Index

Supervised learning versus Unsupervised learning

- In supervised learning, classes are predetermined. The classifier is trained based on a set of examples of correct answers.
- In unsupervised learning, classes are automatically discovered and there is not a training set.

  The goal is to group the data based on their similarity. We call these groups *clusters*.

Data clustering

- We group data, generating clusters, according to their similarity.
- The similarity measure depends on the characteristics of the dataset and the intended use of the results.
- It is an iterative process of knowledge discovery. It is not necessarily a greeddy process, it can involve trial and error.
- Used in applications of data-intensive processing.
  - In business and marketing, detection of groups of buyers with similar buying.
  - In computational biology, detection of genes that exhibit similar behavior.
  - In crime analysis, detection of areas with higher incidence of certain crimes.
- We will see two algorithms (Hierarchical and K-Means clustering)

t1 Consult http://en.wikipedia.org/wiki/Cluster_analysis

Ex: Pigeonholing the bloggers

Objective: We want to identify whether there are bloggers who write about similar topics to group them thematically.

Thus we can complete the automation of the search, category detection and cataloging of blogging.

Preparing the data

First of all we must identify a set of numerical attributes that will be used to compare different *items* or objects.
The need for numeric attributes depends on the measures of similarity we use.
We will group blogs according to the frequency of the words.

In the Blogdata.txt file you can find the information collected from about 120 blogs.

Preparing the data

The blogdata.txt file contains a table with information from blogs where the columns are tab spaced.

Example:

Blog	china	kids	music	yahoo
Gothamist	0	3	3	0
GigaOM	6	0	0	2
QuickOnlineTips	0	2	2	22

The first row contains the words that have been selected (china, kids, music, yahoo).

The remaining rows contain the frequencies of the words in a given blog.

It makes no sense to put all the words, we can establish a lower bound (10%) and an upper bound (50%). The idea is to rule out the words that are too frequent or too rare.

Reading the data

t2 Incorporate the following code into a file clusters.py:

```
def readfile(filename):
  lines=[line for line in file(filename)]
  # First line is the column titles
  colnames=lines[0].strip().split('\t')[1:]
  rownames=[]
  data=[]
  for line in lines[1:]:
    p=line.strip().split('\t')
    # First column in each row is the rowname
    rownames.append(p[0])
    # The data for this row is the remainder of the row
    data.append([float(x) for x in p[1:]])
  return rownames,colnames,data
```

- Test the code:

```
>>> import clusters
>>> blognames, words, data = clusters.readfile('blogdata.txt')
```

Index

Connectivity based clustering (hierarchical clustering)

- Initially, each *item* forms a (*cluster*).
- Construct a hierarchy of clusters iteratively joining the two closest (or similar).
- The attributes of the new cluster are obtained by averaging the corresponding attributes in the original clusters.
- It stops when there is only one group.
- The typical way to visualize the result is via a dendrogram (http://en.wikipedia.org/wiki/Dendrogram). This tree structure graphically preserves the distance between clusters.

Distance between two groups (clusters)

As we have seen we need to compute how similar (distant) are two clusters. We can use different measures [1]:

- Euclidean distance.
- square Euclidean distance.
- Manhattan distance.
- Pearson correlation.
- Pearson correlation square.
- Chebychev distance.
- Spearman distance.

Note: We normalize the functions that implement these measures so that they return a number between 0 and 1, where 1 represents that the attributes of the clusters we compare are identical.

[1] http://www.improvedoutcomes.com/docs/WebSiteDocs/Clustering/Clustering_Parameters/Distance_Metrics_Overview.htm

Distance between two groups (clusters)

t3 Incorporate a function in order to compute the normalized Euclidean distance:

```
from math import sqrt
def euclidean(v1,v2):
...
return 1/(1+distance)
```

Distance between two groups (clusters)

- Incorporate the following function that computes the normalized Pearson correlation:

```
def pearson(v1,v2):
  # Simple sums
  sum1 = sum(v1)
  sum2 = sum(v2)

  # Sums of the squares
  sum1Sq = sum([pow(v,2) for v in v1])
  sum2Sq = sum([pow(v,2) for v in v2])

  # Sum of the products
  pSum = sum([v1[i]*v2[i] for i in range(len(v1))])

  # Calculate r (Pearson score)
  num = pSum-(sum1*sum2/len(v1))
  den = sqrt((sum1Sq-pow(sum1,2)/len(v1))*(sum2Sq-pow(sum2,2)/len(v1)))
  if den==0: return 0

  return 1.0-num/den
```

Implementing the hierarchical clustering

- A cluster is either an internal node with two branches or a leaf of the dendogram associated to an item.
- Each cluster contains the information of an *item* if it is a leaf or the union of the clusters that groups.
- The creation of a cluster is performed by assigning to each attribute the average of the respective attributes of the two clusters that groups. For efficiency, it is advisable to keep the distance between of the clusters we group.

t4 Incorporate the following code:

```
class bicluster:
  def __init__(self,vec,left=None,right=None,dist=0.0,id=None):
    self.left = left
    self.right = right
    self.vec = vec
    self.id = id
    self.distance = distance
```

Lets program ...

t5 Complete and incorporate the algorithm *hcluster*:

```
def hcluster(rows,distance=pearson):
  distances={} # cache of distance calculations
  currentclustid=-1 # non original clusters have negative id

  # Clusters are initially just the rows
  clust = [bicluster(rows[i],id=i) for i in range(len(rows))]

  while ... : # termination criterion
    lowestpair = (0,1)
    closest = distance(clust[0].vec,clust[1].vec)

    # loop through every pair looking for the smallest distance
    for i in range(len(clust)):
      for j in range(i+1,len(clust)):
        if (clust[i].id,clust[j].id) not in distances:
          distances[(clust[i].id,clust[j].id)] = ...

        # update closest and lowespair if needed
        ...
```

Lets program ...

```
    # inside while loop ..

    # calculate the average vec of the two clusters
    mergevec = ...

    # create the new cluster
    newcluster = bicluster(...)

    currentclustid-=1
    del clust[lowestpair[1]]
    del clust[lowestpair[0]]
    clust.append(newcluster)

return clust[0]
```

Solution

```
def hcluster(rows,distance=pearson):
  distances={} # stores the distances for efficiency
  currentclustid=-1 # all except the original items have a negative id

  # Clusters are initially just the rows
  clust = [bicluster(rows[i],id=i) for i in range(len(rows))]

  while len(clust)>1: #stop when there is only one cluster left
    lowestpair = (0,1)
    closest = distance(clust[0].vec,clust[1].vec)

    # loop through every pair looking for the smallest distance
    for i in range(len(clust)):
      for j in range(i+1,len(clust)):
        # distances is the cache of distance calculations
        if (clust[i].id,clust[j].id) not in distances:
          distances[(clust[i].id,clust[j].id)] = distance(clust[i].vec,clust[j].vec)

        d = distances[(clust[i].id,clust[j].id)]

        if d < closest:
          closest = d
          lowestpair = (i,j)

    # calculate the average of the two clusters
    mergevec=[
    (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])/2.0
    for i in range(len(clust[0].vec))]
```

Solution

```
  # create the new cluster
  newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
                       right=clust[lowestpair[1]],
                       dist=closest,id=currentclustid)

  # cluster ids that weren't in the original set are negative
  currentclustid-=1
  del clust[lowestpair[1]]
  del clust[lowestpair[0]]
  clust.append(newcluster)

return clust[0]
```

Visualizing the results

t6 Incorporate the following code:

```
def printclust(clust,labels=None,n=0):
  # indent to make a hierarchy layout
  for i in range(n): print ' ',
  if clust.id<0:
    # negative id means that this is branch
    print '-'
  else:
    # positive id means that this is an endpoint
    if labels==None: print clust.id
    else: print labels[clust.id]

  # now print the right and left branches
  if clust.left!=None:
    printclust(clust.left,labels=labels,n=n+1)
  if clust.right!=None:
    printclust(clust.right,labels=labels,n=n+1)
```

Visualizing the results

- Test the previous code:

```
>>> import clusters
>>> blognames, words, data = clusters.readfile('blogdata.txt')
>>> clust = clusters.hcluster(data)
>>> clusters.printclust(clust, labels=blognames)
```

Building the dendogram

- Install the Python Imaging Library (PIL) (http://pythonware.com)
- Incorporate the code from dendrogram.py
- Generate the dendogram:

```
>>> import clusters
>>> blognames, words, data = clusters.readfile('blogdata.txt')
>>> clust = clusters.hcluster(data)
>>> clusters.printclust(clust, labels=blognames)
>>> clusters.drawdendrogram(clust, blognames,
                            jpeg='blogclust.jpg')
```

Column clustering

Often we may want to also group by columns rather than rows.

Imagine you have a set of buyers and the products they have purchased. While we may be interested in detecting clusters of buyers, also we can be interested in detecting clusters of products.

In the example of bloggers we can be interested in detecting which words are often used together in blogs.

Actually, the only thing we need to do is to rotate the data matrix.

Note: The analysis of clusters is more effective if the number of attributes is much higher than that of *items*

Column clustering

t7 Incorporate a function *rotatematrix* that returns the transpose.

```
def rotatematrix(data):
  newdata=[]
  ...
  return newdata
```

- Test the previous code:

```
>>> import clusters
>>> blognames, words, data = clusters.readfile('blogdata.txt')
>>> rdata = clusters.rotatematrix(data)
>>> wordclust = clusters.hcluster(rdata)
>>> clusters.drawdendrogram(wordclust, labels=words,
                            jpeg='wordclust.jpg')
```

Considerations on hierarchical clustering

- Although we have the dendrogram, we need to do some work in order to obtain a partition that it is easy to describe
- It takes an expensive computation, and it will be a very slow algorithm in large databases.

Index

Centroid-based clustering (K-Means clustering)

- Usually the number of clusters is given by a parameter k.
- It maintains a set of points, centroids, which represent the location of the center of the clusters.
- The *items* are assigned to the cluster whose centroid is closer.
- We want to solve the following optimization problem: Assign the items to the k clusters such that the sum of the squares of the Euclidean distances of the *items* to their respective centroid is minimal
- This is an NP-Hard problem and we usually use approximate algorithms

K-Means clustering

Basic scheme of K-means clustering algorithm.

1. Set the *k* centroids randomly.
2. Assign the *items* to the nearest cluster.
3. Relocate the centroids in the average location of the *items* belonging to its cluster.
4. Recalculate the assignment of the *items* to the clusters.
5. If there is a *item* moving to another cluster go back to 3.

K-Means clustering

t8 Incorporate the function kcluster that implements the algorithm K-Means described previously:

```
import random

def kcluster(rows,distance=pearson,k=4):

  # Determine the minimum and maximum values for each point
  ranges=[(min([row[i] for row in rows]),
  max([row[i] for row in rows])) for i in range(len(rows[0]))]

  # Create k randomly placed centroids
  clusters=[[random.random()*(ranges[i][1]-ranges[i][0])+ranges[i][0]
  for i in range(len(rows[0]))] for j in range(k)]
```

K-Means clustering

```
lastmatches=None
for t in range(100):
  bestmatches=[[] for i in range(k)]

  # Find which centroid is the closest for each row
  for j in range(len(rows)):
    row=rows[j]
    bestmatch=0
    for i in range(k):
      d=distance(clusters[i],row)
      if d<distance(clusters[bestmatch],row): bestmatch=i
    bestmatches[bestmatch].append(j)

  # If the results are the same as last time, done
  if bestmatches==lastmatches: break
  lastmatches=bestmatches

  # Move the centroids to the average of their members
  for i in range(k):
    avgs=[0.0]*len(rows[0])
    if len(bestmatches[i])>0:
      for rowid in bestmatches[i]:
        for m in range(len(rows[rowid])):
          avgs[m]+=rows[rowid][m]
      for j in range(len(avgs)):
        avgs[j]/=len(bestmatches[i])
      clusters[i]=avgs

return bestmatches
```

K-Means clustering

- Test the code:

```
>>> import clusters
>>> blognames, words, data = clusters.readfile('blogdata.txt')
>>> kclust = clusters.kcluster(data, k=10)
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
>>> [blognames[r] for r in kclust[0]]
['GigaOM', '43 Folders', 'Lifehacker', 'Wired News: Top Stories']
>>> [blognames[r] for r in kclust[1]]
["The Superficial - Because You're Ugly",
'Talking Points Memo: by Joshua Micah Marshall',
'Go Fug Yourself', 'PerezHilton.com']
...
```

Lets program ...

t9 Modify the kcluster function so that it returns the sum of the squares of the Euclidean distances of the *items* to their respective centroids. We will call it total distance.

t10 Show the total distance as a function of k.

t11 Improve the function kcluster by introducing restarting policies.