KTH - Royal Institute of Technology

Pattern Recognition (EQ2340) - Exercise Project A.4 - Algorithm Verification and Speech Database

Alessio Russo - alessior@kth.se (911103-T192)
Lars Lindemann - llindem@kth.se (891113-4131)

October 2015

Contents

1 Forward Algorithm and logprob function

The verification of the forward algorithm and the logprob function will be explained in the following section.
The first part describes the method that has been applied to correct the code as well as the deployed tests. Thereafter, the code itself is analyzed in more detail followed by a list of errors. This list will not only contain the error source, but also what consequences the erros could have and how severe they are. Then it is explained, how these issues are fixed. In the last part of this section, the logprob function is depicted.

1.1 Procedure

The general verification approach has been a peer review approach. One person started correcting the code up until the expected results have been obtained. While verifying, the wrong code segments have only been uncommented and additional code has been added. Using this procedure, it is easy for the peer reviewer to see the differences between old and new code. The second person (peer reviewer) reviewed the code once again having a special look at the differences of the original and the corrected version.
For the verification procedure itself, the algorithm stated in the course compendium has been compared with the matlab code by pure looking. Additionally, the matlab debugger has been used in order to check dimensions of variables and in order to find syntax or runtime errors.
After the two separated work sessions, a discussion session has been held in order to finalize the code and run the tests. The following test has been used for the finite HMM

```
%% finite test case
A       = [0.9 0.1 0;0 0.9 0.1];
q       = [1 0];
mc      = MarkovChain(q,A);

B(1)    = GaussD('Mean',0,'StDev',1);
B(2)    = GaussD('Mean',3,'StDev',2);

x   = [-0.2 2.6 1.3];
pX = prob(B,x);
hMM = HMM(mc, B);

[alfaHat, c]=forward(mc,pX)
logprob(hMM,x)
```

and for the infinite test case

```
%% infinite test case
clc; clear all; close all;

A       = [0.3 0.7 0;0 0.5 0.5; 0 0 0.1];
q       = [1;0;0];
mc      = MarkovChain(q,A);

B(1)    = DiscreteD([1 0 0 0]);
B(2)    = DiscreteD([0 0.5 0.4 0.1]);
```

```
B(3)    = DiscreteD([0.1 0.1 0.2 0.6]);

x   = [1 2 4 4 1];

pX  = prob(B,x);

[alfaHat, c]=forward(mc,pX)
```

The correct values of these cases can be found in the course compendium - for the finite case

$$\hat{\alpha}_{finite} = \begin{bmatrix} 1 & 0.3847 & 0.4189 \\ 0 & 0.6153 & 0.5811 \end{bmatrix}$$
$$c_{finite} = \begin{bmatrix} 1 & 0.1625 & 0.8266 & 0.0581 \end{bmatrix}$$

and for the infinite case

$$\hat{\alpha}_{infinite} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{1}{7} & \frac{1}{79} & 0 \\ 0 & 0 & \frac{6}{7} & \frac{76}{79} & 1 \end{bmatrix}$$
$$c_{infinite} = \begin{bmatrix} 1 & 0.35 & 0.35 & \frac{79}{140} & 0.0994 \end{bmatrix}$$

1.2 Code Analysis

For convenience, identification numbers have been assigned to the issues listed in table 1. The table contains additional information about the solution and the type of the error. The type of error is abbreviated by LW for logical flaw, SE for syntax error, RE for runtime error and MC for minor correction. The mentioned line numbers indicate the line numbers in the original matlab file. In general, if line numbers are mentioned in this document, it concerns the original matlab file.

1. The number of time instances T should be a scalar number and is determined by the second dimension of pX. The previous implementation leads to an array, which would cause runtime errors (e.g. in the for-loop in line 67) if there are more states than time instances T or to logical flaws if there are more time instances than states since the loop acts on the first dimension of pX. The consequences would be severe and range from not executing the code, up to a wrong result.

2. The variable cz is a variable that is preassigned in the beginning and combined with the log operation, which makes the variable equal to 0. It doesn't affect the overall code (it's decoupled) and can simply be deleted. The consequences wouldn't be severe (still correct results) since the lines don't indicate a run-time or syntax error, but it would decrease the speed of the program and makes the code unreadable.

3. This is only a small correction, no consequences would be visible, but the brackets are not necessary. That is why this issue is rated as a minor correction.

4. Line 50 is only a minor correction, since the function *finiteDuration* checks in a nicer way, if a MC is finite or not. No difference would be visible between the old and the new implementation. The lines 51 to 54 are a logical flaw, since an extra zero is added to the initial probability in case the MC is infinite. Neither in a finite MC nor in an infinite MC, a zero is added since the MC doesn't start in the terminate state. Executing this code wouldn't make a difference, since the last element is not accessed in the forward algorithm. For better code readability and program speed, the code fragment has been removed.

5. Additionally, the variable c has been preallocated with zeros. The function description requires c to be a row vector and therefore line 72 needs to be changed. The consequences wouldn't be so severe, since the function would still deliver a correct result, but with a column vector. This could probably lead to dimension problems later on.

6. The rows and columns of B have been mixed up. This is a logical flaw and delivers a wrong output which is a severe consequence. Furthermore, if $T < numberOfStates$, an error would occur since an element out of scope would be accessed.

7. This lines result in a run-time error within line 72 since $zeros(numberOfStates)$ creates a two-dimensional array of zeros and not a one-dimensional array

as it should. Consequences are obvious, because the code can't terminate at all.

8. Again, this is only a minor change. The sum is not necessary since it sums over a scalar value. The conjugate-transpose operator has been exchanged by a transpose operator and the brackets have been deleted. Without these changes, the code would still work.

9. The denominator uses only $c(2)$ all the time and obviously this would lead to wrong results. It has simply been replaced by the time variable in the brackets.

10. Another run-time error has been found in line 77. The dimensions for a concatenation are wrong - the conjugate transpose operator has been removed for this purpose. The old code would result in an error with high severity since no result is obtained.

11. The error is a logical flaw. For finite HMM, the value of $c(T+1)$ needs to be added manually. This hasn't been done in the code. $(c(max(rows, columns))$ has been modified to a deterministic value and would result in a wrong result of the c, while alfaHat would still be correct. For checking this condition, the previous explained fD variable has been used. The correct expression has been added according to the course compendium.

There are several other things that could be modified and improved in order to make the code faster and better readable, e.g. a preallocation of $alfaHat$, but this was not within the scope of the task.

ID	Line	Error	Type	Solution
1	43	T = size(pX);	RE, LW	T = size(pX,2);
2	44	cz = 1;	MC	delete
	53	cz = log(cz);	MC	delete
	59	cz = cz/(cz + rand);	MC	delete
	64	cz = cz*rand;	MC	delete
	75	cz = sign(randn(1))*cz;	MC	delete
3	46	q = [mc.InitialProb];	MC	q = mc.InitialProb;
4	50	[rows,columns] = size(A);	MC	fD = finiteDuration(mc);
	51	if(rows ∼= columns)	LW	delete
	52	q = [q;0];	LW	delete
	54	end	LW	delete
5	-	-	MC	if fD; c = zeros(1,T+1);
	-	-	MC	else c = zeros(1,T);
	-	-	MC	end
	72	c(t,1) = sum(alfaTemp);	LW	c(t) = sum(alfaTemp);
6	58	initAlfaTemp(j) = q(j)*B(1,j);	RE, LW	initAlfaTemp(j) = q(j)*B(j,1);
7	66	alfaTemp = zeros(numberOfStates);	RE	alfaTemp = zeros(numberOfStates,1);
8	60	alfaTemp(j) = ... B(j,t)*(sum(alfaHat(:,t-1)'*A(:,j)));	MC	alfaTemp(j) = ... B(j,t)*alfaHat(:,t-1).'*A(:,j);
9	74	alfaTemp(j) = alfaTemp(j)/c(2);	LW	alfaHat(j,t) = alfaTemp(j)/c(t);
10	77	alfaHat = [alfaHat alfaTemp'];	RE	alfaHat = [alfaHat alfaTemp];
11	79	[rows,columns] = size(A);	LW	delete
	80	if(rows ∼= columns)	LW	if fD
	81	c(max(rows,columns)) = 0.0581;	LW	c(T+1) = ... alfaHat(:,T).'*A(:,numberOfStates+1);

Table 1: Error Table: LW for logical flaw, SE for syntax error, RE for runtime error and MC for minor correction.

1.3 Logprob function

The implementation of the *logprob* function is straightforward. The functions computes

$$\ln P[\mathbf{x}_1 \cdots \mathbf{x}_T | \lambda_i], \quad i = 1, \cdots, N$$

for multiple HMMs objects λ_i. First, we have to make sure that the dimensionality of the output distribution is the same of the samples. In the code, samples are stored columnwise, therefore we have to check the total number of rows of the matrix:

$$\begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_T \end{bmatrix}$$

and compare that value to the size of the output distributions. If they are not the same, the probability is set to $-\infty$, as specified by the interface specification of the *logprob* function.
To get numerically robust calculations we first calculate the scaled probability value with the *prob* function, which are used in the in the *forward* algorithm. Then, we just sum the *log c*-values given by the *forward* algorithm and sum the scaling factors. The code is the following:

```
function logP=logprob(hmm,x)
hmmSize=size(hmm);%size of hmm array
T=size(x,2);%number of vector samples in observed sequence
logP=zeros(hmmSize);%space for result
for i=1:numel(hmm)%for all HMM objects
    if (hmm(i).DataSize == size(x,1)) %check dimensionality of output distributions and samples
        [probs,logScaleFactors]=prob(hmm(i).OutputDistr,x); %scaled probability and scaling factors
        [~,c] = forward(hmm(i).StateGen, probs);
        logP(i)= sum(log(c))+sum(logScaleFactors);
    else
        logP(i)= log(0);%sets to -infinity
    end
end;
```

Testing was done on the example from the book, assignment A.3.1, and the result obtained is:

$$P(\underline{X} = \underline{x}|\lambda) = -9.1877$$

.

2 Speech Corpus

In order to test the speech recognition system, a database has been recorded. The database consists of 14 different words with distinct difficulties. Multi-syllabic words as **recognition** and **environment**, similar sounding words as **affect** and **effect** and short words as **I**, **am**, **in** and **is** in order to challenge the system. Other random words are added and listed in table 2. Additionally, this database allows it to form some simple sentences like **I am Lars** or **Alessio is in Stockholm**, which could be tested.

Each word has been recorded at least 10 times, and at most 20 times, by every person listed in table 3. In the end we have at least 50 recordings for each word, making a total of at least 700 recordings. Of the people listed in table 3, only Paolo used his microphone to record the words, for all the other people each recording has been done with the same microphone(built-in mac microphone). The same settings(16 bit mono wav files with 22050 Hz) were used every time, and the software used was *Audacity*. The test person is supposed to sit half a meter away from the microphone in a silent environment.

Type	**Words**
Multisyllabic	recognition, environment
Similar	affect, effect
Short	I, am, in, is
Names	Alessio, Lars
Random	hand, chair, markov, Stockholm

Table 2: Speech Corpus

Person	**Nationality**
Lars	German
Alessio	Italian
Natalie	Dutch
Martin	Swiss
Paolo	Italian

Table 3: Test person

The database contains variation in the following sense: we have people from multiple nationalities, thus different accents. Moreover, we also have a female voice and recordings done with a different microphone.

3 System

In this section a general description of the system is given. First is discussed how we choose the training and validation set, then a general scheme of the classification system is given.

3.1 Training and validation sets

We make use of the *Cross-Validation* technique to split up the dataset. Specifically, we use the *k-fold cross validation*. This method splits the data set in k equal sized samples, of which $k-1$ are used as training set and the remaining one as validation set. Once we have the training and validation sets, we first train the model on the training set and then check the error based on the validation set. We repeat this process k times, choosing in the end the validation and training sets that have given the best performances. An average error of the k-fold process, and also the variance, will be presented.

This method was preferred to *repeated random sub-sampling validation* because in this way we are sure that all data is used at least once in training and validation. A problem of *k-fold cross validation* appears when you have an outlier in the data: it will be used $k-1$ times as training data, skewing the results. It is not a problem in this project since all recordings were checked beforehand.

To make sure that there is an appropriate number of recording for each person in the training set, the following consideration was made.
Suppose we have a total of N_i recordings for a certain word i, then the training set will be composed of:

$$\frac{(k-1)N_i}{k}$$

recordings. For each person j we have N_{ij} recordings, such that

$$N_i = \sum_{j=1}^{5} N_{ij}$$

For each person j, for the word i, we divide the recordings in k folds, so the training set for the person j, word i is given by T_{ij}. Then the final training set is given by:

$$T_i = \cup_{j=1}^{5} T_{ij}$$

The size of T_i is:

$$|T_i| = \sum_{j=1}^{5} |T_{ij}| = \sum_{j=1}^{5} \frac{(k-1)N_{ij}}{k} = \frac{k-1}{k} \sum_{j=1}^{5} N_{ij} = \frac{k-1}{k} N_i$$

as expected.

3.2 Classification system

For a given word, the system first extracts the *MFCC* features of that word as discussed in assignment 2. We obtain a feature vector $\mathbf{x}$ that is fed into each *HMM* $\lambda^i, \forall i$ and we obtain the value $P_i(\mathbf{x}) = P(\mathbf{x}|\lambda^i)$ for each *HMM*. Then we apply the argmax operator on these value in order to choose the most probable *HMM* that generates $\mathbf{x}$. The result is $\hat{i} = \underset{i}{\text{argmax}} P_i(\mathbf{x})$. The scheme is shown in figure 1.

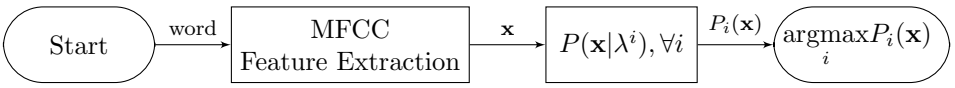

Figure 1: Scheme of the classification system