A probabilistic spellchecker

Olivier Dupin

MSc by Research in Pattern Analysis and Neural Networks

ASTON UNIVERSITY

September 2000

Acknowledgements

First of all, I would like to thank my supervisor, David Barber, for his help, his patience and his time. Thanks to Kevin Murphy for his helpful hidden Markov models toolbox and to all the Neural Computing Research Group.

ASTON UNIVERSITY

A probabilistic spellchecker

Olivier Dupin

MSc by Research in Pattern Analysis and Neural Networks, 2000

Thesis Summary

This thesis studies graphical Models applied to the correction of words. More particularly, hidden Markov models and Markov chains will be used in order to build a probabilistic spellchecker. Several ways to cluster words will be introduced: the batch K-Means clustering algorithm with a specific distance measure and the Expectation-Maximization algorithm in order to learn a mixture of Markov chains. Moreover, a solution for dealing with the suffixes and prefixes will be presented.

Keywords: hidden Markov models, Markov chains. Mixture of models for clustering, batch K-Means clustering algorithm, Expectation-Maximization algorithm. Spellchecking

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The goals of the project

1.1.1 The main ambitions

Spelling correction has been a topic of interest for a long time. However, many spellchecking systems that we have encountered are based on a deterministic application of a small set of rules. Whilst this can work well in words which contain only minor spelling errors, the approach typically fails for more severe corruptions. With a probabilistic framework, however given a rough spelling of an English word, and knowing some features of the language, it should be possible to determine the most likely correct word that the user wanted to type, even for severely corrupted words.

The reasons for miss-spellings are numerous. We can observe 5 principal kinds of errors:

1. The typing error which is basically an error coming from the speed of typing and also from the ignorance of the architecture of the keyboard. For instance, the word *"cqke"* could be written instead of *"cake"*. These are the **spelling errors** and result in words that cannot be found in a dictionary.

2. Lack of knowledge of the correct spelling of a word often results in an observation that is not in the dictionary. Typical for these class of errors is to confuse sequences of letters that have roughly similar sounds. For example, the word *"roufly"* could be written instead of *"roughly"*. Such errors will not be especially considered in this thesis.

3. Correcting spelling errors which come from a confusion of correct words is an other issue. Two typical examples of such words are *"their"* and *"there"* or *"quiet"* and *"quite"*. This type of error is not considered here.

4. We could also consider the **grammatical errors** which need a grammatical context to be corrected (e.g. *"among"* and *"between"*). Again, this is not considered in this thesis.

5. Finally, errors that cross word boundaries (e.g. *"maybe"* and *"may be"*). Such errors are not dealt with here.

When both the corrupted word and the correct one are in the dictionary as in (3), (4) and (5) above, the correction and even the detection of such mistakes are more complicated than for a normal spelling error. Moreover, Peterson has shown that up to 15% of spelling errors that result from elementary typographical error - insertion, deletion and transposition of characters - yield another valid word in the language [12]. This issue has been partially solved with methods dealing with lexical disambiguation and context-sensitive spelling correction: words trigams [11]. Bayesian classifier, decision lists [15] or a Winnow-based approach [7].

As far as this thesis is concerned, we deal solely with typing errors which give a corrupted word not in the language. Other sources of miss-spelling 3, 4 and 5 will not be considered. We aim to create an algorithm which can find the correct word even if the corruption observed is far from the correct one. A typical application of such software would be for a user trying to type a word very quickly as a clumsy typist.

1.1.2 The corruption processes

Let us now present more precisely the corruption processes considered. The entities of a corruption process are:

1. $\mu_1(word, position, char)$: Character insertion

$$\mu_1(\text{`}cake\text{`}, 4, \text{`}r\text{`}) = \text{`}cakre\text{`}$$

2. $\mu_2(word, position)$: Character deletion

$$\mu_2(\text{`}cake\text{`}, 3) = \text{`}cae\text{`}$$

3. $\mu_3(word, position)$: Transposition of two characters

$$\mu_3(\text{`}cake\text{`}, 4) = \text{`}caek\text{`}$$

4. $\mu_4(word, position, char)$: Corruption of a character

$$\mu_4(\text{`}cake\text{`}, 3, \text{`}o\text{'}) = \text{`}caoe\text{`}$$

With a composition of these elementary processes, it is possible to find a function f which provides the correct word given the corruption. One main idea of the approach we take is that the structure of the keyboard is linked to the first and the last elementary processes. Indeed, these four elementary corruption processes are precisely those considered in other well established spell checking systems, in particular Ispell.

1.2 Ispell, the reference spellchecker

Ispell is the spellchecker of Unix. It will be the reference of comparison for our approach. This spellchecker is deterministic. It does not use any contextual information and deals only with spelling errors. Very briefly, we explain how it works and examine its strengths and weaknesses.

Ispell defines some flags which symbolise suffixes and prefixes. The dictionary is a text file containing a list of words (c.f. table 1.1). Each word has an associated list of flags. For instance, the word *"play"* has the flags *DGRSZ* which means that the words *"played"*, *"playing"*, *"player"*, *"plays"* and *"players"* also exist. Ispell considers the same corruption processes mentioned above in section 1.1.2. Given an observation O, Ispell first tries to correct the root of the word (N.B. the roots are words of the dictionary). To do that it generates candidate words M_j which result from only one of the few basic corruption processes $\mu_1, \ldots, \mu_4$. After that, Ispell checks which of the candidate words M_j are in the dictionary. For instance, given the corrupted observation *"plya"* , Ispell generates a list of candidate words using the corruption μ_i: *"aplya"*, *"pwlya"* (Character insertion)..., *"lya"*, *"pya"* (Character deletion)..., *"lpya"*, *"play"* (Transposition)..., *"olya"*, *"pkya"* (Corruption of one letter)... . Since one of these generated candidates is a word in the dictionary, Ispell will propose *"play"*. For a word with a suffix, the procedure is roughly the same except that Ispell first tries to identify the correct flags (suffix/prefix) and then selects in the dictionary the correct $(word, flag)$ couple.

Thanks to using an efficient hash table, the major strength of Ispell is speed. However, being deterministic and considering so few corruption processes, its spellchecking can be disappointing. For instance, the composition of, at least, two processes (different from the identity, function), will never manage to be corrected. For example, $('teelvisiom' = \mu_4(\mu_3('televisiom', 4), 10, 'm'))$ will not be corrected by Ispell. Moreover, given an observation of a word with a suffix, if both the suffix and the root are affected by one of the corruption processes considered before, Ispell is lost and won't propose any correction.

Observing the limits of such a spellchecker, the main target of this work is to apply a probabilistic framework in order to deal with corrupted words.

Taking the previous example, reading the sequence *"teelvisiom"*, most people would immediately recognise the correct word 'television'. The question is why? This is a rather difficult question to answer, but nevertheless important since humans seem to be superior at spellchecking than Ispell. Looking at any simple letter to letter transition in *"teelvisiom"*, it is not clear what the correct word could be. However, over all, it is clear that the only plausible correct word is *"television"*.

One might think therefore that one could correct words by learning the most likely letter to letter transitions in the dictionary, which would alert one to rare combinations. That may help to detect errors but is useless for the corrections of mistake. On the other hand, learning the letter to letter transition for each *word* - i.e. having a Markov model for each word of the dictionary - should help. We hope that the probability $Pr('teelvisiom'|'television')$ is higher than for the other correct words, e.g. $Pr('teelvisiom'|'telescope')$. This is the central idea behind our approach to build a probabilistic model for each word in the dictionary. In order to do so, we need to take into account one of the most fundamental sources of errors: the keyboard.

1.3 The English language and the keyboard

When a user wants to type a word, she first thinks about it and then types a sequences of keys on the keyboard. The structure of the keyboard is very important for the actual

word typed.

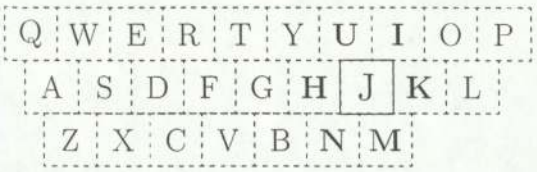


Figure 1.1: The Qwerty keyboards, the notion of levels for the letter J

Examining the Qwerty keyboard figure 1.1 and given that the user wanted to type a *'J'*, it is possible that the real letter typed is in the neighbourhood of *'J'*. We can define a notion of level of mistyping for this letter: $\widetilde{N_1}(J) = \{U, I, K, M, N, H\}$ is the first neighbourhood of the letter considered (the letters in bold face around *'J'*), $\widetilde{N_2}(J) = \{O, L, B, G, Y\}$ is the second one and so on. It is probable that, wanting to type the letter J, the user presses the key J. However, by mistake, she presses a key of the set $\widetilde{N_1}$ or $\widetilde{N_2}$, with a higher probability for the first case than for the second one.

This leads naturally to a probabilistic description of the corruption process μ_4, $Pr(\textit{letter typed}|\textit{intended letter})$.

1.4 The data

Our spellchecking system is based on exactly the same dictionary as used Ispell. Each word in the dictionary has an associated set of flags describing possible prefixes/suffixes that can be attached to the root word (cf. table 1.1).

Format of the dictionary
late/DPRTY
dirty/DGPRST
gray/DGPRSTY
small/PRT
aggregate/DGNPSVXY
create/ADGNSVX
imply/DGNSX
cross/DGJRSYZ
convey/DGRSZ

Table 1.1: Format of the dictionary and the affixes

In total, the dictionary contains 26 057 root words and 36 kinds of affixes (suffix/prefix). As you can see in table 1.2, the suffix or prefix can depend on the two last letters of the word and on the flag: for instance, the verb *"imply"* has the flag D. Since the two last letters of the root word are *"ly"*, we obtain the word with suffix doing

Format of the flags	Examples
flag T:	
E > ST	# As in late > latest
[^AEIOU]Y > -Y,IEST	# As in dirty > dirtiest
[AEIOU]Y > EST	# As in gray > grayest
[^EY] > EST	# As in small > smallest
flag D:	
E >D	# As in create > created
[^AEIOU]Y >-Y,IED	# As in imply > implied
[^EY] >ED	# As in cross > crossed
[AEIOU]Y >ED	# As in convey > conveyed

Table 1.2: Format of the affix flags

-Y,IED, which means that the *'y'* is deleted and *"ied"* is added. Others examples are given in the table 1.2. The complete number of words that the Ispell thus understands is 64429. The words we consider are formed from a vocabulary of 26 characters, the alphabet (no capital letters, no dash).

1.5 Overview

In chapter 2, the main idea of our Bayesian approach will be explained: probabilistic models will be introduced in order to build a spellchecking system. Chapter 3 will demonstrate the necessity of clustering words. Then, chapter 4 shows some results and comparisons with Ispell. Correction of words formed by addition of suffixes and prefixes will not be considered till chapter 5. In chapter 5, three strategies will be studied to partially solve the correction of words with affixes. Finally, chapter 6 proposes the use of mixture of Markov model in order to cluster words.

Chapter 2

The Probabilistic Approach

2.1 Introduction

Imagine that a user wanted to write a correct word C but actually typed M by mistake. A way to correct this mistake would be to compute the probability $Pr(C_i|M)$ for all the correct words C_i of the dictionary, $C_i \in \Delta$. In our probabilistic framework, the most likely correct word, given the corrupted word M is

$$C = Argmax_{C_i \in \Delta}(Pr(C_i|M)) \tag{2.1}$$

Thanks to **Bayes' rule**, we can associate the posterior probability of a word C_i to the likelihood $Pr(M|C_i)$ and prior $Pr(C_i)$,

$$Pr(C_i|M) \; \alpha \; Pr(M|C_i) * Pr(C_i) \tag{2.2}$$

Throughout we assume that the probability $Pr(C_i)$ is uniform, so that the likelihood $Pr(M|C_i)$ determines the suitability of a candidate dictionary word.

Computing the probability $Pr(M|C_i)$ is therefore the main issue of this thesis. However, the real value of this probability is rather difficult to estimate. A possible solution used by Bell Labs for correcting typing errors is a system of tables of error probabilities derived from a corpus of millions of words of typewritten text ([3],[9]). The tables give for example the probabilities of substitution of two letters, or the probability for a *'p'* being inserted after an *'m'*. In theory, we could store many millions of corruptions and correct words with an associated value in respect of the complexity of the corruption processes. However, if we do not want to use giga bytes of hard disk and do not want to be limited by the complexity of the corruption process, this idea is not feasible. An alternative is to find a specific model of the corruption processes.

2.2 Hidden Markov models

The probability $Pr(M|C_i)$ can be written as:

$$\begin{aligned} Pr(M|C_i) &= \sum_S Pr(M, S|C_i) \\ &= \sum_S Pr(M|S, C_i) * Pr(S|C_i) \end{aligned} \tag{2.3}$$

We assume that the corrupted words are generated by a hidden Markov model in which the hidden states correspond to S. In order to develop our model, we introduce the notion of words which could be obtained from corruptions of the correct word, *excluding the character corruption process* μ_4. An instance of a word which could be generated by combinations of the processes μ_1, μ_2, μ_3 is denoted by S. If we interpret S as the result of the elementary processes μ_1, μ_2 and μ_3, then it is clear that $Pr(M|S)$ is the keys-corruption process: it accounts for corruption process μ_4 and $Pr(S|C)$ accounts for combinations of μ_1, μ_2, μ_3. Thus we have assumed that the probability $Pr(M|S, C_i)$ is equal to $Pr(M|S)$. In addition, we assume that $Pr(M|S)$ is independent across letters, that is $Pr(M|S) = \prod_j Pr(m_j|s_j)$. So, equation 2.3 leads to:

$$Pr(M|C_i) = \sum_S \prod_j Pr(m_j|s_j) * Pr(S|C_i) \tag{2.4}$$

For a word of length L_s characters, assuming that the probability $Pr(S|C_i)$ follows a Markov distribution, $Pr(S|C_i) = Pr(s_1|C_i) * \prod_{k=2}^{L_S}(Pr(s_k|s_{k-1}, C_i))$, equation 2.4 gives:

$$Pr(M|C_i) = \sum_S \{\prod_j (Pr(m_j|s_j)) * Pr(s_1|C_i) * \prod_{k=2}^{L_S} Pr(s_k|s_{k-1}, C_i)\} \tag{2.5}$$

The result is equivalent to modelling each correct word of the dictionary with a hidden Markov model, which we now describe more formally.

Hidden Markov models are stochastic graphical models derived from Markov chains, (see [13], [5]). A stationary HMM is described by

1. A hidden state transition probability distribution $\mathbf{A}$, where

$$a_{ij} = Pr(S_j|S_i)$$

2. A confusion matrix $\mathbf{B}$, which describes the probability to generate an observed state Y, dependent on a hidden state S,

$$b_{jk} = Pr(Y_k|S_j)$$

3. An initial state distribution π, where

$$\pi_i = Pr(S_i)$$

A HMM is depicted in figure 2.1 where the upper layer of nodes are the hidden variables, and the lower the observations.

The intuition behind using a Hidden Markov model for spellchecking is that the hidden state transitions should be able to describe roughly the structure of letter to letter transitions, including possible corruption processes μ_1, μ_2, μ_3. The hidden to output process can model the corruption process linked to the architecture of the keyboard (μ_4).

Both the observation and the hidden states of the model are 26 letters of the alphabet. The joint probability of the sequence observation $Y = Y_1, Y_2, ..., Y_T$ and hidden state sequence is given by

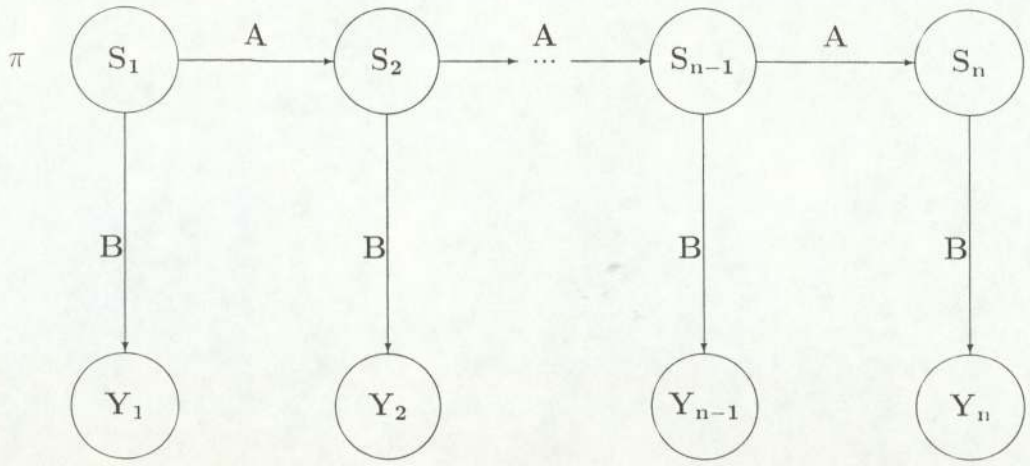


Figure 2.1: Hidden Markov Model

$$Pr(S, Y|\Theta) = Pr(Y_1|S_1)Pr(S_1)Pr(S_2|S_1)Pr(Y_2|S_2)Pr(S_3|S_2)Pr(Y_3|S_3) * \prod_{i=4}^{T} Pr(S_i|S_{i-1})Pr(Y_i|S_i) \quad (2.6)$$

The probability of an observation sequence, $Pr(Y|\Theta)$ can be computed using the Forwards Backwards algorithm (appendix A.1).

To apply HMMs to spellchecking, we could generate training data using an assumed corruption process, and train the model using the standard EM algorithm (Appendix A.2). However, since we have specified the corruption process as $(\mu_1, \mu_2, \mu_3, \mu_4)$, we can set the HMM parameters according to the corruption process directly. A difficulty with this approach is assessing the suitability of the assumed corruption process. However, since we do not have any training data of real (correct word, corrupted) word pairs, we have no principled way to optimise the procedure.

A word is a sequence of states, for example "television" = [20 5 12 5 22 9 19 9 15 14]).

The Confusion Matrix

By definition, this 26×26 matrix is given by:

$$B(i, j) = Pr(Y_j|S_i)$$

Given the interpretation that the confusion matrix represents the corruption process μ_4, this probability depends only on the structure of the keyboard. We assume that the probability of getting the same observation as the current hidden state is equal to 0.7. That means that the values of the diagonal of B are 0.7. The notion of keyboard neighbourhood defined in section 1.3 is now used to set the other values of the confusion matrix: we assume that the probability of typing a letter in the neighbourhood $\widetilde{N_1}$ (set of the nearest neighbours) is 0.25. The possibility of typing a letter outside the first

neighbourhood is 0.05.

$$B(i,j) = \begin{cases} 0.7 & \text{if } j = i \\ \frac{0.25}{card(\widetilde{N_1}(i))} & \text{if } j \in \widetilde{N_1}(i) \\ \frac{0.05}{26 - (card(\widetilde{N_1}(i)+1))} & \text{otherwise} \end{cases}$$

where denominators ensure normalisation $\sum_j B(i,j) = 1$. The matrix is represented in figure 2.2.a.

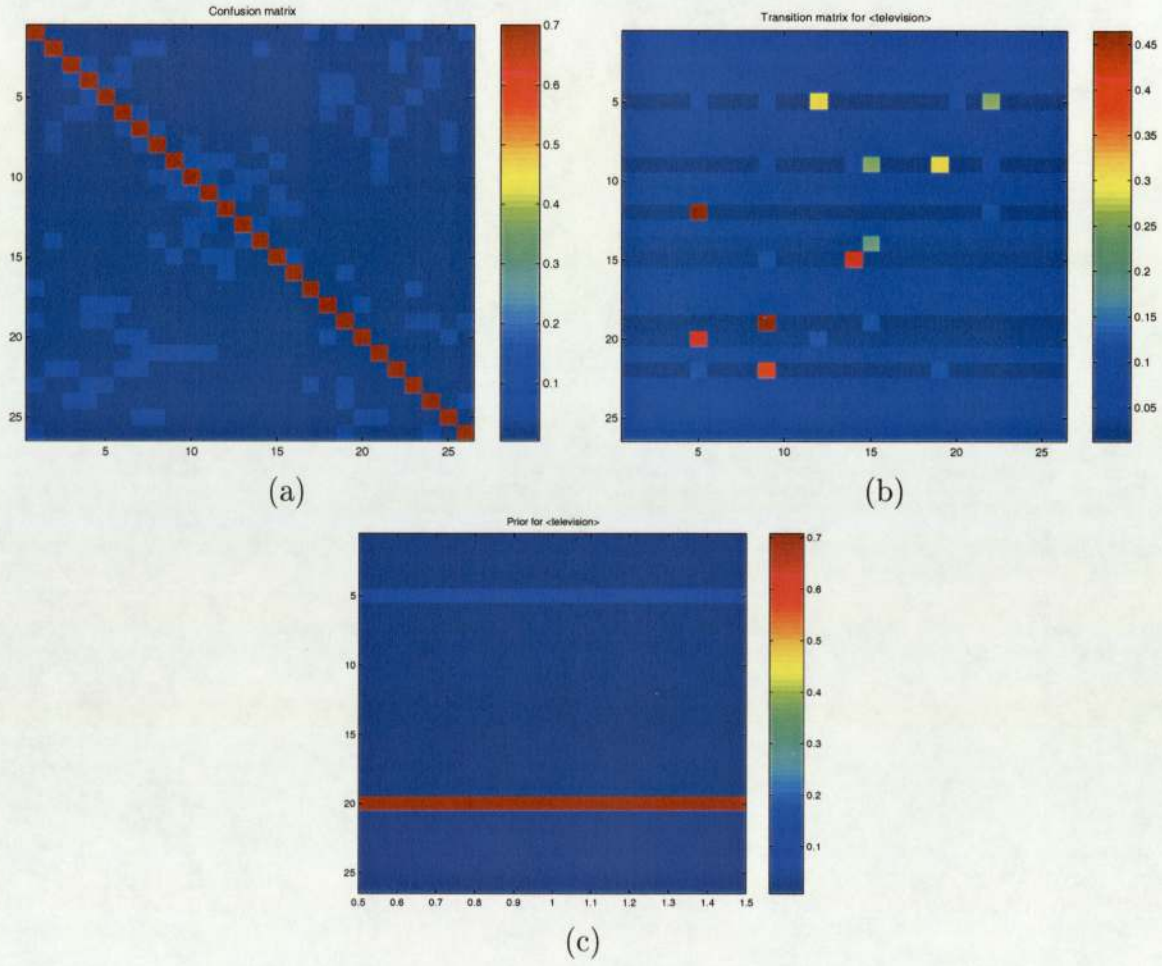


Figure 2.2: Parameters of the models (Confusion, transition and prior) for "television"

The Prior

Given a particular correct word C, we require the probability distribution of initial states. We assume that the probability that the initial state is equal to the first letter of the word is 0.7, i.e. $Pr(S_1 = C_1|\Theta) = 0.7$. Considering only the processes μ_1, μ_2 and μ_3, it is possible that the first state could be the second letter of the word (the case when the first letter has been deleted or swapped with the second one). To account for this, we set $Pr(S_1 = C_2|\Theta) = 0.1$. To include a small probability to have any of

the states at the first node, we add to each case of the prior the value $\frac{0.2}{26}$. There are therefore three possibilities: the first state can be the first letter or the second letter of the correct word (with probabilities 0.7 and 0.1 respectively) or any other letter with probability 0.2.

The Transition Matrix

How can we compute the transition probability $Pr(S_i|S_j)$? One way to do this is to take the distinct letters occurring in the word and store for each of them the previous letter and the two following. Assuming C_2 follows C_1, we add 20 to $A(C_1, C_2)$. However, to account for the possibility that these two characters could be swapped, we add 5 to $A(C_2, C_1)$. In the same way, the characters C_2 can be either deleted or swapped with the character C_3, and we add 5 to $A(C_1, C_3)$. Considering the corruption process μ_1, a letter can be inserted between C_1 and C_2 which we model by adding 5 to $A(C_1, :)$. Finally, we add 1 to all entries in A to avoid null values in that matrix, and normalise A so that

$$\forall i \in [1, 26] \sum_{j=1}^{26} A(i, j) = 1.$$

Perhaps, the best way to explain the algorithm which computes the transition probability is to show an example. The correct word "television" gives the sequence of states [20 5 12 5 22 9 19 9 15 14]. We define the structure shown in table 2.1 which contains the set of the distinct states (Ds), the sets of the two following letters (Fs_1 and Fs_2) and the set of the previous letters (Ps).

Algorithm

For $1 \leqslant i \leqslant length(Ds)$,

$$A(Ds_i, Fs_{1i}) = A(Ds_i, Fs_{1i}) + 20$$
$$A(Ds_i, Fs_{2i}) = A(Ds_i, Fs_{2i}) + 5$$
$$A(Ps_{2i}, Ds_i) = A(Ps_i, Ds_i) + 5$$

End

$A = A + 1$

$A = \text{Normalisation}(A)$

You can observe the transition matrix of this example in figure 2.2.b.

2.2.1 Sanity check

We denote a hidden Markov model built on a word C as $\Theta(C)$. We can check that this model is potentially reasonable with an example:

Construction of the models:

$$Model_1 : \Theta(\text{`}fritillary\text{`})$$
$$Model_2 : \Theta(\text{`}titillate\text{`})$$

Let's take a corruption of the first word:

Ds	Fs_1	Fs_2	Ps
20	5	12	
5	12,22	5,9	20 12
12	5	22	5
22	9	19	5
9	19,15	9,14	22,19
19	9	15	9
15	14		9
14			15

Table 2.1: **The feature of the word 'television'.**
That word is represented by the sequence of states W = [20 5 12 5 22 9 19 9 15 14]. D_s is the set of the distinct states in the word W. The second column gives the set of the following states for each distinct state (for instance, 5 follows 20 and the states 12 and 22 follow the state 5 ...). The third column gives the states following two letters after the state in D_s, e.g. 'l' (12) follows two steps after 't' (20). Finally, the last column gives the previous states of D_s.

$$M = \text{`}rrtulilary\text{`} = \mu_3(\mu_3(\mu_4(\mu_4(\text{`}fritillary\text{`}, 1, \text{`}r\text{`}), 3, \text{`}u\text{`}), 7), 4)$$

We can compute the probabilities based on the HMMs as previously described:

$$Pr(M|Model_1) = 3.6911 * 10^{-10}$$
$$Pr(M|Model_2) = 1.5369 * 10^{-12}$$

This is reasonable since, 'fritillary' is clearly a more suitable correct word than 'titillate' for the miss-spelt word 'rrtulilary'.

2.2.2 Problems with the hidden Markov model

Doing some tests with the hidden Markov models, we noticed some weaknesses. Consider the following example, where $\Theta(C)$ is the model for the word 'television'. For this observation, we have:

$$\begin{aligned} Pr(\text{`}television\text{`}|\Theta(C)) &= 2.1569 * 10^{-7} \\ Pr(\text{`}televosion\text{`}|\Theta(C)) &= 1.9259 * 10^{-8} \\ Pr(\text{`}televsiion\text{`}|\Theta(C)) &= 6.1090 * 10^{-9} \\ Pr(\text{`}televvision\text{`}|\Theta(C)) &= 1.0424 * 10^{-10} \end{aligned}$$

Here, the probability of having a substitution of one letter by another in the neighbourhood is 11.2 times less than the probability of having the right word. The probability of having a single transposition is 35.3 times lower. The problem here is that the probability for transposition is rather too low. Consider the insertion of an extra letter: the fourth calculation of the example demonstrates that the addition of a letter in the middle of the word gives a probability 220 times less important! One of the

reasons is that the confusion matrix affects the computation of the probability when the length of the observation differs from that of the correct word.

A way to try to solve this problem is to incorporate all the corruption processes in the Markov model. We therefore considered an even simpler model, a Markov chain.

An other important weakness in the construction of the Hidden Markov model becomes apparent when we examine the transition matrix in figure 2.2 more closely: When the same letter occurs twice in a dictionary word, the first letter to letter transition is modified by the second because, in the current model, a state is defined as a letter. For instance, when the model for *"television"* is built, the sequence *"ev"* changes the probability to have a 'l' after an 'e'. A possible way to deal with that problem is to make the distinction between the two 'e's of *"television"* assigning two different states to the first and the second 'e'. This approach has not been tried because of a lack of time.

2.3 Using a simple Markov chain

A Markov chain can be considered as a hidden Markov model with an identity confusion matrix. So, contrary to before, the letter to letter transitions should take into account all the four elementary corruption processes. Given a correct word C, we want to compute the prior and the transition matrix of our model.

The Prior

We initialise the prior to the null vector and do the following :

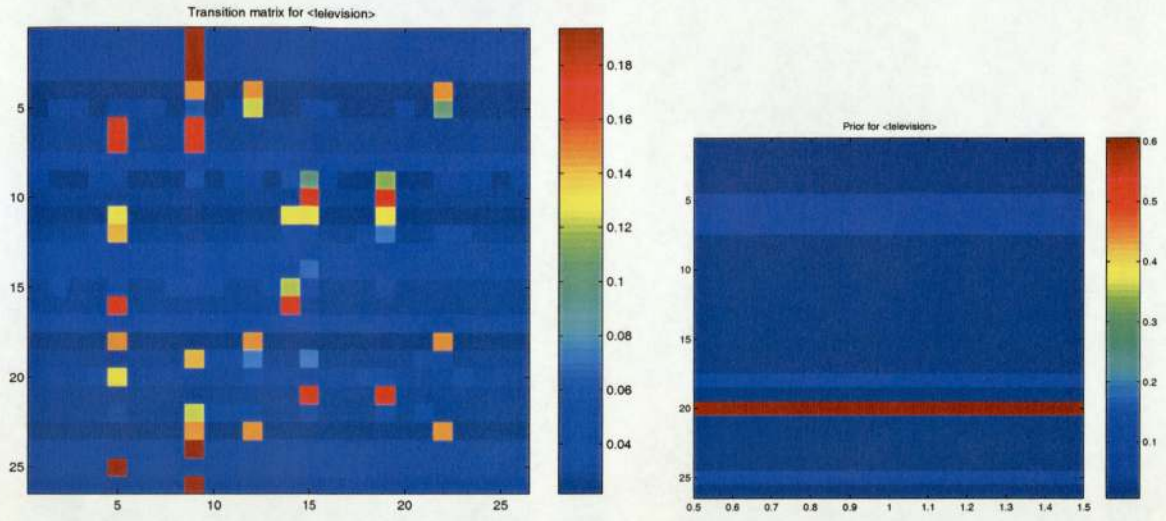
$$\pi(i) = \begin{cases} \pi(i) + 0.6 & \text{if } i = C_1 \quad (1) \\ \pi(i) + \frac{0.2}{card(\tilde{N}(C_1))} & \text{if } i \in \tilde{N}(C_1) \quad (2) \\ \pi(i) + 0.05 & \text{if } i = C_2 \quad (3) \\ \pi(i) + \frac{0.15}{26} & \forall i \quad (4) \end{cases}$$

Doing this for all state i, we assume that the probability that the initial state is equal to the first letter of the word is $0.6 + 0.15/26$ (lines (1) and (4)). The first letter of the word can be substituted with a neighbour with the probability 0.2 (line 2). The deletion and the transposition of letters are considered in line (3): we assess that the initial state is the second letter of the word at 0.05. The fourth line considers the character insertion and ensures that none of the values of the prior are zero.

The Transition matrix

Given the sequence of states (C_i, C_{i+1}, C_{i+2}) in the correct word, in addition to the procedure given for the HMM transition matrix, we put weight on $A_{\tilde{N}(C_i),C_{i+1}}$ (corruption of the first letter by μ_3), on $A_{C_i,\tilde{N}(C_{i+1})}$ (corruption of the second letter by μ_3). You can monitor this matrix and the prior in figure 2.3 for the example 'television'. If we compare the transition matrix obtained with that of figure 2.2.b, we can see that the transition matrix in figure 2.3.a is less diagonally dominant.

Note that none of the values in the prior and the transition matrix are zero. Hence, every observation is possible given such a model. Again, we consider the model for



(a) (b)

Figure 2.3: Parameters of the Markov model (transition and prior) for "television"

'television', and the likelihood of the same typical corruptions:

$$\begin{aligned} Pr(\text{`}television\text{`}|\Theta(C)) &= 2.1569 * 10^{-9} \\ Pr(\text{`}televosion\text{`}|\Theta(C)) &= 5.8137 * 10^{-10} \\ Pr(\text{`}televsiion\text{`}|\Theta(C)) &= 6.7400 * 10^{-10} \\ Pr(\text{`}televwision\text{`}|\Theta(C)) &= 1.0424 * 10^{-10} \end{aligned}$$

The four elementary corruption processes have more reasonable relative probabilities: the transposition of two letters and the corruption of one have quite similar values. Moreover, the addition of an extra letter is less likely, but still 20 times smaller than the correct word.

We therefore decided to use a simple Markov chain for each word in preference to a hidden Markov model. Further tests demonstrated that the separation of the process μ_4 to obtain an hidden Markov model is less useful than the simple Markov chain which considers all the corruptions processes at the same level. Henceforth, throughout the thesis, $\Theta(C)$ denotes the Markov chain derived from the word C.

Chapter 3

Clustering the words of the dictionary

3.1 The need to cluster data

With a dictionary of around 26 000 words, computing a model for each word takes too long. One approach would be to build the 26000 models off-line, and compute the likelihood of the corrupted word for each of the 26000 words. This requires a large amount of storage space, and the process of compiling the likelihood for all the words in the dictionary is also prohibitively slow. With current computational constraints, the number of Markov models that can be built on the fly is around 200. For this reason, we need to rapidly select a small number of candidate words - no more than 200. A Markov model for each of these candidate words can be built and the likelihood of the observed word calculated. The procedure to select a small set of candidate words must ensure that

1. the selection does not take too much time
2. the selection is accurate enough to contain the "correct" word

A further useful requirement is that words with similar Markov structure but with different lengths (e.g.'prefer' and 'preferential') are separated in the selection procedure. The reason for this is that two words can have exactly the same Markov structure, but differ significantly in length.

One approach to this issue is to cluster the dictionary using the batch K-Means clustering algorithm, using a vector representation of a word.

3.2 The frequency vector representation

The ideal representation of the word for a spellchecking system would be a representation which is invariant with respect to the elementary corruption processes μ_1, μ_2, μ_3 and μ_4. A corrupted word and the correct associated ideally have the same representation because, in that case we would be sure to find the right cluster given the corrupted word. Although we will not manage to make a fully invariant representation, transposition and keyboard miss-hits (μ_3 and μ_4) can be dealt with to some extent.

One way to make a representation invariant with respect to the transpositions μ_3 is to use a letter frequency representation. This is a 26 dimensional vector in which each dimension contains the number of occurrences of that letter in the word. For example:

$$\overrightarrow{abbreviate} = [2\ 2\ 0\ 0\ 2\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0]$$

since 'abbreviate' contains two 'a's, two 'b's, none 'c's, etc.

3.2.1 Definition of a distance measure

In order to perform clustering, we need a distance measure between frequency vectors. A first thought would be to take the Euclidean distance. However, a way to take the corruption process μ_4 into account is to choose a distance measure which depends on keyboard distances. For instance, the distance between the two unary vectors $\overrightarrow{a}$ and $\overrightarrow{z}$ ought to be lower than the distance between $\overrightarrow{a}$ and $\overrightarrow{f}$ because, as you can see in figure 1.1 showing the Qwerty keyboard, $z \in N_1('a')$ and $f \in N_3('a')$.

We found two natural ways to build such a distance measure: use a covariance function, or build it by hand. The Euclidean distance between $\underline{v}^1$ and $\underline{v}^2$ is $(\underline{v}^1 - \underline{v}^2)^2$. Our distance will correspond to a generalisation of the Euclidean distance, the Mahalanobis distance defined by $(\underline{v}^1 - \underline{v}^2)^T \Gamma (\underline{v}^1 - \underline{v}^2)$, where Γ is a positive definite matrix. Our two approaches find suitable positive definite matrices Γ.

An automatic construction

The idea is to represent each key of the keyboard as a point in a 2-dimensional space, as in figure 3.1. Functions that automatically generate positive definite matrices are known as covariance functions ([4]). One choice is

$$\Gamma_{ij} = e^{-\lambda(\underline{X}^i - \underline{X}^j)^2}$$

where $\underline{X}^i$ is the point associated with the letter S_i. The parameter λ sets the length scale on the keyboard.

A handmade distance measure

Another approach to constructing a suitable distance is to begin with the identity matrix, which provides the Euclidean distance, and modify it in order to obtain the right distance between two unary vectors. Actually, keeping 1's on the diagonal of this matrix, the distance between the unary vectors $\overrightarrow{u_i}$ and $\overrightarrow{u_j}$ only depends on the coefficient $\Gamma_{i,j}$. If $\Gamma_{i,j}$ is null, the distance between these two vectors is equal to 2, as if the distance were Euclidean. If $\Gamma_{i,j}$ is equal to 0.5, the distance will be 1. We therefore set a value in the interval $[0, 0.5]$ according to the level of neighbourhood. The maximum number of levels of neighbourhood is 9 - for instance, $Q \in \widetilde{N_9}(P)$. Our definition of Γ_{ij} is therefore:

$$S_j \in \widetilde{N_p}(S_i),\ \Gamma_{ij} = 0.5 - 0.05(p-1)$$

Given the matrix Γ, we can easily show that the distance measure obtained satisfies:

$$S_j \in \widetilde{N_p}(S_i),\ d(S_i, S_j) = 1 + 0.1(p-1)$$

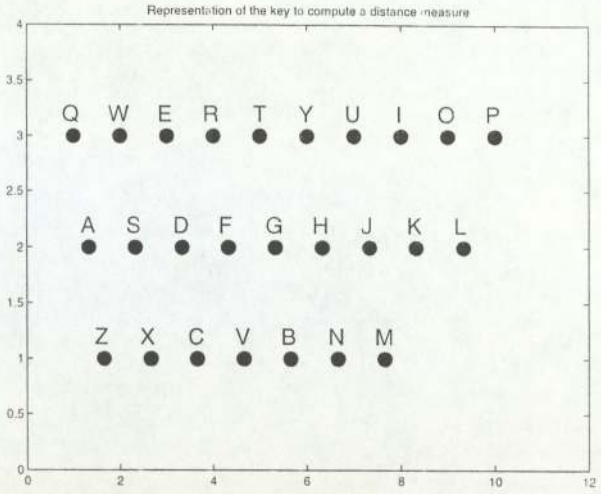


Figure 3.1: Position of the keys on a orthogonal plan

The distance varies from 1 ($d(Q, A)$) to 1.8 ($d(Q, P)$) and is in accordance with the real distance between the keys of the keyboard. Experiments using these two approaches showed that the handmade approach was slightly favourable over the covariance function, and we therefore decided to adopt this as the distance measure used throughout.

3.3 The batch K-Means clustering algorithm

There are 24 960 distinct frequency vectors in the dataset[1], the goal is to find a set of R centres, $K_1, \ldots, K_R$, which reflects the distribution of this points $\mathbf{v^n}$, $n = 1, \ldots, 24960$.

The idea of the algorithm is to find a partition of the data points which minimises the sum-of-squares clustering function given by

$$J = \sum_{j=1}^{R} \sum_{n \in j} Norm(\mathbf{v^n} - \mathbf{K_j})$$

where $\mathbf{K}_j$, $j = 1, \ldots, R$ is given by

$$\mathbf{K}_j = \frac{1}{N_j} \sum_{n \in j} \mathbf{v^n}$$

where N_j is the number of point $\mathbf{v^n}$ belonging to the cluster centre K_j.

To initialise the algorithm, we assign the data points randomly. We compute then the mean vectors of each cluster centre and update the membership of each data by minimising $(Norm(\mathbf{v^n} - \mathbf{K_j}))_\mathbf{j}$. This changes the memberships of clusters K_j. The iteration of these two steps decreases J, and we stop when this reduction becomes insignificant

[1] This is lower than the number of words in the dictionary (26057) since more than one word can have the same frequency representation.

1. Initialisation:

   For $1 \leqslant i \leqslant 24960$,
   $\mathbf{v^i}$ assigned randomly to cluster j

2. Iteration:

   While $|J - J_{old}| \geqslant 0.0001$
   $\mathbf{K_j} = \frac{1}{\mathbf{N_j}} \sum_{\mathbf{n} \in \mathbf{K_j}} \mathbf{v^i}$
   $\mathbf{v^i}$ is assigned to that $\mathbf{K_j}$ with least $Norm(\mathbf{v^i} - \mathbf{K_j})$
   $J = \sum_{j=1}^{R} \sum_{n \in \mathbf{K_j}} Norm(\mathbf{v^i} - \mathbf{K_j})$

3.4 Clustering experiments

We ran the K-means algorithm using the distance measure defined in section 3.2.1. In order to have on average 400 vectors in each cluster, we set the number of clusters R equal to 80.

Given an observation $\overrightarrow{O}$, we find a set of candidates by finding the closest cluster centre from $\overrightarrow{O}$ and then the closest vectors within this cluster.

Due to the possible corruptions, it is likely that the distance between the frequency vector $\overrightarrow{O}$ of the corrupted observation and the correct frequency vector $\overrightarrow{C}$ is large, and these two vectors might well be in different clusters. If this occurs, we will have no chance to correct the spelling error with the probabilistic approach described in chapter 2. In addition we need to limit the number of candidates to 200 because of the time of computation of the parameters of the Markov model. Two strategies could be adopted: either choose the 200 closest candidate vectors from the closest cluster centre, or choose the closest candidate vectors from the P closest cluster centres, $(K_1, K_2, \ldots, K_P)$. This second option allows for the possibility that the corruption process was severe enough to put the correct frequency vector and the observed frequency vector in different clusters. Indeed, after experimentation with corrupted words, we found that, very often, this solution is preferable to selecting only vectors in the nearest single cluster.

Figure 3.2 shows the general procedure: given an observation vector $\overrightarrow{O}$, we first select the closest cluster centres, (K_1, K_2, K_3, K_4) in the figure, and then select the closest vectors in each of them.[2]

[2] An alternative is to directly look for the closest vectors without considering the centres. However, computing 24960 distances takes too much time.

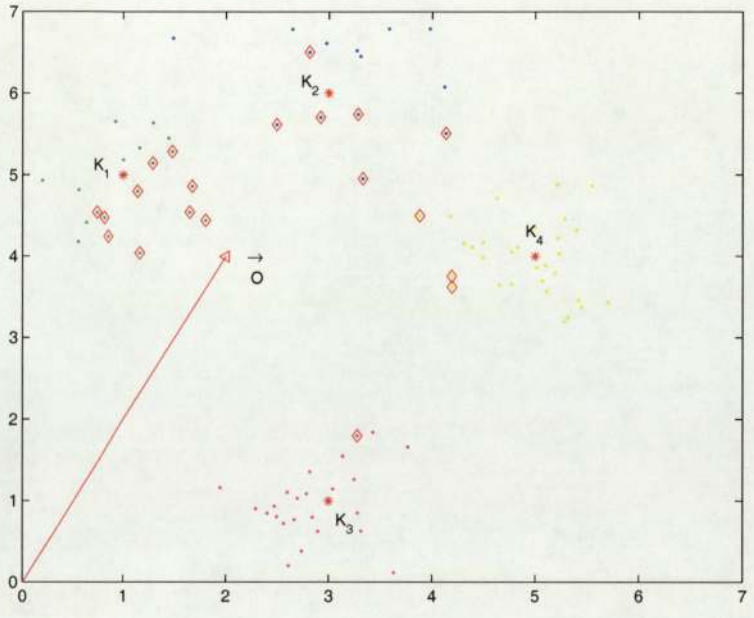


Figure 3.2: **Selection of candidate vectors.**
This 2 dimensional example shows 4 clusters centres (K_1, K_2, K_3, K_4) (red stars in the figure) and the selection of 10 closest vectors in K_1, 6 in K_2, 3 in K_3, and 1 in K_4 (these points are surrounded with red diamonds).

Chapter 4

Results and comparisons

4.1 Spellchecking Algorithm: ProbSpell

With the 80 clusters that the K-Means algorithm provides the probabilistic model, we have now all the tools to propose a practical way to correct words. At this stage, we do not deal with words formed from root words by addition of prefixes and suffixes. This will be dealt with in chapter 5. The algorithm is divided into two parts: the first is to select a set of candidates with the 80 cluster centres and then to use the Markov structure of the words to select the most probable. We call this algorithm "ProbSpell".

The correction of an observation is divided into 7 steps (see figure 4.1):

- **Step 1:** We first compute the frequency vector representation of the given observation.
- **Step 2:** We compare that vector with the 80 cluster centres obtained with the K-Means algorithm.
- **Step 3:** We select the P closest cluster centres from our data vector. In the diagram 4.1, P is equal to 4.
- **Step 4:** A list of vectors are assigned to these P clusters. In the p^{th} cluster, we select the Q_p closest vectors.
- **Step 5:** We now have P lists of vectors, each element in this list representing a set[1] of candidate words.
- **Step 6:** For each word, we compute a Markov chain as explained in section 2.3.
- **Step 7:** Finally, we calculate the probability $Pr(observation|\Theta_k)$ for all candidate models Θ_k. The most likely correction is given by the word associated to the model Θ_{opt} which maximises the likelihood. Along with the most likely word, a ranking of the next probable words is given, this is called the 'level' of returned corrections.

[1]Since more than one word can have the same frequency vector, given a vector, we obtain a set of words.

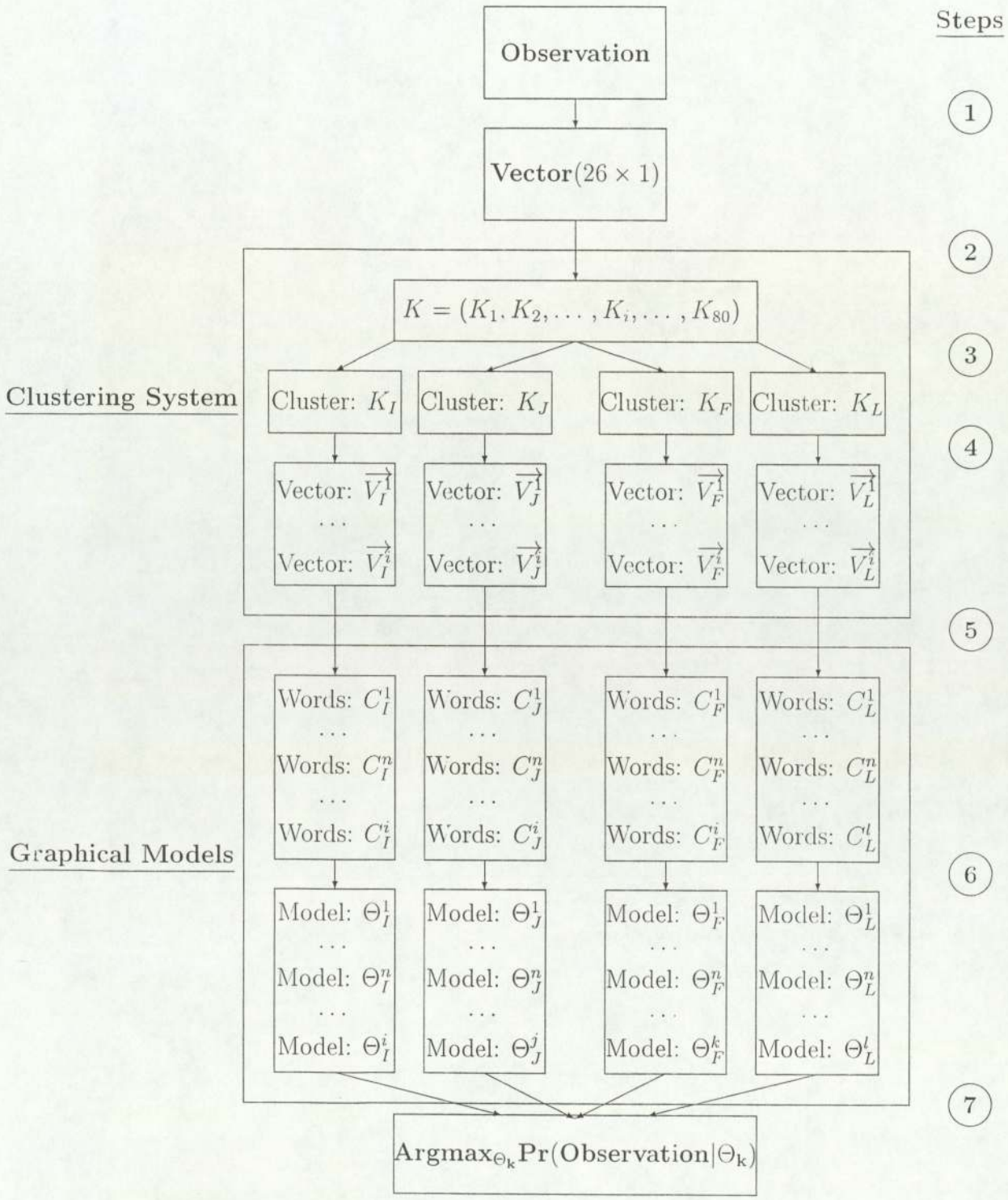


Figure 4.1: Algorithm of ProbSpell: the individual steps 1-7 are explained in the text.

4.2 Tests and results

Since we do not have a data set of mistakes typed by users, we artificially create test sets. We randomly choose words in the dictionary and apply to them a corruption function. We will then be able to check both the clustering approach, which provides a set of candidates, and the probabilistic approach which determines the most probable correct word. The results underline the importance of the length of the correct words. For this reason, we decided to separate the results according to word lengths.

In order to show that the algorithm is more or less coherent, we focus on the elementary corruption processes $(\mu_i)_{1 \leqslant i \leqslant 4}$ and begin by analysing simple corruptions: one deletion, or one addition or the transposition of a letter and/or the corruption of one character. Subsequently, we consider more complicated compositions of these corruption processes.

4.2.1 Basic corruption results

The goal of this test is to measure the efficiency of the clustering idea and of the probabilistic models. We consider 10 randomly chosen words of length 4, 5, ..., 10 and for each of them all the corruptions given by a single application of $(\mu_i)_{1 \leqslant i \leqslant 4}$ and the composition of μ_3 and μ_4 (see table 4.1). Therefore, given a dictionary word, the test set contains corruptions obtained by deleting any letter of the dictionary word, and swapping any two adjacent letters after substituting any letter with one in its first neighbourhood.

Length of words	4	5	6	7	8	9	10	Total
Number of words	10	10	10	10	10	10	10	70
Number of corruptions	2054	2740	3460	4352	5208	6199	9779	33792

Table 4.1: The data set of simple corruptions

In the experiments, we defined the model as follows:

- $P = 4$: the number of clusters centres selected
- $(Q_1, Q_2, Q_3, Q_4) = (50, 40, 30, 20)$: the number of vectors selected in each cluster
- Number of corrections: 3 and 10 (called also the level)

A small set of example words, corresponding corruptions and likely Corrections, is given in table 4.2. For each length of word, we compute the percentage of success of the clustering approach (i.e. when the correct word is assigned to one of the 4 cluster centres found in step 3 of the main algorithm) and the overall percentage of success of ProbSpell when the level is 3 and when the level is 10. We consider a corruption corrected when the word which generated it appears in the list returned by the algorithm. Figure 4.2 shows these percentages.

Since the search of the right cluster is crucial (otherwise, we have no chance to find the right word), we first focus on the clustering performance. The percentage of success of clustering is reasonable for the words of length 4, 5, 6, 8, and 10 (higher than

Corruption	Word	Corruption	Corrections	Probabilities
μ_1	flowery	flowrery	flowery	$0.8944 * 10^{-7}$
			forgery	$0.0456 * 10^{-7}$
			fernery	$0.0434 * 10^{-7}$
	hound	hounmd	hound	$0.6094 * 10^{-5}$
			bound	$0.0682 * 10^{-5}$
			bounds	$0.0667 * 10^{-5}$
	prior	pritor	prodigy	$0.3482 * 10^{-5}$
			profit	$0.3408 * 10^{-5}$
			parrot	$0.2004 * 10^{-5}$
μ_2	introduce	introuce	intrude	$0.1003 * 10^{-7}$
			inventor	$0.0706 * 10^{-6}$
			investor	$0.0202 * 10^{-6}$
	fortune	forune	fortune	$0.1139 * 10^{-4}$
			functor	$0.0094 * 10^{-4}$
			foundry	$0.0071 * 10^{-4}$
	drench	drenh	drench	$0.5641 * 10^{-4}$
			dean	$0.1047 * 10^{-4}$
			dreg	$0.0938 * 10^{-4}$
$\mu_3 o \mu_1$	existence	exitsemce	existence	$0.1439 * 10^{-8}$
			execute	$0.0251 * 10^{-8}$
			executor	$0.0120 * 10^{-8}$
	malfunction	malufnctoon	malfunction	$0.8816 * 10^{-11}$
			nonfunctional	$0.0596 * 10^{-11}$
			mispronounce	$0.0180 * 10^{-11}$
	shoemaker	shoemjaer	shoemaker	$0.2939 * 10^{-8}$
			shoemake	$0.0981 * 10^{-8}$
			shirtmake	$0.0147 * 10^{-8}$

Table 4.2: **Examples of words, simple corruptions and likely corrections**
For a corruption process and a dictionary word given, the third column presents an example of corruption. The fourth column displays the level 3 corrections returned by ProbSpell with the associated probabilities.

88%). The performance is slightly worse for length 7 words (82%). A likely reason for this is that length seven words are most numerous in the dictionary, and distinguishing between them is correspondingly difficult. The limitation of the cluster search is that some kinds of corruptions, especially deletion and insertion, move the observation far away from the correct cluster. For short words, the situation is not so severe since they are less numerous, so that length is almost enough to determine the correct cluster centre.

The percentage of success as a function of the length of the words is rather interesting. The longer the word is, the more information we have of its Markov structure. One or two corruptions will not radically affect too much this structure. For this reason,

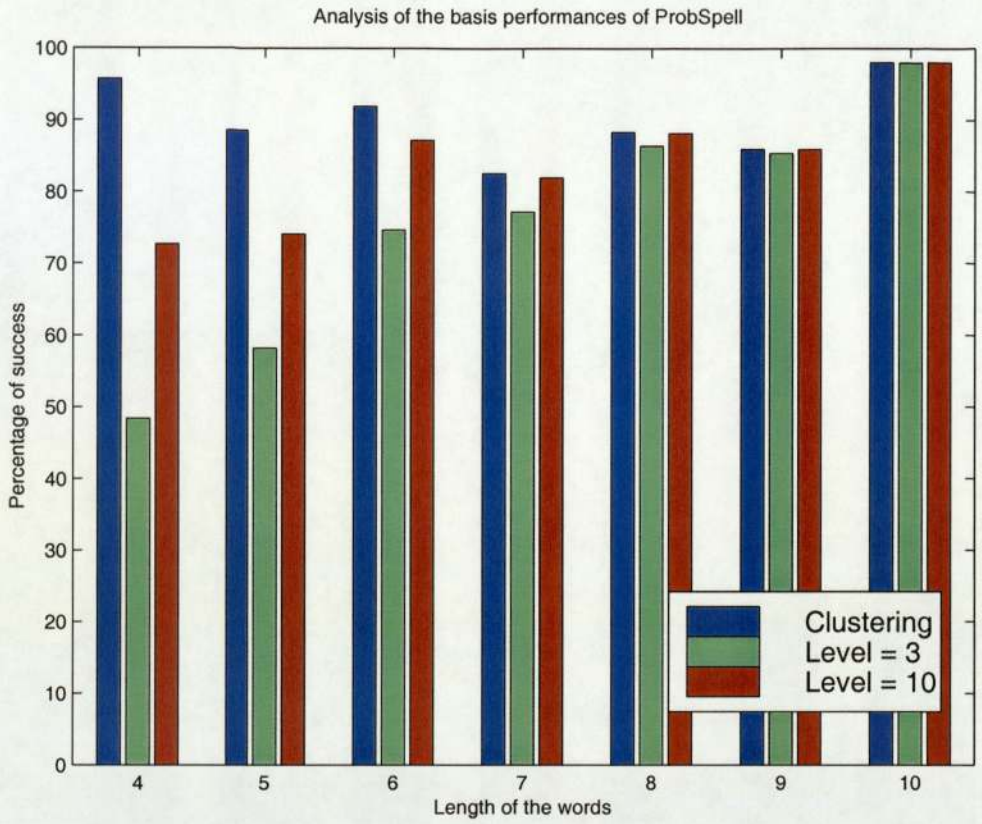


Figure 4.2: Basic tests of ProbSpell

the success for words longer than 8 is high: all of them are higher than 80% and the average per word is 92.06%. Moreover, we can notice that, for these types of words, returning 3 or 10 corrections gives roughly equal performance. This demonstrates that the models capture the structure of the words.

Unfortunately, the percentage of success decreases as the length of the word decreases. This phenomenon is understandable for 4 character letters (48.39% with 3 corrections and 72.69% with 10). With a transposition and a corruption of a letter, the observation can lose its Markov structure and can even be much closer to another word. For example, for the observation $'taut' = \mu_4(\mu_3('that', 2, 'u'), 3)$, we obtain the values:

$$\begin{aligned} Pr('taut'|\Theta('tail')) &= 1.3178 * 10^{-4} \\ Pr('taut'|\Theta('taint')) &= 1.2858 * 10^{-4} \\ Pr('taut'|\Theta('tag')) &= 8.5700 * 10^{-5} \\ Pr('taut'|\Theta('that')) &= 4.3208 * 10^{-5} \end{aligned}$$

This demonstrates that for short words it is feasible that the corruption process generates a corruption closer to a word in the dictionary different from the "correct" word. E.g. '*tail*' is more likely than '*that*'.

Comparison with Ispell on combined transition/mishit

These partial results are encouraging. But what happens when the corruption processes are more complicated? This section describes more complicated tests and a direct comparison with Ispell. As before, we present the results for different lengths of the correct word. We define a corruption complexity using the processes μ_3 and μ_4, representing the number of transpositions and the number of keyboard corruptions. Given a correct word C, we randomly select n positions (not necessarily distinct) and swap these letters with the previous ones. Similarly, we randomly select m letters and apply a nearest neighbourhood corruption. For each word length, we took 50 observations for each complexity (n, m) and each correct word. We then ran ProbSpell with the parameters $P = 6$ and $(Q_1, Q_2, Q_3, Q_4, Q_5, Q_6) = (50, 40, 30, 20, 10, 5)$ and also computed the success of Ispell. The results are shown in figure 4.3.

The first remark about these results is that Ispell is always less accurate than ProbSpell. That is not really a surprise: Ispell is deterministic and only considers the elementary processes, without combination of these processes. Therefore, the percentage of success for a complexity higher than (1,1) should be zero. However, since the choice of the letters corrupted and of the letters swapped is random, it can happen that the corruption process obtained is equivalent to a simpler process. For this reason, Ispell sometimes manages to correct an observation. This argument explains why very often, for two transpositions and a keyboard corruption, Ispell increases its rate of success: two flips can lead to the identity.

Let us have a look more carefully at ProbSpell's results. For long words, ProbSpell works well. The first diagrams in figure 4.3 show that, even if the complexity reaches (3,3) or (2,4), the percentage of success is higher than 65%. To understand the power of the spellchecking, consider the following example of an observation with 3 transpositions and 3 keyboard corruptions:

$$\begin{aligned} O &= \text{`}mipelmentsfinl\text{`} \\ &= \mu_3(\mu_3(\mu_3(\mu_4(\mu_4(\mu_4(\text{`}implementation\text{`}, 10.\text{`}s\text{`}), 11, \text{`}f\text{`}), 13, \text{`}l\text{`}).2), 5), 14) \end{aligned}$$

For that example, ProbSpell maximises the probability $Pr(O|\Theta(C))$ when C is "implementation", returning the correct result. Capturing the Markov structure for long words, ProbSpell has more than 77% of success when less than 4 corruption processes are used (i.e. for the complexities $(n, m)_{n+m\leqslant 4}$). The results are still good when the observations have 7 characters.

Unfortunately, for short words, the success goes down: the corruptions applied are too complicated and either the frequency vector is far away from the correct one (and so the correction is lost during the first step of the algorithm) or the Markov structure of the observation is too different to the correct one (and so, the probability given the model associated to the correct word is too low).

A Few examples of corrupted words and the likely correct words given by ProbSpell are presented in table 4.3.

Deletion and insertion of letters

At this point we consider the performance of ProbSpell for observations generated by corruption processes μ_1 and μ_2. This means that, generally, an observation and the

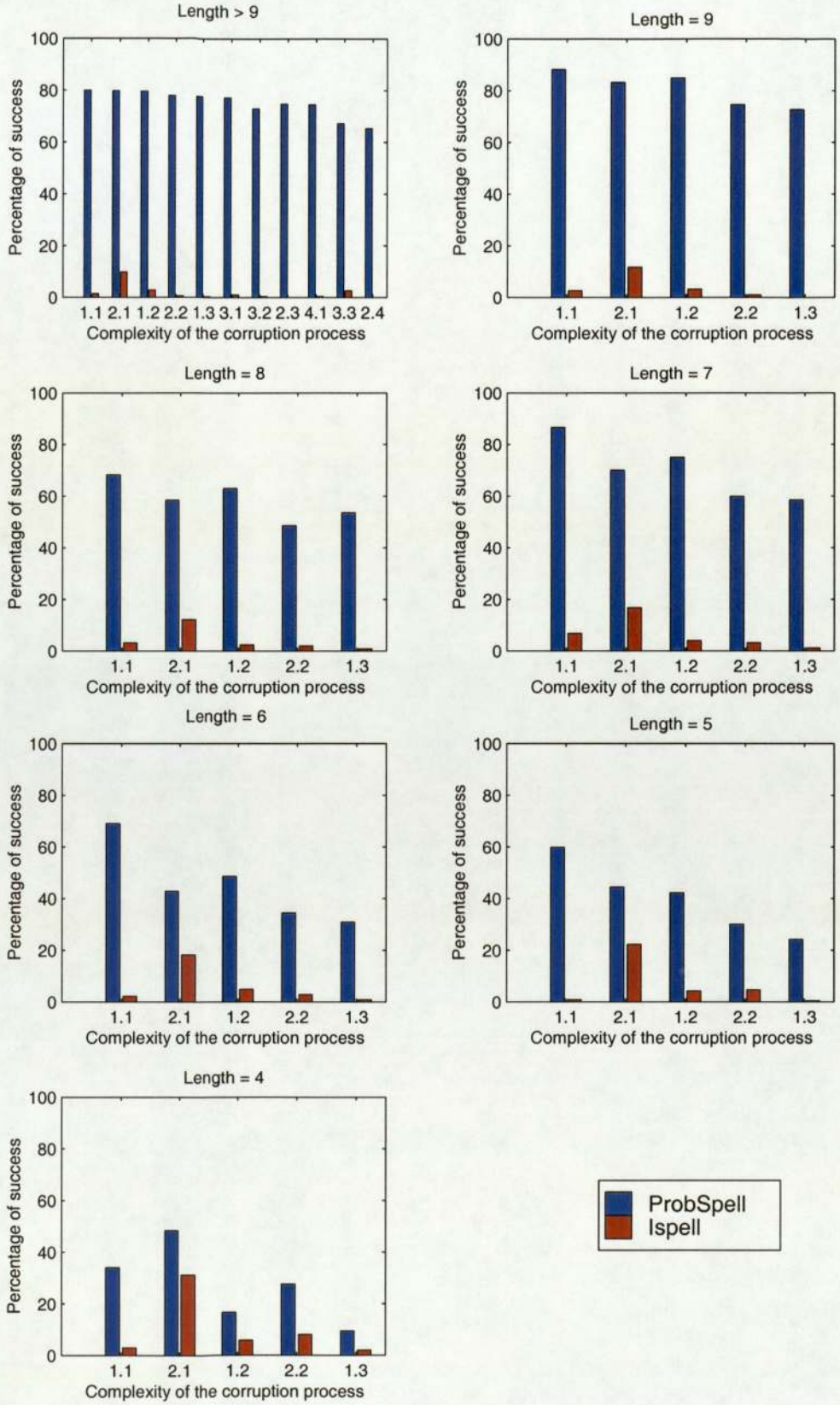


Figure 4.3: **Comparison with Ispell on combined transposition/mishit.** The corruption complexity is the number of transpositions and the number of mishits. These elementary processes are randomly applied to the correct word in order to obtain a test set. These results are for a level of 10.

Word	Corruption	Corrections	Probability
shameless	haxmepess	hammerless	$0.7777 * 10^{-9}$
		harmless	$0.5129 * 10^{-9}$
		harassment	$0.4242 * 10^{-9}$
coverable	cvsodable	coverable	$0.3828 * 10^{-9}$
		crossable	$0.2601 * 10^{-9}$
		catchable	$0.0841 * 10^{-9}$
sensibility	essnibuloty	sensibility	$0.1039 * 10^{-11}$
		feasibility	$0.0708 * 10^{-11}$
		syllabicity	$0.0047 * 10^{-11}$
elasticity	rlasticyjt	elasticity	$0.3392 * 10^{-10}$
		elativity	$0.2524 * 10^{-10}$
		relativist	$0.0325 * 10^{-10}$
incredulous	inrcdeuokus	incredulous	$0.2432 * 10^{-11}$
		incestuous	$0.0947 * 10^{-11}$
		industrious	$0.0424 * 10^{-11}$
misconstrue	miscsonrdue	misconstrue	$0.3061 * 10^{-11}$
		misspecified	$0.0112 * 10^{-11}$
		mischievous	$0.0070 * 10^{-11}$
mnemonically	jneomnicakyl	mnemonically	$0.2300 * 10^{-12}$
		mimetically	$0.0198 * 10^{-11}$
		incommunicado	$0.0079 * 10^{-11}$
backdrop	gcakdrpk	backdrop	$0.1899 * 10^{-9}$
		glacier	$0.1520 * 10^{-9}$
		grosbeak	$0.1240 * 10^{-9}$
balloon	valoono	balloon	$0.2894 * 10^{-6}$
		gallon	$0.0720 * 10^{-6}$
		ballyhoo	$0.0691 * 10^{-6}$

Table 4.3: **Examples of words, corruption with two transpositions and two mishits and likely corrections.**

correct word do not have the same length. As before, results will be compared with Ispell.

The data set is composed of 10 random words of each length (4 till 10). We take 300 observations for each correct word. Deletion of a single letter and a corruption of a single letter is first considered. In a second test, we add a random letter and corrupt another. As before, we obtain the percentage of correction of ProbSpell and Ispell. The results are shown in figure 4.4.

The first comment is that, once again, ProbSpell performs better than Ispell: this is not a surprise because the corruption process considered are too complicated for Ispell. Moreover, the percentages for ProbSpell generally grow with the length of the words and reach a maximum of 95.87% (for the deletions) and 95.97% (for the additions) for the longest words. Note that the results for deletions are worse than those of the additions (except for words of length 8). As pointed out in section 2.2.2, the Markov model overvalues deletion because the length of the observation is shorter. Therefore,

we can expect better results for deletion. A limitation of the clustering approach is that addition and especially deletion of letter can provoke a miss-classification within the 80 cluster centres.

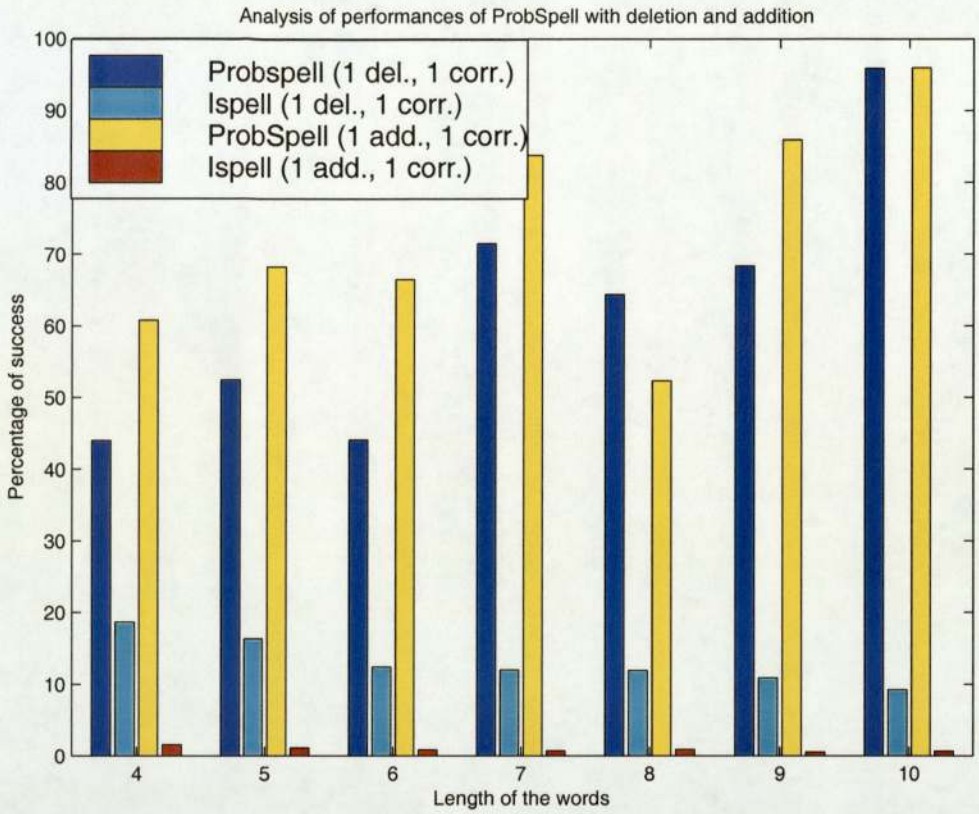


Figure 4.4: Test with deletions and additions of letters with a level of 10.

In conclusion, when the length of the observation and the length of the correct word are different, ProbSpell performs less well. The limitation of the clustering idea observed before on simple corruptions are again put in evidence with corruption processes μ_1 and μ_2.

4.3 Conclusion

For long words, both the clustering and the Markov models work well, to the extent that even quite severe corruptions can be corrected. For short words, the clustering approach is satisfying because of the relative small number of short words in the dictionary. However, the Markov model is less effective. The performance of ProbSpell is encouraging, far exceeding the performance of Ispell in corruptions involving a combination of the elementary corruption processes. However, the performance of Ispell is superior to that of ProbSpell, provided that only the very simplest corruptions are considered. This suggests that combining the two approaches - using Ispell to generate a set of candidate words for the ProbSpell algorithm or used exclusively for short words, could provide a powerful spellchecking system, able to deal very accurately with minor and more severe corruptions.

Chapter 5

Dealing with suffixes and prefixes

Being able to deal with suffixes and prefixes is an important aspect of any spellchecking system since their inclusion greatly expands the number of words in the dictionary. Associated to each word in the dictionary, there is an affix flag describing how to form another word. For example, in the case of the word *'dirty'*, several flags exist, one of which is described by:

- Name of the flag: T
- Last letter of the dictionary word: Y
- Set of the penultimate letters of the dictionary word: $\Omega_V - (A, E, I, O, U)$ (which means every character except *'a','e','i','o','u'*)
- List of letters deleted: Y
- List of letters added (end or beginning): $IEST$

This flag thus describes how to form the word *'dirtiest'* from the dictionary word *'dirty'*. New words are formed by paring the dictionary word *'dirty'* down to *'dirt'* and then adding a set of letters. Generally, the letters deleted and added to obtain the word with the suffix depend on the final and penultimate letters of the dictionary word. Let us consider how we could correct a corruption of the word *"multiplication"* which is actually the dictionary word *"multiply"* where the *'y'* is deleted and *'ication'* added. The corruption process can affect both the beginning of the word and the suffix and moreover, a swap may mix up the root with the suffix. For instance, instead of *"multiplication"*, the observation could be *"multipiclqtion"* (2 swaps and 1 keyboard corruption). To deal with such a mistake, given the observation, we need to find the dictionary word *"multiply"* and the suffix and then build a Markov chain model for the word *'multiplication'*. In the rest of this chapter, we principally consider suffixes since prefixes are less numerous (only three kinds: RE-, UN-, IN-) and pose exactly the same kinds of problem.

We consider three solutions, each of which can be integrated with the work done in chapter 4. The first idea is to use translated cluster centres, the second is to find the most likely dictionary word, and the final option is to build a Markov chain to guess which suffixes are most probable.

5.1 Three Approaches for dealing with suffixes

5.1.1 Vector translation

To explain the idea, we take an example of corruption of the correct word $C =$ *"dirtiest"*. Given, for instance, the observation $O =$ *"dortiets"*, the goal is to select K_n, the cluster centre of the dictionary word $W =$ *"dirty"* associated with the suffix *"iest"*. In order to do that, we can run through the set of suffixes, and for each one, delete these suffix letters from the observation, and add any letters required to obtain a word. E.g. for the suffix flag T, we would delete an *'i'*, *'e'*, *'s'* and *'t'* from the frequency vector representation of the word and add *'y'*. This is equivalent to translating the vector centres and the vector representation of the dictionary words. Finally, we hope that both the distances $d(O, K_n - \overrightarrow{\text{'y'}} + \overrightarrow{\text{'iest'}})$ and $d(O, W - \overrightarrow{\text{'y'}} + \overrightarrow{\text{'iest'}})$ are small enough to identify the correct cluster.

In doing this, when looking for the closest cluster centres, we do not care about the position of the letters added by the suffix. It is possible that, for instance, the letters *'i'*, *'e'*, *'s'* and *'t'* occur in an observation without the suffix: for instance, the observation "disaeter" could be typed instead of "disaster" (the key 'e' is a neighbour of 'r'). Considering the suffix flag T, after deleting the letters 'iest' and adding the letter *'y'*, the vector representation of the result *"daery"* is the same as that of the dictionary word *"ready"*. So, in that case, it is likely that the closest centre is going to be those of the dictionary word *"ready"* associated to the suffix flag T. This problem is especially evident when the suffix considered is short (e.g. addition of a single letter, *'s'*). Nevertheless, if we manage to select the correct cluster with the right suffix, the Markov structure provides a good way to select the most probable word in the list.

5.1.2 Deletion of the suffix

Let us say that the observation is a word with a suffix. The goal is to find, without any knowledge about the suffix flag used, the correct dictionary word which could generate the observation. Ignoring the number of letters added by the flag, we are going to delete one letter after another from the end of the observation and with each deletion, use ProbSpell to return the best corrections. The suffix flags associated to each word of the list of the corrections provide a list of candidate words. A Markov model for each of them and the maximisation of the probability $Pr(Observation|\Theta_k)$ provide a list of corrections with suffixes. A detailed example is shown in table 5.1.

Two main problems are raised with this approach: first, the number of deletions is unknown so, in order to consider all suffixes, we need to delete till a one-letter observation. Moreover, that means that ProbSpell is called as many times as the length of the observation typed, which is quite time consuming.

5.1.3 A Markov chain for the suffix

In total, there are 35 suffix flags in the database, each representing a distinct sequence of letters added at the end of the root. The idea here is to find first a list of likely suffixes and then apply the approach of section 5.1.1, but considering only the translation linked to the likely suffixes found. How can we find the most likely suffixes? In order to do this, we run through the set of suffixes and for each one substitute the last letters of the observation for the suffix. We then build a Markov chain with that sequence of letters and compute the probability $Pr(Observation|\Theta(\text{suffix}_i)$ where suffix_i is the observation altered by substituting the suffix. These values give an estimation of the likelihood of the suffix. For example, given the observation *"supsenidmg"*, the substitution of the last letters for each suffix provides a list of words: *"supsenided"* for the suffix *'ed"*, *"supseniing"* for *"ing"*, *"supsication"* for *"ication"*, etc. A Markov chain is built for each of them and then used to compute the probabilities $Pr(\text{"}supsenidmg\text{"}|\Theta(\text{"}supsenided\text{"}))$, $Pr(\text{"}supsenidmg\text{"}|\Theta(\text{"}supseniing\text{"}))$, etc. Once we have identified a list of likely suffixes, we apply the approach of section 5.1.1, translating the cluster centres appropriately to find the dictionary word generating the observation.

5.2 Test of these three solutions

In order to measure and compare the performances of each of these ideas, we need a data set. We considered five complexities of corruption processes and randomly selected 50 dictionary words with suffix flags, see table 5.2. One of these flags is randomly chosen and then 20 corruptions are computed according to the complexity of the process considered. There are therefore 1000 observations for each complexity.

We compared Ispell, vector translation, the deletion of letters, and the Markov modelling approach described before. Figure 5.1 displays the percentage of success for these four approaches. Analysing the behaviour of Ispell, we first remark that the deterministic approach is perfect for one transposition or one key corruption: Ispell reaches hundred percent for these two complexities. However, as soon as the corruption process is more complicated, the success of Ispell decreases rapidly: less than 20% for one transposition and one key corruption and less than 3% afterwards. Now, let us consider the results for the three suffix ideas for ProbSpell. Regardless of complexity, it seems that deletion of the suffix described in section 5.1.2 is the most effective: for elementary corruption processes μ_3 and μ_4, the rates are over 85% and even when the complexity is $(2, 2)$, 50% of the corruptions are solved. The determination of the most likely suffixes described in section 5.1.3 gives interesting results and improves the initial idea described in section 5.1.1.

Observations	Dictionary words found	Words derived	Probability
supsenidng	semipublic	**semipublic**	$\mathbf{1.5102 * 10^{-11}}$
	suspecting	suspecting	$3.0709 * 10^{-12}$
		unsuspecting	$1.5311 * 10^{-14}$
	supersonic	supersonic	$2.0195 * 10^{-12}$
		supersonics	$1.7143 * 10^{-12}$
	semidrying	semidrying	$1.8610 * 10^{-12}$
	supremacist	supremacist	$1.4098 * 10^{-12}$
	surpassing	surpassing	$1.0352 * 10^{-12}$
		surpassingly	$9.8727 * 10^{-13}$
supsenidm	semipublic	**semipublic**	$\mathbf{1.5102 * 10^{-11}}$
	surmise	**surmise**	$\mathbf{9.7155 * 10^{-12}}$
		surmised	$2.9059 * 10^{-12}$
		surmising	$9.6577 * 10^{-13}$
		surmiseier	$9.6577 * 10^{-12}$
		surmises	$7.1593 * 10^{-12}$
	suspend	suspend	$2.2019 * 10^{-12}$
		suspended	$1.5592 * 10^{-12}$
		suspending	$\mathbf{2.7321 * 10^{-11}}$
		suspender	$1.6536 * 10^{-12}$
		suspends	$1.5164 * 10^{-12}$
		suspenders	$1.5022 * 10^{-12}$
	supersonic	supersonic	$2.0195 * 10^{-12}$
		supersonics	$1.7143 * 10^{-12}$
	euphemism	euphemism	$7.6999 * 10^{-13}$
		euphemisms	$5.0045 * 10^{-13}$
	stupendous	stupendous	$8.6142 * 10^{-13}$
		stupendousness	$6.0848 * 10^{-13}$
		stupendously	$5.6507 * 10^{-13}$
supsenid	suspense	suspense	$2.1336 * 10^{-12}$
		suspension	$1.5531 * 10^{-12}$
		suspenses	$1.6811 * 10^{-12}$
		suspensive	$1.6294 * 10^{-12}$
		suspensions	$1.3675 * 10^{-12}$
	supersonic	supersonic	$2.0195 * 10^{-12}$
		supersonics	$1.7143 * 10^{-12}$
	supernova	supernova	$2.2199 * 10^{-12}$
		supernovas	$2.0278 * 10^{-12}$
	surmise	**surmise**	$\mathbf{9.7155 * 10^{-12}}$
		surmised	$2.9059 * 10^{-12}$
		surmising	$9.6577 * 10^{-13}$
		surmiseier	$5.2334 * 10^{-12}$
		surmises	$7.1593 * 10^{-12}$
	supposed	supposed	$2.8062 * 10^{-12}$
		supposedly	$2.0678 * 10^{-12}$
	snipped	snipped	$2.3455 * 10^{-12}$

Observations	Dictionary words found	Words derived	Probability
supseni	suppose	suppose	$3.1859 * 10^{-12}$
		supposed	$2.8062 * 10^{-12}$
		supposing	$4.2814 * 10^{-12}$
		supposier	$2.5624 * 10^{-12}$
		supposes	$2.3078 * 10^{-12}$
	suspense	suspense	$2.1336 * 10^{-12}$
		suspension	$1.5531 * 10^{-12}$
		suspenses	$1.6811 * 10^{-12}$
		suspensive	$1.6294 * 10^{-12}$
		suspensions	$1.3675 * 10^{-12}$
	shipped	shipped	$3.4552 * 10^{-12}$
	snippet	snippet	$1.9470 * 10^{-12}$
	snipper	snipper	$1.9470 * 10^{-12}$
		snippers	$2.3455 * 10^{-12}$
	snipped	snipped	$2.3455 * 10^{-12}$
supsen	suspense	suspense	$2.1336 * 10^{-12}$
		suspension	$1.5531 * 10^{-12}$
		suspenses	$1.6811 * 10^{-12}$
		suspensive	$1.6294 * 10^{-12}$
		suspensions	$1.3675 * 10^{-12}$
	stipend	stipend	$1.1268 * 10^{-12}$
		stipends	$7.5839 * 10^{-13}$
	sensual	sensual	$5.5019 * 10^{-13}$
		sensually	$3.9736 * 10^{-13}$
	superb	superb	$8.5038 * 10^{-13}$
		superbness	$6.7389 * 10^{-13}$
		superbly	$6.9632 * 10^{-13}$
	spurn	spurn	$9.0207 * 10^{-13}$
		spurned	$2.9974 * 10^{-13}$
		spurning	$\mathbf{1.0520 * 10^{-11}}$
		spurner	$2.2461 * 10^{-13}$
		spurns	$5.4893 * 10^{-13}$

Table 5.1: **The deletion of letter for the observation *"supsenidmg"*.** Given the observation *"supsenidmg"* typed by the user (instead of *"suspending"*), ProbSpell is first run over that observation in order to find the most probable dictionary words. This is also done with the previous observation without the last letter (second line), then without the two last letters (third line) and so on. The second column shows the 6 corrections proposed by ProbSpell for each corruption. The third column is the list of words derived from the dictionary words found according the suffix flags the dictionary words have. Finally, the computation of the probability $Pr(\text{"}supsenidmg\text{"}|\Theta_k)$ for all model Θ_k associated to each words derived provided the most likely correct word. We remark that, in that example, the correct word *"suspending"* has the highest value. $2.7321 * 10^{-11}$. The 5 highest probabilities are highlighted.

Number of words	50	50	50	50	50
Number of corruptions	20	20	20	20	20
Number of swaps	1	0	1	2	2
Number of key corruptions	0	1	1	1	2

Table 5.2: The data set of observations with suffix

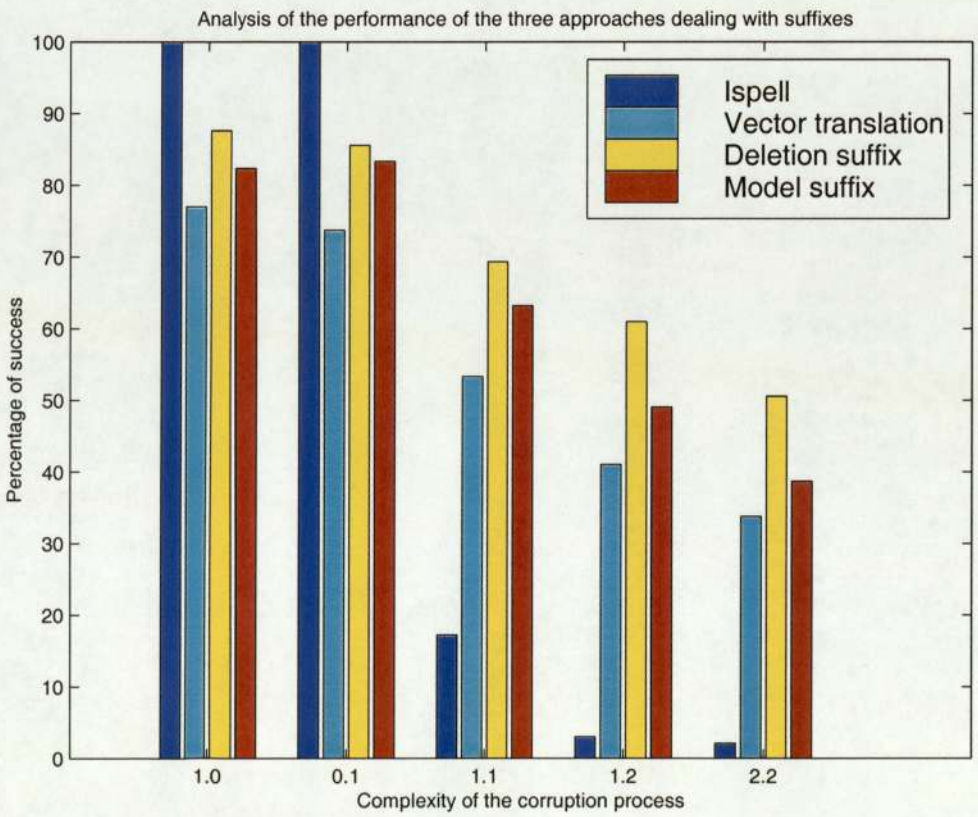


Figure 5.1: Comparison of the three approaches and Ispell

Chapter 6

Mixture of Markov models for clustering

6.1 Introduction

The algorithm proposed in the section 4.1 to correct spelling errors works quite well: given an observation, the clustering scheme rapidly provides a list of candidates, and the probabilistic model usually manages to select the correct word amongst the candidates. However, if the correct word is not identified by the clustering procedure, the method fails, regardless of the subsequent probabilistic approach. The experiments in chapter 4 demonstrate that the weakness of the current method is primarily in the clustering procedure - provided the correct cluster is identified, it is rare to fail to find the correct word. The reason, presumably, that the clustering fails, is that it does not take into account enough of the corruption processes, particularly deletion and insertion. Indeed, the artificial distance measure can lead to some problems of miss-classification: with the current algorithm, a corruption of a correct word could roughly keep its original Markov structure and at the same time lose its frequency vector representation characteristics. This occurs, for example, when the observation has many letters removed from the end of the correct word. This suggests that a method of clustering, based on the same principles as the Markov model approach, should be more appropriate. One possibility is to replace the distance measure based on the keyboard by a probabilistic measure based on the Markov structure of words. That is, we could attempt to cluster words based on their similarity in terms of their letter to letter transitions such that all members of particular cluster would have a similar Markov structure. Each cluster is governed by a Markov model with parameter Θ_k (the initial state probabilities and the transition matrix). This model provides a probabilistic generative model for the sequences of that group. In the following section, the EM algorithm applied to the clustering problem is fully detailed.

6.2 Clustering with a mixture of models

In general, a probabilistic interpretation of any clustering procedure is that it is mixture model. For example, the K-means algorithm is a deterministic version of Gaussian mixture models. This suggests that we can cluster data based on their Markov similarity, by simply using a mixture of Markov models ([2],[14]). In chapter 2, we associate to a word of the dictionary a Markov chain, which represents the letter to letter transition of the original word, incorporating the elementary corruptions. Since we want to include the corruption processes in the clustering, a first idea is to build a Markov chain for each word of the dictionary. We can interpret this as a mixture model, with 26057 components. To cluster these models, we could try to fit this set of models with a mixture model with a greatly reduced number of components.

Formally, the problem is written as:

- $p^g(\underline{x})$: the original given model (a 26057 component mixture model)
- $p^m(\underline{x})$: the mixture of models we will use to fit to $p^g(\underline{x})$

where,

$$p^m(\underline{x}) = \sum_k p(\underline{x}|k)p(k)$$

$p(k)$ are the mixture coefficients and $p(\underline{x}|k)$ the k^{th} mixture model, corresponding to the k^{th} cluster. The KL divergence between the original and the mixture distributions can be used in order to fit the mixture model:

$$KL(p^g(\underline{x}), p^m(\underline{x})) = \sum_{\underline{x}} p^g(\underline{x}) \log(p^g(\underline{x})) - \sum_{\underline{x}} p^g(\underline{x}) \log(p^m(\underline{x})) \tag{6.1}$$

The goal is to find the parameters of the distribution p^m which minimise this KL divergence. Since, p^m only appears in the second term of equation 6.1, we need to maximise the energy E,

$$\begin{aligned} E &= \sum_{\underline{x}} p^g(\underline{x}) \log(p^m(\underline{x})) & (6.2)\\ &\equiv \langle \log p^m(\underline{x}) \rangle_{p^g}, & (6.3) \end{aligned}$$

with respect to the parameters of p^m. We introduce an arbitrary distribution $q(k|\underline{x})$ and consider the KL divergence between $q(k|\underline{x})$ and $p(k|\underline{x})$:

$$\begin{aligned} KL(q(k|\underline{x}), p(k|\underline{x})) &= \sum_k q(k|\underline{x}) \log q(k|\underline{x}) - \sum_k q(k|\underline{x}) \log p(k|\underline{x}) & (6.4)\\ &= \sum_k q(k|\underline{x}) \log q(k|\underline{x}) - \sum_k q(k|\underline{x}) \log \frac{p(\underline{x}|k)p(k)}{p^m(\underline{x})} & (6.5)\\ &\geqslant 0 & (6.6) \end{aligned}$$

Isolating the term $\log p^m(\underline{x})$ in equation 6.5, the inequality 6.6 provides an lower bound on the log likelihood given by

$$\begin{aligned} \log p^m(\underline{x}) \geqslant & -\sum_k q(k|\underline{x}) \log q(k|\underline{x}) + \sum_k q(k|\underline{x}) \log p(\underline{x}|k) \\ & + \sum_k q(k|\underline{x}) \log p(k) & (6.7) \end{aligned}$$

Equations 6.3 and 6.7 leads to an lower bound on the energy

$$\begin{aligned} E \geqslant & -\sum_k \langle q(k|\underline{x}) \log q(k|\underline{x}) \rangle_{p^g} + \sum_k \langle q(k|\underline{x}) \log p(\underline{x}|k) \rangle_{p^g} \\ & + \sum_k \langle q(k|\underline{x}) \rangle_{p^g} \log p(k) \end{aligned} \tag{6.8}$$

We assume that the distribution $q(k|\underline{x})$ is fixed. The parameters of the mixture of models we want to fit appear in the second and the third term in equation 6.8: the prior and the transition matrix of cluster k in the terms $\log p(\underline{x}|k)$ and the mixture coefficients in the term $\log p(k)$. Let us first consider the optimisation of the components. We introduce the distribution $\widetilde{q}(k)$ given by

$$\widetilde{q}(k) \equiv \langle q(k|\underline{x}) \rangle_{p^g}$$

The third term of equation 6.8 can be written as

$$\sum_k \widetilde{q}(k) \log p(k) = KL(\widetilde{q}(k), p(k)) - \sum_k \widetilde{q}(k) \log \widetilde{q}(k) \tag{6.9}$$

Therefore, we obtain immediately the optimal choice of the mixture components given by:

$$\begin{aligned} p(k) &= \widetilde{q}(k) \\ &= \langle q(k|\underline{x}) \rangle_{p^g} \end{aligned} \tag{6.10}$$

Let us focus now on the second term of equation 6.8 which can be written as:

$$\sum_k \langle q(k|\underline{x}) \log p(\underline{x}|k) \rangle_{p^g} = \sum_k \sum_{\underline{x}} q(k|\underline{x}) p^g(\underline{x}) \log p(\underline{x}|k) \tag{6.11}$$

$$\propto \sum_k \sum_{\underline{x}} r(\underline{x}|k) \log p(\underline{x}|k) \tag{6.12}$$

where

$$r(\underline{x}|k)) \propto q(k|\underline{x}) p^g(\underline{x}) \tag{6.13}$$

In addition, since the particular cluster centre k follows a Markov distribution, we can write $p(\underline{x}|k)$ as:

$$p(\underline{x}|k) = \prod_i \pi_i^{d_i(\underline{x})} \prod_{ij} a_{ij}^{f_{ij}(\underline{x})} \tag{6.14}$$

where the prior π_i and transition matrix a_{ij} are the parameters Θ_k of cluster k. $d_i(\underline{x})$ is the count of initial state i in $\underline{x}$. Similarly, $f_{ij}(\underline{x})$ is the count of transitions $i \rightarrowtail j$ in $\underline{x}$. Since our goal is to maximise equation 6.12 with respect to the cluster parameters, let us consider now the Lagrangian function with the Lagrange multipliers λ_l and β associated to the normalisation constraints of the parameters:

$$L(\Theta_k, \{\lambda_l\}, \beta) \equiv \sum_{\underline{x}} r(\underline{x}|k) \log(p(\underline{x}|k)) - \sum_l \lambda_l (\sum_j a_{lj} - 1) - \beta (\sum_i \pi_i - 1) \tag{6.15}$$

Substituting equation 6.14 in equation 6.15, we obtain

$$L(\Theta_k, \{\lambda_l\}, \beta) = \sum_{\underline{x}} \{\sum_i r(\underline{x}|k) d_i(\underline{x}) \log \pi_i + \sum_{ij} r(\underline{x}|k) f_{ij}(\underline{x}) \log a_{ij}\} \\ \sum_l \lambda_l (\sum_j a_{lj} - 1) - \beta(\sum_i \pi_i - 1) \tag{6.16}$$

The conditions for 6.16 to be stationary with respect to the parameter Θ_k, λ_l and β give the following equations

$$\frac{\partial L}{\partial a_{ij}} = 0 \Leftrightarrow \quad a_{ij} \propto \sum_{\underline{x}} r(\underline{x}|k) f_{ij}(\underline{x}) \qquad \text{with} \sum_j a_{ij} = 1 \tag{6.17}$$

$$\frac{\partial L}{\partial \pi_i} = 0 \Leftrightarrow \quad \pi_i \propto \sum_{\underline{x}} r(\underline{x}|k) d_i(\underline{x}) \qquad \text{with} \sum_i d_i = 1 \tag{6.18}$$

So, given the distribution $q(k|\underline{x})$, for a sample drawn from $p^g(\underline{x})$, the computation of the average frequency of transition $i \rightarrowtail j$, weighted by $q(k|\underline{x})$, i.e. $\langle q(k|\underline{x}) f_{ij} \rangle_{p^g}$, and normalised provides the optimal transition matrices. This update of the parameters ensures the upper bound computed in equation 6.8 is maximised with respect of the parameters. Asserting $q(k|\underline{x}) = p(k|\underline{x})$, defines a closed set of equations that are guaranteed to decrease the KL divergence bound between the models p^g and p^m. This is in fact an EM algorithm in which the E step is given by $q(k|\underline{x}) = p(k|\underline{x})$ and the M step is given by equations 6.10, 6.17 and 6.18. We remark that the mixture components (equation 6.10) are the expected average proportion of the sample drawn from p^g in each cluster.

In order to compute the optimal parameters, we need to take a sample from the distribution p^g. In other words, for each Markov model associated to a dictionary word, a set of samples is necessary for the computation of the optimal mixture model parameters. This would be a very costly procedure. Indeed, rather than fitting our model p^m to samples from model p^g, it would seem more appropriate to replace the Markov model mixture p^g by $p^g(\underline{x}) = \sum_{\nu=1}^{N} \delta(\underline{x} - \underline{x}^\nu))$ where $\underline{x}^\nu$ are examples of corrupted words from the dictionary. In that case, the E step and M step become

$$\textbf{E step} \quad p(k|\underline{x}) = \frac{p(\underline{x}|k) p(k)}{\sum_k p(\underline{x}|k) p(k)} \tag{6.19}$$

$$\textbf{M step} \quad p(k) = \frac{1}{N} \sum_{\nu}^{N} p(k|\underline{x}^\nu)) \tag{6.20}$$

$$\pi_i^k = \frac{\sum_\nu^N p(k|\underline{x}^\nu) d_i(\underline{x}^\nu)}{\sum_\nu^N p(k|\underline{x}^\nu)} \tag{6.21}$$

$$a_{ij}^k = \frac{\sum_\nu^N p(k|\underline{x}^\nu) f_{ij}(\underline{x}^\nu)}{\sum_\nu^N p(k|\underline{x}^\nu)} \tag{6.22}$$

6.3 Application to the data

Instead of applying the algorithm described in section 6.2 over all the words of the dictionary, we will take a simple data set composed of 1000 randomly chosen dictionary

words. Moreover, this current section will not consider any kind of corruption, each observation being a dictionary word. The experiment consists of fitting of a mixture of 10 Markov models to the 1000 words. In addition, the K-means algorithm described in section 3.3 is run in order to compare the two approaches. The 10 mixture coefficients found after 20 iterations are between 0.0749 and 0.1138. Since these probabilities represent the proportions of data in each cluster, the 1000 data are evenly distributed between the clusters. Let us consider the two first clusters which contains 75 and 90 words. Figure 6.1 shows the transition matrices and the priors of these clusters. In table 6.1 we have selected a few words from each of these two clusters, and also a few words from the first cluster found by the K means approach.

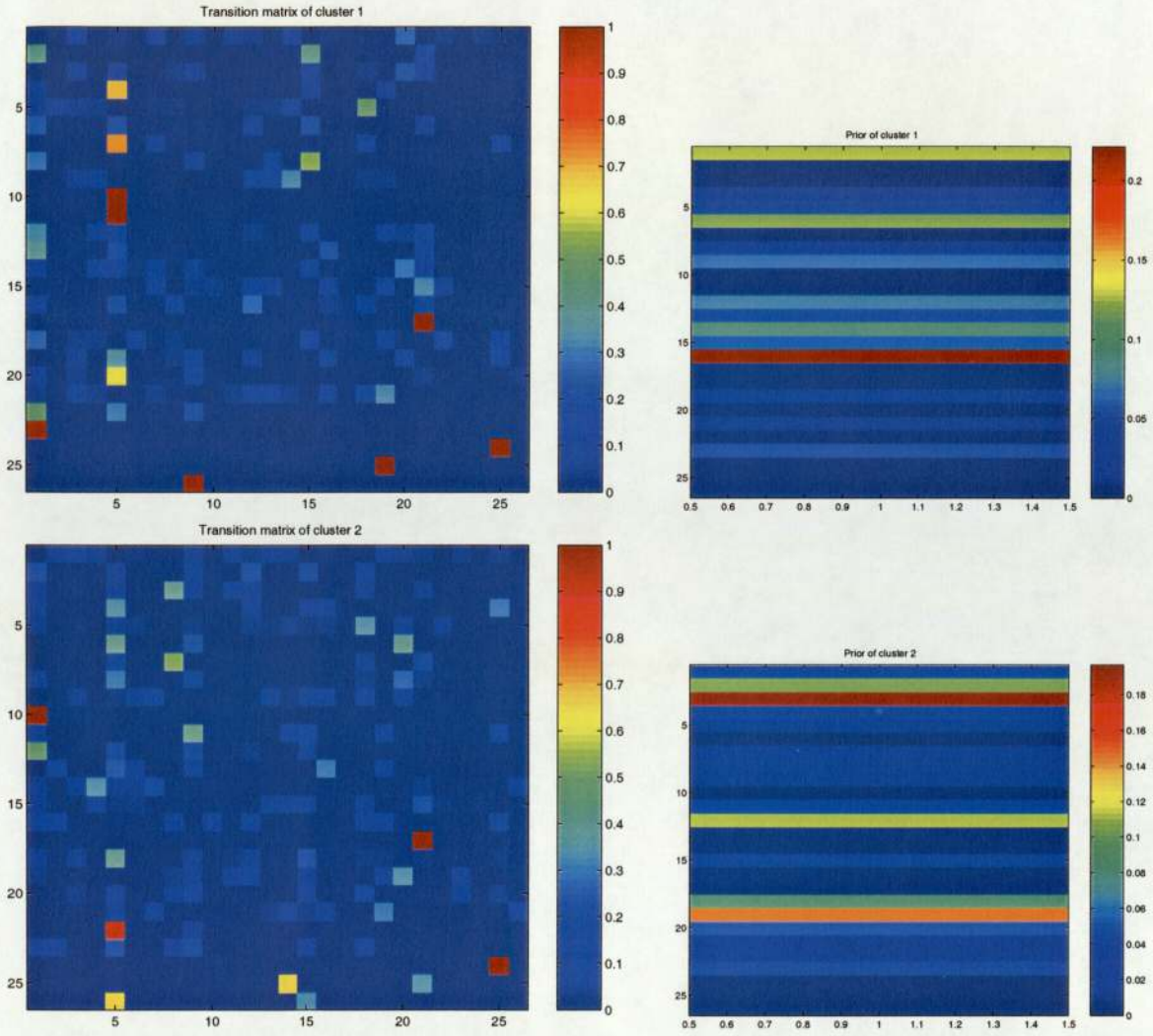


Figure 6.1: Parameters of the Markov models of two first clusters

Observing the transition matrix of cluster 1, we note that a few transitions are equal to 1 (e.g. $Pr(\text{`a'}|\text{`w'})$, $Pr(\text{`e'}|\text{`j'})$, $Pr(\text{`e'}|\text{`k'})$, $Pr(\text{`u'}|\text{`q'})$, $Pr(\text{`y'}|\text{`x'})$, $Pr(\text{`s'}|\text{`y'})$, $Pr(\text{`i'}|\text{`z'})$). However, observing the words classified in that first cluster, it is difficult to distinguish the features of the cluster except for rare sequences of letters such as *'wa'* or *'xy'*. That cluster is therefore mainly characterised by these rare combinations and by the prior: for instance, cluster one has the highest prior probability for the

letter 'p' (π_1('p') = 0.2215) whereas none of words beginning with that letter is in cluster two. Reciprocally, the prior of cluster two has highest values for the letters 'c' (π_2('p') = 0.1969) and 's' (π_2('p') = 0.1448) whereas these values are very low for the first cluster. In comparison with the K means clustering approach described in chapter

	Cluster model 1	Cluster model 2	Cluster 1 (K-means)
Number of words	75	90	57
Examples	'placebo'	'chimpanzee'	'marketplace'
	'servile'	'chip'	'transferral'
	'plane'	'chit'	'irreplaceable'
	'plate'	'stereo'	'inaccurate'
	'nirvana'	'stood'	'heartbreak'
	'plug'	'round'	'supernatant'
	'pluperfect'	'storeroom'	'lawbreaking'
	'federate'	'chickadee'	'fluctuate'

Table 6.1: Two clusters from the mixture and one obtained with the K-means algorithm

3 based on the on the frequency of letters in the word, we observe that the length of the words do not play as strong a role in the Markov clustering process. Even if the interpretation of the Markov clusters is rather difficult, very rare combinations clearly have too much importance in the computation of the parameters of the mixture. If a corruption of a word were to affect a rare letter combination, it is quite possible that the observation and its associated dictionary word would not be in the same cluster.

In clustering the words, we really need to incorporate the elementary corruption processes. A way to do that is to apply the approach described in section 6.2, generating a list of corruptions for each dictionary word. However, in order to train accurately the mixture model, we would need a very large number of corruptions for each dictionary word. With current computational resources, this approach is not feasible.

6.3.1 Possible solutions

We can propose possible methods which can provide a way to cluster dictionary words according to their Markov structure and the elementary corruption process, without generating a huge list of corruptions. One method would be to fit the mixture model with respect to the correct letter to letter transition of the training set. It is possible to take into account the corruption process μ_4 by adding the confusion matrix computed in section 2.2 to obtain a mixture of HMMs. This is rather limited since it does not consider the deletion and the transposition of letters. However, it is possible to improve this by adjusting by hand the letter to letter probabilities of each component after training. These adjustments would consist of adding weights according to the probable sequences as we did in chapter 2 when we built Markov models for each dictionary word. Because of a lack of time, this last idea has not been tested.

6.4 Conclusion

Clustering models requires samples from the models. There is little benefit in doing this over simply taking examples of corrupted words.

Computationally, this could be very expensive, since we would require a great many samples to encode accurately the corruption processes. We did not have sufficient time to fully explore a possible solution to this problem, which could be based on training the mixture models on uncorrupted data, and adjusting a posteriori the parameters of the mixture model to account for the corruption processes.

At this stage, therefore, it remains unclear whether or not clustering based on a Markov structure would lead to an improvement over the K means approach.

Chapter 7

Conclusions

Spellchecking is a difficult problem. The reason for this difficulty is that there is potentially a very large number of corruptions for any word, so that any deterministic approach will rapidly run into computational difficulties in dealing with all but the simplest corruptions. For this reason we considered a probabilistic approach. Our approach is based on the idea that the Markov letter to letter transition structure of both the correct word and the corrupted word should be sufficiently similar in order to be able to identify the correct word. Our experiments bear out the usefulness of this approach, particularly for long words. For short words, the Markov structure can be quite severely affected by even a modest corruption process. The clustering procedure that we used to rapidly identify a set of candidate words is largely successful, but constitutes the major contribution to the percentage of error. It seems to us, therefore, those better clustering procedures, similar to the mixture of Markov models, may provide, ultimately, a more satisfactory approach.

A source of difficulty is prefixes and suffixes. We have introduced three approaches for dealing with this issue. All of them used the Markov framework to firstly identify either the suffix added or the root of the observation. However, the results of these three approaches demonstrate a clear reduction of percentage performance in comparison with the correction of non-suffix/prefix dictionary words. Even simple corruptions can make the task harder since they can affect both the root and the suffix. In that case it becomes difficult to even detect the current suffix. An alternative would be to simply apply ProbSpell with a dictionary of 60000 words containing all the words of English language, including suffixes and prefixes. Computational resources currently have prevented us from pursuing this approach. We can also think of using a second order Markov model to describe a word. This would provide a way to distinguish the deletion of a letter and the transposition of two letters. It could also be interesting to consider more parameters to correct spelling errors such as the neighbourhood extended with associated probabilities, the frequency or the speed of typing.

In conclusion, spellchecking is an old but relevant problem which can indisputably be improved by using probabilistic framework. However, this improvement is limited by the complexity of the corruption processes. When it becomes too complicated so that the observation and the intended dictionary word have little in common, we need to consider a whole sentence and a grammar to have a chance to guess the correct word.

Bibliography

[1] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[2] I. Cadez, S. Gaffney, and P. Smyth. A generalist probabilistic framework for clustering individuals. Technical report, Department of Information and Computer Science, University of California, Irvine, March 2000.

[3] K. W. Church and W. A. Gale. Probability scoring for spelling correction. In *Statistics and Computing*, volume 1, pages 93–103. 1991.

[4] N.A.C. Cressie. *Statistics for Spatial Data.* Wiley, New York, 1993.

[5] R. Dugad and U.B. Desai. A tutorial on hidden markov models. Technical report, Signal Processing and Artificial Neural Networks Laboratory, Department of Electrical Engineering, Indian Institute of Technology, Bombay, May 1996.

[6] Z. Ghahramani. Learning dynamic bayesian networks. In C.L. Giles and M. Gori, editors, *Adaptive Processing of Temporal Information.* Springer-Verlag, October 1997.

[7] A. R. Golding and D. Roth. A winnow-based approach to context-sensitive correction. *Natural Language Learning*, 1999.

[8] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. *Learning Graphical Models*, chapter An introduction to variational methods for graphical models, pages 105–162. Kluwer Academic Publishers, 1998.

[9] Mark D. Kernighan, Kenneth W. Church, and William A. Gale. A spelling correction program based on a noisy channel model. In Hans Karlgren, editor, *COLING-90 13th International Conference of Computational Linguistics*, volume 2, pages 205–210, Helsinki University, 1990.

[10] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, pages 129–137, 1982.

[11] Eric Mays, Fred J. Damerau, and Robert L. Mercer. Context based spelling correction. *Information Processing and Management.* 27(5):517–522, 1991.

[12] James L. Peterson. A note on indetected typing errors. *Communications of the ACM*, 29(7):633–637, July 1986.

[13] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Mag.*, pages 4–16. June 1986.

[14] Padhraic Smyth. Clustering sequences with hidden markov models. Technical report, Information and Computer Scinece, University of California, Irvine.

[15] David Yarowsky. Decision lists for lexical ambiguity resolution: Application to accent restoration in spanish and french. In Las Cruces, editor, *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*. pages 88–95, 1994.

Appendix A

Main hidden Markov models algorithms

$$
\begin{aligned}
T &= \textit{length of the sequence of observations} \\
N &= \textit{number of states} \\
M &= \textit{number of possible observations} \\
\Omega_q &= q_1, ...q_N \ (\textit{finite set of possible states}) \\
\Omega_O &= V_1, ..., V_M \ (\textit{finite set of possible observations}) \\
S_t &= \textit{random variable denoting the state at time } t \ (\textit{state variable}) \\
O_t &= \textit{random variable denoting the observation at time } t \ (\textit{output variable}) \\
\sigma &= O_1, ..., O_T \ (\textit{sequence of actual observations}) \\
\pi_i &= Pr(q_1 = S_i) \\
A_{ij} &= Pr(q_{t+1} = S_j | q_t = S_i) \\
B_j(O_t) &= Pr(V_k = O_t \textit{ at } t | q_t = S_j)
\end{aligned}
$$

A.1 The Forwards-Backwards algorithm

Goal: To compute $Pr(\sigma|\Theta)$

1. Definitions

$$\alpha_t(j) = Pr(O_1 = y_1, \ldots, O_t = y_t, S_t = q_j|\Theta)$$
$$\beta_t(i) = Pr(O_{t+1} = y_{t+1}, \ldots, O_T = y_T|S_t = q_i, \Theta)$$

2. Algorithm

a - the backward coefficients:

$$\alpha_1(i) = \pi(i) * b_i(o_1)$$
$$\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i)a_{ij}]b_j(o_{t+1})$$

b - the forward coefficients:

$$\beta_T(i) = 1$$
$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}b_j(o_{t+1})\beta_{t+1}(j)$$

c - Computation of $Pr(\sigma|\Theta)$:
$P(\sigma|\Theta) = \sum_{i=1}^{N} \alpha_T(i)$

A.2 The EM algorithm

Goal: adjust Θ to maximise $Pr(O|\Theta)$

$$\zeta(\Theta) = \log \sum_{X} P(Y, X|\Theta)$$

To maximise the log likelihood, the idea is to introduce a arbitrary distribution Q defined over the hidden variables, we can find out a lower bound of the likelihood:

$$\begin{aligned}
\log \sum_{X} P(Y, X|\Theta) &= \log \sum_{X} Q(X) \frac{P(Y, X|\Theta)}{Q(X)} && \text{(A.1)} \\
&\geqslant \sum_{X} Q(X) \log \frac{P(Y, X|\Theta)}{Q(X)} && \text{(A.2)} \\
&= \sum_{X} Q(X) \log P(Y, X|\Theta) - \sum_{X} Q(X) \log Q(X) && \text{(A.3)} \\
&= \kappa(Q, \Theta) && \text{(A.4)}
\end{aligned}$$

The lower bound at the line (2.9) is obtained thanks to Jensen's inequality. The bound can be used to find the maximum likelihood parameter estimation. The Expectation-Maximization algorithm is the iteration of two steps: the Expectation step (E step) is the Maximisation of $\kappa(Q, \Theta)$ with respect to distributions Q holding Θ fixed, and the Maximization step (M step) is the maximisation of $\kappa(Q, \Theta)$ with respect to the parameters where Q is fixed. Starting from initial parameter Θ_0:

E step: $Q_{k+1} = Argmax_Q \ \kappa(Q, \Theta_k)$
M step: $\Theta_{k+1} = Argmax_\Theta \ \kappa(Q_{k+1}, \Theta)$

The difference between $\kappa(Q, \Theta)$ and the likelihood $\mathcal{L}(\Theta)$ is the Kullback-Leibler (KL) divergence between $Q(X)$ and $P(X|Y, \Theta)$. It is a standard result that the KL divergence is minimised when $Q(X) = P(X|Y, \Theta)$. So, the M step becomes $Q_{k+1} = P(X|Y, \Theta_k)$. Since the bound becomes an equality ($\kappa(Q_{k+1}, \Theta_k) = \mathcal{L}(\Theta)$), the M step is the computation of :

$$Argmax_\Theta \sum_{X} P(X|Y, \Theta_k) P(X, Y|\Theta)$$

because the expression $\sum_X Q \log Q$ does not depend on Θ.

That algorithm uses the Forwards-Backwards algorithm in order to compute the coefficients αs and βs. Let us introduce a variable $\xi_t(i, j)$, the probability of being in state S_i at time t and states S_j at $t+1$, given the model and the observation sequence. We can write the variable according to the definitions of the backwards and forwards

variables:

$$\begin{aligned}\xi_t(i,j) &= \frac{\alpha_t(i)A_{ij}B_j(O_{t+1})\beta_{t+1}(j)}{Pr(O|\Theta)} \\ &= \frac{\alpha_t(i)A_{ij}B_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)A_{ij}B_j(O_{t+1})\beta_{t+1}(j)}\end{aligned}$$

But, we have the probability of being in state S_i at time t given by:

$$\begin{aligned}\gamma_t(i) &= Pr(S_i \text{ at } t|\Theta) \\ &= \sum_{j=1}^{N}\xi_t(i,j)\end{aligned}$$

1. Interpretations
   $\sum_{t=1}^{T-1}\gamma_t(i)$: expected number of transitions from S_i
   $\sum_{t=1}^{T-1}\xi_t(i,j)$: expected number of transition form S_i to S_j
2. Algorithm
   a - the initialisation of the parameters:
   $\pi(i)$ follows a uniform distribution over the states
   A_{ij} follows a uniform distribution over the states j given the state i
   $B_j(o_i)$ follows a uniform distribution over the observation states i
   b - repeat till the critical point is reached:
   E step: computation of $\gamma_t(i)$ and $\xi_t(i,j)$
   M step: update of the parameter of the model

$$\pi^{new}(i) = \gamma_1(i)$$

$$A_{ij}^{new} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)}$$

$$B_i(k) = \frac{\sum_{t=1\ s.t.O_t=V_k}^{T-1}\gamma_t(i)}{\sum_{t=1}^{T-1}\gamma_t(i)}$$