ＴＲ－ＳＬＴ－００５５

Statistical Transliteration Model

Sebastien Piraud　　Taro Watanabe

December 18, 2003

To deal with Out Of Vocabulary words, we used a totally statistcal method for English- Japanese Transliteration. We could reach 67% of Word Accuracy and 10.2% of Letter Error Rate on an open test set.

（株）国際電気通信基礎技術研究所
音声言語コミュニケーション研究所
〒619－0288「けいはんな学研都市」光台二丁目２番地２ TEL：0774－95－1301

Advanced Telecommunication Research Institute International
Spoken Language Translation Research Laboratories
2-2-2 Hikaridai "Keihanna Science City" 619-0288,Japan
Telephone:+81-774-95-1301
Fax　　:+81-774-95-1308

Contents

I. General presentation

1. Objective

The purpose is to transliterate words between English and Japanese (katakana words "written" in roman letters). The way English to Japanese will be called transliteration and the reverse way backtransliteration. The fact we use roman letters make it easy to use with other languages.

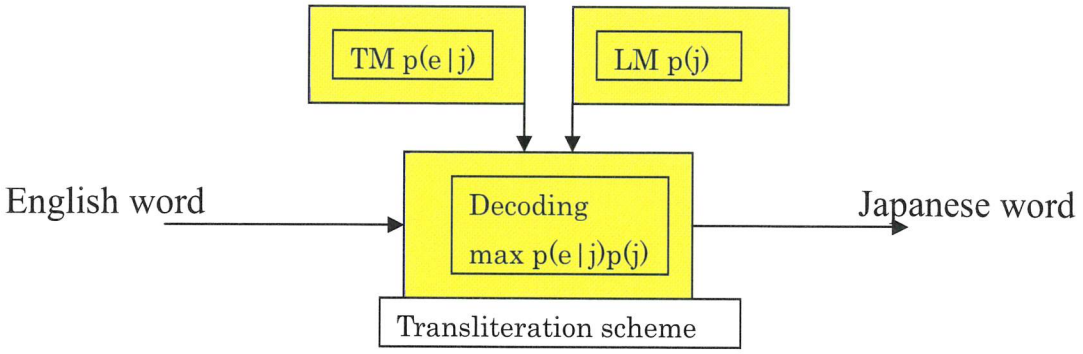
robert ⟶ roba-to
roba-to ⟶ robert

2. Motivation

Transliteration enables to write words from a language into another language, keeping the same pronunciation as much as possible, so that it is useful to deal with proper names such as family names or location names. Especially, such words are often out of vocabularies used in Machine Translation. A transliteration program could be integrated in a MT to deal with OOV words.

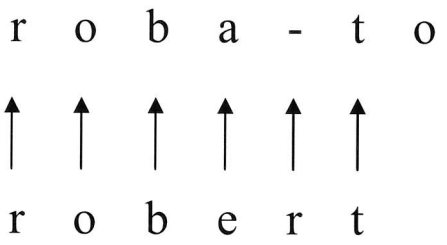
3. Method

We use a statistical method which looks for the most probable transliteration of an English word. A "translation" (or transliteration) model will score the probability that an English word is the transliteration of a Japanese word : p(e|j). A language model will score the probability that the Japanese word is a correct Japanese word : p(j). Following the Bayes rule, we look for the word which maximizes p(e|j)*p(j) : it's decoding.

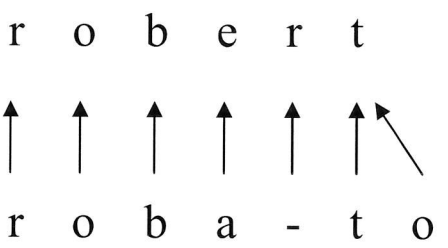


The transliteration model will consider ***alignment*** between the English word and the Japanese word. This method has already been used in translation. But, in the case of transliteration, the alignment must forbid crosses because the pronunciation follows a left to right order.

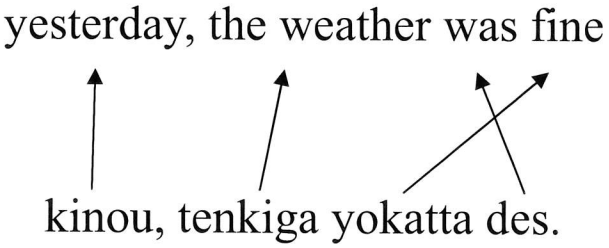
transliteration



backtransliteration



translation



This totally statistical method enables avoiding any phonetic or linguistic study of languages so that it is simple.

II. Corpus

The corpus used was made of names and their Japanese transliteration. This had never been used before that's why some treatments have been done.

First of all, there were a lot of untransliterated words as *abramamovtich* transliterated in a or *vladimorovitch* in v. It concerned lots of Russian names so using a criterion of the ratio between the lengths of the two words, we eliminated these words. Then we notice there were some typing mistakes as *louis* transliterated in *ko-dhinaru*. Using a criterion of Edit distance between the two words of a pair, we eliminated these words.

The data contained words from very different origins: English, French, German, Spanish, Indian, Russian, Arabic...Therefore, we tried to keep only words from English so that we used data which contain the last names and first names found in United States of America to keep only these words and their transliteration. Obviously and unfortunately, it didn't reduce the data to only English names.

Then we randomly separated the data into training set, development set and test set. Eventually, we separated the test set between the words who are also in the training set and those who are not.

File	reng.data	eng_train.data	eng_train1.data	eng_train2.data	eng_test.data
Number of words	113 915	94 980	85 525	9 455	18 935
Number of characters	774 630	645 562	581 305	64 257	129 068
Ratio	6.8	6.8	6.8	6.8	6.8

Statistics on the English data

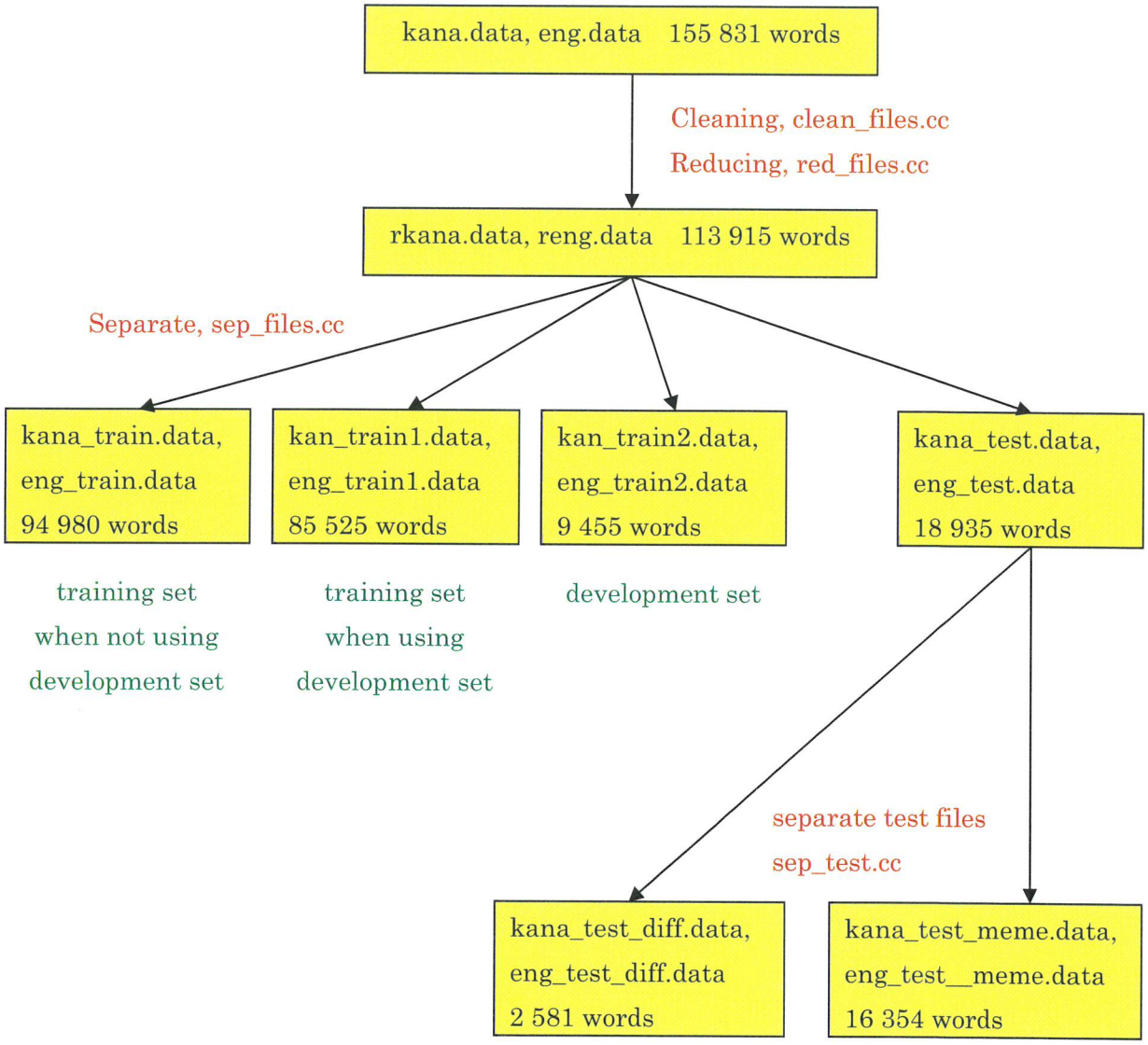
File	rkana.data	kana_train.data	kan_train1.data	kan_train2.data	kana_test.data
Number of words	113 915	94 980	85 525	9 455	18 935
Number of characters	867 774	723 275	651 082	72 193	144 499
Ratio	7.6	7.6	7.6	7.6	7.6

Statistics on the Japanese (katakana) data

File	kana_test_diff.data	eng_test_diff.data	kana_test_meme.data	eng_test_meme.data
Number of words	2 581	2 581	16 354	16 354
Number of characters	22 295	19 568	122 204	109 500
Ratio	8.6	7.6	7.5	6.7

Statistics on the test files

The files "diff" contain words not in the training corpus whereas the files "meme" contain words in the training corpus.


kana.data, eng.data 155 831 words
Cleaning, clean_files.cc
Reducing, red_files.cc
rkana.data, reng.data 113 915 words
Separate, sep_files.cc
kana_train.data,
eng_train.data
94 980 words
kan_train1.data,
eng_train1.data
85 525 words
kan_train2.data,
eng_train2.data
9 455 words
kana_test.data,
eng_test.data
18 935 words
training set
when not using
development set
training set
when using
development set
development set
separate test files
sep_test.cc
kana_test_diff.data,
eng_test_diff.data
2 581 words
kana_test_meme.data,
eng_test__meme.data
16 354 words

III. Language Model

1. Theory

The objective is to determine if the Japanese word proposed as a transliteration of the input English word is correct. The model used is a mixture of N-gram models.

Given a Japanese word $J = j_1 j_2 \ldots j_m$, we want to score P(J). A N-gram model gives us

$$P(J) = \prod_{k=1}^{m} p(j_k \mid j_{k-1} \ldots j_{k-N+1})$$

We also introduce a final letter to score the probability of ending in order to penalize words which would end "too soon". (it is related to the decoding method)

We mix a uniform model, a 1-gram model, a 2-gram model…and a N-gram model

$$P(J) = \prod_{k=1}^{m} \sum_{l=0}^{N} \lambda_l \, p(j_k \mid j_{k-1} \ldots j_{k-l+1})$$

Depending on the number of occurrences of the N-1 letters preceding j_k the models are more or less reliable. When the occurrences are rare, the value of the N-gram can be less reliable so that we compute different values of the λ_l depending of this number of occurrences.

$$P(J) = \prod_{k=1}^{m} \sum_{l=0}^{N} \lambda_l [\# j_{k-N+1} \ldots j_{k-1}] p(j_k \mid j_{k-1} \ldots j_{k-l+1})$$

2. Computing the N-gram models and learning lambda

The first step is to determine the probabilities of each N-gram model. We just have to consider simple counts

$$p(j_k \mid j_{k-1} \cdots j_{k-N+1}) = \frac{count(j_{k-N+1} \cdots j_k)}{count(j_{k-N+1} \cdots j_{k-1})}$$

These counts are realized from the training set : *eng_train1.data.* If this way of computing the probabilities is quite natural, we can notice that is also the one which maximizes the entropy of the training set :

$$\log\left(\prod_{set} P(J)\right)$$

The second step is to determine the most accurate values of the λ_l. We use EM algorithm to compute these values. Using the values of N-gram models computed in the first step, we use the development set: *eng_train2.data* to apply EM algorithm. Here too, we can notice that we will look for the λ_l which will maximize the entropy of the development set using the mixture model.

The Estimation part: for each set of λ_l depending on the occurrences of the first N-1 letters of the N-gram considered, we estimate the expectation of each model as

$$\frac{\lambda_l[\# j_{k-N+1} \cdots j_{k-1}] p(j_k \mid j_{k-1} \cdots j_{k-l+1})}{\sum_{l=0}^{N} \lambda_l[\# j_{k-N+1} \cdots j_{k-1}] p(j_k \mid j_{k-1} \cdots j_{k-l+1})}$$

so that, assuming that $set[\lambda[\# j_{k-N+1} \cdots j_{k-1}]]$ contains the data that use the same λ_l values, we have

$$esp[\lambda_l[\# j_{k-N+1} \cdots j_{k-1}]] = \sum_{set[\lambda[\# j_{k-N+1} \cdots j_{k-1}]]} \frac{\lambda_l[\# j_{k-N+1} \cdots j_{k-1}] p(j_k \mid j_{k-1} \cdots j_{k-l+1})}{\sum_{l=0}^{N} \lambda_l[\# j_{k-N+1} \cdots j_{k-1}] p(j_k \mid j_{k-1} \cdots j_{k-l+1})}$$

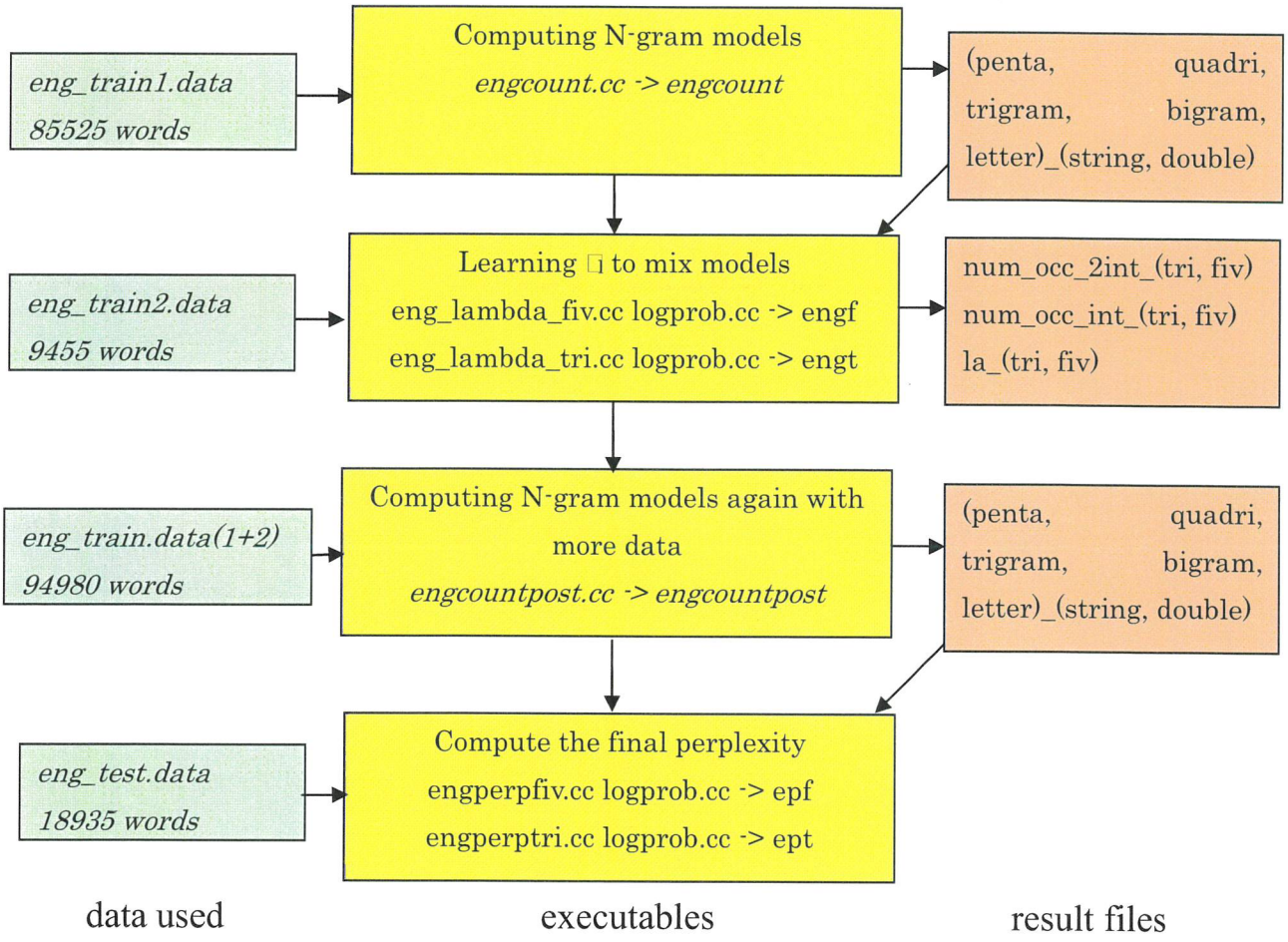
We can interpret this result as if $esp[\lambda_l[\# j_{k-N+1} \cdots j_{k-1}]]$ was the number of times that the mix-probability P(J) comes from the model l so that the Maximization step becomes quite obvious to compute new λ_l* values.

The Maximization part: We divide the expectation by the number of times we use the concerned λ_l values.

$$\lambda_l^*[\# j_{k-N+1} \cdots j_{k-1}] = \frac{esp[\lambda_l[\# j_{k-N+1} \cdots j_{k-1}]]}{card(set[\lambda[\# j_{k-N+1} \cdots j_{k-1}]])}$$

We start with uniform values of □ and loop until the EM algorithm converges.

The third step is to use both training set and development set to compute again the values of the probabilities computed in the first step.



3. Implementation

- engcount.cc, engcountpost.cc

The algorithm is exactly the same for the two files except that engcount.cc

deals with the data of *eng_train1.data* whereas engcountpost.cc deals with the whole data of *eng_train1.data* and *eng_train2.data*.

The algorithm is divided in 2 parts : counting the grams for all models and computing the probabilities. Unfortunately, we cannot use the same count of K-gram to compute the K-gram model and the K+1-gram model. Indeed, let's consider the word robert (or @@@@@robert& as considered in the program).

To compute the 5-gram model, we will consider the 4-grams @@@@@, @@@@r, @@@ro, @@rob, @rob, robe, ober, bert.

To compute the 4-gram model, we will consider the 4-grams @@@@r, @@@ro, @@rob, @rob, robe, ober, bert, ert&.

Therefore, we do have to use different counts.

The results of a N-gram $j_1 \ldots j_N$ are put in 2 files, one containing the string corresponding to the N-gram and an other containing the double corresponding to the probability of $p(j_N | j_1 \ldots j_{N-1})$. For instance the files penta_string and penta_double contains the 5-gram model.

- eng_lambda_fiv.cc, eng_lambda_tri.cc

We just implement the EM algorithm and compute the perplexity after each loop.(see 3. to know the way of computing the perplexity)

We can notice that 2 different files exist : eng_lambda_fiv.cc computes the values of □ to mix an uniform model, a letter model, a bigram model... and a 5-gram model whereas eng_lambda_tri.cc computes the values of □ to mix an uniform model, a letter model, a bigram model and a trigram model.

The results are put in 3 files : The file la_tri (or la_fiv) contains the values of □. But we also have to store which set of □ we have to use. In the case of mixture until 5-gram model, when scoring $j_1 \ldots j_5$, we have to know the number of occurrences of $j_1 \ldots j_4$, we represent $j_1 \ldots j_4$ by 2 integers using the ascii code. In the file num_occ_2int, we store these 2 integers. In the file num_occ_int, we store the integer which represents the set of □ we have to use in the case of $j_1 \ldots j_4$.

- engperptri.cc, engperpfiv.cc

We use the same code than in eng_lambda_fiv.cc, eng_lambda_tri.cc but we had to create a new file to compute again the perplexity of the test data *eng_test.data* after the updating of the probabilities of each N-gram model made by engcountpost.

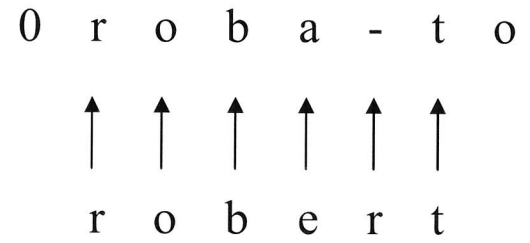
To compute the perplexity of the test data, we just multiply all the scores of the Nw words of the test data which gives us P(E=test set) and apply the formula of perplexity:

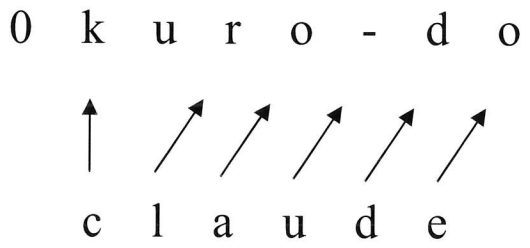
$$perplexity = 2^{\frac{-\log(P(E))}{Nw}}$$

IV. Transliteration Model

1. Theory

The objective is to score if an English word is a good "transliteration" of a Japanese word. The transliteration model will consider ***alignment*** between the letters of the English word and the letters of the Japanese word. This method has already been used in translation. But, in the case of transliteration, the alignment must forbid crosses because the pronunciation follows a left to right order. It is important to notice that we can align an English letter to the NULL letter, it means she is not related to a Japanese letter. For instance, **Charles** and **sharuru** make us think than the final s of Charles is aligned to null letter 0.





It is obvious that a lot of alignments are possible. If we call l the length of the Japanese word and m the length of the English word, there are l^m

possibilities without the constraint of forbidden crosses. The probability of good transliteration will be the highest score among the scores of all alignments acceptable. It means we will look for the *best alignment* possible (also called Viterbi alignment).

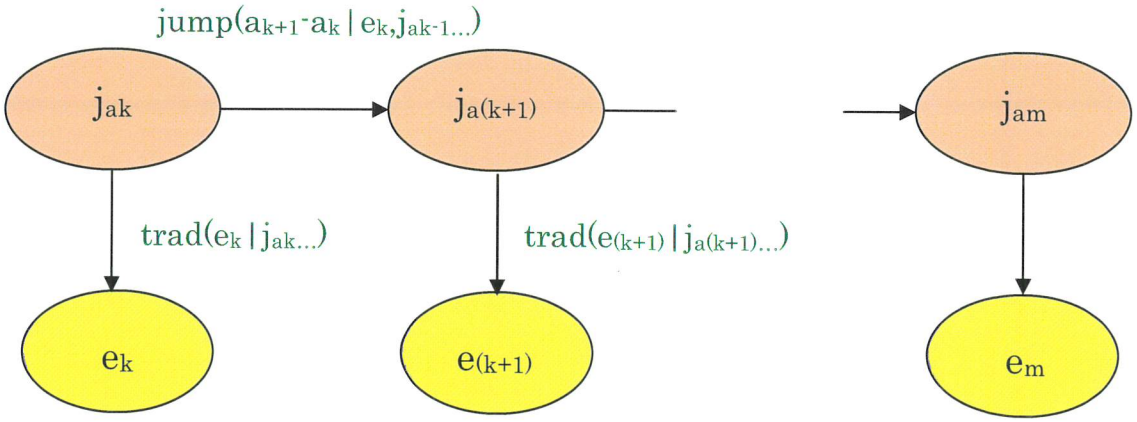
The statistical model used to score the probability P(E|J) is of this form:

$$P(E \mid J) = \prod_{k=1}^{m} p(e_k \mid J)$$

But the interesting thing is to find the model which will describe $p(e_k|J)$. Let's call a_k the index of the Japanese letter to which e_k is aligned. The model must take account of 2 things. First, we want to know if the English letter is aligned to a "good" Japanese letter or not, we will score this thing with $trad(e_k|j_{ak},\ldots)$. Then, we want to know if the position of the letter to which we align is realistic so that we use a model such as $jump(a_k\text{-}a_{k-1} \mid e_k, j_{ak-1},\ldots)$. An important thing will be to determine which dependencies are the more useful. Is it useful to introduce a dependency in e_{k-1} or j_{ak-1} ?

$$P(E \mid J) = \prod_{k=1}^{m} trad(e_k \mid j_{a_k},\ldots) jump(a_k - a_{k-1} \mid e_k, j_{a_k-1},\ldots)$$

From this point, we can begin to see that we are building an Hidden Markov Model. The *states* are the *Japanese letters* to which we align or a_k. It is Markovian because the current alignment depends on the preceding one. The states are hidden since we don't know, a priori, which is the alignment between an English word and its Japanese transliteration. Indeed, our data are composed of English words and their transliteration but we are not given the alignment between the letters of the two words. The *observations* are the *English letters* corresponding to the states.



2. Learning the model

➢ General idea

We will use again the EM algorithm. Here, we use it to do supervised learning. Let's consider the pair **lewis ruisu**, it is difficult to say if the **e** of **lewis** is related to the **u** or the **i** of **ruisu** or maybe to the null letter. If we knew, for instance, that **e** and **w** are related to **u**, we could do simple counts saying that we have one time **l** that comes from **r**, **e** from **u**, **w** from **u**, **i** from **i**, **s** from **s**. Then we will just have to make counts respecting the dependencies chosen and to normalize them to have the values of our model. It would be very similar to the way we compute a N-gram model in the LM.

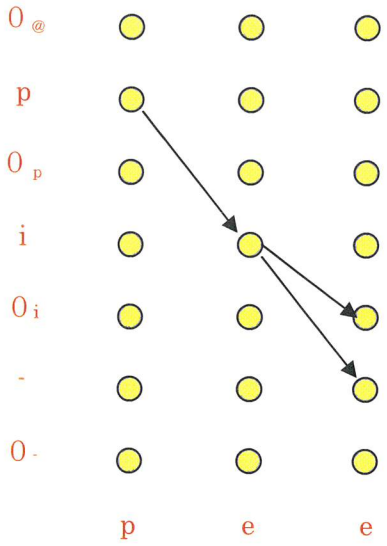
But we do not know the alignment that's why we have to use supervised learning. In fact, after initializing the probabilities of the model, we will consider all the alignments possible for all the words of the training set, will score each alignment and makes the counts described above weighted by the score of the alignment and then used these counts to compute the new probabilities of the model. Then we loop on this process.

Obviously, the initialization, the number of loops and the dependencies are quite important. The EM algorithm only enables to reach a local maximum (of the entropy of the training set) so that a good initialization will enable to avoid this problem. The number of loops is directly related to the problem of over fitting. The dependencies are linked to the issue of data sparseness. We will explain it concretely later.

➢ How to consider all the alignments ?

Considering a pair of words from the learning data, we first implemented a program which built a tree which was exploring all the possible alignments but it was quite long so that we change to use a lattice structure which reduces significantly the complexity of the program.

example of lattice structure with the pair **pee pi-**



Here, we represent two possible alignments :

p->p, e->i, e->-

p->p, e->i, e->0_i

We can remark that we introduce a NULL letter after each real letter in addition of the initial NULL letter. If I aligned to **p** and if I want to align to the NULL letter, I will align to 0_p. This enables us to easily traduce the constraint of non crosses in the constraint of no upraising arrow.

We can easily notice that the arrows are directly linked to the *jump* model (the transition probabilities of our HMM) whereas the nodes are linked to the *trad* model (the observation probabilities of our HMM).

Using lattice structure is a well known process to learn HMM. So we will briefly explain how use the forward-backward procedure. When learning, we build two matrices of the same dimension than the lattice : (2l+1)m. The first one □(i,k) (forward) will contain, for each node(i, k), the sum of the probabilities of all paths from start which result in being in state i after k observations (or that the k^{th} letter of the English word is aligned to the i^{th} letter of the Japanese word). The second one □(i,k) (backward) will contain, for each node(i, k), the sum of the probabilities of all paths which start from the state i after k observations to the end.

□ □

We see that we can compute recursively this two matrices:

$$\alpha(i,0) = trad(e_0 \mid j_i \ldots) jump(i \mid e_0, @\ldots)$$

$$\alpha(i,k) = trad(e_k \mid j_i \ldots) \sum_{u=0}^{i} \alpha(u,k-1) jump(i-u \mid e_k, j_u)$$

and

$$\beta(i,m) = 1$$

$$\beta(i,k) = \sum_{u=i}^{2l} \beta(u,k+1) jump(u-i \mid e_{k+1}, j_i \ldots) trad(e_{k+1} \mid j_u \ldots)$$

Thanks to these two matrices, counting becomes easy. Summing the elements of the last column of □, we have the sum *ptot* of the probabilities of all acceptable alignments, which will enable us to normalize our counts.

For the trad model, we just have to apply for each node(i,k):

$$count(trad(e_k \mid j_i \ldots)) += \frac{\alpha(i,k)\beta(i,k)}{ptot}$$

For the jump model, we have to consider each arrow between the nodes (i,k) and (u,k+1):

$$count(jump(u-i \mid e_{k+1}, j_i \ldots)) += \frac{\alpha(i,k) trad(e_{k+1} \mid j_u \ldots) jump(u-i \mid e_{k+1}, j_i \ldots) \beta(u,k+1)}{ptot}$$

From this lattice structure, we can easily see how we could use a similar method to compute the best alignment just storing for each node the value of the score of the best alignment reaching it and the previous node from which it comes.

$$best(i,0) = trad(e_0 \mid j_i ...) jump(i \mid e_0, @...)$$
$$best(i,k) = trad(e_k \mid j_i ...) \max_{0 \le u \le i} (best(u, k-1) jump(i-u \mid e_k, j_u))$$
$$prev(i,k) = \arg\max_{0 \le u \le i} (best(u, k-1) jump(i-u \mid e_k, j_u))$$

3. Algorithm description

The code is contained in 2 two files Tran.h and Tran.cc. Other files exist they contained different models (different tests on dependencies). But they all follow the same scheme.

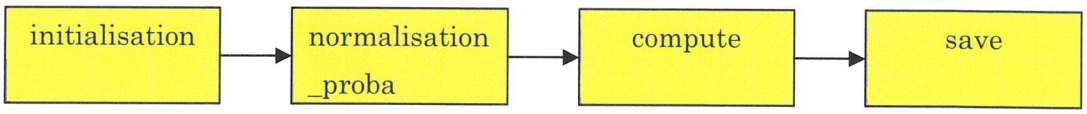
We create an object Tran which have the following methods :

- initialisation() : this method initializes the values of the trad and jump models. It puts 1 for each value of the trad model. For the jump model, we use the value p0 which is the probability of aligning to NULL letter, a "real" jump has a probability (1-p0)*jump(…). As we told before a good initialization enables to avoid the problem of local maxima so that I chose to orient the search with initializing the jump model that way:

$$jump(0 \mid ...) = 0.1$$
$$jump(1 \mid ...) = 0.9$$
$$jump(2 \mid ...) = 0.15$$
$$jump(i \mid ...) = 0.1 * jump(i-1 \mid ...)$$

  it is normalized after, of course.

- normalization_proba : it is just used to normalize the trad model after the initialization.
- Compute() : it implements the EM algorithm.
- Save() : it puts the resulting model in the following files. trad_string, trad_double, jump_know and jump_val. In fact, the trad and jump models are represented by a 3D-vector. The 3 coordinates are directly related to the chosen dependencies. The files trad_string and jump_know contain these 3 coordinates whereas the files trad_double and trad_val contain the probabilities corresponding.

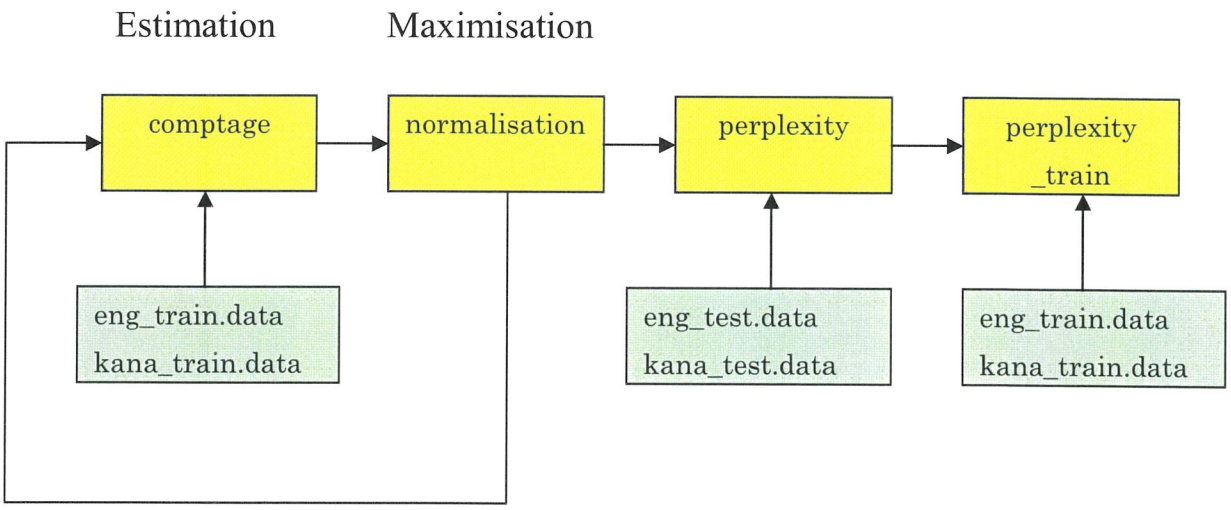


Let's explain how is made compute. Because we can not learn all the dependencies on the same time, we implement in a way such that it is easy to change transfer the parameters of one model to a more complex one. Concretely, we train trad model with only its dependency on the English letter being traduced then we train it with the jump model with only its dependency on the Japanese letter from which we jump and then we introduce other dependencies…So we count using the forward-backward procedure, then we normalize, and eventually, for purpose of control, we display the perplexity of the test set and of training set (we just use the forward procedure to compute it because we just look for the best alignment). Compute uses 4 methods :

- comptage() : this method makes counts necessary for the Estimation part of the EM algorithm. It uses the whole training data (files *eng_train.data* and *kana_train.data*). In fact it applies only applies the method learning() on each pair of words of the training set.
- learning() : realizes the counts on a pair of words using the forward-backward procedure described above.
- normalisation(int, int,….,int) : this method receives 8 integers as parameters, they allow to choose which dependencies we will take

in account (a maximum of 4 dependencies for each model) for computing the new values of the models(which is equivalent to normalize the counts following the chosen dependencies). That's our way to transfer values from a simple model to another one. Indeed the counts have been done with the simple model and then we used these counts to compute the values of a more complex one that we will then use to make new counts. It corresponds to the Maximisation part of the EM algorithm().

- perp() : this method returns the score of the best alignment between a pair of words.
- perplexity() : computes the perplexity of the test set (*eng_test.data* and *kana_test.data*) using perp.
- perplexity_train() : computes the perplexity of the training set (*eng_train.data* and *kana_train.data*) using perp.



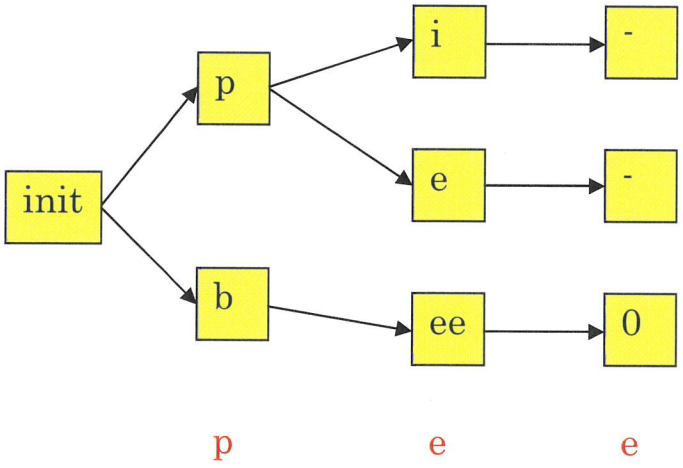
Scheme of compute

Notice : The files Mixtra.h and Mixtra.cc enable to mix 2 TM. They follow the same principle than used to mix the N-gram models of our LM. Obviously, the learning of the weights is made from the development set (*eng_train2.data* and *kana_train2.data*).In this case, the learning must be done only with files *eng_train1.data* and *kana_train1.data*.

V. Decoding process

1. Theory

Using the Language model and the Transliteration model built before, we want to find the best transliteration of an English word. The method used is building a word graph. It means that we will consider each letter of the English word that we want to transliterate following the left to right order. When considering one letter, we will make hypotheses on the way of transliterating it. Obviously, these hypotheses will take in account the hypotheses made for previous letters.



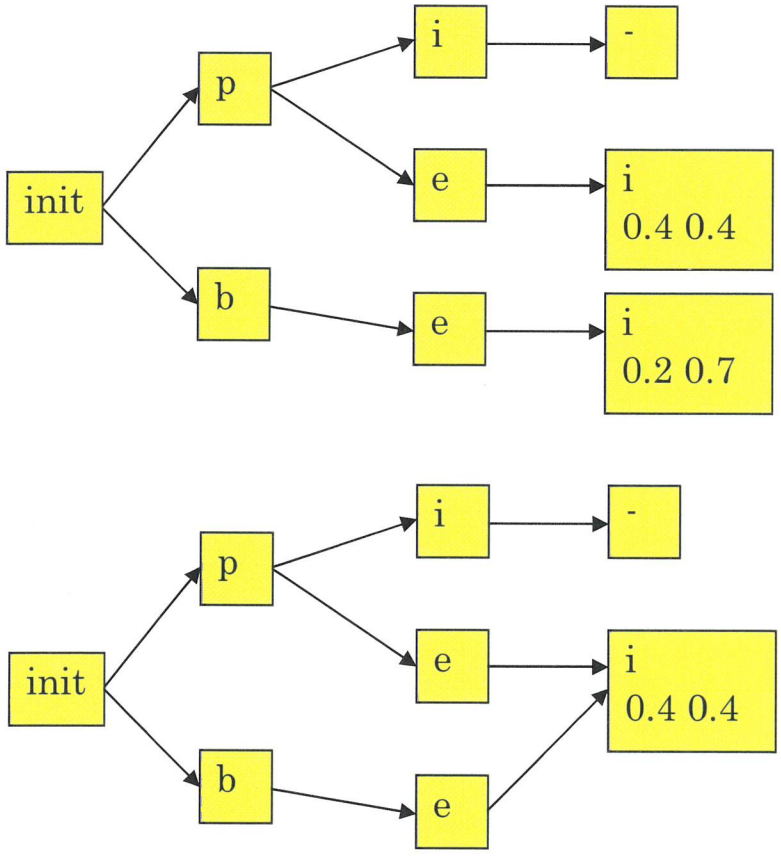
Example of word graph when decoding **pee**

What are the nodes of this graph?

We can see that this graph will be of depth the length of the English name being transliterated. Each node of depth k will be a hypothesis about the transliteration of the k^{th} letter. A node will contain the proposition of transliteration, the score of TM and the score of LM of the best path (partial transliteration) which results in this node and a list of the different nodes who result in this node.

Why many nodes can result in one?

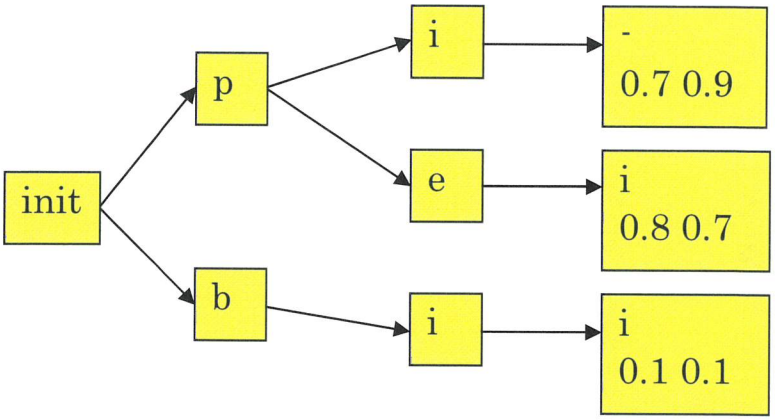
Because of **recombining!** Indeed, to save some memories, we will recombine different nodes in only one. When do we recombine? We recombine when the hypothesis are similar. It means that the N last letters of the partial transliteration are the same. In this case, we choose as scores the best ones and we store all the nodes which resulted in the nodes which are recombined. N corresponds to the N of the N-gram mixture model used as LM.



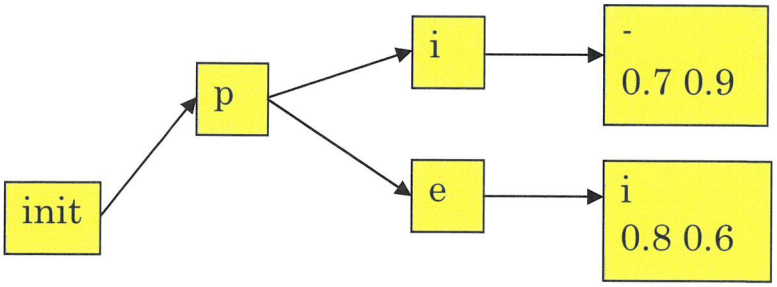
Example with N=2

Do we create only useful nodes?

Probably not, because some hypothesis may be very unlikely… That's why we make some **pruning.** We use both histogram pruning and threshold pruning. After pruning, we erased the nodes which have no more



Then, after pruning with a threshold of 0.2

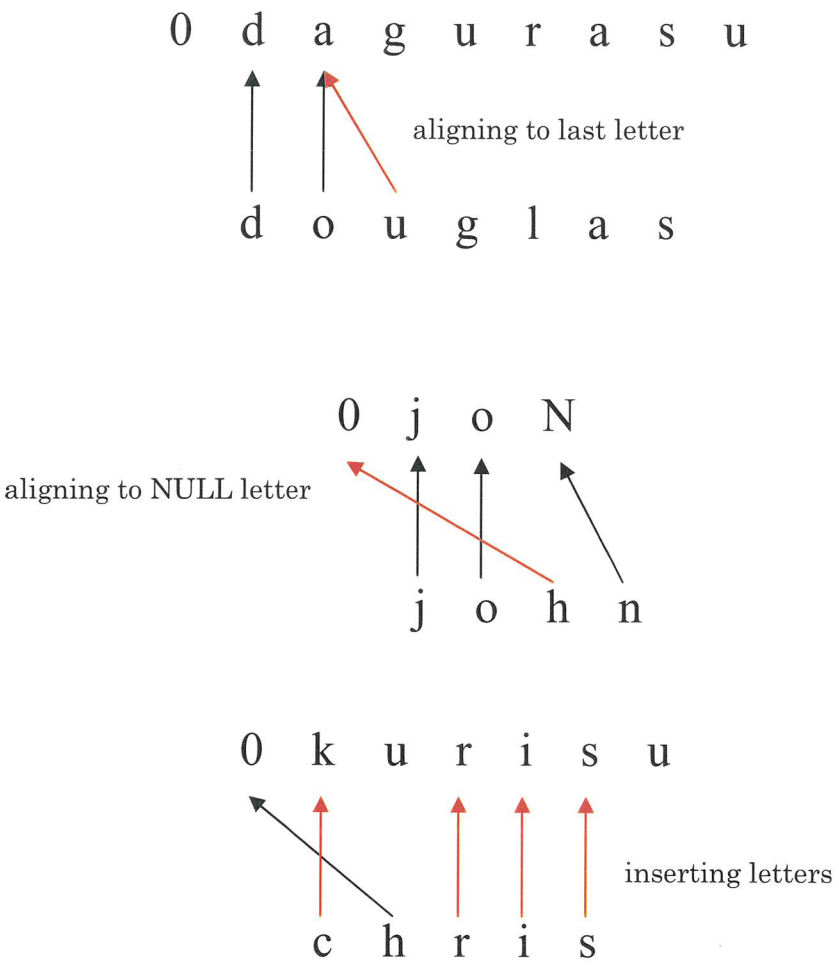


What are the edges?

They just contain the product of the TM and LM scores which make pass from the beginning node of the edge to the final node of the edge.

How do we extend a node?

We have 3 possibilities when considering the transliteration of one English letter: align to the last Japanese letter of the partial transliteration, align to the NULL letter or align to a new letter inserting after the partial transliteration (we can insert many letters)

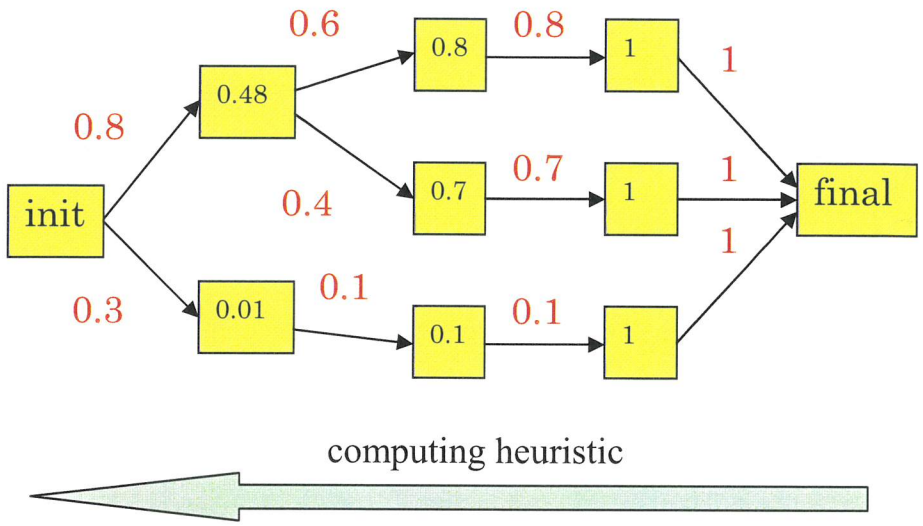


How do we decide the letters to insert?

That is an important question. Indeed, if we do not consider the best transliteration we can score with our models, it will be obviously impossible to choose it! Roughly, the problem is this one: in which letters the English letter I am considering could be transliterated? To determine it, we will act this way. We will compute the Viterbi alignment (see the explanations to compute it in the Transliteration model) between the pairs of the training set. From these alignments we will create 2 tables. The first one contains all the letters (one or more) in which can be transliterated one English letter knowing the last letter of the partial Japanese transliteration. The second one is more general since it contains all the insertions of more than one letter whatever the context. When looking for the best insertions to try, we will use these two tables to choose what insertions to consider.

How extract the "best transliteration" from this word graph?
We will use A* algorithm. For that we need to compute an heuristic which says for each node the best score of the path going from this node to the final node (this final node is a bit factice because the probability to reach it is one). To compute this heuristic, we act recursively from the final node.

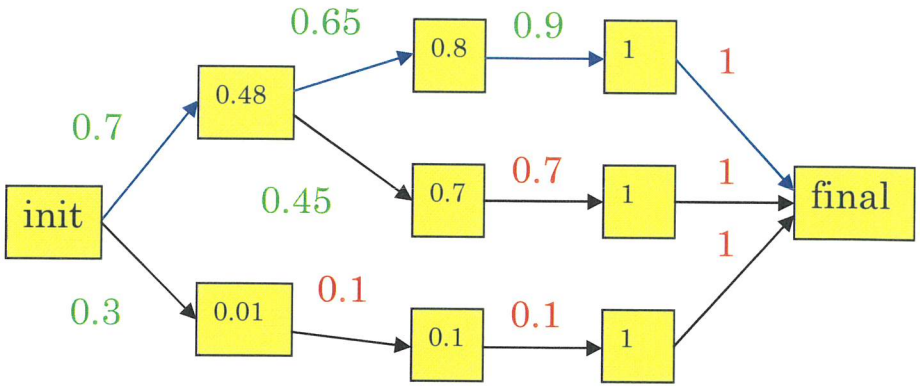
$$heur(father) = \max_{sons}(heur(son) * edge(father, son))$$



Thanks to this heuristic, we can then look for the best path (which will give us our "best transliteration"). We just achieve the A* algorithm. We start from the init node and we always choose the most promising son of the current node.

$$next = \arg\max_{sons}(heur(son) * edge(current, son))$$

We can, at this moment, change the value of the edges, especially for using a more precise model. It is called rescoring. Since we will use this new model only on this best path, we will not have to change all the edges' values. **Rescoring** is useful when you want to build the graph with a fast to compute but not precise model, and then used a more precise but less fast model to achieve A* algorithm.



we found that **pi-** is the best transliteration of **pee**

2. Implementation

The code is contained in files *Decode.h* and *Decode.cc*.

- The constructor

  When building the object Decode, we will compute the Viterbi alignments of the words of the training set in order to create the two tables (named *insertion_dep* and *insertion_gen*) described above and necessary for extending nodes. We also use this computation to quickly calculate the entropy of the training data so that we can control if there is a problem with loading the TM.

- To build the graph

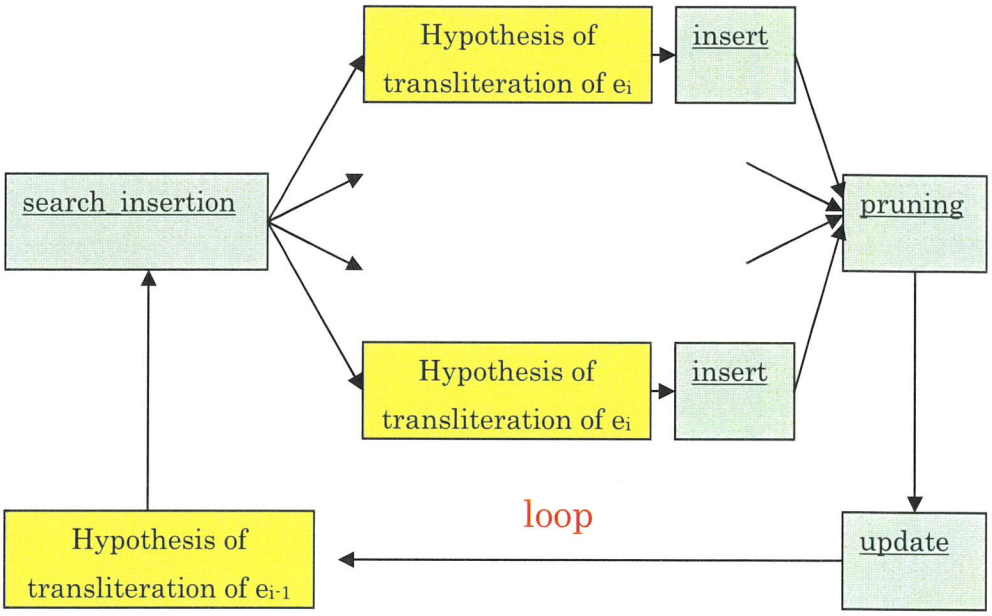
  build() : it is the function which builds the graph of one given English word. Creating an initial node, it will extend it to create hypotheses of transliteration of the first letter (in particular thanks to search_insertion) and insert it in the graph (insert) then it will make pruning (pruning and update). And it loops dealing with each letter of the English word.

search_insertion(…): it will determinate what are the possible letters (one or more) we can insert and put them in a table (best_insertion).
insert(): it inserts the new node in the graph and checks up if we have to recombine it with one other.

before(…): it serves to find the N-gram context of one node. It returns the letters (we can choose the number but no more than N) of the partial hypothesis preceding a node. It is useful for build, insert, rescore.

pruning(): it compares the nodes corresponding to the transliteration of the same English letter and keeps no more than a fixed number of nodes and deletes the really too bad ones in regard of the best one thanks too a threshold.

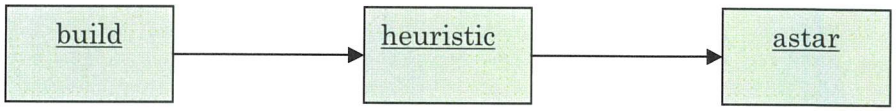
update(): it deletes the nodes which only gave pruned sons.



scheme of build

- To look for the best path.
  heuristic(): it simply implements the procedure described above
  rescore(): computes the new value of a given edge with an other model.
  astar(): it implements the search for the best path as described above and returns the hypothesis with the best score.

- Testing.
  transliterate(): transliterate a word following this scheme:



  test(): applies transliterate on 500 words of the test file containing unlearned words (*eng_test_diff.data* and *kana_test_diff.data*) and on 500 words of the test file containing learned words (*eng_test_meme.data* and *kana_test_meme.data*). These words are chosen uniformly in the test files so that we deal with all the different origins of names.

- Files and functions of control.
  voir_graph(): puts the actual state of the graph in the file *graph*.
  voir_cote(): puts the actual state of the edges in the file *cote*.
  perplan(): computes the perplexity of the test set to check up if there is a problem when loading the LM.

  In the constructor, we also compute the perplexity of the training set for the same reason. Besides, we store the alignments computed for the training set in the file *vital.data*.
  In astar, we compute the 10 best hypothesis, the alignment to which they correspond, and we compute the Viterbi alignment of these 10 involved pairs to check up if we build correctly. These results are stored temporarily in the file *best_hyp*.
  Eventually, we store the errors of transliteration in the file *erreurs*.

VI. Experiments

Those experiments have been conducted mainly in the case of backtransliteration, it means for transliterating Japanese into English.

1. Experiments on the LM.

We built two different LM, one is a mixture of uniform model, letter model, bigram model and trigram-model; we will call it the 3-model. The other is a mixture of uniform model, letter model, bigram model, trigram-model, 4-gram model, and 5-gram model; we will call it the 5-model.

Results for backtransliteration (English LM)

	3-model	5-model
Perplexity	4.05	2.65

Results for transliteration (Japanese LM)

	3-model	5-model
Perplexity	3.48	2.61

Then we tried to see the differences when decoding.

Results for backtransliteration (with best weights between LM and TM) on 500 words **not learned**

	build 3-model rescore 3-model	build 3-model rescore 5-model	build 5-model rescore 5-model
word accuracy	16%	29%	30.8
letter error rate	29.4%	24%	24.1

Results for backtransliteration (with best weights between LM and TM) on 500 words **learned**

	build 3-model rescore 3-model	build 3-model rescore 5-model	build 5-model rescore 5-model
word accuracy	35.8%	67.6%%	69.4%
letter error rate	20.9%	10%	9.5%

Results for transliteration (with best weights between LM and TM) on 500 words **not learned**

	build 3-model rescore 3-model	build 3-model rescore 5-model	build 5-model rescore 5-model
word accuracy	14.6%	13.8%	13.6
letter error rate	28.1%	28.8%	28.9

Results for transliteration (with best weights between LM and TM) on 500 words **learned**

	build 3-model rescore 3-model	build 3-model rescore 5-model	build 5-model rescore 5-model
word accuracy	34.8%	48.4%	49%
letter error rate	23.1%	17.2%	16.9%

We also tested if it was useful to introduce the factice final letter &. It had been introduce to deal with problem of too short words as for instance when **erizabesu** was transliterated in **elizabet** instead of **elizabeth** or (louis and loui...)

Results for 500 unlearned words

	With &	Without &
word accuracy	30.8%	25.6%
letter error rate	24.1%	25.8%

Results for 500 learned words

	With &	Without &
word accuracy	69.4%	57%
letter error rate	9.5%	13%

2. Experiments on the TM.

We tested it only on backtransliteration so it deals with Japanese to English way. We first built a Transliteration Model with many dependencies. The Trad model had dependencies on the following, current and preceding Japanese letters and the preceding English letter. The jump model had dependencies on the previous English letter (not always the preceding letter when inserting many letters), the current and the preceding Japanese letters. It is called **Model 1**.
The perplexity of test set was very high. It was due to the fact that we had not enough data to learn so many dependencies.
So we built a new Trad model, with only dependencies on the current and preceding Japanese letters and the preceding English letter and a new jump model with only dependencies on the previous English letter and the current Japanese letter. This is **Model 2**.
We also tried without the preceding English dependency for the Trad model. This is **Model 3**.
We also mixed **Model 2** with a uniform model, it is **Model 4**.
Then, because we thought it could be useful to have a dependency on the following Japanese letter, we built a New such as the **Model 2** but substituting the dependency on the preceding Japanese letter by a dependency on the following Japanese letter for the Trad model. It is the **Model 5**.
Then we mixed a uniform model, **Model 5** and **Model 2**. It is **Model 6**.

Perplexity of the models

Model	1	2	3	4	5	6
Test set 18935 words	57.2	1.52	1.65	1.48	1.39	1.35
Training set 113915 words	1.21	1.47	1.58	1.47	1.28	1.34

Then we use these models to decode (build and rescore).

Results on 500 unlearned words

Model	2	3	4	5	6
Word accuracy	28.2%	22.8%	28.8%	23.8%	29.4%
Letter error rate	28%	30.9%	28.1%	28.1%	24.7%

Results on 500 learned words

Model	2	3	4	5	6
Word accuracy	67.8%	59.2%	68.4%	66.4%	69.2%
Letter error rate	11.2%	14.4%	11.3%	11.4%	9.7%

Notice: we tried to make the jump model depend on the current English letter (the backtransliteration proposed to the Japanese letter considered). We remarked that was quite bad. It was due to the fact that when looking for the best insertion to try in the decoding process, there was no discrimination. All the scores were too good.

3. Experiments on the decoding process.

We tried to adjust some parameters such as the threshold of pruning. We made go from 0.2 to 0.01 and we saw no differences.

Then we tested what were the best weights to ponder the importance of LM and TM when rescoring. It has been decided because the score of LM was always much higher than the score of TM. (it can be checked with the file *best_hyp*). Since we use the log-probability when decoding we just looked for the best linear combination of the two log-probabilities.

Results with 500 unlearned words

Weight of log(TM)/ Weight of log(LM)	1/1	2/1	3/1	4/1
Word accuracy	28%	25.8%	21.4%	18.6%
Letter error rate	29.2%	27.8%	30.4%	32.9%

Results with 500 learned words

Weight of log(TM)/ Weight of log(LM)	1/1	2/1	3/1	4/1
Word accuracy	69%	62.8%	58.4%	52.4%
Letter error rate	10.8%	12.9%	14.1%	17.2%

Then we tried it also when rescoring **and** building the graph.

Results with 500 unlearned words

Weight log(TM)/ Weight log(LM)	10/10	11/10	12/10	13/10	14/10	15/10	16/10
Word accuracy	28.4%	29.4%	29.4%	30%	29.8%	30.4%	30%
Letter error rate	26.6%	25.8%	24.7%	24.2%	24%	23.6%	23.6%

Results with 500 learned words

Weight log(TM)/ Weight log(LM)	10/10	11/10	12/10	13/10	14/10	15/10	16/10
Word accuracy	70.4%	70%	69.2%	68.2%	68.4%	68.6%	68.2%
Letter error rate	9.7%	9.7%	9.7%	9.8%	9.8%	9.7%	10%

Finally, we tested it on the 1000 first words of the test set (no separation between learned and unlearned words) and we obtained 67% of word accuracy and 10.2% of letter error rate.

4. Examples of transliteration.

We mainly focused on back-transliteration. We will present what kinds of errors we can see.

- back-transliteration

The end of back-transliteration is often a problem.

Input	Output	expected transliteration
a-dhizo-ni	Ardizzon	ardizzone
aN	An	Ann
ferunaNdo	Fernand	fernando
kuro-do	Claud	claude
Gyiyo-mu	Guillaum	guillaume
eburiN	Evelyn	evelyne

But there are also some successes

Input	Output	expected transliteration
beNjamiN	Benjamin	benjamin
hereN	Helen	Helen
adorieNnu	Adrienne	adrienne

We can notice here that the difference between French, Spanish words and English words can be quite important.
Another frequent problem is the one of double consonants.

Input	Output	expected transliteration
Harisu	Harris	Haris
Izaberu	Isabel	isabelle
asumuseN	Asmusen	asmussen
are-guru	Alegre	allegre
Firisu	Phyllis	phylis

And sometimes things are quite ironical…

Input	Output	expected transliteration
Appushou	Apshaw	upshaw
appudagurafu	Updegraf	updegraff
apperubaumu	Apelbaum	applebaum

But it could be good too

Input	Output	Expected transliteration
Whiriamu	William	william
ajeNde	Allende	allende

The fact that Japanese does not use the same sounds is another trouble, among others.

Input	Output	Expected transliteration
airaNdo	Iland	ireland
Agueiasu	Agueas	arguelles
mo-do	Mard	Maud
Rakayo	Lakayo	lacayo
asa-toN	Asarton	atherton
sha-rotto	Sherrot	charlotte

- transliteration

As we can see in the results of LM, transliteration gives less good results than back-transliteration. Here some examples of uncorrect transliteration.

Input	Output	expected transliteration
Claybrook	kure-buruku	kureiburukku
Quarry	kari-	kworii
Kreiner	kureina-	kuraina-
Cartlidge	ka-toriji	ka-torijji
Warnock	wa-noku	wa-nokku

This result is quite surprising because it would be easier to a Human people to transliterate in katakana than to back-transliterate. There are two main reasons to this. The first is that we have lots of words which are written the same way but transliterated differently because of country origins. For instance, **michael** can be transliterated in **maikeru** (English) or **mihyaeru** (Russian) or **mikaeru** (French). The second one is that two different people could give two correct different transliterations.

roba-to	761
robe-ru	115
roberuto	36
ro-beruto	28
ro-beru	3
robaato	3
bobu	2
robaxato	1
ro-be-ru	1
roba-do	1
roba-tsu	1
ro-ba-to	1

Transliterations of Robert and their occurrences found in the corpus

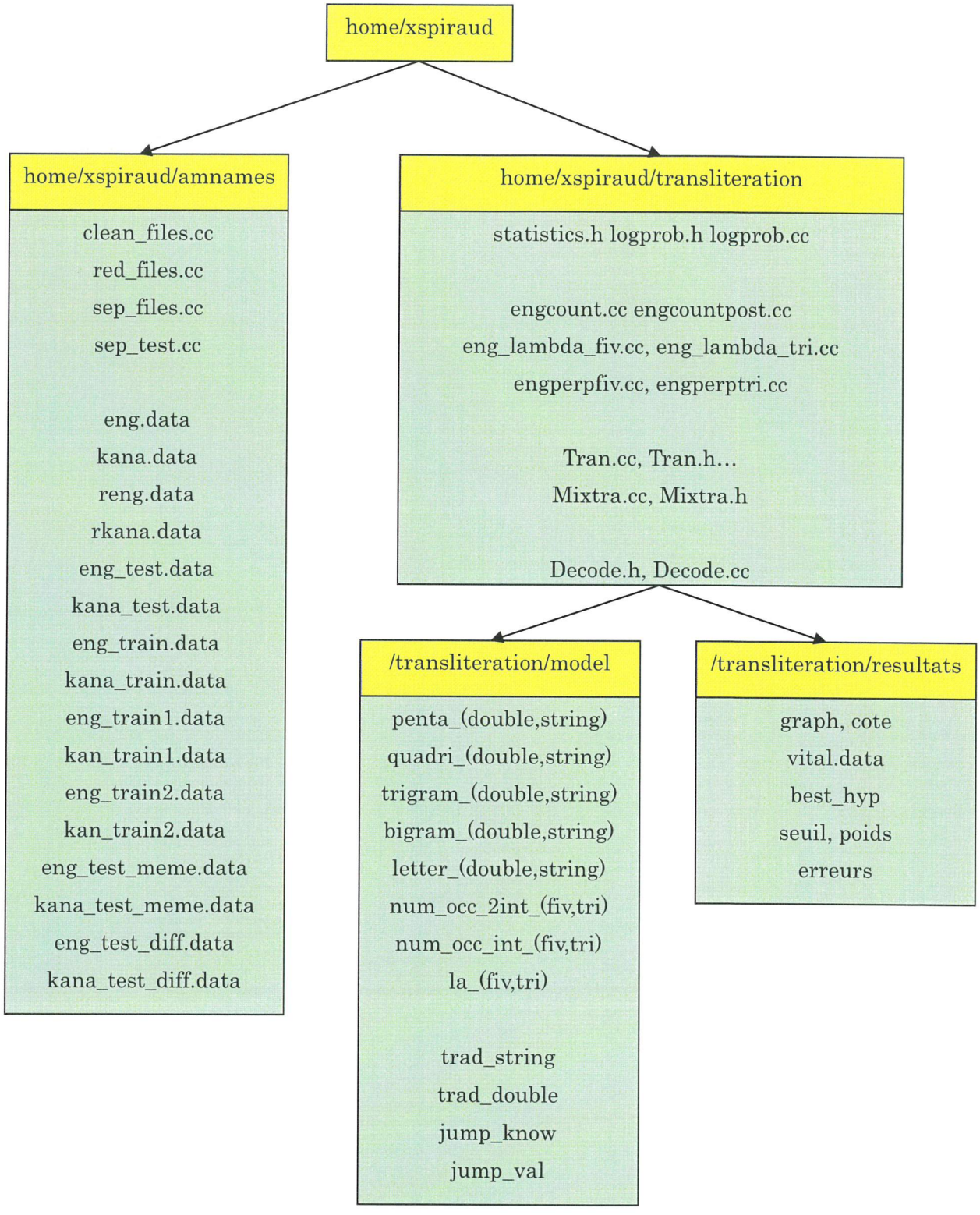
5. What I would have liked to try?

Because, sometimes, the decoding seems to prefer build names already known than "new" names that's why I would have liked to smooth the TM with other ones with less dependencies.

I tried to reward long words (to deal with the problem of ending) but I didn't use accurate weights (see Decodemix.cc the weight w5 should go from 1000 t0 1000 and not from 100 to 100…).

About transliteration, I would have liked to try to apply transliteration and back-transliteration and keep only words which give the same results and learn again only with these words. It could be a way of preventing bad transliterations. I also create a file (in directory *withnewdata*) containing only the most frequent transliteration. But I had no time to test the transliteration with it. It was bad for back-transliteration.

VII. Appendix



VIII. Bibliography

Kevin Knight, *A Statistical MT Tutorial Workbook*, 1999.
Lawrence R. Rabiner, *A tutorial on Hidden Markov Models and selected applications on speech recognition*, 1989.
Brown, Mercer, S Della Pietra, V Della Pietra, t*he mathematics of Machine Translation: parameter estimation*, 1993.
Franz Och, Hermann Ney, *A comparison of Alignment Models for Statistical Machine Translation.*
Franz Och, Hermann Ney, Nicola Ueffing, *Generation of word graphs in Statistical Machine Translation.*
Kevin Knight, Bonnie Glover Stalls, *Translating Names and Technical Terms in Arabic Text.*
Roni Rosenfeld, *The EM algorithm*, 1997.

and also,

Kevin Knight, Yaser Al-Onaizan, *Translating Named Entities Using Monolingual and Bilingual Resources.*
Paola Virga and Sanjeev Khudanpur, *Transliteration of Proper Names in Cross-Lingual Information Retrieval.*
Byung-Ju Kang, Key-Sun Choi, *English-Korean Automatic Transliteration/ Back-transliteraion System and character alignment.*
Byung-Ju Kang, Key-Sun Choi, *Automatic Transliteration and Back-Transliteration by Decision Tree Learning*
Goto, Naoto Kato, Noriyoshi Uratani, Terumasa Ehara, *Transliteration Considering Context Information based on the Maximum Entropy Method*