Keyword Spotting (Coursework 2) - Report

Candidate number: K5013
Total number of words: 2789

1. Introduction

This Keyword Spotting project is about efficiently detecting a particular word or phrase in a stream of audio. For this project, we will assume that the query is a written text. This practical will be focused on tackling several challenges related to keyword spotting such as search efficiency, OOV queries and threshold selection. Note that system combination was excluded from this report due to a lack of time.

The scope of our work will limit itself to 1-best outputs from speech recognition systems.

As required, the code used is publicly available on GitHub and the link to the repository can be provided on demand.

2. Index

2.1. Preliminary

Given a token (word, morpheme or grapheme), we are interested in quickly finding all its occurrences in all documents available using the CTM file. Therefore, our index will be a 2-depth nested *python* dictionary with the following structure: `index[ctm_metadata.file][ctm_metadata.word]`. Doing so will allow a $\mathcal{O}(1)$ average time complexity when retrieving from the index. The objects stored in our index will be instances of `CTM_metadata` defined as followed:

```
@dataclass(frozen=True)
class CTM_metadata():
    """
    Metadata of a word in a Conversation Time Marked (CTM) file
    Attributes:
    - file = name of the audio file
    - channel = channel of the audio file
    - tbeg = start time of the word in seconds
    - dur = duration of the word in seconds
    - word
    - score = score of the word
    - next_word = word that follows the current word in the same utterance

    Note the next_word is not part of the CTM format, but storing it here
    greatly speeds up the search for hits in index search.
    """
    file: str
    channel: int
    tbeg: float
    dur: float
    word: str
    score: float
    next_word: Optional[str]=None
```

2.2. Initialization

To instantiate our `Index` object, we use the following line:

```
ctm_filepath = "lib/ctms/reference.ctm"  # replace this variable with your own filepath
if needed
index = Index(ctm_filepath=ctm_filepath)
```

We will simply iterate over all tokens from the corpus. If we denote N the number of tokens in the corpus, we will therefore have a time complexity $\mathcal{O}(N)$ when initializing our index.

2.3. `Hit` and `HitSequence`

To proceed in an object-oriented programming (OOP), we will define `Hit` and `HitSequence` objects to model the output of the search algorithm. The main idea is to build `HitSequence` from a list of `Hit` objects where a `Hit` corresponds to a match for 1 query word.

A `Hit` instance is actually created from a `CTM_metadata` object. See the `from_ctm_metadata` method in the following code for reference:

```
class Hit:
    """
    A hit represents a word in a CTM file that matches a query word.

    Note that `Hit` share the same attributes as `CTM_metadata`.
    This class was created to clearly separate the two concepts and to supercharge
    the class with useful methods.
    """
    def __init__(self,
                 file: str,
                 channel: int,
                 tbeg: float,
                 dur: float,
                 word: str,
                 score: float,
                 next_word_in_ctm: Optional[str]=None):
        self.file = file
        self.channel = channel
        self.tbeg = tbeg
        self.dur = dur
        self.word = word
        self.score = score
        self.next_word_in_ctm = next_word_in_ctm  # this attribute doesn't caracterize
a hit, but is useful for the search algorithm


    @classmethod
    def from_ctm_metadata(cls, ctm_metadata: CTM_metadata):
        """Create a Hit from a CTM_metadata object"""
        return cls(file=ctm_metadata.file,
                   channel=ctm_metadata.channel,
                   tbeg=ctm_metadata.tbeg,
                   dur=ctm_metadata.dur,
                   word=ctm_metadata.word,
```

```python
                    score=ctm_metadata.score,
                    next_word_in_ctm=ctm_metadata.next_word)


    def __str__(self):
        # Note: next_word_in_ctm is not printed because it doesn't caracterize a hit
        return f'<kw file="{self.file}" channel="{self.channel}" tbeg="{self.tbeg:.2f}"
dur="{self.dur:.2f}" score="{self.score:.6f}" decision="YES"/>\n'


    def __repr__(self):
        # Note: next_word_in_ctm is not printed because it doesn't caracterize a hit
        return (f"Hit(file={self.file}, channel={self.channel}, "
                f"tbeg={self.tbeg}, dur={self.dur}, word={self.word}, "
                f"score={self.score}), next_word_in_ctm={self.next_word_in_ctm})")


    def overlaps_with(self, hit_2: Hit) -> bool:
        start1, start2 = self.tbeg, hit_2.tbeg
        end1, end2 = start1 + self.dur, start2 + hit_2.dur
        return (start1 <= end2) and (start2 <= end1)
```

Appending a `Hit` to a `HitSequence` should:

- Only be possible if the 2 hits come from the same file
- The 2 hits are separated by at most 0.5 seconds.

Doing so should also automatically update the class attributes accordingly. To this respect, we will use the following methods to aggregate the values from the metadata:

Metadata field	Aggregation function
Start time	Minimum
Duration	Sum
Words	List concatenation
Score	Product

Table 1: Aggregation methods to merge several *Hit* objects into a *HitSequence* instance

The aggregation detailed in Table 1 are implemented in Python with the `_aggregate_hits_init` from the `HitSequence` class:

```python
class HitSequence:
    """
```

```
    A sequence of hits that can be aggregated.

    This class is used to aggregate hits that are close to each other
    and that can be aggregated.
    """
    def __init__(self, hits: List[Hit]):
        self.hits = hits
        self._aggregate_hits_init()


    def _aggregate_hits_init(self) -> None:
        assert self.hits, "self.hits is empty"
        self.file = self.hits[0].file
        self.channel = self.hits[0].channel

        def gen_from_list_hits():
            for hit in self.hits:
                yield hit.channel, hit.tbeg, hit.dur, hit.word, hit.score

        df = pd.DataFrame(gen_from_list_hits(), columns=["channel", "tbeg", "dur",
"word", "score"])
        df_agg = df.agg({"tbeg": "min", "dur": "sum", "score": "prod"})

        self.tbeg = df_agg["tbeg"]
        self.dur = df_agg["dur"]
        self.words = df["word"].str.cat(sep=" ").split()
        self.score = df_agg["score"]


    def __str__(self):
        return f'<kw file="{self.file}" channel="{self.channel}" tbeg="{self.tbeg:.2f}"
dur="{self.dur:.2f}" score="{self.score:.6f}" decision="YES"/>\n'


    def __repr__(self) -> str:
        return f"HitSequence(len={len(self.hits)}, file={self.file}, channel=
{self.channel}, tbeg={self.tbeg}, dur={self.dur}, words={self.words}, score=
{self.score})"


    def __len__(self) -> int:
        return len(self.hits)


    def __getitem__(self, index: int) -> Hit:
        return self.hits[index]


    def __contains__(self, hit: Hit) -> bool:
```

```python
        return hit in self.hits


    def __iter__(self) -> Iterator[Hit]:
        return iter(self.hits)


    def __next__(self) -> Hit:
        return next(iter(self.hits))


    def append(self, hit: Hit) -> None:
        assert hit.file == self.file, "Cannot append a hit from another file"
        assert hit.channel == self.channel, "Cannot append a hit from another channel"
        # assert hit.tbeg >= self.tbeg + self.dur, "Cannot append a hit that overlaps with the current sequence"  # TODO: Allow overlaps?
        # assert hit.tbeg <= self.tbeg + self.dur + MAX_SECONDS_INTERVAL, "Cannot append a hit that is too far from the current sequence"  # TODO: Check assert

        self.hits.append(hit)
        self.dur += hit.dur
        self.words += [hit.word]
        self.score *= hit.score


    def __add__(self, other: HitSequence) -> HitSequence:
        assert self.file == other.file, "Cannot add a hit sequence from another file"
        assert self.channel == other.channel, "Cannot add a hit sequence from another channel"
        assert self.tbeg + self.dur + MAX_SECONDS_INTERVAL >= other.tbeg, "Cannot add a hit sequence that is too far from the current sequence"
        assert self.tbeg + self.dur <= other.tbeg + other.dur + MAX_SECONDS_INTERVAL, "Cannot add a hit sequence that is too far from the current sequence"

        hits = self.hits + other.hits
        return HitSequence(hits)


    def is_valid(self) -> bool:
        for hit_1, hit_2 in zip(self.hits, self.hits[1:]):
            if hit_2.tbeg - hit_1.tbeg <= MAX_SECONDS_INTERVAL:
                return False
        return True


    def copy(self) -> HitSequence:
        return copy.deepcopy(self)
```

Note that we made the extra assumption that there is only 1 channel at hand in the data used in this coursework.

2.4. Search

For each query, we will find all occurrences of the first word of the query and store them to a stack data structure as a `HitSequence` instance.

Then, we will keep iterating over the `HitSequence` instances of the stack until we add the next valid word in `HitSequence` then end up with a sequence with the same length as the query. If that is the case, we will then output the completed `HitSequence`. On the contrary - if we cannot find the following word in the CTM file and if it is not valid - we will discard the incomplete `HitSequence` and move on the following one.

The following pseudo-code (translated into Python code in the `_search` function as part of `kws/index.py`) implements this idea:

> 1. Initialize the `index` hashmap
> 2. Initialize an empty list `answers`
> 3. For each query `q`:
>   1. Initialize a Stack priority queue named `stack`
>   2. For each file stored in `index`'s keys:
>   1. If the 1st word of `q` is in the set stored in `index` for the current file, append the current word with its metadata (including the following word in the query as extra data) as a `HitSequence` instance in `stack`
>   2. Else continue
>   3. While `stack` is non-empty:
>   1. Pop the latest `HitSequence` from `stack` and store as `hitseq`
>   2. If `hitseq` is complete with respect to `q`, append `hitseq` to `answers`
>   3. Else:
>   1. If the following word is also the next word of the query and if it is valid timestamp-wise (in terms of start time and duration), add it to `hitseq` and append `hitseq` to `stack`
>   2. Else continue (equivalent to discarding `hitseq`)

Algorithm 1: Vanilla search for keyword spotting

Algorithm 1 is implemented in Python in the `_search` method in the `Index` class defined in the following snippet of code:

```
class Index:
```

```
    def __init__(self, ctm_filepath: str) -> None:
        self.ctm_filepath = Path(ctm_filepath)
        assert self.ctm_filepath.is_file(), f"CTM file not found: {ctm_filepath}"

        self.index = self._build_index()


    def _build_index(self) -> DefaultDict[str, DefaultDict[str, List[CTM_metadata]]]:
        index = defaultdict(lambda: defaultdict(list))

        with self.ctm_filepath.open("r") as f:
            lines = f.readlines()
            for ctm_line, next_ctm_line in zip(lines, lines[1:]):
                next_ctm_metadata = decode_ctm_line(next_ctm_line)
                ctm_metadata = decode_ctm_line(ctm_line,
next_word=next_ctm_metadata.word)
                index[ctm_metadata.file][ctm_metadata.word].append(ctm_metadata)

        return index


    def _search(self, query: Query) -> List[HitSequence]:
        """
        Search for a query in the index.
        """

        list_hitseqs: List[HitSequence] = []
        stack: Deque[HitSequence] = deque()

        # Initialize stack with first word:
        first_word = query.kwtext[0]
        for file in self.index.keys():
            if first_word in self.index[file]:
                for first_word_metadata in self.index[file][first_word]:

 stack.append(HitSequence([Hit.from_ctm_metadata(first_word_metadata)]))

        while stack:
            hitseq = stack.pop()
            next_idx = len(hitseq)

            # If we have a hit sequence:
            if next_idx >= len(query.kwtext):
                list_hitseqs.append(hitseq)
                continue

            # Otherwise, we continue to build the current hit sequence:
            w1_hit = hitseq[-1]
            current_file = w1_hit.file
```

```
            w2 = query.kwtext[next_idx]

            if w2 == w1_hit.next_word_in_ctm:  # Note: This condition implies (w2 in
self.index[current_file]) but the reciprocal is not necessarily true.
                for w2_metadata in self.index[current_file][w2]:
                    w2_hit = Hit.from_ctm_metadata(w2_metadata)
                    if w2_hit.tbeg >= w1_hit.tbeg and w2_hit.tbeg - (w1_hit.tbeg +
w1_hit.dur) <= MAX_SECONDS_INTERVAL:  # allow overlap
                        hitseq_ = hitseq.copy()
                        hitseq_.append(w2_hit)
                        stack.append(hitseq_)
            else:
                pass

        return list_hitseqs


    def _search_gc(self,
                   query: Query,
                   grapheme_confusion: GraphemeConfusion) -> List[HitSequence]:
      # ... cf section 5 on graphemic system for the code


    def search(self,
               query: Query,
               normalize_scores: bool=False,
               gamma: float=1.0,
               grapheme_confusion: Optional[GraphemeConfusion]=None) ->
List[HitSequence]:

        if grapheme_confusion is None:
            list_hitseqs = self._search(query)
        else:
            list_hitseqs = self._search_gc(query,
grapheme_confusion=grapheme_confusion)

        if normalize_scores:
            normalize_scores_hitseqs(list_hitseqs=list_hitseqs, gamma=gamma)

        return list_hitseqs
```

For one query q, assume:

- q has a length of N_q
- The first word in q appears M times

In the worst case scenario, all first words from our utterance match our query q. We would then have a time-complexity of $\mathcal{O}(N_q M)$. In practice, we would achieve a much better complexity as our algorithm discards all hit sequences that do not satisfy the constraints at hand.

The main script in `search.py` iterates over all queries (identified by their `kwid`) and returns all the hits in a dictionary with the following structure:

```
kwid_to_hitseqs = {kwid: [hit_sequence_1, hit_sequence_2, ...]}
```

Content of `search.py`

```
def main(queries_filepath: str,
         ctm_filepath: str,
         output_filename: str,
         normalize_scores: bool=False,
         gamma: float=1.0,
         use_grapheme_confusion: bool=False):
    """
    Search for queries in CTM file and write output to file.
    """
    queries = Queries.from_file(queries_filepath)
    index = Index(ctm_filepath=ctm_filepath)

    kwid_to_hitseqs: DefaultDict[str, List[HitSequence]] = defaultdict(list)

    # If necessary, load grapheme confusion:
    if use_grapheme_confusion:
        grapheme_confusion = GraphemeConfusion(
            grapheme_confusion_filepath=str(DEFAULT_GRAPHEME_CONFUSION_FILEPATH),
            ctm_filepath=ctm_filepath)
    else:
        grapheme_confusion = None


    # Perform search for each query:
    tbar = tqdm(queries.kwid_to_list_queries.items())
    for kwid, list_queries in tbar:
        tbar.set_description(f"Searching for {kwid}")
        for query in list_queries:
            list_hitseqs = index.search(query,
                                        normalize_scores=normalize_scores,
                                        gamma=gamma,
                                        grapheme_confusion=grapheme_confusion)
            kwid_to_hitseqs[kwid].extend(list_hitseqs)
```

```
    output = format_all_queries(kwid_to_hitseqs)

    output_filepath = OUTPUT_DIR / output_filename
    with output_filepath.open("w") as output_file:
        output_file.write(output)

    print(f"Output succesfully written to {output_filepath}")

    return


if __name__ == "__main__":
    typer.run(main)
```

2.5. Implementation validation

First, we had to run the following script:

```
make score-reference
make eval-reference

make score-decode
make eval-decode
```

Note that the `make` command runs pre-written scripts stored in the `Makefile` file which is provided in the Appendix. Basically, it runs the `search.py` and the `scripts/score.sh` scripts.

We then did the same but for the `decode.ctm` file. We eventually ended up with the following results:

System	TWV All	TWV In-vocabulary (TWV IV)	TWV Out-of-vocabulary (TWV OOV)	PFA (probability of false alarm)	PM (probability of missing)	Dec. Tresh
`reference.ctm`	1.000	1.000	1.000	0.00000	0.000	1.0000
`decode.ctm`	0.319	0.401	0.000	0.00002	0.663	0.0425

Table 2: Keyword spotting results using the base configuration

To guarantee that all hit sequences have a score of 1 for `reference.ctm`, we designed a Python test which we validated with success:

```
@pytest.fixture
def queries() -> Queries:
```

```
    queries = Queries.from_file(queries_filepath=DEFAULT_TEST_QUERIES_FILEPATH)
    return queries


@pytest.fixture
def index() -> Index:
    index = Index(ctm_filepath=DEFAULT_TEST_CTM_FILEPATH)
    return index


def test_e2e_index_search_from_reference_all_1(queries: Queries, index: Index):
    for kwid, list_queries in tqdm(queries.kwid_to_list_queries.items()):
        for query in list_queries:
            list_hitseqs = index.search(query)
            assert all(hitseq.score == 1. for hitseq in list_hitseqs), f"Query score is
not 1: {query.kwtext}"
```

As a reminder, the Term-Weighted Value TWV is defined as:

$$\mathrm{TWV}(\tilde{\boldsymbol{w}}, \theta) = 1 - \left[P_{\mathrm{Miss}}\left(\tilde{\boldsymbol{w}}, \theta\right) + \beta P_{FA}(\tilde{\boldsymbol{w}}, \theta)\right] \tag{2}$$

with $\beta = 999.9$ here. Consequently, we are penalizing the False Alarms (FA) much more than missing a query.

Observations:

- As expected, we can see that the score of the reference output is perfect.
- Also as expected, the TWV OOV for `decode.ctm` is null as the set of OOV is by definition made of all words not in the latter file.

2.6. Further work for lattice-based KWS

Our implementation of the 1-best keyword spotting is not suitable for a lattice-based keyword spotting task. The principle reason comes from the structure of the CTM file for a lattice: we should have an extra attribute that gives the IDs of the following words in the lattice.

If we have this correct CTM file structure, then adapting our algorithm is straightforward: for each element popped from the stack in `_search`, we would iterate over all the following nodes of the lattice instead of just looking at the following word in the 1-best case.

3. Morphological decomposition

With morphological decomposition, we are breaking each word from the CTM into smaller pieces that will be treated as new words. As we don't know the actual length proportion of each subword, we will use the sampling assumption that all subwords have the same duration. Hence they are all equal to $\mathrm{Duration}_{original}/N$. We will also assume that subwords have the same updated scores. As scores are multiplied together, we necessarily have $score_{new}^{(1)} = \ldots = score_{new}^{(N)} = (score_{old})^{\frac{1}{N}}$ where $score_{new}^{(i)}$ is the score of the i-th sub-word. To illustrate this process, the following line in `decode.ctm`:

```
BABEL_OP2_202_10524_20131009_200043_inLine 1 3.81 0.25       halo 0.974210
```

Will be turned into:

```
BABEL_OP2_202_10524_20131009_200043_inLine 1 3.81 0.12 ha 0.987021
BABEL_OP2_202_10524_20131009_200043_inLine 1 3.94 0.12 lo 0.987021
```

We are also doing the same for each query as each word of each query is split up in the same fashion. Doing so should allow for more subword combination, thus it will be more likely to correctly match a query. For instance, the following query in `queries.xml`:

```
<kw kwid="KW202-00003">
    <kwtext>nimwachie</kwtext>
</kw>
```

will be turned into:

```
<kw kwid="KW202-00003">
    <kwtext>nim wa chie</kwtext>
</kw>
```

To summarize, a 1-word query will simply be turned into a sentence query and a CTM file will simply have more lines in it. The Python scripts used for this purpose are provided below:

`ctm_to_morph.py`:

```
from pathlib import Path
from typing import Dict, List

from kws.index import decode_ctm_line
from kws.kws_metadata import CTM_metadata
from kws.morph_decomposition.utils import load_morph_dict
```

```
def apply_morph_to_ctm_metadata(ctm_metadata: CTM_metadata,
                                word_to_morphs: Dict[str, List[str]]) ->
List[CTM_metadata]:
    # ------------ EDGE CASE ------------
    if ctm_metadata.word not in word_to_morphs:
        return [ctm_metadata]


    # ------------ MAIN ------------
    morphs = word_to_morphs[ctm_metadata.word]
    list_new_ctm_metadata = []

    # Common metadata for all morphs:
    morph_dur = ctm_metadata.dur / len(morphs)
    morph_score = ctm_metadata.score ** (1 / len(morphs))

    # Iterate over morphs:
    for idx, morph in enumerate(morphs):
        list_new_ctm_metadata.append(CTM_metadata(file=ctm_metadata.file,
                                                  channel=ctm_metadata.channel,
                                                  tbeg=ctm_metadata.tbeg+
(idx*morph_dur),
                                                  dur=morph_dur,
                                                  word=morph,
                                                  score=morph_score)
                                    )

    return list_new_ctm_metadata


def apply_morph_to_ctm_file(ctm_filepath: str,
                            morph_filepath: str,
                            output_filepath: str) -> None:
    assert Path(ctm_filepath).is_file(), f"CTM file not found: {ctm_filepath}"

    word_to_morphs = load_morph_dict(morph_filepath)

    list_new_ctm_metadata: List[CTM_metadata] = []

    with open(ctm_filepath, "r") as f:
        for ctm_line in f.readlines():
            ctm_metadata = decode_ctm_line(ctm_line)
            list_new_ctm_metadata.extend(
                apply_morph_to_ctm_metadata(ctm_metadata,
word_to_morphs=word_to_morphs))

    # Format new CTM file:
    new_ctm = ""
    for ctm_metadata in list_new_ctm_metadata:
```

```
        new_line = f"{ctm_metadata.file} {ctm_metadata.channel} {ctm_metadata.tbeg:.2f}
{ctm_metadata.dur:.2f} {ctm_metadata.word} {ctm_metadata.score:.6f}\n"
        new_ctm += new_line

    # Save new CTM file:
    Path(output_filepath).parent.mkdir(parents=True, exist_ok=True)
    with open(output_filepath, "w") as f:
        f.write(new_ctm)

    print(f"New CTM file saved to: {output_filepath}")
    return
```

`query_to_morph.py`:

```
from typing import Dict, List
from pathlib import Path

from kws.morph_decomposition.utils import load_morph_dict
from kws.query import Query, Queries


def apply_morph_to_query(query: Query,
                         word_to_morphs: Dict[str, List[str]]) -> Query:
    list_queries: List[Query] = []

    for word in query.kwtext:
        if word not in word_to_morphs:
            list_queries.append(Query(kwid=query.kwid, kwtext=word))
        else:
            for morph in word_to_morphs[word]:
                list_queries.append(Query(kwid=query.kwid, kwtext=morph))

    new_query = Query(kwid=query.kwid, kwtext=" ".join([" ".join(q.kwtext) for q in
list_queries]))
    return new_query


def apply_morph_to_queries_file(queries_filepath: str,
                                morph_filepath: str,
                                output_filepath: str) -> None:
    assert Path(queries_filepath).is_file(), f"Query file not found:
{queries_filepath}"
    queries = Queries.from_file(queries_filepath)

    word_to_morphs = load_morph_dict(morph_filepath)
```

```
    list_new_queries: List[Query] = []

    for kwid, list_queries in queries.kwid_to_list_queries.items():
        for query in list_queries:
            list_new_queries.append(apply_morph_to_query(query,
word_to_morphs=word_to_morphs))

    # Create new Queries object:
    new_queries = Queries.from_list_of_queries(list_new_queries)

    # Save new Queries file:
    Path(output_filepath).parent.mkdir(parents=True, exist_ok=True)
    with open(output_filepath, "w") as f:
        f.write(new_queries.to_xml())

    print(f"New Queries file saved to: {output_filepath}")
    return
```

To use our developed scripts `apply_morph_to_ctm.py` and `apply_morph_to_queries.py`, we simply run the following *make* commands:

```
make apply_morph_to_ctm
make apply_morph_to_queries
```

We now have the following files:

- `decode-morph.xml` → decoding output from a morph-based system (already provided)
- `decode-morph-custom.xml` → decoding output from a non-morph system that we transformed using the `morph.dct` mapping
- `queries-morph.xml` → queries resulting from the mapping `morph.kwslist.dct` applied on `queries.xml`

To reproduce our experiments, please run the following:

```
make score-decode-morph
make eval-decode-morph

make score-decode-morph-custom
make eval-decode-morph-custom
```

For the sake of comparison, the following table will present results from the previous section as well as those obtained via morphological decomposition.

We then did the same but for the `decode.ctm` file. We eventually ended up with the following results:

#	System	TWV All	TWV In-vocabulary (TWV IV)	TWV Out-of-vocabulary (TWV OOV)	PFA (probability of false alarm)	PM (probability of missing)	Dec. Tresh
1	`decode.ctm` (base)	**0.319**	**0.401**	0.000	**0.00002**	0.663	0.0425
2	`decode-morph.ctm` (morph-based system)	0.317	0.381	**0.068**	0.00003	**0.651**	0.0373
3	`decode-morph-custom.ctm` (base+mapping)	0.311	0.387	0.018	0.00004	0.652	0.0370

Table 3: Keyword spotting results comparing index search with base configuration and morphological decomposition. Best result for each metric is highlighted.

Observations:

- Both TWV All and TWV IV have slightly decreased while the TWV OOV has slightly increased. In particular, it is now non-null as it is now possible to output out-of-vocabulary sequence of words.
- There is a tradeoff between missing less queries (lower PM) and having more false alarms (higher PFA).
- We expected that the morph-based system (2nd row) would perform better than the base+mapping one (3rd row) as the latter is based on simplifying assumptions (see previous paragraphs).

4. Score normalization

We now want to perform score normalisation at a *query term level* prior to performing scoring.

As a reminder, we have to implement the Sum-To-One (STO) normalisation

$$\hat{S}_{ki} = \frac{S_{ki}^{\gamma}}{\sum_j S_{kj}^{\gamma}} \tag{3}$$

where S_{ki} is the raw score for the i-th hit for query k, the summation over all hits for query k, and γ is a hyperparameter set to 1 for our study.

Therefore, we implemented the following function which will be used to normalize on the result of each query:

```
def normalize_scores_hitseqs(list_hitseqs: List[HitSequence], gamma: float=1.0) ->
None:
    """Normalize the scores of a list of hit sequences according to the Sum-To-One
(STO)
    normalisation. In-place operation."""
    total_score = sum([hitseq.score ** gamma for hitseq in list_hitseqs])
    for hitseq in list_hitseqs:
        hitseq.score = hitseq.score ** gamma / total_score
    return
```

To run the associated experiments, run the following:

```
make score-decode-normalized
make eval-decode-normalized
```

We then obtained the following results:

#	System	TWV All	TWV In-vocabulary (TWV IV)	TWV Out-of-vocabulary (TWV OOV)	PFA (probability of false alarm)	PM (probability of missing)	Dec. Tresh
1	`decode.ctm` (base)	0.319	0.401	0.000	**0.00002**	0.663	0.0425
2	`decode-morph.ctm` (morph-based system)	0.317	0.381	**0.068**	0.00003	**0.651**	0.0373
3	`decode-morph-custom.ctm` (base+mapping)	0.311	0.387	0.018	0.00004	0.652	0.0370
4	`decode.ctm` (base + score normalization)	0.320	**0.402**	0.000	**0.00002**	0.663	0.0074
5	`decode-morph.ctm` (morph-based system + score normalization)	**0.325**	0.391	**0.068**	0.00003	**0.651**	0.0040
6	`decode-morph-custom.ctm` (base+mapping + score normalization)	0.316	0.393	0.018	0.00004	0.662	0.0040

Table 4: Keyword spotting results comparing index search with base configuration, morphological decomposition and score normalization. Best result for each metric is highlighted.

Observations:

- Based on our experiments, score normalization always improve the TWV All and the TWV In scores.
- Score normalization added on system 3 and resulting in system 6 increased the probability of missing from 0.652 to 0.662.

5. Graphemic system

Instead of using morphological decomposition that is based on phonemes, we will use a grapheme confusion approach based on graphemes. The idea is to handle each OOV word by replacing it with the closest IV word with respect to the weighted Levenshtein distance. The weights used should be a function of the frequencies of all grapheme confusions which are provided in the `lib/kws/grapheme.map` file.

If we make the extra assumption that the probability of confusion for one grapheme is independent from the other graphemes in that same word, we can then use the given frequencies as approximations for the probabilities of confusion.

We decided to choose the Levenshtein weights such that the Levenshtein distance is in $[0; 1]$ segment. Thus, we can simply create an associated "posterior probability" score defined as $1 - d_{Levenshtein}(w_1, w_2)$ with w_1 and w_2 the 2 words of interest.

The following pseudo-code (translated into Python code in the `_search_gc` function as part of `kws/index.py`) implements this idea:

> 1. Load the grapheme confusion matrix and the set of in-vocabulary (IV) words
> 2. Initialize the `index` hashmap
> 3. Initialize an empty list `answers`
> 4. For each query `q`:
>   1. Initialize a Stack priority queue named `stack`
>   2. For each file stored in `index`'s keys:
>   1. If the 1st word of `q` is in the set stored in `index` for the current file, append the current word with its metadata (including the following word in the query as extra data) as a `HitSequence` instance in `stack`
>   2. Else if the closest IV word with respect to the weighted Levenshtein distance for the 1st word of `q` is stored in `index` for the current file, append the closest IV word in the same fashion of the previous "if" statement
>   3. Else continue
>   3. While `stack` is non-empty:
>   1. Pop the latest `HitSequence` from `stack` and store as `hitseq`
>   2. If `hitseq` is complete with respect to `q`, append `hitseq` to `answers`
>   3. Else:
>   1. If the following word is also the next word of the query and if it is valid timestamp-wise (in terms of start time and duration), add it to `hitseq` and append `hitseq` to `stack`

> 2. Else if the closest IV word with respect to the weighted Levenshtein distance for the next word is also the next word of the query and if it is valid timestamp-wise (in terms of start time and duration), add it to `hitseq` and append `hitseq` to `stack`
> 3. Else continue (equivalent to discarding `hitseq`)

Algorithm 2: Search with grapheme confusion for keyword spotting

The previous algorithm is implemented in the `_search_gc` method of the `Index` class. Please refer to the [Appendix]() for the full implementation of the `GraphemeConfusion` utility class used in the following script:

```
    def _search_gc(self,
                   query: Query,
                   grapheme_confusion: GraphemeConfusion) -> List[HitSequence]:
        """
        Search for a query in the index with grapheme confusion.
        """

        list_hitseqs: List[HitSequence]= []
        stack: Deque[HitSequence] = deque()

        # Wrap grapheme_confusion.get_closest_iv_word to implement caching in order
speed up search:
        cache: Dict[str, Optional[str]] = {}
        def get_closest_iv_word(oov_word: str) -> Optional[str]:
            if oov_word not in cache:
                cache[oov_word] =
grapheme_confusion.get_closest_iv_word(oov_word=oov_word)
            return cache[oov_word]


        # Initialize stack with first word:
        first_word = query.kwtext[0]

        for file in self.index.keys():
            if first_word in self.index[file]:
                for first_word_metadata in self.index[file][first_word]:

 stack.append(HitSequence([Hit.from_ctm_metadata(first_word_metadata)]))

            else:  # If first word is OOV, we try to find it with grapheme confusion:
                closest_iv_word = get_closest_iv_word(oov_word=first_word)

                if closest_iv_word is None:
                    continue  # If we cannot find a closest IV word, we skip this file.

                if closest_iv_word in self.index[file]:
```

```
                    posterior = grapheme_confusion._similarity_score(first_word,
closest_iv_word)
                    for first_word_metadata in self.index[file][closest_iv_word]:
                        first_word_metadata_posterior =
_get_posterior_metadata(first_word_metadata, posterior)

 stack.append(HitSequence([Hit.from_ctm_metadata(first_word_metadata_posterior)]))



        while stack:
            hitseq = stack.pop()
            next_idx = len(hitseq)

            # If we have a hit sequence:
            if next_idx >= len(query.kwtext):
                list_hitseqs.append(hitseq)
                continue

            # Otherwise, we continue to build the current hit sequence:
            w1_hit = hitseq[-1]
            current_file = w1_hit.file
            w2 = query.kwtext[next_idx]

            if w2 == w1_hit.next_word_in_ctm:  # Note: This condition implies (w2 in
self.index[current_file]) but the reciprocal is not necessarily true.
                for w2_metadata in self.index[current_file][w2]:
                    w2_hit = Hit.from_ctm_metadata(w2_metadata)
                    if w2_hit.tbeg >= w1_hit.tbeg and w2_hit.tbeg - (w1_hit.tbeg +
w1_hit.dur) <= MAX_SECONDS_INTERVAL:  # allow overlap
                        hitseq_ = hitseq.copy()
                        hitseq_.append(w2_hit)
                        stack.append(hitseq_)

            else:
                # If w2 is OOV, we try to find it with grapheme confusion:
                closest_iv_word = get_closest_iv_word(oov_word=w2)

                if closest_iv_word is None or closest_iv_word !=
w1_hit.next_word_in_ctm:
                    continue  # If we cannot find a closest IV word OR if the closest
IV word is not the next word in the query at hand, skip.

                else:  # Note: Necessarily we have closest_iv_word in
self.index[current_file]
                    posterior = grapheme_confusion._similarity_score(w2,
closest_iv_word)
                    for w2_metadata in self.index[current_file][closest_iv_word]:
                        w2_metadata_posterior = _get_posterior_metadata(w2_metadata,
posterior=posterior)
```

```
                        w2_hit = Hit.from_ctm_metadata(w2_metadata_posterior)
                        if w2_hit.tbeg >= w1_hit.tbeg and w2_hit.tbeg - (w1_hit.tbeg +
w1_hit.dur) <= MAX_SECONDS_INTERVAL:  # allow overlap
                            hitseq_ = hitseq.copy()
                            hitseq_.append(w2_hit)
                            stack.append(hitseq_)

        return list_hitseqs
```

Run the following bash code to generate the results of interest:

```bash
make score-decode-grapheme_confusion
make score-decode-normalized-grapheme_confusion
make score-decode-morph-grapheme_confusio
make score-decode-morph-normalized-grapheme_confusion

make eval-decode-grapheme_confusion
make eval-decode-normalized-grapheme_confusion
make eval-decode-morph-grapheme_confusio
make eval-decode-morph-normalized-grapheme_confusion
```

We then obtained the following results:

#	System	TWV All	TWV In-vocabulary (TWV IV)	TWV Out-of-vocabulary (TWV OOV)	PFA (probability of false alarm)	PM (probability of missing)	Dec. Tresh
1	`decode.ctm` (base)	0.319	0.401	0.000	**0.00002**	0.663	0.0425
2	`decode-morph.ctm` (morph-based system)	0.317	0.381	**0.068**	0.00003	0.651	0.0373
3	`decode-morph-custom.ctm` (base+mapping)	0.311	0.387	0.018	0.00004	0.652	0.0370
4	`decode.ctm` (base + score normalization)	0.320	0.402	0.000	**0.00002**	0.663	0.0074
5	`decode-morph.ctm` (morph-based system + score normalization)	0.325	0.391	**0.068**	0.00003	0.651	0.0040
6	`decode-morph-custom.ctm` (base+mapping + score normalization)	0.316	0.393	0.018	0.00004	0.662	0.0040
7	`decode.ctm` (base + grapheme confusion)	0.269	0.346	-0.028	0.00011	**0.623**	0.0425
8	`decode-morph.ctm.ctm` (base + grapheme confusion)	0.285	0.342	0.064	0.00007	0.649	0.0076
9	`decode-morph.ctm.ctm` (base + grapheme confusion + score normalization)	0.323	0.389	0.067	0.00007	0.649	0.0007
10	`decode.ctm` (base + grapheme confusion + score normalization)	**0.337**	**0.407**	0.067	0.00011	**0.623**	0.0006

Table 4: Keyword spotting results comparing index search with base configuration, morphological decomposition, score normalization and grapheme confusion. Best result for each metric is highlighted.

Observations:

- Combining morphological decomposition and grapheme confusion is redundant. Even worse, it actually makes the system less performant in general (see system #8 which achieves the worst TWV All score of all systems).
- The best system with respect to TWV All is system #10 with grapheme confusion and score normalization with a score of 0.337. It is also the model with the lowest probability of missing ($PM = 0.623$). The fact that this system misses more words comes from the fact that grapheme confusion can only output words from the IV set: this bias seems to guide the system in the correct direction.

6. System Combination

The paragraph discusses how combining the outputs of Automatic Speech Recognition (ASR) and Keyword Spotting (KWS) systems can lead to improved performance. System combination involves taking the detections of multiple systems and generating a new list of hits resulting from the aggregation of all systems.

KWS system outputs are combined by merging their query hits if they overlap in time and refer to the same document. Note that merging consists of a simple sum of the hit scores.

To implement our algorithm for system combination, we did the follwoing:

- We started to parse all given output XML files and store their detected keywords in dictionary that maps keyword IDs to a instance of `DetectedKWList` which is a class that models the `<detected_kwlist>` element in the XML file output by the Index search.
- We created an empty dictionary `answer` which we will populate later.
- For each XML file and for each `DetectedKWList`, we updated the `answer` dicitonary accordingly.
- We finally formatted `answer` as a XML output/

For the complete code, refer to the Appendix.

Due to a lack of time, we weren't able to obtain results for this part.

7. Conclusion

The study investigates keyword spotting using written queries on Swahili, a low-resource language.

First, the index for keyword spotting was designed as a 2-depth nested Python dictionary for efficiency while the search algorithm was implemented using a priority queue and a stack data structure. As we are discarding the hit sequences that did not satisfy the constraints on-the-fly, our Index implementation is relatively fast.

Then, we explored the use of morphological decomposition, grapheme systems, and score normalization to handle OOV terms and boost system performance. The system that had the best performance is the one combining grapheme confusion and score normalization with a TWV All of 0.337.

Finally, we were interested in the potential benefits of merging the outputs of our different systems. Unfortunately, we lacked the time needed to generate results using our code.

To conclude, we learned that indexing is a key component in detection of a particular word or phrase in a stream of audio.

8. Appendix

8.1. The `GraphemeConfusion` class

```
from typing import Optional, Set
from kws.grapheme_confusion.utils import get_iv_set, load_grapheme_confusion


SIL_TOKEN = "sil"


class GraphemeConfusionBase:
    """
    Base class for grapheme confusion models.
    Assumes that all cost functions are unit cost.
    """

    def _levenshtein_distance(self, word1: str, word2: str) -> float:
        """Compute Levenshtein distance between two words."""
        n1 = len(word1)
        n2 = len(word2)

        # Use backtracking to compute Levenshtein distance:
        cache = {}

        def helper(p1, p2) -> int:
            # --- CACHING ---
            if (p1, p2) in cache:
                return cache[(p1, p2)]

            # --- MAIN ---
            if p1 == n1:
                total_insertion_cost = 0
                for char in word2[p2:]:
                    total_insertion_cost += self._insertion_cost(char)
                cache[(p1, p2)] = total_insertion_cost
                return cache[(p1, p2)]
            elif p2 == n2:
                total_insertion_cost = 0
                for char in word1[p1:]:
                    total_insertion_cost += self._deletion_cost(char)
                cache[(p1, p2)] = total_insertion_cost
                return cache[(p1, p2)]

            else:
                if word1[p1] == word2[p2]:
                    cache[(p1, p2)] = helper(p1+1, p2+1)
                    return cache[(p1, p2)]
```

```
                else:
                    cache[(p1, p2)] = min(
                        helper(p1+1, p2) + self._deletion_cost(word1[p1]),
                        helper(p1, p2+1) + self._insertion_cost(word2[p2]),
                        helper(p1+1, p2+1) + self._substitution_cost(word1[p1],
word2[p2])
                    )
                    return cache[(p1, p2)]

        return helper(p1=0, p2=0)


    def _insertion_cost(self, char: str) -> float:
        """Unit cost for insertion of a character"""
        return 1.0


    def _deletion_cost(self, char: str) -> float:
        """Unit cost for deletion of a character"""
        return 1.0


    def _substitution_cost(self, char_1: str, char_2: str) -> float:
        """Unit cost for substitution of a character"""
        return 1.0


class GraphemeConfusion(GraphemeConfusionBase):
    """Class that implements grapheme confusion."""

    def __init__(self,
                 grapheme_confusion_filepath: str,
                 ctm_filepath: str):
        super().__init__()
        self.confusion_dict = load_grapheme_confusion(grapheme_confusion_filepath)
        self.iv_set = get_iv_set(ctm_filepath)


    def is_iv_word(self, word: str) -> bool:
        return word in self.iv_set


    def get_closest_iv_word(self, oov_word: str, subset: Optional[Set[str]]=None) ->
Optional[str]:
        if subset is not None:
            iv_set = self.iv_set.intersection(subset)
        else:
            iv_set = self.iv_set
```

```
        min_dist_iv_word = None
        min_dist = float("inf")
        for iv_word in iv_set:
            dist = self._levenshtein_distance(oov_word, iv_word)
            if dist < min_dist:
                min_dist = dist
                min_dist_iv_word = iv_word
        return min_dist_iv_word


    def _similarity_score(self, s0: str, s1: str) -> float:
        max_length = max(len(s0), len(s1))
        if max_length == 0:
            return 0.0
        # Note: We have to divide the distance by the max length to make the
Levenshtein distance output a value in [0, 1].
        return 1.0 - (self._levenshtein_distance(s0, s1) / max_length)


    # --------- COST FUNCTIONS ---------
    # We override the default cost functions with the ones from the grapheme confusion
matrix.
    def _insertion_cost(self, char: str) -> float:
        return 1 - self.confusion_dict[SIL_TOKEN][char]


    def _deletion_cost(self, char: str) -> float:
        return 1 - self.confusion_dict[char][SIL_TOKEN]


    def _substitution_cost(self, char_1: str, char_2: str) -> float:
        return 1 - self.confusion_dict[char_1][char_2]
```

8.2. System combination code

`detected_kwlist.py`:

```
from dataclasses import dataclass
from typing import List

@dataclass()
class DetectedKW():
    """
    Metadata of a HitSequence found and stored in the XML file output by the
    Index search (see <kw> element)
    """
    file: str
    channel: int
    tbeg: float
    dur: float
    score: float
    decision: bool

    def overlap(self, other: 'DetectedKW') -> bool:
        """
        Check if two DetectedKW objects overlap in time
        """
        return self.file == other.file and self.channel == other.channel and self.tbeg
< other.tbeg + other.dur and self.tbeg + self.dur > other.tbeg


@dataclass()
class DetectedKWList():
    """
    Metadata of a <detected_kwlist> element in the XML file output by the Index search
    """
    kwid: str
    oov_count: int
    search_time: float
    list_kw: List[DetectedKW]

    def merge(self, other: 'DetectedKWList') -> None:
        """
        Merge two DetectedKWList objects, i.e. merge the list of DetectedKW objects

        Notes:
        - KWS system outputs are combined by merging their query hits if they refer to
the same document and if the time overlaps.
        - Scores are combined by summing them.
        """
```

```
        assert self.kwid == other.kwid, f"Cannot merge two DetectedKWList objects with
different kwid: {self.kwid} and {other.kwid}"
        for kw in other.list_kw:
            found = False
            for kw2 in self.list_kw:
                if kw.overlap(kw2):
                    kw2.score += kw.score
                    found = True
                    break
            if not found:
                self.list_kw.append(kw)
        return
```

`combine_systems.py`:

```
def combine_systems(list_xml_filepath: List[str]) -> Dict[str, DetectedKWList]:
    """
    Combine the outputs of multiple KWS systems into a single output.

    Notes:
    - KWS system outputs are combined by merging their query hits if they refer to the
same document and if the time overlaps.
    - Scores are combined by summing them.
    """

    # Parse the output of each system
    list_kwid_to_detected_kwlist = [parse_output_xml(filepath) for filepath in
list_xml_filepath]

    answer: Dict[str, DetectedKWList] = {}

    for kwid_to_detected_kwlist in list_kwid_to_detected_kwlist:
        for kwid, detected_kwlist in kwid_to_detected_kwlist.items():
            if kwid not in answer:
                answer[kwid] = detected_kwlist
            else:
                answer[kwid].merge(detected_kwlist)

    return answer
```

`utils.py`:

```
from pathlib import Path
from typing import Dict
```

```
from bs4 import BeautifulSoup

from kws.system_combination.detected_kwlist import DetectedKW, DetectedKWList


def parse_output_xml(filepath: str) -> Dict[str, DetectedKWList]:
    assert Path(filepath).exists(), f"File {filepath} does not exist"

    with open(filepath, 'r') as f:
        soup = BeautifulSoup(f.read(), 'xml')

    list_detected_kwlist = soup.find_all('detected_kwlist')
    detected_kwlists_dict = {}

    for detected_kwlist in list_detected_kwlist:
        kwid = detected_kwlist['kwid']
        oov_count = int(detected_kwlist['oov_count'])
        search_time = float(detected_kwlist['search_time'])
        list_kw = []
        for kw in detected_kwlist.find_all('kw'):
            file = kw['file']
            channel = int(kw['channel'])
            tbeg = float(kw['tbeg'])
            dur = float(kw['dur'])
            score = float(kw['score'])
            decision = True if kw['decision'] == 'YES' else False
            detected_kw = DetectedKW(file=file, channel=channel, tbeg=tbeg, dur=dur,
score=score, decision=decision)
            list_kw.append(detected_kw)
        detected_kwlist = DetectedKWList(kwid=kwid, oov_count=oov_count,
search_time=search_time, list_kw=list_kw)
        detected_kwlists_dict[kwid] = detected_kwlist

    return detected_kwlists_dict
```

8.3. Makefile

```
# ---- SCORING ----

score-reference:
  python search.py lib/kws/queries.xml lib/ctms/reference.ctm reference.xml



score-reference-normalized:
  python search.py lib/kws/queries.xml lib/ctms/reference.ctm reference-normalized.xml
--normalize-scores



score-reference-morph:
  python search.py lib/kws/queries-morph.xml lib/ctms/reference-morph.ctm reference-
morph-custom.xml



score-reference-grapheme_confusion:
  python search.py lib/kws/queries.xml lib/ctms/reference.ctm reference-
grapheme_confusion.xml --use-grapheme-confusion



score-reference-normalized-grapheme_confusion:
  python search.py lib/kws/queries.xml lib/ctms/reference.ctm reference-normalized-
grapheme_confusion.xml --normalize-scores --use-grapheme-confusion



score-decode:
  python search.py lib/kws/queries.xml lib/ctms/decode.ctm decode.xml



score-decode-morph:
  python search.py lib/kws/queries-morph.xml lib/ctms/decode-morph.ctm decode-morph.xml



score-decode-morph-custom:
  python search.py lib/kws/queries-morph.xml lib/ctms/decode-morph-custom.ctm decode-
morph-custom.xml



score-decode-normalized:
  python search.py lib/kws/queries.xml lib/ctms/decode.ctm decode-normalized.xml --
normalize-scores



score-decode-morph-normalized:
```

```
	python search.py lib/kws/queries-morph.xml lib/ctms/decode-morph.ctm decode-morph-normalized.xml --normalize-scores


score-decode-morph-custom-normalized:
	python search.py lib/kws/queries-morph.xml lib/ctms/decode-morph-custom.ctm decode-morph-custom-normalized.xml --normalize-scores


score-decode-grapheme_confusion:
	python search.py lib/kws/queries.xml lib/ctms/decode.ctm decode-grapheme_confusion.xml --use-grapheme-confusion


score-decode-normalized-grapheme_confusion:
	python search.py lib/kws/queries.xml lib/ctms/decode.ctm decode-normalized-grapheme_confusion.xml --normalize-scores --use-grapheme-confusion


score-decode-morph-grapheme_confusion:
	python search.py lib/kws/queries-morph.xml lib/ctms/decode.ctm decode-morph-grapheme_confusion.xml --use-grapheme-confusion


score-decode-morph-normalized-grapheme_confusion:
	python search.py lib/kws/queries-morph.xml lib/ctms/decode.ctm decode-morph-normalized-grapheme_confusion.xml --normalize-scores --use-grapheme-confusion



# ---- EVALUATION ----

eval-reference:
	rm -rf scoring/* \
	&& scripts/score.sh output/reference.xml scoring \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference.xml scoring all \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference.xml scoring iv \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference.xml scoring oov


eval-reference-morph:
	rm -rf scoring/* \
	&& scripts/score.sh output/reference-morph-custom.xml scoring \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference-morph-custom.xml scoring all \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference-morph-custom.xml scoring iv \
	&& scripts/termselect.sh lib/terms/ivoov.map output/reference-morph-custom.xml scoring oov
```

```
eval-reference-normalized:
  rm -rf scoring/* \
  && scripts/score.sh output/reference-normalized.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-normalized.xml scoring
all \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-normalized.xml scoring
iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-normalized.xml scoring
oov


eval-reference-grapheme_confusion:
  rm -rf scoring/* \
  && scripts/score.sh output/reference-grapheme_confusion.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-grapheme_confusion.xml
scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-grapheme_confusion.xml
scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/reference-grapheme_confusion.xml
scoring oov


eval-decode:
  rm -rf scoring/* \
  && scripts/score.sh output/decode.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode.xml scoring oov


eval-decode-morph:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph.xml scoring oov


eval-decode-morph-custom:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph-custom.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom.xml scoring
all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom.xml scoring
iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom.xml scoring
oov
```

```
eval-decode-normalized:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-normalized.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized.xml scoring all
\
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized.xml scoring iv
\
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized.xml scoring oov


eval-decode-morph-normalized:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph-normalized.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized.xml
scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized.xml
scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized.xml
scoring oov


eval-decode-morph-custom-normalized:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph-custom-normalized.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom-
normalized.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom-
normalized.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-custom-
normalized.xml scoring oov


eval-decode-grapheme_confusion:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-grapheme_confusion.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-grapheme_confusion.xml
scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-grapheme_confusion.xml
scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-grapheme_confusion.xml
scoring oov


eval-decode-normalized-grapheme_confusion:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-normalized-grapheme_confusion.xml scoring \
```

```
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized-
grapheme_confusion.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized-
grapheme_confusion.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-normalized-
grapheme_confusion.xml scoring oov


eval-decode-morph-grapheme_confusion:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph-grapheme_confusion.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-
grapheme_confusion.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-
grapheme_confusion.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-
grapheme_confusion.xml scoring oov


eval-decode-morph-normalized-grapheme_confusion:
  rm -rf scoring/* \
  && scripts/score.sh output/decode-morph-normalized-grapheme_confusion.xml scoring \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized-
grapheme_confusion.xml scoring all \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized-
grapheme_confusion.xml scoring iv \
  && scripts/termselect.sh lib/terms/ivoov.map output/decode-morph-normalized-
grapheme_confusion.xml scoring oov



# ---- APPLY MORPH TRANSFORMATION ----

apply_morph_to_ctm:
  python apply_morph_to_ctm.py lib/ctms/reference.ctm lib/dicts/morph.dct reference-
morph-custom.ctm \
  && python apply_morph_to_ctm.py lib/ctms/decode.ctm lib/dicts/morph.dct decode-morph-
custom.ctm


apply_morph_to_queries:
  python apply_morph_to_queries.py lib/kws/queries.xml lib/dicts/morph.kwslist.dct
queries-morph.xml
```