

En Yakın k-Komsu (k-Nearest Neighbor)

Yapay Ogrenim 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 donulur 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. Mesela elde 1 kategorisi altında $[2 \ 2]$, 2 kategorisi altında $[5 \ 5]$ var ise, yeni nokta $[3, \ 3]$ için yakınlık açısından $[2 \ 2]$ bulunmalı ve etiket olarak 1 sonucu dondurulmalıdır.

Ustte tarif edilen basit bir ihtiyaç, yöntem gibi görülebilir. Fakat yapay öğrenim ve yapay zeka çok boyutlarda örüntü tanıma (pattern recognition) ile uğraşır, ve milyonlarca satırlık veri, onlarca boyut (üstteki örnekte 2, fakat çoğunlukla çok daha fazla boyut vardır) işler hakikaten zorlaşabilir. Mesela görüntü tanımadaki veri $M \times N$ boyutundaki dijital imajlar (düzleştirilince $M \cdot N$ boyutunda), ve onların içindeki resimlerin kime ait olduğu etiket bilgisi olabilir. kNN bu tür multimedya, çok boyutlu veri ortamında başarılı şekilde çalışabilmektedir. Ayrıca en yakın k komşunun içeriği tarıfsal bilgi çıkarımı (knowledge extraction) amacıyla da kullanılabilir [2].

“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-) listedeki her nokta ile yeni nokta arasındaki mesafeyi teker teker hesaplayıp en yakın k taneyi içinden seçerdi, bu bir yöntemdir.. Bu basit algoritmanın yuku $O(N)$ 'dir. Eğer tek bir nokta arıyor olsaydık, kabul edilebilir olabilirdi. Fakat genellikle bir sınıflayıcı (classifier) algoritmasının sürekli işlemesi, mesela bir online site için günde milyonlarca kez bazı kararları alması gerekebilir. Bu durumda ve N 'in çok büyük olduğu şartlarda, üstteki hız bile yeterli olmayacaktır.

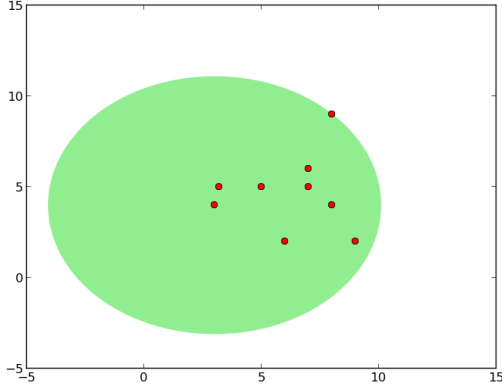
Arama işlemini daha hızlı yapmanın yolları var. Akıllı arama algoritmaları kullanarak eğitim verilerini bir ağacı yapısı üzerinden tarayıp erişim hızını $O(\log N)$ 'e indirmek mümkündür.

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 “kürelere” bakarak hız kazandırmasıdır. Noktaların her biri yerine o noktaları temsil eden kürenin mihenk noktasına (pivot -bu nokta küre içindeki noktaların ortalamasal olarak merkezi 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.

Not: Kure kavrami uc boyutta anlamlı tabii ki, iki boyutta bir cemberden bahsetmek lazım, daha yüksek boyutlarda ise merkezi ve çapı olan bir “hiper yüzeyden” bahsetmek lazım. Tarifi kolaylastırdığı için cember ve kure tanımlarını kullanıyoruz.

Mesela elimizde alttaki gibi noktalar var ve kureyi oluşturduk.

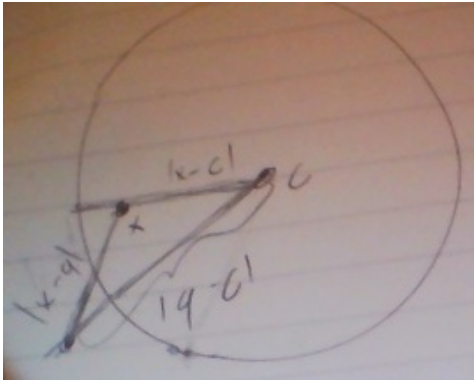


Bu kureyi kullanarak kure dışındaki herhangi bir nokta q 'nın kuredeki “diğer tüm noktalar x 'e” olabileceği en az mesafenin ne olacağını ucgen sel eşitsizlik ile anlayabiliriz.

Ucgen sel eşitsizlik

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

$\|$ operatörü norm operatörü anlamına gelir ve uzaklık hesabının genelleştirilmiş halidir. Konu hakkında daha fazla detay için *Fonksinel Analiz* ders notlarımıza bakabilirsiniz. Kısa söylemek istenen iki nokta arasında direkt 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 ucgen sel eşitsizlik üzerine kurulmuş tüm diğer kurallar geçerli olur.



Şimdi diyelim ki dışarıdaki bir q noktasından bir kure içindeki diğer tüm x noktalarına olan mesafe hakkında bir şeyler söylemek istiyoruz. Üstteki şekilde bir ucgen sel

esitsizlik cikartabiliriz,

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

Bunun dogru bir ifade oldugunu biliyoruz. Peki simdi yaricapi bu ise dahil edelim, cunku yaricap hesabi bir kere yapilip kure seviyesinde depolanacak ve bir daha hesaplanmasi gerekmeyecek, yani algoritmayi hizlandiracak bir sey olabilir bu, o zaman eger $|x - c|$ yerine yaricapi kullanirsak, esitsizlik hala gecerli olur, sol taraf zaten buyuktu, simdi daha da buyuk olacak,

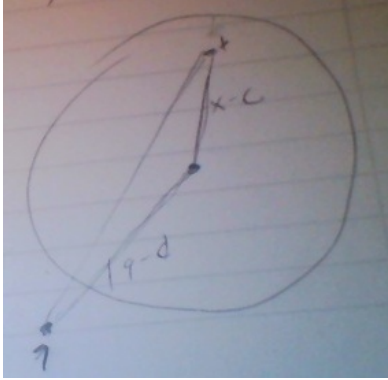
$$radius + |x - q| \geq |q - c|$$

Bunu nasil boyle kesin bilebiliyoruz? Cunku BT algoritmasi radius'u $|x - c|$ 'ten kesinlikle daha buyuk olacak sekilde secer). Simdi 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 isleminde ele gecen rakam yeni noktanin x 'e uzakligina bir “alt sinir (lower bound)” olarak kabul edilebilir. Diger tum mesafeler bu rakamdan daha buyuk olacaktir. 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 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,

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

Ve yine daha genel ve hizli hesaplanan bir kural elde ettik (onceki ifadeye benzemesi icin yer duzenlemesi 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.

Algoritma

Kure Agaclari (BT) metodu once kureleri, agaclari 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 noktalar geometrik olarak kapsamakla goreli kureler olusturulur, kureler noktalar sahiplenmezler. Fakat bu r_{min} sayisina erisince (artik oldukca alttaki) kurelerin uzerine noktalar konacaktır.

Once tek kurenin olusturulusuna bakalim. Bir kure olusumu 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).

Not: Bu arada “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.

Kureyi olusturmanin algoritmasi soyledir: verilen noktalar icinde herhangi birisi mihenk olarak secilir. Sonra bu noktadan en uzakta olan nokta f_1 , sonra f_1 'den en uzakta olan nokta f_2 secilir. Sonra tum noktalara teker teker bakilir ve f_1 'e yakin olanlar bir gruba, f_2 'ye yakin olanlar bir gruba ayrilir.

```
#
# k-nearest neighbor Ball Tree algorithm in Python
#
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 = [np.Inf, np.Inf] # it should be removed immediately
    res = search_tree(newp, [np.Inf, [dummyp]], tree, k=2)
    print "done", res

```

```

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)

```

```

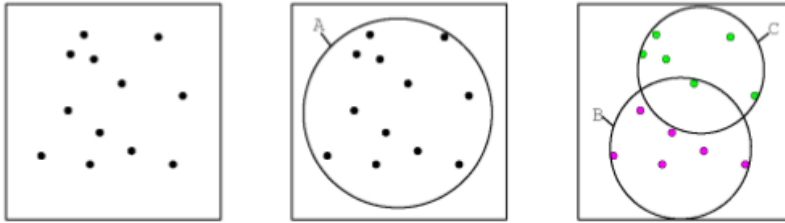
Procedure BallKNN ( $PS^{in}, Node$ )
begin
  if ( $D_{minp}^{Node} \geq D_{sofar}$ ) then                                /* If this condition is satisfied, then impossible
    Return  $PS^{in}$  unchanged.                                     for a point in Node to be closer than the
                                                                previously discovered  $k^{th}$  nearest neighbor.*/
  else if (Node is a leaf)
     $PS^{out} = PS^{in}$ 
     $\forall x \in Points(Node)$ 
    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
     $node_1 = \text{child of Node closest to } q$                    child nodes, then the further. It is likely that
     $node_2 = \text{child of Node furthest from } q$                further search will immediately prune itself.*/
     $PS^{temp} = BallKNN(PS^{in}, node_1)$ 
     $PS^{out} = BallKNN(PS^{temp}, node_2)$ 
end

```

Bu iki grup, o anda islemekte oldugumuz agac dugumun (node) iki cocuklari olacaktir. Cocuk noktaları kararlaştırıldıktan sonra artık sonraki asamaya gecilir, fonksiyon `form_tree` bu cocuk noktaları alarak, ayri ayri, her cocuk grubu icin ozyineli (recursive) olarak kendi kendini cagirir. Kendi kendini cagiran `form_tree`, tekrar basladiginda kendini yeni (bir) nokta grubu ve yeni bir dugum objesi ile bas-basa bulur, ve hicbir seyden habersiz olarak isleme koyulur. Tabii her ozyineli cagri yeni dugum objesini yaratirken bir referansi ustteki ebeveyn dugume koymayi unutmamistir, Boylece ozyineli fonksiyon dunyadan habersiz olsa bile, agacin en ustunden en altina kesintisiz bir baglanti zinciri hep elimizde olur.

Not: `form_tree` icinde bir numara yaptik, tum noktaların f_1 'e olan uzakligi `dist1`, f_2 'e olan uzakligi ise `dist2`. Sonra `diffs = dist1-dist2` ile bu iki uzakligi birbirinden cikartiyoruz ve mesela `points[diffs <= 0]` ile f_1 'e yakin olanlari buluyoruz, cunku bir tarafta f_1 'e yakinklik 4 diger tarafta f_2 'ye yakinklik 6 ise, $4-6=-2$ ie o nokta f_1 'e yakin demektir. Ufak bir numara ile Numpy dilimleme (slicing) teknigini kullanabilmis olduk ve bu onemli cunku Boylece `for` dongusu yazmiyoruz, Numpy'in arka planda C ile yazilmis hizli rutinlerini kullaniyoruz.

Ek bazi bilgiler: kurelerin sinirlari kesisebilir.



Arama

Ustte sozde program (pseudocode) `BallKNN` olarak gosterilen ve bizim kodda `search_tree` olarak anilan fonksiyon arama fonksiyonu. Aranani `new_point`'e olan k en yakin diger veri noktaları. Disaridan verilen degisken `knn_matches` uzerinde fonksiyon ozyineli bir sekilde arama yaparken “o ana kadar bulunmus en yakin k

nokta” ve o noktaların `new_point`’e olan en yakın mesafesi saklanır, arama isleyisi sırasında `knn_matches`, `knn_matches_out` sürekli verilip geri dondurulen degiskenlerdir, sozde programdaki P^{in} , P^{out} ’un karsiligidirlar.

Arama algoritmasi soyle isler: simdi onceden olusturulmus kure hiyerarisisini ustten alta dogru gezmeye baslariz. Her basamakta yeni nokta ile o kurenin mihenkini, yaricapini kullanarak bir “alt sinir mesafe hesabi” yapariz, bu mesafe hesabinin arkasinda yatan dusunceyi yazinin basinda anlatmistik. Bu mesafe kure icindeki tum noktalara olan bir en az mesafe idi, ve eger eldeki `knn_matches` üzerindeki simdiye kadar bulunmus mesafelerin en azindan daha az ise, o zaman bu kure “bakmaya deger” bir kuredir, ve arama algoritmasi bu kureden isleme devam eder. Simdiye kadar bulunmus mesafelerin en azi `knn_matches` veri yapisi icine `min_so_far` olarak saklaniyor, sozde programdaki D_{sofar} .

Bu irdeleme sonrasi (yani vs kuresinden yola devam kararı arkasından) isleme iki sekilde devam edilebilir, cunku bir kure iki turden olabilir; ya nihai en alt kurelerden biridir ve üzerinde gercek noktalar depolanmistir, ya da ara kurelerden biridir (sona gelmedik ama dogru yoldayiz, daha alta inmeye devam), o zaman fonksiyon yine ozyineli bir sekilde bu kurenin cocuklarına bakacaktır - her cocuk için kendi kendini cagiracaktır. Ikinci durumda, kurede noktalar depolanmistir, artik basit lineer bir sekilde o tum noktalara teker teker bakilir, eldekilerden daha yakin olani alinir, eldeki liste sismeye baslamissa (k’den daha fazla ise) en buyuk noktalardan biri atilir [3], vs.

Daha alta inmemiz gereken birinci durumda yapilan iki cagrinin bir ozelligine dikkat cekmek isterim. Yeni noktanin bu cocuklara olan uzakligi da olculuyor, ve en once, en yakin olan cocuga dogru bir ozyineleme yapiliyor. Bu nokta cok onemli: niye boyle yapildi? Cunku icinde muhtemelen daha yakin noktalarin olabilecegi kurelere dogru gidersek, ozyineli cagrilarin teker teker bitip yukari dogru cikmaya baslamasi ve kaldiklari yerden bu sefer ikinci cocuk cagrilarini yapmaya baslamasi ardindan, elimizdeki `knn_matches` üzerinde en yakin noktaları buyuk bir ihtimalle zaten bulunmus olacagiz. Bu durumda ikinci cagri yapilsa bile tek bir alt sinir hesabi o kurede dikkate deger hicbir nokta olamayacagini ortaya cikaracak (cunku en iyiler zaten elimizde), ve ikinci cocuga olan cagrilar hic alta inmeden pat diye geri donecektir, hic asagi inilmeyecektir.

Bu muthis bir kazanimdir: zaten bu stratejiye liteturde “budamak (pruning)” adi veriliyor, bu da cok uygun bir kelime aslinda, cunku agaclarla ugrasiyoruz ve bir dugum (kure) ve onun altindaki hicbir alt kureye ugramaktan kurtularak o dallarin tamamini bir nevi “budamis” oluyoruz. Bir suru gereksiz islemden de kurtuluyoruz bu arada, ve aramayi hizlandiriyoruz.

Mesafeler

Algoritmanin mesafeleri anlatan kisiminda norm ve uzaylar gibi kavramlardan bahsettik. Yeni noktanin mihenke olan uzakliginin o kure icindeki tum diger noktalara olan uzakligini temsil edebilecegini soyledik: peki niye bu kavramlari direk bu sekilde anlatmadik, ve norm, ugensel esitsizlik gibi kavramlardan bahsettik? Cunku

2 ve 3 boyut sonrasi uzaylari gorsel olarak dusunmek mumkun degildir, istedigimiz kadar ellerimizi kollarimizi sallayalim, bu kavramlari gorsel olarak tarif edemeyiz, ve degisik bir norm (mesafe) olcutu kullanmayi secebiliriz. Bu her iki durumda da elimizde soyut matematik baglaminda saglam bir temel oldugunu bilmek algoritmanin genelligini, ve degisik sartlarda uygulanabilirliğini arttirir. Mesela Oklit mesafesi yerine Manhattan mesafesi kullansam bile, bu mesafenin olcutunun norm kurallarini uydugunu bildigim icin kNN yapisinin geri kalanini oldugu gibi kullanabilirim, cunku o yapinin gecerliligini normlar uzerinde gecerli ucgenel esitsizlik uzerinde ispat ettim.

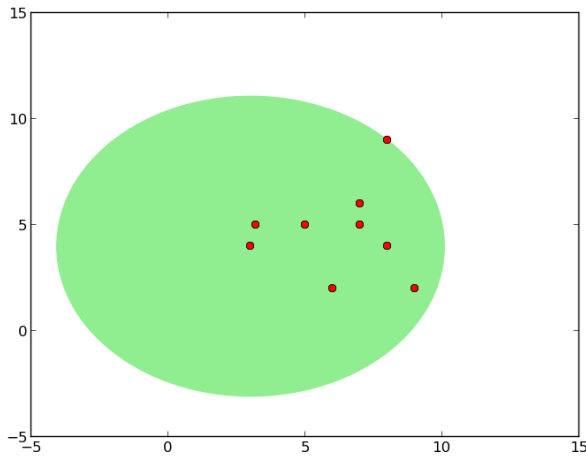
Model

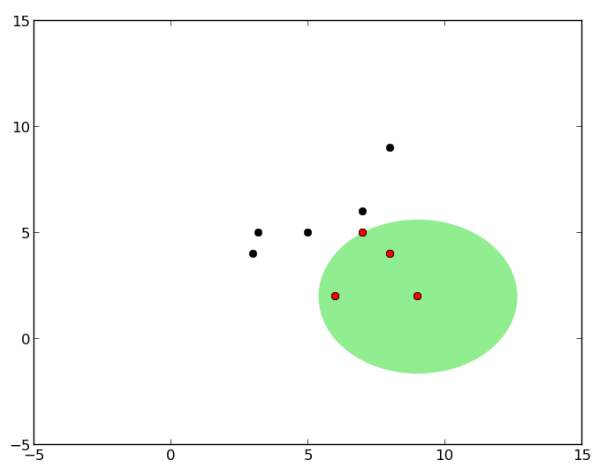
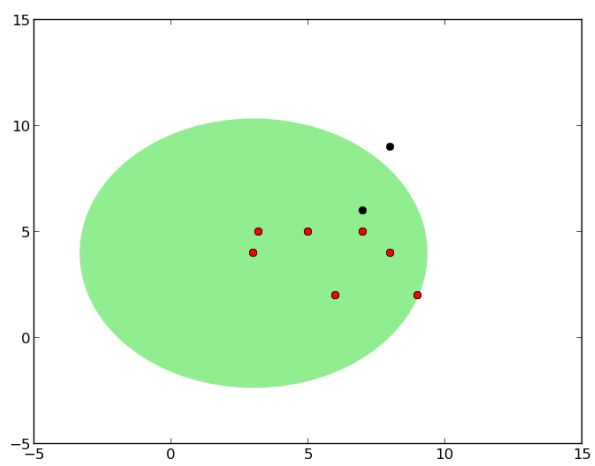
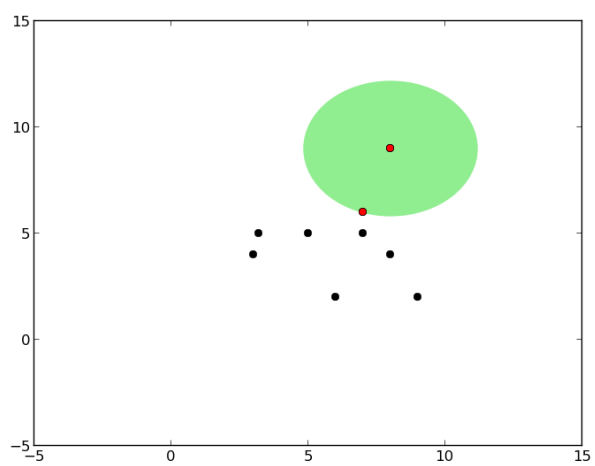
kNN'in model kullanmayan, model yerine verinin kendisini kullanan bir algoritma olarak tanittik. Peki “egitim” evresi sonrasi ele gecen kureler ve agac yapisi bir nevi model olarak gorulebilir mi?

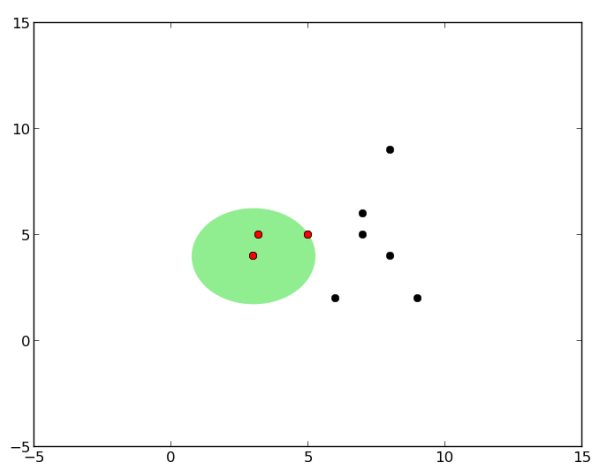
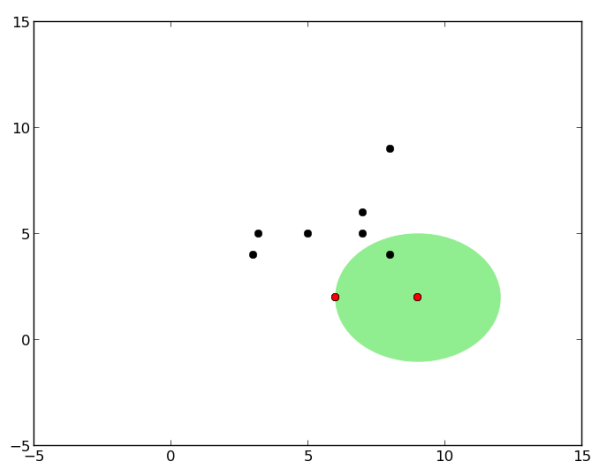
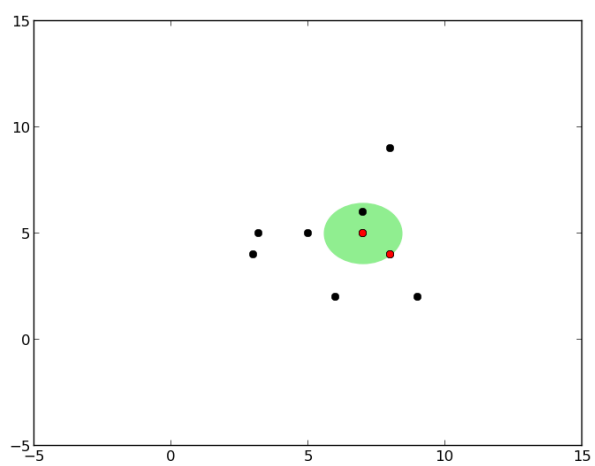
Bu onemli bir soru, ve bir bakima, evet agac yapisi sanki bir modelmis gibi duruyor. Fakat, mesela istatistiksel, grafiksel, yapay sinir aglari (neural net) baglaminda bakilirsam bu yapiya tam bir model denemez. Model bazli metotlarda model kurulunca veri atilir, ona bir daha bakilmaz. Fakat kNN, kure ve agac yapisini hala eldeki veriye erismek icin kullanmaktadır. Yani bir bakima veriyi “indeksliyoruz”, ona erisimi kolaylastirip hizlandiriyoruz, ama ondan model cikartmiyoruz.

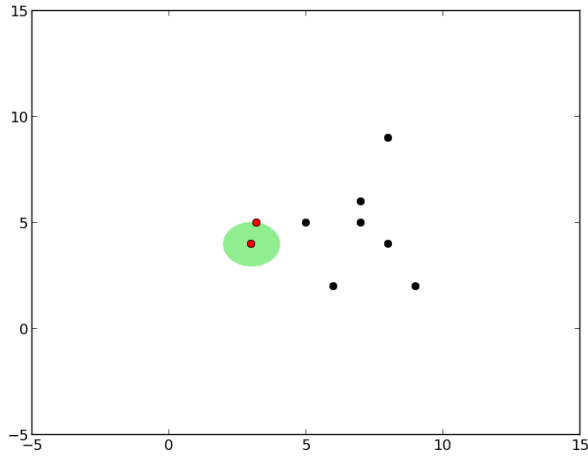
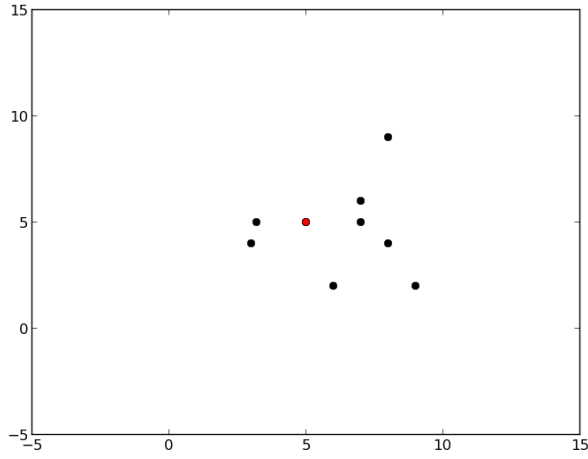
Not: Verilen Python kodu ve algoritma yakin noktalar hesapliyor sadece, onlari etiketlerinden hareketle yeni noktanin etiketini tahmin etme asamasini gerceklestirmiyor. Fakat bu son asama isin en basit tarafı, egitim veri yapısına eklenecek bir etiket bilgisi ve siniflama sonrasi k noktanin agirlikli etiketinin hesabi ile basit sekilde gerceklestirilebilir.

Agaci olusumu sirasinda kurelerin grafigi alttadir.









Kaynaklar, Notlar

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

[2] Alpaydın, *Introduction to Machine Learning*

[3] Silme islemi ornek kodumuzda Python `del` ile gercekleştirildi. Eger bu islem de hizlandirilmak istenirse, en alt kure seviyesindeki veriler bir oncelik kuyruğu (priority queue) uzerinde tutulabilir, ve silme islemi hep en sondaki elemani siler, ekleme islemi ise yeni elemani (hep sirali olan) listede dogru yere koyar.