Rapport TC4

HMM

Par Daro HENG, Hong-lin LI, Yuxiang Wang

Sommaire :

---

1. 1-order HMM

At first, we implemented the code in TP, and we make some tests. We use 1-order HMM with Viterbi Algorithm trained by train10 dataset and the error rate decreases from 10.1% to 6.8% in test10 dataset.

In [11]:
```
with open('test10.pkl','rb') as f:
    data_test = pickle.load(f)
```

In [13]:
```
errors = 0
total = 0
errors_letters = 0

for word in data_test:
    obs = []
    true = []
    for pair in word:
        obs.append(pair[0])
        true.append(hmm.Y_index[pair[1]])
    ls_states = hmm.viterbi(obs)

    for i in range(len(ls_states)):
        total+=1
        if ls_states[i]!=true[i]:
            errors_letters+=1
print "Error rate =", errors_letters*1.0/total
```

```
Error rate = 0.0680327868852
```

In [14]:
```
error_counts(data_test)
```

```
total words :  1501
total letters:  7320
Error words rate:  0.371085942705
Error letter rate:  0.101775956284
```

For the test20 dataset corrected by HMM trained by train10 dataset:

In [11]:
```
with open('test20.pkl','rb') as f:
    data_test = pickle.load(f)
```

In [12]:
```
errors = 0
total = 0
errors_letters = 0

for word in data_test:
    obs = []
    true = []
    for pair in word:
        obs.append(pair[0])
        true.append(hmm.Y_index[pair[1]])
    ls_states = hmm.viterbi(obs)

    for i in range(len(ls_states)):
        total+=1
        if ls_states[i]!=true[i]:
            errors_letters+=1
print "Error rate =", errors_letters*1.0/total
```

```
Error rate = 0.13234677371
```

After automatic correction, the error rate reach at 13.2%.

2. Second order HMM

2.1 Model

To improve the accuracy rate, we have changed our model into second order HMM. For the word less than 2 character, we use the 1-order HMM, if the length equal or more than 3, we use the 2-order-HMM to resolve it. We assume the length of the word L>=3.

The algorithm as follow:

Initialization: $V[0,(i,j)] = p(i)*p(x0|i)*p(j|i)*p(x1|j)$, *for all i,j*

$\Psi[0,(i,j)] = 0$

For $1<t<L-2$, $V[t,(j,k)] = \max[V[t-1,(i,j)*p(k|(i,j))] * p(x_k|j,k)]$ *for all i,j,k*

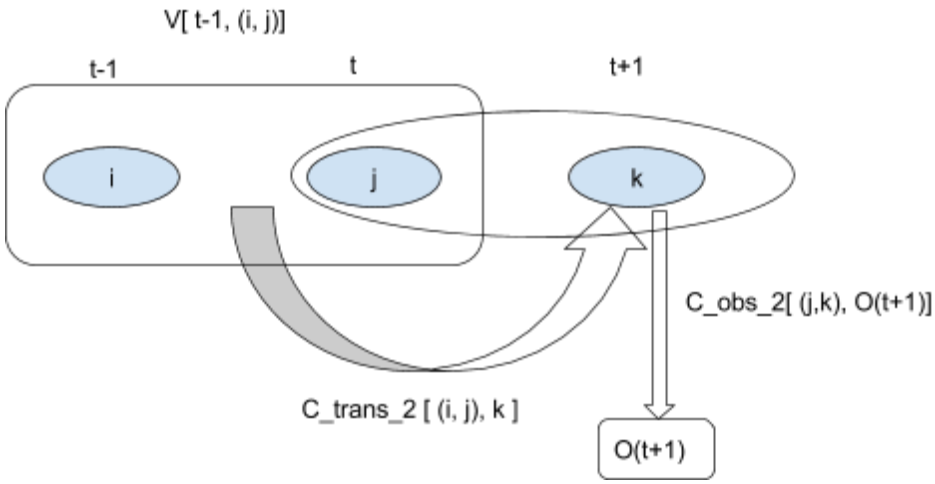
$\Psi[t,(j,k)] = \text{argmax}[V[t-1,(i,j)*p(k|(i,j))]$ *for all i,j,k*

Finally, $P_{MAX} = \max[V[L-2,(i,j)]]$ *for all i,j*

$W_{L-1}, W_L = \arg\max[V[L-2,(i,j)]]$ *for all i,j*

For $1<= l <= L-2$,

$W_l = \Psi[l,(W_{l+1},w_{l+2})]$



2.2 Results

The test is finished on the test10 dataset with the second HMM trained by train10 dataset. We can see the error rate decreases from 10% to 4.68% which is better than the score of first order HMM with 6.8%.

In [9]:
```
with open('test10.pkl','rb') as f:
    data_test = pickle.load(f)
errors = 0
total = 0
errors_letters = 0

for word in data_test:
    obs = []
    true = []
    for pair in word:
        obs.append(pair[0])
        true.append(hmm.Y_index[pair[1]])
    ls_states = hmm.viterbi_2(c_trans_2, obs, c_obs_2, y)

    for i in range(len(ls_states)):
        total+=1
        if ls_states[i]!=true[i]:
            errors_letters+=1
print "Error rate =", errors_letters*1.0/total
```

```
Error rate = 0.0468579234973
```

For the test result of test20, with second HMM trained by train10 dataset:

In [19]:
```
with open('test20.pkl','rb') as f:
    data_test = pickle.load(f)
errors = 0
total = 0
errors_letters = 0

for word in data_test:
    obs = []
    true = []
    for pair in word:
        obs.append(pair[0])
        true.append(hmm.Y_index[pair[1]])
    ls_states = hmm.viterbi_2(c_trans_2, obs, c_obs_2, y)

    for i in range(len(ls_states)):
        total+=1
        if ls_states[i]!=true[i]:
            errors_letters+=1
print "Error rate =", errors_letters*1.0/total
```

```
Error rate = 0.0821999880175
```

Compared to first order HMM, the error rate decreases from 13.2% to 8.2%.

3. Insertion

We have two ideas to resolve this limitation.

First one :

1.We assume that it exists a state “ SPACE ”, it means there is not any character. For the Insertion, we also assume that every state can translate to the state “SPACE”.
2.We assume that every state can observe “SPACE” , It means the user may make a mistake such that the character miss and the “SACER” is instead of this character.
3. We must verify that the probability P(a) * P(b|a) = P(a) * P(“SPACE”| a)* P(b|”SPACE”)
4. For each word to be tested(observation), We make a transformation. For example, “hello” should be transformed to [“SPACE”,”h”,”SPACE”,”e”,”SPACE”,”l”,”SPACE”,”l”,”SPACE”,”o”,”SPACE”]

And now, we can use the algorithm Viterbi for this problem.
If the word is correct, it will return the result as observation. If the word is wrong, Viterbi will correct the wrong character as the first question. If a character is missing, Viterbi will find its observation is different to its state, so it will find the most probable character in this place instead of “SPACE”.

For example [obs0,obs1,obs2]
L = 3
L_transformed = 7
*Initialization T = 0: V[0,i] = p[i] * p[“SPACE”|i], for all i*
Ψ[0,i] = 0 for all i
*for T = t: if obs(t) = “SPACE” V[t,i] = max[V[t-1,j] *p[“SPACE”|i]*p[i|j]] for all i,j*
*if obs(t) is not “SPACE” V[t,i] = max[V[t-1,j]*p[i,j]]*p[obs(t)|i] for all i,j*

So it is the same with before, just add a state “SPACE” and a observation “SPACE”
But, this idea is hard for implementation, because we have no any data for the insertion and we can not make a statistic for the insertion in this data.

So we turned to the second one:

1. We found that if a couple is hardly being together, the probability will be very very small. For example , “b” and “d”. In our mind, it should be “bed” or “bad”. So, we know that the probability p(“b”,”d”) should be very small and also smaller than p(“b”,”e”,”d”).
2. With the theory of probability, we assume that if p(“b”,”d”) is too small and p(“b”,”e”,”d”) is bigger than before, we insert a “e” between (“b”,”d”)

We have tried to implement this idea, it is success to detect some problem and it can insert some characters.
As follow:
The same procedure as the 1-order HMM, but every V[t,i], we add a loop, we try to add a state and calculate its probability

for example [“b”,”d”]
for x in all characters:
*if p[“b”]*p[x|”b”]*p[“d”|x] > p[“b”]*p[“d”|”d”]*
*we add argmax(p[“b”]*p[x|”b”]*p[“d”|x]) between “b” and “d”*

In fact, this method can be realised depend on our data, but it complexity is much bigger than the original HMM. Because we have no data for testing, we can not test its accuracy rate, but we made some unitest, and it will return some good result.
For example: [“b””d”]----[“b”,”e”,”d”]
[“w”,”u”,l”,d”]----[“w”,”o”,”u”,”l”,”d”]

In fact, we can not verify whether our prediction is correct, because, for example [“b””d”], it can be “bad” or “bed” or “bird”, it should depend on the environment.

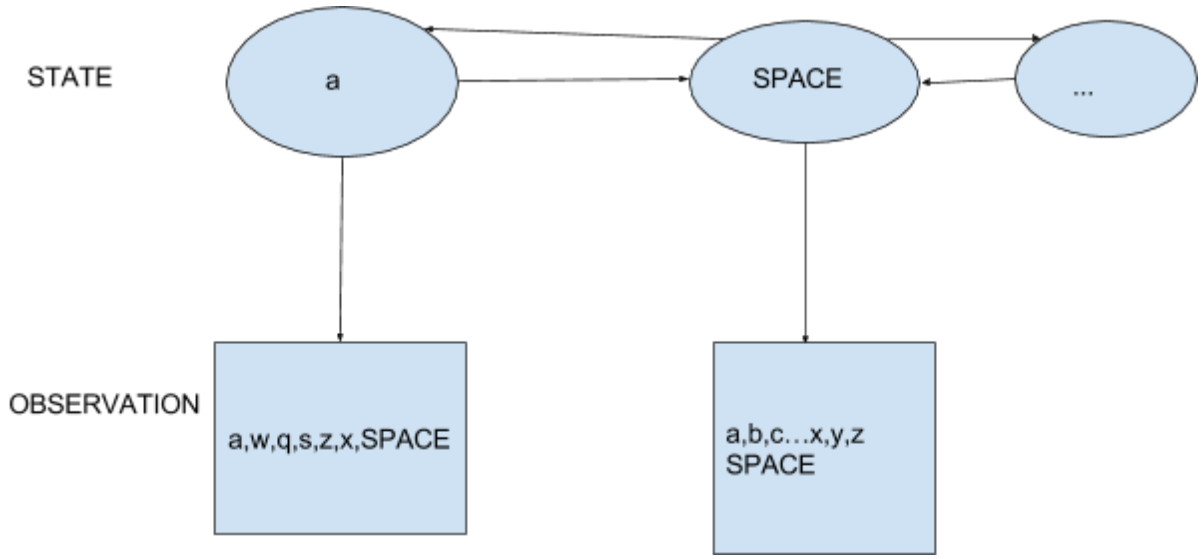
4. Deletion

The same situation for deletion. Depends on the two ideas of the question 3, it is easy to generate two ideas for this part.

First one:

1. ALL hypotheses of the first idea of question 3
2. An additional assumption, the state ” SPACE” can be observation as any other character.

As the graph:



So, we can use the same method for this question

For example [“b”,”z”,”e”,”d”]:
we can find that the couple [“b”,”z”] is rare, and “z” should be deleted.[“b”,”e”,”d”] is more reasonable. So if p[“b”] * p[state = “SPACE”|state = ”b”]*p[obs2 = z|state = ”SPACE”] is the max value in the viterbi matrix, it could be translated to [“b”,”e”,”d”].
But the same reason, we can not implement this idea, because we have no enough data for statistic. We do not know the probability P[state = “a”|state = ”SPACE”] (translation matrix) or P[obs = “a”| state = “SPACE”] …

Second idea:

1. the same as the question 3
2. A little difference,for example [“b”,”z”,”e”,”d”]:
   If the probability
   P(“b”,”z”) = p[state = “b”] * p [state=”z”|state=”b”] * p [obs=”z”|state = “z”]

   P(“b”,x = ANY without e) = p[state = “b”] * p [state=x|state=”b”] * p [obs=”z”|state = x]

   P(“b”,”e”) = p[state = “b”] * p [state=”e”|state=”b”]*p[obs = “e”|state = “e”]

   if P(“b”,”z”) is the biggest, we do not change anything.
   if P(“b”,x =ANY without e) is the biggest one, we correct “z” to other state
   if P(“b”,”e”) is the biggest one , it is probable that we delete “z”.

With this idea, we have implemented our algorithm, it works and it will delete some words, but the composition of english word is too complicated, so in most of situation, the algorithm will prefer correct the character, because its probability is bigger. So, the same problem for us, how to determine the probability. But now, we can only implement a simple algorithm without a complete data set.

Conclusion for deletion and insertion :

We have tried our best to complement our algorithm, and it works. But we also prefer our first idea which is more reasonable and we believe that its performance will be better. But without complete data set, we can do anything. In fact, we tried to assume some values for the P[state = “SPACE”|state = “a”], but it does not work for every test. For better, we need more data and test set.

However, we implement this two operation : insertion and deletion. In most of situation, the probability of correcting a character is bigger than insertion and deletion. So, we can only make some unitests for our algorithm and it return a good result.

5. Unsupervised training

For the part unsupervised, we use the algorithm Baum-Welch. The algorithm of training of Baum-Welch is an algorithm which tries to estimate the parameters <A, B, π> of the HMM. It is an iterative algorithm of re-estimation that works on the same principle as the algorithm of training of Viterbi. In fact, in the Viterbi we count the number of times when every state, every transition and every symbol attached to states is used in the path of Viterbi. Here we want to maximize the real probability of generation, and not those the most likely path. We associate with states, with transitions and with symbols the number of times when they are used for all the sequences and all the path susceptible to generate sequences, balanced by the probability of the path. These balanced accounts are used to ré-estimate the parameters of the model in the same path as for the algorithm of training of Viterbi. First of all, we are going to be interested in algorithm forward and backward which will be used in the algorithm Baum-Welch.

5.1. Forward description :

Instead of using such an extremely exponential algorithm, we use an efficient $O(N^2T)$ algorithm called the forward algorithm. The forward algorithm is a kind of dynamic programming algorithm, that is, an algorithm that uses a table to store intermediate values as it builds up the probability of the observation sequence. The forward algorithm computes the observation probability by summing over the probabilities of all possible hidden state paths that could generate the observation sequence, but it does so efficiently by implicitly folding each of these paths into a single forward trellis.

Each cell of the forward algorithm trellis $\alpha_t(j)$ represents the probability of being in state j after seeing the first t observations, given the automaton λ (the HMM parameters $\lambda = (A, B)$). The value of each cell $\alpha_t(j)$ is computed by summing over the probabilities of every path that could lead us to this cell. Formally, each cell expresses the following probability:

$$\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)$$

Here, $q_t = j$ means "the t th state in the sequence of states is state j". We compute this probability $\alpha_t(j)$ by summing over the extensions of all the paths that lead to the current cell. For a given state qj at time t, the value $\alpha_t(j)$ is computed as :

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t)$$

The three factors that are multiplied in this equation in extending the previous paths to compute the forward probability at time t are :

- $\alpha_{t-1}(i)$ = the previous forward path probability from the previous time step.
- a_{ij} = the transition probability from previous state qi to current state q_j.
- $b_j(o_t)$ = *the state observation likelihood of the observation symbol ot given the current state j*

The pseudo code :

1. Initialization:

$$\alpha_1(j) = a_{0j} b_j(o_1) \quad 1 \le j \le N$$

2. Recursion

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

3. Termination:

$$P(O|\lambda) = \alpha_T(q_F) = \sum_{i=1}^{N} \alpha_T(i) a_{iF}$$

5.2. Backward description :

The second algorithm that we need it's called the backward, which define a useful probability related to the forward probability.
The backward probability β is the probability of seeing the observations from time t + 1 to the end, given that we are in state i at time t (and given the automaton λ):

$$\beta_t(i) = P(o_{t+1}, o_{t+2} \dots o_T | q_t = i, \lambda)$$

It is computed in a similar manner to the forward algorithm :

1. *Initialization:*

$$\beta_T(i) \;=\; a_{iF}, \quad 1 \leq i \leq N$$

2. *Recursion :*

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j), \quad 1 \leq i \leq N, 1 \leq t < T$$

3. *Termination:*

$$P(O|\lambda) = \alpha_T(q_F) = \beta_1(q_0) = \sum_{j=1}^{N} a_{0j}\, b_j(o_1)\, \beta_1(j)$$

5.3. Estimating probability of state transitions :

Now we have the forward and the backward, we can compute the matrix A, B. Let $\xi_t(i,j)$ be the probability of being in state i at time t and state j at time t + 1 :

$$\xi_t(i,j) = P(q_t = s_i, q_{t+1} = s_j \mid O, \lambda)$$

$$\xi_t(i,j) = \frac{P(q_t = s_i, q_{t+1} = s_j, O \mid \lambda)}{P(O \mid \lambda)} = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O \mid \lambda)}$$

Re-estimation the matrix A :

$$\hat{a}_{ij} = \frac{\text{expected number of transitions from state } i \text{ to } j}{\text{expected number of transitions from state } i}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$$

5.4. Estimating observation probabilities :

Let $\gamma_t(i)$ be the probability of being in state i at time t given the observations and the model :

$$\gamma_t(j) = P(q_t = s_j \mid O, \lambda) = \frac{P(q_t = s_j, O \mid \lambda)}{P(O \mid \lambda)} = \frac{\alpha_t(j) \beta_t(j)}{P(O \mid \lambda)}$$

Vector of initial probabilities :

$$\hat{\pi}(s) = \gamma_1(s)$$

Re-estimation the matrix B :

To re-estimate the values in the matrix B, we have to compute the probability of a given symbol v_k from the observation vocabulary V , given a state j :

$$\hat{b}_j(v_k) = \frac{\text{expected number of times in state } j \text{ observing } v_k}{\text{expected number of times in state } j}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1,\text{s.t.o}_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

5.5. The forward-backward algorithm (Baum_Welch)

The forward-backward algorithm starts with some initial estimate of the HMM parameters λ = (*A*, *B*). We must run two steps. Like the EM algorithm (expectation-maximization), the forward-backward algorithm has two steps: the expectation step and the maximization step.

In the expectation step, we compute the expected state occupancy count γ and the expected state transition count ξ from the earlier *A* and *B* probabilities. In the maximization step, we recompute new *A* and *B* probabilities with γ and ξ.

Function fowrad-backward(observations of len T, output vocabulary V, hidden state set Q) :

Initialize A and B iterate until convergence

Expectation step :

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)} \quad \forall\, t \text{ and } j$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)} \quad \forall\, t,\ i, \text{ and } j$$

Maximization step :

$$\hat{\pi}(s) = \gamma_1(s)$$

.

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

Return A,B, π

Implementation :
We tried to implement the unsupervised HMM training part, but we have some bug.
You can find with this paper, the different algorithms that we have implemented.