Please cite this paper as

Klusáček David. “Maximum Mutual Information and Word Classes.” In *WDS’06 Proceedings of Contributed Papers.* MFF UK, Trója, Prague: Matfyzpress, Charles University, 2006, pp. 185-190.

or use the following bibTEX code

```
@inproceedings{ biblio:KlMaximumMutual2006,
title = {Maximum Mutual Information and Word Classes},
author = {David Klus{\'{a}}{\v{c}}ek},
year = {2006},
pages = {185--190},
crossref = {biblio:WDS06Contributed2006},
booktitle = {{WDS}'06 Proceedings of Contributed Papers},
}

@proceedings{ biblio:WDS06Contributed2006,
title = {{WDS}'06 Proceedings of Contributed Papers},
year = {2006},
publisher = {Matfyzpress, Charles University},
address = {{MFF} {UK}, Tr{\'{o}}ja, Prague},
isbn = {80-86732-84-3},
}
```

Maximum Mutual Information and Word Classes

David Klusáček[1]

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. Herein, I present some notes concerning implementation of now classical method of data clustering, called Maximum Mutual Information Clustering. It was introduced in [*Mercer et al.*, 1992] in context of language modeling. The original article contained some cues concerning its implementation. These are carried out in detail here, together with some new tricks. Results of the test run on 110M words long Czech National Corpus are briefly described then. Also, some problems of the original approach are identified and their possible cause is suggested.

1. Introduction

In statistical machine learning we often have to face problem of data sparseness. This problem makes the probability estimation of rare events nearly impossible, since we usually have one or even no observation of such an event in the training data. To overcome this, we have to incorporate some prior knowledge into our model. This prior knowledge reflects our belief that the world is regular in some sense and that we can reasonably deduce probability of unseen events from the encountered ones.

In a bigram language model, for instance, we want to estimate conditional probability of some word given its predecessor. For instance, when we are doing back-off probability smoothing, then we are implicitly expecting that this probability could be reasonably approximated by unigram probability[2].

Background idea of Maximum Mutual Information Clustering is in an intuition that a given word is more "interchangeable" with some words than with the others. Therefore it should be possible to have classes of words, lumping together the words which usually appear in similar contexts. Then, we could work with these classes instead of with the words in estimation of n-gram probabilities. As there will be much less classes than the words, we can expect the probability estimates to be more reliable than those that are using words directly.

1.1. Model

More formally, let us assume, that we are given the set W of possible words, together with a joint probability $P_0(v_k, v_{k-1})$, meaning how probable it is to encounter a fixed pair of consecutive words (v_k, v_{k-1}) in typical input text $v_0, \dots, v_M$. Then, for sake of simplicity, we define probability of a word sequence $w_0, \dots, w_N$ to be

$$P(w_0, \dots, w_N) := P(w_0) \cdot P(w_1, \dots, w_N | w_0) := P(w_0) \cdot \prod_{k=1}^{N} P_1(w_k | C(w_k)) \cdot P_2(C(w_k) | C(w_{k-1})) \qquad (1)$$

where C is the class function $C : W \to 2^W$ satisfying $C(x) \cap C(y) = \emptyset$ for any x, y such that $C(x) \neq C(y)$, and $\bigcup_{x \in W} C(x) = W$. Probabilities concerning the set of words are defined as one would expect:

$$\begin{aligned} P_c(A, B) &:= \sum_{a \in A, b \in B} P_0(a, b) \quad \text{for sets } A, B \subseteq W \\ P_1(x | C(x)) &:= P_c(\{x\}, W) / P_c(C(x), W) \\ P_2(C(x) | C(y)) &:= P_c(C(x), C(y)) / P_c(W, C(y)) \end{aligned} \qquad (2)$$

Our goal is to select function C so as to maximize $P(w_0, \dots, w_N)$ for some heldout data $w_0, \dots, w_N$. This is equivalent with maximizing logarithm of $P(w_1, \dots, w_N | w_0)$, leading to

$$L(C) := \sum_{k=1}^{N} \Big(\log_2 P_1(w_k | C(w_k)) + \log_2 P_2(C(w_k) | C(w_{k-1})) \Big) =$$

[1]This work was supported by the Czech Grant Agency under Contract 201/05/H014 and from the Grant Agency of the Academy of Sciences project No. 1ET201120505.

[2]This example is not very striking, since even the use of bigram model itself is based on our hope that bigrams will approximate true distribution well enough. So it would be surprising if unigrams would not work at all.

$$-\left(-\sum_{k=1}^{N} \log_2 P_c(\{w_k\}, W) \right) + \sum_{k=1}^{N} \log_2 \frac{P_c(C(w_k), C(w_{k-1}))}{P_c(C(w_k), W) \cdot P_c(W, C(w_{k-1}))} =$$
$$N \cdot \Big(\mathrm{I}\big(C(w_0), ..., C(w_{N-1}); C(w_1), ..., C(w_N)\big) - \mathrm{H}\big(w_1, ..., w_N\big) \Big) \tag{3}$$

where $\mathrm{I}\big(C(w_0), ..., C(w_{N-1}); C(w_1), ..., C(w_N)\big) = \frac{1}{N} \sum_{k=1}^{N} \log_2 \frac{P_c(C(w_k), C(w_{k-1}))}{P_c(C(w_k), W) \cdot P_c(W, C(w_{k-1}))}$ is cross mutual information and $\mathrm{H}\big(w_1, ..., w_N\big) = -\frac{1}{N} \sum_{k=1}^{N} \log_2 P_c(\{w_k\}, W)$ is cross entropy. Now it can be clearly seen where the name for the method came from. All we have to do is to find the mapping C maximizing mutual information of some training data (where we have estimated P_0) versus some (different) heldout data, and the result will be one that maximizes probability of heldout data in the framework of our model.

But, unfortunately, I am not aware of any reasonably fast algorithm, achieving this. It is clear that we have to back slightly off the optimality requirement for sake of practical feasibility. The next section explains how MMI method achieves this.

2. MMI clustering method

The MMI clustering method was introduced in [*Mercer et al.*, 1992] and is also described in [*Jelinek*, 1997]. The key idea is to use the same data for P_0 estimation as well as for C selection. This greatly simplifies algebra and also alleviates the need for probability smoothing required by the original formulation (which was needed there so that we would not get $\log_2(0)$ somewhere on the heldout data). On the other hand it is not very natural solution and has to be commented, at least: Working on the single data set means that the optimal classes become the singleton classes, one for each word. But this is not what we want. Here, the second idea takes place: Instead of maximizing $L(C)$ on the heldout data, we will try to keep $L(C)$ as high as possible for the preselected number of classes, putting aside the question of how do we discover right number of them. Still it is too difficult to be done on a computer except for very small input. So we back-off from the optimality even more and instead of trying to find the right classes, we will use eager solution working in a bottom-up way, building a forest of classes (eventually ending with one big classification tree).

Suppose we have the input text $w_0, \ldots, w_N$ and define co-occurrence matrix c_{yx} the following way

$$c_{yx} = \#\{k \mid k \in 1:N, w_k = y, w_{k-1} = x\} \tag{4}$$

Then we assign $P_0(y, x)$ to be c_{yx}/N.
N-times the mutual information $N \cdot \mathrm{I}\big(C(w_0), ..., C(w_{N-1}); C(w_1), ..., C(w_N)\big)$ can then be written as

$$I := N \cdot \mathrm{I}(\ldots; \ldots) = \sum_{X,Y \in \mathrm{Rng}(C)} \Bigg(\sum_{\substack{x \in X \\ y \in Y}} c_{yx} \Bigg) \log_2 \frac{P_c(Y, X)}{P_c(Y, W) \cdot P_c(W, X)} \tag{5}$$

Note that we treat $0 \log_2 0$ as 0 here, to make (5) equivalent to the original cross-entropy formulation (with P_0 estimated on the same data). We will only need special case of (5), having the C an identity, which is[3]

$$I = \sum_{x,y \in W} c_{yx} \log_2 \frac{N \cdot c_{yx}}{c_{y\bullet} \cdot c_{\bullet x}} \tag{6}$$

2.1. Basic algorithm

The following pseudocode describes how a simple clustering can be done. Its input is considered to be the matrix c_{yx} set up by (4) and indexed by words $w \in W$. For sake of simplicity we treat the words as numbers from 1 to A (the resulting classes will then have numbers from A+1 on). Function `I(c)` is defined by formula (6). Note that not all the words are being classified. This is because rarely appearing words (imagine `T` to be say 10) are too sparse to be classified reliably. Nevertheless, they are still in c_{yx} matrix to help the classification of other words. For sake of brevity, the algorithm does not build the tree, it only prints the history of merges as the tree can be easily reconstructed from it.

[3] $c_{y\bullet}$ is Einstein's summing notation meaning $\sum_x c_{yx}$, where x goes over the whole range in question.

```
set(int) active={ y | sum a (c[y,a]+c[a,y]) >= 2*T }
for(n=A+1; #active>1; n++) (
  {l,r}=argmax {y,x}, x in active, y in active ( I(merge(c,y,x,n)) )
  c:=merge(c,l,r,n);   active \= {l,r};   active U= {n}
  output("merging %d and %d into %d", l,r,n)
)
```

where `merge()` is

```
merge([M,M]int c, int k, int l, int n):[M,M]int
(
  for x in M\{k,l} (
    c[n,x]=c[k,x]+c[l,x]    /* set M is assumed to be large enough */
    c[x,n]=c[x,k]+c[x,l]    /* to hold all the new classes */
  )
  c[n,n]=sum (a,b) in {k,l}*{k,l} (c[a,b])
  for a in M ( c[k,a]=c[l,a]=c[a,k]=c[a,l]=0 )
  return c
)
```

3. Optimizations

The above algorithm has a complexity of $\Omega(C^5)$ where $C =$ `#active` is the number of words being classified. Optimizations are therefore necessary. The first step is to minimize the loss of $I()$ occurring after the merge instead of maximizing total $I()$. Although it seems to be a minor modification at the first sight, it turns out that the $I_{loss}()$ can be precomputed into an array and only slightly changed upon each merge, leading to a considerable speedup.

$$I_{loss}(c,l,r) = I(c) - I(\mathtt{merge}(c,l,r,n)) = \sum_{x,y\in W} c_{yx}\log_2\frac{N\cdot c_{yx}}{c_{y\bullet}\cdot c_{\bullet x}} - \Bigg(\sum_{\substack{x\in W\setminus\{r,l\}\\ y\in W\setminus\{r,l\}}} c_{yx}\log_2\frac{N\cdot c_{yx}}{c_{y\bullet}\cdot c_{\bullet x}} + \sum_{x\in W\setminus\{r,l\}}(c_{lx}+c_{rx})\log_2\frac{N(c_{lx}+c_{rx})}{(c_{l\bullet}+c_{r\bullet})\cdot c_{\bullet x}} + \sum_{y\in W\setminus\{r,l\}}(c_{yl}+c_{yr})\log_2\frac{N(c_{yl}+c_{yr})}{c_{y\bullet}\cdot(c_{\bullet l}+c_{\bullet r})} + (c_{ll}+c_{lr}+c_{rl}+c_{rr})\log_2\frac{N\cdot(c_{ll}+c_{lr}+c_{rl}+c_{rr})}{(c_{l\bullet}+c_{r\bullet})\cdot(c_{\bullet l}+c_{\bullet r})}\Bigg) \quad (7)$$

After rather technical manipulations this can be simplified into

$$Q(c_{l\bullet},c_{r\bullet})+Q(c_{\bullet l},c_{\bullet r})+Q(c_{ll},c_{lr})+Q(c_{rl},c_{rr})-Q(c_{ll}+c_{rl},c_{lr}+c_{rr})-\sum_{y\in J_Y}Q(c_{yl},c_{yr})-\sum_{x\in J_X}Q(c_{lx},c_{rx}) \quad (8)$$

where $Q(a,b) = R(a+b) - R(a) - R(b)$ and $R(x) = x\log_2(x)$ for $x > 0$ and $R(0) = 0$. The sets J_Y and J_X can be any supersets of $\{y \mid c_{yl}\cdot c_{yr} \neq 0\}$ and $\{x \mid c_{lx}\cdot c_{rx} \neq 0\}$, respectively[4]. Note that once we precompute $c_{\bullet x}$ and $c_{y\bullet}$ into suitable arrays all the terms in the formula except last two, can be evaluated in constant time. This already has a complexity of $O(A)$ for one evaluation of I_{loss} (where the naive implementation took $O(A^2)$). Moreover, the sets J_X and J_Y over which we are summing usually have much lower cardinality then A, leading to yet more improvement.

The algorithm can now precompute I_{loss} into an array for all $C(C-1)/2$ pairs using formula (8) (this amounts to $O(AC^2)$ operations), and then it can start iterations as before but instead of recomputing I every time, it would simply select the pair with minimal I_{loss}. Let this pair be (l,r). Then it would $\mathtt{merge}(c,l,r,n)$ and compute I_{loss} of the new class n with all other classes (search for minimal I_{loss} takes $O(C^2)$, merging takes $O(A)$, and all new I_{loss} values require $O(AC)$ — doing it C times leads to a complexity of $O(AC^2)$).

Last thing that must be done is the correction of all other I_{loss} values. It must be done since as l and r classes no longer exist (they were merged into a new class n), value of $I_{loss}(c,a,b)$ might have changed.

[4]this freedom is caused by the fact that $Q(0,x) = 0$ and $Q(x,y) = Q(y,x)$

Let the result of $\mathtt{merge}(c,l,r,n)$ be denoted by $\hat{c}$. Now we are about to compute what correction to $I_{loss}(c,a,b)$ has to be added to make it $I_{loss}(\hat{c},a,b)$.

$$\begin{gathered} I_{correction}(c,l,r,a,b) := I_{loss}(\hat{c},a,b) - I_{loss}(c,a,b) = \\ Q(\hat{c}_{a\bullet},\hat{c}_{b\bullet})+Q(\hat{c}_{\bullet a},\hat{c}_{\bullet b})+Q(\hat{c}_{aa},\hat{c}_{ab})+Q(\hat{c}_{ba},\hat{c}_{bb})-Q(\hat{c}_{aa}+\hat{c}_{ba},\hat{c}_{ab}+\hat{c}_{bb})-\sum_{y\in W\cup\{n\}}Q(\hat{c}_{ya},\hat{c}_{yb})-\sum_{x\in W\cup\{n\}}Q(\hat{c}_{ax},\hat{c}_{bx}) \\ -\left(Q(c_{a\bullet},c_{b\bullet})+Q(c_{\bullet a},c_{\bullet b})+Q(c_{aa},c_{ab})+Q(c_{ba},c_{bb})-Q(c_{aa}+c_{ba},c_{ab}+c_{bb})-\sum_{y\in W}Q(c_{ya},c_{yb})-\sum_{x\in W}Q(c_{ax},c_{bx})\right) \end{gathered} \tag{9}$$

Since $\{a,b\}\cap\{l,r\}=\emptyset$, many terms cancel out (see the definition of $\mathtt{merge()}$), leading to

$$\begin{aligned} I_{correction}(c,l,r,a,b) &= \sum_{y\in W}Q(c_{ya},c_{yb})+\sum_{x\in W}Q(c_{ax},c_{bx})-\sum_{y\in W\cup\{n\}}Q(\hat{c}_{ya},\hat{c}_{yb})-\sum_{x\in W\cup\{n\}}Q(\hat{c}_{ax},\hat{c}_{bx}) \\ &= Q(c_{la},c_{lb})+Q(c_{ra},c_{rb})+Q(c_{al},c_{bl})+Q(c_{ar},c_{br})-Q(\hat{c}_{na},\hat{c}_{nb})-Q(\hat{c}_{an},\hat{c}_{bn}) \\ &= Q(c_{la},c_{lb})+Q(c_{ra},c_{rb})+Q(c_{al},c_{bl})+Q(c_{ar},c_{br})-Q(c_{la}+c_{ra},c_{lb}+c_{rb})-Q(c_{al}+c_{ar},c_{bl}+c_{br}) \\ &= U(c_{la},c_{ra},c_{lb},c_{rb})+U(c_{al},c_{ar},c_{bl},c_{br}) \end{aligned} \tag{10}$$

where

$$U(a,b,c,d) := Q(a,c)+Q(b,d)-Q(a+b,c+d) \tag{11}$$

$I_{correction}(c,l,r,a,b)$ has to be added to every pair of $I_{loss}[a,b]$ array (a,b such that $\{a,b\}\cap\{l,r\}=\emptyset$) just before merging l with r. This amounts to $O(C^2)$ operations per iteration. Combining it with complexity estimates already done we have that the total complexity of the algorithm just sketched is $O(AC^2)$.

4. Implementation tricks

To further cut down the execution time (although the worst case complexity measured by means of A and C stays the same[5]) certain tricks are needed. As already hinted, the input data is preprocessed such that numbers are substituted for words. The second idea concerns the matrix c_{yx}. It is typically quite sparse, so it worths to implement it via a hash table.

4.1. Hash table

Note that it has to be a special table, since we need to be able to walk thru columns and rows as well as to delete elements no longer needed after the merge. Memory demands also impose some constraints on the form the hashing table should have. For instance, implementing the row/column walking ability via a linked list would be quite expensive, considering that on a 64 bit machine each pointer occupies 8 bytes and we would probably need two of them. The solution, I've chosen, is using an array (indexed by y) of hashing tables indexed by x. So for each line of the matrix we have an extra hash table. Their entries contain only the key x (32 bit integer) and the counter (32 bit integer) holding the value of c_{yx}. Collisions are resolved by double hashing. Thus, each bigram theoretically occupies 8 bytes of the memory. In practice it is more, since for the hashing to be fast we have keep say 60% of the table unused[6]. According to test runs, this leads to an average collision rate of less than 2 collisions per access[7] which is acceptable. Note that construction of such a table is a three phase process. In the first phase we read all the input data and count unigram frequencies. These frequencies are used as upper bounds for row sizes of the table. In the second phase, we allocate such a table and fill it with bigrams. The only purpose of this table is to count the true number of bigrams appearing in the respective rows. Finally this table

[5]In fact, it will even be worse than that, if we consider that the worst case behavior of hashing is worse than $O(1)$. In the following I will simply ignore that possibility since it is very unlikely to happen, thus not affecting the average performance.

[6]I know that it is not very good performance and that hashing functions exist, behaving well down to 30% of unused slots of the table. But these are more complicated, the one I mean requires size of the table to be from the set of prime twins. Definitely there is a room for improvement. But there are other things to improve that would result in much noticeable speedup, so this does not worth to be changed, now.

[7]this means that to store/retrieve single item to/from the hash, the array (representing the hash) has to be accessed less than 3 times on average.

is deleted and the right-sized table is build which has its row sizes selected such that they will be filled at 40%, unless they have less then two elements — in such cases[8] they are allocated tight, since there would be no speedup from an empty space, anyway.

4.2. Loss Computation and Merging

For I_{loss} computation as well as for merging we need to walk thru rows and columns. Walking along the row is done simply by reading valid entries of the hash table (60% items read are unused but it is acceptable). To walk thru a column, we need special array, one for each column holding y-indexes of entries in that column. The walk is then performed by look-up of y-indexes followed by search in the y-th hash for the key x. Therefore, it is slower than a row-walk.

For I_{loss}, it would be nice to walk only over the intersection of sets of indexes of non-zero entries of the respective rows/columns. But keeping track of the all $O(C^2)$ intersections would be very expensive. Therefore we only keep the track of number of non-zero items stored in a given row/column and select the shorter one for walking. While it is walked the corresponding data from the other column are being looked-up in the hash.

Merging l with r (where $l < r$) is done such that it reuses index l for the new class (instead of introducing new index n as in the basic algorithm). As we still want the new index in the output, we need to maintain translation array. Note that on each merge, number of items in rows and columns may decrease, so after the number of items stored in any given hash table falls below say 1/5 of the original filling, the whole table is rehashed to be smaller (this makes row-walks faster and it is also taking the advantage of CPU's caches, so the extra work pays off). Note that when merging rows, the number of entries of the resulting row l can generally increase. It is implemented such that a new (optimal sized) hash is created and the source hashes are unallocated after the data has been moved.

4.3. Loss-Correction Computation

Symmetries of $U()$ can be used to spare some evaluations of it. It is easy to see, that

$$\begin{aligned} U(a,b,c,d) = U(c,d,a,b) = U(b,a,d,c) = U(d,b,c,a) = U(a,c,b,d) = U(b,c,d,a) \\ U(0,b,c,d) = Q(b,d) - Q(b,c+d) = R(c+d) + R(b+d) - R(d) - R(b+c+d) \\ U(0,0,c,d) = 0 \\ U(0,b,c,0) = -Q(b,c) \end{aligned} \qquad (12)$$

Corrections from $U(c_{la}, c_{ra}, c_{lb}, c_{rb})$ are processed separately from $U(c_{al}, c_{ar}, c_{bl}, c_{br})$ since the first traverses rows while the other columns of c_{yx} (a and b is changing, l, r will be merged). To make it fast, we first gather for $x \in \texttt{active}$ those pairs (c_{lx}, c_{rx}) having at least one of their member non-zero; we can also precompute $Q(c_{lx}, c_{rx})$. This way we get the set X containing those x-es appearing in the non-zero pairs. Then we will compute the value of $U()$ for each $a < b, a \in X, b \in X$. We are using identities (12) to suppress evaluation of $U()$ in cases when

$$(c_{la} = 0 \text{ and } c_{lb} = 0) \;\text{ or }\; (c_{ra} = 0 \text{ and } c_{rb} = 0) \qquad (13)$$

In cases where $c_{la} = 0$ or $c_{ra} = 0$ simplified formula is used.

This trick proved itself to be crucial for the speed. The same program without it can run 4 days while it can only take 30 minutes with it (observed on part (1M words) of the Czech National Corpus (number of unigrams $A = 120187$, number of bigrams $B = 592188$, and the number of words being classified was $C = 10612$)). This optimization was not mentioned in [*Mercer et al.*, 1992] nor in [*Jelinek*, 1997] and constitutes the main theoretical contribution of this work.

4.4. Miscellaneous

There are also some auxiliary arrays there. As already noted, we have an array of arrays used to walk thru columns of the main table. We also have an array of lengths of those arrays. Then, there are mapping arrays translating internal numbering (which originates due to index re-usage) into external numbering and another one which translates it into numbering used by triangular I_{loss} matrix. Used by I_{loss} computations, there are arrays of row and column sums $c_{y\bullet}$, $c_{\bullet x}$ as well as $R(c_{y\bullet})$, $R(c_{\bullet x})$ which saves some evaluations of the $\log_2$ function. Also note that all of the column-walking-arrays may need an update when merging two rows. As this would be too expensive, lazy implementation is used which require some bookkeeping (I will not describe it here since it is marginal and quite long).

[8]these typically occupy half of the rows of the table due to the Zipf's Law

5. Experiments

The experiments were only meant to test the performance of the program. First, 1.2M words of collected Shakespeare's work were processed in 6 minutes (on 2.4 GHz machine, using single CPU), using less than 200 MB of RAM, yielding to 2.8k words classified. Next, larger input of 120M words of the Czech National Corpus was processed. It took 14 hours and 2 GB of RAM (on the same machine) to classify 10k most common words. Below, some selected parts of the tree are presented:

```
především +-------+-/    nic -+-------+--/    účetnictví ---+/    přičemž ++----+/
zejména +-/       |      něco /       |       bankovnictví +/     avšak --/|    |
hlavně -/         |      vůbec ------+/       plavání -----/      byť ----+/    |
nejen -----------+/      nikdy -----+/                            nicméně /     |
buď -----------++/       nikdo ----+/         zvýšit ----++       nýbrž -+-----+/
jedině -------+/|        nijak ---+/          snížit ---+/|       ba ---+/     |
přinejmenším +/ |        nikoho -+/           zvyšovat +/ |       jakož /      |
nejenom -----/  |        nikomu +/            snižovat /  |       případně ---+/
převážně ---+---/        nikde -/             omezit --++-/       respektive +/
výhradně --+/                                 rozšířit /|         popřípadě +/
speciálně +/             dvakrát ----+-+/     zlepšit +-/         natož ----/
výlučně --/              třikrát ---+/ |      posílit /
                         čtyřikrát +/  |                          pivo -+---+/
by ---------+--/         pětkrát --/   |      dobré --------+     čaj -+/   |
jsem ------+/            dávno -------+/      běžné -----+-+/     kávu /    |
jsme -----+/             několikrát -+/       přirozené +/ |      alkohol ++/
bych ---++/              tradičně --+/        obvyklé --/  |      sex --+-/|
bychom +/|               navždy ---+/         nebezpečné ++/      chléb /  |
byste +/ |               mnohokrát /          časté -+---/|       nábytek +/
bys --/  |                                    vzácné /    |       odpad --/
jste -+--/               Martin --+           drahé --+---/
jsi -+/                  David --+/           prosté +/           šance -++
ses +/                   Marek -+/            levné -/            naděje /|
sis /                    Daniel /                                 naději +/
                                                                  pozor -/
```

At the first sight they look quite convincing, but occasionally we can see weird classes (like `naději/pozor` or `bankovnictví/plavání`). I suppose that this is caused by classes which have very small (maybe even empty) sets J_X and J_Y in formula (8). Then, especially for rarely occurring words, I_{loss} becomes very small although the words being eventually merged have little in common. As the loss is very small such pairs are likely to be formed early as the program runs. They may form misleading classes which further spoil classification of other words.

This effect, hugely amplified, can be observed on a short data (having, say, 20k words of length) where we set `T` to be 1. Although for `T` high enough (such that there are few words to classify, say $C = 70$) the classes are acceptable, if we try to classify all the words, the result looks completely arbitrary. Even the words that were classified sufficiently well when C was 70 are wrong now with respect to one another.

6. Conclusions

In this paper I have described how MMI method can be efficiently implemented. Most of the tricks were already mentioned in [*Mercer et al.*, 1992]. However, slightly different and more compact formulas were found — symmetric formula (8) for I_{loss} and formulation of $I_{correction}$ using U-function (11). Their impact is twofold. At first, analysis of symmetries (12) leads to further savings in computation time, since it turns out that many results are 0 (what was not directly visible in the formulas of the original paper). Secondly, they yield to deeper insight into the numbers according to which the classes are selected. It was noticed that from (8) it follows that the criterion which eliminates rarely used words from classification should take the size of sets J_X and J_Y into the account, not only the total number of occurrences of a given word. Suitable formula taking these things into account is the aim of ongoing research.

Acknowledgments. Author would like to thank for computational resources made available to him by Department of Computational Linguistics of Faculty of Mathematics and Physics of the Charles University as well as for the linguistic data.

References

R.L. Mercer et al., "Class-based n-gram Models of Natural Language", *Computational Linguistics*, vol. 18, no.4 pp. 487–80, December 1992.

F. Jelinek "Statistical Methods for Speech Recognition", The MIT Press 1997.