Manual for the JDageem Package (version 1.01)

Shay Cohen

August 19, 2014

1 Introduction

The JDageem Java package includes several implementations of parsing and training algorithms for dependency grammar induction. More specifically, JDageem includes:

- An implementation of the split head automaton parsing algorithm of Eisner and Satta (1999), also specialized to handle the dependency model with valence of Klein and Manning (2004). Both the Viterbi and the mininum Bayes risk decoding (Goodman, 1996) versions are implemented.
- An implementation of the expectation-maximization algorithm for the DMV.
- An implementation of the variational expectation-maximization algorithm of Cohen and Smith (2010), which makes use of the logistic normal priors (for the DMV).
- An implementation of a variant of the harmonic DMV initializer that appears in Klein and Manning (2004)

The JDageem Java package is distributed freely under the GNU license. If you make use of the JDageem package, please cite:

Cohen, S. B. and Smith, N. A. (2010). Covariance in unsupervised learning of probabilistic grammars. *Journal of Machine Learning Research*, 11:3017–3051.

or:

Cohen, S. B. and Smith, N. A. (2009). Shared logistic normal distributions for soft parameter tying in unsupervised grammar induction. In *Proceedings of HLT-NAACL.*

2 Java Dependencies

JDageem makes use of the following freely-distributed Java packages:

- Colt, a set of open source libraries for high performance scientific and technical computing in Java (`http://acs.lbl.gov/software/colt/`).
- Apache commons, a collection of open source reusable Java components from the Apache/Jakarta community (`http://commons.apache.org/`).

- Trove, a library that provides high speed regular and primitive collections for Java (`http://trove.starlight-systems.com/`).

All of the necessary JAR files are provided in the lib/ directory in this package.

3 File Formats

This section describes file formats used by JDageem.

3.1 Corpus file formats

JDageem supports one file format to represent sentences and dependency trees.

format name	sentence only representation	dependency tree representation
plain	a line per sentence, tokens separated by space	a line per sentence (tokens separated by space) followed by a line for the sentence dependency tree (a list of numbers the length of the sentence, separated by space, each number represents the parent of the corresponding word in the sentence, indices start at 1, 0 denotes root), followed by a blank line

Examples for the plain format (both for sentence only representation and for dependency tree representation) are included in the contrib/ directory.

3.2 Model file format

A model file in JDageem has the following format. Each parameter in the dependency model with valence is specified in a single line. Such a line can be either one of the following:

- A blank line or a line that starts with // - these lines are ignored.
- root [pos] : value - root probability for part-of-speech tag pos. value should be in log-domain.
- leftattach [pos-child] <- [pos-parent] : value - dependency probability for an edge pos-parent $\rightarrow$ pos-child oriented towards the left. value should be in the log-domain.
- rightattach [pos-child] -> [pos-parent] : value - dependency probability for an edge pos-parent $\rightarrow$ pos-child oriented towards the right. value should be in the log-domain.
- leftcontinue [pos-head] nochild : value - continue probability before generating the first child on the left when the head is pos-head. value should be in the log-domain.
- leftcontinue [pos-head] haschild : value - continue probability after generating the first child on the left when the head is pos-head. value should be in the log-domain.

- leftstop [pos-head] nochild : value - stop probability before generating the first child on the left when the head is pos-head. value should be in the log-domain.
- leftstop [pos-head] haschild : value - stop probability after generating the first child on the left when the head is pos-head. value should be in the log-domain.
- rightcontinue [pos-head] nochild : value - continue probability before generating the first child on the right when the head is pos-head. value should be in the log-domain.
- rightcontinue [pos-head] haschild : value - continue probability after generating the first child on the right when the head is pos-head. value should be in the log-domain.
- rightstop [pos-head] nochild : value - stop probability before generating the first child on the right when the head is pos-head. value should be in the log-domain.
- rightstop [pos-head] haschild : value - stop probability after generating the first child on the right when the head is pos-head. value should be in the log-domain.

Note that the use of the word "probability" here is inaccurate. In general, a model file can be specified with any weight for all of the parameters. As a matter of fact, the output of the logistic normal variational EM algorithm is a weighted dependency model with valence, and not a probabilistic one (i.e. weights could sum to value other than 1).

An example for a model file is included in the contrib/ directory. This file is the harmonic initializer one gets from the Penn Treebank training set.

4 Running JDageem

In order to run JDageem, you need to run the class "edu.cmu.cs.lti.ark.dageem.Dageem" which resides in JDageem.jar. You can use the shortcut "dageem.sh" which resides in the scripts/ directory.

```
# scripts/dageem.sh

usage: JDageem [-em] [-emdropiter <arg>] [-emiter <arg>] [-evalinput <arg>]
       [-evalinputformat <arg>] [-initmodel <arg>] [-lnem] [-lnemdropiter <arg>]
       [-lnemiter <arg>] [-lninitmodel <arg>] [-lnmemorysave]
       [-lnmodeloutputprefix <arg>] [-lntraininput <arg>] [-lntraininputformat
       <arg>] [-modeloutputprefix <arg>] [-parse] [-parseralg <arg>]
       [-parserinput <arg>] [-parserinputformat <arg>] [-parsermodel <arg>]
       [-parseroutput <arg>] [-parseroutputformat <arg>] [-traininput <arg>]
       [-traininputformat <arg>]
```

Dageem can run in three modes: -em, -lnem or -parse, which correspond to vanilla EM, logistic normal variational EM or parsing, respectively. For EM, the following parameters are available:

- -emdropiter N - the number of iterations between outputting the model file. For example, for N=5, a model file will be written after iteration 1, iteration 6, ..., etc.
- -emiter N - the number of iterations to run EM.

- -initmodel file - a model to initialize EM with. Use @harmonic@ here, if you are interested in using a variant of the harmonic initializer which is described in Klein and Manning (2004). Use @dirichlet=A@ for some $A > 0$, if you are interested in initializing each multinomial in the DMV model with a symmetric Dirichlet draw with parameter A. For example, @dirichlet=1.0@ will initialize all multinomials uniformly. See Section 3 for a description of the model file format.
- -modeloutputprefix prefix - the prefix for the model output files. Dageem uses file names of the form prefixI.gra, where I is the iteration number.
- -traininput - The input file name for training.
- -traininputformat format - File format that the input file uses. Currently only "plain" is supported. See Section 3.

For logistic normal variational EM, the following parameters are available:

- -lnemdropiter N - number of iterations between outputting the model file.
- -lnemiter N - number of iterations to run the variational EM algorithm.
- -lninitmodel file - a model to initialize variational EM with. Can also use @harmonic@, see above.
- -lnmodeloutputprefix prefix - the prefix for the model output files.
- -lntraininput - The input file name for training.
- -lntraininputformat format - File format that the input file uses. Currently only "plain" is supported.
- -lnmemorysave - A flag that tells to run the variational EM algorithm with less memory, but slightly slower.

For parsing, the following parameters are available:

- -evalinput input - file to evaluate the parser's output again (optional).
- -evalinputformat format - format in which the evaluation file is in. Currently only "plain" is supported. See Section 3.
- -parseralg alg - algorithm to use for parsing. Can either be "viterbi" or "mbr".
- -parserinput input - file to parse (should only include sentences).
- -parserinputformat format - format in which the parser's input is in. Currently only "plain" is supported.
- -parsermodel file - model file to use for parsing.
- -parseroutput file - file to write the output of the parser in.
- -parseroutputformat format - format to use for outputting the parsed sentences. Currently only "plain" is supported.

4.1 Memory Issues

You may run into trouble running JDageem when using a large corpora (either with many sentences or with many part-of-speech tags). There are two ways to try and overcome this:

- Increase the amount of memory used by the Java interpreter, either in the jdageem.sh script or by using directly the -Xmx parameter passed to the Java interpreter.
- Run JDageem with the -lnmemorysave flag, which makes JDageem sacrifice the accuracy of the variational inference procedure for better use of memory.

5 How to Extend JDageem

JDageem provides support for the dependency model with valence only. In principle, there is nothing that prevents one from using the logistic normal prior with other multinomial models. Say we have a model $p(x, z \mid \theta)$ where z is a latent structure, x is observed and θ is a collection of multinomials, then one could imagine a model that looks like:

$$\begin{aligned} \theta &\sim \text{LogisticNormal}(\mu, \Sigma) \\ (x, z) &\sim p(x, z \mid \theta) \\ x &= \textit{first coordinate of } (x, z) \end{aligned}$$

In order to extend the use of the logistic normal variational EM with other models (either for dependency grammar induction, or generally any model which has a multinomial structure), the following needs to be done:

- A new class needs to be created, which extends the abstract class "LogisticNormalInsideOutsideInterface". Details are below. Call this class "NewModelIO."
- A new class needs to be created, which extends the abstract class "LogisticNormalVariationalEM" and overrides the "saveModel" method. Details are below. Call this class "NewModelVarEM."
- (optional) A new class needs to be created, which extends the abstract class "Corpus." Call this class "NewCorpus."
- (optional) If a new Corpus class was created, then it might be necessary to create a new class which extends the "Document" class.
- The following code needs to be executed:

```
Alphabet alphabet = new Alphabet();
Corpus corpus = new SentenceCorpus(alphabet);

corpus.read("training.txt");

Chart.Semiring.setSemiringLogReal();

```

```
LogisticNormalInsideOutside io = new NewModelIO(dmvGrammar);
LogisticNormalModel model = io.createModel();
LogisticNormalVariationalEM varEM = new NewModelVarEM(...);
varEM.run(100, 5, "/tmp/model", corpus, io);
```

where "training.txt" is the training data file name (a list of sentences), 100 is the number of iterations to run the variational EM algorithm, 5 is the number of iterations between writing the models, and "/tmp/model" is the prefix for the model output files. If a new corpus class was created, then the first four lines in the listing above need to be replaced with:

```
Corpus corpus = new NewCorpus(...);
// read the corpus
```

5.1 Creating "NewModelIO"

The "NewModelIO" class needs to extend "LogisticNormalInsideOutsideInterface" and implement the following methods:

```
public abstract class LogisticNormalInsideOutsideInterface
        extends InsideOutsideInterface {

    public abstract LogisticNormalModel createModel();
    public abstract LogisticNormalVariationalInference
            createVariationalInferenceObject(LogisticNormalModel model,
                                             Document doc);
    public abstract void getInsideOutside(Document doc,
                                          LogisticNormalVariationalInference varParams);
    public abstract double getInitLogPhi(LogisticNormalModel model,
                                         int i, int j);
    public abstract boolean multinomialParticipates(Document doc, int i);
}
```

We begin with "createModel." This function creates a LogisticNormalModel object. This object contains values for μ and Σ, the hyperparameters for the multinomial collection describing the model. In order to create it, we need to provide an array for the sizes of each multinomial in the family. So, for example, a possible implementation of "createModel" could be:

```
public LogisticNormalModel createModel()
{
    int[] sizes;

    // set the sizes array to
    // the list of multinomial sizes
    // in the model

    return new LogisticNormalModel(sizes);
}
```

Next, we turn to "createVariationalInferenceObject." This function creates a LogisticNormalVariationalInference object. This object contains values for the variational parameters for a given document in the corpus. An example for an implementation of this method would be:

```
public LogisticNormalVariationalInference createVariationalInferenceObject(
    LogisticNormalModel model, Document doc) {

    LogisticNormalVariationalInference ln =
        new LogisticNormalVariationalInference(model,
                                this,
                                doc);

    return ln;
}
```

Note that after creating the "ln" object, one could initialize the variational paramaeters to some values, the way is desired. Look into the "LogisticNormalVariationalInference" class to get more details about this.

Next, we turn to the get "getInsideOutside" method. This method takes a Document object and a LogisticNormalVariationalInference object, and sets all the counts in the latter object to the feature expectations of the multinomial events according to Document. Note that these feature expectations should be computed according to the "phi" weights that appear in the LogisticNormalVariationalInference object. Once the feature expectations are computed, then they need to be copied to the "count" weights that appear again in the LogisticNormalVariationalInference object which is passed as a parameter to the getInsideOutside method.

To see an actual example of how this works, consider the getInsideOutside method that appears in the LogisticNormalDMVInsideOutside class. This specific example uses the agenda algorithm for computing the inside-outside probabilities, but one can imagine using, for example, memoization or any other technique to do that. It is completely up to the user. As a matter of fact, one can imagine using other algorithms than the inside-outside to get such feature expectations. This means that the use of the name "InsideOutside" in the respective classes and the getInsideOutside might be a misnomer.

The "getInitLogPhi" returns an initialization value for the event j of multinomial i. This initialization determines which values to use when we first compute the feature expectations.

Last, we have the "multinomialParticipates" method. It is there only for optimization, and a correct implementation of it could be:

```
public boolean multinomialParticipates(Document doc, int i)
{
    return true;
}
```

If this method returns false, then for that specific Document, the multinomial i will not be run through the variational inference algorithm. This is useful, for eaxmple, with the DMV model. We know that only multinomials that condition on a part-of-speech tag that appears in a given sentence influence variational EM. Therefore, we can (almost) skip all multinomials for part-of-speech tags that do not appear in a given sentence, instead of running an expensive optimization routine for them.

5.2 Creating "NewModelVarEM"

In order to create "NewModelVarEM," we just need to provide a method that writes a model file.

```
public class NewModelVarEM
        extends LogisticNormalVariationalEM {

    public void saveModel(String outputFile) {
        // save the "model" object
        // to a file by using getModel()
    }

}
```

Admittedly, it is not immediate to extend JDageem to use other multinomial models. If you are interested in doing that, but still have difficulties understanding how to do it, do not hesitate to contact the author at scohen@inf.ed.ac.uk.

6 Contacting the Author

For bug reports, questions and comments, please write to scohen@inf.ed.ac.uk.

References

Cohen, S. B. and Smith, N. A. (2009). Shared logistic normal distributions for soft parameter tying in unsupervised grammar induction. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 74–82. Association for Computational Linguistics.

Cohen, S. B. and Smith, N. A. (2010). Covariance in unsupervised learning of probabilistic grammars. *The Journal of Machine Learning Research*, 11:3017–3051.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464. Association for Computational Linguistics.

Goodman, J. (1996). Parsing algorithms and metrics. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 177–183. Association for Computational Linguistics.

Klein, D. and Manning, C. D. (2004). Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 478. Association for Computational Linguistics.