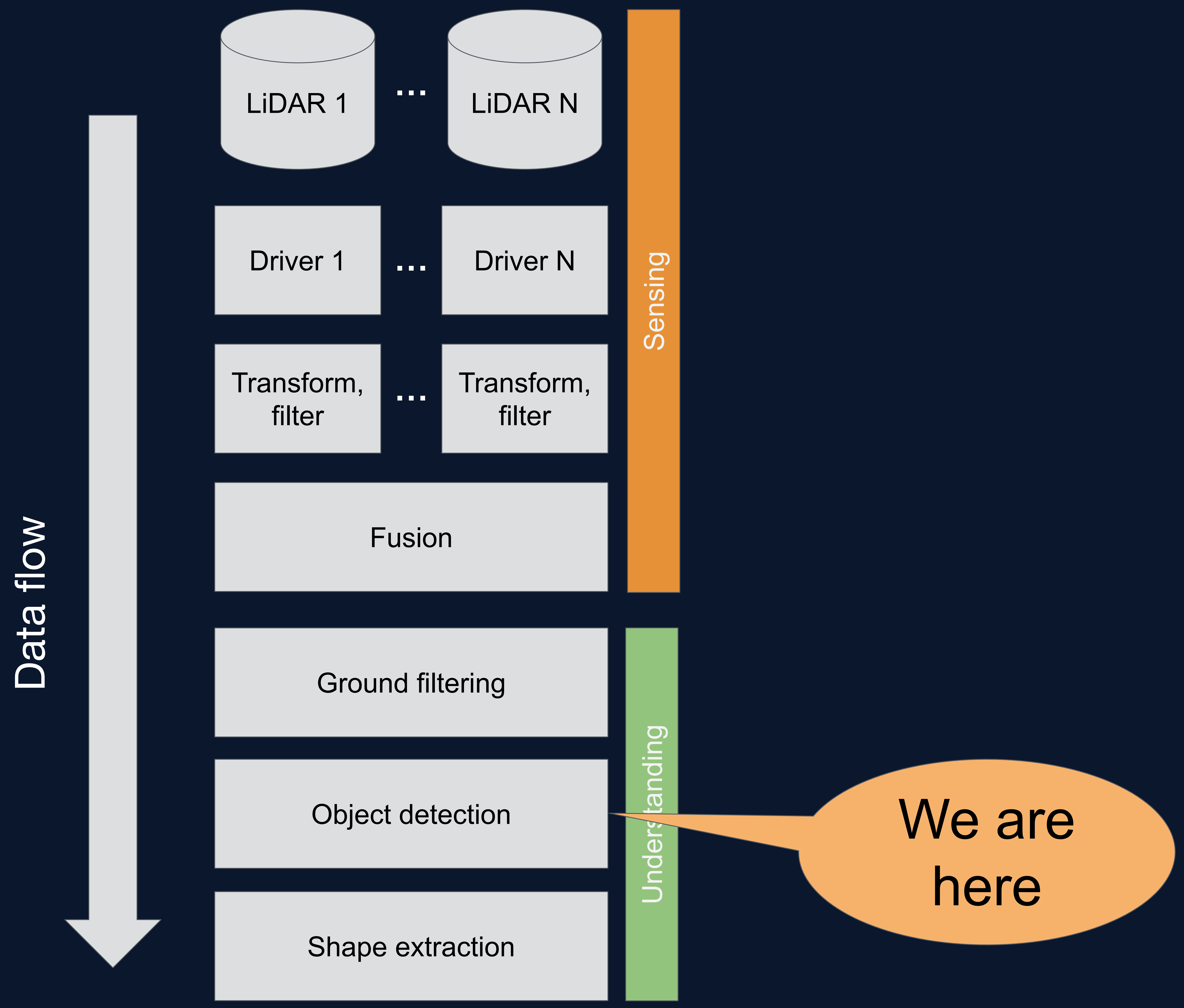
05 / Object Detection

Object Detection in the Classical LiDAR Processing Stack



The Problem of Object Detection

Why?

- Input nonground point cloud is cumbersome alone
- Need to discriminate between separate objects
  - e.g. *segment* nonground point cloud
- Partition point cloud into objects
- Remove some noise

How?

- Fundamentally group point together
  - Somehow
  - Based on some metric

Euclidean Clustering

One of a small handful of classical clustering/segmentation algorithms:

1. create a kd-tree representation for the input point cloud dataset P;
2. set up an empty list of clusters C, and a queue of the points that need to be checked Q;
3. then for every point $p_i \in P$, perform the following steps:
   - add pi to the current queue Q;
   - for every point $p_i \in Q$ do:
     – search for the set P^k_i of point neighbors of p_i in a sphere with radius $r < d_{th}$;
     – for every neighbor $p^k_i \in P^k_i$, check if the point has already been processed, and if not add it to Q;
   - when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list
4. the algorithm terminates when all points $p_i \in P$ have been processed and are now part of the list of point clusters C.

[1] Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments, Rusu, 2009
[2] PCL

Two other ways to look at euclidean clustering

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there's no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point

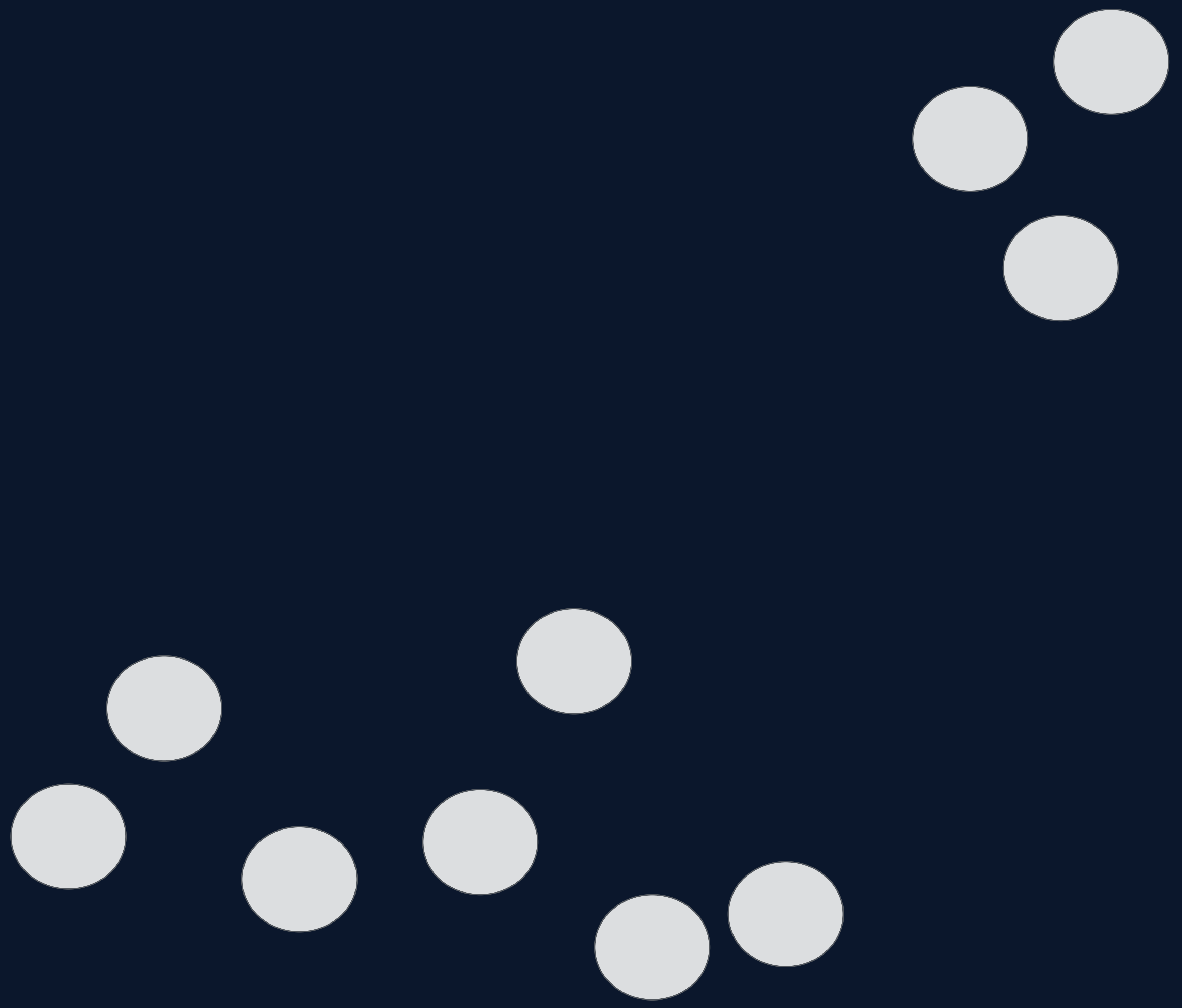
Alternatively, the graph view

- Start with point cloud
- Each point is a vertex in the graph
- Connect each point *near* one another
- A cluster is a connected subgraph

Euclidean clustering, illustrated

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there’s no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point



Unclassified

Cluster 1, Final

Cluster 1, New

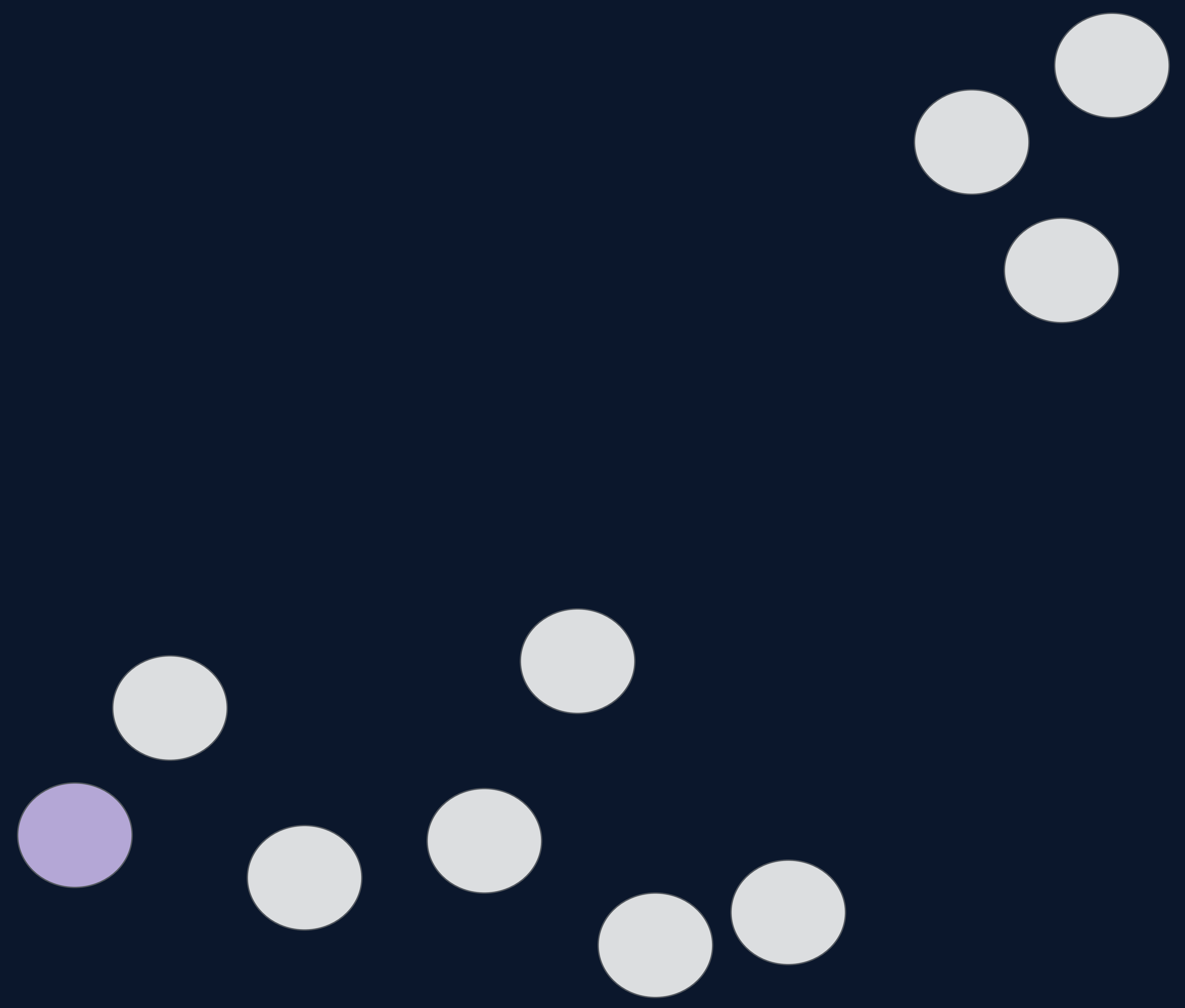
Cluster 2, Final

Cluster 2, New

Euclidean clustering, illustrated

In other words:

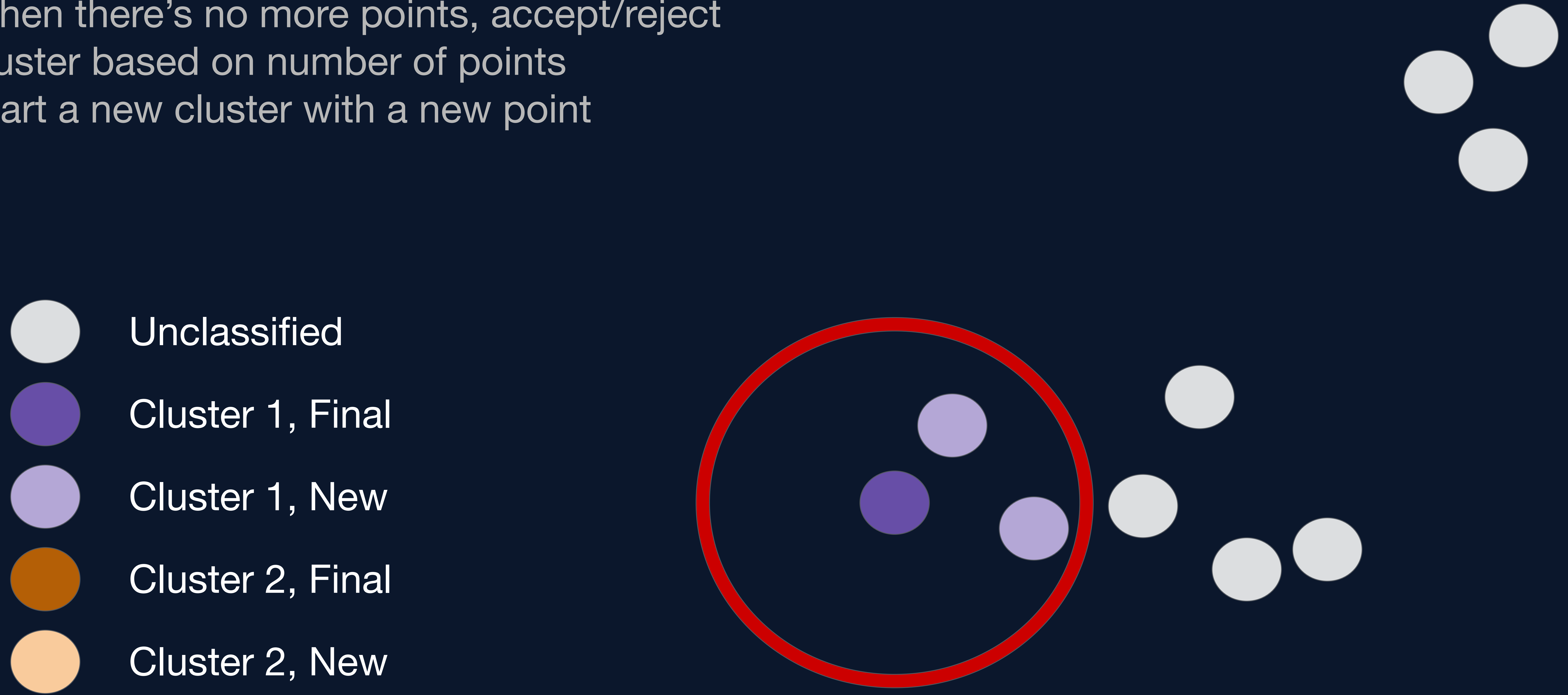
1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there's no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point



Euclidean clustering, illustrated

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there's no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point



Euclidean clustering, illustrated

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there's no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point

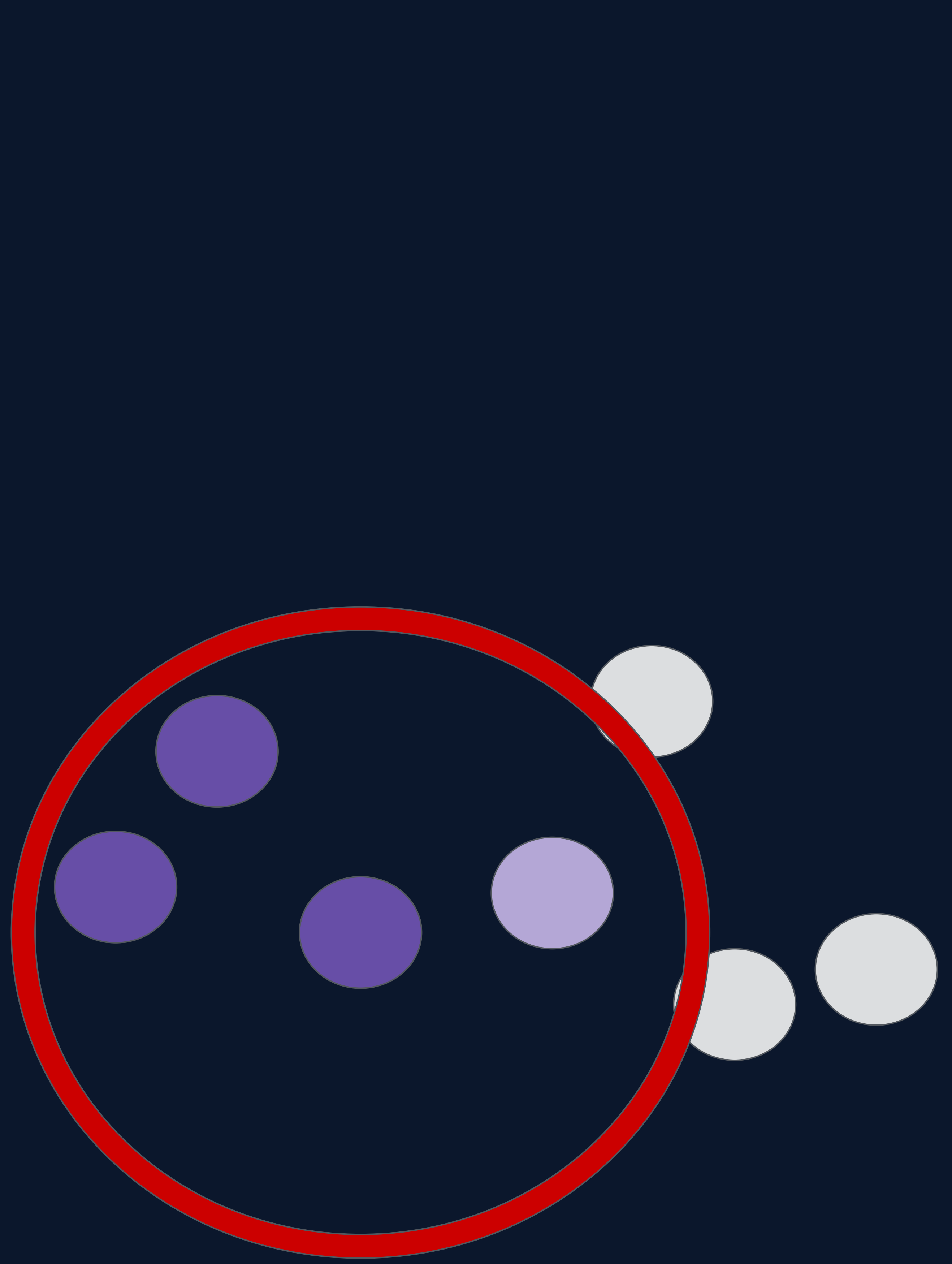
Unclassified

Cluster 1, Final

Cluster 1, New

Cluster 2, Final

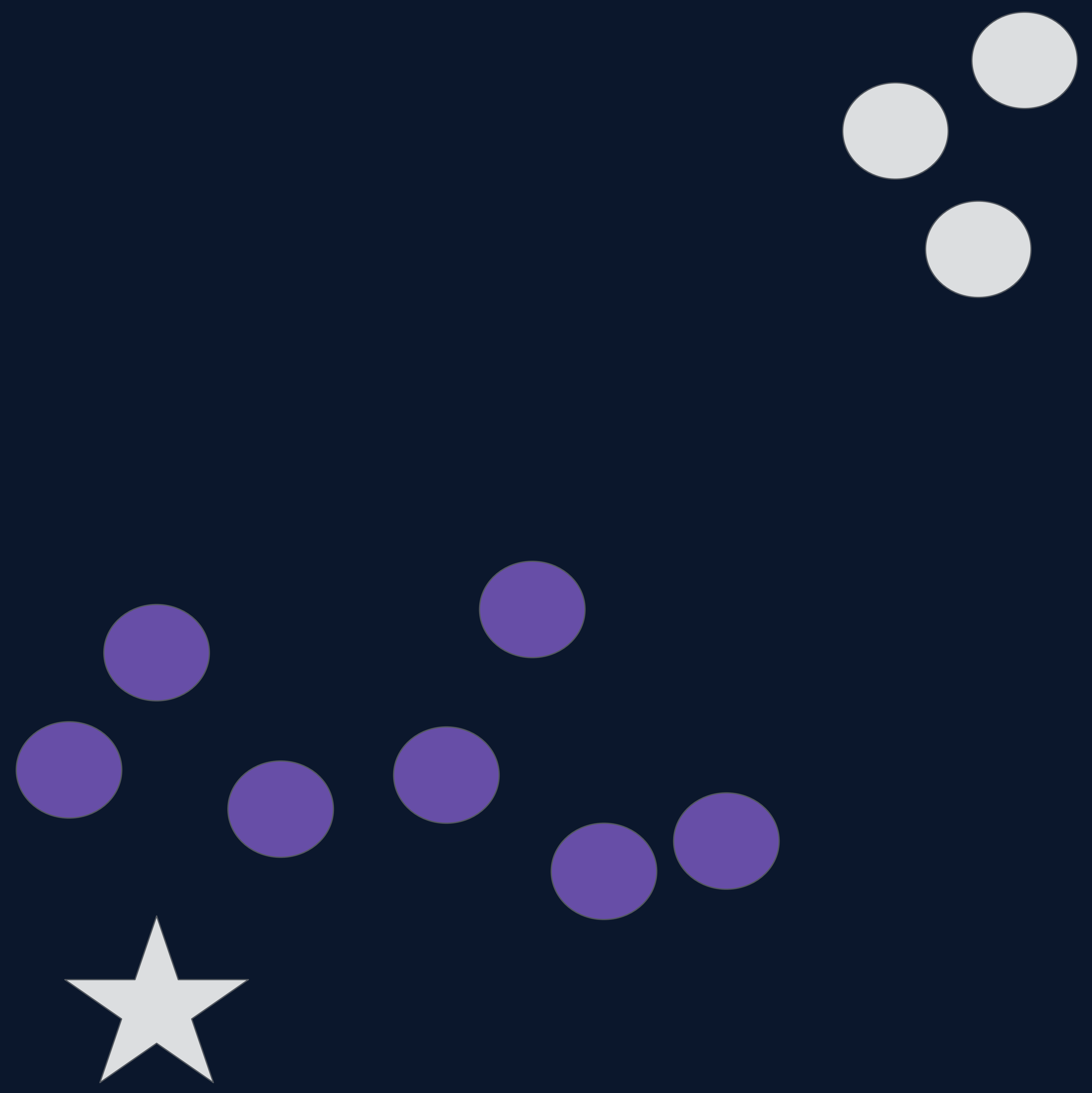
Cluster 2, New



Euclidean clustering, illustrated

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there's no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point



Unclassified

Cluster 1, Final

Cluster 1, New

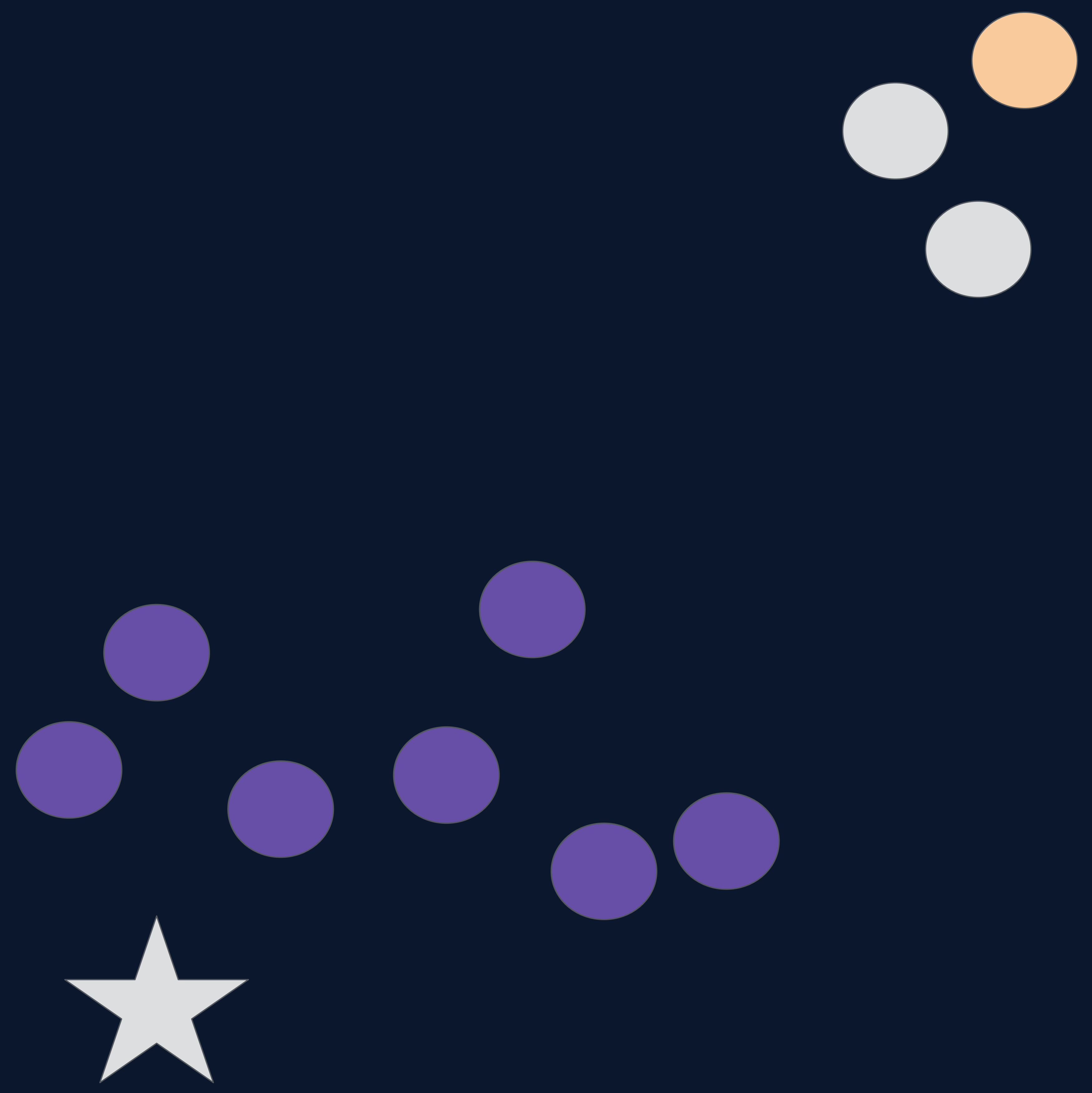
Cluster 2, Final

Cluster 2, New

Euclidean clustering, illustrated

In other words:

1. Start with an empty cluster
2. Add some point to the cluster
3. Find all points *near* this point, add to cluster
4. Repeat (2) and (3) for each new point in the cluster
5. When there’s no more points, accept/reject cluster based on number of points
6. Start a new cluster with a new point



Unclassified

Cluster 1, Final

Cluster 1, New

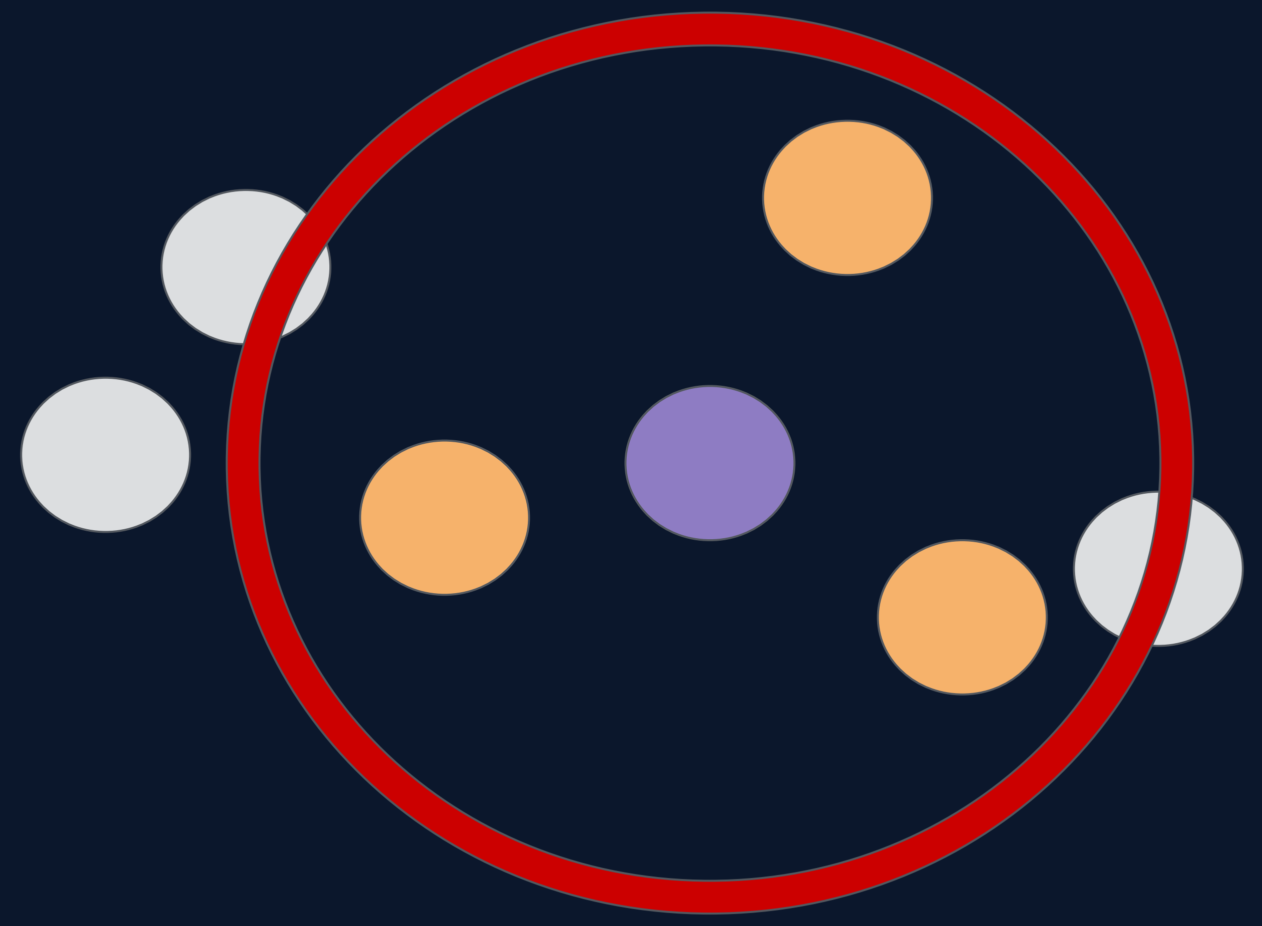
Cluster 2, Final

Cluster 2, New

Near, not nearest

Near:
- All points with distance less than some threshold
- Different from nearest
- Different from k-nearest
- Best data structure for (k-nearest) is k-d tree
  - But not for near-neighbor lookups

Near



Nearest:
- Use k-d tree
- O(log N) for lookup

Nearest

Using the right data structure

Integer lattice/Spatial hash

1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
    - Add to output each point that is *near* the query point

Average query complexity: O(1) * O(1) * O(k) = O(k)

Let *n* be total number of points, *k* be the average number of neighbors

Even has its own wiki page

The spatial hash, illustrated

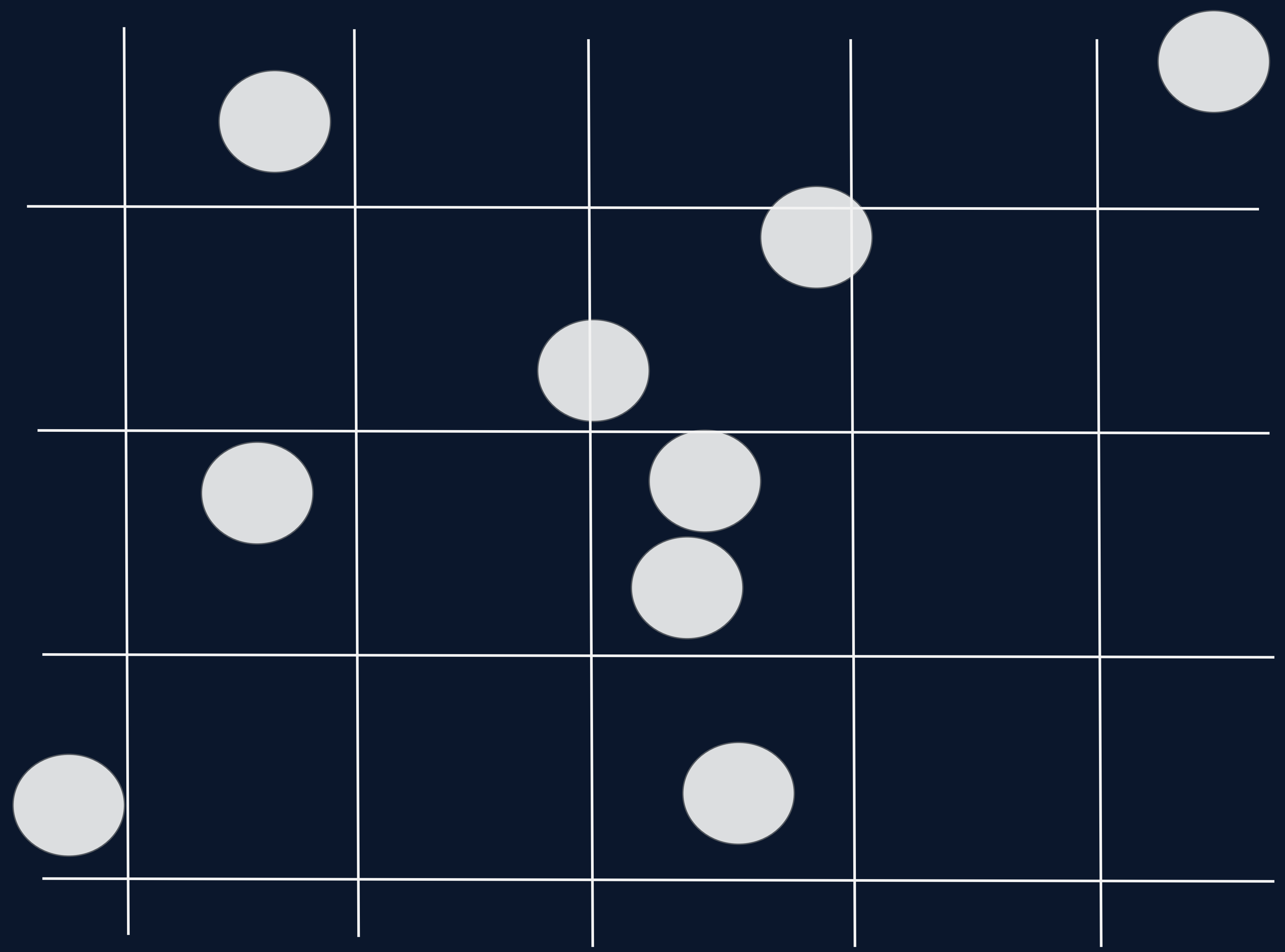
1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
    - Add to output each point that is *near* the query point

Unclassified

Query point

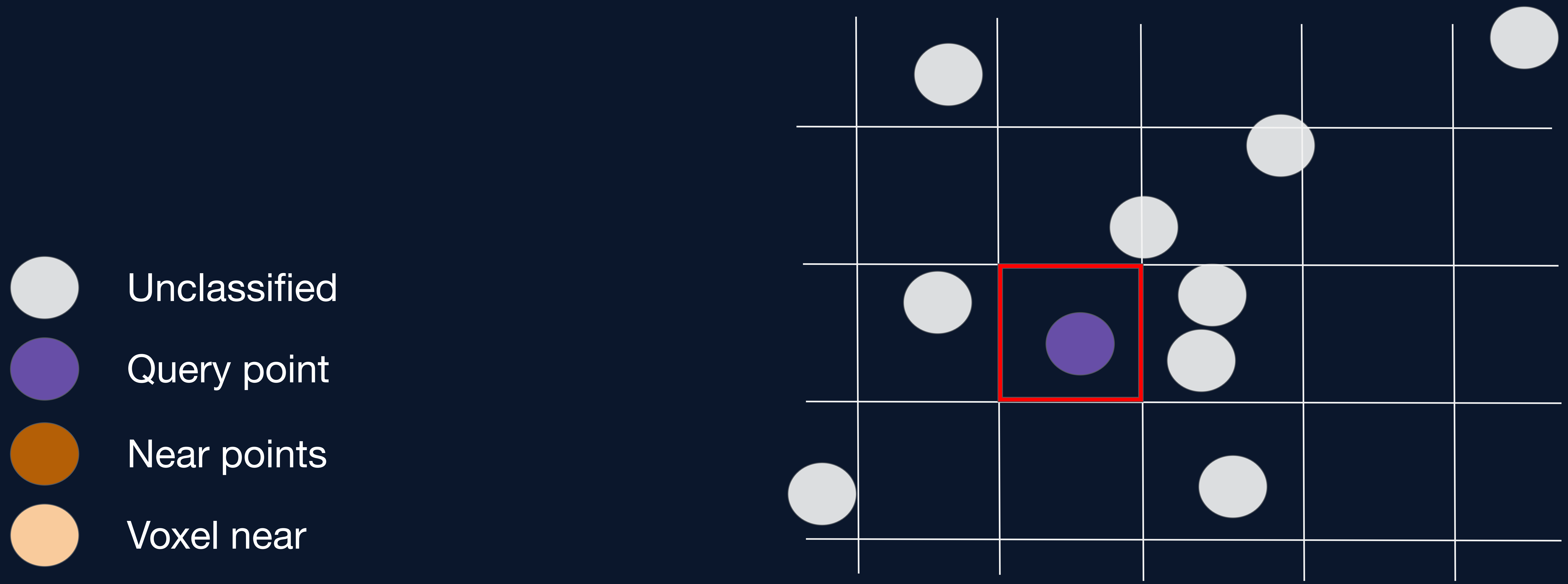
Near points

Voxel near



The spatial hash, illustrated

1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
    - Add to output each point that is *near* the query point



The spatial hash, illustrated

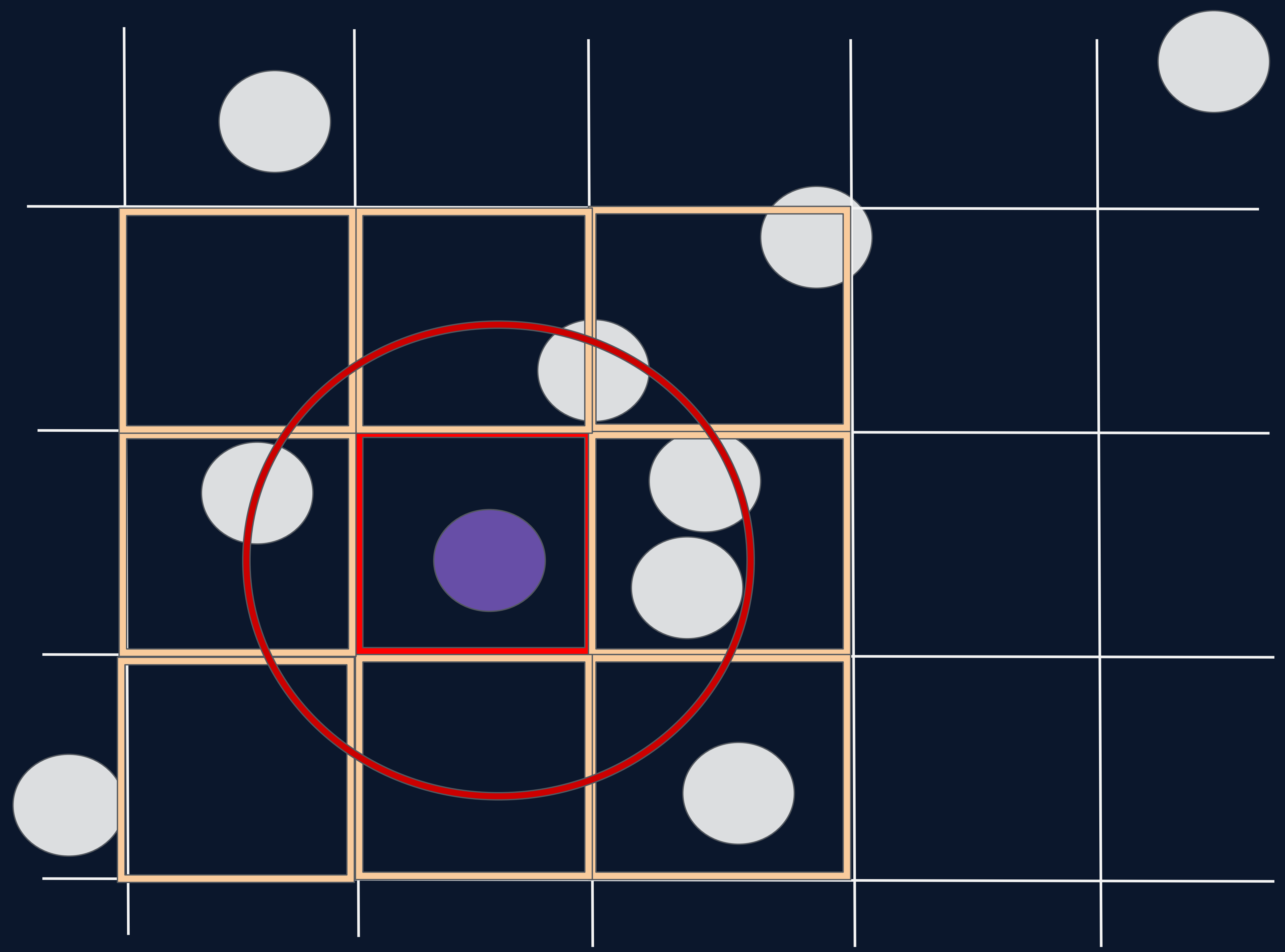
1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
    - Add to output each point that is *near* the query point

Unclassified

Query point

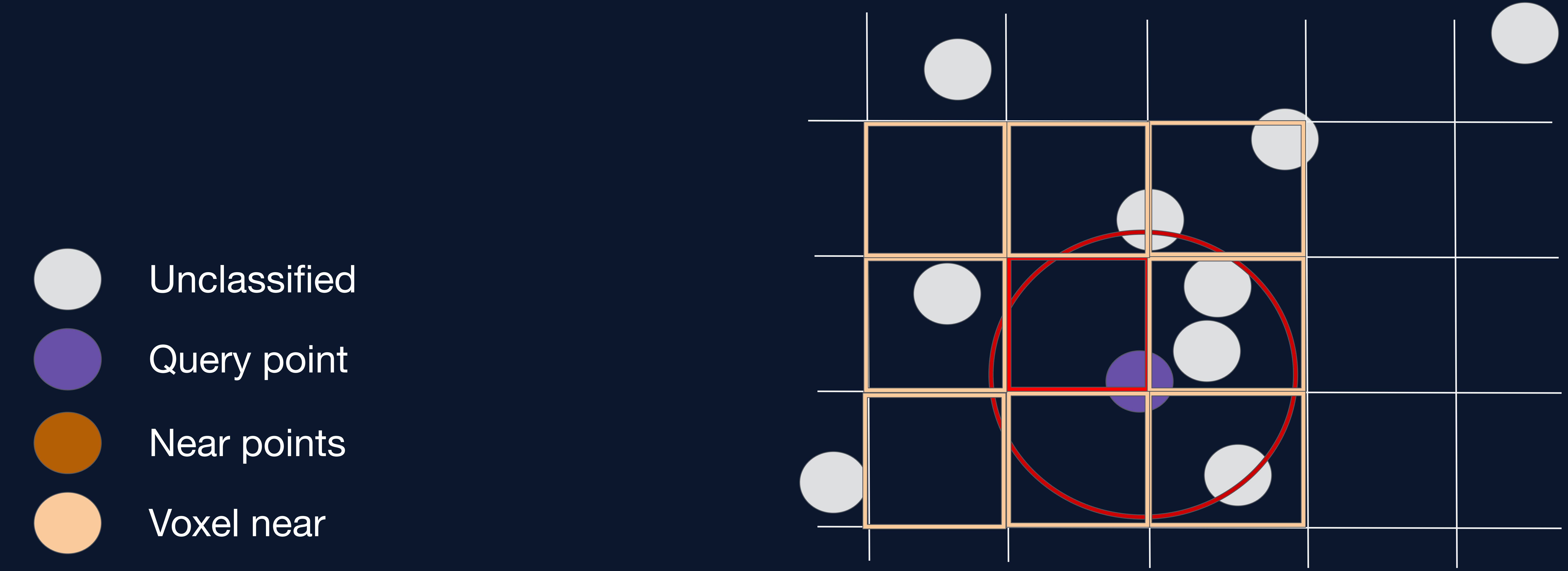
Near points

Voxel near



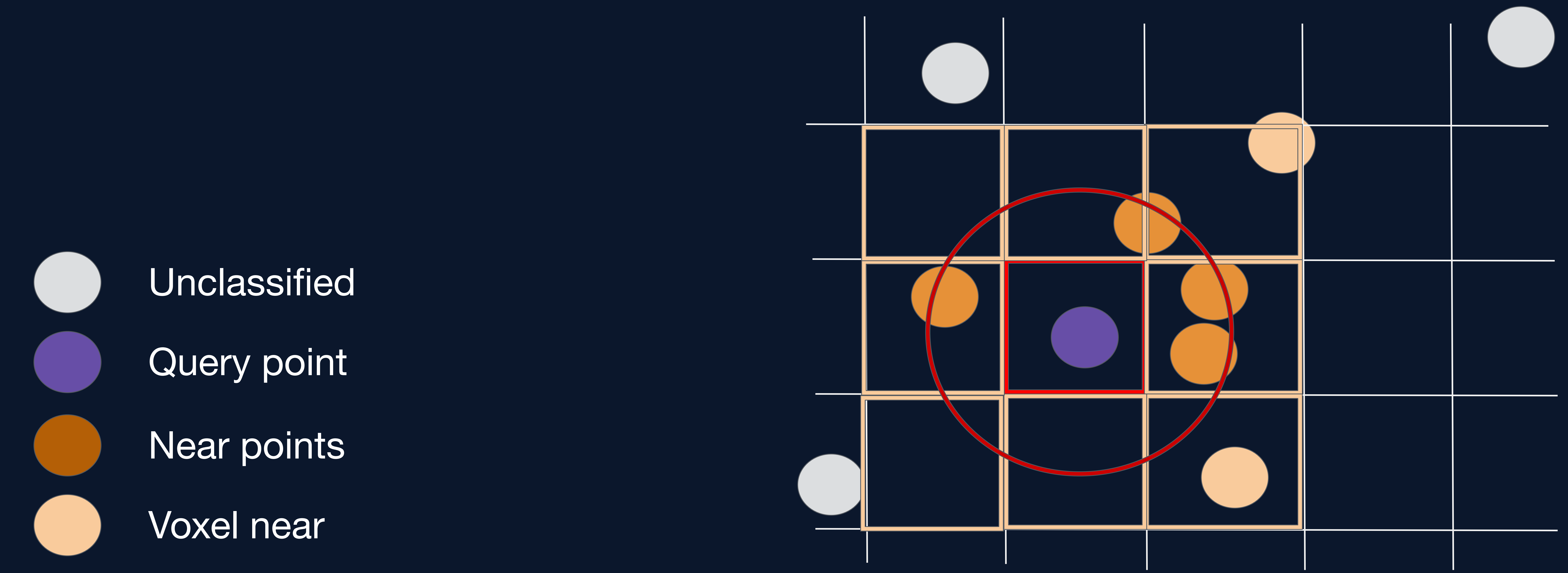
The spatial hash, illustrated

1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
   - Add to output each point that is *near* the query point



The spatial hash, illustrated

1. Subdivide space into voxels: O(n)
2. Find voxel for query point: O(1)
3. Find all voxels a *near* point could possibly fall in: O(1)
4. Iterate over all points in all *near* voxels: O(k)
    - Add to output each point that is *near* the query point



(Average-case) Complexity Analysis

O(n)

O(1)

1. create a kd-tree representation for the input point cloud dataset P;
2. set up an empty list of clusters C, and a queue of the points that need to be checked Q;
3. then for every point $p_i \in P$, perform the following steps:
   - add pi to the current queue Q;
   - for every point $p_i \in Q$ do:
     - search for the set P^k_i of point neighbors of p_i in a sphere with radius $r < d_{th}$;
     - for every neighbor $p^k_i \in P^k_i$, check if the point has already been processed, and if not add it to Q;
   - when the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list
4. the algorithm terminates when all points $p_i \in P$ have been processed and are now part of the list of point clusters C.

O(1)

O(n(k + Q(k)))

O(k + Q(k))

O(Q(k))

O(k)

O(1)

O(1)

Total complexity: $O(n + n(k + Q(k)) = O(n(k + Q(k))$

Original algorithm: $Q(k) = k \log n \rightarrow O(kn(1+\log n))$

Integer lattice: $Q(k) = k \rightarrow O(kn)$

If k is a problem defined constant, we went from linearithmic to linear!

Geometric Object Detection - Summary

- Object detection algorithms are needed to distinguish individual collidable objects
- Most object detection algorithms use some kind of region growing method
  - On voxels
  - In euclidean space
  - In angles in the range-image space
- Autoware.Auto uses a version of euclidean clustering

Important optimizations

- Using the right data structure:
  - O(n log n) -> O(n)
  - Important since *n* can get big: 10k-1m!
- Concretely:
  - Autoware.ai: Barely 100ms runtime with aggressive downsampling
  - Autoware.Auto: Comfortable 10ms runtime with no* downsampling