Exercises

Statistical Processing of Natural Language

Winter 2013

1 Preliminaries: Linux tools for handling texts.

1. If you are not familiar with Linux commands such as: grep, sort, uniq, head, tail, cut, paste, gawk, etc. check their manpages and get acquainted with them. You can get a good introduction following the exercises in:

   K.W. Church, *Unix For Poets*
   `http://www.lsi.upc.edu/ padro/Unixforpoets.pdf`

   NOTE: Modern Linux commands may differ in some option flags or parameters from examples used in this tutorial. If you find problems, check the command manpage to find out the right parameters or options.

2 Zipf's Laws

1. Write a program to check Zipfs first law ($f = K/r$) on a real corpus: Count word frecuencies, sort them by rank, and plot the curve.

2. Compute the proportionality constant (K) between rank and frequency for each word. Compute its average and deviation. Discuss the results. Are they consistent with Zipfs Law?

NOTE: Use the text files `corpus/en.txt` and `corpus/es.txt`

3 Language Models – Entropy

1. Complete the python program `entro.py` to estimate the entropy of a language from a given input text, using:

   (a) Order zero (unigrams): $H = -\sum_x p(x) \log p(x)$

   (b) Order 1 (bigrams): $H = -\sum_x p(x) \sum_y p(y|x) \log p(y|x)$

   (c) Order 2 (trigrams): $H = -\sum_x p(x) \sum_y p(y|x) \sum_z p(z|xy) \log p(z|xy)$

4 Language Models – MLE & Smoothing

1. (a) Complete the python program `mle.py` to estimate via MLE the parameters of a character trigram model, and write them to a file.

   (b) Complete the program `generate.py` to generate a random sequence of characters consistent with the loaded trigram model.

   (c) Run the program `smooth.py` and enter different input sentences. Discuss why some sentences have zero probability. Modify the program `smooth.py` to perform a simple smoothing via Lidstone's or Laplace's Law. Discuss the values chosen for N and B.

2. (a) Extend the program mle.py to estimate the coefficients for a linear Interpolation smoothing. Write the coefficients into the first line of the model file, followed by the trigram parameters.

   Linear Interpolation: $P(z|xy) = \lambda_1 P(z) + \lambda_2 P(z|y) + \lambda_3 P(z|xy)$

   Coefficient estimation via deleted interpolation:

   λ_1=λ_2=λ_3=0
   foreach trigram xyz with $count(xyz) > 0$
      depending on the maximum of the following three values:
         case $\frac{count(xyz)-1}{count(xy)-1}$: increment λ_1 by $count(xyz)$
         case $\frac{count(yz)-1}{count(y)-1}$: increment λ_2 by $count(xyz)$
         case $\frac{count(z)-1}{N-1}$: increment λ_3 by $count(xyz)$
   normalize λ_1,λ_2,λ_3

   (b) Extend the program `smooth.py` to load the Linear Interpolation coefficients in the first line of the file, and use them to smooth the trigram probabilitites. Compare the results with the smoothing obtained in the previous exercise.

5 Language Models – Hidden Markov Models

5.1 Computing the parameters of a model

Given the following sequences of pairs `(state,emission)`:

```
(D,the) (N,wine) (V,ages) (A,alone)
(D,the) (N,wine) (N,waits) (V,last) (N,ages)
(D,some) (N,flies) (V,dove) (P,into) (D,the) (N,wine)
(D,the) (N,dove) (V,flies) (P,for) (D,some) (N,flies)
(D,the) (A,last) (N,dove) (V,waits) (A,alone)
```

1. Draw the graph of the resulting *bigram* HMM, and list all non-zero model parameters that we can obtain via MLE from this data.

2. Draw the graph of the resulting *trigram* HMM, and list all non-zero model parameters that we can obtain via MLE from this data.

3. Compute the probability of the following sequence according to each of the two previous models:
   `(D,the) (N,dove) (V,waits) (P,for) (DT,some) (A,last) (N,wine)`

5.2 The Viterbi Algorithm in log-space

In this exercise we will modify the Viterbi algorithm used to compute the most likely state sequence in HMMs. Recall that we can specify an HMM model as $\mu = (A, B, \pi)$, where A is a matrix of transition parameters, B is a matrix of emission parameters, and π is the initial state distribution.

Let's first recall the computations behind basic Viterbi. Given an observation sequence $O = O_1 \dots O_T$ the algorithm computes:

$$\underset{X=X_1 \dots X_T}{\operatorname{argmax}} P_\mu(X \mid O) = \underset{X}{\operatorname{argmax}} \frac{P_\mu(X, O)}{P_\mu(O)} = \underset{X}{\operatorname{argmax}} P_\mu(X, O)$$

The algorithm proceeds by defining quantities $\delta_j(t)$, which keep track of the most likely way of being at state X_j after emmitting $O_1 \dots O_t$. In parallel, the algorithm also computes variable $\psi_j(t)$, which store the transitions that lead to the most likely state sequence. The computations are as follows:

1. Initialization: $\forall j = 1 \dots N: \quad \delta_j(1) = \pi_j b_{jo_1}\,; \quad \psi_j(1) = 0\,;$

2. Induction: $\forall t = 1 \dots T,\ \forall j = 1 \dots N:$

$$\delta_j(t+1) = \max_{1 \le i \le N} \delta_i(t) a_{ij} b_{jo_{t+1}}\,; \quad \psi_j(t+1) = \underset{1 \le i \le N}{\operatorname{argmax}} \delta_i(t) a_{ij}\,;$$

3. Termination: backwards path readout.

$$\hat{X}_T = \underset{1 \le i \le N}{\operatorname{argmax}} \delta_i(T)\,; \quad \forall t = 1..T-1:\ \hat{X}_t = \psi_{\hat{X}_{t+1}}(t+1)\,; \quad P(\hat{X}) = \max_{1 \le i \le N} \delta_i(T)\,;$$

Moving to log-space

We are now interested in working in the logarithmic space. That is, we want to redefine the computations so that Viterbi computes $\log(P_\mu(X, O))$ instead of $P_\mu(X, O)$. This extension is important in tasks where probabilities of individual emissions or transitions may be very small, such as in NLP applications where the number of symbols is usually very very large. If individual probabilities are small then products of such probabilities will be very very very small. In such cases, our machines may run out of precision, hence yielding unreliable computations. Working in the log space is a common "trick" that solves the problem. We still want the most likely sequence under μ, but we will compute it as:

$$\begin{aligned}
\underset{X=X_1 \ldots X_T}{\operatorname{argmax}} P_\mu(X \mid O) &= \underset{X}{\operatorname{argmax}} \log(P_\mu(X \mid O)) \\
&= \underset{X}{\operatorname{argmax}} \log\left(\frac{P_\mu(X, O)}{P_\mu(O)}\right) \\
&= \underset{x}{\operatorname{argmax}} \log(P_\mu(X, O)) - \log(P_\mu(O)) \\
&= \underset{x}{\operatorname{argmax}} \log(P_\mu(X, O))
\end{aligned}$$

First of all, note that even if we compute log probabilities instead of normal probabilities, the most likely sequence will be the same. This is because the log function preserves the value of the state sequence attaining the maximum probability. Second, as before, we can drop the term $\log(P_\mu(O))$ because O is fixed, and it does not affect the maximum.

Let's assume that the HMM is given in the log-space. That is, $\mu' = (A', B', \pi')$ where

- For any states i and j: $a'_{ij} = \log a_{ij} = \log P(X_{t+1} = s_j \mid X_t = s_i)$
- For any state i and symbol k: $b'_{ik} = \log b_{ik} = \log P(O_t = k \mid X_t = s_i)$
- For any state i: $\pi'_i = \log \pi_i = \log P(X_1 = i)$

Questions:

1. Write $\log(P_\mu(X, O))$ in terms of μ'.
2. Rewrite the recursive expressions for δ and ψ to work with log probabilities.

5.3 Error-augmented Viterbi

In this question we will modify Viterbi to account for a notion of Hamming error of state sequences. In future lectures we will see applications of this algorithm.

Let X and X' be two state sequences of length T. We define an error function that counts the number of different states (also known as Hamming error):

$$\text{error}(X, X') = \sum_{i=1}^{T} I[X_i \neq X'_i]$$

where the function $I[p]$ is an indicator function that returns 1 if predicate p is true and 0 otherwise. For example, error("*abc*","*acb*") = 2.

For this problem, the input will consist of an observation sequence O together with its *correct* state sequence X^*. The goal is to find the *most-erroneous* sequence under an HMM model specified by μ. That is, we are interested in finding a state sequence that has high probability under our model and also has high error. More formally, we would like to find:

$$\underset{X}{\text{argmax}} \ \log(P(X, O)) + \lambda \cdot \text{error}(X^*, X)$$

where the parameter λ controls the trade-off between the two terms (high values give more importance to the error).

Question:

1. Modify the Viterbi algorithm to solve this problem.

   HINT: Note that the error function decomposes in a similar fashion to the computations behind an HMM. Then think of the optimality conditions behind the design of Viterbi that allow us to compute δ and ψ recursively.

6 Supervised Methods – Max. Entropy Classifiers

1. (a) Use the encoded corpus `corpus/efe/f50/train.0` to learn a Maximum Entropy Model using the `megam_i686.opt` executable:
   ```
   ./megam_i686.opt -quiet -fvals multiclass corpus/efe/f50/train.f0 > f50.mem
   ```

   (b) Test the performance of the module running `megam` in test mode on the corpus `corpus/efe/f50/test.f0`,:
   ```
   ./megam_i686.opt -fvals -predict f50.mem multiclass corpus/efe/f50/test.f0 >out
   ```

   (c) Complete the program `classifier.py` to compute the probability of each class for each input example, and produce the same output than `megam` test mode. Use the correct answer in the test files to compute the accuracy statistics.

2. (a) Modify the program `classifier.py` to output not only the most likely class, but all classes with a probability over a given threshold. Modifiy the evaluation to compute also precision, recall, and F1. Check how results vary depending on the given threshold.

   (b) Train and test a classifier using the corpus `corpus/efe/f100/train.f0` for training and the corpus `corpus/efe/f100/test.f0` for testing. Compare the performance of this classifier with that of the classifier obtained in the previous exercise using corpus `f50`. Perform a hypothesis test to find out whether the difference is statistically significant.

   (c) Perform a cross-validation evaluation for the same cases above, using corpus `corpus/efe/f50/train.*` and `corpus/efe/f50/test.*` to train and test five folds of one classifier, and `corpus/efe/f100/train.*` and `corpus/efe/f100/test.*` for the other. Discuss the changes in the statistical significance of the difference between both models.

7 Clustering

1. Compile the program `clustering.cc`:
   `g++ -o clustering clustering.cc -I ttcl.20110510/include`

2. Convert the document classification corpus labels to alphabetical codes (just for clarity in the output, not actually necessary). Select the first 1000 (not to spend much time waiting...):

```
cat corpus/efe/f50/test.f0 |
 gawk 'BEGIN {while (getline<"corpus/efe/EFE.classes") name[$2]=$1;} {$1=name[$1];print}' |
 head 1000 > mydat
```

3. Execute the program textttclustering, requesting 6 clusters:
   `./clustering 6 <mydat >clust.out`

4. The output is the expected class plus the assigned cluster for each example. Complete the `pip.py` program to compute the purity (P) and inverse purity (IP) of the resulting clustering. Experiment with different numbers of clusters and discuss the results

5. Modify the program `clustering.cc` to use a different linking strategy. Compare and discuss the obtained results with the previous ones.