Learning graphical models

Hidden Markov models

Dario Vogogna

Learning graphical models

Hidden Markov models

Index

1.Introduction

A hidden Markov Model (HMM) is a statistical Markov model in which the system being modelled is assumed to be a Markov process with unobserved (*hidden*) states [1]. The state is not directly visible, but the output dependant on the state, is visible. Each state has a probability distribution over the possible output tokens. Therefore, the sequence of tokens generated by an HMM gives some information about the sequence of states. The parameters of the model are known, it's only the state sequence that is hidden. Hidden variables are usually referred as *latent* variables.

An HMM is a special case of a dynamic Bayesian network, a Bayesian network which relates variables to each other over adjacent time steps. It is often called *Two-Timeslice BN* because at any point in time T, the value of a variable can be calculated from the internal regressors and the immediate prior value at time T-1 [2]. At time 0, the Bayesian network is defined only by its values.

An example of an HMM is shown in Figure 1.

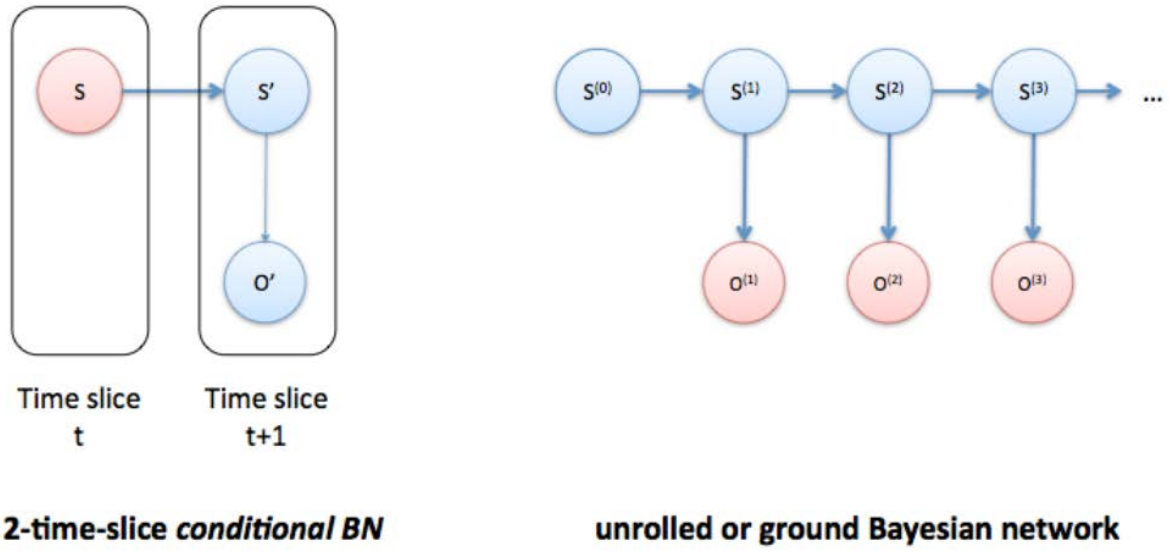

Figure 1 - Example of Hidden Markov model

An HMM is completely represented given a transition matrix A, an observation matrix B and the initial state probabilities π.

S	a	b
a	0.1	0.9
b	0.3	0.7

Table 1 - A=P(S'|S)

S'	B	R
a	0.25	0.75
b	0.67	0.33

Table 2 - B=P(O'|S')

a	b
0.5	0.5

Table 3 - π=P(S|t=0)

The triplet $\lambda =< A, B, \pi >$ fully define an HMM, where A is a distribution governing the probability of transition from any state value q_i to another state value q_j, B is a distribution governing the probability of observing symbol v_k in every state value q_i and π is a distribution specifying for every state value q_i the probability of being in the initial state.

There are some particular cases:

- Factorial Hidden Markov Model, the observation depends upon the value of all states variables in parallel state chains
- Paired Hidden Markov Model, transitions in one HMM condition the one in another, and vice-versa

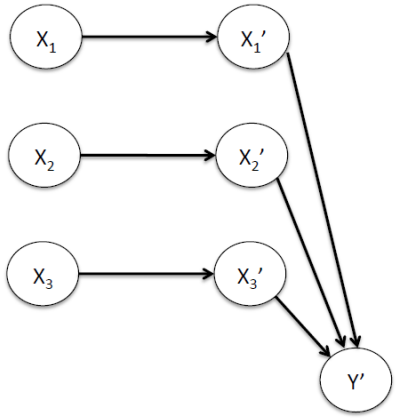

Figure 2 - Factorial Hidden Markov Network

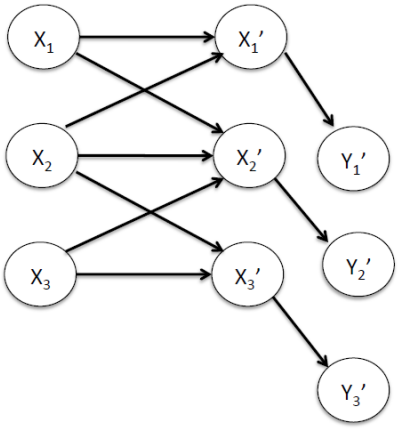

Figure 3 - Paired Hidden Markov Network

Hidden Markov Models allow, among other things, to infer the most likely sequence of states that produced a given output sequence, to infer which will be the more likely next state (and thus predicting the next output) and to calculate the probability that a given sequence of outputs originated from the system (allowing the use of HMM for sequence classification).

There are three problems related to HMM:

1. Given an observation sequence $O = \{o1, o2, \dots, oT\}$, compute the probability $P(O|\lambda)$ which it has been produced by $\lambda =< A, B, \pi >$
2. Given an observation sequence $O = \{o1, o2, \dots, oT\}$ and a model λ, compute the most likely sequence of states in λ which produce O
3. Given an observation sequence $O = \{o1, o2, \dots, oT\}$ and a model λ, estimate A, B and π from λ in order to maximize $P(O|\lambda)$

Forward-backward algorithm

The first problem could be solved with a brute force approach, testing all the possible sequences of states and comparing them, with a complexity $O(N^T T)$, where N is the number of state values and T the number of observations. This is usually unfeasible in practice.

The **forward-backward algorithm** makes use of the principle of dynamic programming to compute efficiently the values that are required to obtain the posterior marginal distributions in two passes: the first goes forward in time while the second goes backward. The first pass computes the probability to end up in any possible state at each time step given the previous observations in the sequence. In the second pass, the algorithm computes a set of backward probabilities which provide the probability of observing the remaining observations given any previous starting point. The two

passes get combined to obtain the distribution over states at any specific point in time, given the entire observation sequence, so the most likely state for any point in time.

$$P(X_k|o_{1:t}) = P(X_k|o_{1:k}, o_{k+1:t})$$

and applying the Bayes' rule and the conditional independence of $o_{1:k}$ and $o_{k+1:t}$ given X_k

$$P(X_k|o_{1:k}, o_{k+1:t}) = P(o_{k+1:t}|X_k) * P(X_k|o_{1:k})$$

Both from the forward and the backward algorithm, $P(O|\lambda)$ can be computed with a complexity of $O(N^2T)$ as $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$ or $P(O|\lambda) = \sum_{i=1}^{N} \pi_i b_i(o_1)\beta_1(i)$ respectively.

Viterbi algorithm

Even if the forward-backward algorithm computes the most likely state at each time step, the most likely sequence of hidden states has to be computed using another dynamic programming algorithm, the **Viterbi algorithm**, that returns the so called Viterbi path.

The base step is defined as the likelihood score of the most likely sequence of hidden states ending in state i and the first t observations.

$$\delta_t(i) = \max_{X_{0:t-1}} p(X_{0:t-1}, X_t = i, o_{1:t}\,|A, B, \pi)$$

The induction step on t gives:

$$\delta_{t+1}(j) = [\max_i \delta_t(i)\, a_{ij}]b_{jo_{t+1}}$$

The actual state sequence is retrieved by backtracking the transitions that maximize the δ scores for each t and j.

The time complexity of this algorithm is $O(N^2T)$.

Baum-Welch algorithm

The third is the most difficult problem of HMM: to adjust the model parameters A, B and π to maximise the probability of the observation sequence given the model. The Baum-Welch algorithm uses an iterative procedure to locally maximize $P(O|\lambda)$. The procedure of re-estimation is composed by iterative update and improvement of HMM parameters [3].

This algorithm uses the variable $\xi_t(i,j)$ as the probability of being in state S_i at time t and S_j at time $t+1$ given the model and the observation sequence (represented in Figure 4)

$$\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j|O, \lambda)$$

and the variable $\gamma_t(i)$ as the probability of being in state S_i at time t, also given the model and the observation sequence

$$\gamma_t(i) = P(q_t = S_i|O, \lambda)$$

Using the forward and backward variables seen in the Forward-backward algorithm we can write $\xi_t(i,j)$ as

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}$$

and $\gamma_t(i)$ as

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}$$

Also we can relate $\gamma_t(i)$ to $\xi_t(i,j)$ by summing over j:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

If we sum $\gamma_t(i)$ over the time index t, we get a quantity which can be interpreted as the expected (over time) number of times that state S_i is visited, or equivalently, the expected number of transitions made from state S_i (excluding time slot $t = T$ from summation). Similarly, summation of $\xi_t(i,j)$ over t can be interpreted as the expected number of transitions from state S_i to state S_j. So, a set of reasonable re-estimation formulas for π, A and B are:

$$\bar{\pi}_i = \text{expected frequency in state } S_i \text{ at time } (t = 1) = \gamma_1(i)$$

$$\bar{a}_{ij} = \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } S_i} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\bar{b}_{j(k)} = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} = \frac{\sum_{\substack{t=1 \\ s.t.O_t = v_k}}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

Iterating this procedure using the newly obtained $\bar{\lambda} =< \bar{A}, \bar{B}, \bar{\pi} >$ we can improve the probability of O being observed from the model until some limiting point is reached.

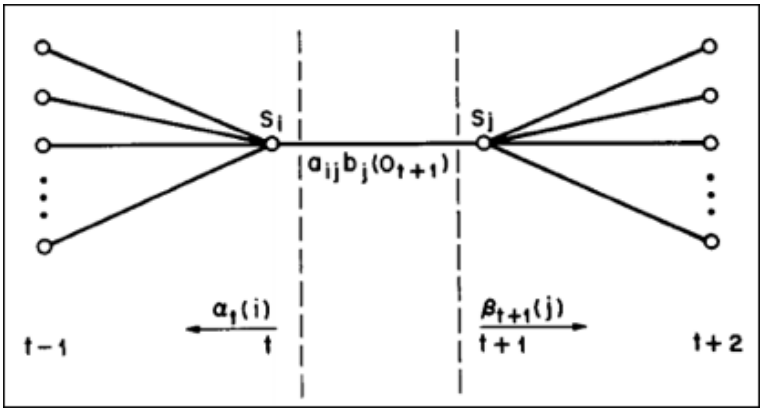

Figure 4 - Illustration of the sequence of operations required for the computation of the joint event that the system is in state S_i at time t and state S_j at time $t + 1$.

The Baum-Welch algorithm can be interpreted as an implementation of the EM (expectation-maximization) algorithm of statistics.

Since the probability computed become really small when there are lots of states or the observation sequence is really long, HMM implementations usually have underflow problems. Some kind of rescaling is usually applied, using logarithms or rescaling factors.

2.Survey about tools available

There are a lot of different tools available online for HMM study, written in different programming languages.

The one used to compare results is written in C++ and is able to compute the algorithms shown in the previous chapter: it's possible to compute the most probable sequence using the *vit* executable and giving in input the HMM name (transition and emission matrix must have the same name and extension .trans and .emit respectively) and the observation sequence, to train a model using *trainhmm* and in input the HMM, the resulting file and the observations used for training, and to generate a collection of observation sequences using the *genseq* executable. The program is implemented using the class *TwoDTable* to represent the transition and emission matrixes meaning that there's no difference in space complexity for sparse and dense matrixes. All the implemented algorithm methods iterate on a *TimeSlot* object (one for each time step) where the result for each state are stored. This implementation uses logarithms to avoid underflow problems. There is no graphical interface, the program can only be used from the terminal.

Another project I personally used as a base is written in C# and is available on CodeProject [4]. It is really well written and easy to understand all the functionalities. It implements, as the previous one, all the algorithms seen, but the sample application is developed for training models only. It uses factor scaling and implements the matrixes as 2D arrays. It offers a nice graphic interface and it is possible to import the model from an .xls file. The program can easily be extended just building the graphical interface to manage the wanted actions. The *HiddenMarkovModel.cs* file contains all the algorithms and could be extended to be used with a graphical interface. It also distinguish between *Ergodic* and *Forward* HMM type.

A Python implementation of HMM is *hmmlearn* [5]. It is built on *scikit-learn* APIs, *NumPy*, *SciPy* and *matplotlib*. It offers a lot more customization than the others, allowing to set the model emissions type (Gaussian, Gaussian mixture, multinomial discrete or a custom one) and saving/loading models to/from file with .pkl extension. I didn't personally try it, but it seemed quite easy to understand looking at the examples on the website. It received a major updated on the 1st of March.

A C implementation is the GHMM Library [6]. It implementing efficient data structures and algorithms for basic and extended HMMs with discrete and continuous emissions and comes with Python wrapper which provide a much nicer interface and added functionality. It also offers HMMEd (Hidden Markov Model editor), a graphical application which allows to create and edit Hidden Markov Models.

A MATLAB implementation is SHMM (Simple HMM) [7], allowing the computation of forward/backward algorithms as well as the Viterbi path. It does not offer the Baum-Welch algorithm. The web page also offers a good description of the algorithms used and some easy examples.

A really simple Java implementation implementing only the training algorithm can be found at [8]. It does not offer scaling, and matrixes are implemented as array of array, but it could be used as a good starting point.

As stated before there are a lot more implementation on internet, most of them offering only the train method as the last one. I didn't find any that uses sparse matrixes for transitions and emissions.

3.Implementation

I decided to implement my project using C#, mostly because it would have been really easy to build a graphical interface. Furthermore, Visual Studio offers a nice environment to manage graphical components and to debug the code. I used 2 external packages: one implementing *Sparse2DMatrix* and one to generate random numbers. For scaling, I decided to use scaling factors instead of logarithms. I followed the Rabiner tutorial [3] and the notes exposed by Rahimi in his tutorial [9] for scaling and multiple observations.

The core of the project is implemented in the class *HiddenMarkovModel.cs*. The code is divided in *regions*, allowing an easier reading.

The transition and emission probabilities are encoded in an extension of the *Sparse2DMatrix*: instead of using $N \times M$ space even if the input matrixes are sparse, the *Sparse2DMatrix* class uses a dictionary to store values, retrieving and setting data using a getter and a setter respectively. It's also possible to use array notation, but it's only syntactic sugar. A value of 0 is returned if the requested index doesn't exist. The class would also allow the use of different kind of indexes: not only integer values but any comparable values i.e. strings. The initial state probabilities are implemented using a simple array of double. A, B, Pi, states and symbols variables are private, they are set on creation of the HMM, but they can be retrieved anytime using the corresponding property. The constructor of the class accepts 4 parameters

```
public HiddenMarkovModel(MySparse2DMatrix transitions, MySparse2DMatrix emissions,
                         double[] probabilities, int emissionSymbols)
```

the 2 matrixes, initial states probabilities and symbols cardinality.

Following the Professor advice, I created a *TemporalState* class that represent the situation at any time slice. It includes all the computation variables needed:

- `c`, the scaling factor used for both forward and backward algorithm;
- `State`, an array of integers containing the most probable next state in the sequence (used for Viterbi);
- `Alpha`, an array of double to store the alpha values computed in the forward algorithm;
- `Beta`, similar to Alpha but used in the backward algorithm;
- `Gamma`, an array of double used to store the probability of being in each state at time t, used in the *update* method of Baum-Welch algorithm;
- `Delta`, an array of double used to store Viterbi algorithm values;
- `Xi`, a *Sparse2DMatrix* containing the probability of transition from each state to any other at time t, also used in Baum-Welch algorithm.

Each array has the same length and it depends on the number of states of the HMM.

An HMM implements an array of *TemporalState* called `tempInstancts` that needs to be initialized for any given observation before any other computation, using the `InitializeObservation` method.

Since the states and the observations could be represented as string and not only integers, I created a simple class *Id2Str* that implements a dictionary to retrieve the integer corresponding to a given string and vice versa: this way, in the *HiddenMarkovModel* class, only integer indexes are needed.

With this base structure, I implemented all the needed methods as private, using some public methods that require less parameters calling them.

Forward-backward algorithm

The Forward-backward algorithm is implemented in two methods with the same name: in the forward method I compute the scaling factor for each time step that is also used in the backward method. As said before, I followed Rabiner's [3] equations for all phases (initialization, induction and termination) using Rahimi's [9] information to return the correct $P(O|\lambda)$ from the one scaled.

Initialization	$\hat{\alpha}_0(i) = \frac{\pi_i b_i(O_0)}{c_0}$	$0 \leq i < States$	
Induction	$\hat{\alpha}_{t+1}(j) = \frac{\left[\sum_{i=0}^{States} \hat{\alpha}_t(i) a_{ij}\right] b_j(O_{t+1})}{c_{t+1}}$	$1 \leq t \leq T-1$	$0 \leq j < States$
Termination	$\hat{P}(O\|\lambda) = \sum_{i=0}^{States} \hat{\alpha}_{T-1}(i)$		
Initialization	$\hat{\beta}_{T-1}(i) = \frac{1}{c_{T-1}}$	$0 \leq i < States$	
Induction	$\hat{\beta}_t(i) = \frac{\sum_{j=0}^{States} a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{c_t}$	$T-2 \geq t \geq 0$	$0 \leq j < States$
Termination	$\hat{P}(O\|\lambda) = \sum_{i=0}^{States} \pi_i b_i(O_0) \hat{\beta}_0(i)$		

The returned value for both methods is the termination step value multiplied by the product of all the scaling factors

$$P(O|\lambda) = \hat{P}(O|\lambda) * \prod_{t=0}^{T} c_t$$

Viterbi algorithm

The Viterbi algorithm is implemented in the homonyms method: it returns the most probable path as an array of integers and the state optimized probability as the second argument. The base step is simply the multiplication between the initial state probabilities and the probability of extracting the observed symbol in each state (given by the emissions matrix). In the induction step, for each time step and for each state, the transition with the highest weight is computed (the most probable one) storing the *Delta* and the state index in the corresponding *TemporalState* variables. The trackback is needed to compute the most probable path and to get the sequence probability. Since probability of each state at each time step is stored in the *State* variable of the *TemporalState*, the backtracking is simply done with a loop

```
for (int t = T - 2; t >= 0; t--)
    path[t] = tempInstancts[t + 1].State[path[t + 1]];
```

Baum-Welch algorithm

The private method takes as input an array of observations, a training set

```
void baum_welch(int[][] observations)
```

The algorithm has two steps:

- Calculating the forward probability and the backward probability for each HMM state;
- On the basis of this, determining the frequency of the transition-emission pair values and dividing it by the probability of the entire string. This amounts to calculating the expected count of the particular transition-emission pair. Each time a particular transition is found,

the value of the quotient of the transition divided by the probability of the entire string goes up, and this value can then be made the new value of the transition.

Doing this for all input observations, the last phase re-estimate the parameters:

$$\bar{\pi}_i = \frac{\sum_{k=0}^{N-1} \gamma_0(i)}{N}$$

$$\bar{a}_{ij} = \frac{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \xi_t(i,j)}{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \gamma_t(i)}$$

$$\bar{b}_i(j) = \frac{\sum_{k=0}^{N-1} \sum_{t=0,s.t.O_t=v_j}^{T_k-2} \gamma_t(i)}{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \gamma_t(i)}$$

As a little optimization, if the result of the division is 0, the index is removed from the corresponding *Sparse2DMatrix*.

Since the forward and backward algorithms use scaled values, it is possible to compute the re-estimated parameters directly from these

$$\bar{a}_{ij} = \frac{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \hat{\alpha}_t(i) a_{ij} b_i(O_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \hat{\alpha}_t(i) \hat{\beta}_t(i) \frac{1}{c_t}}$$

$$\bar{b}_i(j) = \frac{\sum_{k=0}^{N-1} \sum_{t=0,s.t.O_t=v_j}^{T_k-2} \hat{\alpha}_t(i) \hat{\beta}_t(i) \frac{1}{c_t}}{\sum_{k=0}^{N-1} \sum_{t=0}^{T_k-2} \hat{\alpha}_t(i) \hat{\beta}_t(i) \frac{1}{c_t}}$$

Generate sequence

Since the c++ program and many other I found on the web had a generate sequence method, I decided to create one for my project too. It is fairly simple: the initial state is chosen from π with an extraction based on the probability distribution specified; given the chosen state, a symbol is extracted with the probability specified by the emission matrix and it's added to a temporary list; the next state is chosen using the transition matrix; this sequence is repeated in a loop until the FINAL state is extracted. This process is repeated the number of sequences requested by the user. The extraction is done generating a random number between 0 and 1, and visiting the array corresponding to the state until the summed probability isn't bigger than the generated number: at that point, the index of the last visited position of the array is returned.

Public methods

Methods visible from outside the class call a private method with some fixed parameters. I already mentioned the

```
void InitializeObservation(int[] observations)
```

method before; the *Evaluate* method

```
double Evaluate(int[] observations, bool useBackward)
```

offers two overloads allowing a second parameter to force the use of the backward algorithm (even though the forward algorithm needs to be called too to compute the scale factors). The Viterbi path is obtained using the method

```
int[] MostLikelyPath(int[] observations, out double probability)
```

The Baum-Welch algorithm can be called with different parameters, allowing the user to choose the number of iterations to be performed (it is set to 10 in my project)

```
void Learn(int[][] observations, int iterations)
```

The last public method

```
int[][] generateSequence(int seqNum)
```

is the one used to generate sequences.

File parser

Input files follow the same convention used in the c++ project presented:

- .trans file contains the transition matrix and the symbol on the first line is used to represent the π vector. States and values must be separated by a tab ('\t');
- .emit file contains the emission matrix, first symbol meaning a state, second symbol the extraction. States, extractions and values must be separated either by a tab ('\t') or a space (' ');
- Observations can either be loaded, after creating the HMM, from a .input or a .train file: as a convention .input file is used to compute forward, backward probability or the most probable sequence, while .train file usually contains a lot of sequences and is used to adapt the model using the training algorithm. Symbols must be separated by a single space (' ') here.

Graphical interface

After starting the program, from the first interface (Figure 5) it is possible to create an HMM loading the corresponding files as exposed above using the buttons or the dropdown menu "File". In the *Transmissions* and *Emissions* tabs it is possible to see the loaded files. After pressing the *Create HMM* button the interface will automatically switch to the *Computation* tab (Figure 6) allowing the user to *Load observations* and choose the output wanted. All outputs are written in the *Results* tab as simple text. From the *Computation* tab it is possible to choose between forward and backward algorithm, most probable sequence, generating sequences and training the model.

All the graphical components are initialized by default. The *MainForm.cs* file contains the logic behind the buttons and the text fields. The parser is also contained in this class.

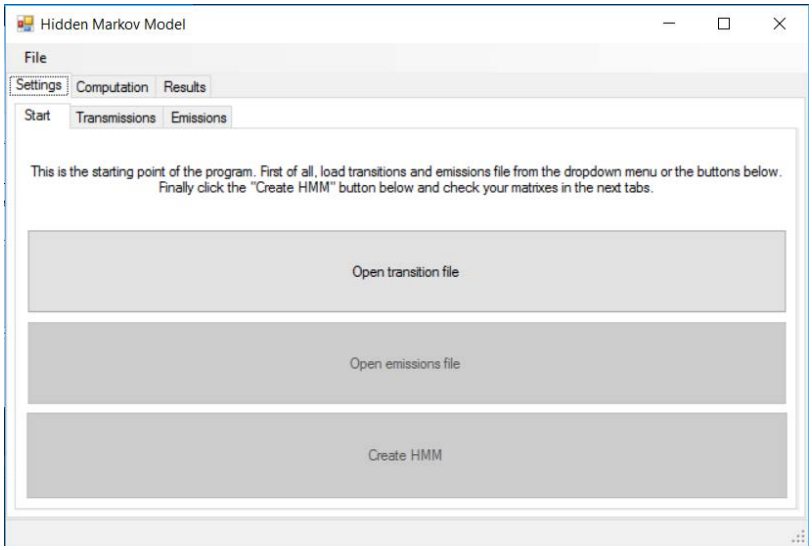

Figure 5 - Starting interface.

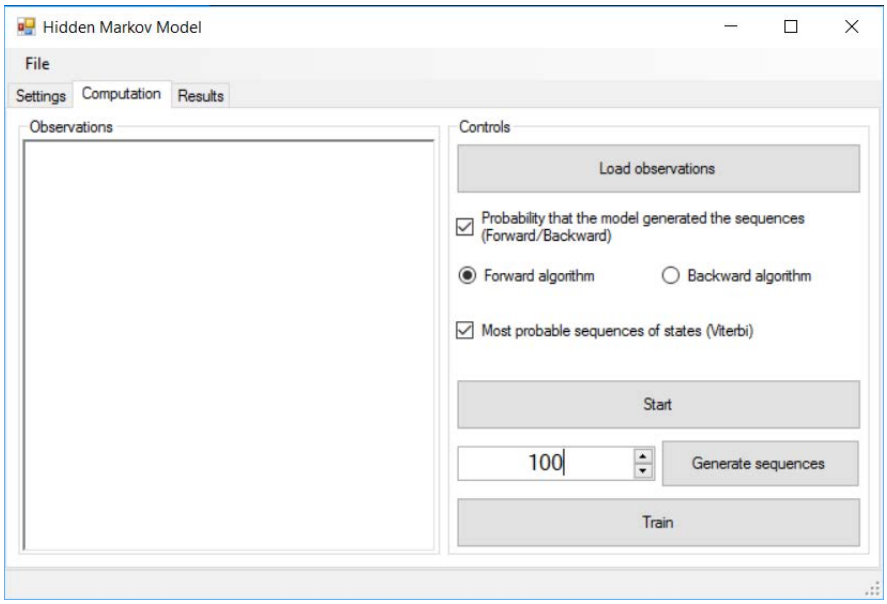

Figure 6 - Computation tab.

4. Testing

To test my program, I followed Professor advice: I started with a deterministic case, then I used the phone example from the c++ program. The results I got on that one were about the same as the c++ program, even though my program was more precise in different cases. The only difference was on the probability of the FINAL state: since there is no observation representing this state, the obtained transition and emissions matrixes don't usually report this. Furthermore, the transition matrix states a self-loop, a transition from last state to himself, with a probability of 1.

A little benchmark was made using a lexicon example: in the c++ program there was a more complex example with 44 states and 39137 (39696 in pos file) symbols, corresponding to 1936 transitions and a total of 46321 (119287 on pos file) emissions. The matrixes represent a part of speech tagger trained with about 41K sentences from the Wall Street Journal (the corpus is not included for copyright reasons). The initial transition probability table allow any state (POS) to follow any other state with equal probability. The initial emission probability table is based on the lexicon in Michael Collins' parser. All emissions in the lexicon are assumed to have equal probability. The format of the training corpus should be the same as the file sample.txt: each line corresponds to a sentence. The tokens are space separated.

The computation to obtain the forward/backward probability and the Viterbi path is about instant; what takes a lot of time is building the emissions matrix, since the parser needs to check over 46000 lines of text. The first observation takes, on average, 9.7ms to be computed by forward or backward algorithm and 8.7ms for Viterbi algorithm, while the second takes 9.2ms and 7.9ms respectively. The average was computed on a sample of 10 program runs. I used the *Stopwatch* class offered by the framework that provides a set of methods and properties that can be used to accurately measure elapsed time.

Examples of forward/backward algorithms, Viterbi algorithm and results comparison between c++hmm and HMM_Solve, is listed in Table 4 and Table 5.

Generate a sequence is really expensive in this example, since the memory needed to store all the visited states grows a lot. All the samples files are stored in the Samples folder: the phone one contains the easier example, while the pos folder contains two examples, pos and pos-init.

C++HMM	HMM_Solve
P(path)=7.71605e-07 path: But CC state NN courts NNS upheld VBD a DT challenge NN by IN consumer NN groups NNS to TO the DT commission NN 's NNP rate NN increase NN and CC found VBD the DT rates NNS illegal JJ . .	P(path)=7,71604938271604E-07 path: But CC state NN courts NNS upheld VBD a DT challenge NN by IN consumer NN groups NNS to TO the DT commission NN 's NNP rate NN increase NN and CC found VBD the DT rates NNS illegal JJ . .
P(path)=7.8125e-05 path: The DT Illinois NNP Supreme NNP Court NN ordered VBD the DT commission NN to TO audit NN Commonwealth NN Edison NN 's NNP construction NN expenses NNS and CC refund NN any DT unreasonable JJ expenses NNS . .	P(path)=7,8125E-05 path: The DT Illinois NNP Supreme NNP Court NN ordered VBD the DT commission NN to TO audit NN Commonwealth NN Edison NN 's NNP construction NN expenses NNS and CC refund NN any DT unreasonable JJ expenses NNS . .

Table 4 - Comparison between c++hmm and HMM_Solve on pos-init HMM.

>>> Observation 0 : <<< Forward/Backward result: 2,81657594802824E-61 Viterbi result: P(path)=0,443871695786695 path: But CC state NN courts NNS upheld VBD a DT challenge NN by RP consumer NN groups NNS to TO the DT commission NN 's POS rate NN increase NN and CC found VBD the DT rates NNS illegal JJ . .
>>> Observation 1 : <<< Forward/Backward result: 3,1524208690057E-66 Viterbi result: P(path)=0,580857819718282 path: The DT Illinois NNP Supreme NNP Court NNP ordered VBD the DT commission NN to TO audit VB Commonwealth NNP Edison NNP 's POS construction NN expenses NNS and CC refund VB any DT unreasonable JJ expenses NNS . .

Table 5 – Another result example of Viterbi path using the pos HMM in my program.

5.Conclusions

Using the provided papers about HMM it was quite easy to implement the program. I tried to follow conventions on variable names to make the code as readable as possible. Having an object containing all the variables (the *tempInstancts* array) it was really easy to access any value from the different methods. An improvement would be to allow the user to set a number of iteration of the training algorithm, or, even better, to iterate until convergence.

I released this project on my Github account [10] and I received an ASP.NET developer job request.

Bibliography

[1] “Hidden Markov model,” [Online]. Available: https://en.wikipedia.org/wiki/Hidden_Markov_model.

[2] “Dynamic Bayesian network,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_Bayesian_network.

[3] L. R. Rabiner, “A Tutorial of Hidden Markov Models and Selected Applications in Speech Recognition,” IEEE.

[4] C. d. Souza, “Hidden Markov Models in C#,” 2010. [Online]. Available: http://www.codeproject.com/Articles/69647/Hidden-Markov-Models-in-C.

[5] Superbobry, “Hmmlearn,” [Online]. Available: https://github.com/hmmlearn/hmmlearn.

[6] R. University, “GHMM Library,” [Online]. Available: http://ghmm.org/.

[7] N. M. Abbasi, “Hidden Markov Methods. Algorithms and Implementation. Final Project Report. MATH 127.,” [Online]. Available: http://12000.org/my_notes/HMM_program/report/report.htm.

[8] “HMM Java implementation,” [Online]. Available: http://cs.nyu.edu/courses/spring04/G22.2591-001/BW%20demo/HMM.java.

[9] A. Rahimi, “An Erratum for "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition",” [Online]. Available: http://alumni.media.mit.edu/~rahimi/rabiner/rabiner-errata/.

[10] D. Vogogna, “HMM_Solver,” 02 2016. [Online]. Available: https://github.com/akyrey/HiddenMarkovModel_Solver.