайших" кластера в один, основываясь на рассчитанных расстояниях или сходствах. Метод объединения может быть агломеративным или дивизивным. В агломеративном подходе каждый объект начинает в отдельном кластере, а затем последовательно объединяется с другими близкими кластерами до тех пор, пока не получим один крупный кластер.

4. Обновление расстояний: Пересчитайте расстояния между новообразовавшимися кластерами и остальными объектами.

5. Повторение: Повторяйте шаги 3 и 4 до тех пор, пока не получите полное дерево кластеров или до того момента, когда выполнится определенное условие остановки (например, достижение определенного числа кластеров).

Parameters:

n_clusters : *int or None, default=2*

The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity : *str or callable, default='euclidean'*

The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by `sklearn.metrics.pairwise_distances` for its metric parameter. If linkage is "ward", only "euclidean" is accepted. If "precomputed", a distance matrix (instead of a similarity matrix) is needed as input for the fit method.

Deprecated since version 1.2: `affinity` was deprecated in version 1.2 and will be renamed to `metric` in 1.4.

metric : *str or callable, default=None*

Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or "precomputed". If set to `None` then "euclidean" is used. If linkage is "ward", only "euclidean" is accepted. If "precomputed", a distance matrix is needed as input for the fit method.

New in version 1.2.

memory : *str or object with the joblib.Memory interface, default=None*

Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

connectivity : *array-like or callable, default=None*

Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from `kneighbors_graph`. Default is `None`, i.e, the hierarchical clustering algorithm is unstructured.

compute_full_tree : *'auto' or bool, default='auto'*

Stop early the construction of the tree at `n_clusters`. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `True` if `distance_threshold` is not `None`. By default `compute_full_tree` is "auto", which is equivalent to `True` when `distance_threshold` is not `None` or that `n_clusters` is inferior to the maximum between 100 or `0.02 * n_samples`. Otherwise, "auto" is equivalent to `False`.

linkage : *{'ward', 'complete', 'average', 'single'}, default='ward'*

Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

- 'ward' minimizes the variance of the clusters being merged.
- 'average' uses the average of the distances of each observation of the two sets.
- 'complete' or 'maximum' linkage uses the maximum distances between all observations of the two sets.
- 'single' uses the minimum of the distances between all observations of the two sets.

New in version 0.20: Added the 'single' option

distance_threshold : *float, default=None*

The linkage distance threshold at or above which clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

New in version 0.21.

compute_distances : *bool, default=False*

Computes distances between clusters even if `distance_threshold` is not used. This can be used to make dendrogram visualization, but introduces a computational and memory overhead.

Третий этап – оценка моделей

- Добавления подбора гиперпараметров (GridSearch) + экспертная оценка

```
labels = algorithm.labels_
```

```
score = silhouette_score(df_two, labels)
```

Оценка silhouette_score представляет собой метрику, используемую для оценки качества кластеризации данных. Она измеряет, насколько объекты внутри одного кластера похожи друг на друга по сравнению с объектами из других кластеров.

s(i) = (b(i) - a(i)) / max(a(i), b(i))

Где:
s(i) - коэффициент силуэта для объекта i
a(i) - среднее расстояние между объектом i и всеми другими объектами в том же кластере (внутри-кластерное расстояние)
b(i) - среднее расстояние между объектом i и всеми объектами из ближайшего соседнего кластера (межкластерное расстояние)
Далее, для каждого объекта i, вычисляются значения s(i), а затем усредняются для получения общего значения silhouette_score для всего набора данных.

Значение близкое к 1 указывает на хорошую кластеризацию, где объекты внутри кластеров очень похожи друг на друга и значительно отличаются от объектов из других кластеров.
Значение близкое к 0 может указывать на наложение кластеров или на то, что объекты были неправильно отнесены к кластерам.
Значение близкое к -1 указывает на плохую кластеризацию, где объекты внутри кластеров сильно различаются, а объекты из разных кластеров похожи друг на друга.

- Пример перебора - обычный цикл по параметрам и сохранение оценки

```
silhouette_scores = []
k_values_used = []
kmeans_array = []
# Perform "Grid Search"
for k in k_values:
    print("K: " + str(k))
    kmeans_i = KMeans(n_clusters=k, random_state=0, n_init='auto', max_iter=100)
    kmeans_i.fit(df)
    score = silhouette_score(df, kmeans_i.labels_,  sample_size=1000)
    silhouette_scores.append(score)
    kmeans_array.append(kmeans_i)
    k_values_used.append(k)
```

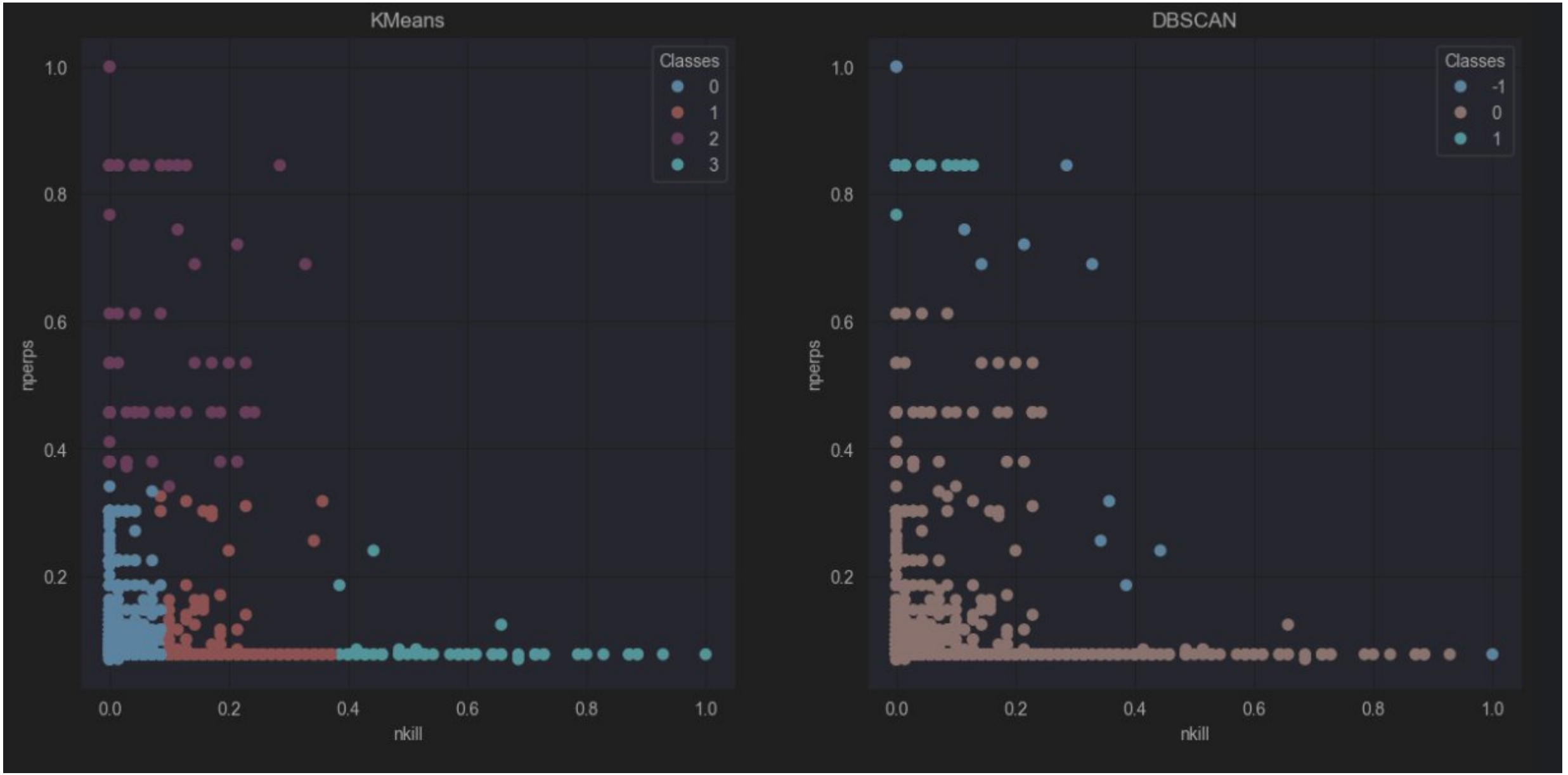
- Сохранение лучшего гиперпараметра и лучшей модели

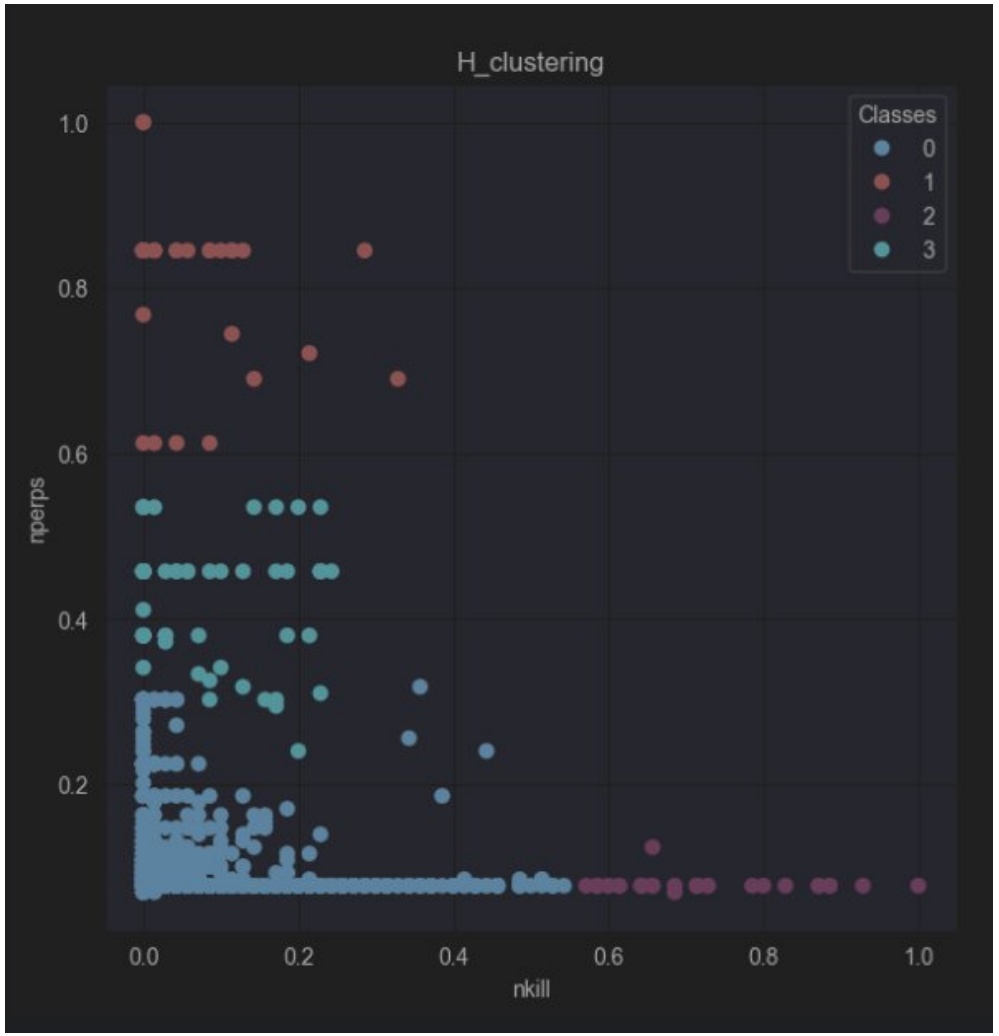
```
best_k = k_values_used[silhouette_scores.index(max(silhouette_scores))]
kmeans = kmeans_array[silhouette_scores.index(max(silhouette_scores))]
```

- Сохранение модели в файл

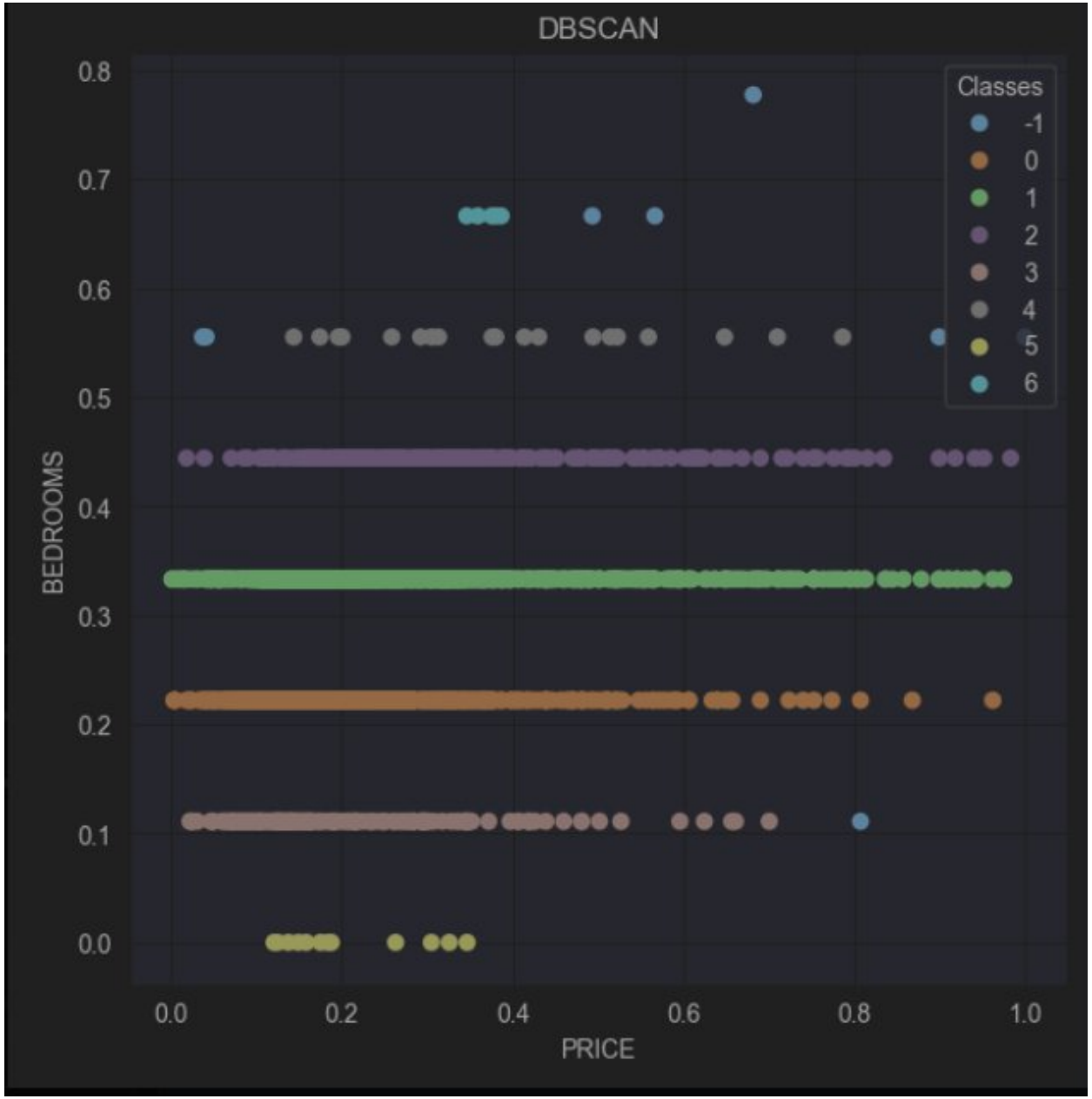
```
# Сохраняем лучшую модель
with open(f'best_{algorithm.__class__.__name__}_model.pkl', 'wb') as f:
    pickle.dump(best_model, f)
```

- Визуализация полученных значений





Более наглядный пример для другого датасета



- Код для выведения диаграмм рассеивания с легендой

```python
def prepare_and_plot(ax, model, title):
    labels = np.unique(model.labels_)  # получаем список уникальных меток
    colors = plt.cm.tab10(np.linspace(0, 1, len(labels)))  # создаем массив цветов
    ax.scatter(df["nkill"], df["nperps"], c=model.labels_, cmap='tab10')  # указываем нашу карту цветов
    # Создаем легенду в виде списка с данными о маркерах, цветах и метках
    handle_list = [plt.plot([], marker="o", ls="", color=color)[0] for color in colors]
    ax.legend(handle_list, labels, title='Classes')
    ax.set_xlabel("nkill")
    ax.set_ylabel("nperps")
    ax.set_title(title)

fig, ax = plt.subplots(2, 2, figsize=(15, 15))

prepare_and_plot(ax[0][0], kmeans, "KMeans")
prepare_and_plot(ax[0][1], dbscan, "DBSCAN")
prepare_and_plot(ax[1][0], h_clustering, "H_clustering")

plt.show()
```

Создаём цвета

plot поверх plot a

Сама легенда

- Сравнение разбиения на кластеры с помощью кластеризации с реальными

```
df_global = pd.read_csv('../Data/global_preprocessed_without_onehot_and_norm.csv', encoding='ISO-8859-1').sample(frac=fraction, random_state=0)
```
Executed at 2023.11.13 23:10:09 in 146ms

Выводим кластеры KMeans

```
df_KMeans = df_global.copy()[["nkill", "nperps"]]
df_KMeans["labels"] = kmeans.labels_
df_KMeans.groupby("labels").mean()
```
Executed at 2023.11.13 23:15:47 in 35ms

Желательно иметь идентичный датасет без нормализации

4 rows × 2 columns pd.DataFrame ↗ CSV

labels	nkill	nperps
0	0.801272	1.380025
2	4.138298	53.723404
1	11.337004	1.755507
3	37.175824	3.241758

Выводим средние значения по классам
(можно попробовать вывести по медиане)

- Можно также добавить к общей картине кластеризации диаграмму с существующими классами

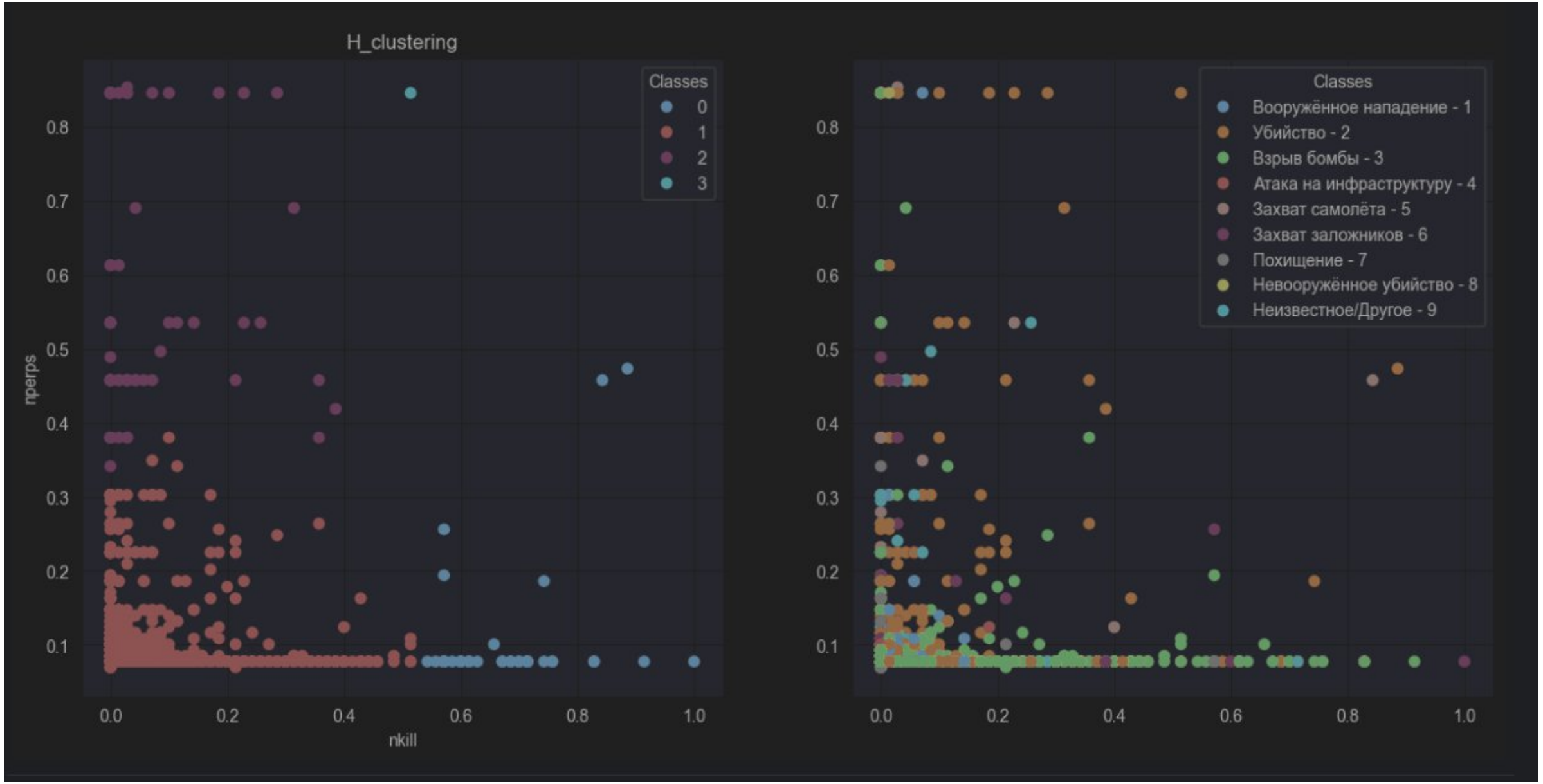
```
# Стандартные результаты кластеризации
fig, ax = plt.subplots(2, 2, figsize=(15, 15))

prepare_and_plot(ax[0][0], kmeans, "KMeans")
prepare_and_plot(ax[0][1], dbscan, "DBSCAN")
prepare_and_plot(ax[1][0], h_clustering, "H_clustering")

# Добавление диаграммы с известными классами
labels = ["Вооружённое нападение - 1", "Убийство - 2", "Взрыв бомбы - 3", "Атака на инфраструктуру - 4",
"Захват самолёта - 5", "Захват заложников - 6", "Похищение - 7", "Невооружённое убийство - 8",
"Неизвестное/Другое - 9"]  # получаем список уникальных меток
colors = plt.cm.tab10(np.linspace(0, 1, len(labels)))    # создаем массив цветов
ax[1][1].scatter(df["nkill"], df["nperps"], c=df_global['attacktype1'], cmap='tab10')  # указываем нашу карту цветов
# Создаем легенду в виде списка с данными о маркерах, цветах и метках
handle_list = [ax[1][1].plot([], marker="o", ls="", color=color)[0] for color in colors]
ax[1][1].legend(handle_list, labels, title='Classes')
plt.show()
```

Executed at 2023.11.13 23:49:28 in 1s 855ms

- Пример сравнения



На этом всё