Project Assignment 3 - Winter 2011 Segmenting Zulu Words by use of Conditional Random Fields

Andrew Heiberg
aheiberg@ucsd.edu

André Christoffer Andersen
andre@andersen.im

Fabian Siddiqi
fsiddiqi@ucsd.edu

March 1, 2011

Abstract

This paper considers segmentation of Zulu words into their syllable components. Conditional random fields are used, and the weights are learnt by use of a Collins perceptron. 10-fold cross validation is performed for parameter selection, resulting in 93.5% letter accuracy and 73.5% word accuracy. Stochastic gradient following is also considered, but implementation problems precluded generative any useful models. Finally, time and memory complexities of the algorithms used are calculated and scalability of the system is considered.

1 Introduction

Conditional random fields (CRFs) provide an alternative to hidden Markov models for labeling sequential data. They are used in many fields, such as bioinformatics (gene finding), computer vision (image segmentation) or computational linguistics, and are a type of Markov random field or undirected graphical model. Log-linear models, an extension to logistic regression, provide the basis for CRFs:

$$p(y|x;w) = \frac{\exp \sum_{j=1}^{J} w_j F_j(x,y)}{\sum_{y'} \exp \sum_{j=1}^{J} w_j F_j(x,y')}$$

where x is the input, y its associated label, F_j are the feature functions (of which there are J different ones) and w_j their corresponding weights. Feature functions $F_j(x,y)$ represent the compatibility of the training example x with the label y, and are generated explicitly by the authors. The denominator in the above expression (also called the partition function) imposes the two following constraints: $p_i \in [0,1] \forall i$ and $\sum_i p_i = 1$, which are necessary since we are considering probabilities.

The output of the model – the most probable label $\hat{y}$ given a novel word x – is calculated by maximizing the above expression:

$$\hat{y} = \text{argmax}_y \sum_{j=1}^{J} w_j F_j(x,y)$$

In this paper, a CRF-based method for segmenting novel Zulu words is presented. The main steps involved in designing such a system are:

1. Generating the feature functions, F_j. These are created by the authors, examples of which are given in Section 2.
2. Learning the weights, w_j. Two methods have been used: the Collins perceptron and Stochastic Gradient Following. Detailed explanations can be found in Section 2 and results in Section 4.
3. Testing the resulting model. Descriptions of the experiments can be found in Section 3.

2 Design of Algorithms

Different sections of the system are described in detail. Initially, the data set is parsed and pre-processed. Feature functions are subsequently generated, and their corresponding weights are calculated. Intermediate steps in this process are described in detail since computational tractability becomes an issue with a large amount of feature functions.

2.1 Data Pre-Processing

The original data set contains a list of $10,040$ segmented lowercase words as strings. These are converted to their ASCII equivalent and subtracted by 96 (such that a $\rightarrow$ 1, b $\rightarrow$ 2, and so on). Each word has an associated binary label of the same length: $y_i = 1$ for the letters at the end of a syllable and $y_i = 0$ otherwise. Special 'START' and 'STOP' tags are added to the beginning and ending of each word and labeling. Using a sequential labeling would open up the space of possible feature functions to include, for example, those dependent on syllable length. This would, however, come with additional computational complexity, as the Viterbi as well as the 'forward and backward' algorithm (discussed later) operate over all possible pairs of labels.

Our feature functions make use of letter sub-sequences of length 2 to 5. Naively, there are $26^2 + 26^3 + 26^4 + 26^5$ sub-sequences. However, not all of these will occur in the language (e.g. '*zzzzz*'). Using the data, sub-sequences which do not occur are eliminated, leaving $P = 30,469$. This is a 99% reduction in considered permutations. It could be argued that this might result in over-fitting, but intuition suggests that a valid sub-sequence in the Zulu language not occurring anywhere in 10,000 words is slim.

2.2 Feature Functions

High level feature functions are denoted F_j and are sums of low level feature functions, f_j, which take as input the position in the word:

$$f_j(y_{i-1}, y_i, \bar{x}, i)$$

where $\bar{x}$ is the example (a word), i is the position in the word being considered, y_i is the label at position i and n is the length of the word.

We chose a feature function space that matches possible sub-sequences and labels at varying positions to the words. More specifically:

$$f_j(y_{i-1}, y_i, \bar{x}, i) = I(SS_p = x[i{-}\text{off}, i{+}L_{SS_p}{-}1{-}\text{off}]) \times I(y[i{-}1, i] = \text{labelpair}_m)$$

where $p \in 1, 2, ..., P$, $off \in 0, 1, ..., \text{MSSL} - 1$, P is the number of possible sub-sequences of length two – MSSL and SS_p is the considered sub-sequence. MSSL is the maximum sub-sequence length and was chosen to be 5. The variable off specifies the required offset of i in the sub-sequence; for example, the feature function $I(\text{ckp} = \{x_{i-1}, x_i, x_{i+1}\}) \times I(01 = \{y_{i-1}, y_i\})$ would return 1 when applied to the word *back-pack* at i = 4 using $off = 1$.

There is a large number of feature functions (FF) and m^2 potential label pairs. Rather than apply each FF to each label pair for a given word, we can instead assert the FFs that will return a 1 by examining the word itself. Refer to Table 1 for an example of the feature functions that would return a 1 when applied to the word '*START E X A M P L E STOP*'.

Sub-sequence	**Position (i)**	**Offset**	**Label**
START e	1	0	all m^2 labelpairs
ex	1	1	all m^2 labelpairs
ex	2	0	all m^2 labelpairs
⋮	⋮	⋮	⋮
all FFs with length(subseq) == 2			
⋮	⋮	⋮	⋮
exa	1	2	all m^2 labelpairs
exa	2	1	all m^2 labelpairs
exa	3	0	all m^2 labelpairs
⋮	⋮	⋮	⋮
all FFs with length(subseq) == 3			
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
mple STOP	7	1	all m^2 labelpairs
mple STOP	8	0	all m^2 labelpairs

Table 1: Example output when calculating the feature function table for the word *example.*

While conceptually there are four dimensions to each feature function, they must be modeled as lying along a single dimension for the sake of sparse representation.

$$FF_{ix} = \Big[(i-1)\times(P\times\text{MSSL}\times m^2)\Big]+\Big[(SS_{ix}-1)\times\text{MSSL}\times m^2\Big]+\Big[\text{off}\times m^2\Big]+\Big[m\times y_0+y_1+1\Big] \quad (1)$$

where FF_{ix} indicates the feature function index and off is the slide offset.

The only undefined term is SS_{ix}, the index of the sub-sequence in the set of possible sub-sequences. To generate the index of a particular sub-sequence, each valid sub-sequence (see Section 2.1) and its index is put into a hash-table ahead of time. When a particular sub-sequence is discovered in a word, its index can be looked up. This hash-table look-up proved to be a performance critical portion of code. To optimize it, `java.Util.Hashtable` with integer keys was used.

The layout of the FF by index, as defined by Equation 1, places all the FFs with the same i position in a contiguous block of length $J = P \times \text{MSSL} \times m^2$. Within each 'i' block, FFs are layed out by the sub-sequence they use in blocks of $MSSL \times m^2$. Within each 'sub-sequence' block, FFs are layed out by slide offset in blocks of m^2. Finally, the m^2 FFs that are defined by particular `(i, sub-sequence, slide offset)` and any one of the m^2 labelings come one after another in the table.

When generating the $FFTable$ to compute the g_i table, each potential labeling pair must be evaluated by each FF, necessitating the construction of an m^2 by $N \times P \times MSSL \times m^2$ table. We find each `(i,sub-sequence,slide-offset)` triple present in a word, compute the first FF_{ix} in this block, and insert an $m \times m$ identity matrix.

2.3 Computing g_i Tables & Best Labeling

Computing the g_i table from this matrix and the weight array $w \in R^J$ is now straightforward. Since all FF_{ix} are organized by their position within the word, i, the FF_{ix} axis can be broken up into N segments of J components each. Each row in each segment is multiplied by the weights and summed. Each of these sums generates a value to be placed in the g_i table. For example, the first J entries in $FFTable$ at row r corresponding to a labeling $L1$,$L2$ would populate $g_1(L1, L2)$.

From the g_i table, the optimal labeling $\hat{y}$ can be computed using the Viterbi algorithm, which uses the recurrence $U(k, v) = \max_u[U(k-1, u) + g_k(u, v)]$. This dictates $U(1, v) = \max_u[U(0, u) + g_1(u, v)]$. Since $U(k, v)$ is defined as the best sequence of tags in positions 1 to k, the $U(0, u)$ term is meaningless. By construction, the only possible tag at position 0 is 'START', so $U(1, v) = g_1(\text{START}, v)$. A dynamic programming implementation using this base case takes $O(N \times M^2)$. Technically the Viterbi algorithm only computes the score of the optimal labeling, but with a little extra work (constant time), the corresponding optimal labeling can be recovered.

2.4 Training the Weights

Determining optimal values for w_j is done using two different methods, each of which is described in the following subsections.

2.4.1 Collins Perceptron

The method used to train the system is a modified version of the standard perceptron. The update rule (with a regulariztion term at the end) is as follows:

$$w_j := w_j + \alpha \times (F_j(x, y) - F_j(x, \hat{y})) - 2 \times \mu \times w$$

for each $< x, y >$ training example, where $\hat{y}$ is the predicted label, y is the actual label, and α is the learning rate. Since the weights remain unchanged when the estimated $\hat{y}$ is the same as the actual labeling, this update rule continues to cycle over the training examples until the change in the w vector falls below a threshold.

2.4.2 Forward and Backward Vectors

Once the Collins perceptron algorithm was created, the next algorithm is considered. The forward (α) and backward (β) un-weighted probabilities are defined as:

$$\begin{aligned} \alpha(i, y) &= \sum_{y\prime} \alpha(i-1, y\prime) M_i(y\prime, y) \\ \beta(y, i) &= \sum_{y\prime} \beta(y\prime, i+1) M_{i+1}(y\prime, y) \end{aligned}$$

where $M_i(y\prime, y) = \exp \sum_{j=1}^{J} w_j f_j(y\prime, y, \bar{x}, i)$. Finally, due to the recursive nature of these functions, it is necessary to define the base cases:

$$\begin{aligned} \alpha(0, y) &= I(y = \text{START}) \\ \beta(y, n+1) &= I(y = \text{STOP}) \end{aligned}$$

	Letter Accuracy			**Word Accuracy**		
	$\alpha = 2$	$\alpha = 1$	$\alpha = 0.1$	$\alpha = 2$	$\alpha = 1$	$\alpha = 0.1$
$\mu = 10^{-3}$	0.820	0.828	0.881	0.211	0.246	0.238
$\mu = 10^{-4}$	0.882	0.882	0.883	0.505	0.514	0.523
$\mu = 10^{-5}$	0.900	0.901	0.881	0.625	0.721	0.691
$\mu = 10^{-6}$	-	0.935	0.851	-	0.739	0.591

Table 2: Accuracy for different values of α and μ.

where the two labels considered are the buffers and n is the length of the word.

$Z(x, w)$, called the partition function, is defined as $Z(x, w) = \sum_{y\prime \in Y} \exp \sum_{j=1}^{J} w_j \times F_j(x, y\prime)$. It was computed explicitly as a sanity check for α and β. Unfortunately, $Z(x, w) \neq \sum\limits_{y} \alpha(n, y) \neq \sum\limits_{y} \beta(y, 1)$. A careful review and restructuring of the code with the correctness of these algorithms in mind did not remedy the problem. This failure is a source of considerable vexation for the authors.

3 Design of Experiments

K-fold cross-validation is performed on the system. The original $10,040$-word data set is randomized once and divided into K equally-sized subsamples. One subsample is retained as a testing set, while the system is trained on the remaining $K - 1$ subsamples, and the process is repeated K times. Weights for the final classifier are determined by averaging the outputs from each of the K folds. In this paper, K has been chosen to be 10.

Two parameters are used in the algorithms and must be selected for. The first parameter is the learning rate, α, and the second the regularization coefficient, μ. Cross-validation is performed on each parameter tuple.

Results and the accuracy of the system are shown in Section 4.

4 Results of Experiments

Parameter selection and resulting accuracy is shown in Table 2.

With $J = 609,900$, a single update step in Collins perceptron takes .2215 s and each sparse representation of an $FFTable$ takes approximately 13MB of memory for a six letter word. In general, one epoch takes approximately 40 minutes.

5 Findings & Analysis

The cross-validation scheme described in Section 3 allows for selection of optimal parameters to maximize the accuracy of the system. Optimal values are calculated to be $[\alpha, \mu] = [1, 10^{-6}]$, as shown in Section 4.

Once selection of optimal parameters was finished, subsequent tests allowed for a more rigorous analysis of the system. Cross-validation was repeated and results were averaged. Accuracy is considered at two levels: letter and word accuracy. The former is the normalized Hamming distance between the actual label, y, and the predicted label, $\hat{y}$

$$\mathrm{A}_{\mathrm{letter}} = 1 - \frac{\mathrm{Hamming}(y, \hat{y})}{L}$$

where L is the length of the word. Word-level hyphenation is considered successful if the Hamming distance between y and $\hat{y}$ is zero. Thus we calculate the word level accuracy as follows

$$\mathrm{A}_{\mathrm{word}} = \frac{1}{N} \sum_{i=1}^{N} I[\mathrm{Accuracy}_{\mathrm{Letter}}(y_i, \hat{y}_i) = 1]$$

Final results using optimal parameters are 93.5% letter accuracy and 73.9% word accuracy.

6 Conclusion and Limitations

Our project illustrates how CRF can be used to identify syllables in the Zulu language. This can be used in several applications. Among other uses, the approach can help improve text-to-speech pronunciation and text hyphenation methods of unknown words. Standard methods could be used for the known words - such as a syllable look-up table with grammar rules - and in case of uncertainty apply the CRF method.

The primary limitation of our experiments was our inability to find a clear optimal $[\alpha, \mu]$ combination. This was due to time constraints that hindered further parameter searching.

Regarding the errors in the forward/backward algorithm implementation, not much can be said other than the reason for the failure is still unclear.