Chem277B: Machine Learning Algorithms

Homework assignment #2: Simulated Annealing

In [184…

```python
import numpy as np
from numpy import linalg as LA
import time
from pylab import *
import matplotlib.pyplot as plt
import math
import scipy
import pandas as pd
import numba
```

1. Classical simulated annealing.

(a) The cooling schedule can be calculated using the linear cooling equation T(t+1) = T(t) - α.

T(initial) = 3000 K. α = 0.5.

When T(end) = 30 K, time t = [T(initial) - T(end)] / α = 5940.

When T(end) = 10 K, time t = [T(initial) - T(end)] / α = 5980.

The tests were repeated 3 times for each linear cooling condition and the results are summarized below:

Schwefel Optimization Eval	Linear Cooling 1 (30K)	Linear Cooling 2 (10K)
Test 1	4753.09	4363.71
Test 2	4562.07	4691.77
Test 3	4224.17	4564.85
Average	4513.11	4540.11

From the results there is no significant difference between the solutions when cooling to different temperatures.

In [49]:

```python
#  418.8929 * D - Sigma_0^D(x_i * (sine(sqrt(x_i))))
def Schwefel(x):
    dim = len(x)    #  find dimension of the input array
    return 418.9829 * dim - np.sum(x * np.sin(np.sqrt(abs(x))))
```

In [50]:
```
# SA function from the reference
def SA(solution,evaluation,delta,boundary,cooling_schedule):
    """ Simulated Annealing for minimization
    solution: np.array. Initial guess of solution
    evaluation: func. Function to evaluate solution
    delta: float. Magnitude of random displacement
    boundary: array of int/float. [lowerbound,upperbound]
    cooling_schedule: np.array. An array of tempretures for simulated annealing
    """
    best_solution=solution.copy()
    lowest_eval=evaluation(best_solution)
    for idx,temp in enumerate(cooling_schedule):
        if idx%1000==0:
            print("%d/%d   temp:%f"%(idx,len(cooling_schedule),temp))
        for n in range(len(solution)):
            trial=solution.copy()
            trial[n]+=delta*(2*np.random.random()-1)     # Visitation function to determine random displacement
            if trial[n]>=boundary[0] and trial[n]<=boundary[1]:
                #fill in acceptance criterion
                if np.exp(-(evaluation(trial) - evaluation(solution))/temp) > np.random.random():
                    solution=trial
                    if evaluation(solution)<lowest_eval:
                        #update solution here
                        best_solution = solution.copy()
                        lowest_eval = evaluation(solution)
    return {"solution":best_solution,"evaluation":lowest_eval}

solution = np.random.random(10) * 1000 - 500   # Create a 10-d array within [-500, 500)
linear_cooling_1 = np.linspace(3000,30,5940)
linear_cooling_2 = np.linspace(3000,10,5980)
boundary = np.array([-500, 500])
delta = 0.5
```

In [41]:
```
# Test linear_cooling_1 schedule 3 times
linear_cooling_1_results = []
test1 = SA(solution, Schwefel, delta, boundary, linear_cooling_1)
linear_cooling_1_results.append(test1['evaluation'])
test2 = SA(solution, Schwefel, delta, boundary, linear_cooling_1)
linear_cooling_1_results.append(test2['evaluation'])
test3 = SA(solution, Schwefel, delta, boundary, linear_cooling_1)
linear_cooling_1_results.append(test3['evaluation'])
print(linear_cooling_1_results, np.mean(np.array(linear_cooling_1_results)))
```

```
0/5940   temp:3000.000000
1000/5940   temp:2499.915811
2000/5940   temp:1999.831621
3000/5940   temp:1499.747432
4000/5940   temp:999.663243
5000/5940   temp:499.579054
0/5940   temp:3000.000000
1000/5940   temp:2499.915811
2000/5940   temp:1999.831621
3000/5940   temp:1499.747432
4000/5940   temp:999.663243
5000/5940   temp:499.579054
0/5940   temp:3000.000000
1000/5940   temp:2499.915811
2000/5940   temp:1999.831621
3000/5940   temp:1499.747432
4000/5940   temp:999.663243
5000/5940   temp:499.579054
[4571.614371347856, 4359.584783264582, 4269.22483744525] 4400.141330685896
```

In [42]:
```python
# Test linear_cooling_2 schedule 3 times
linear_cooling_2_results = []
test1 = SA(solution, Schwefel, delta, boundary, linear_cooling_2)
linear_cooling_2_results.append(test1['evaluation'])
test2 = SA(solution, Schwefel, delta, boundary, linear_cooling_2)
linear_cooling_2_results.append(test2['evaluation'])
test3 = SA(solution, Schwefel, delta, boundary, linear_cooling_2)
linear_cooling_2_results.append(test3['evaluation'])
print(linear_cooling_2_results, np.mean(np.array(linear_cooling_2_results)))
```

```
0/5980   temp:3000.000000
1000/5980   temp:2499.916374
2000/5980   temp:1999.832748
3000/5980   temp:1499.749122
4000/5980   temp:999.665496
5000/5980   temp:499.581870
0/5980   temp:3000.000000
1000/5980   temp:2499.916374
2000/5980   temp:1999.832748
3000/5980   temp:1499.749122
4000/5980   temp:999.665496
5000/5980   temp:499.581870
0/5980   temp:3000.000000
1000/5980   temp:2499.916374
2000/5980   temp:1999.832748
3000/5980   temp:1499.749122
4000/5980   temp:999.665496
5000/5980   temp:499.581870
[4326.027962737391, 4730.867699724717, 4266.0160113315715] 4440.970557931226
```

(b) The tests with new log based cooling schedules were repeated 3 times for each logarithmic cooling condition and the results are summarized below:

Schwefel Optimization Eval	Log Cooling 1 (3000K)	Log Cooling 2 (6000K)
Test 1	2881.58	2411.84
Test 2	3223.17	2747.75
Test 3	2857.94	2824.95
Average	2987.56	2661.51

From the results, the logarithmic cooling clearly perform better than the linear cooling because they consistently produce lower results. However, there is no significant difference between the two log cooling conditions tested.

In [52]:
```python
def log_cooling_schedule(T_SA, sigma, k_counter):
    log_cooling_schedule = []
    for k in range(k_counter):
        T_K = T_SA / (1 + T_SA * math.log10(1 + k) / (3 * sigma))
        log_cooling_schedule.append(T_K)
    return np.array(log_cooling_schedule)

log_cooling_schedule_1 = log_cooling_schedule(3000, 1000, 6000)
print(log_cooling_schedule_1, len(log_cooling_schedule_1))
log_cooling_schedule_2 = log_cooling_schedule(6000, 1000, 6000)
print(log_cooling_schedule_2, len(log_cooling_schedule_2))
```

```
[3000.         2305.86536052 2030.97747759 ...  627.87692403  627.86741004
  627.85789792] 6000
[6000.         3745.17810349 3070.24331475 ...  701.26121292  701.24934505
  701.23747956] 6000
```

In [53]:
```python
solution = np.random.random(10) * 1000 - 500   # Create a 10-d array within [-500, 500)
boundary = np.array([-500, 500])
delta = 0.5
SA(solution, Schwefel, delta, boundary, log_cooling_schedule_1)
```

```
0/6000   temp:3000.000000
1000/6000   temp:749.918619
2000/6000   temp:697.472137
3000/6000   temp:670.051769
4000/6000   temp:651.866607
5000/6000   temp:638.425985
```

Out[53]:
```
{'solution': array([-297.78805729,  465.67875849,  244.20034201,  366.98641343,
         416.63173853,  385.89572533,   75.68436536, -174.02414711,
          15.00723356, -215.08750031]),
 'evaluation': 3130.474626980866}
```

```
In [57]: # Test log_cooling_schedule_1 3 times
log_cooling_1_results = []
test1 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_1)
log_cooling_1_results.append(test1['evaluation'])
test2 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_1)
log_cooling_1_results.append(test2['evaluation'])
test3 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_1)
log_cooling_1_results.append(test3['evaluation'])
print(log_cooling_1_results, np.mean(np.array(log_cooling_1_results)))
```

```
0/6000   temp:3000.000000
1000/6000   temp:749.918619
2000/6000   temp:697.472137
3000/6000   temp:670.051769
4000/6000   temp:651.866607
5000/6000   temp:638.425985
0/6000   temp:3000.000000
1000/6000   temp:749.918619
2000/6000   temp:697.472137
3000/6000   temp:670.051769
4000/6000   temp:651.866607
5000/6000   temp:638.425985
0/6000   temp:3000.000000
1000/6000   temp:749.918619
2000/6000   temp:697.472137
3000/6000   temp:670.051769
4000/6000   temp:651.866607
5000/6000   temp:638.425985
[3117.092624596807, 3189.289866835459, 3014.2122078360003] 3106.864899756089
```

```
In [58]: # Test log_cooling_schedule_2 3 times
log_cooling_2_results = []
test1 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_2)
log_cooling_2_results.append(test1['evaluation'])
test2 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_2)
log_cooling_2_results.append(test2['evaluation'])
test3 = SA(solution, Schwefel, delta, boundary, log_cooling_schedule_2)
log_cooling_2_results.append(test3['evaluation'])
print(log_cooling_2_results, np.mean(np.array(log_cooling_2_results)))
```

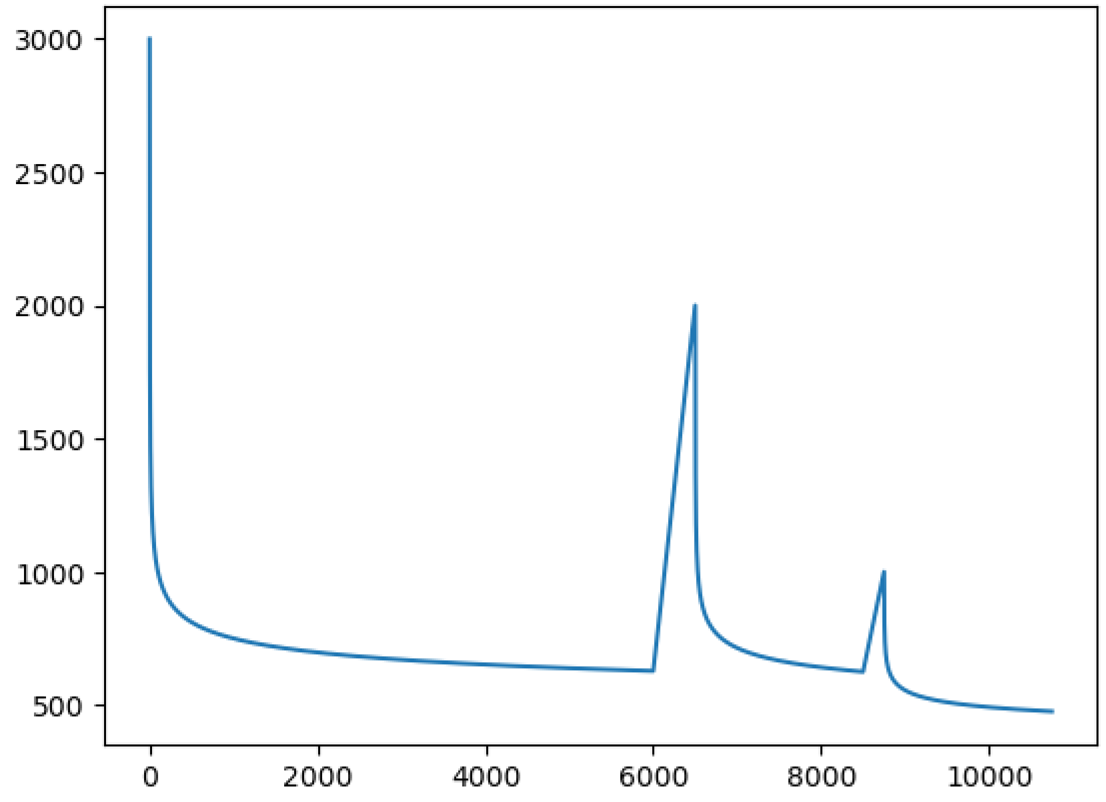
```
0/6000   temp:6000.000000
1000/6000   temp:857.036566
2000/6000   temp:789.214679
3000/6000   temp:754.286991
4000/6000   temp:731.320511
5000/6000   temp:714.446149
0/6000   temp:6000.000000
1000/6000   temp:857.036566
2000/6000   temp:789.214679
3000/6000   temp:754.286991
4000/6000   temp:731.320511
5000/6000   temp:714.446149
0/6000   temp:6000.000000
1000/6000   temp:857.036566
2000/6000   temp:789.214679
3000/6000   temp:754.286991
4000/6000   temp:731.320511
5000/6000   temp:714.446149
[3059.0515495252594, 2944.8449294791217, 3191.877685646612] 3065.2580548836645
```

(c) Self-created cooling schedule is not necessarily better than the previous cooling schedules.

Using scipy.optimize.dual_annealing algorithm, an optimum point was acquired at [420.96879051, 420.96874185, 420.96874932, 420.96879411, 420.96872868, 420.96879546, 420.9688198, 420.9689309, 420.96871069, 420.96872682]. A minimum value of 0.0001272817298740847 was acquired. It's much better than any optimization I've done with the SA algorithm.

In [67]:
```
schedule=np.append(log_cooling_schedule_1, np.linspace(log_cooling_schedule_1[-1],2000,500))  # slow linear heating to
schedule=np.append(schedule,log_cooling_schedule(2000, 1000, 2000))     # slow log cooling
schedule=np.append(schedule,np.linspace(schedule[-1],1000,250))     # slow linear heating to 1000K
schedule=np.append(schedule,log_cooling_schedule(1000, 1000, 2000))     # slow log cooling
plt.plot(schedule)
```

Out[67]:
```
[<matplotlib.lines.Line2D at 0x11ab3b1f0>]
```



In [68]:
```
solution = np.random.random(10) * 1000 - 500   # Create a 10-d array within [-500, 500)
boundary = np.array([-500, 500])
delta = 0.5

# Test log_cooling_schedule_2 3 times
schedule_results = []
test1 = SA(solution, Schwefel, delta, boundary, schedule)
schedule_results.append(test1['evaluation'])
test2 = SA(solution, Schwefel, delta, boundary, schedule)
schedule_results.append(test2['evaluation'])
test3 = SA(solution, Schwefel, delta, boundary, schedule)
schedule_results.append(test3['evaluation'])
print(schedule_results, np.mean(np.array(schedule_results)))
```

```
0/10750   temp:3000.000000
1000/10750   temp:749.918619
2000/10750   temp:697.472137
3000/10750   temp:670.051769
4000/10750   temp:651.866607
5000/10750   temp:638.425985
6000/10750   temp:627.857898
7000/10750   temp:714.313313
8000/10750   temp:641.521766
9000/10750   temp:555.589125
10000/10750   temp:492.024503
0/10750   temp:3000.000000
1000/10750   temp:749.918619
2000/10750   temp:697.472137
3000/10750   temp:670.051769
4000/10750   temp:651.866607
5000/10750   temp:638.425985
6000/10750   temp:627.857898
7000/10750   temp:714.313313
8000/10750   temp:641.521766
9000/10750   temp:555.589125
10000/10750   temp:492.024503
0/10750   temp:3000.000000
1000/10750   temp:749.918619
2000/10750   temp:697.472137
3000/10750   temp:670.051769
4000/10750   temp:651.866607
5000/10750   temp:638.425985
6000/10750   temp:627.857898
7000/10750   temp:714.313313
8000/10750   temp:641.521766
9000/10750   temp:555.589125
10000/10750   temp:492.024503
[3553.9006187553496, 3465.91439418382, 3426.443981019757] 3482.0863313196423
```

In [79]:
```
from scipy import optimize
lw = [-500] * 10
up = [500] * 10
bounds=list(zip(lw, up))
res_SA = optimize.dual_annealing(Schwefel, bounds, maxiter=1000, initial_temp=3000, x0=solution)
print(res_SA.x, res_SA.fun)
```

```
[420.96879051 420.96874185 420.96874932 420.96879411 420.96872868
 420.96879546 420.9688198  420.9689309  420.96871069 420.96872682] 0.00012728172987408847
```

2. Clustering and simulated annealing.

(a) The original dataset was normalized with the following pandas functions. A new DataFrame wines_norm was created.

In [282…
```
wines = pd.read_csv('wines.csv')
wines_norm = wines.loc[:, 'Alcohol %':'Proline']
wines_norm = (wines_norm - np.mean(wines_norm, axis=0)) / np.std(wines_norm, axis=0)
wines_norm = wines_norm.merge(wines[['Start assignment','ranking']], left_index=True, right_index=True)
wines_norm.head()
```

Out[282]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD280 315	Prolin
0	1.518613	-0.562250	0.232053	-1.169593	1.913905	0.808997	1.034819	-0.659563	1.224884	0.251717	0.362177	1.847920	1.01300
1	0.295700	0.227694	1.840403	0.451946	1.281985	0.808997	0.663351	0.226796	0.401404	-0.319276	0.362177	0.449601	-0.0378
2	2.259772	-0.625086	-0.718336	-1.650049	-0.192495	0.808997	0.954502	-0.578985	0.681738	0.061386	0.537671	0.336606	0.9493
3	1.382733	-0.768712	-0.170035	-0.809251	-0.332922	-0.152402	0.402320	-0.820719	-0.036617	-0.025128	0.932531	0.294232	1.6976
4	0.925685	-0.544297	0.158946	-1.049479	-0.754202	0.488531	0.733629	-0.578985	0.383884	0.234414	0.844785	0.407228	1.8250

(b) Centroid will be a [3, 13] array. By grouping using 'start assignment' column, we can acquire the centroid by taking mean value of the chemical descriptors. The centroid is shown below.

In [283…
```
chem_descriptors = list(wines_norm.columns)[0:13]
centroid = wines_norm.groupby('Start assignment')[chem_descriptors].agg(np.mean)
centroid
```

Out[283]:

	Alcohol %	Malic Acid	Ash	Alkalinity	Mg	Phenols	Flavanoids	Phenols.1	Proantho-cyanins	Color intensity	Hue	OD28 31
Start assignment												
1	-0.026321	-0.022878	0.039202	-0.011425	0.001197	0.046232	-0.014499	-0.092738	0.015342	-0.122680	0.072159	-0.02113
2	-0.030284	-0.043279	-0.117993	-0.122667	-0.180594	-0.110306	-0.040446	0.035593	-0.147087	-0.218465	0.084808	0.07781
3	0.054317	0.063613	0.076850	0.129509	0.173535	0.062731	0.052907	0.053751	0.127677	0.327948	-0.150638	-0.05517

(c) I first define the cost function based on the provided equation. The cost function is shown below. By using vectorized calculation, I was able to acquire a sum for the current dataset as 2320.10

In [284…
```
def cluster(feats, column):
    sum=0;
    for i in range(len(centroid.index)):
        sum += np.sum(np.sum((feats[feats[column] == centroid.index[i]] - centroid.iloc[i, :])**2, axis=1))
    return sum
sum = cluster(wines_norm, 'Start assignment')
sum
```

```
Out[284]:  2288.2043588628485
```

(d) I was able to complete the algorithm and saw a little change. But the algorithm takes too long to run. So I changed step to 2 and saw changes happening to the ranking and evaluation in each step.

```
In [280…
def simulated_annealing(feats,ranks,centers,start_temp,alpha,steps=10000):
    """ Simulated Annealing for clustering
    feats: pd.DataFrame. Normalized chemical descriptors.   wines_norm.
    ranks: np.array shape(178,). Initial assignment. wines_norm['ranking'].
    centers: np.array shape (3,13). Fixed centers.    centroid.
    start_temp: float. Initial tempreture.     500.
    alpha: float. Hyperparameter for geometric cooling         0.999.
    steps: int.        5000.
    """
    best_rank=ranks.copy()
    # evaluate the cost function with current best rank
    lowest_eval=cluster(feats, 'ranking')

    # Make geometric cooling schedule based on hyper-parameters
    geometric_cooling_schedule = []
    for i in range(steps):
        geometric_cooling_schedule.append(start_temp * alpha**i)
    geometric_cooling_schedule = np.array(geometric_cooling_schedule)

    for step in (range(steps)):

        # update temperature according to geometric cooling schedule
        temp=geometric_cooling_schedule[step]

        if step%10==0:
            print(step,temp,lowest_eval)

        for n in range(len(ranks)):
            trial=ranks.copy()
            rand_choice=np.random.randint(3)+1            # Randomly switch ranking assignment
            trial[n]=rand_choice
            feats['trial'] = trial
            delta = cluster(feats, 'trial') - cluster(feats, 'ranking')
            # Metropolis acceptance criterion
            if np.exp(-delta/temp) > np.random.random():
                feats['ranking'] = trial
                new_eval=cluster(feats, 'ranking')
                if new_eval<lowest_eval:
                    #update best rank and lowest_eval
                    best_rank=ranks.copy()
                    lowest_eval=new_eval
    return {"solution":best_rank,"evaluation":lowest_eval}
```

```
In [281…  simulated_annealing(wines_norm,wines_norm['ranking'],centroid,500,0.999,steps=2)

          0 500.0 2318.730662089869
Out[281]:  {'solution': 0       1
           1       1
           2       1
           3       1
           4       1
                  ..
           173     1
           174     1
           175     1
           176     1
           177     2
           Name: ranking, Length: 178, dtype: int64,
           'evaluation': 2313.6091114732476}
```

```
In [285…  def validate(solution, df):
              """Prints out how many wines are corretly assigned to its cultivar
              solution: np.array shape(178,). Your solution.
              df: pd.DataFrame. Read-in of the wines.csv dataset
              """
              # correct classification
              ranking = df['ranking'].values
              cluster_1 = list(df[df['ranking']==1].index)
              cluster_2 = list(df[df['ranking']==2].index)
              cluster_3 = list(df[df['ranking']==3].index)
              clusters =[cluster_1,cluster_2,cluster_3]

              for i in range(1,4):
                  #loop over solutions
                  counts=[]
                  scores=[]
                  for j in range(3):
                      #loop over clusters of true assignments
                      sol_i= [idx for idx,k in enumerate(solution) if k==i]
                      counts.append(len(np.intersect1d(sol_i, clusters[j])))
                      scores.append(counts[-1]/len(clusters[j]))
                  idx = np.argmax(scores)
                  print(f'Class {i} - cultivar {idx+1}: {counts[idx]} out \
          of {len(clusters[idx])} are classified correctly')
```

```
In [286…  validate(wines_norm['ranking'], wines_norm)

          Class 1 - cultivar 1: 59 out of 59 are classified correctly
          Class 2 - cultivar 2: 71 out of 71 are classified correctly
          Class 3 - cultivar 3: 48 out of 48 are classified correctly
```

(e) The concept of the algorithm is: (1) use visitation function to shift the centroid. (2) loop through all 178 wines to re-assign their groups. (3) assess the values and compare to current best.

I was able to classify all the wines correctly with the designed simulated annealing algorithm.

In [292…
```
def simulated_annealing_e(feats,ranks,centers,start_temp,alpha,delta,steps=10000):
    """ Simulated Annealing for clustering
    feats: pd.DataFrame. Normalized chemical descriptors.   wines_norm.
    ranks: np.array shape(178,). Initial assignment. wines_norm['ranking'].
    centers: np.array shape (3,13). Fixed centers.    centroid.
    start_temp: float. Initial tempreture.     500.
    alpha: float. Hyperparameter for geometric cooling         0.999.
    delta: float. Hyperparameter.   0.01.
    steps: int.        5000.
    """
    best_rank=ranks.copy()
    # evaluate the cost function with current best rank
    lowest_eval=cluster(feats, 'ranking')

    # Make geometric cooling schedule based on hyper-parameters
    geometric_cooling_schedule = []
    for i in range(steps):
        geometric_cooling_schedule.append(start_temp * alpha**i)
    geometric_cooling_schedule = np.array(geometric_cooling_schedule)
    centroid_new = centroid.copy()
    [i, j] = centroid_new.shape

    for step in (range(steps)):

        # update temperature according to geometric cooling schedule
        temp=geometric_cooling_schedule[step]

        if step%10==0:
            print(step,temp,lowest_eval)

        # Update centroid with the new visitation function
        for a in range(i):
            for b in range(j):
                centroid_new.iloc[a, b] += (np.random.random() * 2 - 1) * delta

        # re-assign ranking based on new centroid and distances
        for n in range(len(ranks)):

            # Calculate distances and find the min as new ranking assignment
            distances = []
            for k in range(len(centroid_new.index)):
                distance = np.sqrt(np.sum((feats.iloc[n, :] - centroid.iloc[k, :])**2))
                distances.append(distance)
            ranks[n] = distances.index(min(distances)) + 1

            trial = ranks.copy()
            feats['trial'] = trial
            delta = cluster(feats, 'trial') - cluster(feats, 'ranking')
            # Metropolis acceptance criterion
            if np.exp(-delta/temp) > np.random.random():
                feats['ranking'] = trial
                new_eval=cluster(feats, 'ranking')
                if new_eval<lowest_eval:
                    #update best rank and lowest_eval
                    best_rank=ranks.copy()
                    lowest_eval=new_eval
    return {"solution":best_rank,"evaluation":lowest_eval}
```

In [295…
```
simulated_annealing_e(wines_norm,wines_norm['ranking'],centroid,500,0.999,0.01,steps=100)
```

```
0 500.0 2160.171014209908
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_60915/317164022.py:44: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a
-view-versus-a-copy
  ranks[n] = distances.index(min(distances)) + 1
10 495.02244010487414 2160.171014209908
20 490.09443241476737 2160.171014209908
30 485.2154836315429 2160.171014209908
40 480.3851053679059 2160.171014209908
50 475.6028140985157 2160.171014209908
60 470.8681311115841 2160.171014209908
70 466.18058246095626 2160.171014209908
80 461.5396989186681 2160.171014209908
90 456.945015927976 2160.171014209908
```

Out[295]:
```
{'solution': 0      3
 1      3
 2      1
 3      1
 4      1
       ..
 173    3
 174    3
 175    3
 176    3
 177    2
 Name: ranking, Length: 178, dtype: int64,
 'evaluation': 2160.171014209908}
```

In [296…
```
validate(wines_norm['ranking'], wines_norm)
```

```
Class 1 - cultivar 1: 42 out of 42 are classified correctly
Class 2 - cultivar 2: 68 out of 68 are classified correctly
Class 3 - cultivar 3: 68 out of 68 are classified correctly
```