Please note that the code for my files is kept at MLSALT5. The file that runs all the main scripts is called build.py.

Q1) Implementation of an Index

My implementation of an index for the output of an ASR has two methods, and is detailed in the appendix.

- The first method initialise() reads the ASR output file, stores the entries and sets up other important variables that will be used for querying.
- The second method query() takes as input a query file in xml format and outputs an xml file of hits.

I shall now describe the initialisation method in further detail. The initialise method sets two important variables, **all_words** and **index**, and relies on the class **wordEntry,** which are all described below.

- **all_words** will contain the entries of the ASR output file in time order. The special **None** value is inserted between entries that are 0.5 seconds apart. This will make the job of parsing valid phrases (i.e. those containing pauses shorter than 0.5 seconds) easier, since **None** will act as a marker to breaks up valid phrases.
- **wordEntry** is a class I created to represent a (word) entry from the ASR output. A wordEntry object has all the same fields that a real word-entry has, for instance: the start time, the string of characters that constitute the speech act, the score etc.
- **index** is a mapping from words to the places where they occur in **all_words**. So, if the word 'hello' occurs at places 7 and 8 in the **all_words** list, then index['hello'] returns the list [7,8]. This will make querying phrases much faster, as we shall see in the next part.

I shall now explain how the method query() processes an xml file of queries. For a given query, say 'My name is Bert', we split the query into a list of words. We then use **index** to find the places in **all_words** where the first word in the query occurs. For the example given, we would call Index['My'] and might have returned a list like [8,63,932]. Using this list of indices, we then check if the phrases from **all_words**, which stores the ASR output, match the query_phrase 'My name is Bert'. For the example case, we would check if **all_words[8:12]**== 'My name is Bert'; **all_words[63:67]**== 'My name is Bert'**; all_words[932:936]**== 'My name is Bert' . If any of the following phrases in all_words match the query, we would then store it as a hit and write the information to an output xml file.

I shall now analyse the running time. Because we only ever need to consider the first m=length(query) words from **all_words** at the indices of the hits for the first word in the query, the run time for a single query will be :

$$\text{O(length(query)} \cdot \text{\# hits for 1st word in query)}$$

On average, this is approximately≈O(average # of hits for a single word) because: (1) the length of a query is likely to be less than sentence length which is a small constant; (2) on average [# hits for 1st word in query]=[average # of hits for a single word].

Results:

I then checked that my indexer worked correctly by evaluating the performance the reference file, once indexed, over the given set of queries in the query file queries.xml. To obtain the results tabulated below, I ran the following commands:

```
./scripts/termselect.sh lib/terms/ivoov.map Q1_score.xml scoring all
./scripts/termselect.sh lib/terms/ivoov.map Q1_score.xml scoring iv
./scripts/termselect.sh lib/terms/ivoov.map Q1_score.xml scoring oov
```

The table below shows that my index file for the reference text was working as it should. For both the queries that are in the vocabulary of the speech recognition system and those that are not, we have a MTWV of 1, with a threshold of 1. What this means is that counting only as hits, the entries with scores ≥ 1, there are no misses or false alarms generated. As a note, even though the entry 'oov' stands for out of vocabulary, these words are of course not out of vocabulary for the reference file, but they will be for the outputs of other ASRs.

For future reference, the MTWV is the maximum of the average term weighted value, which is:

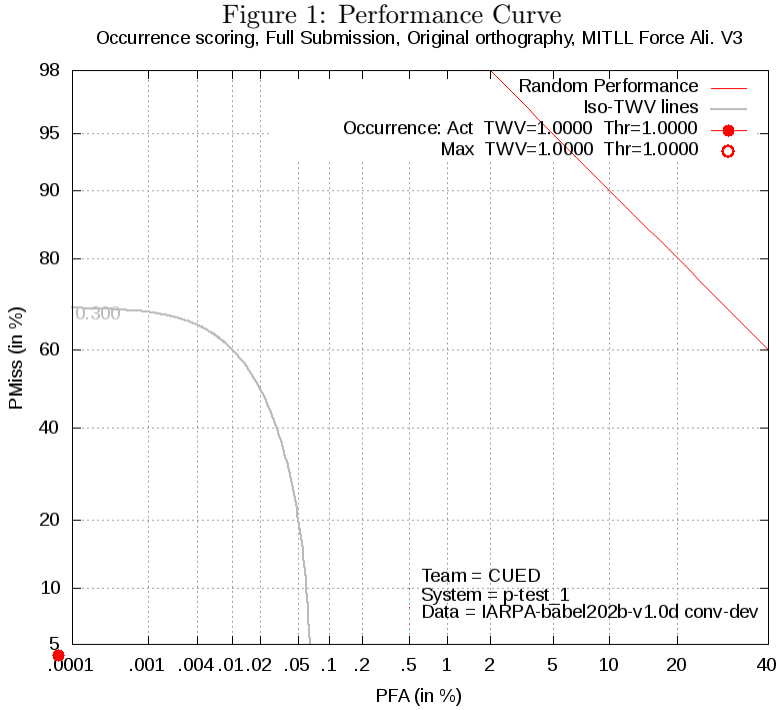
$$\text{ATWV} = 1 - (\text{P}_{\text{miss}}(\theta) + \text{P}_{\text{FA}}(\theta))$$

where beta is some constant, (for the Babel systems $\beta = 999.999$).

	TWV	Threshold	number
all	1	1	488
iv	1	1	388
oov	1	1	100

I have also included the DET (Detection Error Trade Off) curve which, in this case, shows perfect performance. The reason the DET curve is a dot, and not a curve as we might expect is because the best TWV is 1, any less will reduce the performance. As such, there is no need to lower the threshold (the score at which we consider a word/entry from our querying program a hit) .

Figure 1: Performance Curve



Q2) Querying an Index, based on the decoded file from a word based system

For this question, I used the same indexer program from question 1 to generate an index from the file decode.ctm, which is the decoding output from a word-based recognition system. As part of the question, it was remarked that we should be careful about how we combine confidence scores for given phrases. When querying the indexer for a phrase, my program will multiply the scores from the constituent words of a phrase to produce a score for the phrase as a whole. This is just the same as the probability that a 1-best list will assign to seeing a particular phrase, since a phrase is a segment of the 1 best list constituted of edges with particular scores.

Results:

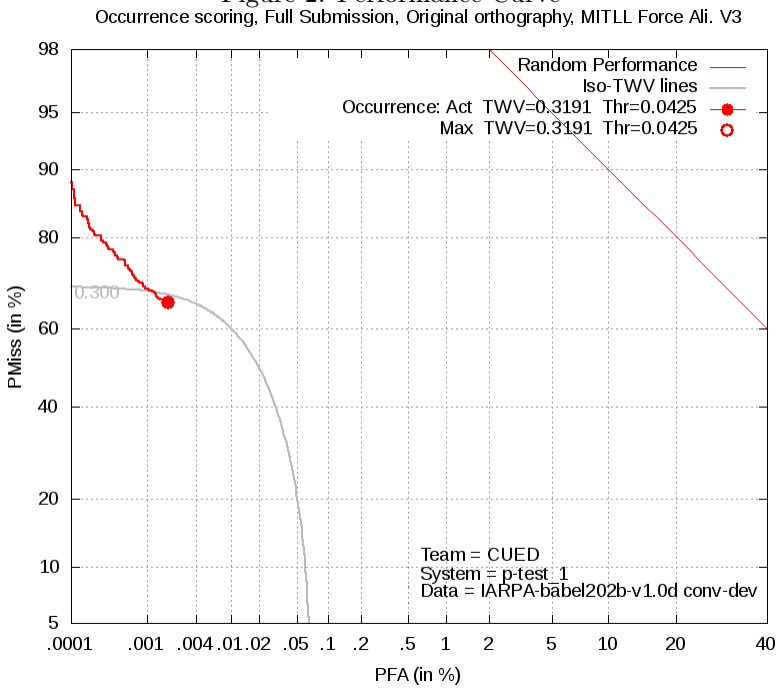
Below I present the performance of the decode.ctm file, once indexed, over the given set of queries in queries.xml. As we should expect, the performance is worse than that of the reference file from question 1. The TWV is much lower at roughly 0.4011 for in vocabulary queries and, as expected, 0 for queries that are not in the vocabulary of the ASR.

	TWV	Threshold	number
all	0.3189	0.043	488
iv	0.4011		388
oov	0.0		100

I have also included the DET curve, so that one can see how as the threshold for a hit is lowered from 1, the number of 'misses' decreases but the number of 'false alarms' increases. In our case, the scoring script settled for a threshold of 0.043, which is

much lower than a value of 1 which represents absolute certainty. What this shows is that in general the ASR systems are not very confident (as measured by probabilities from 0 to 1) when giving predictions about whether a word or phrase has occurred.

Figure 2: Performance Curve



Q3)

The program **MorphDecomp** listed in the appendix performs a morphological decomposition on both the output from an ASR and an xml query file. It has two essential parts: (1) a function dictmap() that reads the morphological mappings from given input files into 2 separate dictionaries; (2) a second function convertQuery(), which will convert both a query file composed of ordinary words and an ASR output file composed of ordinary words into respective query and ASR files that have the words replaced with their morphologically decompositions. I shall now explain the function in slightly more detail.

1. There are two files that contains a morphological mapping from words to sub-words, one for the ASR output , the other for the query file. Within dictmap(), I simply read these mappings into two separate dictionaries.
2. The function convertQuery() first converts the query file, which is an xml file with query phrases stored in certain elements that have a tag 'kwtext'. These phrase are then mapped, using the dictionaries created in part (1), to phrases whose basic units are the morphological decompositions. Next, the function takes the output of the ASR, which is in the form of a 1-best list of words with their respective attributes. The function takes this 1-best list and replaces the words with phrases whose basic units are the appropriate morphological decompositions.

After all of this has been accomplished, we can use the method described in Q1 to generate an xml file of hits for the relevant queries with respect to the output from an ASR. The only difference is that the query file and the ASR output, used, are now the ones that have had the morphological decompositions applied.

NB: An important point to note is that when decomposing the ASR output, I have tried to ensure that the probability of a word is equal to that of its decomposition. So, if the word 'halo' is decomposed into 'ha' and 'lo', and there is an entry for 'halo' in the ASR output file with score x, then in the decomposed file, there will be two entries 'ha' and 'lo' with scores such that score('ha')*score('lo')=x.

Results:

Below, I present the performance, of the morphologically decomposed and indexed file, over the given set of queries. There seems to be little to no change in performance from Question 2. In fact, pretty much all of the key variables, summarised in the table below, take on the same value. This is for essentially two reasons.

1. The score of a given hit that existed before the morphological decomposition is the same after. This is due to the explanation given in the note (**NB**). Eg. If there was a hit for 'halo' when querying without morphological decomposition, then there will be a corresponding hit for the morphological decomposition 'ha' 'lo' with precisely the same score.

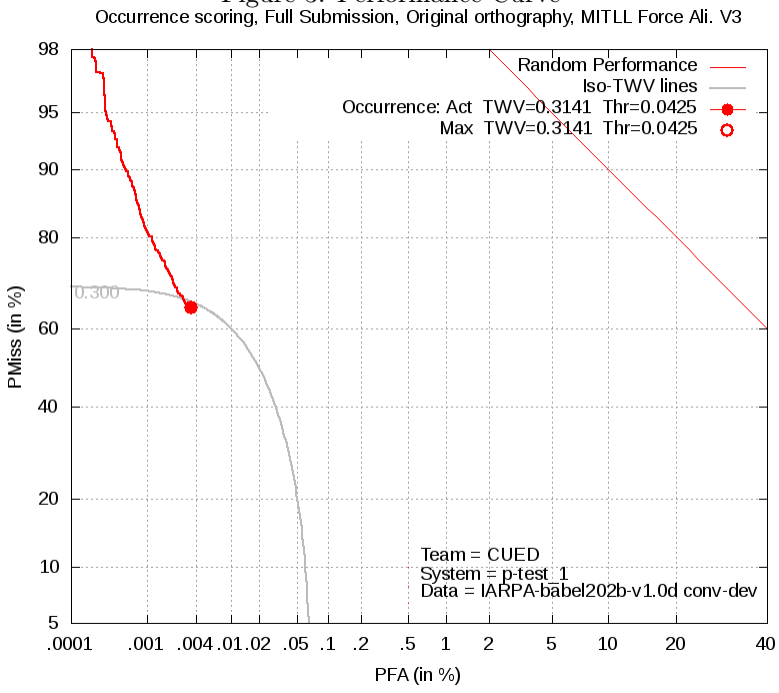
2. The number of hits is not decreased by morphological decomposition but only slightly increased. Before morphological decomposition the word 'catapulted' might not score a hit for 'catapult' but afterwards it will, assuming it is decomposed to 'catapult'-'ed'. However, the effect of this decomposition is limited to only very short phrases, and so does not have a great effect on improving the MTWV as a hole.

An interesting point to note is that the OOV TWMV has increased, while the IV TWMV has decreased from Q2. This is mainly because of point (2) as mentioned above. If, for example, 'catapult' is not in the vocabulary but 'catapulted' is, then a morphological decomposition can help us score more highly on the word 'catapult', thereby improving the OOV score. However the converse effect can also be observed, if the phrase 'the Alabama men' in the decoded file is decomposed to 'the' 'alabam' 'a' 'men', then we may get a gratuitous hit for 'amen'. Essentially, decompositions can increase the number of hits in OOV phrases but also increase the number of FAs in IV phrases.

	TWV	Threshold	number
all	0.3189	0.043	488
iv	0.3901	0.043	388
oov	0.018	0.043	100

I have also included the DET curve below. As we can see, the DET curve moves to a position with more false alarms but fewer misses than in Q2. In our case, the scoring script settled for a threshold of 0.0425, which is still very similar to that of Q2.

Figure 3: Performance Curve



Q4) Normalisation Sum to One

For this question, I implemented a sum-to-one normaliser, with exponent set to 1. The code for this question is listed in the appendix. To implement a sum-to-one normaliser, we need to ensure that for each query, the scores of its respective hits sum to one. This is summarised by the formula:

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

where s_{ki} is the i the hit for keyword k and the summation is over all hits for keyword k. In my implementation, once the normalisation has been performed for a given score file, I then output the normalised file.

Results:

The first set of results are generated from the normalising the word based ASR output in Q2. There is very little improvement that can be seen when normalising. According to [1], 'normalisation is motivated by the desire to boost the scores of keywords with generally low scores (low denominator) and to give lower cost to misses for keywords with more true tokens'. Intuitively, this makes sense. If we have few hits for a query, we would want to boost the scores for those hits so that they are not discounted because their scores don't pass the threshold.

However, there seems to be very little effect of score normalisation on the systems we have produced. This might be due to the peculiarities of the ASR system or because with real datas 'sometimes approaches that should work do not, and results are not always consistent.'

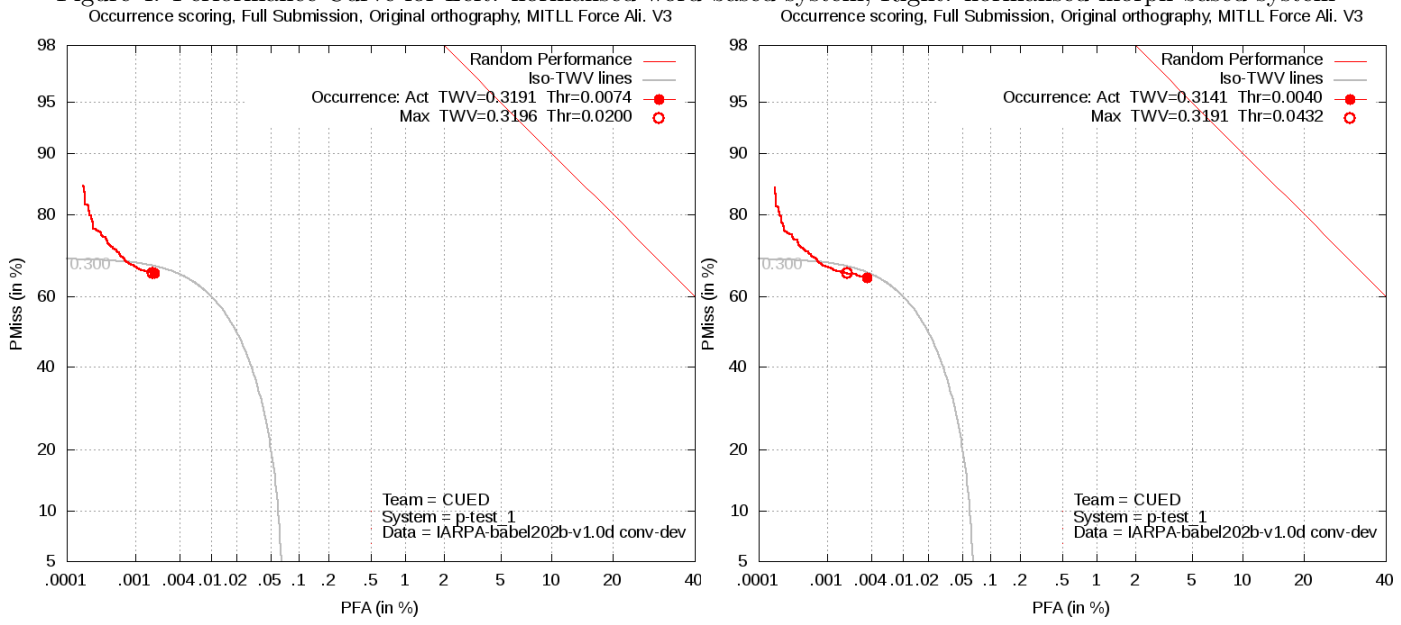
Another interesting point: the threshold has been reduced from the value in pervious questions. This may well be because for most queries, the hits generated are greater than one, and so normalisation will tend to lower scores.Ceteris paribus, with lower scores we should expect lower thresholds found by the scoring function .

	TWV	Threshold	number		TWV	Threshold	number
all	0.3195	0.020	488	all	0.3189	0.020	488
iv	0.4019		388	iv	0.3964		388
oov	0		100	oov	0.018		100

Table 1: Left: normalised word-based system; Right: normalised morph-based system

The DET curves below look shallower than in previous questions, and the shape has changed because the normalisation has altered the trade off between misses and false alarms.

Figure 4: Performance Curve for Left: normalised word-based system; Right: normalised morph-based system



Q5) System Combination

For this question, I developed a script that would combine the scores from a list of scored files. My code is detailed in the appendix, under the heading System Combination. There are some important features to note.

[1] http://www.fit.vutbr.cz/research/groups/speech/publi/2013/karakos_asru2013_0000210.pdf

- A special requirement of my system is that the first file must be complete, in sense that it should have 'kwid' entries for each of the queries we wish to consider.
- Another special feature of my function is that there is a hitclass class.
  - A hitclass object is designed to store all of the relevant information from a hit from the scored file.
  - **V. important**: the class has a specially designed equality method, where two hits will be considered as equal if their time-instances overlap and they are from the same file.
  - There is also a method to combine two hits, which sums the respective scores of each hit.

The hitclass makes combining a set of hits very easy and also **fast**. To combine a set of hits we do the following:

```
Function: combine_set
Input: set of hits

For hit in set of hits:
    if Hash(hit) =None:         // No overlapping hit found, set hash(hit) to itself
        Hash(hit):=hit
    else:                       // An overlapping (by time stamps) hit hound
        Hash(hit)=hit.combine(Hash(hit))

Return Range of Hash
```

What we the above code does is combine hits in a set. It works because hits are only considered equal if their time-stamps overlap. So if hit 1 and hit 2 overlap and we hashed hit1 first, then hash(hit1)=hash(hit2)=hit1. Because of this, we know we must combine hash(hit2)=hit1 with hit1 in the code above. Since we have used a hash function, combining the hits in a given set will take only O(|set of hits|) time.

Recombination

Now that I have explained the background information, we can move on to how system combination is performed. To score all the files, we simply do the following in the pseudo code below:

```
Input: A set of query files

xml tree := new xml tree
For each query from the queries in the first file:
    Bag of hits:= all of the hits from other files also belonging to that query .
    new_hits   :=combine_set(Bag of Hits)
    store the query and its hits in xml tree
write the xml tree to a file
```

Please note that the pseudo-code above is designed to be instructive and, for instance, there is no function combine_set() within my code; instead the commands undertaken by the function call are performed within the function combine(). After recombination, we normalise the newly combined file.

Part 1)

I carried out the investigation required from the first part of Q5, by combining scores from the morphological and the word based systems I had produced. The table below demonstrate the results.

	TWV	Threshold	number		TWV	Threshold	number
all	0.3206	0.041	488	all	0.3205	0.052	488
iv	0.3985		388	iv	0.3984		388
oov	0.0180		100	oov	0.0180		100

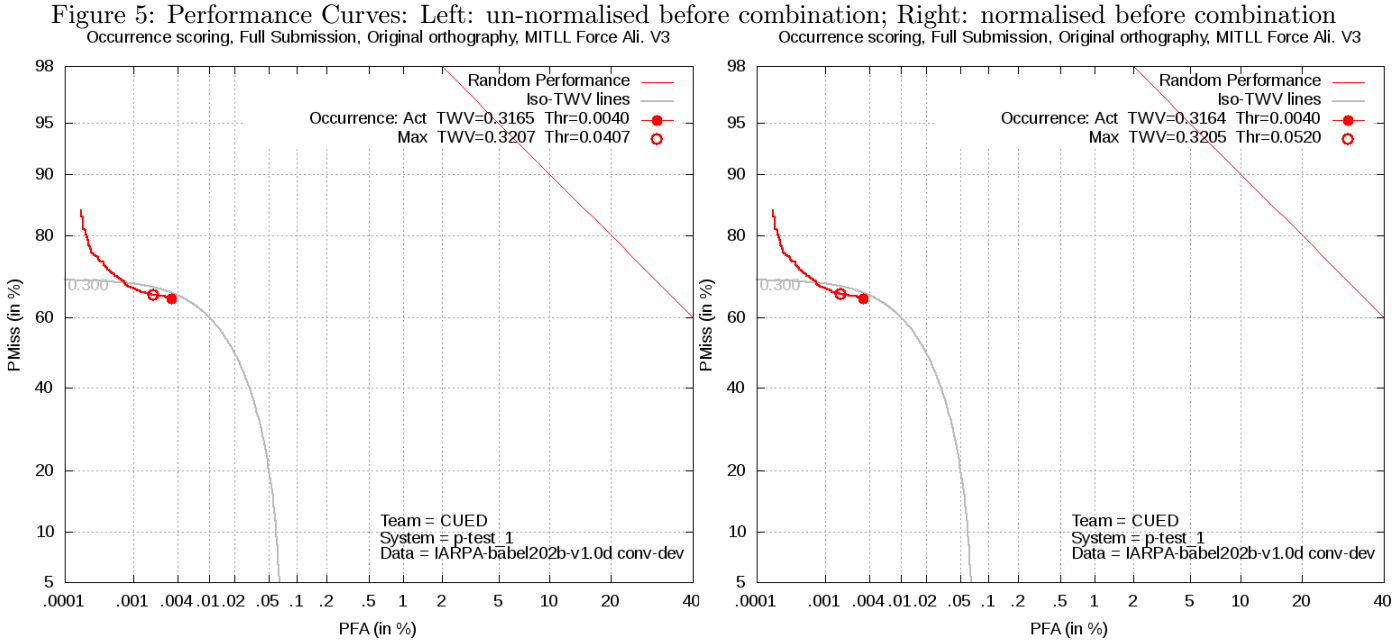
Table 2: Combination of previous word based and morphological based systems; Left: un-normalised before combination; Right: normalised before combination

The system combination resulted in an overall performance increase in TWMV of less than 0.001 for both the un-normalised and the normalised systems. This is unsurprising. There ought to be very little benefit from combining systems that essentially agree on the scores for almost all phrases as the systems above do. Also, it seems there was both very little performance benefit from

STO normalisation both before and after system combination. Our conclusions hold true for both oov queries and IV queries. There was little benefit from combination.

From the DET curves below, we can see that both the systems with and the system without normalisation have a similar Pmiss/FA trade-off as the scoring thresholds are lowered, most liekly because both systems make the same sorts of predictions.

Figure 5: Performance Curves: Left: un-normalised before combination; Right: normalised before combination



Part 2)

Before we move on to system combination for the newly released scored files, it would be good to question whether normalising those files would result in improvements.

Before normalisation, the files obtained the following scores when scored using the score.sh commands:

System	all TWMV	iv TWMV	oov TWMV	Thresh
Morph	0.3597	0.4296	0.0885	0.071
Word	0.4033	0.5072	0	0.047
Word-Sys2	0.3986	0.5014	0	0.030

Table 3: Before Normalisation

However, after normalisation, the scores were greatly improved, as can be seen from the table below. Why the first set of scores did not see an improvement after normalisation, but the IBM scores saw an improvement of between 0.7 and 0.14, but is difficult to know without understanding how the underlying ASR systems work. What is evident, is that the STO normalisation seems to generally improve scores (albeit occasionally by a small amount) and for the released scores can have a large impact on TWMV performance.

Normalisation of the morph based system also resulted in a higher TWV score for oov words, which may be due to reasons expounded from our discussion of morphological decomposition and oov queries from question (Q3).

System	all TWMV	iv TWMV	oov TWMV	Thresh
Morph	0.5180	0.5568	0.3674	0.039
Word	0.4615	0.5804	0	0.038
Word-Sys2	0.4619	0.5810	0	0.030

Table 4: After Normalisation

Following the considerations made in previous sections, I decided to normalise all system scores before recombination. In my final investigation, I combined all subsets of the normalised scored files I had received so far (i.e. [Word-mine, Morph-mine, Morph, Word, Word-Sys2], but I only display some of my results below. Since the Morph system produced, when scored, had by far the best TWV, it would be a good idea to include it in all or most recombinations.

I present the results below:

System Number	Morph-Mine	Word-Mine	Morph	Word	Word-Sys2	all TWMV	iv TWMV	oov TWMV	Thresh
1	**x**	**x**	**x**	**x**	**x**	0.5338	0.5776	0.3637	0.030
2	**x**		**x**	**x**	**x**	0.5285	0.5710	0.3637	0.030
3		**x**	**x**	**x**	**x**	0.5334	0.5776	0.3621	0.030
4			**x**	**x**	**x**	0.4564	0.5741	0	0.034
5	**x**	**x**	**x**		**x**	0.5353	0.5800	0.3618	0.028
6	**x**		**x**		**x**	0.5340	0.5781	0.3629	0.029
7		**x**	**x**		**x**	0.5397	0.5856	0.3612	0.029
8			**x**		**x**	0.5346	0.5786	0.3640	0.033
9	**x**	**x**		**x**	**x**	0.4643	0.5793	0.018	0.032
10	**x**			**x**	**x**	0.4604	0.5744	0.018	0.033
11		**x**	**x**	**x**		0.5371	0.5822	0.3621	0.030

Table 5: Final Investigation: Combination

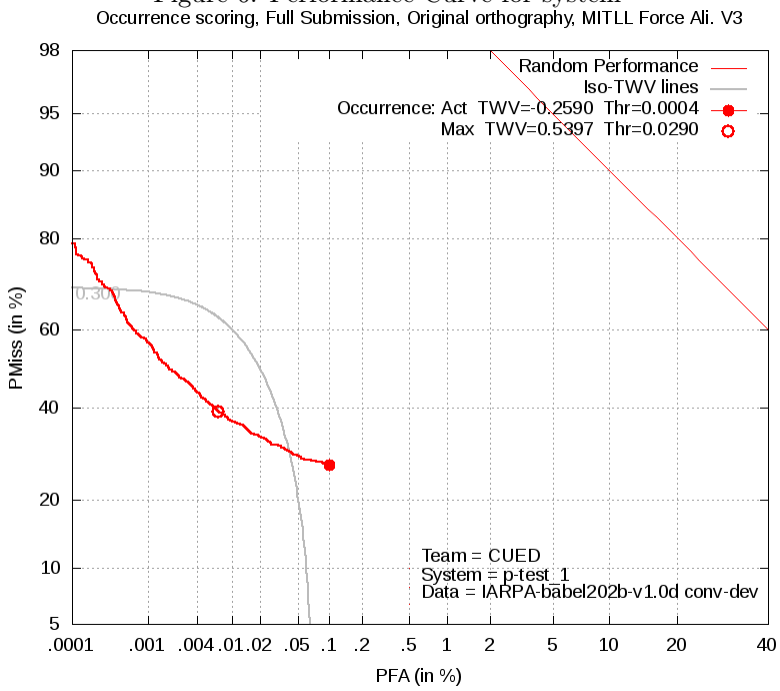
There are a number of conclusions that we can draw from this experiment. In general, system combination seemed to have little effect on oov performance, since the oov TWMV scores are not much higher than that of the oov scores for the best subsystem. I suspect that this is because the phrases that were designated as oov were designated as oov for almost every ASR sub-system.

However in general, system combination should improve oov performance because system 1 might be oov for the word 'cat', and not for the word 'dog', whereas for system 2 the converse might hold true. When combining systems 1 and systems 2, you'd expect the combined system to do reasonable well at recognising both the words 'cat' and 'dog'.

The IV performance tended to improve under system combination. The best system from the table was system 7, which attained a TWMV of 0.5397, which was higher than both of its constituent systems by at least 0.02. A further interesting phenomenon can be observed. The IV TWMV of system 7 is 0.5856, which is much closer to the best TWMV of its constituent systems 0.5810, than the lowest at 0.32. Similarly, the oov score of system 7 is much closer to the highest oov score of its constituent parts, 0.3674, than the lowest at 0. This means that through system recombination, the newly developed systems are much more likely to inherit the better performances of the subsystems they are composed of.

Below, I plot the DET curve for system 7. As one can see, through system combination we make far fewer omissions but also pick up quite a few more false alarms. This might be because when we perform system combination, we can pick up false alarms from any of the systems and so their numbers naturally rise. With regards to target phrases, we are much less likely to miss a target phrase if the scoring is informed by multiple systems.

Figure 6: Performance Curve for system



NB: I should have also liked, if time and space permitted, to do some analysis on the length of queries by modifying the ivoov.map. The file 'home/mec68/MLSALT5/query' is a function that generates a file like ivoov.map but for query lengths.

Appendix:

Indexer:

The code for my implementation is detailed below:

Listing 1: Indexer

```
from operator import mul
import ElementTree as ET
import collections

all_words=[]
index=dict()

class wordEntry:
   def __init__(s,l):
      [s.filn,s.chan,s.tbeg,s.dur,s.word,s.prob]=l
      s.tbeg=float(s.tbeg);s.dur=float(s.dur);s.prob=float(s.prob);s.word=s.word.lower()
      s.end=s.tbeg+s.dur
   def returnValues(s):
      return [s.filn,s.chan,s.tbeg,s.dur,s.word,s.prob]

def initialise(in_dir):
   # create the index indexed by word, with a pointers to occurences in all_speech
   global all_words,index
   f= open(in_dir,'r')
   for line in f:
      nextw=wordEntry(line.split()[0:6])
      if all_words!=[] and  abs(all_words[-1].end-nextw.tbeg)>0.5:
         all_words.append(None);
      if index.get(nextw.word)==None:
         index[nextw.word]=[len(all_words)]
      else:
         index[nextw.word].append(len(all_words))
      all_words.append(nextw)

   f.close()
```

```
def query(to_score,out_score):
   target=open(out_score,'w')
   tree=ET.parse(to_score); root=tree.getroot()
   root.attrib=collections.OrderedDict([('kwlist_filename', "IARPA-babel202b-v1.0d_conv-dev.kwlist.xml"), ('↩
       language', "swahili"), ('system_id', "")])
   root.tag='kwslist'
   for child in root:
      child.tag='detected_kwlist'; child.attrib['oov_count']='0'; child.attrib['search_time']='0.0'
      search_words=child[0].text.split()
      child.remove(child[0])
      if index.get(search_words[0])!=None:
         for ind in index[search_words[0]]:
            wEnt=all_words[ind:ind+len(search_words)]
            entry_words =  [x.word if x!=None else None for x in wEnt ]
            if entry_words== search_words:
               b=ET.SubElement(child,'kw');
               dur=wEnt[-1].end-wEnt[0].tbeg
               prob =reduce(mul,[x.prob for x in wEnt],1)
               b.attrib=collections.OrderedDict([('file',wEnt[0].filn),('channel',str(wEnt[0].chan)),('tbeg',str(↩
                   wEnt[0].tbeg)),('dur',str(dur)),('score',str(prob)), ('decision', "YES")])

   tree.write(target)
```

Morphological Decomposition

Listing 2: Morphological Decomposition

```
import sys
import re
import ElementTree as ET
import Indexer as I
import numpy

all_speech=[]
ref_dict=dict()
query_dict=dict()
ref_dir='/usr/local/teach/MLSALT5/Practical/lib/dicts/morph.dct'
query_dir='/usr/local/teach/MLSALT5/Practical/lib/dicts/morph.kwslist.dct'


def dictMap():
    # Mapping for query
    with open(query_dir) as f:
       for line in f:
        items   = re.split(r'\t+',line.strip('\n'));
        global query_dict
        query_dict[items[0]]=items[1]

    # Mapping for reference
    with open(ref_dir) as f:
       for line in f:
        items   = re.split(r'\t+' ,line.strip('\n'));
        global ref_dict
        ref_dict[items[0]]=items[1]


def convertQuery(in_ref,morph_ref,in_queries,morph_que):
    # convert query
    target=open(morph_que,'w')
    tree=ET.parse(in_queries);root=tree.getroot()
    for kwtext in root.iter('kwtext'):
        #cheat for a bit use ref not morph
        phones= ' '.join([query_dict[x] if query_dict.has_key(x) else x for x in kwtext.text.split()])
        kwtext.text=phones
    tree.write(target)

    # convert reference
    target=open(morph_ref,'w')
    with open(in_ref,'r') as f:
        for line in f:
            [filn,chan,tbeg,dur,word,prob] = I.wordEntry(line.split()[0:6]).returnValues()
            phones= ref_dict[word].split() if ref_dict.has_key(word) else [word]
            n=len(phones)
            t_dur=dur/n
            score=str(round(numpy.exp(numpy.log(prob)/n),4))
            i=0
            for phone in phones:
               star_time=str(round(tbeg+t_dur*i,2))
               duration=str(round(t_dur,2))
               new_line=' '.join([filn,chan,star_time,duration,phone,score])+'\n'
```

```
                target.write(new_line)
                i=i+1

    target.close()

def main(argv):
    dictMap()
    if len(argv)==4:
        convertQuery(argv[0],argv[1],argv[2],argv[3])
    else:
        convertQuery('/usr/local/teach/MLSALT5/Practical/lib/ctms/decode.ctm','/remote/mlsalt-2015/mec68/MLSALT5/↩
            morph_ref.ctm',
```

Normalisation

Listing 3: Normalise

```
import sys
import ElementTree as ET

def normalise(in_score, out_score):
    tree=ET.parse(in_score);root=tree.getroot()
    target=open(out_score,'w')
    for kwquery in root.findall('detected_kwlist'):
        queries=[float(child.attrib['score']) for child in kwquery]
        sumScore=sum(queries);
        for child in kwquery:
            child.attrib['score']=str(float(child.attrib['score'])/sumScore)
    tree.write(target)
```

System Combination:

Listing 4: System Combination

```
import ElementTree as ET
import numpy as np
import Indexer as I
import copy
class hitclass():
    wE=None
    def __init__(s,l):
        s.wE=I.wordEntry(l)
    def __eq__(s,other):
        return np.sign(other.wE.tbeg-s.wE.end)*np.sign(other.wE.end-s.wE.tbeg)<=0 \
               and s.wE.filn==other.wE.filn and  '''s.wE.chan==other.wE.chan''' and s.wE.word==other.wE.word
    def combine(s,other):
            print(2)
            '''if other.wE.prob>s.wE.prob:
                victor=other
            else:
                victor=s
            tbeg=victor.wE.tbeg; end=victor.wE.end; dur=victor.wE.dur'''
            tbeg=min(s.wE.tbeg,other.wE.tbeg);end=max(s.wE.end,other.wE.end);dur=end-tbeg
            prob=s.wE.prob+other.wE.prob
            chan=s.wE.chan; filn=s.wE.filn; word=s.wE.word
            s.wE=I.wordEntry([filn,chan,tbeg,dur,'',prob])
            return s

    def returnValues(s):
      return s.wE.returnValues()

    def __hash__(s):
        return hash((s.wE.filn,s.wE.chan))

    def returnhit(s):
        newElem= ET.Element('kw')
            newElem.attrib={'score':str(s.wE.prob),'tbeg':str(s.wE.tbeg),'file':s.wE.filn,'decision':'YES','dur':↩
                str(s.wE.dur),'channel':str(s.wE.chan)}
        return newElem

def combine(list,out):

    template=copy.deepcopy(ET.parse(list[0])); root=template.getroot()
    roots   =    [ET.parse(x).getroot() for x in list]
    for i in range(0,len(root)):
        que=root[i]
```

```
        hitSet=dict()
        hits=[hit for rooty in roots for query in rooty if query.attrib['kwid']==que.attrib['kwid'] for hit in ↩
            query]
        for hit in hits:
            nhit=hitclass([hit.attrib['file'],hit.attrib['channel'],hit.attrib['tbeg'],hit.attrib['dur'],'',hit.↩
                attrib['score']])

            if hitSet.has_key(nhit):
                if hitSet[nhit].wE.tbeg==111.03: print(1)
                '''print('test')
                print(hitSet[nhit].wE.tbeg)
                hitSet[nhit]=nhit.combine(hitSet[nhit])
                print(hitSet[nhit].wE.tbeg)'''
            else:
                hitSet[nhit]=nhit
                if nhit.wE.tbeg==111.03: print(1)
        root[i]._children=[]
        for hit in hitSet.values():
            root[i].append(hit.returnhit())
            if hitSet[hit].wE.tbeg==111.03: print(1)

    template.write(out)
```