MLSALT5: Speech and Language Processing Applications
Practical: Automatic MT Evaluation

FEYNMAN LIANG

1. 1-BEST KEYWORD SPOTTING

1.1. Word based KWS.

1.1.1. *Implementation description.* We implement a word-based KWS system in the `KWSIndex` class (Listing 2). Our KWS system is constructed in the `KWSIndex#apply` method, which takes a CTM file, reads each line, and parses it into a `CTMEntry` (Listing 1). Each `CTMEntry` contains the same information present in a line in a CTM file (e.g. kw file, start time, duration, token). In addition, we add some additional information such as the previous token and end times (if applicable) to each `CTMEntry` during parsing in order to allow us to identify `CTMEntry`s which are consecutive within the 1-best output.

These `CTMEntry`s are then added to an inverted index for quick lookup by token. We represent the inverted index using a hashmap which is keyed on the entry's token. We chose to use a hashmap because it provides insertions, updates, deletions, and lookups all in $O(1)$ amortized time and has amortized space complexity $O(N)$ where N is the number of entries.

At query time, the `KWSIndex.kws` method is called with a path to `queries.xml`. Each query string is passed to `KWSIndex.get`, where it is split on whitespaces to yield a sequence of individual words (i.e. tokens), and each token is looked up in the inverted index. This yields a sequence of collections of `CTMEntry`s, each collection of `CTMEntry`s representing the possible hits for that position in the query string.

We can view this as a lattice where the time axis is the position within the query string and the collection of hits at each position nodes. Since the hits for a query sequence should be consecutive in time, we need to find paths through the lattice such that each node within the path appears consecutively within the CTM file. This is where the `prevToken` and `prevEndTime` fields of the `CTMEntry` class come in. Using this information, we can filter all possible paths through this lattice to just those whose sequences of `CTMEntry`s appear consecutively within the CTM file. During this construction of paths, we also enforce the 0.5 second rule (i.e. the end time of each word in a path must be withn 0.5 seconds of the next word's start time). For each path, we set the score to be equal to the average of all the CTM entry scores in the path because this yielded identical performance to the reference results provided in `/usr/local/teach/MLSALT5/Practical/scoring`.

After finding all valid paths, we are left with a collection of hits for the query sequence. These hits are used to instantiate a `QueryResults` class, whose `QueryResults.toXML` method is used to write the results in the required KWS output format.

1.1.2. *Results for word-based KWS.* We ran our word-based KWS system on `reference.ctm` and `decode.ctm` and ensured that our resuts (Table 2.1) matched the provided results in `/usr/local/teach/MLSALT5/Practical/scoring`.

System	IV	OOV	All
Word (`reference.ctm`)	1.000	1.000	1.000
Word (`decode.ctm`)	0.402	0.000	0.320

TABLE 1. TWV for word-based KWS on 1-best reference

Table 2.1 shows that our KWS system achieves 1.0 TWV on `reference.ctm`. This is expected because we built the KWS index on the same reference we scored against.

When we run the system on the 1-best decoding output of a word-based ASR system (`decode.ctm`), we obtain significantly worse results. These errors are due to errors from the ASR system causing incorrect

Date: March 11, 2016.

hits to be added to the KWS index as well as other entries to be omitted, resulting in worse performance. Also, notice that our out-of-vocabulary performance is 0.000, meaning that we were not able to find any hits for any out of vocabulary words. Since a word based system is keyed on words in its vocabulary, any OOV words will result in a miss. Hence, this 0.000 TWV on OOV words is expected not only for this system, but for any word-based KWS system.

1.2. **Morphological Decomposition.** The motivation for using morph decomposition is to better handle OOV queries. Although a particular query word may not be in the vocabulary, by decomposing both the query as well as the keys in the KWS inverted index into morphs, there is a chance that the decomposed query's morphs will match with some decomposed entry's morphs and produce a hit.

1.2.1. *Implementation details.* The `MorphDecompose` class (Listing 4) takes a morph dictionary for both the CTM entries (`morph.dct`) as well as the queries (`morph.kwslist.dct`) and provides the methods `MorphDecompose.decomposeQuery` and `MorphDecompose.decomposeEntry` for decomposing query sequences and `CTMEntry`s respectively into their corresponding morph sequences. When decomposing `CTMEntry`s, we break up each entry into multiple entries corresponding to the individual morphs. We split the duration of a word equally amongst its corresponding morphs. We chose score of each morph entry to be equal to the score of the original entry so that the score assigned to a hit sequence (see **??**) remains the same before and after morph decomposition. In addition, we update the `prevToken` and `prevEndTime` fields to correctly reflect the decomposition of the previous consecutive entry (if applicable).

The `KWSIndexMorph` class (Listing 5) uses `MorphDecompose` to automatically decompose any queries submitted as well as the the inverted index keys (if required, e.g. if using a word-based rather than morph-based ASR decoding output) into morphs before performing the query. After decomposing the query (and the index inf applicable), `KWSIndexMorph.kws` performs an identical procedure to `KWSIndex.kws` to construct a lattice and find valid paths satisfying the 0.5 seconds rule.

Note that the morph-based system output (`decode-morph.xml`) is no longer a one-best list. For example, on

Hence, the entries in the CTM file are no longer disjoint and it does not make sense to enforce contiguity of the resuts for a query sequence. Hence, in `KWSIndexMorph` we remove the contiguity requirement when generating hit sequences (i.e. paths through the lattice of hits for each query sequence position).

1.2.2. *Results.* We first ran our morph-based KWS system on the output of a word-based ASR system (`decode.ctm`). This required `KWSIndexMorph` to perform morph decomposition of the inverted index as initially after parsing the CTM file each `CTMEntry`'s token was a word rather than a morph. We also tried our system using the output of a morph-based ASR system (`decode-morph.ctm`). In this case, morph decomposition of the inverted index was not required as the tokens for each `CTMEntry` were already morphs.

System	IV	OOV	All
Morph (`decode.ctm`)	0.378	0.016	0.304
Morph (`decode-morph.ctm`)	0.373	0.065	0.310

TABLE 2. Best possible TWVs for word-based vs morph-based systems

Table 2.1 shows our results. Notice that unlike word-based systems, the OOV performance of morph-based systems is no longer zero. This is because queries that are OOV may decompose into morphs which happen to be present in the KWS index, allowing a hit to be found. However, this only allows us to handle OOV words with IV morph decompositions, which represent a limited subset of all possible OOV words. Accordingly, while OOV perfomrance is non-zero it is far from the performance for IV words.

Also, notice that the TWV score for IV words dropped significantly, and that despite the increase in OOV TWV our overall TWV has also dropped. To understand this, recall that TWV is computed as

$$TWV = 1 - (P_{miss} + \beta P_{FA}) \tag{1.1}$$

where $\beta = 999.9$ for Babel. Notice that both the number of true positives as well as the number of false positives affect the TWV score. Since multiple words may share a common morph decomposition and our KWS system no longer requires adjacent entries in a hit sequence appear contiguously in the CTM file, the number of hits has increased from #Sys=725 to #Sys $\in \{1230, 1308\}$ (Table 3), resulting in more true positives (#CorDet=405 increases to be $\in \{420, 415\}$) but also more false alarms (#FA=320

System	#Sys	#CorDet	#FA	#Miss
Word (`decode.ctm`)	725	405	320	558
Morph (`decode.ctm`)	1308	420	888	543
Morph (`decode-morph.ctm`)	1230	415	815	548

TABLE 3. Detailed performance metrics for word vs morph based KWS

increases to be $\in \{888, 815\}$). The increase in number of false alarms P_{FA} offsets any reduction in P_{miss} and hence the overall TWV decreases.

1.3. **Score Normalization.** In the definition of TWV (Equation 1.1), the penalty for a miss depends on the frequency of a word and is more costly for rare words (i.e. less total number of occurences) [WM14]. This suggests that TWV may be improved by boosting the scores of rare words and surpressing the scores of frequent words.

Sum-to-one (STO) normalization is one method of achieving this effect by adjusting the score s_{ki} for the ith hit for keyword k to be

$$\hat{s}_{ki} = \frac{s_{ki}^{\gamma}}{\sum_{j \in \{\text{hits for } k\}} s_{kj}^{\gamma}} \tag{1.2}$$

$\gamma > 0$ is a tunable parameter, which in our experiments was fixed to $\gamma = 1$.

We implement STO normalization in `QueryResult` (Listing 3), where it is applied in the class constructor and used when writing the KWS query results to `xml`.

System	Threshold	IV	OOV	All
Word (`decode.ctm`)	0.167	0.402	0.000	0.320
Morph (`decode.ctm`)	0.205	0.378	0.016	0.304
Morph (`decode-morph.ctm`)	0.301	0.373	0.065	0.310
Word STO (`decode.ctm`)	0.029	0.402	0.000	0.320
Morph STO (`decode.ctm`)	0.056	0.391	0.015	0.314
Morph STO (`decode-morph.ctm`)	0.047	0.389	0.065	0.323

TABLE 4. Best possible TWVs for systems with vs without sum-to-one (STO) score normalization

Table 4 compares the effects of STO normalization on our word-based, morph-based using `decode.ctm`, and morph-based using `decode-morph.ctm` KWS systems. Notice that in all cases the best threshold decreases significantly, consistent with the observation that raw scores tend to overapproximate the true posterior probability and are better adjusted by score normalization [WM14].

In general, we see that STO usually helps improve the over all TWV. For both the morph-based KWS systems, gains of 0.010 and 0.013 in overall TWV were observed for the `decode.ctm` and `decode-morph.ctm` systems respectively. Interestingly, the performance of the word-based system did not change after applying STO.

Table 5 compares more detailed metrics of STO vs unnormalized KWS systems. In general, we see that STO leads to little difference in performance when applied to any of the systems independently.

System	#Sys	#CorDet	#FA	#Miss
Word (`decode.ctm`)	725	405	320	558
Morph (`decode.ctm`)	1308	420	888	543
Morph (`decode-morph.ctm`)	1230	415	815	548
Word STO (`decode.ctm`)	725	405	320	557
Morph STO (`decode.ctm`)	1308	420	888	543
Morph STO (`decode-morph.ctm`)	1229	415	814	548

TABLE 5. Extension of Table 3 with results after STO score normalization

2. System Combination

In this section, we investigate system combinations involving three outputs with unnormalized scores from IBM WFST software (a morph-based system "Morph" and two word-based systems "Word" and Word Sys2).

2.1. **STO normalization on IBM WFST outputs.** We first consider the provided outputs in isolation. Table 6 investigate how STO affects each of the individual systems. Here, the increase in TWV after STO normalization is dramatic, with overall TWV gains ranging from 0.061 to 0.160. One possible explanation for this could be that the IBM WFST software yields scores which vary significantly depending on keywords queried. Hence, the effects of STO normalization very significantly across keywords: common hits are assigned lower scores and the scores of rarer hits are increased.

System	IV	OOV	All
Morph	0.430	0.089	0.360
Word	0.501	0.000	0.399
Word Sys2	0.507	0.000	0.403
Morph STO	0.556	0.367	0.520
Word STO	0.579	0.000	0.460
Word Sys2 STO	0.585	0.000	0.465

Table 6. Performance of individual IBM WFST KWS system outputs

The results in Table 6 show that the Morph STO system is the best we have considered so far, achieving 0.520 overall TWV. It's strong performance is largely due to its strong OOV TWV of 0.367. Notice that just like in and , the word-based system achieves OOV TWV of 0.000 while the morph-based systems are able to achieve non-nil OOV TWV. Again, this is due to the fact that different (e.g. OOV) word sequences may decompose to the same morph sequence (e.g. which could happen to be IV), allowing OOV queries to be handled.

2.2. **Combining IBM WFST outputs and the effects of score normalization.** To combine outputs, we provide the `ResultsCombiner` class (Listing 6), who's `ResultsCombiner#combine` method takes two `QueryResults` (possibly from different KWS systems) and returns the union of the two `QueryResults`.

One design decision was whether to apply STO before combining different outputs or over the combined output. We compare the performance of applying no STO normalization, STO just before combining, STO just after combining, and STO both before and after in Table 7. Our results show that the best performing system performed STO only before combining the outputs, but that the difference in performance is minimal. Applying STO before combining results can be justified by two factors. Firstly, scores from different KWS systems can vary so STO normalization before combination ensures combined scores are on the same scale. Secondly, we do not apply STO after combining because doing so would penalize multiple systems returning duplicate hits, which would increase the sum of the score for that keyword and hence result in all scores being scaled by a smaller factor. Due to these justifications, we apply STO only before output combination for remaining experiments.

STO	IV	OOV	All
None	0.382	0.087	0.322
After Combination	0.488	0.368	0.464
Before Combination	0.491	0.365	0.465
Both Before/After	0.489	0.358	0.462

Table 7. Effect of STO before and/or after combination on combined system Morph+Word+Sys2

The next item we investigated was which system combination could yield the best performance. Accordingly, we investigated all $\binom{3}{2} + \binom{3}{3}$ possible combinations, both with and without STO, in Table 8.

The overall trends in Table 8 show that STO before combination yields an improvement in both IV and OOV TWV across all combinations, but particularly for OOV performance of combinations involving

System	IV	OOV	All
Morph+Word+Sys2	0.382	0.087	0.322
Morph+Word	0.413	0.094	0.347
Morph+Sys2	0.427	0.094	0.359
Word+Sys2	0.447	0.000	0.355
Morph STO+Word STO+Sys2 STO	0.491	0.365	0.465
Morph STO+Word STO	0.530	0.363	0.496
Morph STO+Sys2 STO	0.539	0.362	0.502
Word STO+Sys2 STO	0.544	0.000	0.432

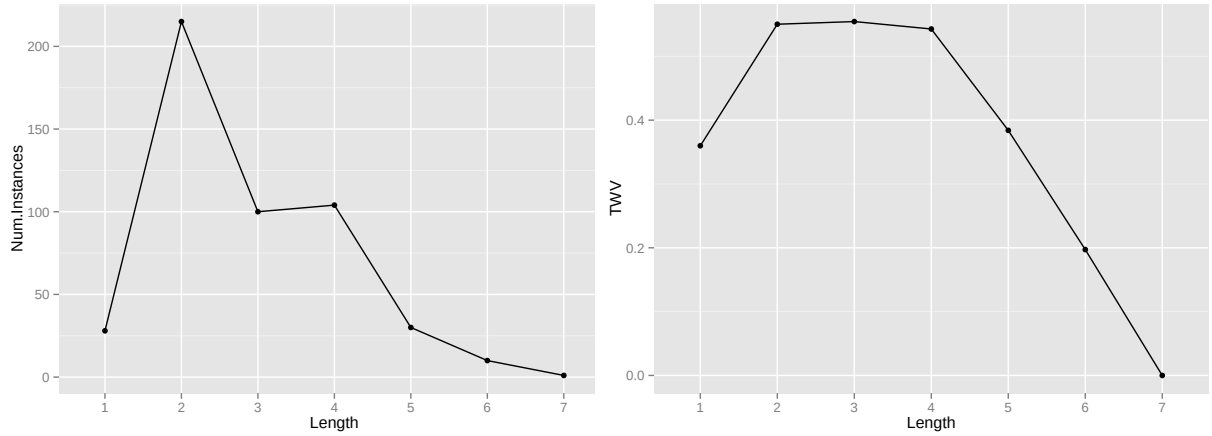
TABLE 8. Performance of combinations of IBM WFST KWS systems

morph-based systems. Combing with Table 6, we see a general trend that combining more sources tends to boost OOV performance while hurting IV performance. Intuitively, this makes sense. Adding additional outputs will increase the number of total hits, which for well-covered IV queries leads to an increase in P_{fa} (degrading IV TWV) where as for poorly-covered OOV queries tends to decrease P_{miss} (improving OOV TWV).

The best results overall were obtained from using just Morph STO without combination. This suggests that the word-based results do not add anything on top of the morph-based results when using the combination techniques we investigated.

2.3. **Impact of query length.** We also consider how the length of a query affects performance by segmenting the output of our best KWS system (IBM WFST Morph with STO). We wrote the `LengthMap` (**??**) to read `queries.xml` and generate a custom `.map` file mapping queries to their lengths. Note that since the KWS system is morph based, we defined the length of a query to be the number of morphs in a query. `scripts/termselect.sh` is then used with this `.map` file to produce TWV scores segmented by query length.

Figure 1a shows the distribution of queries over their lengths and Figure 1b shows the TWV segmented by query length. Notice that the TWV starts low at queries with a single morph. This can be explained by the large number of hits any single morph will generate, yielding a high P_{fa} which causes a low TWV (Equation 1.1). With $2-4$ morph long queries, the morph sequence is unique enough to keep P_{fa} low without increasing P_{miss}, explaining the high TWVs of around 0.55 for these lengths. After the number of morphs in a query exceeds 4, TWV rapidly degrades. This trend can be explained by noting that a longer sequence of morphs has less probability of occuring, hence it is less likely to be covered by our KWS system. This means that P_{miss} is larger for these longer queries, leading to lower TWV.



(A) Number of instances by query length

(B) TWV by query length

FIGURE 1. Performance of IBM Morph STO system for various query lengths (i.e. number of morphs in query)

References

[WM14] Yun Wang and Florian Metze. An in-depth comparison of keyword specific thresholding and sum-to-one score normalization. 2014.

Appendix A. Code Listings

Our code is available at `/home/fl350/MLSALT5-practical/` on the MLSALT cluster. In subsection A.1 we provide listings of our implementations and in subsection A.2 we provide listings of our unit tests.

A.1. **Implementations.**

Listing 1. An entry in KWS index

```scala
package com.feynmanliang.kws

import scala.xml.Elem

case class CTMEntry(
    kwFile: String,
    channel: Int,
    startTime: Double,
    duration: Double,
    token: String,
    prevToken: Option[String],
    prevEndTime: Double, // for testing contiguity during phrase query
    score: Double) extends Ordered[CTMEntry] {
  import scala.math.Ordered.orderingToOrdered

  def compare(that: CTMEntry): Int =
    (this.kwFile, this.startTime) compare (that.kwFile, that.startTime)

  def toXML(): Elem = {
    <kw
      file ={kwFile}
      channel={f"$channel"}
      tbeg={f"$startTime%.2f"}
      dur={f"$duration%.2f"}
      score={f"$score%.6f"}
      decision="YES" />
  }
}

object CTMEntry {
  def fromXMLNode(e: scala.xml.Node): CTMEntry = CTMEntry(
    kwFile=(e \ "@file").text,
    channel=(e \ "@channel").text.toInt,
    startTime=(e \ "@tbeg").text.toDouble,
    duration=(e \ "@dur").text.toDouble,
    token="", // TODO: allow initializing token from XML (OK for just combining)
    prevToken=None, // TODO: allow initializing token from XML (OK for just combining)
    prevEndTime=0D, // TODO: allow initializing token from XML (OK for just combining)
    score=(e \ "@score").text.toDouble)
}

// vim: set ts=2 sw=2 et sts=2:
```

Listing 2. Word-based KWS system

```scala
package com.feynmanliang.kws

import java.io.{BufferedWriter, File, FileWriter}
import scala.io.Source
import scala.xml.Elem

class KWSIndex(
    val index: Map[String, Set[CTMEntry]],
```

```
    val scoreNormalization: String = "NONE") {
  def get(tokens: String): Option[Set[CTMEntry]] = {
    val res = tokens.split("\\s+")
      .map(_.toLowerCase)
      .map(index.get)
    if (res.exists(_.isEmpty)) None
    else Some(res
        .map(_.get.map(entry => entry.copy(score=(1.0/res.size) * entry.score)))
        .reduceLeft { (acc, x) =>
          (for {
            prevEntry <- acc;
            entry <- x;
            if (
              entry.kwFile == prevEntry.kwFile // phrases must belong to same file
              && prevEntry.startTime < entry.startTime && entry.startTime < (prevEntry.startTime + prevEntry.duration)
                    + 0.5
              && prevEntry.startTime + prevEntry.duration == entry.prevEndTime) // TODO: generalize to morphs?
          } yield {
            prevEntry.copy(
              duration = entry.startTime + entry.duration - prevEntry.startTime,
              token = prevEntry.token ++ " " ++ entry.token,
              score = prevEntry.score + entry.score
            )
          }).toSet
      })
  }

  def kws(queryFilePath: String): QueryResult = {
    val queryFile = scala.xml.XML.loadFile(queryFilePath)
    val results = (queryFile \ "kw").map { kw =>
      this.get((kw \ "kwtext").text) match {
        case None => (kw \ "@kwid").text -> Set()
        case Some(hits) => (kw \ "@kwid").text -> hits
      }
    }.toMap.asInstanceOf[Map[String, Set[CTMEntry]]]

    new QueryResult(queryFilePath.split("/").last, results, scoreNormalization)
  }


  def write(ctmPath: String): Unit = {
    def entry2line(entry: CTMEntry): String = {
      s"${entry.kwFile} ${entry.channel} ${entry.startTime} ${entry.duration} " +
      s"${entry.token} ${entry.score}"
    }
    val file = new File(ctmPath)
    val bw = new BufferedWriter(new FileWriter(file))
    index.values.toList.flatten.sorted.foreach { entry =>
      bw.write(entry2line(entry))
    }
    bw.close()
  }
}

object KWSIndex {
  def apply(ctmPath: String, scoreNorm: String = "NONE"): KWSIndex = {
    def line2entry(line: String, prevToken: Option[String], prevEndTime: Double): CTMEntry = {
      val items = line.split("\\s+")
      val startTime = items(2).toDouble
      val entry = CTMEntry(
        kwFile = items(0),
        channel = items(1).toInt,
        startTime = startTime,
        duration = items(3).toDouble,
        token = items(4).toLowerCase,
        prevToken = prevToken,
        prevEndTime = prevEndTime,
        score = items(5).toDouble)
      entry
    }
```

```
78      def line2endTime(line: String): Double = {
79        val items = line.split(" ")
80        items(2).toDouble + items(3).toDouble
81      }
82      val ctmFile = Source.fromFile(ctmPath)
83      val index = ctmFile.getLines().map { line =>
84        line2entry(line, None, 0D)
85      }.foldLeft(List.empty[CTMEntry]) { case (acc, entry) =>
86        acc match {
87          case Nil =>
88            entry.copy(
89              prevToken = None,
90              prevEndTime = 0D) :: acc
91          case prevEntry :: _ =>
92            if (prevEntry.kwFile != entry.kwFile) {
93              entry.copy(
94                prevToken = None,
95                prevEndTime = 0D) :: acc
96            } else {
97              entry.copy(
98                prevToken = if (prevEntry.startTime >= entry.startTime) {
99                  prevEntry.prevToken
100               } else {
101                 Some(prevEntry.token)
102               },
103               prevEndTime = prevEntry.startTime + prevEntry.duration) :: acc
104           }
105       }
106     }.foldLeft(Map.empty[String, Set[CTMEntry]]) { (acc, entry) =>
107       acc + (entry.token -> (acc.getOrElse(entry.token, Set.empty[CTMEntry]) + (entry)))
108     }
109     new KWSIndex(index, scoreNorm)
110   }
111
112   def main(args: Array[String]):Unit = {
113     case class Config(
114       ctmFile: File = new File("."),
115       queryFile: File = new File("."),
116       scoreNorm: String = "NONE",
117       morphDecompose: Boolean = false,
118       out: File = new File("."))
119
120     val parser = new scopt.OptionParser[Config]("KWSIndex") {
121       head("kwsindex")
122       opt[File]('c', "ctmFile") required() valueName("<file>") action { (x, c) =>
123       c.copy(ctmFile = x) } text("ctmFile is a required file property")
124       opt[File]('q', "queryFile") required() valueName("<file>") action { (x, c) =>
125       c.copy(queryFile= x) } text("queryFile is a required file property")
126       opt[String]('q', "scoreNorm") required() valueName("<string>") action { (x, c) =>
127       c.copy(scoreNorm= x) } text("scoreNormis a required string property")
128       opt[File]('o', "out") required() valueName("<file>") action { (x, c) =>
129       c.copy(out = x) } text("out is a required file property")
130     }
131     // parser.parse returns Option[C]
132     parser.parse(args, Config()) match {
133       case Some(config) =>
134         // do stuff
135         val index = KWSIndex(config.ctmFile.getPath(), config.scoreNorm)
136         val queryResults = index.kws(config.queryFile.getPath())
137         scala.xml.XML.save(config.out.getPath(), queryResults.toXML())
138       case None =>
139         // arguments are bad, error message will have been displayed
140         sys.error("Error parsing arguments!")
141     }
142   }
143 }
144
145 // vim: set ts=2 sw=2 et sts=2:
```

LISTING 3. Result of a KWS query

```
package com.feynmanliang.kws

import scala.xml.Elem

case class QueryResult(
    val file: String,
    private[kws] val results: Map[String, Set[CTMEntry]],
    scoreNormalization: String = "NONE") {

  val resultsNorm = scoreNormalization match {
    case "NONE" => results
    case "STO" => results.map { case (k,v) =>
      val sumScore = v.map(_.score).sum
      k -> v.map(entry => entry.copy(
        score = entry.score / sumScore
      ))
    }
    case _ => sys.error(s"Unknown score normalization, got: ${scoreNormalization}")
  }

  def toXML(): Elem = {
    <kwslist
        kwlist_filename="IARPA-babel202b-v1.0d_conv-dev.kwlist.xml"
        language="swahili"
        system_id="">
      {for (kw <- resultsNorm.keys) yield {
        <detected_kwlist kwid={s"$kw"} oov_count="0" search_time="0.0">
          {for {
            entry <- resultsNorm(kw)
            // if (entry.score >= 0.042)
          } yield {
            entry.toXML()
          }}
        </detected_kwlist>
      }}
    </kwslist>
  }

}

object QueryResult {
  def fromXML(e: Elem, scoreNorm: String = "NONE"): QueryResult = {
    QueryResult(
      (e \ "@kwlist_filename").text,
      (e \ "detected_kwlist").map { kwlist =>
        (kwlist \ "@kwid").text -> ((kwlist \ "kw").map { kw =>
          CTMEntry.fromXMLNode(kw)
        }.toSet)
      }.toMap,
      scoreNorm)
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

LISTING 4. Performs morphological decomposition using morph dicts

```
package com.feynmanliang.kws

import scala.io.Source

class MorphDecompose private (
    qDict: Map[String, List[String]], obDict: Map[String, List[String]]) {
  def decomposeQuery(tokens: Iterable[String]): List[List[String]] = {
    tokens.map(decomposeQuery).toList
  }
  def decomposeQuery(token: String): List[String] = {
```

```
    qDict.getOrElse(token, List(token))
  }

  /** Decompose a CTM entry into a list of morphemes
      We distribute startTime/durations and scores uniformly across all morphemes.
   */
  def decomposeEntry(entry: CTMEntry): List[CTMEntry] = {
    obDict.get(entry.token) match {
      case None => List(entry)
      case Some(morphs) => {
        val morphDur = (entry.duration / morphs.size)
        if (morphs.size < 2) {
          List(entry)
        } else {
          morphs.sliding(2).zipWithIndex.flatMap { case (morphPair, i) =>
            if (i == 0) {
              List(
                entry.copy(
                  duration = morphDur,
                  token = morphPair(0),
                  prevToken = entry.prevToken.flatMap(obDict.get).map(_.last)
                ),
                entry.copy(
                  startTime = entry.startTime + (i+1)*morphDur,
                  duration = morphDur,
                  token = morphPair(1),
                  prevToken = Some(morphPair(0)),
                  prevEndTime = entry.startTime + (i+1)*morphDur)
              )
            } else {
              List(
                entry.copy(
                  startTime = entry.startTime + (i+1)*morphDur,
                  duration = morphDur,
                  token = morphPair(1),
                  prevToken = Some(morphPair(0)),
                  prevEndTime = entry.startTime + (i+1)*morphDur)
              )
            }
          }.toList
        }
      }
    }
  }
}

object MorphDecompose {
  private def parseMorphDict(dictPath: String): Map[String, List[String]] = {
    Source.fromFile(dictPath).getLines().map { line =>
      val words = line.split("\\s+")
      words.head -> words.tail.toList
    }.toMap
  }

  def apply(qDictPath: String, obDictPath: String) = {
    val qDict = parseMorphDict(qDictPath)
    val obDict = parseMorphDict(obDictPath)
    new MorphDecompose(qDict, obDict)
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

LISTING 5. Morph-based KWS system which automatically applies morph decomposition to the query as well as to index entries if required

```
package com.feynmanliang.kws

```

```
import java.io.File

class KWSIndexMorph(
    index: Map[String, Set[CTMEntry]],
    md: MorphDecompose,
    scoreNorm: String = "NONE") extends KWSIndex(index, scoreNorm) {

  val morphIndex = if (index.values.reduce(_++_).exists(entry => md.decomposeEntry(entry).size > 1)) {
    // Apply entry decomposition (for decode.ctom)
    println("Decomposing index => morphIndex")
    index.values
      .flatMap(_.flatMap(md.decomposeEntry))
      .foldLeft(Map[String, Set[CTMEntry]]()) { (acc, x) =>
      acc + (x.token -> (acc.getOrElse(x.token, Set[CTMEntry]()) + x))
    }
  } else {
    // No need to decompose (for decode-morph.ctm)
    println("index already built over morphs, setting morphIndex <- index")
    index
  }

  override def get(tokens: String): Option[Set[CTMEntry]] = {
    val hitsPerMorph = md.decomposeQuery(tokens.split("\\s+").map(_.toLowerCase))
      .flatten // treat morphs same as words
      .map(morphIndex.get)
    if (hitsPerMorph.exists(hits => hits.isEmpty)) None
    else Some(
      hitsPerMorph
        .map(hits =>
          hits.get.map { entry =>
            entry.copy(score=(1.0/hitsPerMorph.size) * entry.score)
          }
        )
        .reduceLeft { (acc, x) =>
          (for {
            prevEntry <- acc;
            entry <- x;
            if (
              prevEntry.kwFile == entry.kwFile
              && prevEntry.startTime < entry.startTime && entry.startTime < (prevEntry.startTime + prevEntry.duration)
                  + 0.5)
              //&& (entry.prevToken.isEmpty || (entry.prevToken.get == prevEntry.last.token))
              //&& prevEntry.startTime + prevEntry.duration == entry.prevEndTime
          } yield {
            entry.copy(
              startTime = prevEntry.startTime,
              duration = entry.startTime + entry.duration - prevEntry.startTime,
              token = prevEntry.token ++ " " ++ entry.token,
              prevEndTime = prevEntry.prevEndTime,
              score = prevEntry.score + entry.score
            )
          }).toSet
        })
  }
}

object KWSIndexMorph {
  def apply(
      ctmPath: String, obDictPath: String, qDictPath: String, scoreNorm: String = "NONE"): KWSIndexMorph = {
    val index = KWSIndex(ctmPath)
    val md = MorphDecompose(qDictPath, obDictPath)

    new KWSIndexMorph(index.index, md, scoreNorm)
  }

  def main(args: Array[String]):Unit = {
    case class Config(
      ctmFile: File = new File("."),
      queryFile: File = new File("."),
      scoreNorm: String = "NONE",
```

```
72        dict: File = new File("."),
73        kwDict: File = new File("."),
74        morphDecompose: Boolean = false,
75        out: File = new File("."))
76
77      val parser = new scopt.OptionParser[Config]("KWSIndex") {
78        head("kwsindex")
79        opt[File]('c', "ctmFile") required() valueName("<file>") action { (x, c) =>
80        c.copy(ctmFile = x) } text("ctmFile is a required file property")
81        opt[File]('q', "queryFile") required() valueName("<file>") action { (x, c) =>
82        c.copy(queryFile= x) } text("queryFile is a required file property")
83        opt[File]('d', "dict") required() valueName("<file>") action { (x, c) =>
84        c.copy(dict= x) } text("dict is a required file property")
85        opt[File]('k', "kwDict") required() valueName("<file>") action { (x, c) =>
86        c.copy(kwDict= x) } text("kwDict is a required file property")
87        opt[String]('q', "scoreNorm") required() valueName("<string>") action { (x, c) =>
88        c.copy(scoreNorm= x) } text("scoreNormis a required string property")
89        opt[File]('o', "out") required() valueName("<file>") action { (x, c) =>
90        c.copy(out = x) } text("out is a required file property")
91      }
92      parser.parse(args, Config()) match {
93        case Some(config) =>
94          val indexMorph = KWSIndexMorph(
95            ctmPath = config.ctmFile.getPath(),
96            obDictPath = config.dict.getPath(),
97            qDictPath = config.kwDict.getPath(),
98            scoreNorm = config.scoreNorm)
99          val queryResults = indexMorph.kws(config.queryFile.getPath())
100         scala.xml.XML.save(config.out.getPath(), queryResults.toXML())
101       case None =>
102         // arguments are bad, error message will have been displayed
103         sys.error("Error parsing arguments!")
104     }
105   }
106 }
107
108
109 // vim: set ts=2 sw=2 et sts=2:
```

LISTING 6. Performs system combination by running multiple KWS systems in parallel and unioning the `QueryResults`

```
1  package com.feynmanliang.kws
2
3  import java.io.File
4  import scala.xml.Elem
5
6  object ResultCombiner {
7    def combine(qr1: QueryResult, qr2: QueryResult): QueryResult = qr1.copy(
8      scoreNormalization="NONE", // do not renormalize resultsNorm since it is reused in accumulator
9      results =(qr1.results.keys++qr2.results.keys).map {
10       k => k -> (qr1.resultsNorm.getOrElse(k,Set()) ++ qr2.resultsNorm.getOrElse(k,Set()))
11     }.toMap)
12
13   def main(args: Array[String]):Unit = {
14     case class Config(
15       postingLists: Seq[File] = Seq(),
16       scoreNorms: Seq[String] = Seq(),
17       finalScoreNorm: String = "NONE",
18       out: File = new File("."))
19
20     val parser = new scopt.OptionParser[Config]("ResultCombiner") {
21       head("resultCombiner")
22       opt[Seq[File]]('p', "postingLists") required() valueName("<postingList1>,...") action { (x,c) =>
23       c.copy(postingLists = x) } text("posting lists to combine")
24       opt[Seq[String]]('n', "scoreNorms") required() valueName("<scoreNorm1>,...") action { (x,c) =>
25       c.copy(scoreNorms = x) } text("score normalizations to apply before combining (same order as posting lists)")
26       opt[String]('n', "finalScoreNorm") required() valueName("<finalScoreNorm>") action { (x,c) =>
```

```
      c.copy(finalScoreNorm = x) } text("score normalizations to apply to combined output")
      opt[File]('o', "out") required() valueName("<file>") action { (x, c) =>
      c.copy(out = x) } text("file to output to")
    }
    parser.parse(args, Config()) match {
      case Some(config) =>
        require(config.postingLists.size == config.scoreNorms.size)
        val combinedQr = config.postingLists.zip(config.scoreNorms)
          .map { case (pl, sn) =>
            QueryResult.fromXML(scala.xml.XML.load(pl.getPath()), sn)
          }
          .reduceLeft(ResultCombiner.combine)
        scala.xml.XML.save(
          config.out.getPath(),
          combinedQr.copy(scoreNormalization = config.finalScoreNorm).toXML())
      case None =>
        // arguments are bad, error message will have been displayed
        sys.error("Error parsing arguments!")
    }
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

LISTING 7. Generates a `map` file mapping keyword IDs to their lengths

```
package com.feynmanliang.kws

import java.io.{BufferedWriter, File, FileWriter}

object LengthMap {
  def main(args: Array[String]):Unit = {
    case class Config(
      queryFile: File = new File("."),
      lengthType: String = "WORD",
      out: File = new File("."))

    val parser = new scopt.OptionParser[Config]("LengthMap") {
      head("lengthmap")
      opt[File]('q', "queryFile") required() valueName("<file>") action { (x, c) =>
      c.copy(queryFile= x) } text("queryFile is a required file property")
      opt[String]('l', "lengthType") required() valueName("<WORD|MORPH>") action { (x, c) =>
      c.copy(lengthType= x) } text("lengthType is a required file property")
      opt[File]('o', "out") required() valueName("<file>") action { (x, c) =>
      c.copy(out = x) } text("out is a required file property")
    }
    // parser.parse returns Option[C]
    parser.parse(args, Config()) match {
      case Some(config) =>
        val md = MorphDecompose("lib/dicts/morph.kwslist.dct", "lib/dicts/morph.dct")

        val file = new File(config.out.getPath())
        val bw = new BufferedWriter(new FileWriter(file))
        val queryFile = scala.xml.XML.loadFile(config.queryFile.getPath())
        (queryFile \ "kw").map { kw =>
          val kwId = (kw \ "@kwid").text.drop(6)
          val length = config.lengthType match {
            case "WORD" => (kw \ "kwtext").text.split("\\s+").size
            case "MORPH" => (kw \ "kwtext").text.split("\\s+").flatMap(md.decomposeQuery).size
          }
          s"${length} ${kwId}\n"
        }
        .foreach(bw.write)
        bw.close()

      case None =>
        // arguments are bad, error message will have been displayed
        sys.error("Error parsing arguments!")
```

```
    }
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

A.2. Tests.

```
package com.feynmanliang.kws

import org.scalatest.FlatSpec

class KWSIndexSpec extends FlatSpec {
  val ctmPath = "lib/ctms/reference.ctm"
  val queryFilePath = "lib/kws/queries.xml"
  val index = KWSIndex(ctmPath)
  val queryResults = index.kws(queryFilePath)

  "A KWSIndex" should "contain words known to be in the CTM" in {
    assert(!index.get("halo").isEmpty)
  }

  it should "be case insensitive " in {
    assert(!index.get("rachael").isEmpty)
    assert(!index.get("kisumu").isEmpty)
    assert(index.get("Wenga fans").size === 1)
  }

  it should "only contain phrase queries when words are <0.5 sec apart" in {
    val tokens = "what she has gone"
    assert(index.get("what she has gone").get.size === 1)
    assert(index.get("pat pat pat").get.size === 1)
  }

  it should "perform KWS when given a queryFile" in {
    val ctmPath = "lib/ctms/reference.ctm"
    val index = KWSIndex(ctmPath)

    val queryFilePath = "lib/kws/queries.xml"
    val queryResults = index.kws(queryFilePath)

    assert(queryResults.file === "queries.xml")
    assert(!(queryResults.toXML() \ "detected_kwlist").isEmpty)
  }

  it should "perform the same as provided scoring/decode" in {
    val ctmPath = "lib/ctms/decode.ctm"
    val index = KWSIndex(ctmPath)

    assert(index.get("that's why am").isEmpty)
    assert(index.get("kae tukae kae").isEmpty)
  }
}


// vim: set ts=2 sw=2 et sts=2:
```

```
package com.feynmanliang.kws

import org.scalatest.FlatSpec

class QueryResultSpec extends FlatSpec {
  val ctmPath = "lib/ctms/reference.ctm"
  val queryFilePath = "lib/kws/queries.xml"
  val index = KWSIndex(ctmPath)
```

```
  val queryResults = index.kws(queryFilePath)

  // TODO: "NONE" vs "STO" score normalization tests
  "Multiple QueryResults" should "be able to be read from XML" in {
    val qr = QueryResult.fromXML(scala.xml.XML.loadFile("lib/kws/morph.xml"))
    assert(qr.results.get("KW202-00001").isDefined)
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

```
package com.feynmanliang.kws

import org.scalatest.FlatSpec

class MorphDecomposeSpec extends FlatSpec {
  val qPath = "lib/dicts/morph.kwslist.dct"
  val obPath = "lib/dicts/morph.dct"
  val md = MorphDecompose(qPath, obPath)

 "A MorphDecompose" should "decompose a query string" in {
    assert(md.decomposeQuery("walini") === List("wali","ni"))
    assert(md.decomposeQuery("very vile".split("\\s+")) === List(List("ve", "ry"), List("vi", "le")))
  }

  it should "return the query unchanged if no entry in morph dict" in {
    assert(md.decomposeQuery("abcdefg hijkl".split("\\s+")) === List(List("abcdefg"), List("hijkl")))
  }

  it should "decompose a CTMEntry" in {
    val entry = CTMEntry("", 1, 0.5, 0.4, "vile", None, 0.33, 0.09)
    assert(md.decomposeEntry(entry) == List(
      CTMEntry("", 1, 0.5, 0.2, "vi", None, 0.33, 0.09),
      CTMEntry("", 1, 0.7, 0.2, "le", Some("vi"), 0.7, 0.09)
    ))
  }

  it should "return the CTMEntry unchanged if no entry in morph dict" in {
    val entry = CTMEntry("", 1, 0.5, 0.4, "abcdefg", None, 0.33, 0.09)
    assert(md.decomposeEntry(entry) == List(entry))
  }
}

// vim: set ts=2 sw=2 et sts=2:
```

```
package com.feynmanliang.kws

import org.scalatest.FlatSpec

class KWSIndexMorphSpec extends FlatSpec {
  val ctmPath = "lib/ctms/decode-morph.ctm"
  val queryFilePath = "lib/kws/queries.xml"
  val obPath = "lib/dicts/morph.dct"
  val qPath = "lib/dicts/morph.kwslist.dct"

  val indexMorph = KWSIndexMorph(ctmPath, obPath, qPath)

  "A KWSIndexMorph" should "be queryable" in {
    val queryResults = indexMorph.kws(queryFilePath)
    assert((queryResults.toXML() \ "detected_kwlist").length > 100)
  }

  it should "return the same results as the reference" in {
    assert(indexMorph.get("hangesikia").isEmpty)
  }
}
```

```
22
23
24 // vim: set ts=2 sw=2 et sts=2:
```