

En Yakın k-Komsu (k-Nearest Neighbor)

Yapay Öğrenim alanında örnek bazlı öğrenen algoritmalarından bilinen kNN, eğitim verinin kendisini sınıflama (classification) amaçlı olarak kullanır, yeni bir model ortaya çıkartmaz. Algoritma şöyle işler: etiketleri bilinen eğitim verisi alınır ve bir kenarda tutulur. Yeni bir veri noktası görüldüğünde bu veriye geri dönülür ve o noktaya “en yakın” k tane nokta bulunur. Daha sonra bu noktaların etiketlerine bakılır ve çoğunluğun etiketi ne ise, o etiket yeni noktanın etiketi olarak kabul edilir.

“En yakın” sözü bir koordinat sistemi anlamına geliyor, ve kNN, aynen k-Means ve diğer pek çok koordinatsal öğrenme yöntemi gibi eldeki çok boyutlu veri noktalarının elemanlarını bir koordinat sistemindeymiş gibi görür. Kiyasla mesela APriori gibi bir algoritma metin bazlı veriyle olduğu gibi çalışabilirdi.

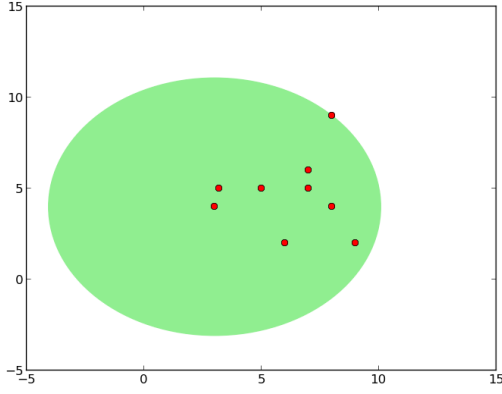
Peki arama bağlamında, bir veri obesi içinden en yakın noktaları bulmanın en basit yolu nedir? Listeyi bastan sonra taramak (kaba kuvvet yöntemi -brute force-) ve her listedeki nokta ile yeni nokta arasındaki mesafeyi teker teker hesaplayıp en yakın k taneyi içinden seçmek bir yöntem. Bu basit algoritmanın yuku $O(N)$ 'dir. Eğer tek bir nokta alıyorsa olsaydı, bu kabul edilebilir olabilirdi. Fakat genellikle bir sınıflayıcı algoritmanın sürekli işlemesi, mesela bir online site için günde milyonlarca kez bazı kararları alması gerekebilir. Bu durumda N 'in çok büyük olduğu şartlarda, üstteki hız bile yeterli olmayabilir.

Arama işlemini daha hızlı yapmanın yolları var. Akıllı arama algoritmaları kullanarak eğitim verilerini bir ağacı yapısı üzerinden tarayıp erişim hızını $O(\log N)$ 'e indirmek mümkündür.

Küre Ağaçları (Ball Tree -BT-)

Bir noktanın diğer noktalara yakın olup olmadığının hesabında yapılması gereken en pahalı işlem nedir? Mesafe hesabıdır. BT algoritmasının puf noktası bu hesabi yapmadan, noktalara değil, noktaları kapsayan “kurelere” bakarak hız kazandırmasıdır. Noktaların her biri yerine o noktaları temsil eden kurenin mihişik noktasına (pivot -bu nokta merkez de olabilir, herhangi bir başka nokta da-) bakılır, ve oraya olan mesafeye göre bir küre altındaki noktalara olabilecek en az ve en fazla uzaklık hemen anlaşılmış olur.

Mesela elimizde alttaki gibi noktalar var ve küreyi oluşturduk.

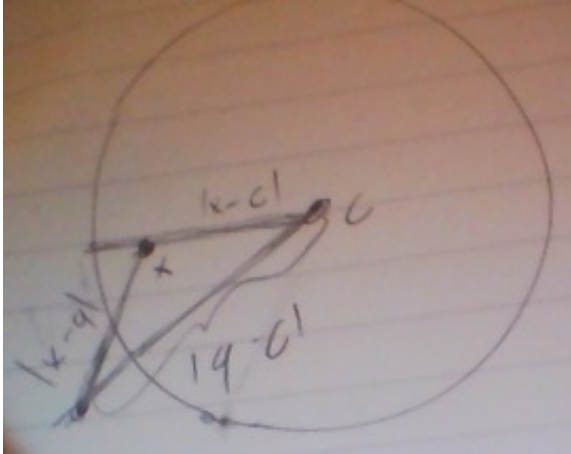


Bu kureyi kullanarak kure disindaki herhangi bir nokta q 'nun kuredeki “diger tum noktalar x 'e” olabilecegi en az mesafenin ne olacagini ucgensel esitsizlik ile anlayabiliriz.

Ucgensel esitsizlik

$$|x, y| \leq |x, z| + |z, y|$$

$||$ operatoru norm operatoru anlamina gelir ve uzaklik hesabının genelleştirilmiş halidir. Konu hakkında daha fazla detay için *Fonksinel Analiz* ders notlarımıza bakabilirsiniz. Kısaca söylenmek istenen iki nokta arasında direk gitmek yerine yolu uzatırsak, mesafe artacaktır. Tabii uzaklık, yol, nokta gibi kavramlar tamamen soyut matematiksel ortamda da işleyecek şekilde ayarlanmıştır. Mesela mesafe (norm) kavramını değiştirebiliriz, Oklitsel yerine Manhattan mesafesi kullanırız, fakat bu kavram bir norm olduğu ve belirttiğimiz uzayda geçerli olduğu için ugensel esitsizlik üzerine kurulmuş tüm diğer kurallar geçerli olur.



Simdi diyelim ki disaridaki bir q noktasından bir kure icindeki diger tum x noktalarına olan mesafe hakkında bir seyler soylemek istiyoruz. Ustteki sekilden bir ugensel esitsizlik cikartabiliriz,

$$|x - c| + |x - q| \geq |q - c|$$

Bunun dogru bir ifade oldugunu biliyoruz. Peki simdi yaricapı bu ise dahil edelim, cunku bu hesap bir kere yapilip kure seviyesinde depolanacak ve bir daha hesaplanmasi gerekmeyecek, algoritmayı hizlandiracak bir sey olabilir bu. Eger $|x - c|$ yerine yaricapı kullanırsak, esitsizlik hala gecerli olur,

$$radius + |x - q| \geq |q - c|$$

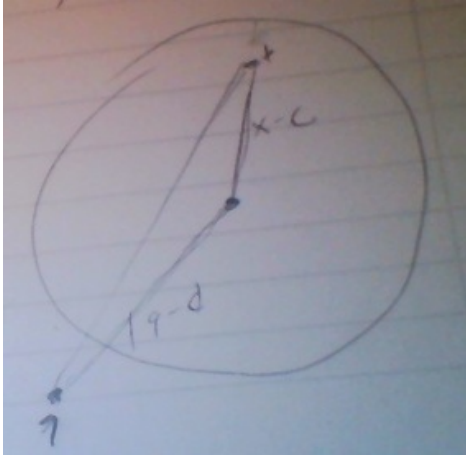
cunku radius $|x - c|$ 'ten kesinlikle daha buyuktur (Kure Agaclari yaricapı aynen Boyle olmasi icin ayarlayacak). Yaricapı saga gecirelim,

$$|x - q| \geq |q - c| - radius$$

Boylece guzel bir tanim elde ettik. Yeni noktanin kuredeki herhangi bir nokta x 'e olan uzakligi, yeni noktanin mihenke olan uzakliginin yaricapı cikartilmis halinden *muhakkak* fazladir. Yani bu cikartma isleminde ele gecen rakam yeni noktanin x 'e uzakligina bir “alt sinir (lower bound)” olarak kabul edilebilir. Diger tum mesafeler bu rakamdan daha buyuk olacaktır.

Boylece ne elde ettik? Sadece bir yeni nokta, mihenk ve yaricap kullanarak kuredeki “diger tum noktalar hakkında” bir irdeleme yapmamiz mumkun olacak. Bu noktalara teker teker bakmamiz gerekmeyecek. Bunun nasil ise yaradigini algoritma detaylarında gorecegiz.

Benzer sekilde



Bu ne diyor?

$$|q - c| + |x - c| \geq |q - x|$$

$|x - c|$ yerine yaricapı kullanırsak, sol taraf buyuyecegi icin buyukluk hala buyukluk olarak kalir,

$$|q - c| + radius \geq |q - x|$$

ama yine daha genel ve hizli hesaplanan bir kural elde ederiz (onceki ifadeye benzeri icin yer degisikligi yapalim

$$|q - x| \leq |q - c| + radius$$

Bu ifade ne diyor? Yeni noktanin mihenke olan uzakligina yaricap “eklenirse” bu uzaklikten, buyuklukten daha buyuk bir yeni nokta - kure mesafesi olamaz, kuredeki hangi nokta olursa olsun. Bu esitsizlik te bize bir ust sinir (upper bound) vermis oldu.

```

from matplotlib.patches import Circle
import matplotlib.pyplot as plt
import pprint
import numpy as np
import time
import dist

__rmin__ = 2

# node: [pivot, radius, points, [child1, child2]]
def new_node(): return [None, None, None, [None, None]]

def zero_if_neg(x):
    if x < 0: return 0
    else: return x

def form_tree(points, node):
    pivot = points[0]
    radius = np.max(dist.dist(points, pivot))
    node[0] = pivot
    node[1] = radius
    if len(points) <= __rmin__:
        node[2] = points
        return
    idx = np.argmax(dist.dist(points, pivot))
    furthest = points[idx, :]
    idx = np.argmax(dist.dist(points, furthest))
    furthest2 = points[idx, :]
    dist1 = dist.dist(points, furthest)
    dist2 = dist.dist(points, furthest2)
    diffs = dist1 - dist2
    p1 = points[diffs <= 0]
    p2 = points[diffs > 0]
    node[3][0] = new_node() # left child
    node[3][1] = new_node() # right child
    form_tree(p1, node[3][0])
    form_tree(p2, node[3][1])

# knn: [min-so-far, [points]]
def search_tree(new_point, knn_matches, node, k):
    pivot = node[0]
    radius = node[1]
    node_points = node[2]
    children = node[3]

    # calculate min distance between new point and pivot
    # it is direct distance minus the radius
    min_dist_new_pt_node = dist.norm(pivot, new_point) - radius

    # if the new pt is inside the circle, its potential minimum

```

```

# distance to a random point inside is zero (hence
# zero_if_neg). we can only say so much without looking at all
# points (and if we did, that would defeat the purpose of this
# algorithm)
min_dist_new_pt_node = zero_if_neg(min_dist_new_pt_node)

knn_matches_out = None

# min is greater than so far
if min_dist_new_pt_node >= knn_matches[0]:
    # nothing to do
    return knn_matches
elif node_points != None: # if node is a leaf
    print knn_matches_out
    knn_matches_out = knn_matches[:] # copy it
    for p in node_points: # linear scan
        if dist.norm(new_point, p) < radius:
            knn_matches_out[1].append([list(p)])
            if len(knn_matches_out[1]) == k+1:
                tmp = [dist.norm(new_point, x) \
                        for x in knn_matches_out[1]]
                del knn_matches_out[1][np.argmax(tmp)]
                knn_matches_out[0] = np.min(tmp)

    else:
        dist_child_1 = dist.norm(children[0][0], new_point)
        dist_child_2 = dist.norm(children[1][0], new_point)
        node1 = None; node2 = None
        if dist_child_1 < dist_child_2:
            node1 = children[0]
            node2 = children[1]
        else:
            node1 = children[1]
            node2 = children[0]

    knn_tmp = search_tree(new_point, knn_matches, node1, k)
    knn_matches_out = search_tree(new_point, knn_tmp, node2, k)

return knn_matches_out

if __name__ == "__main__":
    points = np.array([[3., 4.], [5., 5.], [9., 2.], [3.2, 5.], [7., 5.],
                       [8., 9.], [7., 6.], [8, 4], [6, 2]])
    tree = new_node()
    form_tree(points, tree)
    pp = pprint.PrettyPrinter(indent=4)
    print "tree"
    pp.pprint(tree)
    newp = np.array([7., 7.])
    dummyp = [100, 100] # it should be removed immediately
    dummydist = dist.norm(dummyp, newp)
    res = search_tree(newp, [dummydist, [dummyp]], tree, k=2)
    print "done", res

```