I would approach this by setting possible points that could be centers, i.e. your coastline.

This way, for each iteration, instead of choosing a mean, a point out of the possible set would be chosen by proximity to the cluster.

I’ve simplified the conditions to only 2 data columns (lon. and lat.) but you should be able to extrapolate the concept. For simplicity, to demonstrate, I based this on code from [here][1].

## In this example, the purple dots are places on the coastline. If I understood correctly, the optimal Coastline locations should look something like this:

![Coastline Optimum][2]

## See code below:

#! /usr/bin/python3.6

# Code based on:

# https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/

import matplotlib.pyplot as plt

import numpy as np

import random

##### Simulation START #####

# Generate possible points.

def possible\_points(n=20):

y=list(np.linspace( -1, 1, n ))

x=[-1.2]

X=[]

for i in list(range(1,n)):

x.append(x[i-1]+random.uniform(-2/n,2/n) )

for a,b in zip(x,y):

X.append(np.array([a,b]))

X = np.array(X)

return X

# Generate sample

def init\_board\_gauss(N, k):

n = float(N)/k

X = []

for i in range(k):

c = (random.uniform(-1, 1), random.uniform(-1, 1))

s = random.uniform(0.05,0.5)

x = []

while len(x) < n:

a, b = np.array([np.random.normal(c[0], s), np.random.normal(c[1], s)])

# Continue drawing points from the distribution in the range [-1,1]

if abs(a) < 1 and abs(b) < 1:

x.append([a,b])

X.extend(x)

X = np.array(X)[:N]

return X

##### Simulation END #####

# Identify points for each center.

def cluster\_points(X, mu):

clusters = {}

for x in X:

bestmukey = min([(i[0], np.linalg.norm(x-mu[i[0]])) \

for i in enumerate(mu)], key=lambda t:t[1])[0]

try:

clusters[bestmukey].append(x)

except KeyError:

clusters[bestmukey] = [x]

return clusters

# Get closest possible point for each cluster.

def closest\_point(cluster,possiblePoints):

closestPoints=[]

# Check average distance for each point.

for possible in possiblePoints:

distances=[]

for point in cluster:

distances.append(np.linalg.norm(possible-point))

closestPoints.append(np.mean(distances))

return possiblePoints[closestPoints.index(min(closestPoints))]

# Calculate new centers.

# Here the 'coast constraint' goes.

def reevaluate\_centers(clusters,possiblePoints):

newmu = []

keys = sorted(clusters.keys())

for k in keys:

newmu.append(closest\_point(clusters[k],possiblePoints))

return newmu

# Check whether centers converged.

def has\_converged(mu, oldmu):

return (set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu]))

# Meta function that runs the steps of the process in sequence.

def find\_centers(X, K, possiblePoints):

# Initialize to K random centers

oldmu = random.sample(list(possiblePoints), K)

mu = random.sample(list(possiblePoints), K)

while not has\_converged(mu, oldmu):

oldmu = mu

# Assign all points in X to clusters

clusters = cluster\_points(X, mu)

# Re-evaluate centers

mu = reevaluate\_centers(clusters,possiblePoints)

return(mu, clusters)

K=3

X = init\_board\_gauss(30,K)

possiblePoints=possible\_points()

results=find\_centers(X,K,possiblePoints)

# Show results

# Show constraints and clusters

# List point types

pointtypes1=["gx","gD","g\*"]

plt.plot(

np.matrix(possiblePoints).transpose()[0],np.matrix(possiblePoints).transpose()[1],'m.'

)

for i in list(range(0,len(results[0]))) :

plt.plot(

np.matrix(results[0][i]).transpose()[0], np.matrix(results[0][i]).transpose()[1],pointtypes1[i]

)

pointtypes=["bx","yD","c\*"]

# Show all cluster points

for i in list(range(0,len(results[1]))) :

plt.plot(

np.matrix(results[1][i]).transpose()[0],np.matrix(results[1][i]).transpose()[1],pointtypes[i]

)

plt.show()

[1]: https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/

[2]: https://raw.githubusercontent.com/Alex-Chervony/kmeans/master/Figure\_1.png "tooltip"