Randomized Algorithms

Markov Chains

and

Handwritten Text

2005-05-05
Johan Henriksson
mahogny@areta.org
830730-7135

Idea

To use Markov chains or equivalent to recognize written text. It should be suitable for use on a PDA. Typical image analysis methods tend to fail because of the complexity of handwritten symbols. Much better performance can be achieved if the correlation between letters is considered.

Mathematical Formalism

Given a written text, we wish to know

$P(z_1z_2z_3z_4z_5z_6.. \mid text)$

where z is a sequence of symbols. It is assumed that the sequence maximizing the probability is the wanted string. Stating a Markov type property*, we can rewrite it:

$P(z_1z_2z_3z_4z_5z_6.. \mid text) = P(z_1 \mid text\ z_2z_3z_4z_5z_6..)\ P(z_2z_3z_4z_5z_6.. \mid text)$ *=

$P(z_1 \mid text\ z_2z_3)\ P(z_2z_3z_4z_5z_6.. \mid text) = P_{next}(z_1 \mid z_2z_3)\ P_{sym}(z \mid text)\ P(z_2z_3z_4z_5z_6.. \mid text)$

The P_{next} probability can actually be computed given lots and lots of text in the language considered. I will use English in this project. The accuracy of this guess depends on how many prior symbols are considered, the estimate of P on the amount of training text.

Finding P_{next}

In the above formula, I use symbols 2-3, that is, two in total. I downloaded a book from the Gutenberg project and estimated P_{next}. Just putting random texts together, I got this:

1 prior symbol:

wareson thertiectres go aifila thondlifou s, mincagheg s,"Whesshiold niloomindemowhocuiknk Br, s icuree, blliofatheis o hes tutond revecofethathaingare Ea he I o Bel heppand arlleid stces traic, bue y

2 prior symbols:

Bere. A grom they tacrition ad the leat hatesed ric mothe on uponew then, itheireat covere re orizind do ontoled. Wit."Do was the he on uslot gold enated of whater in to waybefuld becoped bods mitelig

The probability table grows too big for memory with more symbols, however, 2 symbols actually cause some English to pop out and should be enough for the purpose.

Finding P_{sym}

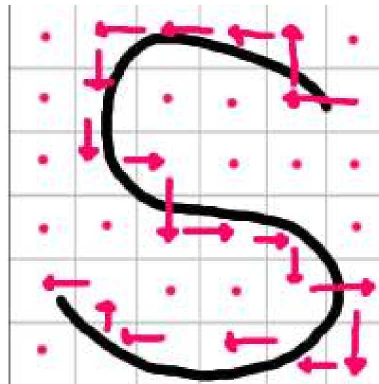
The problems with training a Markov chain to recognize a letter are the following:

1. Some letters require you to lift the pen
2. Letters are sometimes put together
3. Letters are of varying size
4. Restricted memory and very restricted training set

Because these problems are really image analysis and not randomized algorithms, I will cheat by giving the algorithm the letters one and one, as a sequence of points along the letters. I will also assume the letters are done one-by-one and not finished after the word is complete.

The last problem is an open question; the state space must be kept very small, yet be able to differentiate between all letters. Because we have some support from P_{next}, we should not try to get the guess perfect, just good enough. Some ideas:

- Normalize sketch length. Use curvature
- Map onto a grid. Consider probabilities of moving from one cell to another
- Mix the above

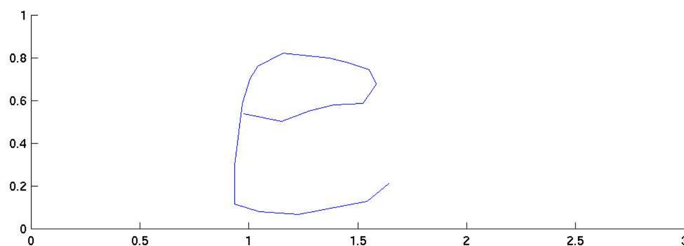


I decided to map onto a grid/image as seen in the above image. The “pixels” hold 4 probabilities each, one for each possible direction (actually, 3 values would suffice but it makes code more complicated). Initially, all transitions are of equal probability. For training, 5 of each letter was drawn and the transitions counted. To reduce the need for training, the state space was made small, e.g. the grid size set to 6x6, exactly as in the image above. P_{sym} is computed by

$P_{sym}=1*MUL_i\ P_{trans\ i}$

where i loops over the set of affected cells (i.e. if the line drawn does not cross a cell, those probabilities are not considered). From this formula it can be seen that for a perfect match, $P_{sym}=1$, while $P_{sym} \to 0$ quickly as transitions with less probabilities are made (0.25 is the probability for an untrained cell).

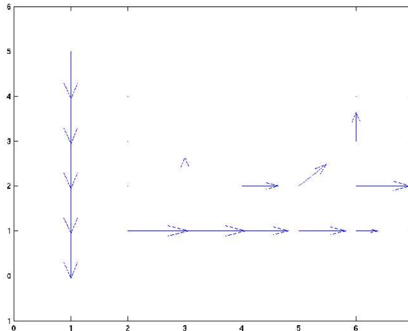
I used the Matlab plot window with ginput() to draw letters; an example below:



The probability transitions can be visually inspected; below is the letter L where transitions in a cell has been made to form a vector V in the major direction:

$V_x=(-P_{left}+P_{right})/2$

$V_y=(-P_{down}+P_{up})/2$



From this, V=(0,0) if all probabilities are equal.

I found P_{sym} to be a quite good measure (almost always picks the right letter) but I haven't bothered to collect full statistics of this.

Finding the Maximum Likelihood

For short sequences, it might well be possible to enumerate all possible solutions. However, this has complexity $O(n^{127})$ so it obviously doesn't scale at all. The property to use here is the multiplicativity between the probabilities. If one probability drops on the floor, it is not likely the considered combination will be the one sought for. The choices between symbols can be considered a tree. By doing a breadth-first search and cut off branches with the lowest probability, a partial search only involving the best candidates is made. The pseudo-code becomes the following:

```
Set=[{"", 1}]
ForAll characters written
      ForAll {comb, P} <- Set
            ForAll zi, {comb || zi, P*Psym*Pnext} => Set'
      Set = Set'
      While ||Set|| > SomeConstantDependingOnCPU
            Remove element with least probability from Set
Return maximum element from Set
```

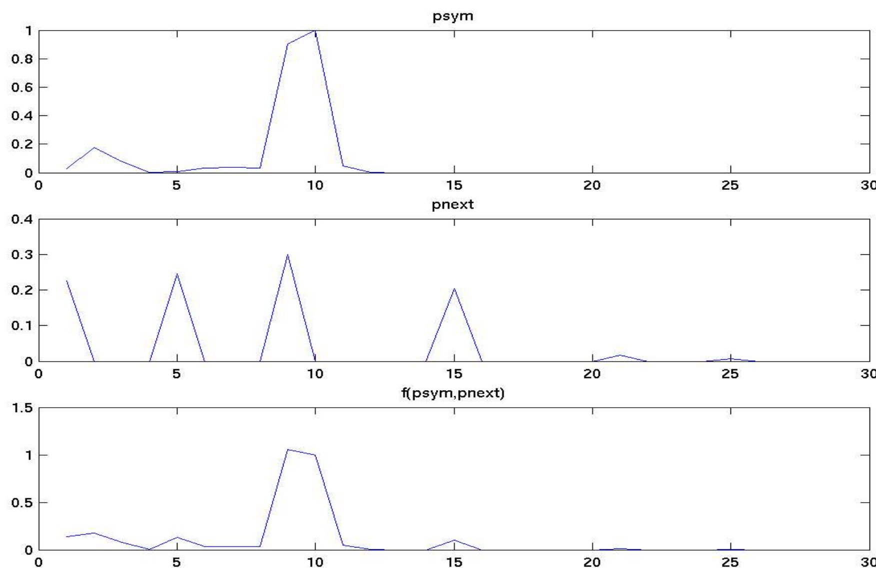
Although nice, it seems for this simple input data, this algorithm is overkill. Combined with the fact that it would be a pain to implement in Matlab, I decided to do the simplest version with ||Set||=1, corresponding to a greedy choice of the next character.

Test of Whole System

I found myself sitting with insufficient data. The book wasn't long enough and this had caused P_{next} to not be in equilibrium but hold many gaps. This had to be fixed somehow as it would otherwise cause $P_{next}*P_{sym}=0$ which obviously makes some words impossible to decipher. The solution is a hack, a new score function $P_{sym}+C*P_{next}$ where $C<1$ to make sure it doesn't dominate the guess. If P_{next} had been better estimate, I think a better formula would have been $P_{sym}*\log(D+P_{next})$ which is closer to the original formula but still evens out P_{next}. The modified pseudo code becomes:

```
Letters={space,space}
ForAll graph g
      ForAll symbol s
            Pnext[s]=f(s,take 2 last from Letters)
            Psym[s]=h(s,g)
      Normalize Psym, Psym=Psym/max(Psym)
      Ptot=Psym+Constant*Pnext, weighted score (constant~=0.5)
      Find max c <- Ptot
      Letters=Letters || c
Return Letters, except the first two symbols
```

Below is the current probabilities calculated just after 'li' has been written (that is, probabilities for the letter being 'i'); f() is the linear scoring function. It is seen how the scoring function clearly picks out the letter i with a score of factor 5 higher than the second best matches.



I wrote in 'life' and it accepted!

Conclusions

- Bayesian methods work well to predict English words
- My cell mapping is cheap and works very well
- Matlab is one weird program to draw letters in

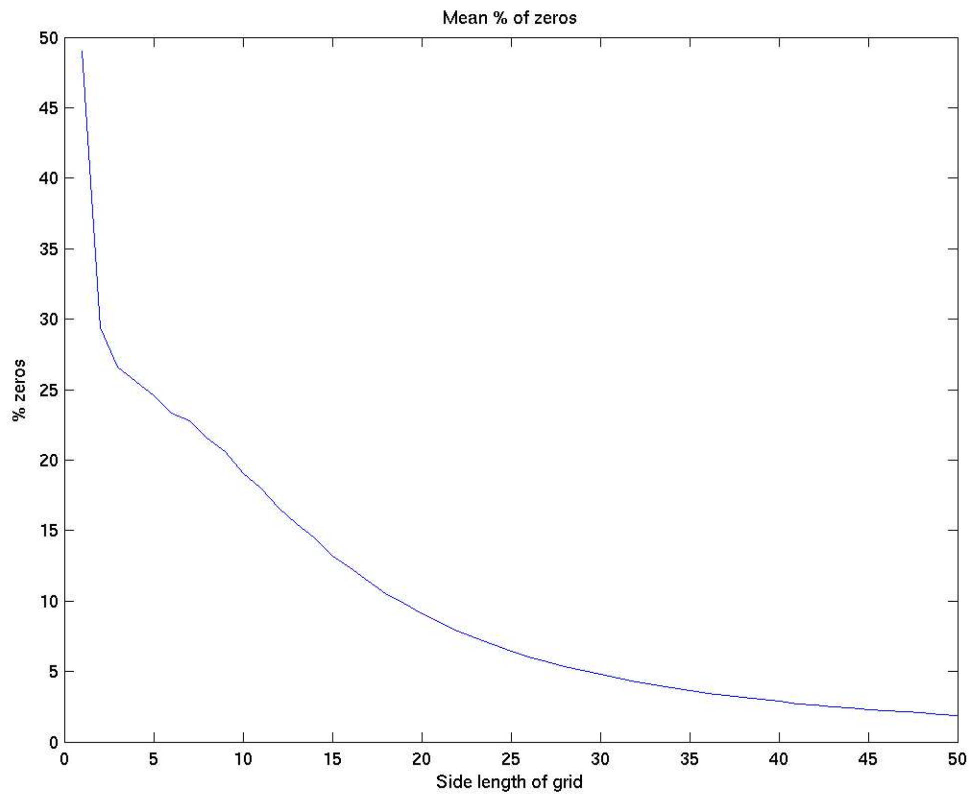
In overall, I think it would be possible to use this algorithm in a PDA.

References

I have seen the idea of Bayesian interference before, but I don't know where. The way of detecting letters (P_{sym}) is completely original.

Bonus

I made exercise 7.2 in the book as a warm-up so here goes:



For one symbol, it is obvious it should be 50% (boils down to a two-state chain with 50% chance for all transitions). Some sort of asymptote would be 2%.