Implementation of Hidden Markov Model to Weka

Alexandre Savard
Schulich School of Music - McGill University
555 Sherbrooke St. West
Montreal, QC Canada H3A 1E3

I. Introduction

THE goal of this project is to include an Hidden Markov Model to Weka, an open source data mining and machine learning algorithms software. Version 3.4.7 of Weka has been explored. A Java HMM library written by Jean-Marc Franois has been found. Version 0.5.0 will be discussed here. It presents a very important non-compatible feature. NetBeans IDE 5.0 has been used for compilation.

II. Weka

A. Architecture of Weka

Weka can be used at different levels. It provides command-line and graphical interfaces that make easier comparison between different classifiers. However, what makes it so powerful is that it can be easily adapted and used in our own Java code. Weka source code is divided in three main packages : the core, the classifiers, and the filters.

The core is composed of all classes that allow the user to manipulate data and files. One of the most important classes is the class Attribute that contains numeric, nominal, or string description of the data. The class Instance stores data values related to one element as floating points. It represents a line in the ARFF file format while the header of the file gives informations for the attributes. A third important class is Instances, this last one hold a set of instances.

The classifier package contains actually all the machines learning algorithm. Each of them needs to implement two required method : the *buildClassier()* method and the *classifyInstance()* method. The *buildClassier()* method is called to generate the classifier while the *classifyInstance()* method processes data to be categorized. The first one takes, as an argument, a set of trainning instences. The second one, needs a set of instance to classify. For HMM, the *buildClassier()* method should contain all methods that, in a first step, preprocess the data in order to make them compatible to sequential observation. The second step should be the initialization of the HMM using the Baulm-Welch learning algorithm.

The *classifyInstance()* method is the one that needs to perform the Viterbi algorithm and to calculate the probability of an observation sequence in order to get useful information to classify the input instance given a trainned HMM. The classification process using HMM needs as many models as there is class. Each model is trainned with Then, hopefully, the two previous become characteristic features for the class.

B. The ARFF file format

Weka is optimized to handle non-sequential data sets. Integration of Hidden Markov Model to Weka is problematic due to this constraint. Considering the ARFF file format, since that for each columns is specified a particular attribute, we are restricted to use a limited number of column. Otherwise the header of the file becomes quickly very extended and hard to manipulate.

It has been chosen that the data should be written on two columns using this file format. The first column represents the observation while the second column is a flag The file is read from up to down. See /DynamicHmm/Dist/obsrvSet.arff for an exemple.

III. Jahmm

Jahmm is a library written in the Java language that implements HMM related lagorithims. Its primary feature is probably the fact that it can be applied on many different types of observation. It has been meant to read and write simple file within the *jUnit* environment containing only observation separated by semi-colons.

The main class is obviously *Hmm* class that contains all the information about a given HMM: transition probability matrix, initial probability distribution, and observation probability distribution for each state. The class *Observation* is the building block of the library and is mostly extended by all the other classes. Both the *Hmm* and the *BaulmWelchLearner* are instantiated by specifying a particular *Observation* classe and its appropriate *OpdfFactory*. The letters *Opdf* stand for observation probability distribution function. These two classes are used internally to specify to calculators what datas are going to be handled.

The *BaulmWelchLearner* class implements the learning algorithm needs to be instantiated while specifying a given set of sequences of observation in argument. Then the learning process is applied by iteration returning a new updated HMM given the previous one.

IV. HMM fundamental problems

A. Generic Programming

BaulmWelchLearner is generic, as well as the whole library and needs to be instantiated specifying the datatypes to be handled. The Constructor needs as a second argument a *List()*. *List()* is an interface That handles collection. In Java, generics type correctness are checked at compile time. An important drawback is that the generic type information is not known at runtime. So, *List<ObservatioDiscrete>* and a *List()*Listt<ObservationVectort> are seen as the same class: *List()*. It is not possible to create an array of generic type since it is impossible to evaluate its type. Then, once created, an observation sequence cannot be modified. Errors may occur when instantiate the Learner with a different type than the one expected.

To be used in an environment such as Weka, the Library needs to be modified. "DynamicHmm" is a try to make the BaulmWelchLearner dynamic replacing all Listt<Listt<Observation>> by ArrayList. ArrayList is a direct extension of the List interface.

B. Solution

To be used in an environment such as Weka, the Library needs to be modified. "DynamicHmm" is a try to make the BaulmWelchLearner dynamic replacing all *Listt<Listt<Observation>>* by *ArrayList()*. *ArrayList()* extends *List()* interface so that a sequence of observation can be dynamically modified. *ArrayList()* can contain any type (even boaleen) and type is checked at runtime.

V. Conclusion

Version 0.6.1 of Jahmm that presents a non-generic Baum-Welch Learner have been released in april. This version presents a lot of change compared to version 0.5.0 and version a first version 0.6.0 was not compatible with Java 1.5.0 on mac OSX. It is for this reason that Jahmm version 0.5.0 have been prefered first. Next try should be using this later version of Jahmm. Hopefully a new library that can be include in Weka won't have to be written.

References

[1] Baum L. and T. Petrie. 1966. Statistical inference for probabilistic functions of finite state Markov chains. *The Annals of Mathematical Statistics*. 37. 1554–63.

[2] Rabiner, L. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77. 257–85.

[3] Witten, I. and E. Frank. 2000. *Practical machine mearning tools and techniques with Java implementations.* San Francisco: Morgan Kaufmann. 265–320.