**Practical Assignment**

**Objective: - Object Tracking for any objects.**

Object tracking is the process of identifying where a particular object is present in the image.

Library :-https://github.com/tryolabs/norfair

**Dataset Link: -**

Use anyone of your choice.

**Use any 2 object for Tracking.**

**Task: -** Create a Desktop App where the videos can be used and get the count of 2 objects in the video.

**Deployment: -** Any Free Platform(Try to look out for free options.)

**Assignment Submission: -** Only submit the hosted app link. OR GitHub Link

from logging import warning

from typing import Any, Callable, Hashable, List, Optional, Sequence, Tuple, Union

import numpy as np

from rich import print

from norfair.camera\_motion import CoordinatesTransformation

from .distances import (

AVAILABLE\_VECTORIZED\_DISTANCES,

ScalarDistance,

get\_distance\_by\_name,

)

from .filter import FilterFactory, OptimizedKalmanFilterFactory

from .utils import validate\_points

class Tracker:

"""

The class in charge of performing the tracking of the detections produced by a detector.

Parameters

----------

distance\_function : Union[str, Callable[[Detection, TrackedObject], float]]

Function used by the tracker to determine the distance between newly detected objects and the objects that are currently being tracked.

This function should take 2 input arguments, the first being a [Detection][norfair.tracker.Detection], and the second a [TrackedObject][norfair.tracker.TrackedObject].

It has to return a `float` with the distance it calculates.

Some common distances are implemented in [distances][], as a shortcut the tracker accepts the name of these [predefined distances][norfair.distances.get\_distance\_by\_name].

Scipy's predefined distances are also accepted. A `str` with one of the available metrics in

[`scipy.spatial.distance.cdist`](https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html).

distance\_threshold : float

Defines what is the maximum distance that can constitute a match.

Detections and tracked objects whose distances are above this threshold won't be matched by the tracker.

hit\_counter\_max : int, optional

Each tracked objects keeps an internal hit counter which tracks how often it's getting matched to a detection,

each time it gets a match this counter goes up, and each time it doesn't it goes down.

If it goes below 0 the object gets destroyed. This argument defines how large this inertia can grow,

and therefore defines how long an object can live without getting matched to any detections, before it is displaced as a dead object, if no ReID distance function is implemented it will be destroyed.

initialization\_delay : Optional[int], optional

Determines how large the object's hit counter must be in order to be considered as initialized, and get returned to the user as a real object.

It must be smaller than `hit\_counter\_max` or otherwise the object would never be initialized.

If set to 0, objects will get returned to the user as soon as they are detected for the first time,

which can be problematic as this can result in objects appearing and immediately dissapearing.

Defaults to `hit\_counter\_max / 2`

pointwise\_hit\_counter\_max : int, optional

Each tracked object keeps track of how often the points it's tracking have been getting matched.

Points that are getting matched (`pointwise\_hit\_counter > 0`) are said to be live, and points which aren't (`pointwise\_hit\_counter = 0`)

are said to not be live.

This is used to determine things like which individual points in a tracked object get drawn by [`draw\_tracked\_objects`][norfair.drawing.draw\_tracked\_objects] and which don't.

This argument defines how large the inertia for each point of a tracker can grow.

detection\_threshold : float, optional

Sets the threshold at which the scores of the points in a detection being fed into the tracker must dip below to be ignored by the tracker.

filter\_factory : FilterFactory, optional

This parameter can be used to change what filter the [`TrackedObject`][norfair.tracker.TrackedObject] instances created by the tracker will use.

Defaults to [`OptimizedKalmanFilterFactory()`][norfair.filter.OptimizedKalmanFilterFactory]

past\_detections\_length : int, optional

How many past detections to save for each tracked object.

Norfair tries to distribute these past detections uniformly through the object's lifetime so they're more representative.

Very useful if you want to add metric learning to your model, as you can associate an embedding to each detection and access them in your distance function.

reid\_distance\_function: Optional[Callable[["TrackedObject", "TrackedObject"], float]]

Function used by the tracker to determine the ReID distance between newly detected trackers and unmatched trackers by the distance function.

This function should take 2 input arguments, the first being tracked objects in the initialization phase of type [`TrackedObject`][norfair.tracker.TrackedObject],

and the second being tracked objects that have been unmatched of type [`TrackedObject`][norfair.tracker.TrackedObject]. It returns a `float` with the distance it

calculates.

reid\_distance\_threshold: float

Defines what is the maximum ReID distance that can constitute a match.

Tracked objects whose distance is above this threshold won't be merged, if they are the oldest tracked object will be maintained

with the position of the new tracked object.

reid\_hit\_counter\_max: Optional[int]

Each tracked object keeps an internal ReID hit counter which tracks how often it's getting recognized by another tracker,

each time it gets a match this counter goes up, and each time it doesn't it goes down. If it goes below 0 the object gets destroyed.

If used, this argument (`reid\_hit\_counter\_max`) defines how long an object can live without getting matched to any detections, before it is destroyed.

"""

def \_\_init\_\_(

self,

distance\_function: Union[str, Callable[["Detection", "TrackedObject"], float]],

distance\_threshold: float,

hit\_counter\_max: int = 15,

initialization\_delay: Optional[int] = None,

pointwise\_hit\_counter\_max: int = 4,

detection\_threshold: float = 0,

filter\_factory: FilterFactory = OptimizedKalmanFilterFactory(),

past\_detections\_length: int = 4,

reid\_distance\_function: Optional[

Callable[["TrackedObject", "TrackedObject"], float]

] = None,

reid\_distance\_threshold: float = 0,

reid\_hit\_counter\_max: Optional[int] = None,

):

self.tracked\_objects: Sequence["TrackedObject"] = []

if isinstance(distance\_function, str):

distance\_function = get\_distance\_by\_name(distance\_function)

elif isinstance(distance\_function, Callable):

warning(

"You are using a scalar distance function. If you want to speed up the"

" tracking process please consider using a vectorized distance"

f" function such as {AVAILABLE\_VECTORIZED\_DISTANCES}."

)

distance\_function = ScalarDistance(distance\_function)

else:

raise ValueError(

"Argument `distance\_function` should be a string or function but is"

f" {type(distance\_function)} instead."

)

self.distance\_function = distance\_function

self.hit\_counter\_max = hit\_counter\_max

self.reid\_hit\_counter\_max = reid\_hit\_counter\_max

self.pointwise\_hit\_counter\_max = pointwise\_hit\_counter\_max

self.filter\_factory = filter\_factory

if past\_detections\_length >= 0:

self.past\_detections\_length = past\_detections\_length

else:

raise ValueError(

f"Argument `past\_detections\_length` is {past\_detections\_length} and should be larger than 0."

)

if initialization\_delay is None:

self.initialization\_delay = int(self.hit\_counter\_max / 2)

elif initialization\_delay < 0 or initialization\_delay >= self.hit\_counter\_max:

raise ValueError(

f"Argument 'initialization\_delay' for 'Tracker' class should be an int between 0 and (hit\_counter\_max = {hit\_counter\_max}). The selected value is {initialization\_delay}.\n"

)

else:

self.initialization\_delay = initialization\_delay

self.distance\_threshold = distance\_threshold

self.detection\_threshold = detection\_threshold

if reid\_distance\_function is not None:

self.reid\_distance\_function = ScalarDistance(reid\_distance\_function)

else:

self.reid\_distance\_function = reid\_distance\_function

self.reid\_distance\_threshold = reid\_distance\_threshold

self.\_obj\_factory = \_TrackedObjectFactory()

def update(

self,

detections: Optional[List["Detection"]] = None,

period: int = 1,

coord\_transformations: Optional[CoordinatesTransformation] = None,

) -> List["TrackedObject"]:

"""

Process detections found in each frame.

The detections can be matched to previous tracked objects or new ones will be created

according to the configuration of the Tracker.

The currently alive and initialized tracked objects are returned

Parameters

----------

detections : Optional[List[Detection]], optional

A list of [`Detection`][norfair.tracker.Detection] which represent the detections found in the current frame being processed.

If no detections have been found in the current frame, or the user is purposely skipping frames to improve video processing time,

this argument should be set to None or ignored, as the update function is needed to advance the state of the Kalman Filters inside the tracker.

period : int, optional

The user can chose not to run their detector on all frames, so as to process video faster.

This parameter sets every how many frames the detector is getting ran,

so that the tracker is aware of this situation and can handle it properly.

This argument can be reset on each frame processed,

which is useful if the user is dynamically changing how many frames the detector is skipping on a video when working in real-time.

coord\_transformations: Optional[CoordinatesTransformation]

The coordinate transformation calculated by the [MotionEstimator][norfair.camera\_motion.MotionEstimator].

Returns

-------

List[TrackedObject]

The list of active tracked objects.

"""

if coord\_transformations is not None:

for det in detections:

det.update\_coordinate\_transformation(coord\_transformations)

# Remove stale trackers and make candidate object real if the hit counter is positive

alive\_objects = []

dead\_objects = []

if self.reid\_hit\_counter\_max is None:

self.tracked\_objects = [

o for o in self.tracked\_objects if o.hit\_counter\_is\_positive

]

alive\_objects = self.tracked\_objects

else:

tracked\_objects = []

for o in self.tracked\_objects:

if o.reid\_hit\_counter\_is\_positive:

tracked\_objects.append(o)

if o.hit\_counter\_is\_positive:

alive\_objects.append(o)

else:

dead\_objects.append(o)

self.tracked\_objects = tracked\_objects

# Update tracker

for obj in self.tracked\_objects:

obj.tracker\_step()

obj.update\_coordinate\_transformation(coord\_transformations)

# Update initialized tracked objects with detections

(

unmatched\_detections,

\_,

unmatched\_init\_trackers,

) = self.\_update\_objects\_in\_place(

self.distance\_function,

self.distance\_threshold,

[o for o in alive\_objects if not o.is\_initializing],

detections,

period,

)

# Update not yet initialized tracked objects with yet unmatched detections

(

unmatched\_detections,

matched\_not\_init\_trackers,

\_,

) = self.\_update\_objects\_in\_place(

self.distance\_function,

self.distance\_threshold,

[o for o in alive\_objects if o.is\_initializing],

unmatched\_detections,

period,

)

if self.reid\_distance\_function is not None:

# Match unmatched initialized tracked objects with not yet initialized tracked objects

\_, \_, \_ = self.\_update\_objects\_in\_place(

self.reid\_distance\_function,

self.reid\_distance\_threshold,

unmatched\_init\_trackers + dead\_objects,

matched\_not\_init\_trackers,

period,

)

# Create new tracked objects from remaining unmatched detections

for detection in unmatched\_detections:

self.tracked\_objects.append(

self.\_obj\_factory.create(

initial\_detection=detection,

hit\_counter\_max=self.hit\_counter\_max,

initialization\_delay=self.initialization\_delay,

pointwise\_hit\_counter\_max=self.pointwise\_hit\_counter\_max,

detection\_threshold=self.detection\_threshold,

period=period,

filter\_factory=self.filter\_factory,

past\_detections\_length=self.past\_detections\_length,

reid\_hit\_counter\_max=self.reid\_hit\_counter\_max,

coord\_transformations=coord\_transformations,

)

)

return self.get\_active\_objects()

@property

def current\_object\_count(self) -> int:

"""Number of active TrackedObjects"""

return len(self.get\_active\_objects())

@property

def total\_object\_count(self) -> int:

"""Total number of TrackedObjects initialized in the by this Tracker"""

return self.\_obj\_factory.count

def get\_active\_objects(self) -> List["TrackedObject"]:

"""Get the list of active objects

Returns

-------

List["TrackedObject"]

The list of active objects

"""

return [

o

for o in self.tracked\_objects

if not o.is\_initializing and o.hit\_counter\_is\_positive

]

def \_update\_objects\_in\_place(

self,

distance\_function,

distance\_threshold,

objects: Sequence["TrackedObject"],

candidates: Optional[Union[List["Detection"], List["TrackedObject"]]],

period: int,

):

if candidates is not None and len(candidates) > 0:

distance\_matrix = distance\_function.get\_distances(objects, candidates)

if np.isnan(distance\_matrix).any():

raise ValueError(

"\nReceived nan values from distance function, please check your distance function for errors!"

)

# Used just for debugging distance function

if distance\_matrix.any():

for i, minimum in enumerate(distance\_matrix.min(axis=0)):

objects[i].current\_min\_distance = (

minimum if minimum < distance\_threshold else None

)

matched\_cand\_indices, matched\_obj\_indices = self.match\_dets\_and\_objs(

distance\_matrix, distance\_threshold

)

if len(matched\_cand\_indices) > 0:

unmatched\_candidates = [

d for i, d in enumerate(candidates) if i not in matched\_cand\_indices

]

unmatched\_objects = [

d for i, d in enumerate(objects) if i not in matched\_obj\_indices

]

matched\_objects = []

# Handle matched people/detections

for (match\_cand\_idx, match\_obj\_idx) in zip(

matched\_cand\_indices, matched\_obj\_indices

):

match\_distance = distance\_matrix[match\_cand\_idx, match\_obj\_idx]

matched\_candidate = candidates[match\_cand\_idx]

matched\_object = objects[match\_obj\_idx]

if match\_distance < distance\_threshold:

if isinstance(matched\_candidate, Detection):

matched\_object.hit(matched\_candidate, period=period)

matched\_object.last\_distance = match\_distance

matched\_objects.append(matched\_object)

elif isinstance(matched\_candidate, TrackedObject):

# Merge new TrackedObject with the old one

matched\_object.merge(matched\_candidate)

# If we are matching TrackedObject instances we want to get rid of the

# already matched candidate to avoid matching it again in future frames

self.tracked\_objects.remove(matched\_candidate)

else:

unmatched\_candidates.append(matched\_candidate)

unmatched\_objects.append(matched\_object)

else:

unmatched\_candidates, matched\_objects, unmatched\_objects = (

candidates,

[],

objects,

)

else:

unmatched\_candidates, matched\_objects, unmatched\_objects = [], [], objects

return unmatched\_candidates, matched\_objects, unmatched\_objects

def match\_dets\_and\_objs(self, distance\_matrix: np.ndarray, distance\_threshold):

"""Matches detections with tracked\_objects from a distance matrix

I used to match by minimizing the global distances, but found several

cases in which this was not optimal. So now I just match by starting

with the global minimum distance and matching the det-obj corresponding

to that distance, then taking the second minimum, and so on until we

reach the distance\_threshold.

This avoids the the algorithm getting cute with us and matching things

that shouldn't be matching just for the sake of minimizing the global

distance, which is what used to happen

"""

# NOTE: This implementation is terribly inefficient, but it doesn't

# seem to affect the fps at all.

distance\_matrix = distance\_matrix.copy()

if distance\_matrix.size > 0:

det\_idxs = []

obj\_idxs = []

current\_min = distance\_matrix.min()

while current\_min < distance\_threshold:

flattened\_arg\_min = distance\_matrix.argmin()

det\_idx = flattened\_arg\_min // distance\_matrix.shape[1]

obj\_idx = flattened\_arg\_min % distance\_matrix.shape[1]

det\_idxs.append(det\_idx)

obj\_idxs.append(obj\_idx)

distance\_matrix[det\_idx, :] = distance\_threshold + 1

distance\_matrix[:, obj\_idx] = distance\_threshold + 1

current\_min = distance\_matrix.min()

return det\_idxs, obj\_idxs

else:

return [], []

class \_TrackedObjectFactory:

global\_count = 0

def \_\_init\_\_(self) -> None:

self.count = 0

self.initializing\_count = 0

def create(

self,

initial\_detection: "Detection",

hit\_counter\_max: int,

initialization\_delay: int,

pointwise\_hit\_counter\_max: int,

detection\_threshold: float,

period: int,

filter\_factory: "FilterFactory",

past\_detections\_length: int,

reid\_hit\_counter\_max: Optional[int],

coord\_transformations: CoordinatesTransformation,

) -> "TrackedObject":

obj = TrackedObject(

obj\_factory=self,

initial\_detection=initial\_detection,

hit\_counter\_max=hit\_counter\_max,

initialization\_delay=initialization\_delay,

pointwise\_hit\_counter\_max=pointwise\_hit\_counter\_max,

detection\_threshold=detection\_threshold,

period=period,

filter\_factory=filter\_factory,

past\_detections\_length=past\_detections\_length,

reid\_hit\_counter\_max=reid\_hit\_counter\_max,

coord\_transformations=coord\_transformations,

)

return obj

def get\_initializing\_id(self) -> int:

self.initializing\_count += 1

return self.initializing\_count

def get\_ids(self) -> Tuple[int, int]:

self.count += 1

\_TrackedObjectFactory.global\_count += 1

return self.count, \_TrackedObjectFactory.global\_count

class TrackedObject:

"""

The objects returned by the tracker's `update` function on each iteration.

They represent the objects currently being tracked by the tracker.

Users should not instantiate TrackedObjects manually;

the Tracker will be in charge of creating them.

Attributes

----------

estimate : np.ndarray

Where the tracker predicts the point will be in the current frame based on past detections.

A numpy array with the same shape as the detections being fed to the tracker that produced it.

id : Optional[int]

The unique identifier assigned to this object by the tracker. Set to `None` if the object is initializing.

global\_id : Optional[int]

The globally unique identifier assigned to this object. Set to `None` if the object is initializing

last\_detection : Detection

The last detection that matched with this tracked object.

Useful if you are storing embeddings in your detections and want to do metric learning, or for debugging.

last\_distance : Optional[float]

The distance the tracker had with the last object it matched with.

age : int

The age of this object measured in number of frames.

live\_points :

A boolean mask with shape `(n\_points,)`. Points marked as `True` have recently been matched with detections.

Points marked as `False` haven't and are to be considered stale, and should be ignored.

Functions like [`draw\_tracked\_objects`][norfair.drawing.draw\_tracked\_objects] use this property to determine which points not to draw.

initializing\_id : int

On top of `id`, objects also have an `initializing\_id` which is the id they are given internally by the `Tracker`;

this id is used solely for debugging.

Each new object created by the `Tracker` starts as an uninitialized `TrackedObject`,

which needs to reach a certain match rate to be converted into a full blown `TrackedObject`.

`initializing\_id` is the id temporarily assigned to `TrackedObject` while they are getting initialized.

"""

def \_\_init\_\_(

self,

obj\_factory: \_TrackedObjectFactory,

initial\_detection: "Detection",

hit\_counter\_max: int,

initialization\_delay: int,

pointwise\_hit\_counter\_max: int,

detection\_threshold: float,

period: int,

filter\_factory: "FilterFactory",

past\_detections\_length: int,

reid\_hit\_counter\_max: Optional[int],

coord\_transformations: Optional[CoordinatesTransformation] = None,

):

if not isinstance(initial\_detection, Detection):

raise ValueError(

f"\n[red]ERROR[/red]: The detection list fed into `tracker.update()` should be composed of {Detection} objects not {type(initial\_detection)}.\n"

)

self.\_obj\_factory = obj\_factory

self.dim\_points = initial\_detection.absolute\_points.shape[1]

self.num\_points = initial\_detection.absolute\_points.shape[0]

self.hit\_counter\_max: int = hit\_counter\_max

self.pointwise\_hit\_counter\_max: int = max(pointwise\_hit\_counter\_max, period)

self.initialization\_delay = initialization\_delay

self.detection\_threshold: float = detection\_threshold

self.initial\_period: int = period

self.hit\_counter: int = period

self.reid\_hit\_counter\_max = reid\_hit\_counter\_max

self.reid\_hit\_counter: Optional[int] = None

self.last\_distance: Optional[float] = None

self.current\_min\_distance: Optional[float] = None

self.last\_detection: "Detection" = initial\_detection

self.age: int = 0

self.is\_initializing: bool = self.hit\_counter <= self.initialization\_delay

self.initializing\_id: Optional[int] = self.\_obj\_factory.get\_initializing\_id()

self.id: Optional[int] = None

self.global\_id: Optional[int] = None

if not self.is\_initializing:

self.\_acquire\_ids()

if initial\_detection.scores is None:

self.detected\_at\_least\_once\_points = np.array([True] \* self.num\_points)

else:

self.detected\_at\_least\_once\_points = (

initial\_detection.scores > self.detection\_threshold

)

self.point\_hit\_counter: np.ndarray = self.detected\_at\_least\_once\_points.astype(

int

)

initial\_detection.age = self.age

self.past\_detections\_length = past\_detections\_length

if past\_detections\_length > 0:

self.past\_detections: Sequence["Detection"] = [initial\_detection]

else:

self.past\_detections: Sequence["Detection"] = []

# Create Kalman Filter

self.filter = filter\_factory.create\_filter(initial\_detection.absolute\_points)

self.dim\_z = self.dim\_points \* self.num\_points

self.label = initial\_detection.label

self.abs\_to\_rel = None

if coord\_transformations is not None:

self.update\_coordinate\_transformation(coord\_transformations)

def tracker\_step(self):

if self.reid\_hit\_counter is None:

if self.hit\_counter <= 0:

self.reid\_hit\_counter = self.reid\_hit\_counter\_max

else:

self.reid\_hit\_counter -= 1

self.hit\_counter -= 1

self.point\_hit\_counter -= 1

self.age += 1

# Advances the tracker's state

self.filter.predict()

@property

def hit\_counter\_is\_positive(self):

return self.hit\_counter >= 0

@property

def reid\_hit\_counter\_is\_positive(self):

return self.reid\_hit\_counter is None or self.reid\_hit\_counter >= 0

@property

def estimate\_velocity(self) -> np.ndarray:

"""Get the velocity estimate of the object from the Kalman filter. This velocity is in the absolute coordinate system.

Returns

-------

np.ndarray

An array of shape (self.num\_points, self.dim\_points) containing the velocity estimate of the object on each axis.

"""

return self.filter.x.T.flatten()[self.dim\_z :].reshape(-1, self.dim\_points)

@property

def estimate(self) -> np.ndarray:

"""Get the position estimate of the object from the Kalman filter.

Returns

-------

np.ndarray

An array of shape (self.num\_points, self.dim\_points) containing the position estimate of the object on each axis.

"""

return self.get\_estimate()

def get\_estimate(self, absolute=False) -> np.ndarray:

"""Get the position estimate of the object from the Kalman filter in an absolute or relative format.

Parameters

----------

absolute : bool, optional

If true the coordinates are returned in absolute format, by default False, by default False.

Returns

-------

np.ndarray

An array of shape (self.num\_points, self.dim\_points) containing the position estimate of the object on each axis.

Raises

------

ValueError

Alert if the coordinates are requested in absolute format but the tracker has no coordinate transformation.

"""

positions = self.filter.x.T.flatten()[: self.dim\_z].reshape(-1, self.dim\_points)

if self.abs\_to\_rel is None:

if not absolute:

return positions

else:

raise ValueError(

"You must provide 'coord\_transformations' to the tracker to get absolute coordinates"

)

else:

if absolute:

return positions

else:

return self.abs\_to\_rel(positions)

@property

def live\_points(self):

return self.point\_hit\_counter > 0

def hit(self, detection: "Detection", period: int = 1):

"""Update tracked object with a new detection

Parameters

----------

detection : Detection

the new detection matched to this tracked object

period : int, optional

frames corresponding to the period of time since last update.

"""

self.\_conditionally\_add\_to\_past\_detections(detection)

self.last\_detection = detection

self.hit\_counter = min(self.hit\_counter + 2 \* period, self.hit\_counter\_max)

if self.is\_initializing and self.hit\_counter > self.initialization\_delay:

self.is\_initializing = False

self.\_acquire\_ids()

# We use a kalman filter in which we consider each coordinate on each point as a sensor.

# This is a hacky way to update only certain sensors (only x, y coordinates for

# points which were detected).

# TODO: Use keypoint confidence information to change R on each sensor instead?

if detection.scores is not None:

assert len(detection.scores.shape) == 1

points\_over\_threshold\_mask = detection.scores > self.detection\_threshold

matched\_sensors\_mask = np.array(

[(m,) \* self.dim\_points for m in points\_over\_threshold\_mask]

).flatten()

H\_pos = np.diag(matched\_sensors\_mask).astype(

float

) # We measure x, y positions

self.point\_hit\_counter[points\_over\_threshold\_mask] += 2 \* period

else:

points\_over\_threshold\_mask = np.array([True] \* self.num\_points)

H\_pos = np.identity(self.num\_points \* self.dim\_points)

self.point\_hit\_counter += 2 \* period

self.point\_hit\_counter[

self.point\_hit\_counter >= self.pointwise\_hit\_counter\_max

] = self.pointwise\_hit\_counter\_max

self.point\_hit\_counter[self.point\_hit\_counter < 0] = 0

H\_vel = np.zeros(H\_pos.shape) # But we don't directly measure velocity

H = np.hstack([H\_pos, H\_vel])

self.filter.update(

np.expand\_dims(detection.absolute\_points.flatten(), 0).T, None, H

)

detected\_at\_least\_once\_mask = np.array(

[(m,) \* self.dim\_points for m in self.detected\_at\_least\_once\_points]

).flatten()

now\_detected\_mask = np.hstack(

(points\_over\_threshold\_mask,) \* self.dim\_points

).flatten()

first\_detection\_mask = np.logical\_and(

now\_detected\_mask, np.logical\_not(detected\_at\_least\_once\_mask)

)

self.filter.x[: self.dim\_z][first\_detection\_mask] = np.expand\_dims(

detection.absolute\_points.flatten(), 0

).T[first\_detection\_mask]

# Force points being detected for the first time to have velocity = 0

# This is needed because some detectors (like OpenPose) set points with

# low confidence to coordinates (0, 0). And when they then get their first

# real detection this creates a huge velocity vector in our KalmanFilter

# and causes the tracker to start with wildly inaccurate estimations which

# eventually coverge to the real detections.

self.filter.x[self.dim\_z :][np.logical\_not(detected\_at\_least\_once\_mask)] = 0

self.detected\_at\_least\_once\_points = np.logical\_or(

self.detected\_at\_least\_once\_points, points\_over\_threshold\_mask

)

def \_\_repr\_\_(self):

if self.last\_distance is None:

placeholder\_text = "\033[1mObject\_{}\033[0m(age: {}, hit\_counter: {}, last\_distance: {}, init\_id: {})"

else:

placeholder\_text = "\033[1mObject\_{}\033[0m(age: {}, hit\_counter: {}, last\_distance: {:.2f}, init\_id: {})"

return placeholder\_text.format(

self.id,

self.age,

self.hit\_counter,

self.last\_distance,

self.initializing\_id,

)

def \_conditionally\_add\_to\_past\_detections(self, detection):

"""Adds detections into (and pops detections away) from `past\_detections`

It does so by keeping a fixed amount of past detections saved into each

TrackedObject, while maintaining them distributed uniformly through the object's

lifetime.

"""

if self.past\_detections\_length == 0:

return

if len(self.past\_detections) < self.past\_detections\_length:

detection.age = self.age

self.past\_detections.append(detection)

elif self.age >= self.past\_detections[0].age \* self.past\_detections\_length:

self.past\_detections.pop(0)

detection.age = self.age

self.past\_detections.append(detection)

def merge(self, tracked\_object):

"""Merge with a not yet initialized TrackedObject instance"""

self.reid\_hit\_counter = None

self.hit\_counter = self.initial\_period \* 2

self.point\_hit\_counter = tracked\_object.point\_hit\_counter

self.last\_distance = tracked\_object.last\_distance

self.current\_min\_distance = tracked\_object.current\_min\_distance

self.last\_detection = tracked\_object.last\_detection

self.detected\_at\_least\_once\_points = (

tracked\_object.detected\_at\_least\_once\_points

)

self.filter = tracked\_object.filter

for past\_detection in tracked\_object.past\_detections:

self.\_conditionally\_add\_to\_past\_detections(past\_detection)

def update\_coordinate\_transformation(

self, coordinate\_transformation: CoordinatesTransformation

):

if coordinate\_transformation is not None:

self.abs\_to\_rel = coordinate\_transformation.abs\_to\_rel

def \_acquire\_ids(self):

self.id, self.global\_id = self.\_obj\_factory.get\_ids()

class Detection:

"""Detections returned by the detector must be converted to a `Detection` object before being used by Norfair.

Parameters

----------

points : np.ndarray

Points detected. Must be a rank 2 array with shape `(n\_points, n\_dimensions)` where n\_dimensions is 2 or 3.

scores : np.ndarray, optional

An array of length `n\_points` which assigns a score to each of the points defined in `points`.

This is used to inform the tracker of which points to ignore;

any point with a score below `detection\_threshold` will be ignored.

This useful for cases in which detections don't always have every point present, as is often the case in pose estimators.

data : Any, optional

The place to store any extra data which may be useful when calculating the distance function.

Anything stored here will be available to use inside the distance function.

This enables the development of more interesting trackers which can do things like assign an appearance embedding to each

detection to aid in its tracking.

label : Hashable, optional

When working with multiple classes the detection's label can be stored to be used as a matching condition when associating

tracked objects with new detections. Label's type must be hashable for drawing purposes.

embedding : Any, optional

The embedding for the reid\_distance.

"""

def \_\_init\_\_(

self,

points: np.ndarray,

scores: np.ndarray = None,

data: Any = None,

label: Hashable = None,

embedding=None,

):

self.points = validate\_points(points)

self.scores = scores

self.data = data

self.label = label

self.absolute\_points = self.points.copy()

self.embedding = embedding

self.age = None

def update\_coordinate\_transformation(

self, coordinate\_transformation: CoordinatesTransformation

):

if coordinate\_transformation is not None:

self.absolute\_points = coordinate\_transformation.rel\_to\_abs(

self.absolute\_points

)