Notes on LDA1PFS

Anthony Di Franco

06/17/2009

Introduction

Latent Dirichlet Allocation (LDA) is a fairly general graphical latent variable model wherein a set of observations is taken to be partitioned into groups. Within each group, there is assumed to exist a distribution over a set of hidden variables. Each hidden variable, in turn, is a distribution over observables. Both the distributions over hidden variables within groups and the hidden-variable distributions over observables are assumed to be Dirichlet-prior-smoothed multinomials.

LDA can be applied to clustering and extrapolating from discrete data as well as matrix factorization. The hidden variables are interpreted as clusters of items with similar associated observables. LDA can be viewed conceptually as a form of principle components analysis with smoothed multinomial components instead of the normal components. It and variations of it have been the subject of much recent research in information retrieval, language modeling, and data assimilation in the statistics and computer science communities.

The explicitly probabilistic formulation of LDA makes it suited to using the very scalable sequential Bayesian inference techniques to fit models to data. Such techniques are distinguished by not necessarily needing to access any single observation more than once.

1PFS is a method for sequential Bayesian inference with this property. It uses a Gibbs sampler, which must make many passes through its input data, on a small, fixed-size subset of data to obtain an initial set of samples for a particle filter, and then uses the particle filter with kernel smoothing to maintain parameter estimates while streaming over the rest of the data. It needs only to store these samples, which are fixed in size by the number of hidden variables assumed in the model and the number of distinct observables, but not the number of observation groups. The particle filter accomodates straightforward parallelization by splitting the storage and maintenance of the sample set. The Gibbs sampler can also be parallelized, especially if certain approximations are acceptable, if needed.

0.1 Implementation

I have applied the 1PFS inference technique described in S. Balakrishnan, D. Madigan, "A one-pass sequential Monte Carlo method for Bayesian analysis of massive datasets," 2004 to the LDA model described in D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet allocation," 2003.

The implementation is in Java and uses CERN's COLT library for matrix data structures and linear algebra algorithms.

0.1.1 Interface

void initialize(int Z, int gibbsSize, int burnin, int samplesize)

$Z$is the number of hidden variables to use in the model. *gibbsSize* is the number of rows to incorporate in the Gibbs-sampled model. *burnin* is the number of Gibbs sampler iterations to perform before collecting samples, and *samplesize* is the number of samples to collect from the Gibbs sampler after that.

void addRow(Object rowID, Map<Object, Integer> row)

Accumulates rows for the Gibbs sampler if *gibbsSize* rows have not yet been seen, performing Gibbs sampling upon receipt of the *gibbsSize*th row; updates the Φ estimate via the particle filter otherwise. If a row is given that has already been seen according to *rowID*, removes the row and resamples and readds it according to its contents as given by *row*.

void removeRow(Map<Object, Integer> row)

Samples hidden variable values for the given row and modifies state to imitate the absence of that row from the model's data set. Generally intended to be a private method to be called in updating rows, which is why it lacks a *rowID* parameter.

void updatePhiEstimateByRow(Map<Object, Integer> row, int miniburnin, int minisamplesize)

Estimates Θ for the row with a mini Gibbs sampling of just the row's parameters w.r.t. those of the whole model, and uses this to update the entire model's parameters. This contains most of the 1PFS machinery.

double[][] currentPhiEstimate()

Just so. Useful for diagnostic purposes or visualization of the clustering in the model.

double[] estimateThetaOfRow(Map<Object, Integer> row, int miniburnin, int minisamplesize)

Just so. Useful for diagnostic purposes or visualization of the clustering in the model.

double[] predictForRow(Map<Object, Integer> row, int miniburnin, int minisamplesize)

Returns predicted probability of a new observation associated with the given *row* falling into each of the possible observables represented by the columns in the model. Used to predict votes.

Map<Object, Integer> getColumnCodes()

Used to associate column indices with the objects they were assigned to.

void saveState(String filename)

Saves state to the file named by *filename* relative to the process' working directory.

void loadState(String filename)

Replaces the current model with state loaded from the file named by *filename* relative to the process' working directory using a static method of the LDA1PFS class.

0.1.2 Runtime characteristics

Currently only adding a row and obtaining predictions and parameter estimates are supported, and these will likely necessarily be the common cases even when deleting / updating are implemented, since frequent use of deleting will put stress on the approximation used to delete rows.

The algorithm will store maps encoding the row and column objects to their corresponding row and column indices, as well as the entire encoded matrix up to the *gibbsSize*th row and counts of assignments of rows to hidden variables and hidden variables to columns. This storage is bounded from above by $NZ + ZF + NF$ where N is *gibbsSize*, Z is the number of hidden variables, and F is the number of columns (twice the number of target users) but counts are stored sparsely. Additionally, *samplesize* samples of size ZF are collected for subsequent use in the particle filter.

No additional storage is used by adding a row after the Gibbs sampler has run, except in that adding a row may make the contingency tables representing parameter estimates less sparse.

0.1.3 Implementation

The server object wraps an instance of the LDA1PFS class so as to be able to save and load state locally across an RPC barrier. The inner instance of LDA1PFS implements the functionality of the interface described above, relying on a Gibbs sampling module to perform the main run of Gibbs sampling (the Gibbs sampling statistical technique is used elsewhere in the code, but in an ad-hoc way that refers to the parameter estimates in the 1PFS instance). Some utility classes provide data display, random number generation, and type munging.

Notable departures from ideals:

- The count matrices should be stored as integer matrices, but COLT does not support this well at all, so I used double matrices.

Algorithm 0.1 usage example

```
System.out.println("small matrix scaled by 20");
LDA1PFS lm = new LDA1PFS(2, 20, 100, 20);
Map evenrow = new HashMap();
Map oddrow = new HashMap();

evenrow.put(0, 20); evenrow.put(1, 0);
evenrow.put(2, 20); evenrow.put(3, 0);
oddrow.put(0, 0); oddrow.put(1, 20);
oddrow.put(2, 0); oddrow.put(3, 20);

for(Integer i = 0; i < 100; i++) {
 if (i % 2 == 0) { lm.addRow(i, evenrow);}
 else { lm.addRow(i, oddrow); } }

double[][] p = lm.currentPhiEstimate();

double[] q = lm.predictForRow(row, 60, 5);
```

- This version is entirely serial.

0.1.4 Example