Keyword Spotting For Low Resource Languages

Chris Tegho

March 23, 2017

1 Introduction

This practical investigates keyword spotting (KWS) with written queries on low resource languages. The Swahili corpus from the IARPA Babel project's 2015 surprise evaluation language is used. A KWS system is implemented and 1-best ASR decoding output from a word-based system and a morph-based system are used. Mapping out-of-vocabulary keywords to proxy keywords using graphemic confusion networks and the use of score normalization are investigated.

Outputs from a WFST-based KWS system are also used and their hits and scores combined with the systems using 1-best output. The effect of score normalization on system combination is also investigated.

All development for this practical can be found in the directory `/home/ct506/Desktop/kws`. Some scripts were added in the appendixes.

2 1-Best Keyword Spotting

To enable query words and phrases in a 1-best list to be found, an indexer is implemented. The output of a 1-best ASR decoder with posterior scores is used. A hashmap is used to store all terms from the 1-best ASR output in an index. A hashmap lookup is then employed to perform a query.

2.1 KWS Search

A position hashmap is maintained separately where each position will have as a value the word at that position. This hashmap is used for efficient phrase search: a lookup is performed for the first term in the phrase. For each of the hits for the first term, the position hashmap is used to determine if the next word in the 1-best output, from the same file, corresponds to the next word in the phrase query. If this is the case, the hit is still valid, and the requirement that the time between the two words is less or equal than 0.5 is verified. If the requirement is satisfied, the hit is kept as valid, and the process is repeated until all terms in the phrase query are visited. The score for a phrase query is set to the average score of all the terms in a hit.

2.1.1 Computational Complexity

Each term query requires $O(1)$ which corresponds to one lookup operation. Let n_i be the number of hits per term in phrase query of N terms. For each hit term, the complexity for a phrase query with N terms is then $O(\prod_{i=1}^{N} n_i)$. However, because the term at the position following a token position is verified first before querying for that term, for each hit of the first term, only one lookup operation is needed for each of the terms in the phrase query following the first term. The complexity for the worst case scenario, where all hits are valid, is then $O(N * n_0)$. The pseudocode for the hits search is illustrated below:

```
query = "What she has done" → N = 4
5 hits for term "What" → n_0 = 5
i ← 0 i is term "What"
for each term in phrase query do
  while i ≤ length of phrase query do
    if term at i+1 in phrase query = term at position+1 in file of hit n then
      if time requirement is satisfied then
        query term information
        i ← i + 1
      end if
    end if
  end while
end for
```

2.1.2 Experimental results

Results using the reference output `reference.ctm` and the ASR 1-best decoding output `decode.ctm` are summarized in Table 1. To assess results in the practical, Term Weighted Value is used [1]:

$$TWV = 1 - (P_{miss} + \beta P_{FA})$$

MTWV is the maximum TWV over the range of all possible values of the detection threshold.

System	All	OOV	IV	Threshold
`reference.ctm`	1.000	1.000	1.000	1.000
`decode.ctm`	0.319	0.402	0.000	0.167

Table 1: MTWV results for KWS using the reference output `reference.ctm` and the ASR 1-best decoding output `decode.ctm`.

The system achieves 1.000 TWV on the reference transcription (`reference.ctm`) as expected. The maximum TWV for the 1-best ASR output (`decode.ctm`) is 0.32, much lower than the MTWV obtained with the reference transcription. This is mainly because (1) the Out-Of-Vocabulary (OOV) terms are not dealt with, resulting in a 0 MTWV for OOV words, and (2) errors are accumulated in the ASR transcriptions, which affects In-Vocabulary queries performance.

2.2 Morphological Decomposition

For languages with rich resources, such as English, more data is often the best solution to decrease the OOV rates and obtain better results with ASR or KWS, since the required data is readily available [3]. However, for low-resource languages, using more data is difficult since annotated data is less readily available. To improve the performance of OOV terms, morphological decomposition is applied to the query terms and the 1-best list.

`CTMtoMorph.py` is used to decompose the 1-best list `decode.ctm` with `morph.dct`. The time duration is split equally among the morph sub-units and the scores for each sub-unit is equal to the score of the original term. The beginning time of the sub-unit is adjusted to make sure the beginning time and duration of the word are the same as when the morph sub-units are combined. `transformToMorph.py` is used to decompose the queries. Results are shown in Table 2. Results for KWS using the morphological decomposed queries on the morph system decoding output (`decode-morph.ctm`) are also shown.

2.2.1 Experimental results

System	All	OOV	IV	Threshold
`decode.ctm + morph.dct`	0.315	0.019	0.391	0.205
`decode-morph.ctm`	0.320	0.069	0.384	0.56

Table 2: MTWV of KWS with morphological decomposition

2.2.2 OOV rates and Morphological Decomposition

With morphological decomposition, OOV terms are decomposed into two or many sub-words units. Some or all of these sub-units can be IV which results in lower OOV rate. For instance, if the CTM file contained the word

abandoned but not the word abandon, querying for abandon gives 0 hits. With a morphological decomposition, abandoned in the cTM file is decomposed to abandon ed and querying for abandon gives at least one hit. This hit could be a true positive resulting in an increased TWV in situations where abandon was erroneously transcribed to abandoned in the CTM file. The OOV MTWV is higher for both systems decode-morph.ctm and decode.ctm + morph.dct, when compared to the 0 TWV for OOV for decode.ctm. The overall MTWV for decode.ctm and decode-morph.ctm are similar, mainly because of a decrease in the IV MTWV for decode-morph.ctm.

	#Sys	#CorDet	#FA	#Miss	FA%
decode.ctm	725	405	320	558	44
decode.ctm +morph.dct	1010	413	597	550	59
decode-morph.ctm	947	406	541	557	60

Table 3: Performance metrics for KWS before and after applying morphological decomposition

When looking at the performance metrics for all systems (see Table 3), the percentage of correct hits is larger for decode-morph.ctm than for decode.ctm, because of the lower OOV rate. However, the rate of false alarms is significantly higher for decode-morph.ctm (60%) compared to decode.ctm(44%), which explains the lower IV MTWV. AS OOV terms are decomposed to sub-units, the number of IV terms increases. The number of false positives increases as some terms that were OOV before the segmentation will get hits after the segmentation, even in cases where the non-segmented transcription does not present any error for that OOV term. For instance, when querying abandon, the system will find a hit with the segmented ASR output abandon ed. This hit is a true positive if the reference for the transcription of the word abandoned is erroneous and should have been abandon; and is a false positive if the reference for that term is abandoned.

2.2.3 decode-morph.ctm vs decode.ctm + morph.dct

The overall MTWV for KWS with decode-morph.ctm is higher than the one obtained with decode.ctm + morph.dct. Two observations can be made:

1. The OOV MTWV with decode-morph.ctm is higher than the one with decode.ctm + morph.dct. A morph-based system for decoding is more suitable for OOV words when doing morphological decomposition. Performing a mapping on the 1-best system can result in more erroneous morph decompositions, especially if the 1-best ASR output is erroneous itself.

2. The IV MTWV is lower with decode-morph.ctm than the one with decode.ctm + morph.dct. As IV words are decomposed, the number of hits increases, and the PFA increases. The PFA is highest with decode-morph.ctm resulting in lower IV MTWV, compared to decode.ctm + morph.dct.

2.3 Score normalization

Rather than using the raw scores derived from the KWS process, it is possible to perform score normalization at query term level prior to performing scoring. The Sum-to-One (STO) normalization is used:

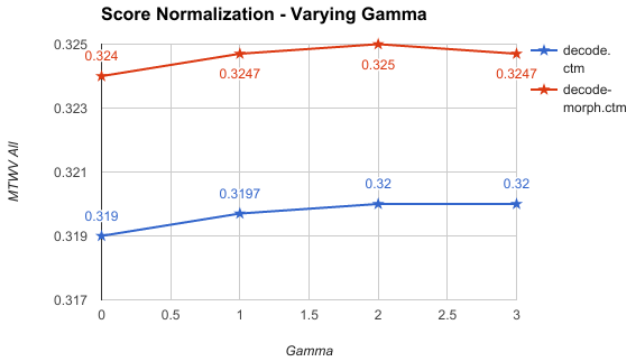
$$\hat{s}_{ki} = \frac{s_{ki}^{\gamma}}{\sum_j s_{kj}^{\gamma}}$$

where s_{ki} is the score of the i^{th} hit for keyword k, and the summation is over all hits for keyword k. Score normalization was tested for $\gamma = [0, 1, 2, 3]$ and $\gamma = 2$ gave highest TWV for both decode-morph.ctm and decode.ctm. For the rest of the experiments, score normalization was performed with $\gamma = 2$.

2.3.1 Results

System	All	OOV	IV	Threshold
decode.ctm + morph.dct	0.320	0.000	0.402	0.030
decode-morph.ctm	0.325	0.068	0.391	0.046

Table 4: MTWV after applying STO score normalization with γ=2



(a)

Figure 1: MTWV with respect to parameter γ with score normalization.

Score normalization improved the MTWV for both `decode-morph.ctm` and `decode.ctm`, at the OOV and the IV level. MTWV is directly affected by missed detections and false alarms [1]. s_k is smaller for rare keywords, and larger for frequent keywords [2]. The number of missed detections is higher for rare keywords because their score tends to be lower than the best deciding threshold, and the number of the false alarms is higher for frequent keywords. By normalizing using the summation of scores over all hits for a keyword, the detection scores of rare keywords is boosted and the detection scores of frequent keywords is decreased, resulting in less false alarms and missed detections, and higher MTWV [2].

2.4 Grapheme Segmentation

Instead of using morphological decomposition, word-based decoding output can be used with a grapheme-confusion matrix to handle OOV word. OOV keywords are mapped to proxy IV keywords by computing the minimum edit distance or Levenshtein distance, and choosing the IV word with smallest distance. The probability that a hypothesis (what a reference grapheme is recognized as) is observed given the reference is computed as:

$$P(hyp|ref) = \frac{C_{ref}(hyp)}{C_{ref}},$$

where $C_{ref}(hyp)$ is the number of times a reference is substituted by a hypothesis hyp, and C_{ref} is the number of substitution errors associated with the reference [4].

The cost for substitution was set to $1-P(hyp|ref)$, and the costs for deletion and addition were set to $1-P(hyp|sil)$ and $1-P(sil|ref)$ respectively. Computing the distance was implemented with a tabular computation [5]. For each OOV term, the computational complexity is O(kMN), where k is the number of IV terms, M the length of the OOV term, and N is the length of the IV term. The pseudo code is found below, and the corresponding script is `toGrapheme.py`.

Initialization
$D(i, 0) \leftarrow i$
$D(0, j) \leftarrow j$
M is length of OOV term, N is length of IV term
for each i = 1...M **do**
  for each j = 1...N **do**
    $cost_{substition} = 0$ if X(i) = Y(j), else $cost_{substition}$
    $D(i, j) = min[D(i-1, j) + cost_{deletion}, D(i, j-1) + cost_{addition}, D(i-1, j-1) + cost_{substition}]$
  end for
end for
Distance is D(N, M)

2.4.1 Results

System	All	OOV	IV	Threshold
`decode.ctm + morph.dct`	0.271	0.000	0.340	0.203
`decode-morph.ctm`	0.295	0.054	0.356	0.396
`decode.ctm + morph.dct + STO`	**0.337**	0.081	0.404	0.036
`decode-morph.ctm + STO`	0.325	0.065	0.388	0.046

Table 5: MTWV after using the grapheme confusion network to handle OOV

Using the grapheme-confusion matrix improved the overall MTWV and the OOV MTWV when applied on the decoding output (`decode.ctm`) with STO score normalization. The performance of the grapheme based system is higher than the one observed with morphological decomposition. No improvements were observed when the grapheme confusion matrix was used with the queries with morphological decomposition and with the decoding output from a morph-based system (`decode-morph.ctm`). Without score normalization, the grapheme based system performed worse than KWS with `decode.ctm`.

For lattice based KWS, the grapheme-based system offers two advantages over the morph-based system [6]:

- The probabilistic description of pronunciation variation in the grapheme model helps represent all possible pronunciations of a query term in a single form.
- The grapheme model incorporates both acoustic and phonological cues in the lattice, thus improving the decision-making process in the search phase.

Given the increased acoustic variation associated with graphemes compared with phones, the advantage arises from postponing hard decisions and keeping multiple decoding paths alive [6].

In this practical however, only 1-best decoding output was available and one reason the grapheme based system improved over the morphological based one is because the morphological decomposition is nonprobabilistic and represents a potentially error-prone hard decision. By using grapheme-based units, the hard decision is replaced with a probabilistic one [6]. By mapping all OOV keywords to IV proxy keywords, the number of false alarms increases. Although the overall results were an improvement over the previous approaches described here, it might be possible to obtain higher performance by only mapping OOV words to proxy IV words with a minimum distance below a certain threshold. This would have the effect of reducing the number of false alarms.

3 System Combination

Multiple sets of possible hits for a query can be combined. Hits returned by different systems may be merged together. Scores are merged if they refer to the same document and when the time-stamp information for these hits overlaps.

3.1 CombSUM

Different methods exist for merging scores. The approach adopted here is the CombSUM. Let h_{ik} denote a hit for th keyword k from the i-th system among n systems, and $s(h_{ik})$ the score for that hit. CombSUM [1] consists of computing the summation of the hit scores from the different systems as $\sum_{i=1}^{n} \alpha_i s(h_{i,k})$, where α_i is the weight of contribution for a system i.

System combination was tested with hits from the KWS with the 1-best decoding output `decode.ctm`, the morph-based system `decode-morph.ctm` and the graphemic-based system `decode-graph.ctm` with and without STO score normalization. Hits common to 2 systems have their scores merged as described by the CombSUM method. Hits that exist in one system only have their scores multiplied by the weight for that system α_i. This has the effect of decreasing the score for a hit that only occurs in one system but not the other.

3.1.1 Results

System	All	OOV	IV	Threshold
decode.ctm + decode-morph.ctm	0.348	0.068	0.420	0.135
decode-morph.ctm + decode.ctm + decode-grapheme.ctm	0.340	0.069	0.411	0.526
decode-STO.ctm + decode-morph-STO.ctm	0.362	0.069	0.437	0.037
decode-morph-STO.ctm + decode-STO.ctm + decode-grapheme-STO.ctm	**0.374**	0.113	0.441	0.083

Table 6: MTW after combining KWS with 1-best output system (decode.ctm, morph-based system (decode-morph.ctm and graphemic-based system decode-graphemic.ctm. Results are shown for the systems after applying STO score normalization, denoted with -STO.

Without STO score normalization, the highest improvement in MTWV for IV words was obtained by combining the 1-best ASR output system and the morph-based system (note that the α_i were tuned). All other combinations led to worse MTWV. By using the same systems With STO score normalization, the same combination led to an overall MTWV of 0.362. The highest MTWV was obtained by combining the 1-best ASR output system, the morph-based system and the grapheme based system with STO score normalization (an improvement of 10% compared to the same combination but with systems without STO). These results highlight one advantage of score normalization: the scores from different systems without normalization are not comparable. Hence, score normalization improves results, in part because it boosts scores of rare keywords and decreases the score of frequent keywords and in part because the scores provided by the different systems become comparable, when combining systems.

One possible extension is to use the MTWV-weighted CombMNZ introduced in [1], where the hit scores from a system are weighted in proportion to the MTWV of that system.

3.2 Combination with KWS with word-lattice based decoding output

In addition to combining systems from the 1-best ASR output, IBM WFST-based KWS systems are provided and used for system combination. Results are summarized below (only the most performing system combinations are shown).

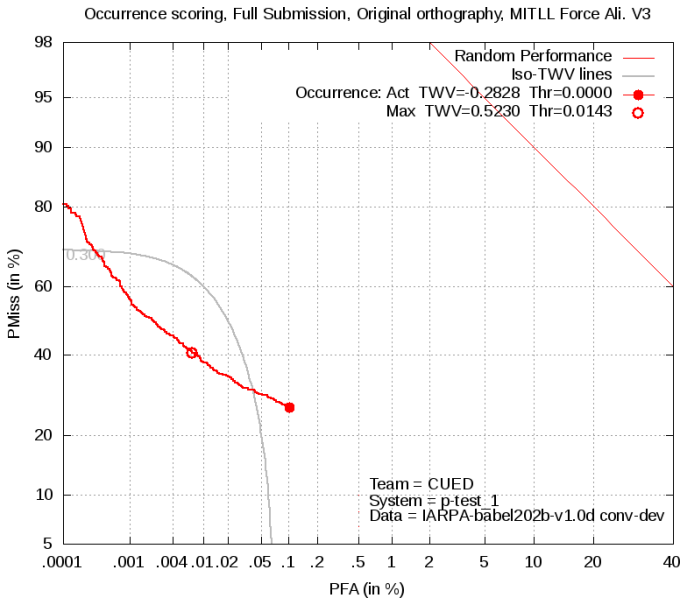
3.2.1 Results

System	All	OOV	IV	Threshold
morph.ctm	0.360	0.089	0.430	0.071
word.ctm	0.399	0.000	0.501	0.077
word-sys2.ctm	0.403	0.507	0.000	0.047
morph-STO.ctm	**0.509**	0.364	0.546	0.008
word-STO.ctm	0.456	0.000	0.573	0.007
word-sys2-STO.ctm	0.462	0.000	0.582	0.011

Table 7: MTWV for KWS with the IBM WFST-based systems. -STO denotes that score normalization was applied.

System	All	OOV	IV	Threshold
word-sys2+decode+decode-morph+decode-graphemic	0.420	0.108	0.501	0.065
word+word-sys2	0.384	0.000	0.483	0.066
word-STO+morph-STO	**0.522**	0.372	0.561	0.014
morph-STO+word-STO	0.517	0.37	0.556	0.006

Table 8: MTWV after combining systems. -STO denotes systems with score normalization were combined.



(a)

Figure 2: DET curve for the combined system `morph-STO + word-STO`.

Without score normalization and without system combination, the highest MWTV was obtained with the `word-sys2.ctm` system. With score normalization, the highest MWTV was obtained with `morph-STO.ctm`.

System combination did improve results: When applied to the WFST-systems and the systems from the 1-best decoding outputs, morph, and graphemic based systems, the highest MTWV was obtained by combining all three 1-best based systems with `word-sys2`, without score normalization. Applying score normalization is important for system combination (improvement of 23% for overall MTWV when compared to systems without STO normalization), with the highest overall MTWV (0.52) obtained by combining `word-STO.ctm` with `morph-STO.ctm`. System combination improved performance for both IV and OOV keywords, with a 2.2% improvement for OOV keywords and 2.8% improvement for IV keywords when comparing the highest MTWV obtained without system combination (with STO normalization) to the one obtained with system combination (with STO normalization).

These results show that system combination improves results, even when score normalization is not used. Without system combination, WFST-based systems achieved the best results, emphasizing the advantage of using word lattices ASR output instead of 1-best. The combination of the WFST based systems outperformed any other combination involving the 1-best output based systems, especially after score normalization. When using the normalized systems for system combination, scores from different systems were weighted equally. Using a different score merging method such as MTWV-weighted CombMNZ can lead to better results when combining systems, including the less performing systems with 1-best decoding output.

The improvements observed with system combination are less significant than the ones obtained by Mamou et al [1]; however results cannot be compared since different systems, different score merging methods and a different language were used.

3.2.2 Performance wrt to query length

The repartition of the queries according to query length (measured by the number of keywords) is provided in Table 9. STO normalization, system combination and STO-system combination improve MTWV for all query lengths except for queries of length 5 (no hits were obtained for 5-keywords queries with any of the systems considered in this practical), with the effect being strongest for the longer queries (length 3 and 4). The strong system combination (with STO normalization) improvement for 4-keywords queries is due to the high improvement in OOV rates.

Query Length	morph	morph-STO	word-sys2 +decode +decode-morph +decode-graph	morph-STO +word-STO	%queries	% STO improv	% comb improv	% STO-comb improv
1	0.372	0.511	0.430	0.520	57.8	37.37	15.59	39.78
2	0.384	0.567	0.455	0.573	34.8	47.66	18.49	49.22
3	0.197	0.294	0.241	0.391	4.6	49.24	22.34	98.48
4	0.000	0.003	0.100	0.097	2			
5	0.000	0.000	0.000	0.000	0.8			

Table 9: STO normalization and system combination MTWV relative improvement as a function of query length. Overall MTWV are reported in columns 2-5 and percentage improvement are relative to the results with morph WFST based system.

4 Conclusion

In this practical, keyword spotting was investigated on a low-resource language (Swahili) with written term queries. For handling OOV words without system combination, graphemic based systems achieved highest performance (graphemic confusion matrix was not tested with WFST based systems). STO normalization improved overall MTWV for all systems considered in the practical and was essential when performing system combination. With score normalization, WFST morph-based system achieved highest MTWV for both IV and OOV keywords. Combining systems improved results further, however further improvements could be obtained by improving the score merging methodology. Highest improvements with STO normalization and system combination were observed with longer queries.

References

[1] Mamou, Jonathan, Jia Cui, Xiaodong Cui, Mark JF Gales, Brian Kingsbury, Kate Knill, Lidia Mangu et al. "System combination and score normalization for spoken term detection." In *Acoustics, Speech and Signal Processing (ICASSP)*, 2013 IEEE International Conference on, pp. 8272-8276. IEEE, 2013.

[2] Wang, Yun, and Florian Metze. "An in-depth comparison of keyword specific thresholding and sum-to-one score normalization." (2014).

[3] Narasimhan, Karthik, Damianos Karakos, Richard Schwartz, Stavros Tsakalidis, and Regina Barzilay. "Morphological Segmentation for Keyword Spotting." *2014 Conference on Empirical Methods on Natural Language Processing* (October 2014).

[4] Xu, Di, Yun Wang, and Florian Metze. "EM-based phoneme confusion matrix generation for low-resource spoken term detection." In *Spoken Language Technology Workshop (SLT)*, 2014 IEEE, pp. 424-429. IEEE, 2014.

[5] Jurafsky, Dan. Minimum Edit Distance. Available: https://web.stanford.edu/class/cs124/lec/med.pdf

[6] Tejedor, Javier, Dong Wang, Joe Frankel, Simon King, and José Colás. "A comparison of grapheme and phoneme-based units for Spanish spoken term detection." Speech Communication 50, no. 11 (2008): 980-991.

A Indexing and Token Search

`indexer.py`: script for indexing CTM files and performing KWS. See `getIndexer()` for script to get index $I(w)$. See `searchToken()` for function that performs keyword spotting given a query.

```
import argparse
def getIndexer(filename):
        indir = './lib/ctms/' + filename
        text = open(indir).read()
        entries = text.split('\n')
        indexer = {}
        scorer = {}
        positioner = {}
        position = 0
```

```
        fileArcs = {}
        for entry in entries:
                arc = entry.split(' ')
                if (len(arc) > 4):
                        if (not(arc[0] in scorer)):
                                scorer.update({arc[0] : []})
                                positioner.update({arc[0] : []})
                                fileArcs.update({arc[0] : []})
                                position = 0
                                print arc
                                scorer[arc[0]].append(float(arc[5]))
                                positioner[arc[0]].append(arc[4].lower())
                                fileArcs[arc[0]].append(arc)
                        else:
                                position += 1
                                scorer[arc[0]].append(float(arc[5]))
                                positioner[arc[0]].append(arc[4].lower())
                                fileArcs[arc[0]].append(arc)
                        if (not(arc[4].lower() in indexer)):
                                indexer.update({arc[4].lower() : {}})
                        if (not(arc[0] in indexer[arc[4].lower()])):
                                indexer[arc[4].lower()].update({arc[0] : {}})
                        indexer[arc[4].lower()][arc[0]].update({arc[2]:{'ch':arc[1], 'dur':arc[3], 'score
        return indexer, scorer, positioner, fileArcs

def getOutput(inFile, outFile, queriesFile, snpower):
        indexer, scorer, positioner, fileArcs = getIndexer(inFile)
        indir = './lib/kws/' + queriesFile
        text = open(indir).read()
        entries = text.split('</kwtext>\n  </kw>\n  <kw ')
        output = '<kwslist kwlist_filename="IARPA-babel202b-v1.0d_conv-dev.kwlist.xml" language="swahili"
        entries[0] = entries[0].split('>\n  <kw ')[1]
        entries[len(entries)-1] = entries[len(entries)-1].split('</kwtext>\n  </kw>\n</kwlist>')[0]
        for entry in entries:
                entry = entry.split('">\n    <kwtext>')
                query = entry[1].split(' ')
                kwid = entry[0].split('="')[1]
                print query, kwid
                output +='\n<detected_kwlist kwid="' + kwid + '" oov_count="0" search_time="0.0">'
                output += searchToken(query, indexer, scorer, positioner, fileArcs, snpower)
                output += '\n</detected_kwlist>'
        output += '\n</kwslist>'
        text_file = open("output/"+outFile, "w")
        text_file.write(output)
        text_file.close()

def searchToken(query, indexer, scorer, positioner, fileArcs, snpower):
        result = ''
        tempRes = []
        tempRes2 = []

        fileA = []
        channelA = []
        tbegA = []
        durA = []
        scoreA = []

        for indexWord in range(len(query)):
                if query[indexWord] in indexer:
                        tempRes.append(indexer[query[indexWord]])
```

```
        else:
                return result

if len(tempRes) < 2:
        for file in indexer[query[indexWord]]:
                for arc in indexer[query[indexWord]][file]:
                        dur = indexer[query[indexWord]][file][arc]['dur']
                        score = float(indexer[query[indexWord]][file][arc]['score'])
                        pos0 = int(indexer[query[indexWord]][file][arc]['pos'])
                        fileA.append(file)
                        channelA.append(indexer[query[indexWord]][file][arc]['ch'])
                        tbegA.append(arc)
                        durA.append(dur)
                        scoreA.append(score)
        scoreTotal = sum(i**snpower for i in scoreA)
        for hit in range(len(tbegA)):
                result += '\n'
                result += '<kw file="' + fileA[hit] + '" channel="' + channelA[hit] + '" tbeg="'
        return result

else:
        for file in tempRes[0]:
                validFile = True
                wordI = 1
                while (wordI < len(tempRes) and validFile):
                        if (file in tempRes[wordI]):
                                wordI += 1
                        else :
                                validFile = False
                if (validFile):
                        positions = [i for i, j in enumerate(positioner[file]) if j == query[0]]
                        for position in positions:
                                indexWord = 1
                                validPosition = True
                                while(validPosition and indexWord < len(query)):
                                        #print indexWord,position,file
                                        if position+indexWord < len(positioner[file]):
                                                if (query[indexWord] == positioner[file][position
                                                        indexWord += 1
                                                else:
                                                        validPosition = False
                                        else:
                                                validPosition = False
                                if (validPosition):
                                        tpRes = []
                                        for indexWord in range(len(query)):
                                                tpRes.append(fileArcs[file][position+indexWord])
                                        tempRes2.append(tpRes)
        for tpRes in tempRes2:
                validTpRes = True
                inTpRes = 1
                while(validTpRes and inTpRes < len(tpRes)):
                        if (float(tpRes[inTpRes][2])-float(tpRes[inTpRes-1][2])-float(tpRes[inTpR
                                inTpRes += 1
                        else:
                                validTpRes = False
                if(validTpRes):
                        tbeg = tpRes[0][2]
                        dur = float(tpRes[len(tpRes)-1][2])+float(tpRes[len(tpRes)-1][3])-float(t
                        score = 0
```

```
                                for res in tpRes:
                                        score += float(res[5])
                                score = score / float(len(tpRes))
                                pos0 = indexer[query[0]][tpRes[0][0]][tpRes[0][2]]['pos']
                                posF = indexer[query[len(query)-1]][tpRes[0][0]][tpRes[len(tpRes)-1][2]][
                                fileA.append(tpRes[0][0])
                                channelA.append(tpRes[0][1])
                                tbegA.append(str(tbeg))
                                durA.append(str(dur))
                                scoreA.append(score)
                scoreTotal = sum(i**snpower for i in scoreA)
                for hit in range(len(tbegA)):
                        result += '\n'
                        result += '<kw file="' + fileA[hit] + '" channel="' + channelA[hit] + '" tbeg="'
                return result

def test():
        indexer, scorer, positioner, fileArcs = getIndexer('reference.ctm')
        query = ['elisa']
        return searchToken(query, indexer, scorer, positioner, fileArcs)


def main() :

    parser = argparse.ArgumentParser(description='KWS Indexer.')
    #parser.add_argument('--ta', dest='timalign', action='store', default=0.2,      help='timalign')
    parser.add_argument('--in', dest='inFile', action='store', required=True,
                        help='In File CTM')
    parser.add_argument('--out', dest='outFile', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--queries', dest='queriesFile', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--sn', dest='snpower', action='store', required=True,
                        help='Out File XML')

    args = parser.parse_args()
    getOutput(args.inFile, args.outFile, args.queriesFile, float(args.snpower))


if __name__ == '__main__':
    main()
```

B Grapheme Confusion Matrix

`toGrapheme.py`: script for applying grapheme confusion matrix, by computing the minimum edit distance between OOV and IV proxy keywords. see function `editDistance()` for computing minimim edit ditance, and `getCosts()` for getting confusion matrix.

```
import argparse

def editDistance(s1, s2, costs):

    m=len(s1)+1
    n=len(s2)+1

    tbl = {}
    for i in range(m): tbl[i,0]=i
    for j in range(n): tbl[0,j]=j
    for i in range(1, m):
        for j in range(1, n):
```

```
            if 'sil' in costs[s1[i-1]]:
                    costSil = 1-costs[s1[i-1]]['sil']/costs[s1[i-1]]['totalcount']
            else:
                costSil = 1
            if s1[i-1] == s2[j-1] :
                cost = 0
            else:
                if s2[j-1] in costs[s1[i-1]]:
                        cost = 1-costs[s1[i-1]][s2[j-1]]/costs[s1[i-1]]['totalcount']
                else:
                        cost = costSil

            tbl[i,j] = min(tbl[i, j-1]+costSil, tbl[i-1, j]+costSil, tbl[i-1, j-1]+cost)

    return tbl[i,j]


def getCosts():
        indir = '/home/ct506/Desktop/kws/lib/kws/grapheme.map'
        text = open(indir).read()
        entries = text.split('\n')
        costs = {}
        for entry in entries:
                if (entry!=''):

                        entry = entry.split(' ')
                        hyp = entry[1]
                        ref = entry[0]
                        refcost = float(entry[2])
                        if hyp not in costs:
                                costs[hyp] = {}
                                costs[hyp]['totalcount'] = 0
                        costs[hyp][ref] = refcost
                        costs[hyp]['totalcount'] += refcost
        return costs

def getIV(filename):
        indir = './lib/ctms/' + filename
        text = open(indir).read()
        entries = text.split('\n')
        iv = []
        for entry in entries:
                arc = entry.split(' ')
                if (len(arc) > 4):
                        if (not(arc[4] in iv)):
                                iv.append(arc[4].lower())
        return iv

def transformQueries(iv, outFile, costs, queriesFile):
        oov = 0;
        indir = './lib/kws/' + queriesFile
        text = open(indir).read()
        entries = text.split('</kwtext>\n  </kw>\n  <kw ')
        output = '<kwlist ecf_filename="IARPA-babel202b-v1.0d_conv-dev.ecf.xml" language="swahili" encodi
        entries[0] = entries[0].split('>\n  <kw ')[1]
        entries[len(entries)-1] = kwid='kwid="KW202-00091">\n    <kwtext>kazi ya ko'#entries[len(entries)
        for entry in entries:

                entry = entry.split('">\n    <kwtext>')
                query = entry[1].split(' ')
```

```
                kwid = entry[0].split('="')[1]
                output +='\n  <kw kwid="'+kwid+'">'
                output += '\n    <kwtext>'
                for word in range(len(query)):
                        if (query[word] in iv):
                                output += query[word]
                        else:
                                closestIV = getClosestIV(query[word], iv, costs)
                                output += closestIV
                                oov+=1
                                print query[word], closestIV
                        if (word != len(query)-1):
                                output += ' '
                output +='</kwtext>'
                output += '\n  </kw>'

        output += '\n</kwslist>'
        print oov
        text_file = open('./lib/kws/' + outFile, "w")
        text_file.write(output)
        text_file.close()

def getClosestIV(word, iv, costs):
        costsOOV = []
        for wordIV in iv:
                costsOOV.append(editDistance(word.replace("'",""), wordIV.replace("'",""), costs))
        return iv[costsOOV.index(min(costsOOV))]

def main() :
    parser = argparse.ArgumentParser(description='KWS Indexer.')
    parser.add_argument('--in', dest='inFile', action='store', required=True,
                        help='In File XML')
    parser.add_argument('--out', dest='outFile', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--queries', dest='queries', action='store', required=True,
                        help='Out File XML')


    args = parser.parse_args()
    costs = getCosts()
    print 'Got costs'
    iv = getIV(args.inFile)
    print 'Got IV words'
    print editDistance('lol','pop', costs)
    transformQueries(iv, args.outFile, costs, args.queries)



if __name__ == '__main__':
    main()
```

C System Combination

`combine.py`: Script for performing system combination.

```
import argparse
def combine(hitsFile1, hitsFile2, alpha, beta, outputFile):
        indir1 = './output/' + hitsFile1
        indir2 = './output/' + hitsFile2
        text1 = open(indir1).read()
```

```
        text2 = open(indir2).read()
        entries1 = text1.split('</detected_kwlist>\n')
        entries2 = text2.split('</detected_kwlist>\n')
        output = '<kwslist kwlist_filename="IARPA-babel202b-v1.0d_conv-dev.kwlist.xml" language="swahili"
        entries1[0] = entries1[0].split('<detected_kwlist ')[1]
        entries2[0] = entries2[0].split('<detected_kwlist ')[1]

        for index in range(len(entries1)):
                hits1 = entries1[index].split('\n<kw ')
                hits2 = entries2[index].split('\n<kw ')
                combinedHits = {}
                for indHit in range(1,len(hits1)):
                        hitInfo = hits1[indHit].split(' ')
                        #print hitInfo
                        tbeg = hitInfo[2].split('=')[1]
                        file = hitInfo[0].split('=')[1]
                        key = tbeg.split('"')[1]+file.split('"')[1]
                        #print key
                        combinedHits[key] = {'file':file, 'dur':hitInfo[3].split('=')[1], 'score':alpha*f

                for indHit in range(1,len(hits2)):
                        hitInfo = hits2[indHit].split(' ')
                        tbeg = hitInfo[2].split('=')[1]
                        file = hitInfo[0].split('=')[1]
                        key = tbeg.split('"')[1]+file.split('"')[1]
                        if key in combinedHits:
                                combinedHits[key]['score'] = combinedHits[key]['score']
                +beta*float(hitInfo[4].split('="')[1].split('"')[0])
                        else:
                                print key
                                combinedHits[key] = {'file':file, 'dur':hitInfo[3].split('=')[1],
                'score':beta*float(hitInfo[4].split('="')[1].split('"')[0]), 'tbeg': tbeg}
                if (len(hits1)>1 or len(hits2) >1):
                        output += hits1[0]+'\n'
                else:
                        output += hits1[0]
                for hit in combinedHits:
                        output += '<kw file='+combinedHits[hit]['file']+' channel="1" tbeg=' + combinedHi
                if (index != len(entries1)-1):
                        output += '</detected_kwlist>\n'


        text_file = open('./output/' + outputFile, "w")
        text_file.write(output)
        text_file.close()


def main() :
    parser = argparse.ArgumentParser(description='KWS Indexer.')
    parser.add_argument('--hits1', dest='hits1', action='store', required=True,
                        help='In File XML')
    parser.add_argument('--hits2', dest='hits2', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--alpha', dest='alpha', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--beta', dest='beta', action='store', required=True,
                        help='Out File XML')
    parser.add_argument('--out', dest='out', action='store', required=True,
                        help='Out File XML')
```

```
    args = parser.parse_args()
    combine(args.hits1, args.hits2, float(args.alpha), float(args.beta), args.out)

if __name__ == '__main__':
    main()
```