Cel laboratorium

Celem laboratorium była poprawa zaimplementowanych 3 wybranych biegunowych deskryptorów kształtu przedstawionych na wykładzie przez prowadzącego.

Przebieg laboratorium

Prowadzący przedstawiał biegunowe deskryptory kształtu, z których należało wybrać 3 dowolne, a następnie je zaimplementować. Ich działanie należało przetestować za pomocą porównania odległości euklidesowych wyniki pomiędzy nauczonymi deskryptorami a testowymi. Z obrazów z każdej klasy wybrano 5 obrazów uczących oraz 5 obrazów testowych.

Do zaimplementowania prostych deskryptorów wykorzystano język Python. Zaimplementowano 3 poniższe biegunowe deskryptory:

- CDP
- LogPol,
- Centroid PDH.

Do ich przetestowania wykorzystano implementację klasyfikatora sieci neuronowej z biblioteki sklearn.

Wybrane numery obrazów z każdej klasy:

```
representative_images_number: dict = {
   'gambles-quail': [6, 9, 10, 2, 1],
   'glossy-ibis': [10, 9, 8, 2, 3],
   'greator-sage-grous': [5, 6, 9, 10, 8],
   'hooded-merganser': [1, 2, 5, 9, 4],
   'indian-vulture': [4, 2, 6, 8, 9],
   'jabiru': [1, 6, 7, 10, 5],
   'king-eider': [8, 2, 5, 7, 6],
   'long-eared-owl': [4, 3, 1, 8, 9],
   'tit-mouse': [8, 5, 6, 10, 1],
   'touchan': [6, 2, 1, 10, 9],
}
```

Kod programu

Do implementacji deskryptorów stworzono abstrakcyjną klasę deskryptora:

```
class AbstractDescriptor(ABC):
   def __init__(self):
       self.points_count: int = ...

   @abstractmethod
   def descript_image(self, image: np.array) -> AbstractDescriptor:
       ...

   @abstractmethod
   def calc_distance_to_other_descriptor(self, descriptor: AbstractDescriptor) ->
float:
       ...
```

```
    @abstractmethod
    def _calc_centroid(self, points: np.array):
        ...

    @staticmethod
    def _find_contour_points(image: np.array):
        return np.array(list(zip(*np.where(image == 0))))

    def _select_points_from_array(self, array: np.array) -> np.array:
        indexes_distribution = np.linspace(0, array.shape[0] - 1,
self.points_count).astype(int)
        distances = [array[0]]

        if indexes_distribution[2] - indexes_distribution[1] <= 1:
            return array

        index_iter = 1

        for index in indexes_distribution[1:]:
            prev_index = indexes_distribution[index_iter - 1]
            between_values = array[prev_index + 1:index]

            mean_value = np.mean(between_values)

            distances[index_iter - 1] = mean_value
            distances.append(mean_value)

            index_iter += 1

        return np.array(distances)
```

Klasa deskryptora biegunowego CDP:

```
class CDP(AbstractDescriptor):
    def __init__(self, points_count: int = 200):
        super().__init__()

        self.points_count: int = points_count

        self.centroid = None
        self.distances = None

    def descript_image(self, image: np.array) -> CDP:
        points = self._find_contour_points(image)
        self.centroid = self._calc_centroid(points)
        self.distances = self.__calc_distances_to_centroid(points)
        self.distances = self._select_points_from_array(self.distances)

        return self

    def calc_distance_to_other_descriptor(self, descriptor: CDP) -> float:
        distances = self.distances - descriptor.distances
        distances = np.sqrt(distances ** 2)

        return np.mean(distances)

    def _calc_centroid(self, points: np.array) -> np.array:
        y_sum = np.sum(points[:, 0])
        x_sum = np.sum(points[:, 1])
```

```
        return np.array([x_sum / points.size, y_sum / points.size])

    def __calc_distances_to_centroid(self, points: np.array) -> np.array:
        distances = np.empty(points.shape[0])

        for point_iter, point in enumerate(points):
            distances[point_iter] = self.__calc_distance_to_centroid(point)

        return distances

    def __calc_distance_to_centroid(self, point: np.array) -> float:
        return np.sqrt((point[1] - self.centroid[1]) ** 2 + (point[0] -
self.centroid[0]) ** 2)
```

Klasa deskryptora biegunowego LogPol:

```
class LogPol(AbstractDescriptor):
    """
    Converts the original image (x, y) into another (p, w) in which the angular
coordinates are placed on the vertical
    axis and the logarithm of the radius coordinates are placed on the horizontal
one (furthermore a normalization has
    to be carried out in order to implement the transformation).
    """
    def __init__(self, points_count: int = 200):
        super().__init__()

        self.points_count: int = points_count

        self.centroid = None
        self.p = None  # logarytm współrzędnych promienia
        self.w = None  # współrzędne kątowe

    def descript_image(self, image: np.array) -> LogPol:
        points = self._find_contour_points(image)
        self.centroid = self._calc_centroid(points)
        self.p, self.w = self.__calc_distances_to_centroid(points)

        self.p = self._select_points_from_array(self.p)
        self.w = self._select_points_from_array(self.w)

        return self

    def calc_distance_to_other_descriptor(self, descriptor: LogPol) -> float:
        distances_p = np.sqrt((self.p - descriptor.p) ** 2)
        distances_w = np.sqrt((self.p - descriptor.w) ** 2)

        return np.sqrt((np.mean(distances_p) - np.mean(distances_w)) ** 2)

    def _calc_centroid(self, points: np.array):
        y_sum = np.sum(points[:, 0])
        x_sum = np.sum(points[:, 1])

        return np.array([x_sum / points.size, y_sum / points.size])

    def __calc_distances_to_centroid(self, points: np.array) -> Tuple:
        p = np.empty(points.shape[0])
        w = np.empty(points.shape[0])

        for point_iter, point in enumerate(points):
```

```
            p[point_iter] = self.__calc_p_for_point(point)
            w[point_iter] = self.__calc_w_for_point(point)

        return p, w

    def __calc_p_for_point(self, point: np.array):
        return np.log(np.sqrt((point[1] - self.centroid[1]) ** 2 + (point[0] -
self.centroid[0]) ** 2))

    def __calc_w_for_point(self, point: np.array):
        return np.arctan((point[0] - self.centroid[0]) / (point[1] -
self.centroid[1]))
```

Klasa deskryptora biegunowego Centroid PDH:

```
class CentroidPDH(AbstractDescriptor):
    def __init__(self, points_count: int = 5):
        super().__init__()

        self.points_count: int = points_count  # r in PDH terminology

        self.centroid = None
        self.h = None

    def descript_image(self, image: np.array) -> CentroidPDH:
        points = self._find_contour_points(image)
        self.centroid = self._calc_centroid(points)

        oi = self.__calc_o(points)
        pi = self.__calc_p(points)

        for oi_iter, oi_val in enumerate(oi):
            oi[oi_iter] = np.floor(oi_val) if oi_val - np.floor(oi_val) < 0.5 else
np.ceil(oi_val)

        oi, pi = self.__sort_oi_and_pi_by_oi(oi, pi)

        pk = self.__calc_pk_from_sorted_oi_and_pi(oi, pi)
        pk = self.__normalize_vector(pk)

        lk = self.__calc_lk(pk)
        lk = self.__normalize_vector(lk)

        self.h = self.__calc_h(lk)

        return self

    def calc_distance_to_other_descriptor(self, descriptor: CentroidPDH) -> float:
        if self.h.size < descriptor.h.size:
            return np.mean(self.h)

        distances = self.h[:descriptor.h.shape[0]] - descriptor.h
        distances = np.sqrt(distances ** 2)

        return np.mean(distances)

    def _calc_centroid(self, points: np.array):
        y_sum = np.sum(points[:, 0])
        x_sum = np.sum(points[:, 1])
```

```
        return np.array([x_sum / points.size, y_sum / points.size])

    def __calc_o(self, points: np.array):
        o = np.empty(points.shape[0])

        for point_iter, point in enumerate(points):
            o[point_iter] = np.arctan((point[0] - self.centroid[0]) / (point[1] -
self.centroid[1]))

        return o

    def __calc_p(self, points: np.array):
        points = points.copy().astype('float')

        for centroid_iter, centroid_val in enumerate(self.centroid):
            points[:, centroid_iter] -= centroid_val
            points[:, centroid_iter] **= 2

        return points.sum(axis=1)

    @staticmethod
    def __sort_oi_and_pi_by_oi(oi: np.array, pi: np.array):
        return zip(*sorted(zip(oi, pi)))

    @staticmethod
    def __calc_pk_from_sorted_oi_and_pi(oi: np.array, pi: np.array):
        uniques_oi = np.unique(oi)

        buckets = np.empty(uniques_oi.size, dtype=object)
        buckets[...] = [[] for _ in range(buckets.shape[0])]

        for oi_iter, oi_val in enumerate(oi):
            bucket_number = np.where(uniques_oi == oi_val)[0][0]
            pi_val = pi[oi_iter]

            buckets[bucket_number].append(pi_val)

        return np.array(list(map(lambda pi_list: np.max(pi_list), buckets)))

    @staticmethod
    def __normalize_vector(vector: np.array):
        max_val = np.max(vector)

        return vector / max_val

    def __calc_lk(self, pk: np.array):
        lk = np.empty(pk.size)

        for pk_iter, pk_val in enumerate(pk):
            lk[pk_iter] = self.points_count if pk_val == 1 else
np.floor(self.points_count * pk_val)

        return lk

    @staticmethod
    def __calc_h(lk: np.array):
        h = np.empty(lk.size)

        for lk_iter, lk_val in enumerate(lk):
```

```
            h[lk_iter] = 1 if lk_iter == lk_val else 0

        return h
```

Rezultaty

	score
CDP	42.0000%
LogPol	10.0000%
CentroidPDH	8.0000%

	class	CDP	LogPol	CentroidPDH	score
0	1	60.0000%	0.0000%	80.0000%	46.6667%
1	2	80.0000%	0.0000%	0.0000%	26.6667%
2	3	80.0000%	100.0000%	0.0000%	60.0000%
3	4	20.0000%	0.0000%	0.0000%	6.6667%
4	5	20.0000%	0.0000%	0.0000%	6.6667%
5	6	0.0000%	0.0000%	0.0000%	0.0000%
6	7	0.0000%	0.0000%	0.0000%	0.0000%
7	8	80.0000%	0.0000%	0.0000%	26.6667%
8	9	60.0000%	0.0000%	0.0000%	20.0000%
9	10	20.0000%	0.0000%	0.0000%	6.6667%