Spoken Language Processing and Generation
Practical: Keyword Spotting in Swahili

MLMI candidate H801D

Preamble

We implement and test a keyword spotting (KWS) system taking as input the 1-best output from a speech recognition system. We also experiment with morphological decomposition, score normalization, grapheme-confusion matrices and system combination.

1 Introduction

Following instructions from [Gal21b], we implement a KWS system that searches for keywords in a written transcription of audio data produced by a 1-best speech recognition (SR) system. The data comes from the Babel Project (see [Gro]); the language is Swahili.

As this is coursework and following the indications in [Gal21b], we spend more time on the practical details of our implementation than we would in a research article. In particular, we discuss in each section the key points of our Python code, which can be found in its entirety in Appendix A. All experiments were run on 'mlsalt-cpu1'. General theoretic references are [Gal21a], [Woo20] and [GKA17].

2 Swahili and low-resource languages

Performing KWS on a language for which there are few available resources (be it dictionaries or audio data) such as Swahili can be challenging; there are often no preexisting language models (LM), morphological analysers or parts of speech taggers available, and it can be hard to find quality data on which to train the SR system[1]; in turn, the Token Error Rate (WER) will be high, which is correlated with lower KWS performance (see [GKA17]).

Regarding Swahili in particular, its status as a lingua franca in East Africa means that there are many dialects and a great variety of accent, which increases speaker variability and makes the task even more arduous. Moreover, it is a highly agglutinative language, where prefixes are pivotal in the formation of meaning: for example, the prefix "m-", related to the general idea of "person", is used to form "mtoto" - "child", from "utoto" - "childhood".

This results in great lexical variety: in [GBP12], a $400'000$ words vocabulary is reported to yield a typical out-of-vocabulary (OOV) rate of 10.3%. This is made even worse by the fact that only small vocabularies can usually be obtained from small training corpora. High OOV rates, in turn, have a strong negative effect on KWS performance, where missing a word occurrence is typically more severely penalized than a false alarm (see below, or *e.g.* [RSZ+17] for more on that). Hence we can expect high OOV rates to be a major obstacle in what follows.

3 Building and searching an index

3.1 Implementation

Our implementation of a KWS index consists of a class 'Index', and an auxiliary class 'Speech_entry' whose instances represent lines in

[1] Using scraped written web data can help, as explained in [MCS+15] - there are for example about $61'000$ Wikipedia articles in Swahili.

[2] We refer to [Gal21b] for additional details concerning the various formats of files provided.

a .ctm file[2]. The key points are the following:

- The method 'index_CTM_document' of 'Index' reads each line of a provided .ctm document and stores them as instances of 'Speech_entry' in the same order in an attribute called 'formatted_speech'. Whenever there is more than 0.5 seconds of silence between two entries, a special value None is inserted between them. This allows for easy handling of the "0.5s rule" (see [Gal21b]).

- 'index_CTM_document' also updates another attribute, called occurrencee_index'. It is a dictionary whose keys are the words encountered in the .ctm file, and whose values are the positions of (the 'Speech_entry' corresponding to) those words in 'formatted_speech'. For example, if "hello" appears (only) in position 245 and 310 in 'formatted_speech', then occurrence_index["hello"] = [245,310,355].

- 'Index' has another method called 'perform_KWS', which takes as input a list of queries in a .xml, and outputs a list of hits as a new .xml file. To interact easily with the .xml format, we use 'xml.etree.ElementTree'.

- For each query of length n in the input .xml file, 'perform_KWS' considers the locations of its first word (stored in occurrence_index) in 'formatted_speech', and compares the following words with the query. For example, given the query "hello beautiful world", we might have occurrence_index["hello"] = [13,67]. We would then check whether formatted_speech[14] corresponds to "beautiful" and formatted_speech[15] to "world" (and similarly for 67). If so, it is a hit.

- The score of a newly created hit is the product of the confidence score of its constituent words: as we only have access to a 1-best decoding, we consider that the probability $P(w_1, \dots, w_l)$ of a sentence $w_1 \dots, w_l$ occurring is equal to $\prod_{i=1}^{l} P(w_i)$. This would not necessarily be the case if we were using a more sophisticated lattice model: we would have to use the lattice to compute the total probability in a subtler way.

The crucial algorithmic idea here is to store the locations in the decoding of each word in a dictionary. Accessing an entry in a Python dictionary has complexity $O(1)$ on average, and $O(n)$ in the worst case (where n is the size of the dictionary), as the operation uses hash functions (see the Python documentation [Pyt]). This makes the time needed to search a query linear on average in the number of potential hits (which cannot be avoided) once an Index has been constituted, instead of being linear in the size of the audio data (as searching every entry in the data would be), and allows our method to scale well to large data. The code could also be easily modified to operate online, by continuously updating the Index.

3.2 Experiments

In what follows, we use the Maximum Term Weighted Value (MTWV)

$$\text{MTWV} = \max_{\theta} \frac{1}{N} \sum_{w} \left[\frac{N_{hit}(w)}{N_{true}(w)} - \beta \frac{N_{FA}(w)}{T - N_{true}(w)} \right]$$

as our metric (see [Gal21a]), where the sum is over all N queries, $T = 36'000$ is the length of the data (in seconds), $N_{true}(w)$ is the number of occurrences of a word and $N_{hit}(w)$ and $N_{FA}(w)$ are respectively the number of hits and of false alarm at a given threshold θ. The parameter β, equal to 999.9 here, is chosen so that the penalty for a false alarm is not too small (as usually $T >> N_{true}(w)$). Observe that

$$\frac{N_{hit}(w)}{N_{true}(w)} - \beta \frac{N_{FA}(w)}{T - N_{true}(w)} \cong \frac{N_{hit}(w)}{N_{true}(w)} - \frac{N_{FA}(w)}{36}.$$

This means that the penalty for a false alarm is almost constant (and relatively small), while the penalty for a missed hit is much greater for rare words than it is for common ones[3].

We start by testing our KWS system on the reference file[4] 'lib/ctms/reference.ctm' using the following commands:

[3] While this might be reasonable in most cases, one can imagine situations in which another metric would be preferable.
[4] We use the same naming conventions as in the handout [Gal21b].

```
scripts/score.sh results/Q1_hits.xml scoring
scripts/termselect.sh lib/terms/ivoov.map...
results/Q1_hits.xml scoring all
scripts/termselect.sh lib/terms/ivoov.map...
results/Q1_hits.xml scoring iv
scripts/termselect.sh lib/terms/ivoov.map...
results/Q1_hits.xml scoring oov
```

As expected (due to 'reference.ctm' being a perfect decoding), we find a perfect MTWV of 1, as seen in Table 1. The Detection Error Tradeoff (DET) curve generated by the scoring script is a single point - the threshold is inconsequential (as long as it is smaller than 1), as all hits have a score of 1.

all	IV	OOV
1	1	1

Table 1: MTWV per category of words after indexing and searching file 'reference.ctm'.

We then run our system on 'decode.ctm'; results[5] are in Table 2. "Threshold" refers to the optimal threshold chosen by the system to generate the maximum TWV. They are fairly bad; this is not unexpected, given the difficulties mentioned in Section 2. Using a 1-best encoding penalizes us further; switching to a lattice-based system (see *e.g.* [GKA17] for more on that), or at least n-best encoding, would allow us to retain other hypotheses for a given word, many of which would generate correct hits (as ASR systems tend to have high TER of $30\% - 70\%$ in such low-resource scenarios). Observe that the OV MTWV is 0, as words that do not appear in 'decode.ctm' have no chance of being detected.

all	IV	OOV	Threshold
0.319	0.401	0	0.043

Table 2: Indexing and searching 'decode.ctm'.

Note also that the optimal threshold is very low; this is because many hits have a low score (especially for longer queries), due to the ASR system lacking confidence. Another reason for that is the comparably higher penalty for missed hits than for false alarms, especially for rare words, which means that a relatively high false alarm rate in exchange for a higher hit rate will often be an advantageous tradeoff, as illustrated in Figure 1, where performance increases monotonously as the threshold decreases (until about $\theta = 0.043$).

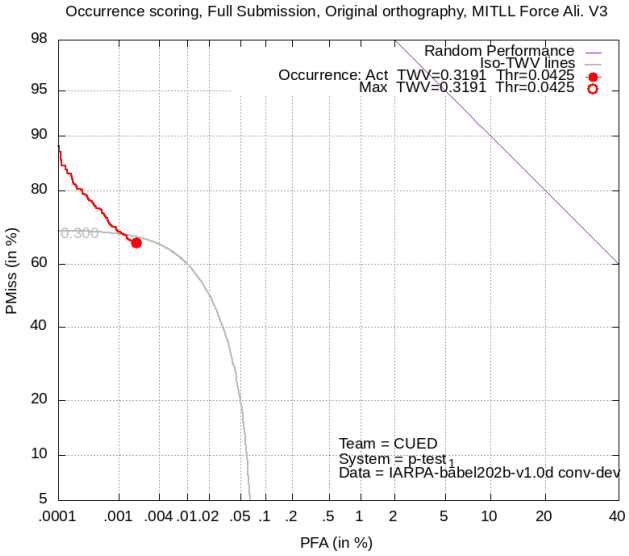

Figure 1: DET curve for 'decode.ctm'.

4 Morphological Decomposition

4.1 Implementation

We use a class called 'Morph_Decomposer', which reads .dct files and stores morphological decomposition rules in dictionary attributes, then apply them to both .ctm files and list of queries in .xml files. We then utilize 'Index' as before to index the data and perform KWS. The key points of our implementation are the following:

- When reading a .dct file, morphological decomposition rules are stored in a dictionary (*e.g.* my_mapping["blacksmith"] = ["black","smith"]). As noted in Section 3, this allows for very fast access (on average), which makes the morphological decomposition efficient.
- We simply replace each word of each query by its decomposition: *e.g.*, "sad honeybird"

[5]Note that due to the relatively small datasets used, differences of less than 0.001 in MTWV should not be considered significant.

might become "sad honey bird".

- If an entry in a .ctm file corresponds to a word that has confidence score p and duration T and is decomposed into n subwords, there will be n entries in the new .ctm file. Moreover, each of them will have a score of $p^{\frac{1}{n}}$ and a duration of $\frac{T}{n}$, so that the score and duration of the sequence of entries is the same as that of the original word. If we had access to a lattice model, we could assign durations and scores to the new entries in a less simplistic fashion.

4.2 Experiments

We apply morphological decompositions from 'morph.kwslist.dct' and 'morph.dct' to 'queries.xml' and 'decode.ctm', and search both the decomposed 'decode.ctm' file (which we refer to as 'decode+morph'), and the output from a morph-based system 'decode-morph.ctm'. Results[6] can be seen in Table 3.

	all	IV	OOV
decode+morph	0.315	0.391	0.018
decode-morph	0.318	0.383	0.068

Table 3: MTWV for 'decode+morph' and 'decode-morph'.

We see a small decrease in MTWV for IV words compared to Table 2, compensated by an increase for OOV words, leading to an overall slight increase in performance.

The reason for that is that some OOV words were decomposed into subwords of which some were IV and got detected: *e.g.* "abandon" might have been falsely decoded as "abandoned", which would then be decomposed as "abandon" + "ed" and would generate a hit for "abandon", leading to an increase in hits (the same can happen to IV words). Conversely, correctly detected words can generate false alarms once decomposed, either within the word itself ("abandoned" yielding a hit for "abandon") or across several words ("abandoned ward" yielding a hit for "edward"). This increase both in hits and in false alarms can be verified in the detailed '*-res.txt' file.

It is no surprise that the dedicated morph-based system would slightly outperform an a posteriori application of morphological decomposition, or that neither system is very performant: a good morph system would need to incorporate more language-specific rules than could be learned from the low-resources available, especially for an agglutinative language like Swahili (see [CHK+07] for more on that). Taking into considerations how morphs can be joined ("abandon-ed", but not "ed-abandon"), as in [RSZ+17], could also improve performance.

5 Score Normalization

5.1 Implementation

We implemented three different methods of score normalization. Let $\gamma_1, \dots, \gamma_n$ be the scores of the hits associated to a given query w. We consider:

- Sum-to-one (STO) normalization, where we set as new scores

$$\tilde{\gamma_i} := \frac{\gamma_i^\alpha}{\sum_{i=1}^n \gamma_i^\alpha}$$

for some $\alpha > 0$. STO has the effect of increasing the scores of hits for rare words (for which missed hits are more penalized); a large α will increase the divide between confident hits for a given query and those with a low score, and conversely for a small α.

- Keyword specific thresholding (KST), where we define

$$thr(w) := \frac{\beta\alpha \sum_{i=1}^n \gamma_i}{T + (\beta - 1)\alpha \sum_{i=1}^n \gamma_i},$$

where $\alpha > 0$ is chosen such that $\alpha \sum_{i=1}^n \gamma_i$ is a good approximation of $N_{true}(w)$ - in what follows, we use $\alpha = 1.5$, as recommended in [WM14]. As in [KST+13], we then set

$$\tilde{\gamma_i} := \gamma^{-1/\log(thr(w))}.$$

This is equivalent to using $thr(w) \cdot \theta$ as a threshold for the hits associated to w, where θ is a global threshold. As STO, KST increases the score of rare words, but it keeps

[6] There might be some variations depending on the way special characters and letter case are handled.

the score of very common words from becoming so small that they are never detected (see [WM14] for details).

- Query length (QL) normalization, where we let
$$\tilde{\gamma_i} := \gamma^{\frac{1}{l(w)}}$$
and $l(w)$ is the average length (in seconds) of the hits associated to w. As explained in [MCC+13], this counters the tendency for longer queries to return low score hits (as their score will be the product of the score of the words in the query). One can also replace $l(w)$ by the number of words in the query, as in [MRS07].

5.2 Experiments

We first apply STO with different values of α to 'decode-morph.ctm'. Results can be found in Table 4, with the best MTWV obtained highlighted in blue. We see that the choice of α has little impact on performance[7]; from now on, we use $\alpha = 1$.

α	0.5	1	1.5
MTWV	0.3258	0.3260	0.3228
Threshold	0.053	0.057	0.039
α	2	2.5	3
MTWV	0.3205	0.3208	0.3209
Threshold	0.007	0.005	0.002

Table 4: MTWV and optimal threshold for 'decode-morph+STO' for different values of α.

The results of applying STO, KST and QL normalization to 'decode', 'decode+morph' and 'decode-morph' are illustrated in Table 5.

	STO	KST	QL
decode	0.320	0.320	0.319
decode+morph	0.318	0.314	0.314
decode-morph	0.326	0.320	0.318

Table 5: MTWV for various decodings and score normalization methods.

We see that score normalization yields a very small increase (for the reasons detailed above) in performance, that 'decode-morph' remains the best decoding, and that STO slightly outperforms other systems. As expected, different normalization methods also yield very different optimal thresholds (0.057 for 'decode-morph+STO' and 0.474 for 'decode-morph+KST').

We would need additional testing and larger datasets to reach definitive conclusions regarding the compared efficiency of those systems, as differences are very small here. KST is reported as having a small but significant advantage in [WM14], especially if lattices are used to estimate $N_{true}(w)$ (instead of $\alpha \sum_{i=1}^{n} \gamma_i$, which is shown to be a poor approximation).

6 Graphemic segmentation

6.1 Implementation

We created a class called 'Grapheme_Based_Mapper', whose key functionalities are the following:

- It can store a confusion matrix from a .map file as a dictionary of dictionaries, where self.confusion_matrix["a"]["b"] is the log probability of "b" being observed when "a" was the correct grapheme, estimated as the number of times such a substitution was recorded in the .map file.

- It can read a list of queries in an .xml file and replace each OOV word with the closest IV word for the edit distance adapted to the confusion matrix (see below). This mapping is stored in a dictionary for future uses.

Our approach follows that of [XWM14]. What we call the adapted edit distance is the usual edit distance, where the deletion, insertion and substitution penalties are given by confusion_matrix["a"]["sil"], confusion_matrix["sil"]["a"] and confusion_matrix["a"]["b"] respectively (here, "a" is the reference and "b" was observed). We compute it using a variant of the Wagner-Fisher algorithm (see [Jur]). Note that the edit distance is not a proper distance - in particular, it is not symmetric. In our implementation, we treat the query as the reference, and the words in the decoded data as the potentially faulty observations.

[7] In fact, optimal α depends on the decoding used ('decode', 'decode+morph' or 'decode-morph').

6.2 Experiments

We apply our graphemic segmentation system to 'queries.xml' and search 'decode.ctm', with and without STO score normalization. Results can be found in Table 6.

	all	IV	OOV	Thres.
Grapheme	0.314	0.401	-0.024	0.043
Grapheme+STO	0.321	0.402	0.007	0.020

Table 6: MTWV for graphemic segmentation applied to 'decode.ctm', with and without STO score normalization.

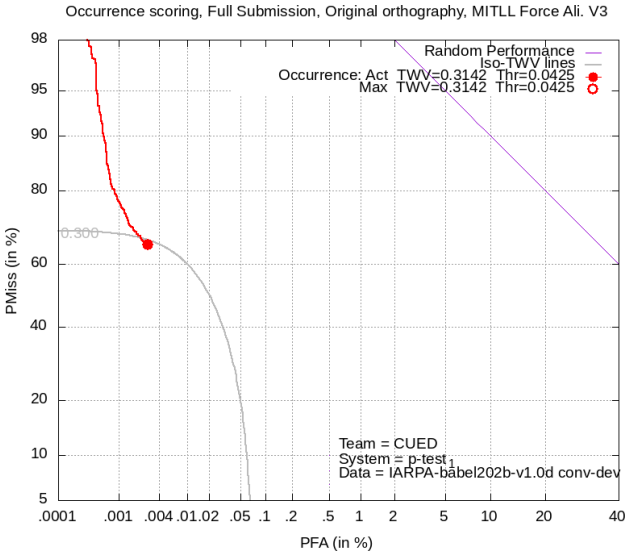

Figure 2: DET curve for 'decode.ctm' and 'queries.xml' with graphemic segmentation. Note how the false alarm rate remains very high even for large thresholds.

Results for IV are left unchanged. The number of hits for OOV words increases compared to using 'decode.ctm' alone, but it is countered by a drastic increase in the number of false alarms, as OOV words are falsely mapped to unrelated IV words (this can also be seen in Figure 2). This results in a slightly lower performance without normalization, and a very slightly better one with normalization. Mapping OOV words only to IV words that are closer for the adapted distance than a certain threshold (and leaving them untouched otherwise) might improve our system by reducing the false alarm rate.

Our method is too crude (for example, it takes no contextual information into account when comparing two characters), but grapheme-based systems can be very efficient (see [TWF+08]), especially those using lattices: they allow for a more probabilistic description of the output, even at subword level (instead of making hard decisions), which in turns allows for more flexible modelling - in particular, it is well suited to representing multiple pronunciations of the same word (a key issue with Swahili, as noted in Section 2). However, efficient lattice pruning techniques are required (see *e.g.* [GKA17]), as working with very small units such as graphemes means that lattices can quickly become gigantic.

7 System Combination

7.1 Implementation

To implement system combination, we use two classes: 'Hit', which represents a hit for a given query, and 'System_Combiner'. The key elements are the following:

- An instance of 'Hit' stores the basic information associated to a hit (time, duration, etc.), as well as the scores from several systems that have detected that hit. We overloaded the '__eq__' function so that two instances are deemed equivalent if they overlap by more than 30% of the duration of the shortest one.
- 'System_Combiner' reads several .xml files, each the output of a different KWS system, and merges for each query the corresponding lists of hits. When two hits overlap (see above), they are combined; the new hit stores all the scores associated to its parents[8].
- Merging several lists of hits is only of linear complexity in the number of hits (as opposed to quadratic if done naively), thanks

[8]Note that the order in which hits are compared and merged can add some (very limited) randomness to the final result.

to using as before hash functions and dictionaries judiciously .

- Finally, a combined score is computed for each combined Hit.

Depending on the parameter chosen, 'System_Combiner' can combine scores (for a given hit) in four different ways, with the convention that the score associated to a system that has not detected the hit is 0. The first three methodologies can be found in [MCC+13]. "CombSum" simply sums the scores γ_s over all systems s. "CombMNZ" lets the new score be

$$\tilde{\gamma} := K \sum_s \gamma_s,$$

where the sum is over all systems and K is the number of systems that have non-zero score for that hit. Finally, "WCombMNZ" lets the new score be

$$\tilde{\gamma} := K \sum_s \gamma_s \cdot \frac{\mathrm{MTWV}(s)}{\sum_r \mathrm{MTWV}(r)},$$

where $\mathrm{MTWV}(r)$ is the MTWV of system r for the same data and list of queries. We also implemented a method from [KST+13], with which

$$\tilde{\gamma} := \sum_s \gamma_s \cdot 2^{\mathrm{MTWV}(s) - \max \mathrm{MTWV}},$$

where $\max \mathrm{MTWV}$ is the MTWV of the best performing system, and MTWVs are expressed in percents (*e.g.* 16 instead of 0.16). We denote it as "Power2".

7.2 Experiments

We first test CombSUM combined with STO normalization on the hits obtained previously from 'decode.ctm' and 'decode-morph.ctm'. Following [MCC+13], we apply score normalization either before combination (N+C), after (C+N), before and after (N+C+N), or not all (C). Results can be found in Table 7.

We observe an increase both for IV and OOV words, the effect being stronger for OOV queries. Score combination has the effect of making the system more robust by taking into account a wider variety of features, as it combines the insight from different systems - the more different they are (for example GMM-based and DNN-based systems), the better. As expected, normalizing before combining yields the best performance, as there is no reason for the scores of two different systems to be similarly calibrated.

	CombSUM		
	all	IV	OOV
C	0.348	0.421	0.068
C+N	0.358	0.433	0.068
N+C	0.363	0.439	0.068
N+C+N	0.363	0.439	0.068

Table 7: MTWV for STO normalization ("N") and CombSUM combination ("C") applied to the results from 'decode.ctm' and 'decode-morph.ctm'.

We then analyze in further details the effects of the various combination techniques implemented on the provided lists of hits 'morph.xml', 'word.xml' and 'word-sys2.xml', whose pre-combination MTWV are in Figure 8.

	all	IV	OOV
morph	0.360	0.430	0.089
morph + STO	0.520	0.559	0.367
word	0.399	0.501	0
word + STO	0.460	0.579	0
word-sys2	0.403	0.507	0
word-sys2 + STO	0.465	0.585	0

Table 8: MTWV with and without STO normalization for 'morph.xml', 'word.xml' and 'word-sys2.xml'.

As predicted in the previous sections, those lattice-based systems achieve much higher performance than the 1-best outputs considered so far; they also seem to benefit much more from score normalization (especially 'morph.xml'), perhaps because the differences between their scores are more meaningful due to their better quality (thus making the choice of a good threshold more impactful).

We observe in Figure 3 the effects of the various combination techniques implemented, and how they interact with STO normalization. We see that 'CombSUM', 'CombMNZ' and

MTWV												
	Power2			CombSUM			CombMNZ			WCombMNZ		
	all	IV	OOV	all	IV	OOV	all	IV	OOV	all	IV	OOV
C	0.412	0.518	0	0.413	0.497	0.087	0.413	0.499	0.077	0.412	0.498	0.078
C+N	0.549	0.599	0.352	0.548	0.596	0.362	0.557	0.606	0.365	0.554	0.606	0.356
N+C	0.528	0.570	0.364	0.556	0.605	0.366	0.554	0.607	0.351	0.555	0.607	0.351
N+C+N	0.527	0.569	0.363	0.555	0.605	0.364	0.556	0.605	0.365	0.555	0.605	0.359

Figure 3: MTWV for different system combination techniques applied to 'morph.xml', 'word.xml' and 'word-sys2.xml'. "N" refers to STO score normalization.

'WCombMNZ' perform similarly, with more experiments being needed to make definitive comparisons. 'Power2' is much less efficient.

As before, we see that performing score normalization is crucial, though it surprisingly seems to matter little whether we do it before or after combination. It might be because the number of systems having detected a hit has comparatively more impact (especially for 'CombMNZ' and 'WCombMNZ') than the precise score each system attributes to it, or because those systems happen to output scores that tend to be in the same range.

The best MTWV of 0.557 obtained, highlighted in blue, is in line with that of 0.551 from [MCC+13], modulo small differences due to implementation choices. We see that the corresponding system manages to have "the best of both worlds": it retains the performance of 'morph.xml' on OOV queries, while also maintaining (and even improving upon) the performance of 'word-sys2.xml' on IV queries. This shows that information is efficiently combined.

Figure 4 shows the DET curve of the CombMNZ+STO combined system. We see that the false alarm rate is actually higher than for individual systems, but that it is compensated by a much lower miss rate. The reason for this is simply that the combined set of hits is the union of those of the combined systems: more hits, both correct and incorrect, are detected.

Finally, we observe in Table 9, where improvement in MTWV between 'morph.xml' and 'morph.xml+STO' and between 'morph.xml+STO' and 'CombMNZ+STO' is shown for different query lengths, that score normalization improves MTWV for all query lengths, but that the effect is stronger for longer queries (as their unnormalized score tends to be very small). Combination improves short and medium length queries, with the greatest "per query" improvement being for medium length queries. It can have a negative on long queries due to the increased false alarm rate mentioned above.

QL ↓	# queries	% STO improv	% C. improv
1	288	16.1	43.4
2	168	12.9	29.0
3	20	38.8	60.1
4	10	0	-2.8
5	2	0	0

Table 9: "% STO improv" measures the improvement in MTWV from 'morph.xml' to 'morph.xml+STO' for queries of various lengths. "% C. improv" measures the improvement from 'morph.xml+STO' to 'CombMNZ+STO'.

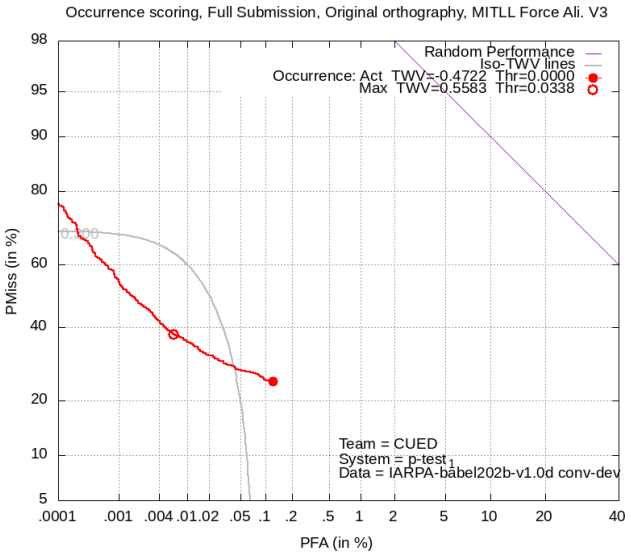

Figure 4: DET curve for 'CombMNZ+STO'.

8 Conclusion

We have seen the importance of score normalization and system combination for keyword spotting, as well as the challenges inherent to working with low-resource languages. We have also implicitly observed the limitations of 1-best decoding systems for KWS. Some of our conclusions would need to be verified on larger datasets, as performance differences between methods were often small.

Given more time, we could have tested subtler machine-based system combination techniques, such as those described in [KST+13]. Directly combining ASR systems, rather than their output (as suggested in [Gal21a]), might also be interesting.

References

[CHK+07] M. Creutz, T. Hirsimäki, M. Kurimo, A. Puurula, J. Pylkkönen, V. Siivola, M. Varjokallio, E. Arisoy, M. Saraclar, and A. Stolcke. Analysis of morph-based speech recognition and the modeling of out-of-vocabulary words across languages. *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings*, 2007.

[Gal21a] M. Gales. Lecture notes in spoken language processing and generation, 2020-2021. `mjfg@eng.cam.ac.uk`.

[Gal21b] M. Gales. Spoken language processing and generation practical handout. `https://www.vle.cam.ac.uk/course/view.php?id=121351&sectionid=3233751`, 2020-2021.

[GBP12] H. Gelas, L. Besacier, and F. Pellegrino. Developments of swahili resources for an automatic speech recognition system. *Spoken Language Technologies for Under-Resourced Languages*, 2012.

[GKA17] M. Gales, K. Knill, and Ragni A. Low-resource speech recognition and keyword-spotting. *International Conference on Speech and Computer*, 2017.

[Gro] CUED Speech Group. Babel Program. http://mi.eng.cam.ac.uk/ mjfg/BABEL/index.html.

[Jur] D. Jurafsky. Minimum edit distance. https://web.stanford.edu/class/cs124/lec/med.pdf.

[KST+13] D. Karakos, R. Schwartz, S. Tsakalidis, L. Zhang, S. Ranjan, T. Ng, R. Hsiao, G. Saikumar, I. Bulyko, L. Nguyen, J. Makhoul, F. Grezl, M. Hannemann, M. Karafiat, I. Szoke, K. Vesely, L. Lamel, and V. Le. Score normalization and system combination for improved keyword spotting. *IEEE Workshop on Automatic Speech Recognition and Understanding*, 2013.

[MCC+13] J. Mamou, J. Cui, X. Cui, M. Gales, B. Kingsbury, K. Knill, L. Mangu, D. Nolden, M. Picheny, B. Ramabhadran, R. Schlüter, A. Sethy, and P. Woodland. System combination and score normalization for spoken term detection. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[MCS+15] G. Mendels, E. Cooper, V. Soto, J. Hirschberg, M. Gales, K. Knill, A. Ragni, and H. Wang. Improving speech recognition and keyword search for low resource languages using web data. *Interspeech 2015 - 16th Annual Conference of the International Speech Communication Association*, 2015.

[MRS07] J. Mamou, B. Ramabhadran, and O. Siohan. Vocabulary independent spoken term detection. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, 2007.

[Pyt] Python. Time complexity and data structures. https://wiki.python.org/moin/TimeComplexity.

[RSZ+17] A. Ragni, D. Saunders, P. Zahemszky, J. Vasilakes, M. Gales, and K. Knill. Morph-to-word transduction for accurate and efficient automatic speech recognition and keyword search. *ICASSP*, 2017.

[TWF+08] J. Tejedor, D. Wang, J. Frankel, S. King, and J. Colás. A comparison of grapheme and phoneme-based units for spanish spoken term detection. *Speech Communication*, 2008.

[WM14] Y. Wang and F. Metze. An in-depth comparison of keyword specific thresholding and sum-to-one score normalization. *Interspeech 2014 - 15th Annual Conference of the International Speech Communication Association*, 2014.

[Woo20] P. Woodland. Lecture notes in speech recognition, 2020. `Woodlandpcw@eng.cam.ac.uk`.

[XWM14] D. Xu, Y. Wang, and F. Metze. Em-based phoneme confusion matrix generation for low-resource spoken term detection. *IEEE Spoken Language Technology Workshop (SLT)*, 2014.

A Code

A.1 Index

```
import xml.etree.ElementTree as ET
import numpy as np
from collections import OrderedDict

class Speech_entity():
        """
        Represents an entry in a CTM ASR output file
        The "entry" argument should be a string (en entire line of the CTM file)
        """
        def __init__(self,entry):
                split_entry=entry.split()
                self.file_name=split_entry[0]
                self.channel = split_entry[1]
                self.tbeg=float(split_entry[2])
                self.tdur=float(split_entry[3])
                self.tend=self.tbeg+self.tdur
                self.token=split_entry[4].lower()
                self.posterior=float(split_entry[5])

        def get_values(self):
                return [self.file_name,self.channel,self.tbeg,...
                self.tdur,self.tend,self.token,self.posterior]

class Index():
        """
        Main indexing class
        Its attributes are:
        - formatted_speech : a list whose entries are either Speech_entity or None  ,
        where each entry represents either a word/subword with associated data (beginning time, etc.),
        or a silence of more than 0.5 second (in the case of None)
        (this makes the identification of phrases easier)
        - occurence_index: a dictionary where each word is mapped to the list of indices
        of the entries in formatted_speech where it occurs
        (example: occurence_index["abandon"]=[2, 36])
        Its main methods are:
        - index_CTM_document, which takes as input the path to a CTM ASR output file
        and stores its entries as Speech_entity objects in formatted_speech
        (it also adds None entries to represent silences of >0.5s),
        as well as creating a dictionary indexing the location of each word in the file,
        and storing it in occurence_index
        - perform_KWS, which takes as input the path to a XML file which is a list
        of queries and the path to the desired output file,
        and creates a XML file at that location which contains the list of hits
        corresponding to the queries and the current content of formatted_speech and occurence_index
        """
        def __init__(self):
                self.formatted_speech=[]
                self.occurence_index=dict()

        def get_occurence_index(self):
                return self.occurence_index
```

```
def index_CTM_document(self,path_to_CTM_document):
        self.formatted_speech=[]
        self.occurence_index=dict()
        for line in open(path_to_CTM_document,"r"):
                entry = Speech_entity(line)
                if self.formatted_speech!=[] and ...
                (entry.tbeg - self.formatted_speech[-1].tend >0.5 ) :
                        self.formatted_speech.append(None)
                self.formatted_speech.append(entry)
                if self.occurence_index.get(entry.token) != None:
                        self.occurence_index[entry.token].append(len(self.formatted_speech)-1)
                else:
                        self.occurence_index[entry.token]= [len(self.formatted_speech)-1]

def perform_KWS(self,path_to_XML_query_list,path_to_XML_output):
        tree=ET.parse(path_to_XML_query_list)
        root = tree.getroot()
        for kw in root:
                kw.tag = "detected_kwlist"
                kw.attrib["oov_count"]="0"
                kw.attrib["search_time"]= "0.0"
                query_text=kw[0].text.split()
                kw.remove(kw[0])
                if self.occurence_index.get(query_text[0]) != None:
                        for partial_hit in self.occurence_index[query_text[0]]:
                            possible_hit = self.formatted_speech[partial_hit:partial_hit+...
                            len(query_text)]
                            tokens_in_file = [entry.token if entry!= None else None...
                            for entry in possible_hit]
                            if tokens_in_file == query_text:
                                hit = ET.SubElement(kw, "kw")
                                total_proba = np.prod([entry.posterior if entry!= ...
                                None else 0 for entry in possible_hit])
                                hit.attrib = ...
                                OrderedDict([("file", possible_hit[0].file_name),...
                                ("channel" , possible_hit[0].channel),...
                                ("tbeg" , str(possible_hit[0].tbeg)),...
                                ("dur",str(np.round(possible_hit[-1].tend -...
                                possible_hit[0].tbeg,2))),...
                                ("score",str(np.round(total_proba,6))),("decision","YES")])
                                # possible indentation problems
                                hit.tail = "\n"
        root.tag = "kwslist"
        del root.attrib["ecf_filename"],root.attrib["language"], root.attrib["encoding"],...
        root.attrib["compareNormalize"], root.attrib["version"]
        root.attrib = ...
        OrderedDict([("kwlist_filename","IARPA-babel202b-v1.0d_conv-dev.kwlist.xml"),...
        ("language","swahili"),("system_id","")])
        tree.write(path_to_XML_output)
```

Code 1: Classes 'Index' and 'Speech_entity'.

A.2 Morphological Decomposer

```
import xml.etree.ElementTree as ET
import numpy as np
from collections import OrderedDict
from Index import Speech_entity

class Morph_Decomposer():
        """
        Stores morphological decomposition mappings from .dct files and ...
        performs decomposition on CTM ASR output files and XML query lists files

        Mappings are stored using load_decomposition_mapping_decoded_speech and ...
        load_decomposition_mapping_query_list in self.decomposition_mapping_decoded_speech ...
        and self.decomposition_mapping_query_list as dictionaries
        (where for example self.decomposition_mapping_decoded_speech["weekend"] = ...
        ["week","end"])

        morph_decompose_CTM_decoding and morph_decompose_XML_queries then read a CTM or XML file ...
        and output a file of the same format where morphological decomposition has been applied

        When a word entry in a CTM file which had a duration of T and a posterior probability...
        of p gets decomposed into n subwords, each is attributed a duration of T/n and...
        a posterior probability of p^(1/n), so that the sequence of subwords has
        the same duration and posterior probability as the original word
        """
        def __init__(self):
                self.decomposition_mapping_decoded_speech=dict()
                self.decomposition_mapping_query_list=dict()
                self.vocabulary=[]

        def load_decomposition_mapping_decoded_speech(self,path_to_dictionary_document):
                self.decomposition_mapping_decoded_speech =...
                self.read_dct_output_dictionary(path_to_dictionary_document)

        def load_decomposition_mapping_query_list(self,path_to_dictionary_document):
                self.decomposition_mapping_query_list = ...
                self.read_dct_output_dictionary(path_to_dictionary_document)

        def read_dct_output_dictionary(self,path_to_dictionary_document):
                decomposition_dictionary = dict()
                f = open(path_to_dictionary_document,"r")
                for line in f:
                        decomposition_dictionary[line.split()[0]]=line.split()[1:]
                return decomposition_dictionary

        def morph_decompose_CTM_decoding(self,path_to_CTM_input,path_to_CTM_output):
                f_input = open(path_to_CTM_input,"r")
                f_output =open(path_to_CTM_output,"w")
                for line in f_input:
                        entry = Speech_entity(line)
                        [file_name, channel, tbeg, tdur, tend,word, posterior] = entry.get_values()
                        if word in self.decomposition_mapping_decoded_speech:
                                decomposed_word = self.decomposition_mapping_decoded_speech[word]
                                n = len(decomposed_word)
```

```
                                decomposed_posterior = str(np.round(np.power(posterior,1/n),6))
                                decomposed_duration = np.round(tdur/n,2)
                                for index, subword in enumerate(decomposed_word):
                                    new_line = " ".join([file_name, channel,...
                                    str(tbeg + index*decomposed_duration),...
                                    str(decomposed_duration),subword,decomposed_posterior]) + "\n"
                                    f_output.write(new_line)
                        else:
                                f_output.write(line)

        def morph_decompose_XML_queries(self,path_to_XML_input,path_to_XML_output):
                tree=ET.parse(path_to_XML_input)
                root = tree.getroot()
                for kw in root:
                        kw[0].text =...
                        " ".join([" ".join(self.decomposition_mapping_query_list[word])...
                        if word.lower() in self.decomposition_mapping_query_list...
                        else word.lower() for word in kw[0].text.split()])
                tree.write(path_to_XML_output)
```

Code 2: The 'Morphological_Decomposer' class.

A.3 Grapheme Segmenter

```
import xml.etree.ElementTree as ET
import numpy as np
from collections import OrderedDict
from Index import Speech_entity

class Grapheme_Based_Mapper():
        """
        Stores vocabulary from a .dct file and confusion matrix from a .map file,
        and replaces OOV words in queries with closest IV words. More precisely:

        - self.confusion_matrix is a dictionary of dictionaries. Example:
        - self.confusion_matrix["a"]["t"] = log(P("t" recognized| "a" reference))
        - self.vocabulary is a set (for O(1) access on average)
        - self.map_to_proxy_IV_words is a dictionary that stores
        (in order not to do the same computations twice)
        the closest IV word associated to an OOV word (once it has appeared in a query)

        - load_confusion_matrix reads a .map file and stores the information in self.confusion_matrix
        - read_dct_get_vocabulary reads a .dct file
        (which maps IV words to their morphological decomposition)
        and stores the vocabulary in self.vocabulary (the morphological decomposition is ignored)
        - map_XML_queries_into_proxy_IV_XML_queries reads a XML file of queries and
        replaces every OOV word by the closes IV word using self.closest_IV_word
        - closest_IV_word(word) finds the closest word in self.vocabulary for the distance
        given by self.distance
        - distance(word1,word2) computes the log likelihood of word2 being recognized
        when word1 was said (it is not really a distance; in particular, it is not symmetric)
```

```
The distance is computed using the confusion matrix
and a variant of the Wagner-Fisher algorithm

"""
def __init__(self):
        self.confusion_matrix=dict()
        self.vocabulary={}
        self.map_to_proxy_IV_words=dict()

def load_confusion_matrix(self,path_to_grapheme_confusion_file):
        self.confusion_matrix=dict()
        grapheme_map = open(path_to_grapheme_confusion_file,"r")
        for line in grapheme_map:
                ref_grapheme = line.split()[0]
                obs_grapheme = line.split()[1]
                count = float(line.split()[2])
                if ref_grapheme not in self.confusion_matrix:
                        self.confusion_matrix[ref_grapheme] = dict()
                        self.confusion_matrix[ref_grapheme]["total_count"] = 0
                self.confusion_matrix[ref_grapheme][obs_grapheme] = count
                self.confusion_matrix[ref_grapheme]["total_count"] += count
        for grapheme in self.confusion_matrix:
                for observation in self.confusion_matrix[grapheme]:
                        if observation != "total_count":
                            self.confusion_matrix[grapheme][observation] =...
                            np.log((self.confusion_matrix[grapheme][observation])/...
                            (self.confusion_matrix[grapheme]["total_count"]))

# rmk: non-symetric, word1 is the reference word and word2 is the observed word
def distance(self,word1,word2):
        m1 = len(word1)
        m2 = len(word2)
        D = dict()
        for i in range(m1+1):
                for j in range(m2+1):
                        D[i,j] = 0
        for i in range(1,m1+1,1):
                        D[i,0] = D[i-1,0] + self.deletion_penalty(word1[i-1])
        for j in range(1,m2+1,1):
                D[0,j] = D[0,j-1] + self.insertion_penalty(word2[j-1])
        for i in range(1,m1+1,1):
                for j in range(1,m2+1,1):
                    D[i,j] = max(D[i-1,j]+self.deletion_penalty(word1[i-1]), D[i,j-1] +...
                    self.insertion_penalty(word2[j-1]), D[i-1,j-1] +...
                    self.substitution_penalty(word1[i-1],word2[j-1]))
        return -D[m1,m2]

def deletion_penalty(self,character):
        if "sil" in self.confusion_matrix[character]:
                return self.confusion_matrix[character]["sil"]
        else:
                return -40
def insertion_penalty(self,character):
        if character in self.confusion_matrix["sil"]:
                return self.confusion_matrix["sil"][character]
        else:
```

```
                return -40
        def substitution_penalty(self,character1,character2):
                if character2 in self.confusion_matrix[character1]:
                        return self.confusion_matrix[character1][character2]
                else:
                        return -40

        def read_dct_get_vocabulary(self,path_to_dictionary_document):
                vocabulary=[]
                f = open(path_to_dictionary_document,"r")
                for line in f:
                        vocabulary.append(line.split()[0])
                # to make access O(1) on average
                self.vocabulary = set(vocabulary)

        def map_XML_queries_into_proxy_IV_XML_queries(self,path_to_XML_input,path_to_XML_output):
                tree=ET.parse(path_to_XML_input)
                root = tree.getroot()
                for kw in root:
                        query_words = kw[0].text.split()
                        new_query_words = []
                        for word in query_words:
                                if word not in self.vocabulary:
                                        word = word.lower()
                                        word = "".join(letter for letter in word if letter.isalnum())
                                        if word not in self.map_to_proxy_IV_words:
                                                self.map_to_proxy_IV_words[word] =...
                                                self.closest_IV_word(word)
                                        new_query_words.append(self.map_to_proxy_IV_words[word])
                                else:
                                        new_query_words.append(word)
                        kw[0].text = " ".join(new_query_words)
                tree.write(path_to_XML_output)

        def closest_IV_word(self,word):
                distances=dict()
                for IV_word in self.vocabulary:
                        distances[IV_word]= ...
                        self.distance(word.replace("'",""),IV_word.replace("'",""))
                return min(distances, key=distances.get)
```

Code 3: The 'Grapheme_Based_Mapper' class.

A.4 Score Normalizer

```
import xml.etree.ElementTree as ET
import numpy as np
from collections import OrderedDict
from Index import Speech_entity

def Normalize_score(path_to_input_XML_list_of_hits, path_to_output_XML_list_of_hits, ...
normalization_method = "STO", alpha = 1, T = 10*60*60, beta = 999.9):
        """
        Opens an XML file that contains a list of KWS hits,
        and performs (depending on the normalization_method argument) either
        - "STO": Sum-To-One normalization of the scores (with parameter alpha>0)
        - "KST": Keyword-specific thresholding (with parameter alpha>0,
        total duration T (in seconds), and weight beta)
        - "QL": Query length normalization
        Outputs the result in a new XML file
        """
        tree=ET.parse(path_to_input_XML_list_of_hits)
        root = tree.getroot()
        for detected_kwlist in root:
                if len(detected_kwlist) != 0:
                        if normalization_method == "STO":
                            scores=np.array([float(kw.attrib["score"]) for kw in detected_kwlist])
                            normalization_factor =  np.sum(scores**alpha)+10**(-6)
                            for  kw in detected_kwlist :
                                kw.attrib["score"]= str(np.round( float(kw.attrib["score"])**alpha...
                                / normalization_factor , 6))
                        elif normalization_method == "KST":
                            posterior_sum = np.sum( np.array ([float(kw.attrib["score"])...
                            for kw in detected_kwlist])  )
                            log_kw_specific_tresh = np.log( beta * alpha * posterior_sum/...
                            (T+ (beta - 1)*alpha * posterior_sum) )
                            for  kw in detected_kwlist :
                                kw.attrib["score"]=str(np.round(float(kw.attrib["score"])**(-1/...
                                log_kw_specific_tresh) ,6))
                        elif normalization_method == "QL":
                            average_duration = np.mean( np.array ([float(kw.attrib["dur"])...
                            for kw in detected_kwlist])  ) + 10**(-6)
                            for  kw in detected_kwlist :
                                kw.attrib["score"]=str(np.round(float(kw.attrib["score"])**(1/...
                                average_duration),6))
                        else:
                            print("Choose valid normalization method")
        tree.write(path_to_output_XML_list_of_hits)
```

Code 4: The 'Normalize_score' function.

A.5 System Combiner

```
import xml.etree.ElementTree as ET
import numpy as np
from collections import OrderedDict
import copy

class Hit():
        """
        Represents a hit from a query
        - self.scores is a dictionary which contains the score
        for each of the system that has recognized this hit
        (example: self.scores = {"System1":0.5, "System2":0.6})
        - We overload the __eq__ function so that two hits are considered equal
        if their intervals overlap by more than 30% of the shortest of the two
        - self.combine_hits combine two hits: the resulting hit will have the basic
        characteristics of the hit that contained the highest score in its self.scores,
        and the new self.scores is the union of the self.scores of the two hits
        - self.combined_score takes as argument a dictionary that specifies the
        MTWV of each of the system for which it has a score in self.scores,
        and computes the combined score out of self.scores using the methodology
        provided as its second argument.
        Options are: "Power2", "CombSUM", "CombMNZ" and "WCombMNZ"
        """
        def __init__(self, info, scores):
                self.file_name = info[0]
                self.channel_name = info[1]
                self.tbeg = float(info[2])
                self.tdur = float(info[3])
                self.scores = scores

        def __eq__(self,other_Hit):
                i = [self.tbeg,other_Hit.tbeg].index( min([self.tbeg,other_Hit.tbeg]) )
                intersection = max(0,[self.tbeg+self.tdur,other_Hit.tbeg+other_Hit.tdur][i]-...
                [self.tbeg,other_Hit.tbeg][1-i])
                return (intersection>0.3*min(self.tdur,other_Hit.tdur) and self.channel_name ==...
                other_Hit.channel_name and self.file_name == other_Hit.file_name)

        def __hash__(self):
                return hash((self.file_name,self.channel_name))

        def combine_hits(self, other_Hit):
                if max(self.scores.values())>= max(other_Hit.scores.values()):
                        combined_hit = copy.deepcopy(self)
                        combined_hit.scores.update(other_Hit.scores)
                else:
                        combined_hit = copy.deepcopy(other_Hit)
                        combined_hit.scores.update(self.scores)
                return combined_hit

        def get_kw_element(self,systems_MTWV,combination_methodology):
                kw= ET.Element("kw")
                score = self.combined_score(systems_MTWV,combination_methodology)
                kw.attrib = {"file": self.file_name, "channel":self.channel_name,...
                "tbeg": str(np.round(self.tbeg,2)), "dur":str(np.round(self.tdur,2)),...
```

```
                "score":str(np.round(score,6)), "decision": "YES"}
                kw.tail = "\n"+ 2*"  "
                return kw

        def combined_score(self, systems_MTWV,combination_methodology):
                combined_score = 0
                sum_of_MTWV = sum([systems_MTWV[system] for system in systems_MTWV])
                max_of_MTWV = max([systems_MTWV[system] for system in systems_MTWV])
                for system_name, system_MTWV in systems_MTWV.items():
                        if system_name in self.scores:
                                if combination_methodology == "CombSUM":
                                        combined_score +=  self.scores[system_name]

                                elif combination_methodology == "CombMNZ":
                                        combined_score+=(len(self.scores)*self.scores[system_name])
                                elif combination_methodology == "WCombMNZ":
                                        combined_score+=len(self.scores)*...
                                        (system_MTWV/sum_of_MTWV)*self.scores[system_name]
                                elif combination_methodology == "Power2":
                                        combined_score +=  2**(system_MTWV - max_of_MTWV)...
                                        * self.scores[system_name]
                return combined_score

class System_Combiner():
        """
        self.merge_XML_lists_of_hits_files takes as argument:
        - a list of (paths to) XML lists of hits that it must merge
        - the path to the new merged XML list
        - a list of the names of the systems associated to each XML list of hits
        - the MTWV of each of these systems (in a dictionary)
        - the methodology with which to merge the scores

        The hits are stored as instances of the class Hit
        For each query, it uses self.merge_list_of_hits to create a single list
        out of all the hits associated to that query (from all systems),
        and merge the hits when necessary using the methods from the Hit class
        Combining options are: "Power2", "CombSUM", "CombMNZ" and "WCombMNZ"
        """
        def __init__(self):
                self.systems = []
                self.systems_MTWV = dict()

        def merge_XML_lists_of_hits_files(self,list_of_paths_to_XML_files, ...
        path_to_output_merged_XML_file, list_of_systems_names,...
        dict_of_systems_MTWV,combination_methodology):
                self.systems = list_of_systems_names
                self.systems_MTWV = dict_of_systems_MTWV
                new_tree = copy.deepcopy(ET.parse(list_of_paths_to_XML_files[0]))
                new_root = new_tree.getroot()
                root_dict = dict()
                for index, system_name in enumerate(self.systems):
                        root_dict[system_name]=ET.parse(list_of_paths_to_XML_files[index]).getroot()
                for detected_kwlist in new_root:
                        for kw in list(detected_kwlist):
                                detected_kwlist.remove(kw)
                        kwid = detected_kwlist.attrib["kwid"]
```

```
                list_of_hits = []
                for system, root in root_dict.items():
                    for query in root:
                        if query.attrib["kwid"] == kwid:
                            for kw in query:
                                info=[kw.attrib["file"],kw.attrib["channel"],....
                                kw.attrib["tbeg"],kw.attrib["dur"]]
                                list_of_hits.append(Hit(info,...
                                {system: float(kw.attrib["score"])}))
                merged_list_of_hits = self.merge_list_of_hits(list_of_hits)
                for hit in merged_list_of_hits:
                    detected_kwlist.append(hit.get_kw_element(self.systems_MTWV,...
                    combination_methodology))
        new_tree.write(path_to_output_merged_XML_file)

    def merge_list_of_hits(self,list_of_hits):
        merged_list = dict()
        for hit in list_of_hits:
                # We use the overloaded __eq__ function of the Hit class
                if hit in merged_list:
                        merged_list[hit] = hit.combine_hits(merged_list[hit])
                else:
                        merged_list[hit] = hit
        return [merged_list[hit] for hit in merged_list]
```

Code 5: Classes 'Hit' and 'System_Combiner'.