

Кластеризация

В этом домашнем задании предлагается реализовать три различных метода кластеризации, понять, в каких случаях стоит применять те или иные методы, а также применить один из алгоритмов к задаче уменьшения цветности изображения.

In [31]:

```
import inspect
import itertools
import random
from abc import ABC
from typing import Callable, Dict, Tuple

import matplotlib.pyplot as plt
import numpy as np
from numpy import ndarray
from PIL import Image
from sklearn.datasets import make_blobs, make_moons
from sklearn.neighbors import KDTree
```

In [32]:

```
def set_seed(seed=42):
    np.random.seed(seed)
    random.seed(seed)

# Этой функцией будут помечены все места, которые необходимо дозаполнить
# Это могут быть как целые функции, так и отдельные части внутри них
# Всегда можно воспользоваться интроспекцией и найти места использования
def todo():
    stack = inspect.stack()
    caller_frame = stack[1]
    function_name = caller_frame.function
    line_number = caller_frame.lineno
    raise NotImplementedError(f"TODO at {function_name}, line {line_number}")

SEED = 0xC0FFEE
set_seed(SEED)
```

In []:

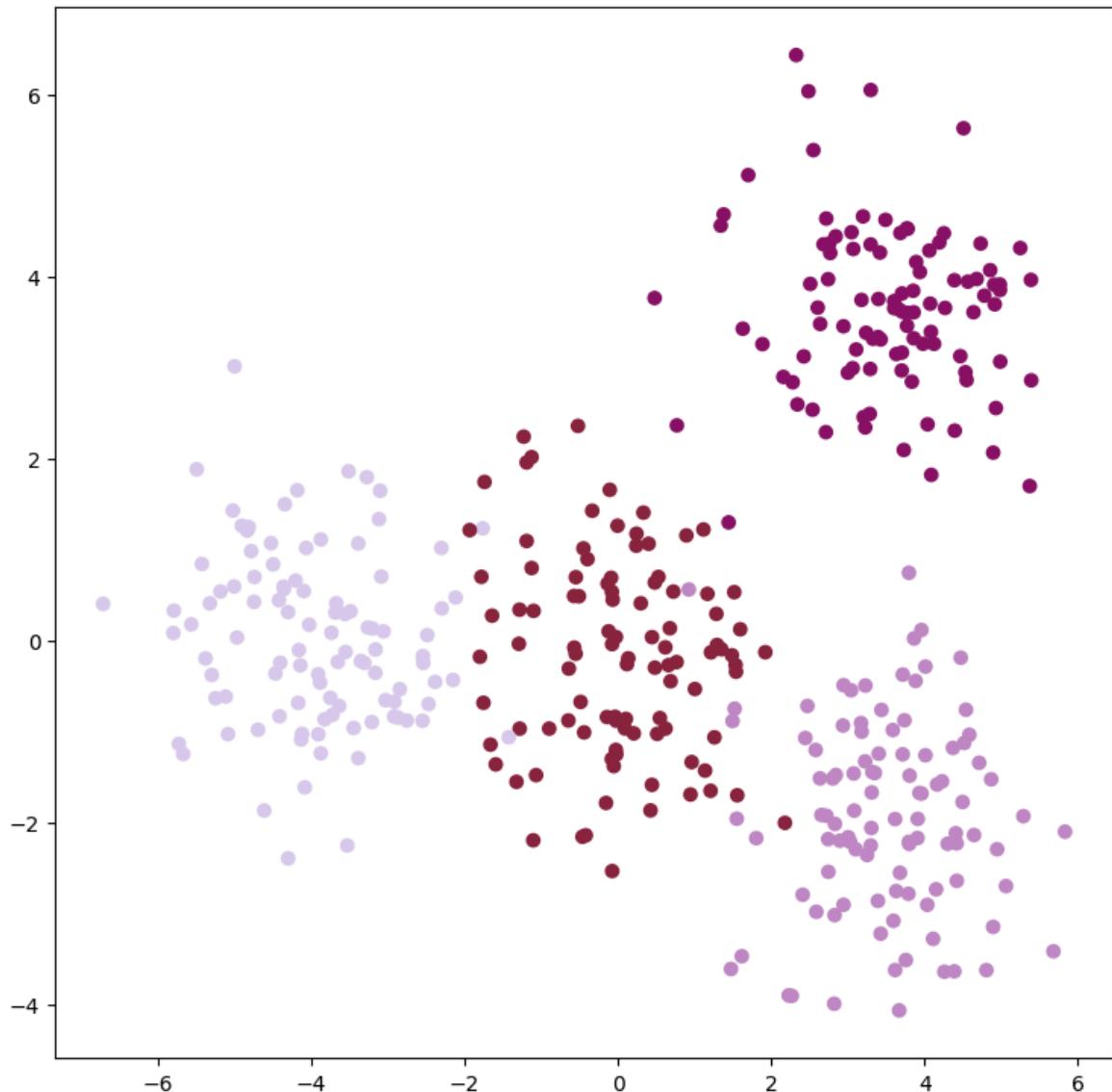
```
def visualize_clusters(X, labels):
    unique_labels = np.unique(labels)
    unique_colors = np.random.random((len(unique_labels), 3))
    colors = [unique_colors[l] for l in labels]
    plt.figure(figsize=(9, 9))
    plt.scatter(X[:, 0], X[:, 1], c=colors)
    plt.show()

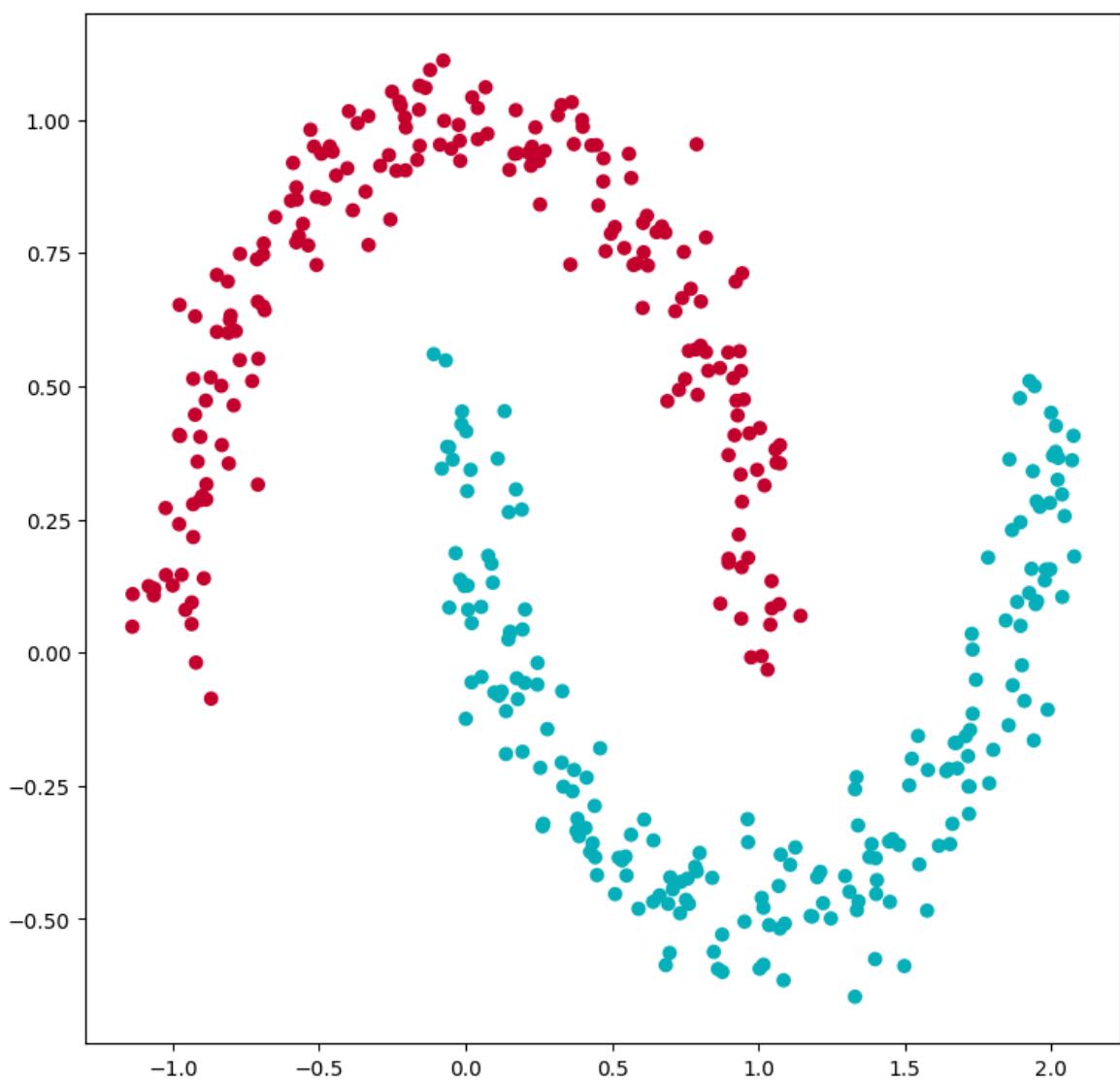
def visualize_clusters_on_ax(X, labels, ax, title=""):
    unique_labels = np.unique(labels)
    unique_colors = np.random.random((len(unique_labels), 3))
    colors = [unique_colors[l] for l in labels]
    ax.scatter(X[:, 0], X[:, 1], c=colors)
    if title:
        ax.set_title(title, )
```

```
def clusters_statistics(flatten_image, cluster_colors, cluster_labels):
    fig, axes = plt.subplots(3, 2, figsize=(12, 16))
    for remove_color in range(3):
        axes_pair = axes[remove_color]
        first_color = 0 if remove_color != 0 else 2
        second_color = 1 if remove_color != 1 else 2
        axes_pair[0].scatter(
            [p[first_color] for p in flatten_image],
            [p[second_color] for p in flatten_image],
            c=flatten_image,
            marker=".",
        )
        axes_pair[1].scatter(
            [p[first_color] for p in flatten_image],
            [p[second_color] for p in flatten_image],
            c=[cluster_colors[c] for c in cluster_labels],
            marker=".",
        )
    for a in axes_pair:
        a.set_xlim(0, 1)
        a.set_ylim(0, 1)
    plt.show()
```

Генерируем два синтетических набора данных для кластеризации. Далее будем тестировать наши алгоритмы на них.

```
In [34]: X_1, true_labels = make_blobs(400, 2, centers=[[0, 0], [-4, 0], [3.5, 3.5]])
visualize_clusters(X_1, true_labels)
X_2, true_labels = make_moons(400, noise=0.075)
visualize_clusters(X_2, true_labels)
```





Также реализуем необходимые функции для вычисления расстояний между точками

```
In [35]: def get_euclidean_distances(from_points: ndarray, to_points: ndarray) ->
    """Calculate euclidian distance between points

Args:
    from_points: array of [source_points; dim] points
    to_points: array of [target_points; dim] points

Returns:
    distances: array of [source_points; target_points]
"""
if from_points.ndim == 1:
    from_points = from_points.reshape(1, -1)
if to_points.ndim == 1:
    to_points = to_points.reshape(1, -1)

if from_points.shape[1] != to_points.shape[1]:
    raise ValueError(f"Dimension mismatch: from_points has dimension {from_points.shape[1]} and to_points has dimension {to_points.shape[1]}")

from_points_reshaped = from_points[:, np.newaxis]
to_points_reshaped = to_points[np.newaxis, :]
```

```

diff = from_points_reshaped - to_points_reshaped

return np.linalg.norm(diff, axis=2)

def get_manhattan_distances(from_points: ndarray, to_points: ndarray) ->
    """Calculate manhattan distance between points

Args:
    from_points: array of [source_points; dim] points
    to_points: array of [target_points; dim] points

Returns:
    distances: array of [source_points; target_points]
"""
if from_points.ndim == 1:
    from_points = from_points.reshape(1, -1)
if to_points.ndim == 1:
    to_points = to_points.reshape(1, -1)

if from_points.shape[1] != to_points.shape[1]:
    raise ValueError(f"Dimension mismatch: from_points has dimension {from_points.shape[1]} and to_points has dimension {to_points.shape[1]}")

from_points_reshaped = from_points[:, np.newaxis, :]
to_points_reshaped = to_points[np.newaxis, :, :]

abs_diff = np.abs(from_points_reshaped - to_points_reshaped)

sum_diff = np.sum(abs_diff, axis=2)

return sum_diff

def get_chebyshev_distance(from_points: ndarray, to_points: ndarray) ->
    """Calculate chebyshev distance between points

Args:
    from_points: array of [source_points; dim] points
    to_points: array of [target_points; dim] points

Returns:
    distances: array of [source_points; target_points]
"""
if from_points.ndim == 1:
    from_points = from_points.reshape(1, -1)
if to_points.ndim == 1:
    to_points = to_points.reshape(1, -1)

if from_points.shape[1] != to_points.shape[1]:
    raise ValueError(f"Dimension mismatch: from_points has dimension {from_points.shape[1]} and to_points has dimension {to_points.shape[1]}")

from_points_reshaped = from_points[:, np.newaxis, :]
to_points_reshaped = to_points[np.newaxis, :, :]

abs_diff = np.abs(from_points_reshaped - to_points_reshaped)
max_diff = np.max(abs_diff, axis=2)

return max_diff

```

K-Means (3 балла)

Первый метод, который предлагается реализовать - метод К средних.

Описание методов

`fit(X, y=None)` ищет и запоминает в `self.centroids` центроиды кластеров для набора данных.

`predict(X)` для каждого элемента из `X` возвращает номер кластера, к которому относится данный элемент.

Инициализация кластеров

Есть несколько вариантов инициализации кластеров. Нужно реализовать их все:

1. `random` - центроиды кластеров являются случайными точками
2. `sample` - центроиды кластеров выбираются случайно из набора данных
3. `k-means++` - центроиды кластеров инициализируются при помощи метода K-means++

Не забудьте реинициализировать пустые кластеры!

```
In [36]: class AbstractCentroidBuilder(ABC):
    name: str = None

    def _generate_centroids(self, x: ndarray, k: int) -> ndarray:
        """Generating centroids from points

        Args:
            x: array of [n_points; dim] points
            k: number of centroids to generate
        """
        raise NotImplementedError()

    @staticmethod
    def _validate_centroids(x: ndarray, centroids: ndarray) -> bool:
        """Validating that each centroid has at least one point next to it
        k = centroids.shape[0]
        distances = get_euclidean_distances(x, centroids)

        closest_centroid_indices = np.argmin(distances, axis=1)
        unique_assigned_centroids = np.unique(closest_centroid_indices)

        return len(unique_assigned_centroids) == k

    def build(self, x: ndarray, k: int) -> ndarray:
        if k <= 0:
            raise ValueError("Number of centroids (k) must be positive.")

        centroids = self._generate_centroids(x, k)
        while not self._validate_centroids(x, centroids):
```

```
        centroids = self._generate_centroids(x, k)
    return centroids

class RandomCentroidBuilder(AbstractCentroidBuilder):
    name: str = "random"

    def _generate_centroids(self, x: ndarray, k: int) -> ndarray:
        dim = x.shape[1]

        min_vals = np.min(x, axis=0)
        max_vals = np.max(x, axis=0)

        centroids = np.empty((k, dim), dtype=x.dtype)
        for i in range(dim):
            centroids[:, i] = np.random.uniform(min_vals[i], max_vals[i],)

        return centroids

class SampleCentroidBuilder(AbstractCentroidBuilder):
    name: str = "sample"

    def _generate_centroids(self, x: ndarray, k: int) -> ndarray:
        n_points = x.shape[0]
        if k > n_points:
            raise ValueError(
                f"Cannot sample {k} unique centroids from {n_points} data"
            )

        random_indices = np.random.choice(n_points, size=k, replace=False)

        centroids = x[random_indices]
        return centroids

class KMeansPlusCentroidBuilder(AbstractCentroidBuilder):
    name: str = "kmeans++"

    def _generate_centroids(self, x: ndarray, k: int) -> ndarray:
        n_points, dim = x.shape
        if k > n_points:
            raise ValueError(
                f"Cannot sample {k} unique centroids from {n_points} data"
            )

        centroids = np.empty((k, dim), dtype=x.dtype)
        first_centroid_idx = np.random.randint(n_points)
        centroids[0] = x[first_centroid_idx]

        for i in range(1, k):
            distances_to_all_chosen = get_euclidean_distances(x, centroid

            min_distances_to_a_chosen_centroid = np.min(distances_to_all_
d_sq_values = min_distances_to_a_chosen_centroid ** 2

            sum_d_sq = np.sum(d_sq_values)
```

```

    if sum_d_sq == 0:
        next_centroid_idx_in_x = np.random.randint(n_points)
    else:
        probabilities = d_sq_values / sum_d_sq
        next_centroid_idx_in_x = np.random.choice(n_points, p=probabilities)

    centroids[i] = x[next_centroid_idx_in_x]

return centroids

```

```

In [37]: class KMeans:
    _known_centroid_builders: Dict[str, AbstractCentroidBuilder.__class__] = {
        RandomCentroidBuilder.name: RandomCentroidBuilder,
        SampleCentroidBuilder.name: SampleCentroidBuilder,
        KMeansPlusCentroidBuilder.name: KMeansPlusCentroidBuilder,
    }

    _centroids: ndarray = None
    _x: ndarray = None
    _clusters_ids: ndarray = None

    def __init__(self, n_clusters: int, init: str = "random", max_iter: int = 100):
        """KMeans clusterization

        Args:
            n_clusters: number of clusters
            init: strategy to initialize clusters, one of "random", "sample"
            max_iter: maximum number of iterations
        """
        self._n_clusters = n_clusters
        self._max_iter = max_iter
        if init not in self._known_centroid_builders:
            raise ValueError(f"Unknown way to initialize clusters centroid builder")
        self._centroid_builder = self._known_centroid_builders[init]()

    def fit(self, x: ndarray, y: ndarray = None):
        """Build centroids based on X

        Args:
            x: input points, array of [n_points; dim]
            y: ignored, to match sklearn behaviour
        """
        if x.shape[0] < self._n_clusters:
            raise ValueError(f"Number of samples ({x.shape[0]}) must be > n_clusters")

        self._x = x
        n_points, dim = x.shape
        current_centroids = self._centroid_builder.build(x, self._n_clusters)

        prev_cluster_ids = np.full(n_points, -1, dtype=int)

        for i in range(self._max_iter):
            cluster_ids = np.argmin(get_euclidean_distances(x, current_centroids), axis=0)

            new_centroids = np.empty_like(current_centroids)
            for k in range(self._n_clusters):
                points_in_cluster_k = x[cluster_ids == k]

                if len(points_in_cluster_k) > 0:
                    new_centroids[k] = np.mean(points_in_cluster_k, axis=0)

            current_centroids = new_centroids

```

```

        else:
            new_centroids[k] = current_centroids[k]

        current_centroids = new_centroids
        prev_cluster_ids = cluster_ids

        self._centroids = current_centroids
        self._clusters_ids = prev_cluster_ids

    def predict(self, x: np.array) -> np.array:
        """For each element of input returns corresponding cluster index

        Args:
            x: input points, array of [n_points; dim]

        Return
            labels: cluster ids, array of [n_points]
        """
        if self._centroids is None:
            raise RuntimeError("Fit estimator before predicting")

        if x.ndim == 1:
            if x.shape[0] == self._centroids.shape[1]:
                x = x.reshape(1, -1)
            else:
                raise ValueError(f"Input for predict has wrong shape {x.shape}")

        if x.shape[1] != self._centroids.shape[1]:
            raise ValueError(f"Input for predict has {x.shape[1]} feature")

        return np.argmin(get_euclidean_distances(x, self._centroids), axis=1)

```

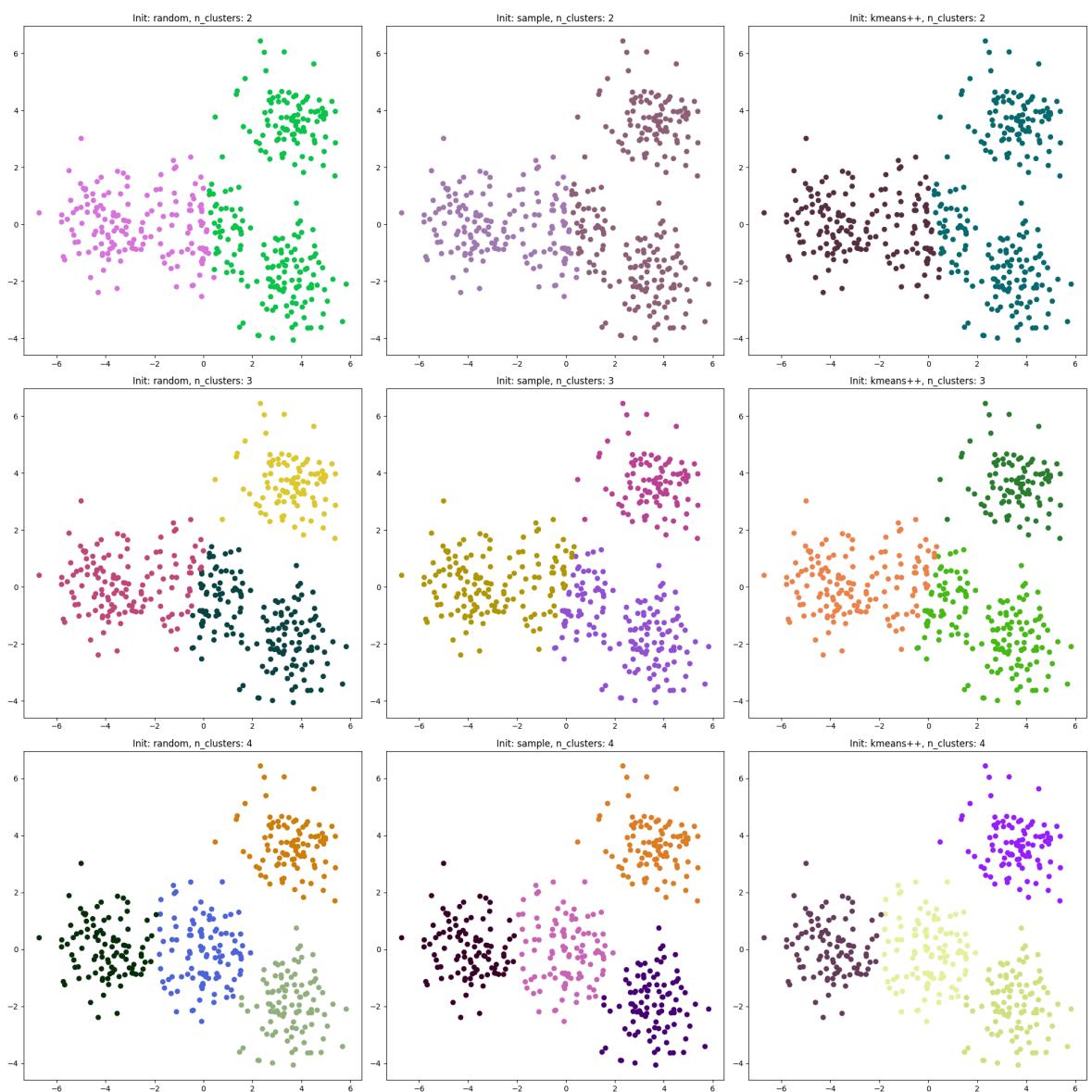
Протестируйте алгоритм на синтетических данных. При необходимости подберите гиперпараметры для достижения лучшего качества кластеризации.

Blobs clustering

```
In [85]: fig, axes = plt.subplots(3, 3, figsize=(20, 20))
axes = axes.flat

for i, (n_clusters, init) in enumerate(itertools.product(np.arange(2, 5),
                                                       kmeans = KMeans(n_clusters=n_clusters, init=init)
                                                       kmeans.fit(X_1)
                                                       labels = kmeans.predict(X_1)
                                                       cur_ax = axes[i]
                                                       visualize_clusters_on_ax(X_1, labels, cur_ax, title=f"Init: {init}, n
                                                       plt.tight_layout()
                                                       plt.show()
```

clustering



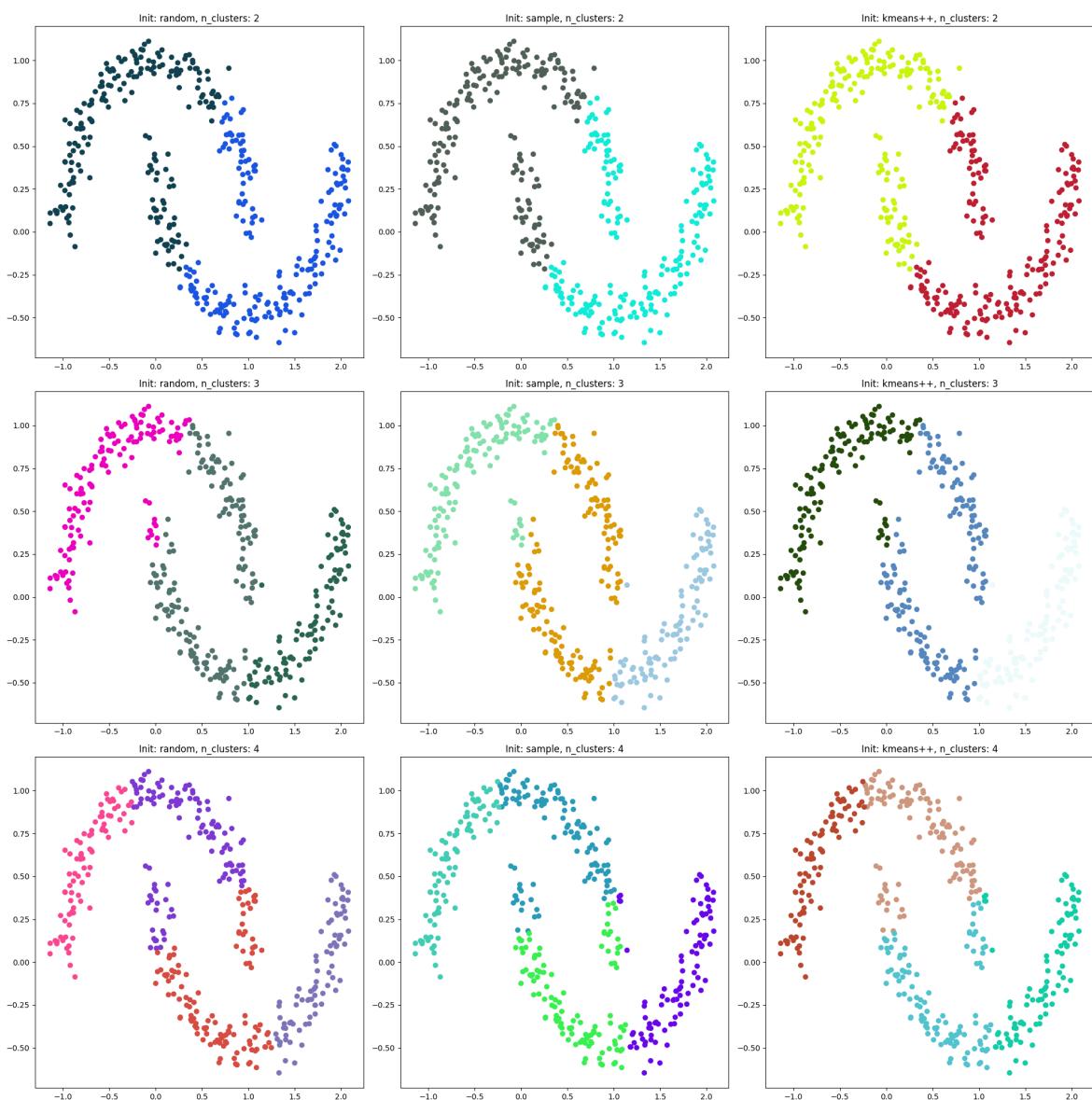
Moons clustering

```
In [84]: fig, axes = plt.subplots(3, 3, figsize=(20, 20))
axes = axes.flat

for i, (n_clusters, init) in enumerate(itertools.product(np.arange(2, 5),
    kmeans = KMeans(n_clusters=n_clusters, init=init)
    kmeans.fit(X_2)
    labels = kmeans.predict(X_2)
    cur_ax = axes[i]
    visualize_clusters_on_ax(X_2, labels, cur_ax, title=f"Init: {init}, n

plt.tight_layout()
plt.show()
```

clustering



KMeans подбирает центроид, минимизируя расстояния от каждой точки до центроида. Из-за этого он хорошо справляется с Blobs dataset, в котором кластеры похожи на сферы, и совсем не справляется с Moons dataset, в котором кластеры имеют вытянутый вид. Метод выбора начальных кластеров практически не влияет на итоговый результат из-за малого числа данных и несложной сходимости.

DBScan (3 балла)

В отличие от K-means, DBScan не позволяет задать количество кластеров, на которое будут разбиты данные. Руководствуясь геометрической интерпретацией, он позволяет выделять кластеры более сложной формы.

Описание методов

`fit_predict(X, y=None)` для каждого элемента `X` возвращает метку кластера, к которому он относится.

Возможные метрики

`euclidean`, `manhattan`, `chebyshev`

Для быстрого поиска соседей используйте `sklearn.neighbors.KDTree`

```
In [ ]: import collections

class DBScan:
    _known_metric_functions: Dict[str, Callable] = {
        "euclidean": get_euclidean_distances,
        "manhattan": get_manhattan_distances,
        "chebyshev": get_chebyshev_distance,
    }
    NOISE = -1
    UNVISITED = -2

    def __init__(self, eps: float = 0.5, min_samples: int = 5, leaf_size: int = 40):
        """
        DBScan clusterization

        Args:
            eps, min_samples: parameters for selecting core samples.
                Core samples is a samples with at least `min_samples` samples.
            metric: metric to use to calculate distance between samples,
            leaf_size: minimum leaf size for KDTree.
        """
        self._eps = eps
        self._min_samples = min_samples
        if metric not in self._known_metric_functions:
            raise ValueError(f"Unknown metric function: {metric}")
        self._metric = metric
        self._get_distance = self._known_metric_functions[metric]
        self._kdtree_leaf_size = leaf_size

    def fit_predict(self, x: ndarray, y=None) -> ndarray:
        """
        Cluterize and return index for each point in x
        """
        Args:
            x: input points, array of [n_points; dim]
            y: ignored, to mactch sklearn behaviour
        Return:
            labels: cluster ids, array of [n_points]
        """
        n_points = x.shape[0]
        if n_points == 0:
            return np.array([], dtype=int)

        labels = np.full(n_points, self.UNVISITED, dtype=int)
        current_cluster_id = -1

        tree = KDTree(x, leaf_size=self._kdtree_leaf_size, metric=self._metric)
        neighborhoods = tree.query_radius(x, r=self._eps)

        for point_idx in range(n_points):
            if labels[point_idx] != self.UNVISITED:
                continue

            current_point_neighbors = neighborhoods[point_idx]
            if len(current_point_neighbors) < self._min_samples:
```

```

continue

current_cluster_id += 1
labels[point_idx] = current_cluster_id

seed_set = collections.deque(current_point_neighbors)

while seed_set:
    q_idx = seed_set.popleft()

    if labels[q_idx] != self.UNVISITED and labels[q_idx] != s
        continue

    labels[q_idx] = current_cluster_id

    q_idx_actual_neighbors = neighborhoods[q_idx]
    if len(q_idx_actual_neighbors) >= self._min_samples:
        for nn_idx in q_idx_actual_neighbors:
            if labels[nn_idx] == self.UNVISITED or labels[nn_idx] != current_cluster_id:
                seed_set.append(nn_idx)
return labels

```

Протестируйте алгоритм на синтетических данных. При необходимости подберите гиперпараметры для достижения лучшего качества кластеризации.

Blobs clustering

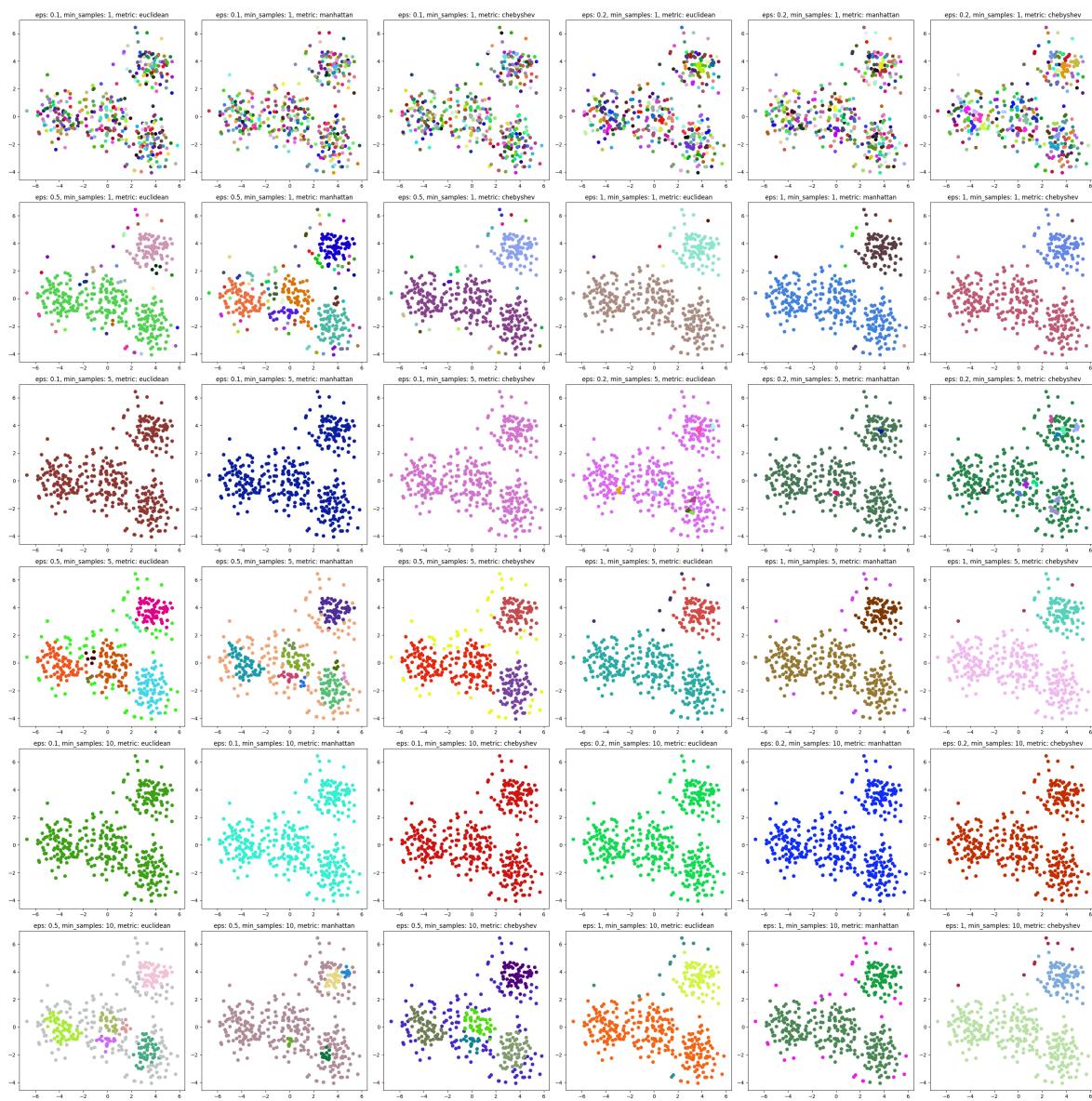
```
In [71]: epsilon = [0.05, 0.1, 0.5, 1]
min_samples = [1, 5, 10]
metrics = ["euclidean", "manhattan", "chebyshev"]
```

```
In [83]: fig, axes = plt.subplots(6, 6, figsize=(30, 30))
axes = axes.flat

for i, (min_s, eps, metric) in enumerate(itertools.product(min_samples, epsilon)):
    dbSCAN = DBScan(eps=eps, min_samples=min_s, metric=metric)
    labels = dbSCAN.fit_predict(X_1)
    cur_ax = axes[i]
    visualize_clusters_on_ax(X_1, labels, cur_ax, title=f"eps: {eps}, min_s: {min_s}, metric: {metric}")

plt.tight_layout()
plt.show()
```

clustering



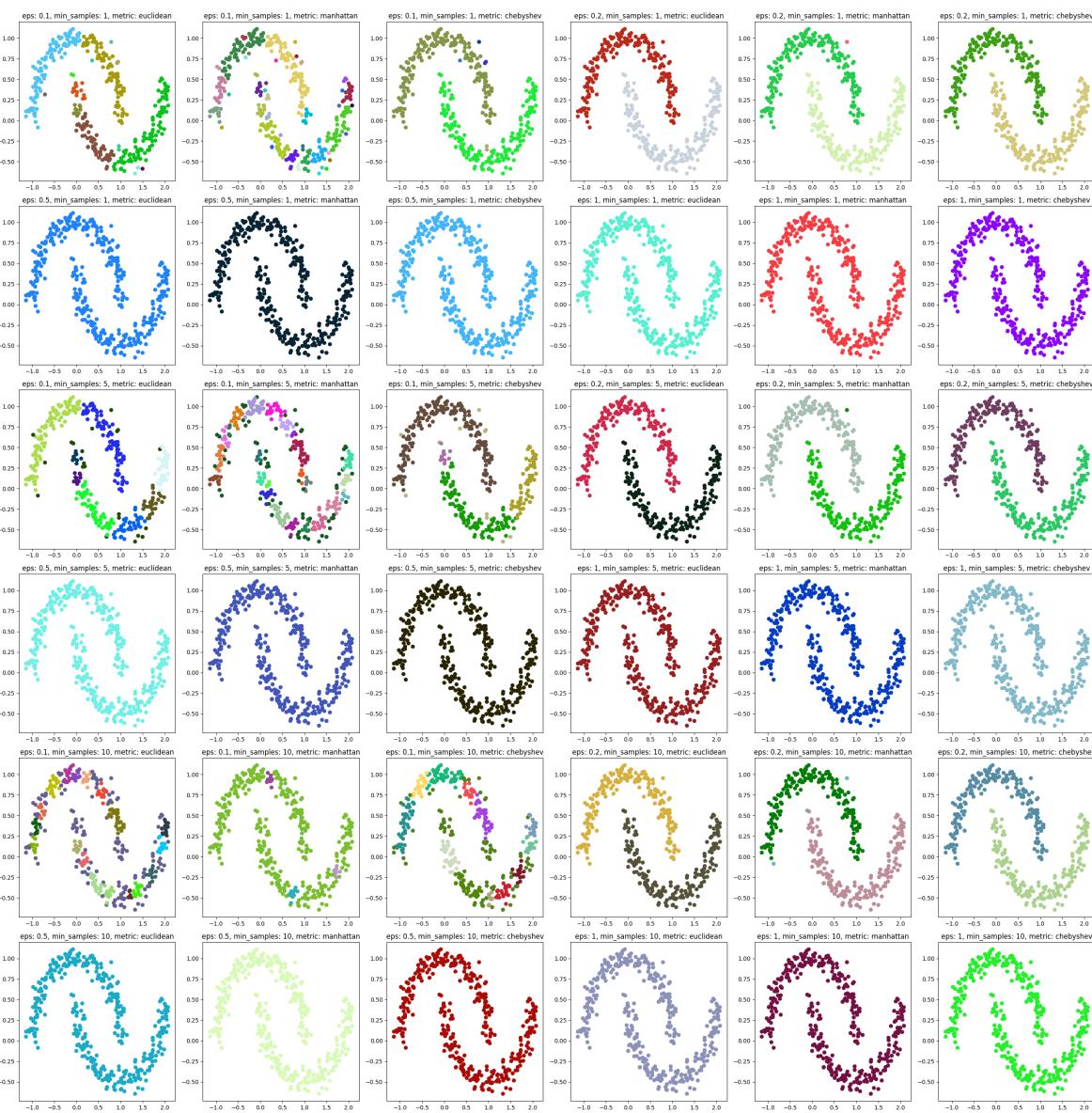
Moons clustering

```
In [78]: epsilon = [0.1, 0.2, 0.5, 1]
min_samples = [1, 5, 10]
metrics = ["euclidean", "manhattan", "chebysev"]
```

```
In [79]: fig, axes = plt.subplots(6, 6, figsize=(27, 27))
axes = axes.flat

for i, (min_s, eps, metric) in enumerate(itertools.product(min_samples, e
    dbscan = DBScan(eps=eps, min_samples=min_s, metric=metric)
    labels = dbscan.fit_predict(X_2)
    cur_ax = axes[i]
    visualize_clusters_on_ax(X_2, labels, cur_ax, title=f"eps: {eps}, min
    plt.tight_layout()
    plt.show()
```

clustering



DBScan строит кластеры последовательно, основываясь на количестве точек вокруг каждой. В Blobs dataset все точки находятся достаточно близко к друг другу, из-за чего результат не удовлетворительный. В отличие от Moons dataset, в котором кластеры явно разделены.

Agglomerative Clustering (3 балла)

Идея AgglomerativeClustering заключается в том, чтобы итеративно объединять кластеры с наименьшим расстоянием между ними. Данный метод обладает высокой вычислительной сложностью, поэтому применим только для относительно небольших наборов данных.

Описание методов

`fit_predict(X, y=None)` для каждого элемента X возвращает метку кластера, к которому он относится.

Linkage

Linkage - это способ, которым будет рассчитываться расстояние между кластерами. Предлагается реализовать три варианта такой функции:

1. `average` - расстояние рассчитывается как среднее расстояние между всеми парами точек, где одна принадлежит первому кластеру, а другая - второму.
2. `single` - расстояние рассчитывается как минимальное из расстояний между всеми парами точек, где одна принадлежит первому кластеру, а другая - второму.
3. `complete` - расстояние рассчитывается как максимальное из расстояний между всеми парами точек, где одна принадлежит первому кластеру, а другая - второму.

Для быстрого вычисления воспользуемся формулой Ланса-Уильямса

$$D(W, S) = \alpha_U D(U, S) + \alpha_V D(V, S) + \beta D(U, V) + \gamma |D(U, S) - D(V, S)|$$

Метод агломерации	α_U	α_V	β	γ
Single linkage	0.5	0.5	0	-0.5
Complete linkage	0.5	0.5	0	+0.5
Average linkage	$n_U / (n_U + n_V)$	$n_V / (n_U + n_V)$	0	0
Centroid linkage	$n_U / (n_U + n_V)$	$n_V / (n_U + n_V)$	$-n_U n_V / (n_U + n_V)^2$	0
Ward's method	$(n_S + n_U) / (n_S + n_U + n_V)$	$(n_S + n_V) / (n_S + n_U + n_V)$	$-n_S / (n_S + n_U + n_V)$	0

```
In [18]: # Return coefficients based on Lance–Williams formula for different linkages

def get_average_linkage_params(u_size: int, v_size: int) -> Tuple[float, float]:
    return u_size / (u_size + v_size), v_size / (u_size + v_size), 0.0, 0.0

def get_single_linkage_params(u_size: int, v_size: int) -> Tuple[float, float]:
    return 0.5, 0.5, 0.0, -0.5

def get_complete_linkage_params(u_size: int, v_size: int) -> Tuple[float, float]:
    return 0.5, 0.5, 0.0, 0.5
```

```
In [19]: class AgglomerativeClustering:

    _known_linkages = {
        "average": get_average_linkage_params,
        "single": get_single_linkage_params,
        "complete": get_complete_linkage_params,
    }

    def __init__(self, n_clusters: int = 16, linkage: str = "average"):
        """Agglomerative clusterization

        Args:
            n_clusters (int, optional): Number of clusters to form. Defaults to 16.
            linkage (str, optional): The linkage criterion to use. Defaults to "average".
```

```

        n_clusters: number of clusters
        linkage: linkage to use, one of "average", "single", and "com
"""
self._n_clusters = n_clusters
if linkage not in self._known_linkages:
    raise ValueError(f"Unknown linkage: {linkage}")
self._linkage = self._known_linkages[linkage]

def fit_predict(self, x: ndarray, y=None) -> ndarray:
    """Cluterize and return index for each point in x

    Args:
        x: input points, array of [n_points; dim]
        y: ignored, to mactch sklearn behaviour
    Return
        labels: cluster ids, array of [n_points]
"""
n_points = x.shape[0]

if n_points == 0:
    return np.array([], dtype=int)
if self._n_clusters >= n_points:
    return np.arange(n_points)
if self._n_clusters == 1:
    return np.zeros(n_points, dtype=int)

clusters = {i: [i] for i in range(n_points)}

distances = get_euclidean_distances(x, x)
np.fill_diagonal(distances, np.inf)

while len(clusters) > self._n_clusters:
    min_i, min_j = np.unravel_index(np.argmin(distances), distances.shape)
    if min_i > min_j:
        min_i, min_j = min_j, min_i

    u_size = len(clusters[min_i])
    v_size = len(clusters[min_j])

    a_i, a_j, b, g = self._linkage(u_size, v_size)

    for k in range(n_points):
        if k == min_i or k == min_j or k not in clusters:
            continue

        new_dist = a_i * distances[k, min_i] + a_j * distances[k,
                                                               min_j]
        distances[k, min_i] = new_dist
        distances[min_i, k] = new_dist

    clusters[min_i].extend(clusters[min_j])
    del clusters[min_j]

    distances[:, min_j] = np.inf
    distances[min_j, :] = np.inf

labels = np.zeros(n_points, dtype=int)
for i, (cluster_id, point_indices) in enumerate(clusters.items()):
    for point_idx in point_indices:
        labels[point_idx] = i

```

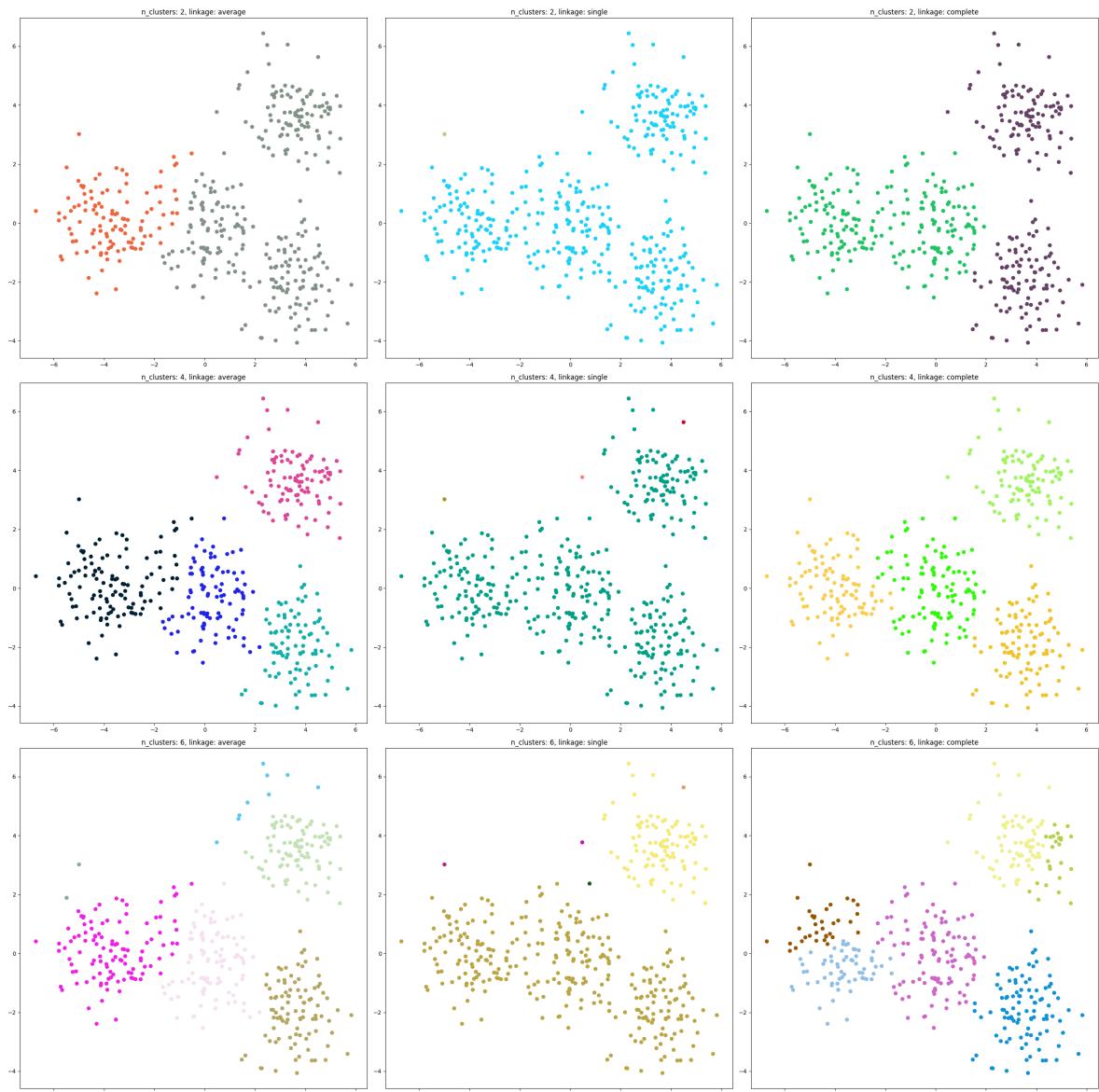
```
    return labels
```

Протестируйте алгоритм на синтетических данных. При необходимости подберите гиперпараметры для достижения лучшего качества кластеризации.

Blobs clustering

```
In [81]: fig, axes = plt.subplots(3, 3, figsize=(27, 27))
axes = axes.flat

for i, (n_clusters, linkage) in enumerate(itertools.product(np.arange(2,
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linka
labels = agg_clustering.fit_predict(X_1)
cur_ax = axes[i]
visualize_clusters_on_ax(X_1, labels, cur_ax, title=f"n_clusters: {n_
plt.tight_layout()
plt.show()
```

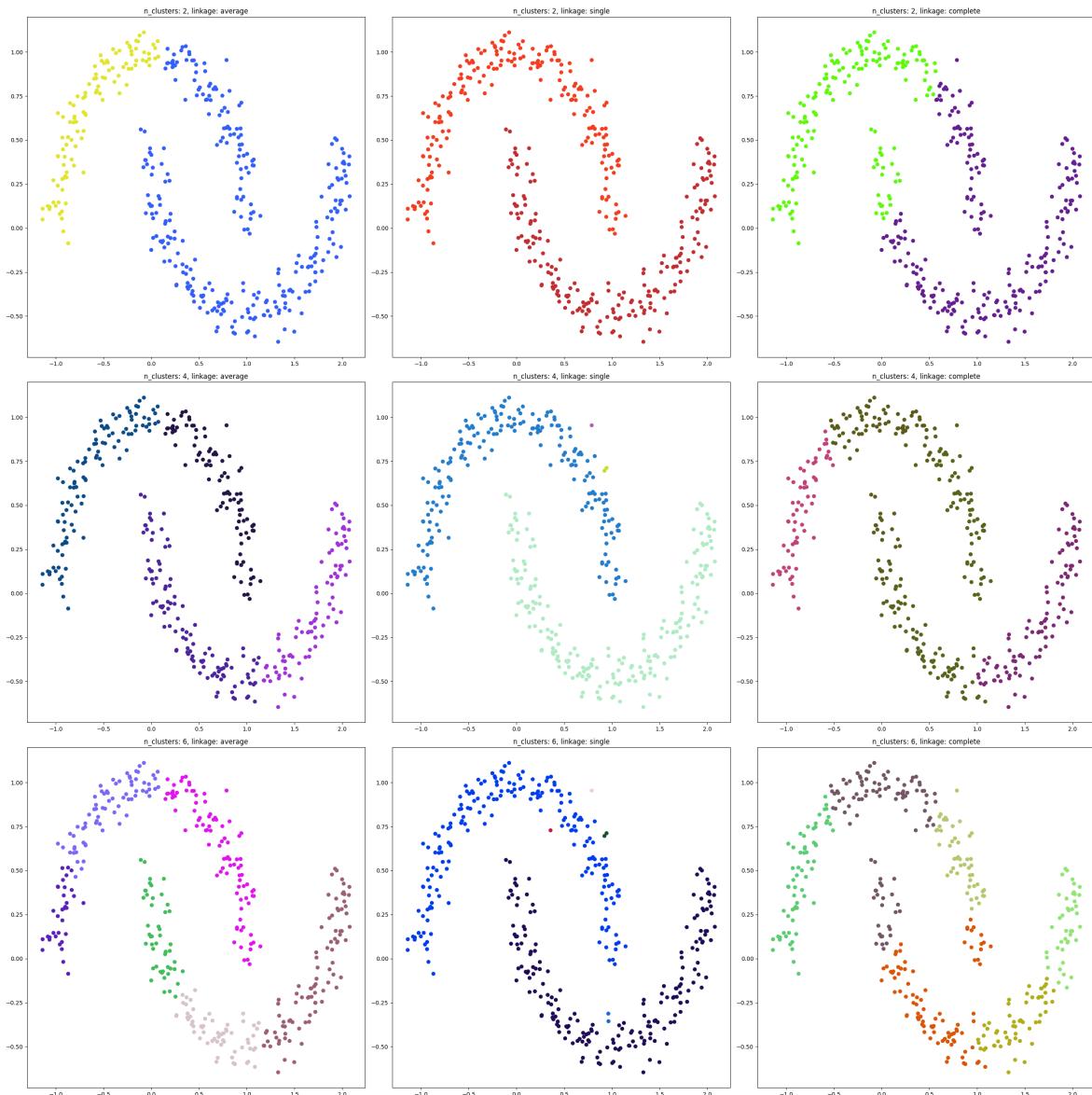


```
In [82]: fig, axes = plt.subplots(3, 3, figsize=(27, 27))
axes = axes.flat
```

```

for i, (n_clusters, linkage) in enumerate(itertools.product(np.arange(2,
    agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linka
labels = agg_clustering.fit_predict(X_2)
cur_ax = axes[i]
visualize_clusters_on_ax(X_2, labels, cur_ax, title=f"n_clusters: {n_
plt.tight_layout()
plt.show()

```



Agglomerative Clustering объединяет точки/кластеры с нименьшим расстоянием в один кластер, пока не получим необходимое число кластеров. Из-за этого он хорошо справляется и с Blobs и с Moons: для первого с avarage, из-за близкого расположения точек, для второго --- single, из-за явного разделения.

Сжатие изображений (1 балл)

Реализуйте методы считывания и записи изображения при помощи библиотеки `Pillow`.

Нормализованное изображение - это изображение, у которого все значения пикселей находятся в $[0; 1]$.

```
In [21]: def read_image(path: str) -> ndarray:
    """Read and return image from file

    Args:
        path: path to the image file

    Returns:
        image: normalized image array of shape [H, W, 3]
    """
    image = Image.open(path)
    image_array = np.array(image)
    return image_array / 255.0

def show_image(image: ndarray):
    """Display the image using matplotlib

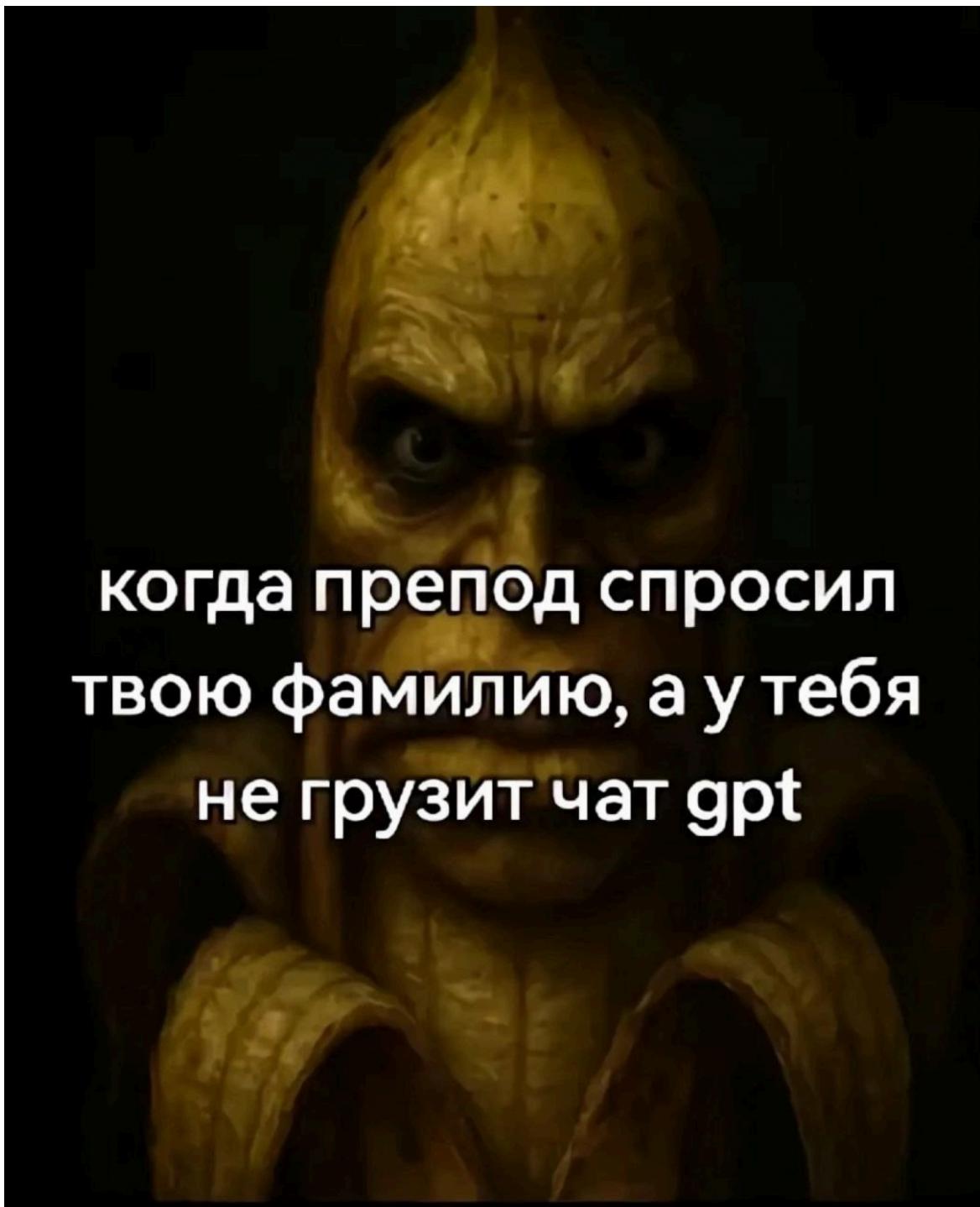
    Args:
        image: normalized image array of shape [H, W, 3]
    """
    plt.figure(figsize=np.array(image.shape[:-1]) / 50)
    plt.imshow(image)
    plt.axis("off")
    plt.tight_layout()
    plt.show()

def save_image(image: ndarray, path: str):
    """Save the image to file

    Args:
        image: normalized image array of shape [H, W, 3]
        path: path to save the image
    """
    denormalized_image = (image * 255).astype(np.uint8)
    restored_image = Image.fromarray(denormalized_image)
    restored_image.save(path)
```

```
In [29]: # Put your favorite image and read it
```

```
image = read_image("src/homeworks/homework7/gpt.jpg")
show_image(image)
```



Реализуйте функцию, которая будет кластеризовать цвета изображения одним из реализованных алгоритмов. Интерфейс этой функции можно менять. Функция должна возвращать новое изображение, в котором цвета заменены на цвета кластеров.

Затем примените ее к цветам изображения.

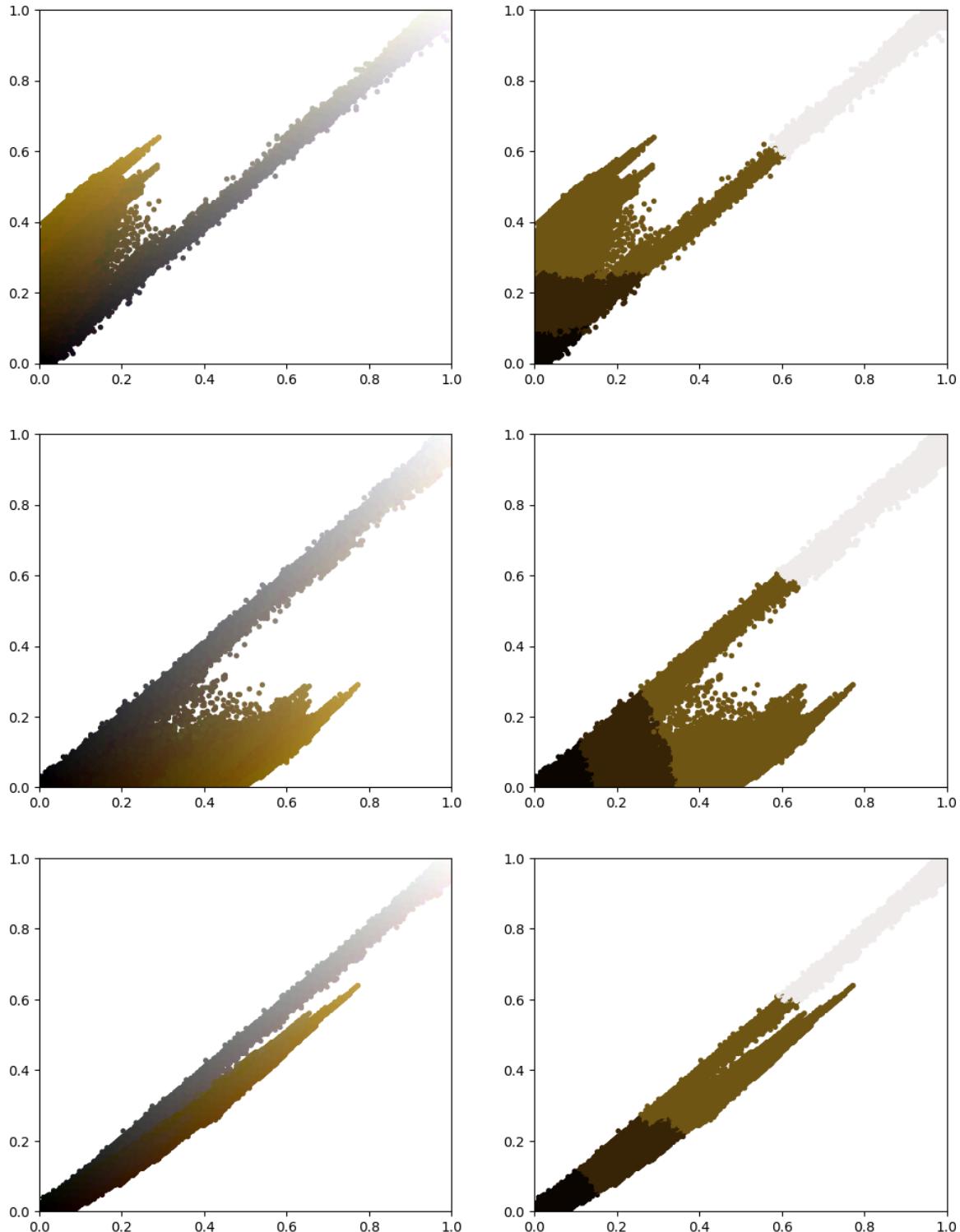
Примечание: это задание проще всего выполнить при помощи KMeans

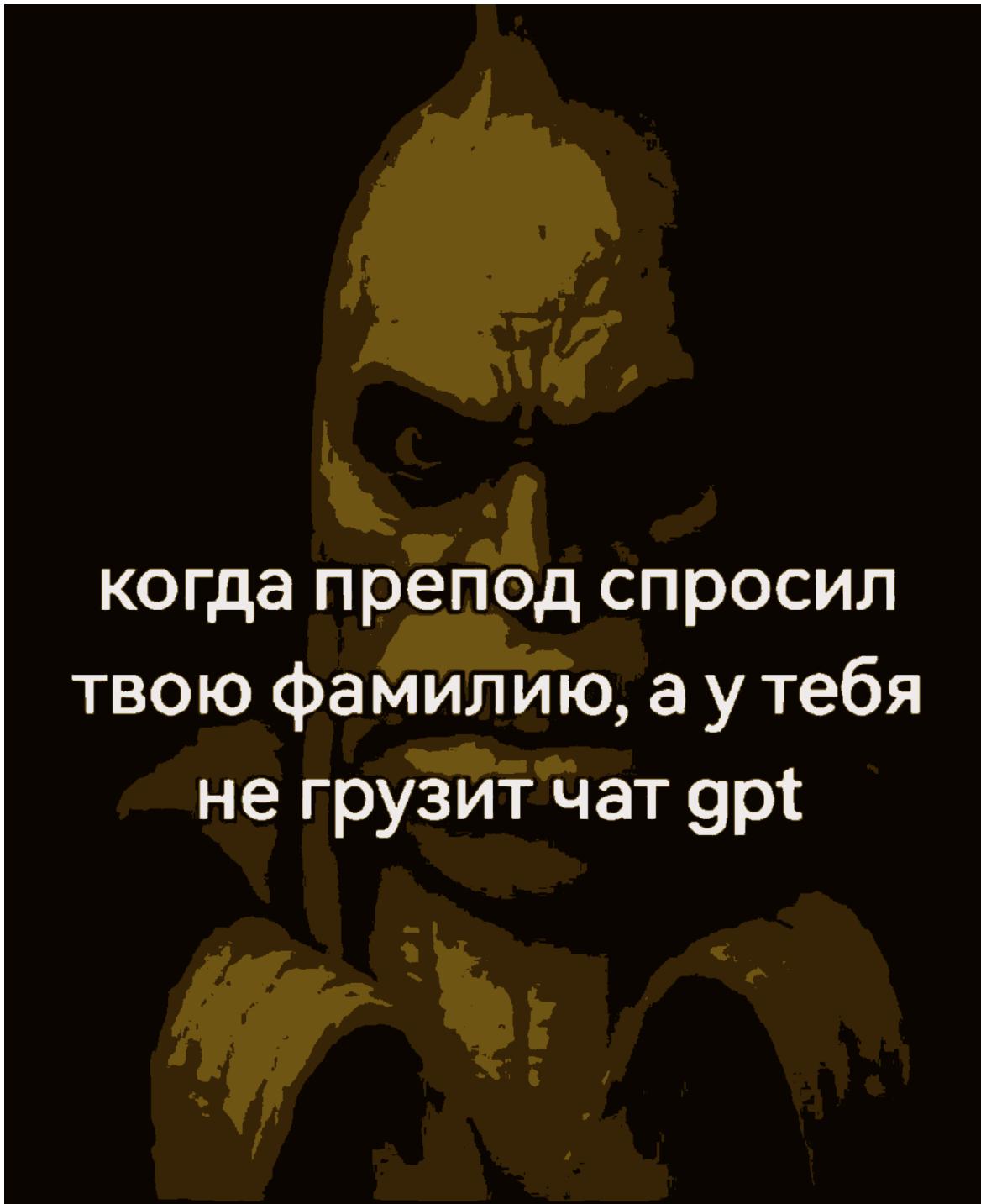
```
In [25]: def clusterize_image(image: ndarray, n_clusters: int):  
    model = KMeans(n_clusters, "kmeans++")  
    model.fit(image.reshape(-1, 3))  
    cluster_colors = model._centroids  
  
    clusters = model.predict(image.reshape(-1, 3))
```

```
recolored = cluster_colors[clusters].reshape(image.shape)

clusters_statistics(image.reshape(-1, 3), cluster_colors, clusters)
return recolored
```

```
In [28]: image = read_image("src/homeworks/homework7/gpt.jpg")
result = clusterize_image(image, 4)
show_image(result)
save_image(result, "result.jpg")
```





когда препод спросил
твою фамилию, а у тебя
не грузит чат gpt