Bayesian Hierarchical Clustering in Python

Joseph Marion

May 1st, 2015

Introduction

Bayesian Hierarchical Clustering (BHC) is an agglomerative hierarchical clustering method that uses a probabilistic framework. Hierarchical clustering, often used in unsupervised learning, structures data into a binary dendrogram using a greedy bottom-up approach. This dendrogram is meant to reflect the natural hierarchy and organization of the data.

BHC was proposed by Heller and Ghahramani (2005) in their paper of the same name. The algorithm fits a probabilistic model to the data and then uses marginal likelihoods to determine which nodes should be merged. BHC has several advantages over traditional agglomerative clustering. Using a probabilistic model also facilitates natural prediction and tree-pruning. Choosing an appropriate probibalistic model for BHC may be easier than determining which distance metric to use in traditional clustering. Finally, BHC provides an easy measure for the "goodness of fit" of the tree in the marginal likelihood of the tree. This facilitates easy comparison between trees.

Algorithm Overview

For each data point, we model each feature as using a $\text{Bin}(n, p)$ likelihood with a $\text{Beta}(a, b)$ prior on p, the binomial probability. The observations are then modeled as a Dirichlet Process Mixture (DPM) of beta-binomial distributions. The DPM hyper-parameter α informs the concentration of the tree, controlling the prior probability of merging two nodes. BHC uses an approximation to the DPM, considering only the set of tree-consistent partitions of the data rather than all partitions. This can be viewed as a fast, deterministic alternative to an MCMC.

The algorithim is executed as follows. At initialization, each data point is assigned to its own tree. Then, at each step in the algorithm, we consider the probability that two different trees should be merged. The two trees with the highest probability of merging are then merged. This continues until all the data has been clustered into one tree.

Python Implementation

My implementation of BHC can be found in the library *BHC* where algorithm is implemented in *bhc.cluster*. The submodule *bhc_tools* contains useful helper functions such as *drawdendrogram*, which displays the tree, *prune_tree* which truncates the tree based on the probability of merging, and *ALPHA_grad* which can be used to learn the optimal value of α, the DPM hyper-prior. A Cythonized version of the algorithm can be found in *cy_bhc.cluster*.

Numerical Issues

This algorithm is difficult to implement on large data sets due to numerical issues. I used the log scale where possible to manage large/small numbers. However, the additions/subtractions of large factorials necessitated working outside the log scale. One example of this is the prior-probability of merging two nodes. Let π_k be the prior probability of merging nodes i and j. We compute π_k for each merge usign the following formula:

$$\pi_k = \alpha\Gamma(n_k)/d_k$$
$$d_k = \alpha\Gamma(n_k) + d_j * d_i$$

Where d_k is computed in a bottom up fashion and n_k is the total number of observations in the proposed node. The d_k values become large very quickly. I computed a loose lower bound for d_n, the d value of the root node of a tree with n observations/leaves.

$$d_n \geq \sum_{l=1}^{n} \alpha^{n-l}\Gamma(l) > (n-1)!$$

In practice d_n is much larger. The largest value of Γ that can be computed is $\Gamma(172)$ so this value quickly becomes *inf*. This problem is exascerbated by the addition step, which precludes working in the log-scale. I've limited my examples to small data sets where this is not an issue.

Code Optimization

Using a small data set I profiled *bhc.cluster*. As shown below, most of the computing time was spent calling functions that computed the probability of merging two nodes.

```
Thu Mar 10 12:17:27 2016    restats

         2784 function calls in 0.782 seconds
```

```
Ordered by: internal time
List reduced from 24 to 5 due to restriction <5>

ncalls  tottime  percall  cumtime  percall filename:linenofunction
   458    0.745    0.002    0.757    0.002 bhc.py:18h_k
   380    0.012    0.000    0.636    0.002 bhc.py:69r_k
   458    0.012    0.000    0.012    0.000 {sum}
   418    0.009    0.000    0.009    0.000 bhc.py:34p_k
     1    0.003    0.003    0.782    0.782 bhc.py:124cluster
```

Therefor, I re-implemented the 5 functions associated with probability calculations in Cython. These functions can be found in the module *cy_bhc_tools*. This resulted in a roughly 10x increase in computation time

Unfortunately, I had difficulty compiling the .pyx file into a python module. This had something to do with the cython_gsl module. These results can be observed by running *cy_bhc.ipynb*.

Additional gains in speed might be made by re-writing *bhc.cluster*. In particular, deleting merged nodes from the current node list is computationally expensive.

Application

I applied the BHC algorithm to the Spambase data set from the UCI Machine Learning Repository. The data set is composed of 4601 emails that have been classified as spam or not-spam. For each email, there are 57 features that are thought to be associated with whether or not an email is spam. Before clustering, we converted each feature to binary and sub-sampled 20 observations of spam and 20 observations of non-spam.

I set the DPM hyper-prior α to be 0.001. Heller and Ghahramani propose an EM-like algorithm to learn the optimal value of α, which I implemented by alternating calls to *bhc.cluster* and *ALPHA_grad*. This algorithm suggested even smaller values of α were appropriate for the data, however, my tree started to become non-sensical so I chose a suitably small value for the hyper-prior.

For the Beta(a, b) priors I chose a and b so that the expected value of the prior was the sample probability of seeing each feature with a somewhat sparse variance. The dendrogram showing the results of the clustering can be found below.

Leaves 0-19 are all spam; leaves 20-39 are not spam. Pruning the tree by

cutting where the probability of a merge is greater than 0.5 suggests there are four clusters in the data.

Cluster	Spam	Not-Spam
Upper	13	0
Upper Middle	0	9
Lower Middle	3	9
Lower	4	2

Table 1: Spam/Non-Spam Counts by Cluster for BHC

The top two clusters are pure; perfectly separating spam and non-spam. The lower clusters are more of a mixed bag. There is certainly a suggestion that each cluster tends more towards one-type of email than the other. This analysis is certainly add-hoc, but it suggests that the algorithm is performing appropriately.

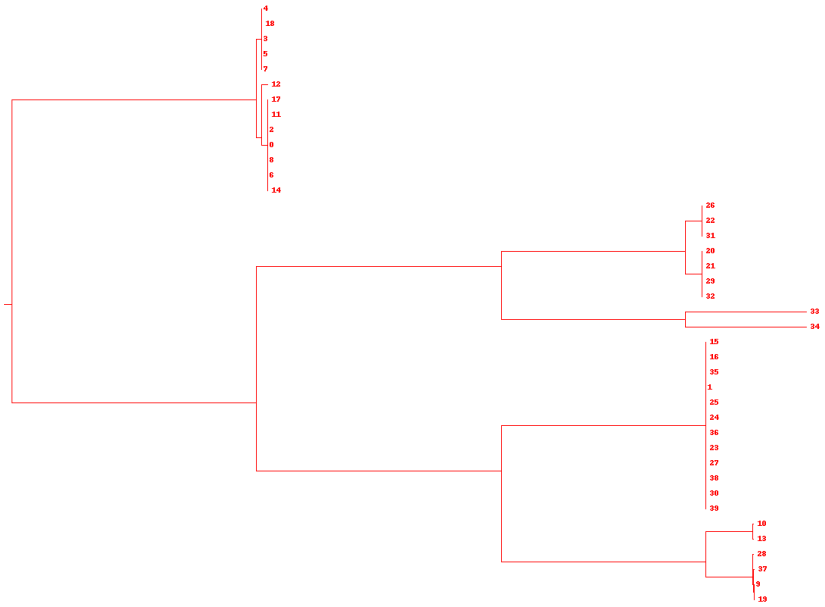

Figure 1: The results of BHC clustering on the spambase data. This figure is quite large because it took forever to create and I am pleased with it.

Comparison to Hierarchical Clustering

I compared BHC to traditional Hierarchical Agglomerative Clustering (HAC) using the same data from spambase. A literature review suggested that complete linkage was the most effective method for clustering binary data.

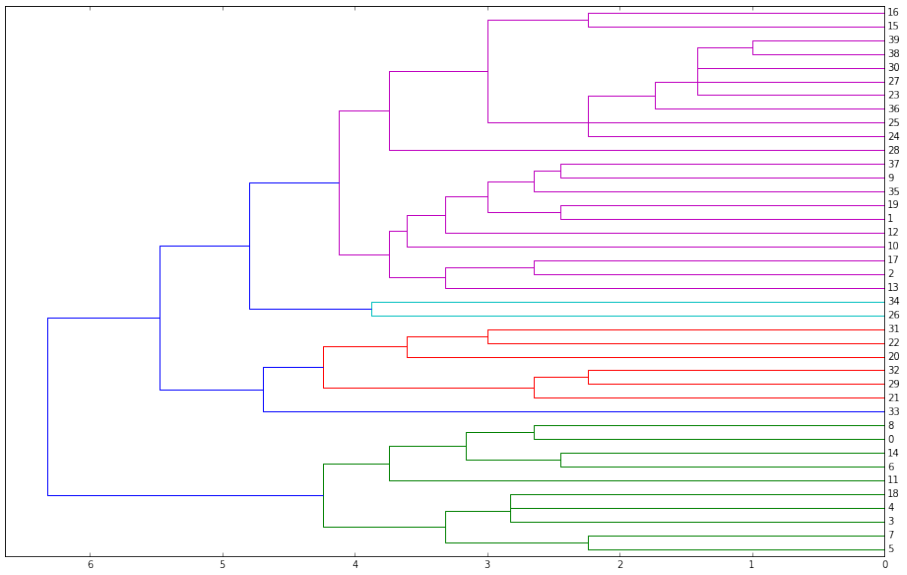

Figure 2: Dendrogram from HAC

Cluster	Spam	Not-Spam
Upper	10	13
Middle	0	7
Lower	10	0

Table 2: Spam/Non-Spam Counts by Cluster for HAC

Using thresholding to prune the tree suggests that there are three clusters in the spambase data The middle and lower clusters have excellent purity and correctly cluster spam and non-spam emails. The middle cluster is quite poor, however, with similar numbers of spam and non-spam emails. In general BHC classifies more spam/non-spam emails into pure trees than HAC. In addition, the impure trees created by BHC tend to be either spam heavy or spam light. In contrast the impure HAC tree seems to have similar numbers of Spam and Non-spam emails. If we classify a cluster as spam/not-spam based on the majority of emails in the cluster, BHC miss-classifies 5 emails whereas HAC miss-classifies 10 emails. This suggests BHC may be more effective at clustering data with binary features.

Conclusion

Bayesian Hierarchical Clustering is an effective method for clustering data with binary features. There are numerical issues in the algorithm which make it difficult to implement for large data sets. I was able to effectively reduce the runtime of the algorithm by 10 times through the use of Cython.