En Yakin k-Komsu (k-Nearest Neighbor)

Yapay Ogrenim alaninda ornek bazli ogrenen algoritmalardan bilinen kNN, egitim verinin kendisini siniflama (classification) amacli olarak kullanir, yeni bir model ortaya cikartmaz. Algoritma soyle isler: etiketleri bilinen egitim verisi alinir ve bir kenarda tutulur. Yeni bir veri noktasi gorulunce bu veriye geri donulur ve o noktaya "en yakin" k tane nokta bulunur. Daha sonra bu noktalarin etiketlerine bakilir ve cogunlugun etiketi ne ise, o etiket yeni noktanin etiketi olarak kabul edilir.

"En yakin" sozu bir kordinat sistemi anlamina geliyor, ve kNN, aynen k-Means ve diger pek cok kordinatsal ogrenme yontemi gibi eldeki cok boyutlu veri noktalarinin elemanlarini bir kordinat sistemindeymis gibi gorur. Kiyasla mesela APriori gibi bir algoritma metin bazli veriyle oldugu gibi calisabilirdi.

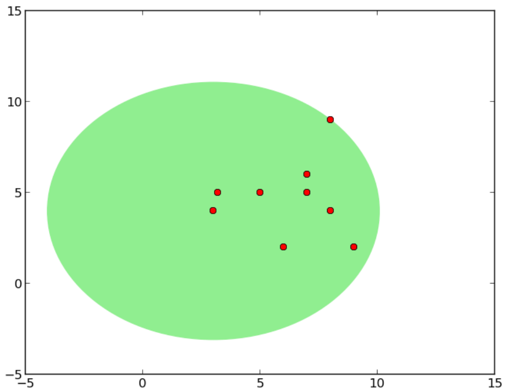
Peki arama baglaminda, bir veri obegi icinden en yakin noktalari bulmanin en basit yolu nedir? Listeyi bastan sonra taramak (kaba kuvvet yontemi -brute force-) ve her listedeki nokta ile yeni nokta arasindaki mesafeyi teker teker hesaplayip en yakin k taneyi icinden secmek bir yontem. Bu basit algoritmanin yuku $O(N)$'dir. Eger tek bir nokta ariyor olsaydik, bu kabul edilebilir olabilirdi. Fakat genellikle bir siniflayici algoritmanin surekli islemesi, mesela bir online site icin gunde milyonlarca kez bazi kararlari almasi gerekebilir. Bu durumda ve N'in cok buyuk oldugu sartlarda, ustteki hiz bile yeterli olmayabilir.

Arama islemini daha hizli yapmanin yollari var. Akilli arama algoritmalari kullanarak egitim verilerini bir agac yapisi uzerinden tarayip erisim hizini $O(\log N)$'e indirmek mumkundur.

Küre Agaçları (Ball Tree -BT-)

Bir noktanin diger noktalara yakin olup olmadiginin hesabinda yapilmasi gereken en pahali islem nedir? Mesafe hesabidir. BT algoritmasinin puf noktasi bu hesabi yapmadan, noktalara degil, noktalari kapsayan "kurelere" bakarak hiz kazandirmasidir. Noktalarin her biri yerine o noktalari temsil eden kurenin mihenk noktasina (pivot -bu nokta merkez de olabilir, herhangi bir baska nokta da-) bakilir, ve oraya olan mesafeye gore bir kure altindaki noktalara olabilecek en az ve en fazla uzaklik hemen anlasilmis olur.

Mesela elimizde alttaki gibi noktalar var ve kureyi olusturduk.

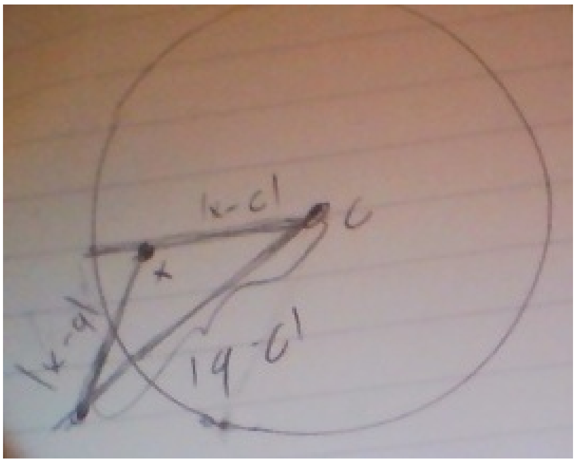


Bu kureyi kullanarak kure disindaki herhangi bir nokta q'nun kuredeki "diger tum noktalar x'e" olabilecegi en az mesafenin ne olacagini ucgensel esitsizlik ile anlayabiliriz.

Ucgensel esitsizlik

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

$||$ operatoru norm operatoru anlamina gelir ve uzaklik hesabinin genellestirilmis halidir. Konu hakkinda daha fazla detay icin *Fonksinel Analiz* ders notlarimiza bakabilirsiniz. Kisaca soylenmek istenen iki nokta arasinda direk gitmek yerine yolu uzatirsak, mesafe artacagidir. Tabii uzaklik, yol, nokta gibi kavramlar tamamen soyut matematiksel ortamda da isleyecek sekilde ayarlanmistir. Mesela mesafe (norm) kavramini degistirebiliriz, Oklitsel yerine Manhattan mesafesi kullaniriz, fakat bu kavram bir norm oldugu ve belirttigimiz uzayda gecerli oldugu icin ucgensel esitsizlik uzerine kurulmus tum diger kurallar gecerli olur.



Simdi diyelim ki disaridaki bir q noktasindan bir kure icindeki diger tum x noktalarina olan mesafe hakkinda bir seyler soylemek istiyoruz. Ustteki sekilden bir ucgensel esitsizlik cikartabiliriz,

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

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

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

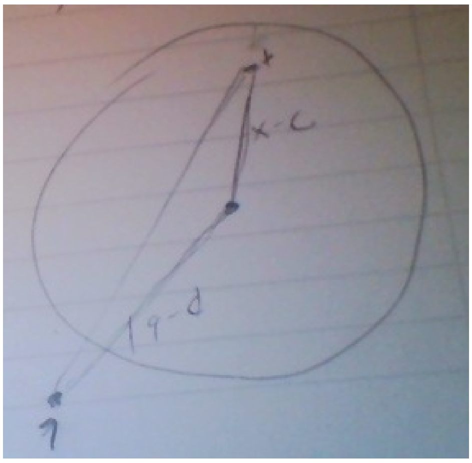
cunku radius $|x-c|$'ten kesinlikle daha buyuktur (Kure Agaclari yaricapi aynen boyle olmasi icin ayarlayacak). Yaricapi saga gecirelim,

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

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

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

Benzer sekilde



Bu ne diyor?

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

$|x-c|$ yerine yaricap kullanirsak, 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 benzemesi icin yer degisikligi yapalim

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

Bu ifade ne diyor? Yeni noktanin mihenke olan uzakligina yaricap "eklenirse" bu uzakliktan, 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.

Algoritma

```
Procedure BallKNN (PS^in, Node)
begin
  if (D^Node_minp ≥ D_sofar) then          /* If this condition is satisfied, then impossible
    Return PS^in unchanged.                  for a point in Node to be closer than the
                                              previously discovered k^th nearest neighbor.*/
  else if (Node is a leaf)
    PS^out = PS^in
    ∀x ∈ Points(Node)
    if (| x − q |< D_sofar) then            /* If a leaf, do a naive linear scan */
      add x to PS^out
      if (| PS^out |== k + 1) then
        remove furthest neighbor from PS^out
        update D_sofar
  else                                       /*If a non-leaf, explore the nearer of the two
    node_1 = child of Node closest to q        child nodes, then the further. It is likely that
    node_2 = child of Node furthest from q     further search will immediately prune itself.*/
    PS^temp = BallKNN(PS^in, node_1)
    PS^out = BallKNN(PS^temp, node_2)
end
```

Kure Agaclari (BT) metotu once kureleri olusturmalidir. Bu kureler hiyerarsik sekilde planlanir, tum noktalarin icinde oldugu bir "en ust kure" vardir her kurenin iki tane cocuk kuresi olabilir. Belli bir (disaridan tanimlanan) minimum r_{min} veri noktasina gelinceye kadar sadece noktalari kapsayan kureler olusturulur, kureler noktalari sahiplenmezler. Fakat bu r_{min} sayisina erisince (artik oldukca alttaki) kurelerin uzerinde noktalar konur.

Once tek kurenin tanimina bakalim: bir kure tanimi icin eldeki veri icinden herhangi bir tanesi mihenk olarak kabul edilebilir. Daha sonra bu mihenkten diger tum noktalara olan uzaklik olculur, ve en fazla, en buyuk olan uzaklik yaricap olarak kabul edilir (her seyi kapsayabilmesi icin). "Tum diger noktalara bakilmasi" dedik, bundan kacinmaya calismiyor muyduk? Fakat dikkat, "kure olusturulmasi" evresindeyiz, k tane yakin nokta arama evresinde degiliz. Yapmaya calistigimiz aramalari hizlandirmak - egitim / kure olusturmasi bir kez yapilacak ve bu egitilmis kureler bir kenarda tutulacak ve surekli aramalar icin ardi ardina kullanilacaklar.

```
import numpy as np
import itertools

def dist(vect,x):
    return np.fromiter(itertools.imap
                       (np.linalg.norm, vect-x),dtype=np.float)

def norm(x,y): return np.linalg.norm(x-y)

if __name__ == "__main__":
    # small test
    points = np.array([[3.,3.],[2.,2.]])
    q = [1.,1.]
```

```
    print "diff", points-q
    print "dist", dist(points,q)
```

```
import pprint
import numpy as np
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
```

Kaynaklar

[1] Liu, Moore, Gray, *New Algorithms for Efficient High Dimensional Non-parametric Classification*