Viterbi 알고리즘을 이용한

Continuous Speech Recognizer 구현

https://github.com/doobee98

1. 요약

이번 과제물에서는 제공되는 파일들과 viterbi 알고리즘을 이용하여 간단한 음성 인식을 구현해본다. 이미 학습이 된 모델과 테스트 데이터를 이용하여, Acoustic Model과 Language Model을 만들어 viterbi 알고리즘을 수행하며, Results.exe 실행파일을 이용하여 인식 결과를 분석해본다. 추가적인 목표는, 과제에만 국한하지 않고 최대한 General하게 만들어서 다른 모델을 분석할 때 코드의 수정이 최대한 적게 만드는 것을 목표로 한다.

Python으로 구현하며, 외부 라이브러리로는 행렬 연산 기능을 제공하는 numpy, numpy에 대한 Type Hint를 제공하는 nptyping을 사용한다.

압축 파일 내에 있는 executable.exe를 실행하여 테스트할 수 있으며, 용량 문제상 테스트 데이터 파일은 제외되어 있으므로 prevData 폴더 내에 tst 폴더를 넣어서 실행하면 된다.

2. 데이터 파일 파싱 (parser/*.py, types/*.py, config.py)

우선 파일들을 파싱하여 읽어 들여야 한다. 제공된 모델 데이터 파일들은 다음과 같으며, 파일의 경로 정보는 config.py에 저장되어 있다.

- hmm.txt – 음소별로 hmm 데이터가 주어진다. 각각의 hmm은 최대 3개(entry와 exit은 제외)의 state를 가지며, 2개의 pdf를 가지고 있다. 각 pdf는 weight(가중치), mean(평

균), variance(분산)로 구성된다. 프로그램 상에서 음성 feature vector는 39차원으로 표현되기 때문에, mean과 variance 모두 39 dimension vector이다. 또한 hmm은 transition probability matrix를 가지며, entry와 exit을 포함해서 (State_Num, State_Num)의 Matrix로 표현된다.

- dictionary.txt – 각 단어의 음소 list 구성 정보를 담고 있다. Silence는 <s>로 표현된다. zero 단어에 대한 음소 list가 두 종류이기 때문에 utterance HMM을 구현할 때 이를 고려해야 할 것임을 알 수 있다.
- unigram.txt, bigram.txt – 각 단어의 unigram, bigram probability 정보를 담고 있다.

```
# Parse Model Files
hmm_dict = HMMParser.parse()
uni_dict = UnigramParser.parse()
bi_dict = BigramParser.parse()
word_list, phone_list_list = DictionaryParser.parse()

# Initialize Word Dictionary
word_dictionary = Dictionary(word_list, phone_list_list, uni_dict, bi_dict)
```

해당 데이터들을 인식하는 각자의 parser를 만든 후, 해당되는 타입의 객체에 넣어주었다. hmmParser는 데이터를 HMM, STATE, PDF 타입 객체로 파싱하며, dictionaryParser는 Dictionary 타입 객체로 파싱한다. unigramParser와 bigramParser에 대응되는 타입은 따로 없으며, 해당 정보를 담은 dict(파이썬 내장 타입)로 파싱하여 Dictionary 타입에서 같이 관리한다. 이에 따라 DictionaryItem 객체에서는 해당 단어의 이름, 음소 list, unigram value, bigram list를 보관한다.

다른 parser들을 제작하는 김에 테스트 파일들을 파싱하는 testParser도 미리 구현하였다. 파일 이름을 parse함수에 전달해주면 feature vector list를 반환한다.

3. Multivariate Gaussian Distribution 구성 (gaussian.py)

Continuous Observation Probability를 계산하기 위해 Gaussian Distribution을 사용한다. 다만 대상 값이 39 Dimension Vector이기 때문에 Multivariate Gaussian Distribution을 사용해야 한다. 공분산 행렬을 구하는 데에 어려움이 있기 때문에 rough하게 계산한다. 또한, 제공받은 데이터 모델에서 2개의 gaussian distribution을 이용하기 때문에, 이를 반영하여 계산한다. 두 요소가 반영된 계산식은 다음과 같다.

$$
\begin{aligned}
b_s(e) &= \sum_{g=1}^{G} c_{sg}\, \mathcal{N}\left(e \middle| \mu_{sg}, \Sigma_{sg}\right) \\
&= \sum_{g=1}^{G} c_{sg}\, \frac{1}{(2\pi)^{D/2}\left|\Sigma_{sg}\right|^{1/2}} \exp\left(-\tfrac{1}{2}\left(e-\mu_{sg}\right)^{T}\Sigma_{sg}^{-1}\left(e-\mu_{sg}\right)\right) \\
&\approx \sum_{g=1}^{G} c_{sg}\, \frac{1}{(2\pi)^{D/2}\prod_i \sigma_{sgi}} \exp\left(-\tfrac{1}{2}\sum_i \frac{\left(e_i-\mu_{sgi}\right)^2}{\sigma_{sgi}^2}\right)
\end{aligned}
$$

해당 식은 각 gaussian distribution의 weight, mean vector, variance vector를 알면 특정할 수 있다. 위의 식을 기반으로 하여 gaussian.py에서는 SingleGaussian, MultiGaussian 클래스를 정의한다. SingleGaussian 클래스는 mean vector와 variance vector로 정의되고, MultiGaussian 클래스는 singleGaussianList와 그에 대응되는 weightList로 정의된다. 두 클래스 모두 특정 vector의 확률을 계산하기 위한 probability 메소드를 제공한다.

4. Word HMM 구성 (utteranceHMM.py: wordHMM() method)

전체 Utterance HMM을 구성하기 위해서 우선 각 단어의 Word HMM을 구성해야 한다. utteranceHMM 클래스의 wordHMM 메소드에서 해당 구성 기능을 제공한다. Word HMM은 Phone HMM을 이어 붙인 모양으로, Phone 사이사이는 앞쪽 Phone의 Exit (Dummy) State가 뒤쪽 Phone의 Entry (Dummy) State로 연결되도록 한다.

Word HMM의 전체적인 구성 과정은 다음과 같다.

1. total_size를 미리 계산해서 전체 transition matrix를 (total_size, total_size)로 초기화한다. total_size는 (Dummy(Entry, Exit) 개수 2) + (모든 Phone의 state 개수의 합)이다.
2. 해당 word의 phone_list를 참조해서 순서에 맞게 phone interation을 한다.
    A. phone HMM을 hmmDict에서 찾아 가져온다.
    B. phone_hmm의 state 순서에 맞게 MultiGaussian 객체를 생성한 후, state_mgd_list에 덧붙인다. phone_hmm에 있는 pdf 정보를 이용한다.

```
if cur_size == 0:
    cur_matrix[:new_size, :new_size] = new_matrix
    cur_size = new_size
else:
    # index naming
    cur_inner_start, cur_inner_size = 1, cur_size - Dummy
    cur_inner_end = cur_inner_start + cur_inner_size
    new_inner_start, new_inner_size = cur_size - 1, new_size - Dummy
    new_inner_end = new_inner_start + new_inner_size

    # set inter (in and out) matrix (right top)
    cur_out = cur_matrix[cur_inner_start-1:cur_inner_end, [new_inner_start]]
    new_in = new_matrix[0, 1:]
    inter_matrix = cur_out * new_in          # numpy broadcasting
    cur_matrix[cur_inner_start-1:cur_inner_end, new_inner_start:new_inner_end+1] = inter_matrix

    # set next inner matrix (right down)
    cur_matrix[new_inner_start:new_inner_end, new_inner_start:new_inner_end+1] = new_matrix[1:-1, 1:]

    # next cur_size
    cur_size += new_size - Dummy
```

C. phone_hmm.tp(transition matrix)를 word matrix에 추가한다. 전체 phone 중에서 첫 추가라면 그냥 copy하고, 아니라면 원래 있던 matrix의 exit이 새로 추가되는 matrix의 entry가 되도록 값을 조정한다. 해당 과정에서 exit state는 열에 해당하고 entry state는 행에 해당하므로, 다른 차원의 array 연산 시 편리하게 계산할 수 있도록 하는 numpy의 broadcasting 성질을 사용한다.

(참조 - (한글) https://sacko.tistory.com/16)

D. 다음 phone에 대해 A~C를 반복한다.

3. 모든 phone에 대해 matrix와 gaussian distribution을 모두 계산하고 추가했으면, 만들어진 state_mgd_list(observation probability)와 matrix(transition probability)를 반환한다.

이와 같이 구성된 Word HMM은, State 개수가 약간 많아진 Phone HMM과 큰 차이를 보이지 않는다. Continuous Speech의 특성상 한 word에서는 state가 거꾸로 돌아가지 않기 때문에, Word Matrix를 출력해보면 많은 값이 0으로 채워져 있는 것을 확인할 수 있다.

5. Utterance HMM 구성 (utteranceHMM.py: __init__() constructor method)

각 Word의 HMM을 만들었으므로 그것들을 연결해 Utterance HMM을 만든다. 기본적인 연결 방법은 wordHMM 메소드와 같지만, phone간의 연결과 달리 word간의 연결은 state의 연속성이 보장되지 않기 때문에 exit과 entry의 연결 방법이 약간 달라진다.

Utterance HMM의 전체적인 구성 과정은 다음과 같다.

1. total_size를 미리 계산해서 전체 transition matrix를 (total_size, total_size)로 초기화한다. total_size는 (Dummy(Entry) 개수 1) + (모든 Word의 state 개수의 합)이다. Word Matrix와 다르게 Exit Dummy State는 존재하지 않는다(모든 feature vector를 검사하면 Exit).

2. 다른 클래스 변수들을 초기화한다. Dummy (Entry) State를 가정하기 때문에 초기 Matrix Size가 1인 것을 유의한다. matrix size는 matrix 정사각행렬의 사이즈이고, gaussianList는 모든 state에 대응되는 MultiGaussian 객체를 보관한다. startIndexList는 각 단어가 matrix에서 어느 index의 state부터 시작되는지를 저장하여, utteranceMatrix에서 word간의 구분자 역할을 한다.

3. wordDictionary의 wordInfoList에 있는 DictionaryItem 객체에 대해 iteration을 한다.

   A. 해당 wordItem에 대한 wordHMM을 계산하여 gaussian_list와 word_matrix를 얻는다.

   B. word_matrix에 대한 몇 가지 가정을 체크한다. 이 조건들을 위배할 시 오류를 발생시킨다.

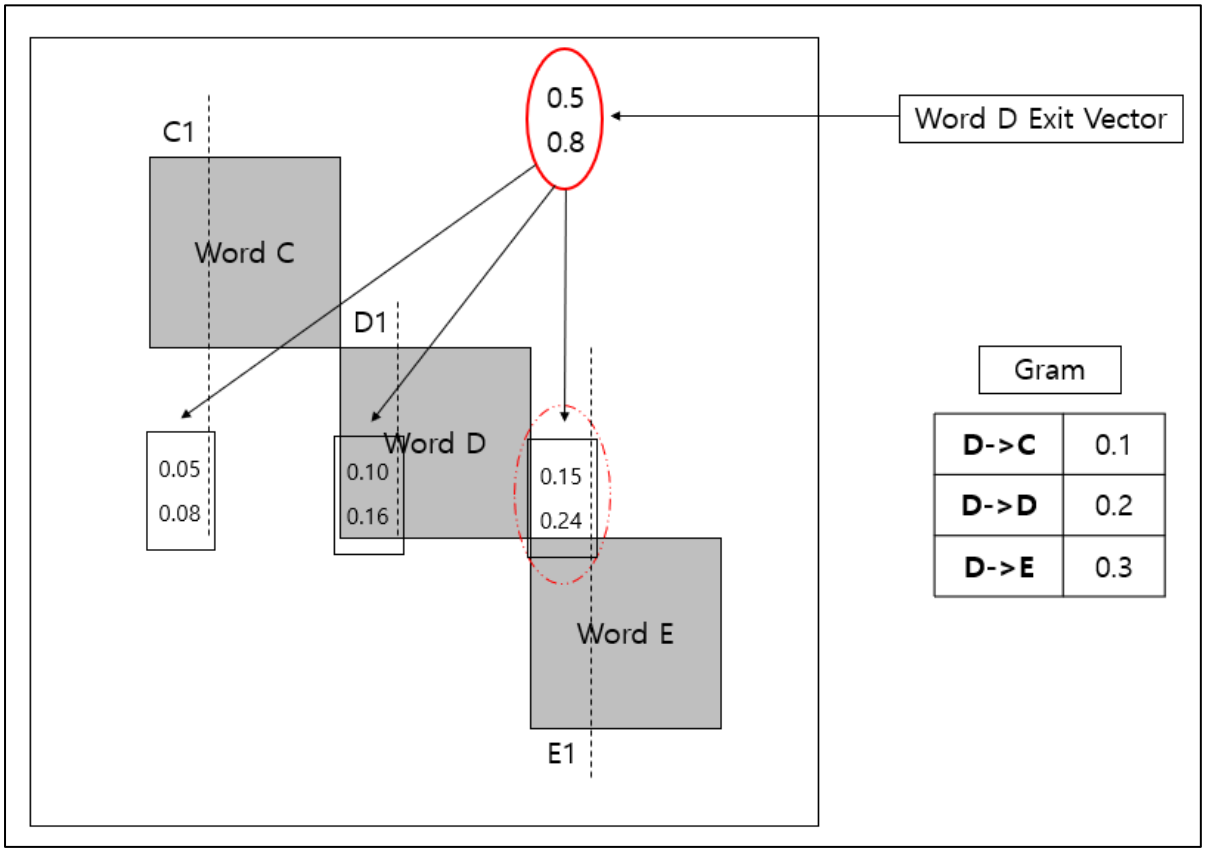
      - word_matrix가 정사각행렬이어야 한다.
      - m[0][1] == 1: word_matrix의 entry가 state 1로만 이루어져야 한다 (연산의 편리를 위한 가정).
      - m[0][-1] == 0: word_matrix에서 entry하자마자 exit하면 안 된다(Python에서 배열의 Index가 -1이면 마지막 Item을 가리킨다).

   C. word_matrix를 utterance_matrix에 추가한다. word matrix에서 Dummy State(Entry, Exit)을 제외한 Inner Matrix를 대각선으로 이어 붙인다. Entry는 B에서 entry가 state 1로만 이루어진다고 가정했기 때문에 따로 저장하지 않고, Exit은 여러 값으로 나올 수 있으므로 Exit Vector를 따로 list에 저장해 둔다. Initial Entry와 Exit의 처리는 Step 4, 5에서 다룬다.

   D. iteration을 위해 클래스 변수들(matrix size, gaussian list, start index list)을 업데이트한다. 이 과정에서 Gaussian List가 업데이트된다.

E. 다음 word(dictItem)에 대해 A~D를 반복한다.

4. 모든 word의 정보를 반영했으면, Initial Entry 정보를 먼저 업데이트한다. State 0(Initial State)에서는 이전 state나 word에 대한 정보가 없으므로 unigram만을 사용할 수 있다. startIndexList를 참조하여 각 word의 start_index에 대해 m[0][start_index]를 해당 word의 unigram 값으로 세팅한다.



5. 이제 Exit 정보를 업데이트한다. Word에서 Exit하면 다음 Word의 Entry로 연결되어야 한다. Step 3-C에서 저장한 Exit Vector List를 참조하며, exit vector의 각 확률에 다음에 올 word에 대한 Gram 확률을 곱해서 그 word의 entry에 값을 분배해준다. 이때 Gram 값은 Unigram이나 Bigram을 사용할 수 있는데, Unigram의 정확도가 더 높다고 판단되어 Unigram을 사용했다(D->C, D->D, D->E가 아닌, C, D, E의 확률을 계산한다고 생각하면 된다). 과정에 대한 설명은 상단의 그림을 참조하며, Unigram과 Bigram의 테스트에 관한 설명은 파트 7에서 간단하게 다룬다.

여기까지 수행하면 Transition Matrix와 Gaussian List, startIndexList를 가지는 Utterance HMM이 구성된다. 제공된 데이터 파일에 대해 계산시, 127 x 127 사이즈의 Matrix가 계산된다.

6. Viterbi Algorithm 구현 및 Test 메소드 구성 (recognizer.py)

Viterbi Algorithm for HMM

- Input: $e_1, \cdots, e_t, P(s_j) \equiv P(X_0 = j), T_{ij} \equiv P(X_t = j | X_{t-1} = i), P(e_k|s_j) \equiv P(e_k|X_k = j)$

1. **for** each state value j
2. $m_1(j) \leftarrow P(s_j) \times P(e_1|s_j)$
3. $m'_1(j) \leftarrow 0$
4. **end**
5. **for** each time k from 2
6. **for** each state value j
7. $m_k(j) \leftarrow \max_i m_{k-1}(i) T_{ij} P(e_k|s_j)$
8. $m'_k(j) \leftarrow \arg\max_i m_{k-1}(i) T_{ij}$
9. **end**
10. **end**
11. $q_t = \arg\max_i m_t(i)$
12. **for** each time k from $t-1$ to 1
13. $q_k = m'_k(q_{k+1})$
14. **end**

- Output: $q_1, \cdots, q_t$

이제 알고리즘에서 사용할 모델 객체를 모두 생성했으니 Viterbi 알고리즘으로 인식하는 부분을 구현하면 된다. 강의자료의 알고리즘(상단 이미지)를 토대로 구현하였고, 이 부분에서는 구현 이슈 및 수정점만 다룬다.

```
# probability function
plog_initial = lambda index: np.log(utter.matrix[0][index])
plog_trans = lambda index_from, index_to: np.log(utter.matrix[index_from][index_to])

def plog_evidence(index, vec):
    g_value = np.log(utter.gaussianList[index].probability(vec))
    if g_value == -np.inf:
        return -np.finfo(float).max
    else:
        return language_model_weight * g_value
```

- Gaussian을 통해 계산한 Observation Probability가 매우 낮아 프로그램에서 제대로 인식하는 것이 어렵기 때문에 전체적인 확률 연산에 log를 씌워서 계산한다. 알고리즘의 반환값은 argmax의 list이기 때문에 전체 확률 연산에 log를 씌운다 하여도 출력 값이 바뀌지 않는다.

- Observation Probability는 정규분포의 값이기 때문에 0이 나올 수 없다. 따라서 0으로 계산될 경우 0은 아니지만 아주 작은 값을 반환해야 한다. 이 함수에서는 log를 씌워 연산하기 때문에 -inf는 아니지만 가장 작은 값을 반환하도록 하였다.

- Word Transition Penalty를 abstraction한 Language Model Weight를 부여한다. 해당 값으로는 몇 번의 테스트를 거쳐 0.2를 결정하여 사용하였다. 테스트 과정은 파트 7에서 간단하게 다룬다.

- Silence Start와 Silence End를 가정한다. 즉, q_1는 silence의 start index인 1이고, q_t는 silence의 end index인 3이다. General하게 설정되지 않은 변수이기 때문에 Dictionary가 바뀔 경우 이 부분도 바꾸어 주어야 한다.

- Python의 성능이 좋지 않기 때문에 수행 시간을 조금이라도 줄이기 위해서 원래의 viterbi 함수를 수정하여 optimizedViterbi 함수를 작성하였다. 기존의 viterbi 함수는 수업자료에 주어진 수도코드를 그대로 옮겨 놓는 것을 목적으로 하였다. 최적화 함수에서는 Dot(.) 연산을 최소화하고, 반복 연산을 최소화하기 위해 메모리에 값들을 미리 올려 놓는 등의 수정을 하였다. 기존의 함수는 파일을 하나 연산하는 시간이 평균 58.76초였는데, 최적화를 한번 거친 optimizedViterbi 함수의 연산 시간은 평균 14.71초로 크게 향상된 것을 볼 수 있었다. 함수의 의미가 바뀌지는 않는다.

```
        (1228/1242) [   17408s] Start: ./prevData/tst/m/sw/7o585o8.txt
        (1229/1242) [   17420s] Start: ./prevData/tst/m/sw/8989615.txt
        (1230/1242) [   17433s] Start: ./prevData/tst/m/sw/918753o.txt
        (1231/1242) [   17446s] Start: ./prevData/tst/m/sw/9553936.txt
        (1232/1242) [   17460s] Start: ./prevData/tst/m/tc/1474568.txt
        (1233/1242) [   17473s] Start: ./prevData/tst/m/tc/1743z44.txt
        (1234/1242) [   17486s] Start: ./prevData/tst/m/tc/2521o7o.txt
        (1235/1242) [   17496s] Start: ./prevData/tst/m/tc/265532o.txt
        (1236/1242) [   17508s] Start: ./prevData/tst/m/tc/2895175.txt
        (1237/1242) [   17519s] Start: ./prevData/tst/m/tc/4z3667z.txt
        (1238/1242) [   17535s] Start: ./prevData/tst/m/tc/5233199.txt
        (1239/1242) [   17549s] Start: ./prevData/tst/m/tc/634o487.txt
        (1240/1242) [   17562s] Start: ./prevData/tst/m/tc/69118z2.txt
        (1241/1242) [   17575s] Start: ./prevData/tst/m/tc/8221367.txt
        (1242/1242) [   17588s] Start: ./prevData/tst/m/tc/ooo4611.txt
FINISH Recognize! - [   17601s]

Process finished with exit code 0
```

테스트 함수에서는 파트 2에서 만든 testParser와 위에서 만든 viterbi Algorithm을 사용하여 데이터를 읽어 들인 후, 인식 결과를 recognized.txt로 내보낸다.

7. HyperParameter 테스트 (hyperParameterTest/*)

총 두 가지의 HyperParameter를 조정하였다.

- Unigram / Bigram: Exit후 다음 Entry의 Connection에 사용할 Language Model
- Language Model Weight: Language Model의 가중치

전체 데이터를 테스트하는 데에는 시간이 너무 오래 걸려서(5시간 가량), 앞 쪽 100개의 데이터를 가지고 테스트를 진행하였다.

Language Model Weight의 경우 기본값(1.0), 0.2, 0.1을 테스트하였으며, 이 파트에서 그 결과를 분석하지는 않는다. 결과적으로 Unigram과 0.2를 선택하였다.

```
Unigram / Language Model Weight - 0.2
====================== HTK Results Analysis =======================
  Date: Wed Jun 10 09:49:07 2020
  Ref : reference100.txt
  Rec : recognized100_uni_02.txt
------------------------ Overall Results --------------------------
SENT: %Correct=89.00 [H=89, S=11, N=100]
WORD: %Corr=99.14, Acc=98.14 [H=694, D=2, S=4, I=7, N=700]
------------------------ Confusion Matrix -------------------------
       z   o   o   t   t   f   f   s   s   e   n
       e   h   n   w   h   o   i   i   e   i   i
       r       e   o   r   u   v   x   v   g   n
       o               e   r   e       e   h   e
                       e               n   t       Del [ %c / %e]
zero  65   0   0   0   0   0   0   0   0   0   0     0
  oh   0  46   0   0   0   0   0   0   0   0   0     1
 one   0   0  75   0   0   0   0   0   0   0   0     0
 two   0   0   0  58   1   0   0   0   0   0   0     0 [98.3/0.1]
thre   0   0   0   0  76   0   0   0   0   0   0     0
four   0   0   0   0   0  61   1   0   0   0   0     0 [98.4/0.1]
five   0   0   0   0   0   0  59   0   0   0   0     0
 six   0   0   0   0   0   0   0  66   0   0   0     0
seve   0   0   0   0   0   0   0   0  65   0   0     0
eigh   0   1   0   0   0   0   0   0   0  63   1     1 [96.9/0.3]
nine   0   0   0   0   0   0   0   0   0   0  60     0
Ins    0   6   0   0   0   0   0   0   0   1   0
===================================================================
```

8. 결과 데이터 분석

```
====================== HTK Results Analysis =======================
  Date: Wed Jun 10 12:48:07 2020
  Ref : reference.txt
  Rec : recognized.txt
------------------------ Overall Results --------------------------
SENT: %Correct=79.69 [H=989, S=252, N=1241]
WORD: %Corr=97.84, Acc=96.59 [H=8499, D=76, S=112, I=108, N=8687]
------------------------ Confusion Matrix -------------------------
       z   o   o   t   t   f   f   s   s   e   n
       e   h   n   w   h   o   i   i   e   i   i
       r       e   o   r   u   v   x   v   g   n
       o               e   r   e       e   h   e
                       e               n   t       Del [ %c / %e]
zero  805   2   0   4   1   0   0   0   0   0   0    3 [99.1/0.1]
 oh     0 702   1   0   0   7   3   0   1   0   3   32 [97.9/0.2]
 one    0   3 792   0   0   1   1   0   0   0  11    2 [98.0/0.2]
 two    0   5   0 786   4   0   0   1   0   0   1    6 [98.6/0.1]
thre    0   0   0   0 814   0   0   0   0   0   0    0
four    0   4   1   0   0 769   7   0   0   0   0    2 [98.5/0.1]
five    0   4   0   0   0   0 779   0   0   0   0    0 [99.5/0.0]
 six    0   0   0   0   0   0   0 799   2   0   0    0 [99.8/0.0]
seve    1   2   1   0   0   0   0   0 787   0   0    0 [99.5/0.0]
eigh    0   9   3   3  11   0   2   0   0 764   3   31 [96.1/0.4]
nine    0   4   5   0   0   0   1   0   0   0 702    0 [98.6/0.1]
Ins     0  58   0  16   1   0   0   0   0  32   1
===================================================================
```

파트 7에서 결정한 HyperParameter를 설정한 후 main.py를 실행하여 전체 데이터에 대한 인식을 진행한 결과 다음과 같은 결과가 나왔다. 전체적으로 oh와 eight에 관한 오류율이 높다.

```
two
two
zero
three
seven
three
.
"tst/m/ip/5724o29.rec"
.
"tst/m/ip/7789719.rec"
seven
seven
eight
nine
seven
one
```

전체 데이터가 1242개인데 N=1241로, 1개의 데이터가 제대로 인식되지 못했다. recognized.txt를 찾아 본 결과, 우측의 이미지와 같이 1006번 index인 "m/ip/5724o29.txt"에 대해 제대로 인식을 하지 못한 것을 알 수 있다. 해당 데이터 파일에 대해 디버깅을 진행했다.

```
[0, 1, 3]
FINISH Recognize! - [      10s]
```

그 결과, 우측 이미지와 같이 문장 전체가 Silence로 인식되는 것을 확인할 수 있었다. 따라서 전체 결과에서 Deletion 오류가 7개 더 발생했다는 것을 알 수 있다.