Machine Learning

Carlos José Ansótegui Gil

Área: Ciencias de la Computación e Inteligencia Artificial (CCIA)
Departamento de Informática e Ingeniería Industrial
Universidad de Lleida

September 2012

Index

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

Sources of this document

- Programming Collective Intelligence, Toby Segaran.
- http://iie.fing.edu.uy/ense/asign/recpat/material/tema3_00-01/node25.html

Index

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

Ex: Predicting Sign-ups

Objective: Detect potential clients.

We have a new Internet service. To enhance its visibility we offer free trials.

After some weeks, the users can choose to upgrade their account to a *basic* or *premium* service, or cancel the service.

During trial time, we have collected information on the users activity and the final decision.

This information is collected passively through user interaction with the system.

Data: user interaction and final decision

Referrer	Location	Read FAQ	Pages viewed	**Service**
slashdot	USA	yes	18	None
google	France	yes	23	Premium
digg	USA	yes	24	Basic
kiwitobes	France	yes	23	Basic
google	UK	no	21	Premium
(direct)	New Zealand	no	12	None
(direct)	UK	no	21	Basic
google	USA	no	24	Premium
slashdot	France	yes	19	None
digg	USA	no	18	None
google	UK	no	18	None
kiwitobes	UK	no	19	None
digg	New Zealand	yes	12	Basic
slashdot	UK	no	21	None
google	UK	yes	18	Basic
kiwitobes	France	yes	19	Basic

Lets program ...

t1 Download the data file *descision_tree_example.txt* from the virtual campus at folder */lab/learning*.

t2 Create a file named *treepredict.py*

t3 Define a function to load the data into a bidimensional list named *data*

```
def read(data, file_name):
```

The problem of classifying data

- Our goal is to obtain automatically a classifier
- We have a set of classes or categories
- We also have a set of objects defined as a function of attributes to values
- Objective: depending of the values of the attributes of a given object we want to classify it, i.e., we want to determine the category (special attribute) the object belongs to, named *goal attribute*
- In particular, we want to build a function that takes as arguments the values of the attributes of an object and returns the value of the goal attribute
- We initially have a training set or prototype set where we know the value of all the attributes, even the goal attribute

Classifying through decision trees

- Decision trees are one of the most simple methods of supervised learning
- The classifier looks like a set of *if-then* statements organized as a directed tree
- Every internal node contains a question on a particular attribute (one child per possible answer). Every arc corresponds to an answer and every leaf node refers to a decision (classification).
- The classification of the objects is performed through a sequence of questions on the attribute values. It starts at the root node, and it follows the path determined by the answers to the questions of the internal nodes, till a leaf node is reached. The label assigned to a leaf corresponds to the value assigned to the goal attribute

Classifying through decision trees

- Lets imagine that our root node contains initially the objects in our training set. Through every arc (n_1, n_2) we send a copy of the original set containing those objects that match the answer to the question of n_1. At the leaves, the remaining objects share the value of the goal attribute

Classifying through decision trees

- Classifiers based on decision trees (ID3, C4, C4.5, Bayesian trees, etc.)
- We will implement CART (Classification And Regression Trees)
- The main differences have to do with the pruning strategies and the way nodes are split
- CART applies binary partitions and uses a pruning strategy based on a cost-complexity criterion

Classifying through decision trees

The methodology we will follow, can be described into two steps:

- **Learning.** Consist in the construction of a decision tree. This is the more complex step and determines the final result
- **Classification.** Consist in the labelling of an object, not into the training set. We basically answer to all the questions associated to the internal nodes, following the path to a leaf node

Classifying through decision trees

A classification tree T represents a recursive partition of the space representation, P, based on a set of prototypes, S.

Classifying through decision trees

- Every node of T has assigned a subset of prototypes in S.
- The root node has assigned S.
- Every leave, t, has assigned a region, R_t, in P. If leaves(T) is the set of leaf nodes in T, then, $\cup_{t \in leaves(T)} R_t = P$, i.e, the sets of prototypes assigned to the leave nodes form a partition of P
- Every internal node n has assigned a region in P, which is the union of the regions assigned to the leaf nodes of the subtree with root node n
- The union of the sets of prototypes assigned to the nodes at the same level is equal to S.

Building the classification tree

We follow this recursive scheme:

- **Advance**: Partition a node according to some rule.

  (ex: evaluate a condition on an attribute).

  The prototypes that verify (falsify) the condition are assigned to the left (right). Obsv: these subsets are disjoint
- **Stop condition**: stops the partition of nodes.

  The nodes that satisfy this condition are leaf nodes.

Building the classification tree

There are still some questions to answer:

- How do we perform the partitions and how do we select the best?
- When do we declare a node to be a leaf?
- How do we assign a label to a leaf?

Depending on the answers we get different algorithms.

Selecting the partitions

- In CART partitions are binary. The answer is always yes or not
- The objective of a partition is to increase the homogeneity (class-based) of the resulting subsets
- We assign a purity measure to each partition, that we will use:
  - To select the best partition.
  - As stop criterion.

Formulating the partition rule

In CART we define the standard set of questions (Q) in the following way:

- Each partition depends on a unique attribute.
- If x_i is a categorical attribute, s.t., $x_i \in C = \{c_1, c_2, \ldots, c_k\}$, Q includes the questions:

  ¿$x_i \in C'$? , where $C' \subset C$
- If x_i is a continuous attribute, Q includes questions as : ¿$x_i \leq v$?, where v is a real number or value.

  In CART, v is the middle point of two consecutive values x_i.

Partitioning criterion

Impurity (purity) measure:
Given a function of impurity ϕ, we define the measure of impurity of a node t, named $i(t)$, as:

$$i(t) = \phi(p(1|t), p(2|t), ..., p(k|t))$$

where $p(j|t)$ is the probability that a prototype assigned to t belongs to class j, i.e.,

$$p(j|t) = \frac{N_j(t)}{N(t)}$$

$N(t)$ is the number of prototypes in t, and $N_j(t)$ is the number of prototypes of class j in t

Lets program ...

t4 Define a function *unique_counts* that counts the number of prototypes of a given class in a partition *part*. We assume that the goal attribute is the last one. We need to return a dictionary that has as keys the different classes in *part* and as values the number of prototypes of the given class

```
# Create counts of possible results
# (the last column of each row is
#  the result)
def unique_counts(part):
    results={}
    ...
    return results
```

Partitioning criterion

Lets observe that:

- $p(j|t) \geq 0$
- $\sum_k p(j|t) = \sum_k \frac{N_j(t)}{N(t)} = \frac{1}{N(t)} \sum_k N_j(t) = 1$
- The maximum impurity is obtained when all the classes are equally represented in t.
- the minimum impurity is obtained when in t all prototypes belong to the same class (maximum homogeneity).

Partitioning criteria

Criteria to measure impurity:

- Gini index: measures the expected error if a class is randomly assigned to one the prototypes in the node:

$$i(t) = \sum_{\substack{j_1, j_2 = 1 \\ j_1 \neq j_2}}^{K} p(j_1|t) \cdot p(j_2|t) = 1 - \sum_{j=1}^{K} p(j|t)^2$$

- Entropy: measures the amount of disorder or chaos in a set:

$$i(t) = -\sum_{j=1}^{K} p(j|t) \cdot log\ p(j|t)$$

Lets program ...

t5 Define a function that computes the Gini index of a node:

```
def gini_impurity(part):
  total = len(part)
  results = unique_counts(part)
  imp = 0
  ...
  return imp
```

Lets keep programming ...

t6 Define a function that computes the entropy of a node:

```
# Entropy is the sum of p(x)log(p(x))
# across all  the different possible
# results
def entropy(rows):
    from math import log
    log2 = lambda x:log(x)/log(2)
    results = unique_counts(rows)
    # Now calculate the entropy
    imp = 0.0
    ...
    return imp
```

Partitioning criteria

Goodness of a partition: The goodness of a partition s in a node t has to be related to the impurity of the node, and the impurity of the resulting nodes after the partition, t_L y t_R.

The goodness of a partition s in a node t, $\Phi(s,t)$, is defined as the decrease of impurity achieved:

$$\Phi(s,t) = \Delta i(s,t) = i(t) - p_L \cdot i(t_L) - p_R \cdot i(t_R)$$

where, p_L y p_R is the proportion of prototypes of t in t_L and t_R, respectively.

As we know how to compute $i(t)$, we can compute each partition s and select the best partition, i.e., the one that maximizes the drop in impurity.

Lets program ...

t7 Define a function that partitions a previous partition, taking into account the values of a given attribute (column). *column* is the index of the column and *value* is the value of the partition criterion.

```
# Divides a set on a specific column. Can handle
# numeric or categorical values
def divideset(part, column, value):
   isplit_function = None

   if isinstance(value, int) or isinstance(value, float):
      split_function = lambda prot: prot[column]>=value
   else:
   ...
   return (set1, set2)
```

Stop criterion

We set a value $\beta > 0$. A node t is terminal iff:

$$\max_s \Delta i(s,t) < \beta$$

- If β is low, it is complicated to stop the building process since the impurity has to be low. This can result into very large trees
- Si β is high, the height of the trees is low. Although, we may find nodes where $\max_s \Delta i(s,t)$ is low, it may be possible to find a partition of its descendants that results in a bigger decrease of the impurity.

How to label leaves

The objective is to assign a class, j, to every terminal node $t \in leaves(T)$.

The simplest way is to select the class j where $p(j|t)$ is maximum.

We can break ties randomly selecting a class.

Lets program ...

t8 Define a new class *decisionnode*, which represents a node in the tree, using the following constructor:

```
def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
```

We have five member variables.

- *col* is the column index which represents the attribute we use to split the node
- *value* corresponds to the answer that satisfies the question
- *tb* y *fb* are internal nodes, representing the positive and negative answers, respectively.
- *results* is a dictionary that stores the results for this branch. Is *None* except for the leaves.

Lets keep programming ...

t9 Define a new function *buildtree*. This is a recursive function that builds a decision tree using any of the impurity measures we have seen. The stop criterion is $\max_s \Delta i(s,t) < \beta$:

```
def buildtree(part, scoref=entropy, beta=0):
    if len(part)==0: return decisionnode()
    current_score = scoref(part)

    #Set up some variables to track the best criteria
    best_gain = 0
    best_criteria = None
    best_sets = None
    ...
    else:
       return decisionnode(results=unique_counts(part))
```

t10 Define the iterative version of the previous function.

Drawing the tree ...

t11 Include the following function:

```
def printtree(tree,indent=''):
   # Is this a leaf node?
   if tree.results!=None:
      print str(tree.results)
   else:
      # Print the criteria
      print str(tree.col)+':'+str(tree.value)+'? '

      # Print the branches
      print indent+'T->',
      printtree(tree.tb,indent+'  ')
      print indent+'F->',
      printtree(tree.fb,indent+'  ')
```

Lets program ...

t12 Build a function *classify* that allows to classify new objects. It must return the dictionary that represents the partition of the leave node where the object is classified.

```
def classify(obj, tree):
```

Evaluating a learning algorithm

A learning algorithm can be considered good if it is capable to classify accurately objects that have never seen before.

A basic methodology to evaluate a learning algorithm is the following:

1. Retrieve a wide set of examples.
2. Divide this set into two sets: the training set and the test set.
3. Build the classifier with the training set.
4. Measure the percentage of examples of the test set that are correctly classified.
5. Repeat steps 2 and 4 for different sizes of training and test sets chosen randomly.

We can plot the measure of prediction quality as a function of the size of the training set.

Lets program ...

t13 Define a function *test* that takes a test set and a training set and computes the percentage of examples correctly classified.

```
def test_performance(testset, trainingset):
```

t14 Show the quality of the classifier increasing by a 20% the training set. You can retrieve data from the following database: http://archive.ics.uci.edu/ml/

Missing data

- An advantage of the decision trees on is their ability to treat missing data
- In our working example, a user's geolocazation through the IP may not be possible
- Suppose we want to compare the variation of impurity as a function of attribute A. However, for attribute A some of the examples have no value
- Solutions:
  - We assign to the example the most frequent value of attribute A

t15 Suggest other solutions ...

Stop criterion: prunning strategy

- The use of the threshold β to stop the process of partition presents difficulties, as we have seen above
- The procedure is more complex and will be based on a pruning strategy
- Idea: build a very large tree and prune towards the root those subtrees that produce subtrees small benefits into the decrease of impurity
- Obsv: it is more efficient to prune a tree than stopping its growth. Pruning allows a subtree of a node to remain and the other disappear, while stopping the growth prunes all branches simultaneously

General prunning strategy

- Build the maximum tree (T_{max}) partitioning nodes till any of these conditions is reached:
  - The node is pure
  - $N(t) < N_{min}$ (usually, $N_{min} = 5$)
- Prune T_{max} obtaining a decreasing sequence of nested trees. if T' is obtained by prunning T, then $T \succ T'$.

$$T_{max} \succ T1 \succ T2 \succ \ldots \succ \{t\}$$

where $\{t\}$ is a tree of a single node.

- Assign an error measure to every tree and choose the least mistake

  It is expected that pruned trees will have more capacity of generalization since they are not so adjusted to the training set (the problem of overfitting)

A particular prunning strategy ...

- Build completely the decision tree.
- For every pair of leaves with a common father check if their union increases the entropy below a given *threshold*. If that is the case, delete those leaves by joining their prototypes in the father.
- Repeat till it is not possible to delete more leaves.

t16 Define a function that implements the above prunning strategy:

```
def prune(tree, threshold):
```

Index

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

Ej:*Spam* filtering

Objective: Classify documents according to their content. One of the best known applications is *spam* filtering.

Spam not only affects email, it also affects webs that have become more interactive: requesting user comments and content creation.

Whenever you create an application to request the general public to contribute, you need an strategy to eliminate spam.

The idea is to apply a learning algorithm capable of classifying documents into categories, one of which is *spam*

Document classification

- We classify documents according to their characteristics
- An *item* is a document
- We use as characteristics (*features*) the words that appear in the document. We can also use combinations of words, phrases, or anything that can be classified as absent or present in the document
- We classify as *bad* | *good* an *item* depending whether it is *spam* or not

Recognizing the words in a document

t1 Create a file docclass.py. Incorporate the function *getwords*. This function splits the text into words using as delimiters any character which is not a letter. All words are converted to lower case.

```
import re
import math

def getwords(doc):
  splitter=re.compile('\\W*')
  print doc
  # Split the words by non-alpha characters
  words=[s.lower() for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # Return the unique set of words only
  return dict([(w,1) for w in words])
```

Recognizing the words in a document

- Test the program.

```
>>> import docclass
>>> docclass.getwords('quick money at Cayman-Casino')
quick money at Cayman-Casino
{'quick': 1, 'money': 1, 'casino': 1, 'cayman': 1}
```

Which are the good *features*

- To determine which *features* to use is crucial and difficult
- They must be common enough to appear frequently, but not appearing in all documents
- Extreme cases are: the whole document or the individual characters
- If we use words, we must decide how to split them, which punctuation should be included, ...

Lets program ...

We need a class that represents our classifier. That will allow us to instantiate multiple classifiers for different users, groups, *queries,* and train them according to our particular needs

t2 Incorporate the definition of the class *classifier*:

```
class classifier:
  def __init__(self,getfeatures,filename=None):
    # Counts of feature/category combinations
    self.fc = {}
    # Counts of documents in each category
    self.cc = {}
    self.getfeatures = getfeatures
```

Lets program ...

Dictionary *fc* contains the number of appearances by category of a given *feature*. Ej:

```
{'python': {'bad': 0, 'good': 6}, 'the': {'bad':3, 'good': 3}}
```

Dictionary *cc* indicates how many times a classification has been used.

t3 Complete the following functions:

```
# Increase the count of a feature/category pair
def incf(self,f,cat):
  self.fc.setdefault(f,{})
  self.fc[f].setdefault(cat,0)
  ...

# Increase the count of a category
def incc(self,cat):
  self.cc.setdefault(cat,0)
  ...
```

Lets program ...

```
# The number of times a feature is in a category
def fcount(self, f, cat):
  if f in self.fc and cat in self.fc[f]:
    ...
  return 0.0

# The number of items in a category
def catcount(self, cat):
  if cat in self.cc:
    ...
  return 0.0

# The total number of items
def totalcount(self):
  ...

# The list of all categories
def categories(self):
  return self.cc.keys()
```

Lets program ...

t4 Incorporate a function *train* which takes an *item* and a category. This function uses *getfeatures* in order to obtain the *features* of the *item*, and calls appropriately functions *incf* and *incc*.

```
def train(self,item,cat):
  features=self.getfeatures(item)
  # Increment the count for every
  # feature with this category
  for f in features:
    ...

  # Increment the count for this category
  ...
```

Lets check the program ...

- Test the previous program.

```
>>> import docclass
>>> cl = docclass.classifier(docclass.getwords)
>>> cl.train('the quick brown fox jumps over the lazy dog','good')
the quick brown fox jumps over the lazy dog
>>> cl.train('make quick money in the online casino','bad')
make quick money in the online casino
>>> cl.fcount('quick','good')
1.0
>>> cl.fcount('quick','bad')
1.0
```

t5 Incorporate the following function in order to test your program:

```
def sampletrain(cl):
  cl.train('Nobody owns the water.','good')
  cl.train('the quick rabbit jumps fences','good')
  cl.train('buy pharmaceuticals now','bad')
  cl.train('make quick money at the online casino','bad')
  cl.train('the quick brown fox jumps','good')
```

Computing probabilities

- We are interested on computing the probability that a *feature* appears in a given category. In probability theory, a conditional probability is the probability that an event would have, conditional that another occurs
- The probability of A given B, $p(A|B)$ is, taking into account the worlds where B holds, the fraction in which also A holds, i.e., $p(A|B) = \frac{p(A \cap B)}{p(B)}$

t6 Define and incorporate a function which computes the probability that a given *feature* appears into a given category

```
def fprob(self, f, cat):
  if self.catcount(cat)==0: return 0
  # The total number of times this feature appeared
  # in this category divided by the total number of
  # items in this category
  ...
```

Computing probabilities

- Test the program:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl = docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
Nobody owns the water.
the quick rabbit jumps fences
buy pharmaceuticals now
make quick money at the online casino
the quick brown fox jumps
>>> cl.fprob('quick','good')
0.6666666666666663
```

Assumed probabilities

- *fprob* is particularly sensitive in the early stages of training and for words that do not appear so often. Ej:
  The word 'money' only appears into one document classified as 'bad'. Therefore, its *fprob* for class 'good' is 0
- It is more realistic that *fprob('money','good')* gradually approaches 0 as 'money' is found in more and more documents of category 'bad'
- One approach to avoid that, is to think about an assumed probability. For example, we can start with 0.5
- We must establish which weight has this assumed probability. For example, a weight of 1 means that the assumed probability is weighted the same as one word. The weighted probability returns the weighted average between the basic probability and the assumed probability

Assumed probabilities

- In the example of 'money', the weighted probability for the word money starts at 0.5 for all categories. When the classifier detects 'money' into a 'bad' document, the probability of ('money','bad') becomes a 0.75, i.e.,:

$$(weight * assumedprob + count * fprob)/(count + weight)$$

$$(1 * 0.5 + 1 * 1)/(1 + 1) = 0.75$$

where *count* is the number of times this *feature* has appeared in all the classes.

t7 Which is the weighted probability for ('money','good')?

Assumed probabilities

t8 Define a function that computes the weighted probability.

```
def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
  # Calculate current probability
  basicprob = prf(f,cat)

  # Count the number of times this feature has appeared in
  # all categories
  totals = ...

  # Calculate the weighted average
  wp = ...
  return wp
```

Assumed probabilities

- Load twice our training set, and analyze the result of the weighted probability for ('money','good') after each load:

```
>>> import docclass
>>> cl = docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.25
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.16666666666666666
```

Naive Bayesian Classifier

- We need to combine all the previous individual probabilities in order to obtain the probability that a given *item* belongs to a class
- Naive: because it is assumed that the probabilities that we have to combine are independent
- Although the probability corresponding to an *item* does not represent the true probability of belonging to a class, you can compare the results for various classes and choose the biggest probability
- Bayes: because we will use the Bayes' Theorem in order to invert the conditional probabilities
- The previous strategy has been shown to be effective as a classification criterion for documents

Probability of an *item* given a category

- In the first place, we must determine the probability of an *item* given a category.
- Assuming that the probabilities are independent we just need to multiply the probabilities of all the *features* that appear into the *item*. Ex:
  If $p('python'|'bad') = 0.2$ and $p('casino'|'bad') = 0.8$, we say that the probability that both appear into an *item* of class *bad* (assuming independence) is
  $p('python\&casino'|'bad') = 0.8 \cdot 0.2 = 0.16$.
- Therefore, we can define $p(item|category)$ in the following way:

$$p(item|category) = p(feature_1|category) \cdot \ldots \cdot p(feature_1|category)$$

Probability of an *item* given a category

t9 Define and incorporate a subclass of *classifier* called *naivebayes*, and create a member function *docpro* which analyzes the *features* of an *item* and returns the probability of this item given the category *cat*.

```
class naivebayes(classifier):
  def docprob(self,item,cat):

    features=self.getfeatures(item)
    # Multiply the probabilities of all the features together
    p=1
    for f in features:
      ...
    return p
```

Probability of an *item* given a category

- Test the previous program:

```
>>> import docclass
>>> cl = docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.docprob('quick rabbit','good')
quick rabbit
0.26041666666666663
>>> cl.docprob('quick rabbit','bad')
quick rabbit
0.125
```

Bayes' Theorem

- Previously, we have seen how to compute $p(item|category)$. However, we actually need to compute $p(category|item)$
- Bayes' Theorem:

$$p(A|B) = p(B|A) \cdot \frac{p(A)}{p(B)}$$

in our example,

$$p(category|item) = p(item|category) \cdot \frac{p(category)}{p(item)}$$

t10 Do we really need to compute $p(item)$?

Bayes' Theorem: Máximo A Posteriori (MAP)

- This name is given to the most probable hypothesis using Bayes's theorem

$$h_{MAP} = \arg\max_{h \in H} p(h|D)$$

$$= \arg\max_{h \in H} \frac{p(D|h) \cdot p(h)}{p(D)}$$

$$= \arg\max_{h \in H} p(D|h) \cdot p(h)$$

- Sometimes, these hypothesis are equiprobable. In that case, we refer to maximum likelihood:

$$h_{ML} = \arg\max_{h \in H} p(D|h)$$

Lets program ...

t11 Define a function *prob* that computes $p(category|item)$.

```
def prob(self, item, cat):
  catprob = ...
  docprob = ...
  return ...
```

- Compute p('quick rabbit','good') and p('quick rabbit','bad')

```
>>> import docclass
>>> cl = docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.prob('quick rabbit','good')
quick rabbit
0.15624999999999997
>>> cl.prob('quick rabbit','bad')
quick rabbit
0.05000000000000003
```

t12 Which class would you assign to 'quick rabbit'?

How to assign a category to an *item*

- We can choose the highest probability
- In some applications it is better for the classifier to admit it does not know how to classify a document than assigning it to a class due to marginally superior probability. Ex: it is better to do not classify a document as *spam* if it is not clear enough
- In order to avoid this problem we can establish a minimum *threshold*
- In order to classify an *item* in a category its probability has to be at least *threshold* times greater than any other probability

How to assign a category to an *item*

- Ex: In order to filter *spam*, the threshold for 'bad' could be 3, while for 'good' could be 1.

t13 Incorporate in the class *naivebayes*:

```
def __init__(self,getfeatures):
  classifier.__init__(self,getfeatures)
  self.thresholds={}
def setthreshold(self,cat,t):
  self.thresholds[cat]=t

def getthreshold(self,cat):
  if cat not in self.thresholds: return 1.0
  return self.thresholds[cat]
```

How to assign a category to an *item*

t14 Define a function *classify* that classifies an *item* returning a category or a default value when the class can not be determined

```
def classify(self,item,default=None):
  probs={}
  # Find the category with the highest probability
  max = 0.0
  best = None
  for cat in self.categories():
    probs[cat]= ...
    if probs[cat]>max:
      max = ...
      best = ...

  # Make sure the probability exceeds threshold*next best
  for cat in probs:
    if cat==best: continue
    if ...:
      return default
  return best
```

How to assign a category to an *item*

- Test your classifier:

```
>>> import docclass
>>> cl = docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.classify('quick rabbit', default='unknown')
'good'
>>> cl.classify('quick money', default='unknown')
'bad'
>>> cl.setthreshold('bad',3.0)
>>> cl.classify('quick money', default='unknown')
'unknown'
>>> for i in range(10): docclass.sampletrain(cl)
...
>>> cl.classify('quick money', default='unknown')
'bad'
```

Index

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

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

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

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

1. Supervised Learning
   - Decision Trees
   - Bayesian Learning
2. Unsupervised learning
   - Hierarchical clustering
   - K-Means clustering

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.