Implementation Documentation: Group Coding Topic c1

Group Coding Topic c1 Members

- Preinsperger Christopher, e01529038
- Maliakel Paul Joe, e12012422
- Kowarsch Florian, e11777780

GitHub: https://github.com/ChrisPrein/TUW_SelfOrganizingSystems_WS2021

Topographic error

The Topographic error for the whole map is defined as percentage of data samples for which the 2nd-best matching unit is not adjacent (in output space) to the best-matching unit (using 4 or 8 neighbors). A unit-wise value can be obtained be either counting the number of faulted best matching input vectors per unit (default approach) or be calculating the percentage of input vectors that do not have a their 2nd-best matching unit in the neighborhood (percentage approach).

Our code implements both approaches in the following way:

For each unit it is count how often the unit is the best matching for an input vector (*n_best_matching*) and how often the second best matching unit of this input vector is not in this unit's neighborhood (*n_2nd_best_not_neighbor*). The topographic error for default approach is then defined as:

```
1 - ((n_2nd_best_not_neighbor - min(n_2nd_best_not_neighbor)) /
(max(n_2nd_best_not_neighbor) - min(n_2nd_best_not_neighbor)) * 0.6 + 0.2)
```

The topographic error for the percentage approach is then defined as:

```
n_2nd_best_not_neighbor / n_best_matching
```

In detail the following calculations are done for each input vector:

First the distance of the input vector to all units is calculated. Than the best and second best matching unit's index is determined using numpy's argmin. After that the indexes are converted to 2D coordinates. The coordinates are used to check if the second best matching unit is in the best matching unit's neighorhood. If this is false *n_2nd_best_not_neighbor* is increased by *1* for the best matching unit.

```
    #calculate distance between current input vector and all units
    vec_to_weights_dis = np.sqrt(np.sum(np.power(self._main._weights - vector, 2),
axis=1))
```

```
    #extract first and 2nd best unit index
    idx_best = np.argmin(vec_to_weights_dis)
    vec_to_weights_dis[idx_best] = np.amax(vec_to_weights_dis)
    idx_2nd_best = np.argmin(vec_to_weights_dis)

    pos_best = self._caclulate_position_from_index(idx_best)
    pos_2nd_best = self._caclulate_position_from_index(idx_2nd_best)

    n_best_matching[idx_best] += 1
     if not self._is_in_neighborhood(pos_best, pos_2nd_best, neighborhood_type):
        n_2nd_best_not_neighbor[idx_best] += 1
```

Based on *neighborhood_type* the neighborhood membership of the second best matching unit is determined by the row- & columwise absolute difference:

```
    def _is_in_neighborhood(self, pos_unit: Tuple[int, int], pos_other_unit:
Tuple[int, int], neighborhood_type: int) -> bool:
        row_dif = abs(pos_unit[1] - pos_other_unit[1])
        col_dif = abs(pos_unit[0] - pos_other_unit[0])

        if neighborhood_type == 4:  # 4-Unit Neighbhorhood
            return (row_dif == 0 and col_dif == 1) or (col_dif == 0 and row_dif ==
1)
        elif neighborhood_type == 8:  # 8-Unit Neighbhorhood
            return row_dif <= 1 and col_dif <= 1
```

Based on *approach* the resulting topographic error per unit is calculated differently. For *approach* == 0 the default calculation as defined in the Java SOM toolbox is used, which is the number of best matching input vectors for which the second best matching unit is not neighboring the current unit. This value is min-max-scaled and inverted.

```
     if approach == 0:
        n_not_max = np.amax(n_2nd_best_not_neighbor)
        n_not_min = np.amin(n_2nd_best_not_neighbor)
        n_not_range = n_not_max - n_not_min
        units_with_error_mask = n_2nd_best_not_neighbor > 0

        # using the same scaling as defined in the Java SOM toolbox --> min-max
scaled
        topoerror[units_with_error_mask] = 1.0 -
((n_not_neighbor_matching[units_with_error_mask] - n_not_min) / n_not_range * 0.6
+ 0.2)
        topoerror[units_with_data_mask] = np.array(1.0 - (n_2nd_best_not_neighbor
/ n_best_matching))[units_with_data_mask]
```

For *approach* == 1 the percentage of input vectors that do not have their second best matching unit in the current unit's neighborhood is computed:

```
    elif approach == 1:
        units_with_data_mask = n_best_matching > 0
        # using different aproach: measuring the relativ number of matching vectors that do not have 2nd best unit as neighboor
        topoerror[units_with_data_mask] = np.array(n_2nd_best_not_neighbor / n_best_matching)[units_with_data_mask]
```

The idea of the percentage approach is to be less effected by the unit's density. E.g: A high topographic error is more likely for a unit with high number of mapped input vectors. This effect should be dampened if only the percentage of mismatched input vectors is considered (= normalization by density).

Intrinsic distance

The Intrinsic Distance is a quality metric that tries to express how well the topology of the input data is preserved. It combines both aspects from the Quantization Error and from the Topographic Error. To Compute the Intrinsic distance we have to compute the BMU and second BMU for each sample. Then we take the shortest path from the BMU to the second BMU and sum the Mapping distances of all units along the path including the BMU und second BMU.

Our code implements this the following way:

For each Input we search the BMU and second BMU using the following code.

```
    #get the distances between input and weights
    vec_to_weights_dis: np.ndarray = np.sqrt(np.sum(np.power(self._main._weights - vector, 2), axis=1)).reshape(self._main._m,self._main._n)

    #select the unit with the minimum distance as bmu
    idx_best: Tuple[int, int] = np.unravel_index(np.argmin(vec_to_weights_dis, axis=None), vec_to_weights_dis.shape)

    #set value for bmu in distance vector to highest distance to exclude it
    vec_to_weights_dis_wihtout_best: np.ndarray = np.copy(vec_to_weights_dis)
    vec_to_weights_dis_wihtout_best[idx_best] = np.amax(vec_to_weights_dis)

    #select the unit with the minimum distance as second bmu
    idx_2nd_best: Tuple[int, int] = np.unravel_index(np.argmin(vec_to_weights_dis_wihtout_best, axis=None), vec_to_weights_dis_wihtout_best.shape)
```

After that we have to calculate the distance along the shortest path and add it to the intrinsic distance matrix at the position of the bmu.

```
    intrinsic_distance[idx_best] += self._calculate_distance(distances=vec_to_weights_dis, idx_best=idx_best, idx_2nd_best=idx_2nd_best)
```

To calculate the distance we have to setup a graph including all units as vertices. The vertices are connected to their neighbours according to their position within the grid. The weights of the edges depend on the direction. As a weight the distance between the weightvector of the targetvertex and the inputvector is used. After the graph is constructed dijkstra is used to find the path with the minimum sum of distances. The sum of the distances of each unit containted in the minimum path is returned.

```
    graph: Graph = Graph(self._generate_edges(distances))

    path: List[Tuple[int, int]] = graph.dijkstra(idx_best, idx_2nd_best)

    return sum([distances[vertex] for vertex in path])
```

The edges are generated using the following code.

```
    edges = []
    n_col = distances.shape[0]
    n_row = distances.shape[1]

    #all indices are created using two for loops. The edges are created using all possible neighbors in a neighborhood of size 4.
    for x in range(n_col):
        for y in range(n_row):
            if x < n_col - 1:
                neighbor = (x + 1, y)
                edges.append(((x, y), neighbor, distances[neighbor]))
            if x > 0:
                neighbor = (x - 1, y)
                edges.append(((x, y), neighbor, distances[neighbor]))

            if y < n_row - 1:
                neighbor = (x, y + 1)
                edges.append(((x, y), neighbor, distances[neighbor]))
            if y > 0:
                neighbor = (x, y - 1)
                edges.append(((x, y), neighbor, distances[neighbor]))

    return edges
```

The implementation of the graph has been found in the web (https://morioh.com/p/ec5873a4c9f5) and has been created by Shubham Ankit.