NN_MarkusWiktorin_041217

December 9, 2017

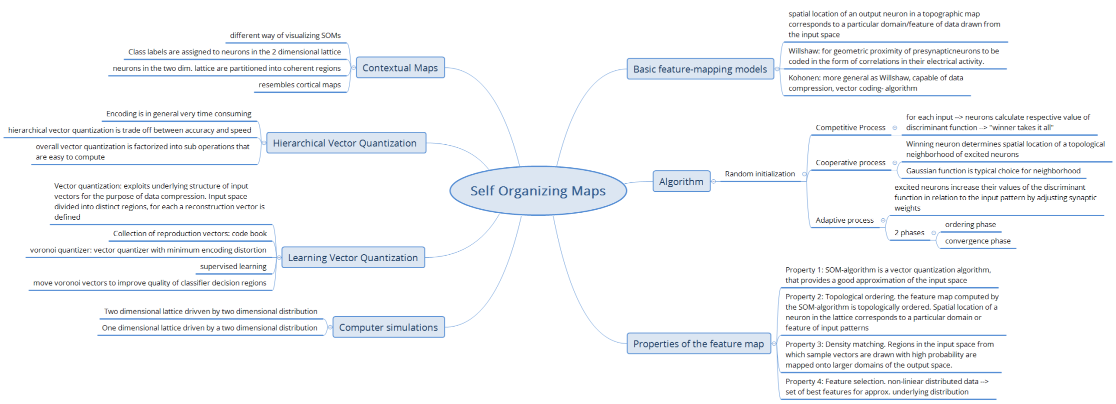
1 Assignment 9

Linda Koine, Jens Weimann, Markus Wiktorin

1.1 1

```
In [2]: from IPython.display import Image
        Image("soms.png")
```

Out[2]:



1.2 2

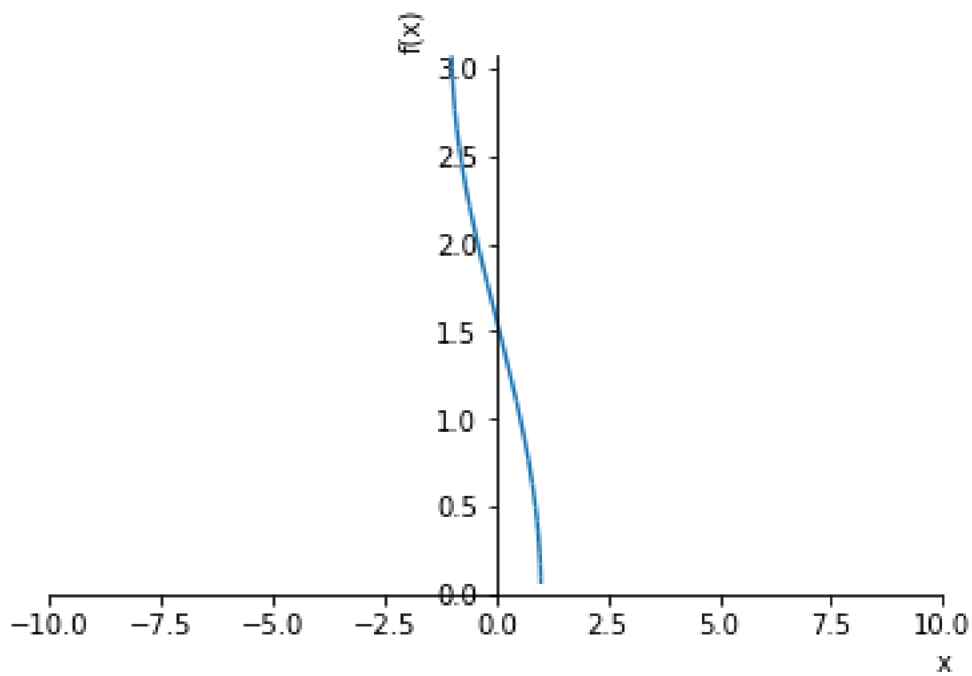
```
In [27]: import sympy as sp
         import numpy as np
         from sympy.plotting import plot
         sp.init_printing()
```

Show that in the SOM algorithm the winner neuron for an input x is that neuron k whose weight vector w_k maximizes the inner product $< w_k, x >$ of x and w_k, take x and w_k as normalized.

Provided the input vector and weight vector are normalized, minimizing the Euclidean distance is equivalent to maximizing the dot product. The largest dot product corresponds to the smallest angle between the vectors.

$\cos(\alpha) = \frac{<w_k,x>}{|w_k|*|x|}$
angle $= \alpha = \arccos(\frac{<w_k,x>}{|w_k|*|x|})$
As shown in the following plot $\arccos(x)$ gets min if x is equal to 1.

```
In [22]: x=sp.symbols("x")
         h= sp.acos(x)
         plot(h)
```




```
Out[22]: <sympy.plotting.plot.Plot at 0x7f5c0093c518>
```

If the angle in equal to 0 the dot product is equal to 1 and the vectors point in the exact same direction. The smaller the dot product gets, the bigger the angle is. The angle is 180 degree, if the vectors point in the opposide directions and the dot product is -1.

Assume the input vector x and the weight vector w_k of neuron j are most simular. It follows, that the angle of those vectors is minimal and so the dot product $< x, w_k >$ is maximal.

1.3 3

```
In [82]: import time
         import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline
```

```
In [184]: class SOM:
              def __init__(self, input_dimension, map_size, distance_function, lear
```

```
        self.input_dimension = input_dimension
        self.map_size = map_size
        self.distance_function = distance_function
        self.learning_rate = learning_rate

        if weights is None:
            self.weights = np.random.rand(map_size, input_dimension)
        else:
            self.weights = weights

    def plot_weights(self):
        plt.figure()
        plt.axis("equal")
        x = self.weights[:,0]
        y = self.weights[:,1]
        plt.plot(x, y, c="r")
        plt.scatter(self.weights[:,0], self.weights[:,1], c="r")

    def get_closest_weight_index(self, sample, exclude_indices=None):
        min_idx = 0
        while (exclude_indices != None) and (min_idx in exclude_indices):
            min_idx = min_idx + 1

        min_distance = self.distance_function(self.weights[min_idx], samp
        idx = 0
        for weight in self.weights:
            distance = self.distance_function(weight, sample)
            if distance < min_distance:
                if exclude_indices == None or idx not in exclude_indices:
                    min_distance = distance
                    min_idx = idx
            idx = idx + 1
        return min_idx

    def train(self, sample_set, distance_treshold, epsilon, sigma):
        changed = 100
        while changed > epsilon:
            changed = 0
            for sample in sample_set:
                min_idx = self.get_closest_weight_index(sample)
                for i in range(len(self.weights)):
                    distance = self.distance_function(self.weights[min_ic
                    if i == min_idx or distance < distance_treshold:
                        direction = sample - self.weights[i]
                        delta = self.learning_rate * np.exp(-distance**2
                        self.weights[i] = self.weights[i] + delta
                        changed = changed + np.sum(delta)
```

```
    def fit_to_points(self, sample_set):
        altered_weights = []
        for sample in sample_set:
            min_idx = self.get_closest_weight_index(sample, altered_weigh
            self.weights[min_idx] = sample
            altered_weights.append(min_idx)
```

```
In [68]: def simple_distance_function(x, y):
             return abs(x - y)
```

```
In [183]: input = [0.1, 0.2, 0.4, 0.5]
          learning_rate = 0.1
          input_dimension = 1
          nodes = 2
          epsilon = 0.01
          distance_treshold = 0

          weights1 = np.array([0.15, 0.45])
          weights2 = np.array([0.3, 0.9])

          som = SOM(input_dimension, nodes, simple_distance_function, learning_rate
          som.train(input, distance_treshold, epsilon, 1)
          print(som.weights)

          som = SOM(input_dimension, nodes, simple_distance_function, learning_rate
          som.train(input, distance_treshold, epsilon, 1)
          print(som.weights)
```

```
[ 0.1505  0.4505]
[ 0.30632  0.9    ]
```

1.4 4

```
In [29]: def one_dim_circle_distance(p1, p2):
             return np.linalg.norm(p1 - p2)
```

```
In [30]: def plot_problem(samples, weights):
             plt.figure()
             plt.axis("equal")
             plt.plot(weights[:,0], weights[:,1], c="r")
             plt.scatter(weights[:,0], weights[:,1], c="orange")
             plt.scatter(samples[:,0], samples[:,1], c="g")
             plt.legend(["solution", "weights", "cities"])
```

```
In [139]: def tsp_distance_som(weights):
              distance = 0
              start = weights[0]
              last = start
```

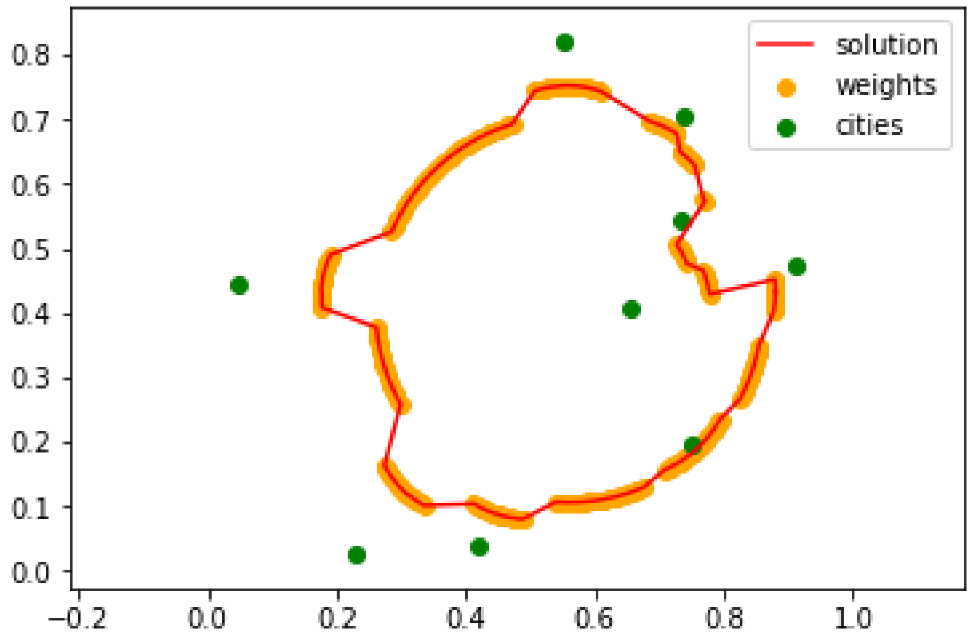
```
        for weight in weights[1:]:
            distance = distance + np.linalg.norm(last - weight)
            last = weight
        distance = distance + np.linalg.norm(last - start)
        return distance

    def tsp_distance(coords, order):
        distance = 0
        last = 0
        for i in range(1, len(order)):
            distance = distance + np.linalg.norm(coords[order[last]] - coords
            last = i
        distance = distance + np.linalg.norm(coords[order[0]] - coords[order
        return distance
```

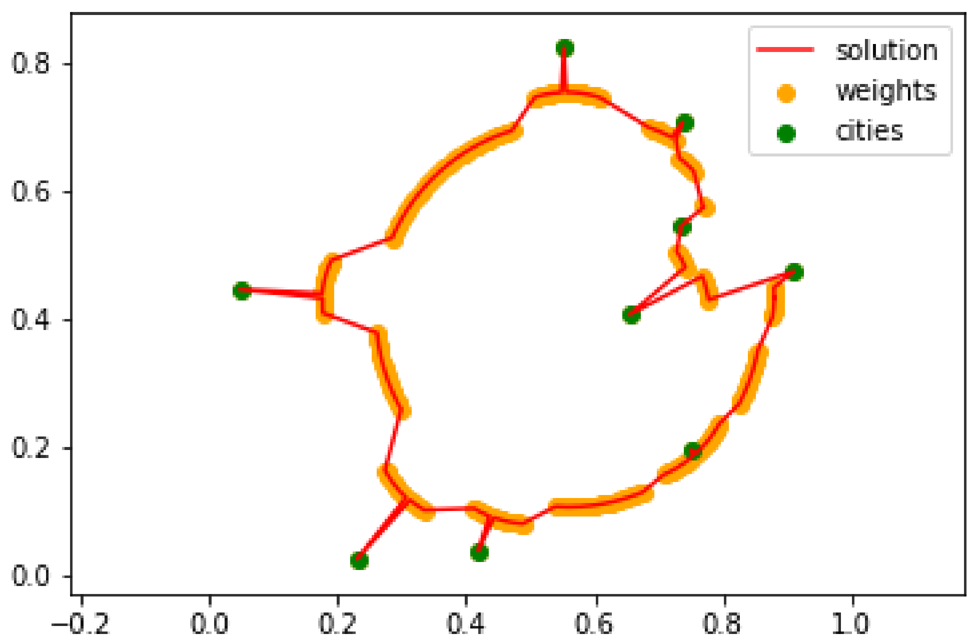
In [163]:
```
import itertools
def tsp_brut_force(cities, plot=True):
    permutations = itertools.permutations(range(len(cities)))
    min_dist = 9999999999999999
    for order in permutations:
        dist = tsp_distance(cities, order)
        if dist < min_dist:
            min_dist = dist
            min_order = order

    if plot:
        plt.figure()
        plt.axis("equal")
        x = [cities[i][0] for i in min_order]
        y = [cities[i][1] for i in min_order]
        x.append(cities[min_order[0]][0])
        y.append(cities[min_order[0]][1])
        plt.title("Best solution with brut force")
        plt.plot(x, y, c="r")
        plt.scatter(x, y, c="g")
        plt.legend(["solution", "cities"])
        plt.show()
    return min_order
```

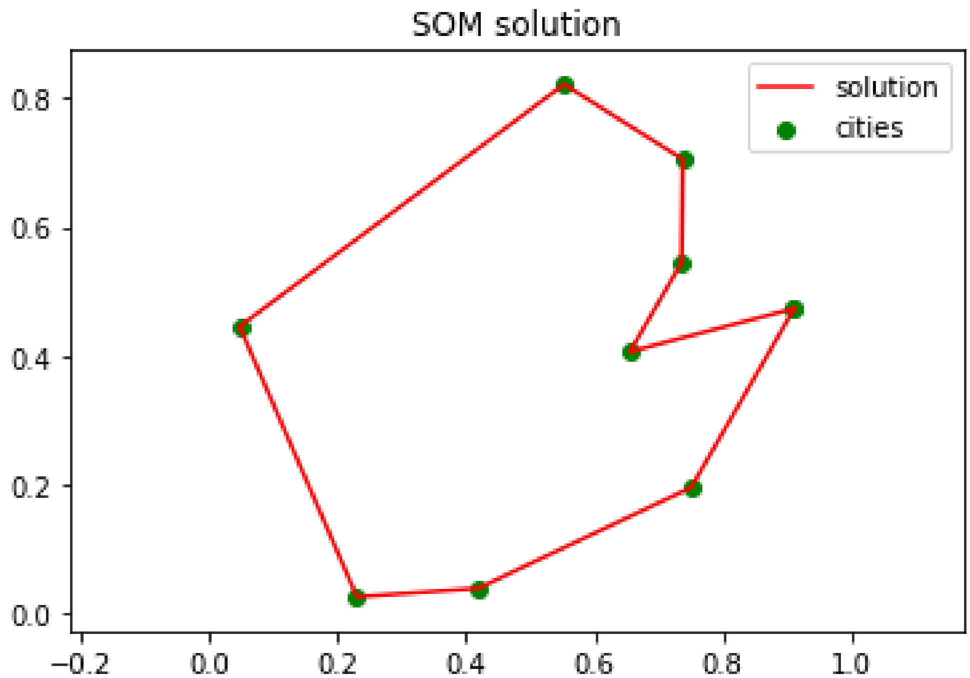
In [192]:
```
num_cities = 9
circle_points = 300
learning_rate = 0.4
input_dimension = 2
distance_treshold = 0.1
training_epsilon = 0.05
sigma = 2

cities = np.random.rand(num_cities, input_dimension)
```

```
mean_x = np.mean(cities, 0)[0]
mean_y = np.mean(cities, 0)[1]

angles = np.linspace(0, 2 * np.pi, circle_points)
x = 0.3 * np.cos(angles) + mean_x
y = 0.3 * np.sin(angles) + mean_y
weights = np.column_stack((x, y))

start_time = time.time()
som = SOM(input_dimension, circle_points, one_dim_circle_distance, learni
som.train(cities, distance_treshold, training_epsilon, sigma)
plot_problem(cities, som.weights)
som.fit_to_points(cities)
elapsed_time = time.time() - start_time
plot_problem(cities, som.weights)

cities_in_order = []
for weight in som.weights:
    if weight in cities:
        cities_in_order.append(weight)

cities_in_order = np.array(cities_in_order)

plt.figure()
plt.axis("equal")
x = cities_in_order[:,0]
y = cities_in_order[:,1]
x = np.append(x, cities_in_order[0,0])
y = np.append(y, cities_in_order[0,1])
plt.title("SOM solution")
plt.plot(x, y, c="r")
plt.scatter(x, y, c="g")
plt.legend(["solution", "cities"])

plt.show()
print("Solved in", elapsed_time, "seconds")
our_distance = tsp_distance_som(cities_in_order)

if num_cities < 10:
    print("Solve TSP brut force...")
    start_time = time.time()
    best_order = tsp_brut_force(cities)
    elapsed_time = time.time() - start_time
    print("Solved in", elapsed_time, "seconds")
    best_distance = tsp_distance(cities, best_order)

print("Our Distance: ", our_distance)
if num_cities < 10:
```

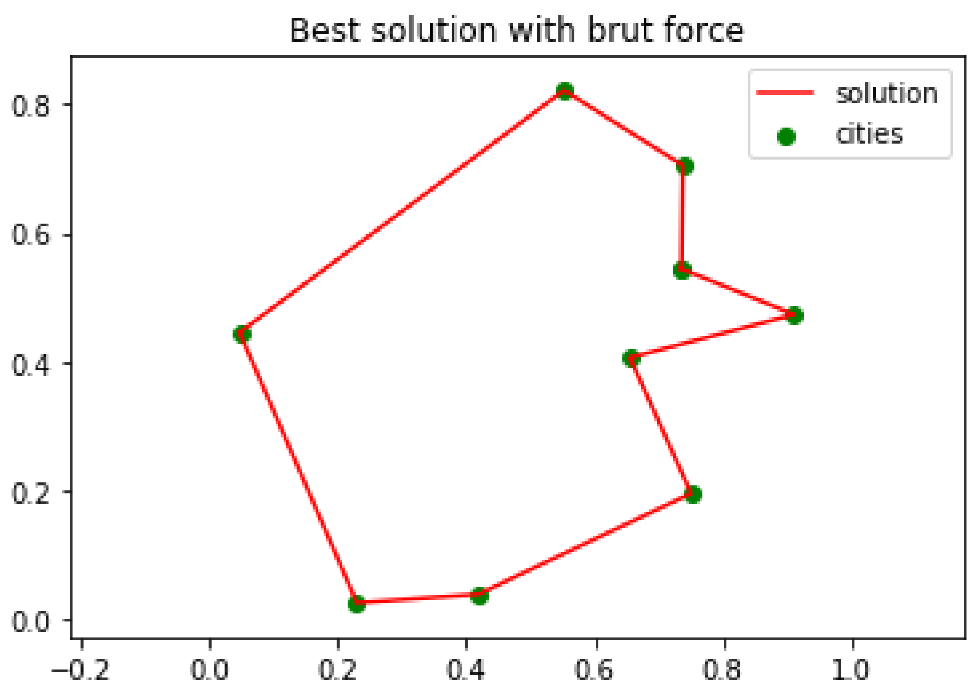
```
print("Best distance: ", best_distance)
print("How close to optimum: ", best_distance / our_distance * 100, '
```








```
Solved in 0.19530510902404785 seconds
Solve TSP brut force...
```



```
Solved in 40.70014977455139 seconds
Our Distance:  2.77011910124
Best distance:  2.7099071492
How close to optimum:  97.8263767785 %
```

In [ ]: