

K-Means Kumeleme Metodu

Yapay Ogrenim (Machine Learning) alanında populer kumeleme algoritmalarından biri k-means algoritmasıdır. K-means kumelemesinde kaç tane kumenin olması gerektiği bastan tanımlanır (k parametresi ile), algoritma bunu kendisi bulmaz. Metotun geri kalanı basit - bir dongu (iteration) içinde her basamakta:

- 1) Her nokta için, eldeki kume merkezleri teker teker kontrol edilir ve o nokta en yakın olan kumeye atanır
- 2) Atamalar tamamlandıktan sonra her kume içinde hangi noktaların olduğu bilindiği için her kumedeki noktaların ortalaması alınarak yeni kume merkezi hesaplanır. Eski merkez hesapları atılır.
- 3) Basa donulur

Dongu tekrar ilk adima dondugunde, bu sefer yeni kume merkezlerini kullanarak, aynı adımlar tekrar yapılacaktır.

Fakat bir problem yok mu? Daha birinci dongu baslamadan kume merkezlerinin nerede oldugunu nereden bileceğiz? Burada bir tavuk-yumurta problemi var, kume merkezleri olmadan noktaları atayamayız, atama olmadan kume merkezlerini hesaplayamayız.

Bu probleme pratik bir çözüm ilk basta kume merkezlerini (ya da kume atamalarını) rasgele bir şekilde seçmektir. Pratikte bu yöntem çok iyi işliyor. Tabii bu rasgelelik yüzünden K-means'in doğru sonuca yaklaşması (convergence) garanti değildir, ama gerçek dünya uygulamalarında çoğunlukla kullanışlı kümeler bulunur. Bu potansiyel problemlerden kaçınmak için k-means pek çok kez işletilebilir (her seferinde yeni rasgele başlangıçlarla yani) ve aynı sonuca ulaşıp ulaşılmadığı kontrol edilebilir.

Pek en iyi k nasıl bulunur? Burada da yapay öğrenim literatüründe pek çok yaklaşım vardır [1], veriyi pek çok parçaya bölüp, farklı k kume sayısı için kumeleme yapmak ve capraz sağlama (cross-validation) kullanmak, SVD kullanarak grafiğe bakmak (bu yazının sonunda anlatılıyor), vs.

K-Means EM algoritmasının bir türevi olarak kabul edilebilir, EM kümeleri bir Gaussian (ya da Gaussian karışımı) gibi görür, ve her basamakta bu dağılımların merkezini, hem de kovaryansını hesaplar. Yani kumenin "şekli" de EM tarafından saptanır. Ayrıca EM her noktanın tüm kümelere olan üyeliklerini "hafif (soft)" olarak hesaplar (bir olasılık ölçütü üzerinden), fakat K-Means için bu atama nihai (hard membership). Nokta ya bir kumeye aittir, ya da değildir.

EM'in belli şartlarda yaklaşıksallığı için matematiksel ispat var. K-Means akıllı tahmin yaparak (heuristic) çalışan bir algoritma olarak biliniyor. Sonuca yaklaşması bu sebeple garanti değildir, ama daha önce belirttiğimiz gibi pratikte faydalıdır. Bir sürü alternatif kumeleme yöntemi olmasına rağmen hala K-Means'den vazgeçilemiyor! Burada bir etken de K-Means'in çok rahat paralelleştirilebilmesi. Bu konu başka bir yazıda işlenecek.

Örnek test verisi altta

```

from pandas import *
data = read_csv("synthetic.txt",names=['a','b'],sep=" ")
print data.shape
data = np.array(data)

```

```

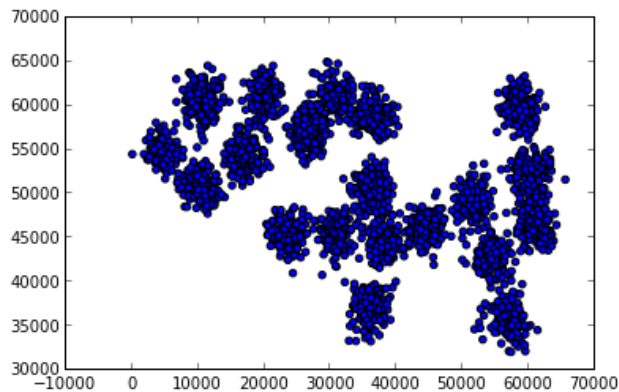
(3000, 2)

```

```

plt.scatter(data[:,0],data[:,1])
plt.savefig('kmeans_1.png')

```



```

def euc_to_clusters(x,y):
    return np.sqrt(np.sum((x-y)**2, axis=1))

class KMeans():
    def __init__(self,k,iter):
        self.k = k
        self.iter = iter
    def fit(self,X):
        # her veri noktası için rasgele kume merkezi ata
        labels = [random.randint(0,self.k-1) for i in range(X.shape[0])]
        self.labels_ = np.array(labels)
        self.centers_ = np.zeros((self.k,X.shape[1]))
        for i in range(self.iter):
            # yeni kume merkezleri uret
            for j in range(self.k):
                # eger kume j icinde hic nokta yoksa, ortalama (mean)
                # hesabi yapma, cunku o zaman nan degeri geliyor, ve
                # hesabin geri kalani bozuluyor.
                if len(X[self.labels_ == j]) == 0: continue
                center = np.mean(X[self.labels_ == j],axis=0)
                self.centers_[j,:] = center
            # her nokta için kume merkezlerine gore kume atamasi yap
            self.labels_ = []

```

```

        for point in X:
            c = np.argmin(euc_to_clusters(self.centers_, point))
            self.labels_.append(int(c))

        self.labels_ = np.array(self.labels_)

cf = KMeans(k=5,iter=20)
cf.fit(data)
print cf.labels_

[2 2 2 ..., 0 0 0]

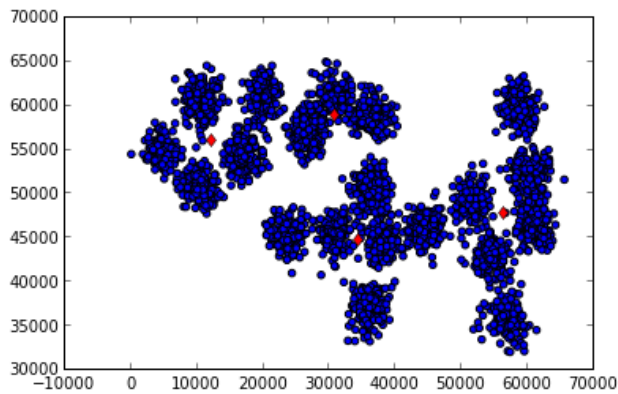
```

Ustteki sonucun icinde iki ana vektor var, bu vektorlerden birincisi icinde 2,0, gibi sayilar goruluyor, bu sayilar her noktaya tekabul eden kume atamalari. Ikinci vektor icinde iki boyutlu k tane vektor var, bu vektorler de her kumenin merkez noktasi. Merkez noktalarini ham veri uzerinde grafiklersek (kirmizi noktalar)

```

plt.scatter(data[:,0],data[:,1])
plt.hold(True)
plt.ylim([30000,70000])
for x in cf.centers_: plt.plot(x[0],x[1], 'rd')
plt.savefig('kmeans_2.png')

```

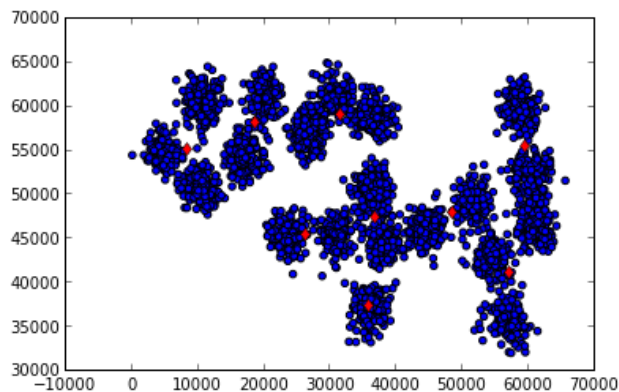


Goruldugu gibi 5 tane kume icin ustteki merkezler bulundu. Fena degil. Eger 10 dersek

```

cf = KMeans(k=10,iter=30)
cf.fit(data)
plt.scatter(data[:,0],data[:,1])
plt.ylim([30000,70000])
plt.hold(True)
for x in cf.centers_: plt.plot(x[0],x[1], 'rd')
plt.savefig('kmeans_3.png')

```



Kategorik ve Numerik Iceren Karisik Veriler

Bazen verimiz hem kategorik hem de numerik degerler iceriyor olabilir, KMeans yeni kume merkezlerini hesaplarken ortalama operasyonu kullandigi icin sadece numerik veriler uzerinde calisabilir (kategorik verilerin nasil ortalamasini alalim ki?). Bu durumda ne yapacagiz?

Bir secenek su olabilir, kategorik her kolonu her degisik degeri bir yeni kolona tekabul edecek sekilde saga dogru acariz, ve o degerin yeni kolonuna 1 degeri digerlerine 0 degeri veririz. Bu kodlamaya 1-in-q kodlamasi, 1-in-n kodlamasi, ya da Ingilizce one-hot encoding ismi veriliyor.

Ornek olarak UCI veri bankasindan Avustralya Kredi Verisine bakalim:

```
import pandas as pd
df = pd.read_csv("crx.csv")
print df[:2]
```

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
0	b	30.83	0.00	u	g	w	v	1.25	t	t	1	f	g	00202	0	+
1	a	58.67	4.46	u	g	q	h	3.04	t	t	6	f	g	00043	560	+

Bu veride A1, A2, gibi kolon isimleri var, kategorik olanlarda 'g','w' gibi degerler goruluyor. Bu kolonlari degistirmek icin

```
from sklearn.feature_extraction import DictVectorizer
def one_hot_dataframe(data, cols, replace=False):
    vec = DictVectorizer()
    mkdict = lambda row: dict((col, row[col]) for col in cols)
    vecData = pd.DataFrame(vec.fit_transform(data[cols]).toarray())
    vecData.columns = vec.get_feature_names()
    vecData.index = data.index
    if replace is True:
        data = data.drop(cols, axis=1)
        data = data.join(vecData)
```

```

    return (data, vecData, vec)

df2, _, _ = one_hot_dataframe(df, ['A1', 'A4', 'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13'])
print df2.ix[0]

A2      30.83
A3       0.00
A8       1.25
A11      1.00
A14     202.00
A15       0.00
A16       1.00
A10=f     0.00
A10=t     1.00
A12=f     1.00
A12=t     0.00
A13=g     1.00
A13=p     0.00
A13=s     0.00
A1=a      0.00
A1=b      1.00
A4=l      0.00
A4=u      1.00
A4=y      0.00
A5=g      1.00
A5=gg     0.00
A5=p      0.00
A6=aa     0.00
A6=c      0.00
A6=cc     0.00
A6=d      0.00
A6=e      0.00
A6=ff     0.00
A6=i      0.00
A6=j      0.00
A6=k      0.00
A6=m      0.00
A6=q      0.00
A6=r      0.00
A6=w      1.00
A6=x      0.00
A7=bb     0.00
A7=dd     0.00
A7=ff     0.00
A7=h      0.00
A7=j      0.00

```

```
A7=n      0.00
A7=o      0.00
A7=v      1.00
A7=z      0.00
A9=f      0.00
A9=t      1.00
Name: 0, dtype: float64
```

İşlem sonucunda A12=f mesela için 1 verilmiş, ama A12=t (ve diğer her mümkün değer için yani) 0 değeri verilmiş (sadece bu tek satır için). Böylece kategorik veriyi sayısal hale çevirmiş olduk.

Fakat isimiz bitti mi? Hayır. Şimdi KMeans bu tür veriyle acaba düzgün çalışır mıydı onu kendimize soralım. İçinde pek çok 0, bazen 1 içeren veri satırları arasında uzaklık hesabı yapmak ise yarar mı?

Yapay Öğrenim literatüründe bu tür veriler üzerinde kosinus benzerliği (cosine similarity) kullanmak daha yaygındır. Bu konuyu *SVD, Toplu Tavsiye* yazısında daha iyi görebilirsiniz. Kosinus benzerliği bize 0 ile 1 arasında bir değer döndürür. Benzerliği uzaklığa çevirmek için basit bir şekilde 1-benzerlik formülünü kullanabiliriz.

O zaman karışık veriler üzerinde KMeans kullanmak için, verinin en baştan numerik olan kısmı için Oklit uzaklığı, diğer kalan kısmı için kosinus uzaklığı kullanabiliriz. Her iki kısımdan elde edilen uzaklık değerlerini toplarız.

```
from numpy import linalg as la
import pandas as pd, os
import scipy.sparse as sps
import numpy, random

def cos_dist(inA,inB):
    num = float(np.dot(inA.T,inB))
    denom = la.norm(inA)*la.norm(inB)
    sim = 0.5+0.5*(num/denom)
    return 1. - sim

def mixed_to_clusters(vect,x,euc_n,weights):
    res1 = euc_to_clusters(vect[:,0:euc_n],x[0:euc_n])
    res2 = map(lambda y: cos_dist(x[euc_n:],y), vect[:,euc_n:])
    res = np.array(res1)*weights[0] + np.array(res2)*weights[1]
    return res

class MixedKMeans():
    def __init__(self,k,iter,euc_n,weights=[1.,1.]):
        self.k = k
        self.iter = iter
        self.euc_n = euc_n
        self.weights = weights
```

```

def fit(self,X,iter=10):
    labels = [random.randint(0,self.k-1) for i in range(X.shape[0])]
    self.labels_ = np.array(labels)
    self.centers_ = np.zeros((self.k,X.shape[1]))
    for i in range(self.iter):
        for j in range(self.k):
            if len(X[self.labels_ == j]) == 0: continue
            center = np.mean(X[self.labels_ == j],axis=0)
            self.centers_[j,:] = center
        self.labels_ = []
        for point in X:
            c = np.argmin(mixed_to_clusters(self.centers_, \
                                           point, self.euc_n,self.weights))
            self.labels_.append(int(c))

        self.labels_ = np.array(self.labels_)

df = pd.read_csv("crx.csv",sep=',',na_values=['?'])
df = df.dropna()

df['A16'] = df['A16'].str.replace('+','1')
df['A16'] = df['A16'].str.replace('-','0')
df['A16'] = df['A16'].astype(int)

df2, _, _ = one_hot_dataframe(df, ['A1','A4','A5','A6','A7','A9','A10','A12','A13'])
df2 = df2.drop('A16',axis=1)

df2 = np.array(df2)

# veriyi normalize et, ortalama cikar ve standart sapmaya bol
df2 -= np.mean(df2, axis=0)
df2 /= np.std(df2, axis=0)

cf = MixedKMeans(2,iter=10,euc_n=6,weights=[1.,3.])
cf.fit(df2)

labels_true = np.array(df['A16'])
labels_pred = cf.labels_
match = np.sum((labels_true == labels_pred).astype(int))
print float(match)/len(df)

0.816232771822

```

Bu veri icinde iki tane kume vardi, kumeler A16 kolonunda + ya da - olarak isaretli. Kumeleme takip edilmeyen (unsupervised) bir Yapay Ogrenim metotudur, hangi noktanin hangi kumeye ait oldugunu onceden bilmeyiz, ornek veri setini kullanirken bu isaretleri gormemizlikten geliyoruz, sadece kontrol amaclı olarak sonradan bu

isaretlere bakıyoruz. Ve üstteki kod ile yüzde 82 civarı (bazen 82, bazen 18 değeri gelebilir, çünkü küme atamaları “1. küme” ya da “0. küme” arasında bir seçim yapamaz, sadece onların ayrı küme olduklarını bulabilir, tahminler bu sebeple bazen tam tersi çıkabiliyor, fakat bu çok önemli değil) oranında bir başarıyla doğru kümeyi bulabilmisiz.

Parametre olarak geçilen `euc_n` değişkeni her veri noktası için “ilk kaç noktanın numerik” olduğunu belirtiyor. Böylece uzaklık fonksiyonu sadece o kısımda Oklit uzaklığı kullanıyor. Peki numerik değerler niye hep basta? Sebep `one_hot_dataframe` çağırışının yeni kolonları yaratırken eskileri silmesi ve eklenen yeni kolonların hep en sona konması, böylece en baştakiler hep numerik kolon olacak!

Ağırlıklar

Oklit ve kosinus uzaklıklarını birbirine toplarken, birine diğerinden daha fazla ağırlık vermek mümkün, belki de bir veri seti için numerik veriler kategorik olanlardan daha önemli olabilir, bu durumda ağırlıkları, mesela üstte 1’e 3 olarak tanımladık, Oklit uzaklığına 3 kat daha fazla önem / ağırlık vermiş oluruz çünkü kategorik verileri 3 kat daha “fazlaletiriyoruz”, “uzakletiriyoruz”. Tabi bu konuyu değişik bir açıdan görmek de mümkün, eğer kategorik kısmın sayıları numerik olanlar ile aynı ölçekte değilse, carparak her iki kısmı eşitlemiş olduğumuz da söylenebilir. Her neyse - ağırlıkların ne olduğu tahmin, deneme / yanılma ile bulunabilir, her veri setine göre değişik olacaktır.

Normalize Etmek

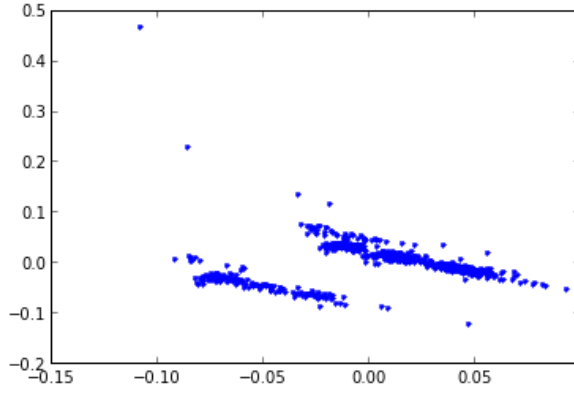
Üstteki örnekte veriyi 1-in-n kodlamasıyla çevirdikten sonra bir de normalize ettik, yani her kolon bazında o kolonun ortalamasını o kolondaki tüm değerlerden çıkarttık ve standart sapmaya bolduk, böylece her kolonu 0 etrafında ortalayıp onun iki tarafına düşebilecek -/+ değerleri küçültüyoruz. Bu veriyi bir tür “sekle sokma” işlemidir, ne zaman kullanılacağı tecrübeyle ortaya çıkar, mesela üstteki karışık veride bunun işleyebileceğini tahmin ettik.

Küme sayısını bulmak

KMeans’e küme sayısının önceden verilmiş olması gerekiyor, ve bu sayıyı KMeans bağlamında bastan bilmiyoruz. Bu sayıyı bir şekilde bulmanın yolu olamaz mı?

Boyut azaltma tekniği SVD yardımcı olabilir. SVD sonrası gelen matrisin “önemli” kolonlarında daha azaltılmış bir veri seti elde edebiliriz, bu veride en basta olan kolonlar en önemli olanlardır, ve bu kolonları, mesela ilk ikisini alarak ekrana basabiliriz. Avustralya Kredi setinde bunu yaparsak şunu görürüz:

```
import scipy.linalg as lin
u,s,v=lin.svd(df2) # normalize edilmiş veri üzerinde
plt.plot(u[:,0], u[:,1],'.')
plt.savefig('kmeans_4.png')
```

İki tane ana blok olduğu açık bir şekilde görülüyor. Demek ki küme sayısı $k = 2$ kullanmak gerekir.

Bazı ek notlar

[1] http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set

[2] nbviewer.ipython.org/url/cbcb.umd.edu/~hcorrada/PML/src/kmeans.ipynb