1. For this question, it is recommended that you use the GRM library and FSM or OpenFst libraries. In fact, try as much as possible to use the utilities of these libraries to answer the questions. However, you need to justify your responses and not just mention the library utilities used.

(a) Download the following training corpus $S$ and test corpus $\hat{S}$:
http://www.cs.nyu.edu/~mohri/asr10/train.txt

(b) Extract the vocabulary $\Sigma_1$ of $S$ and define a start and end symbol.

(c) Create the following language models:
- bigram back-off model;
- trigram back-off model;

These can be created as indicated in the lecture slides using the utilities grmmake and grmconvert.

Report for each of the weighted automata obtained
- the number of states;
- the number of transitions;
- the number of $\epsilon$-transitions;
- the number of $n$-grams found ($n = 2$ for bigram models, $n = 3$ for trigram models).

For these questions, you can use the utility fsminfo of the FSM library. You should however explain how you determine the number of $n$-grams.

The number of states, transitions, and $\epsilon$-transitions are obtained directly using fsminfo. For the bigram models, by construction, the number of unigrams is the number of states minus one (back-off state), minus the initial and final state if one wishes to exclude the start and end symbols. By construction, the number of bigrams is the number of non-$\epsilon$-transitions minus the number of non-$\epsilon$-transitions leaving the back-off state since each non-$\epsilon$-transition labeled with $b$ leaving a non-backoff state $a$ precisely corresponds to the occurrence of bigram $ab$. 

Now, there is exactly one non-$\epsilon$-transition from the back-off state to all states except to the back-off state itself and the initial state. Thus,

$$N(\text{bigrams}) = N(\text{transitions}) - N(\epsilon-\text{transitions}) - N(\text{states}) + 2.$$  

This can be also re-written as

$$N(\text{bigrams}) = N(\text{transitions}) - 2N(\text{states}) + 4.$$  

The number of trigrams can be obtained in a similar fashion from the trigram language model as

$$N(\text{trigrams}) = N(\text{transitions}) - 2N(\text{states}) + 4 - N(\text{bigrams}).$$

If you constructed the bigram model with:

```bash
$ grmcount -n 2 -s 1 -f 2 train.far \
   | grmmake > train.2.lm.fsm
```

you should have:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of states</td>
<td>44692</td>
</tr>
<tr>
<td># of arcs</td>
<td>487160</td>
</tr>
<tr>
<td># of eps</td>
<td>44690</td>
</tr>
<tr>
<td># of bigrams</td>
<td>397780</td>
</tr>
</tbody>
</table>

Similarly for the trigram model constructed with:

```bash
$ grmcount -n 3 -s 1 -f 2 train.far \
   | grmmake > train.3.lm.fsm
```

you should have:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of states</td>
<td>435987</td>
</tr>
<tr>
<td># of arcs</td>
<td>1707629</td>
</tr>
<tr>
<td># of eps</td>
<td>435985</td>
</tr>
<tr>
<td># of trigrams</td>
<td>437879</td>
</tr>
</tbody>
</table>

(d) Randomly generate 100 sequences from the first model and compare the likelihood given by the two models to the sample formed by these sentences.

It is not hard to generate random sentences from a model using `fsmrandgen`. While this may be subjective and depend on the sentences you have generated, in general the trigram models should be closer to the sentences that served for training and thus could appear closer to English than the bigram models.
(e) Compute the perplexity of these models using the test corpus.

If the $n$-gram language model is represented as a weighted automaton $A$ over the log semiring, then, by definition, the negative log of the probability of the text is obtained by computing the $\oplus_{\log}$-sum of the weights of all paths of $A \circ B$, where $B$ is a deterministic automaton representing the text. Thus composition followed by the application of a shortest-distance algorithm (over the log semiring) yields the result. This can be used to compute the perplexity of the model. The shortest-distance can be obtained using the utilities `fsmpush` or `fsmpotentials`. The automaton $B$ can be a long linear chain. Instead, one can use a transducer $B'$ union of the sentences mapping each sentence to each rank in the text. $A \circ B'$ followed by determinization gives the negative log probability of each sentence. The models created in (c) are defined over the tropical semiring, not the log semiring. Also, the negative log probability of each sentence is computed separately due to the presence of start and end symbols. A standard shortest-distance (over the tropical semiring) can be used to compute the negative log probability of each sentence from $A \circ X_s$, where $X_s$ is an automaton representing sentence $s$. The sum over all sentences $s$ can be used to compute the perplexity.

A (not very fast) way of computing the perplexity on the test set is:

```
cat test.far | \ 
  farfilter "fsmcompose lm.fsm - | 
  fsmrepsilons | 
  fsmdeterminize | 
  fsmpush -c -f" | \ 
  farprintstrings -c -i labels \ 
  awk '{ tot_words += NF - 2; 
    h += $NF / log(2.0) } 
  END { print 2.0 ^ (h / tot_words) }'
```

For the bigram model the perplexity should be around 173, for the trigram it should be around 102.

(f) Shrink both of these models with the option $-s4$. What are the perplexity estimates for these models.

The models can shrinken using `grmshrink`. The perplexities can be computed as in the previous question. They are expected to be higher, around 223 for the bigram model, and 175 for the trigram model.

2. Class-based model
(a) First, for information purposes, we obtain the ten most frequent bigrams, for which we can use the srilm library to output the bigrams and their counts, and then sort them with the following command:

```
ngram-count --text ../train.txt  --order 2  --write BigramCounts.txt  
  2 --write BigramCounts.txt | sort -k2 -t $'\t' -n ...
```

This gives us the following most frequent bigrams and their counts (in descending order):

<table>
<thead>
<tr>
<th>Bigram</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>cts vs</td>
<td>2675</td>
</tr>
<tr>
<td>for the</td>
<td>2676</td>
</tr>
<tr>
<td>mln vs</td>
<td>3206</td>
</tr>
<tr>
<td>said the</td>
<td>3579</td>
</tr>
<tr>
<td>said it</td>
<td>4062</td>
</tr>
<tr>
<td>mln dlrs</td>
<td>4497</td>
</tr>
<tr>
<td>in the</td>
<td>6175</td>
</tr>
<tr>
<td>of the</td>
<td>6779</td>
</tr>
<tr>
<td>said &lt;/s&gt;</td>
<td>7782</td>
</tr>
<tr>
<td>&lt;/s&gt; the</td>
<td>10893</td>
</tr>
</tbody>
</table>

Note that the Mutual Information can be approximated as:

\[
I(w_1,w_2) \approx \log_2 \frac{N \cdot c(w_1w_2)}{c(w_1)P(w_2)}
\]

where \( N = |V| \) is the corpus size. Using this formula, we can obtain the 20 elements with the highest PMI with the following simple python implementation, which uses the unigram and bigram counts obtained with the SRILM library.

```python
BiCounter=Counter()
UniCounter = Counter()
MutualInfo=Counter()
with open('BigramCounts.txt','r') as f:
    reader=csv.reader(f,delimiter='\t')
    for row in reader:
        bigram = row[0]
        BiCounter[bigram]=float(row[1])
with open('UnigramCounts.txt','r') as f:
    reader=csv.reader(f,delimiter='\t')
    for row in reader:
        UniCounter[row[0]]=float(row[1])
```

4
The results are shown in Table (1).

<table>
<thead>
<tr>
<th>$w_1w_2$</th>
<th>$I(w_1, w_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>blankets soap</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>rwevs 61805000</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>harlan ullman</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>parra gil</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>andrzej doroscz</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>mochtar kusumaatmadja</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>heron cay</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>unexplored moere</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>fiberglass architectural</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>brewers castlemaine</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>rm nicel</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>beau bolter</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>robbie mupawose</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>43124 42476</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>hissene habre</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>606352 659271</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>zheng tuobin</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>11895 228999</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>ramar intercapital</td>
<td>20.4852013649</td>
</tr>
<tr>
<td>culpable homicide</td>
<td>20.4852013649</td>
</tr>
</tbody>
</table>

It is no surprise that all of these bigrams have the same MI, since for a
pair of words that only appear once each and they appear as a bigram, then

\[ I(w_1, w_2) \approx \log_2 \frac{P(w_1 w_2)}{P(w_1) P(w_2)} = \log |V|^{-1} = 20.4852 \]

It is clear what the mutual information is measuring. Bigrams that frequently appear together have high values of MI, and thus could be considered as a block phrase in the language model.

(b) Note that it doesn’t make much sense to define classes for pairs of numbers (of which the corpus has many examples), since they naturally appear only once due to their individual low probability of occurring and thus manage to get high values of Mutual Information. Filtering these cases, we now pick the 2000 bigrams with highest MI in python, and then write a text file dictionary with them.

With python, we can very easily create the classes and write the text files in the format required to define a transducer for the fsm library. We use

```python
MostCommon2000 = MutualInfoNoNum.most_common(2000)
nonMappedUnigrams = set(UniCounter)
Dict = dict([jj[0], jj[1]] for jj in MostCommon2000)
f = open("classes.txt","w")
f2 = open("ClassesCorpus.txt","w")
label = 44693  #Starting label of symbols file
for key in Dict.keys():
    unigrams = key.split()
    klass = unigrams[0] + "\n" + unigrams[1]
f.write("0 0 " + unigrams[0] + " +" + klass + "\n")
f.write("0 0 " + unigrams[1] + " +" + klass + "\n")
f2.write(klass + " +str(label) + "\n")
label += 1
nonMappedUnigrams = nonMappedUnigrams - ... 
    set(unigrams[0]) - set(unigrams[1])
f.close()
f2.close()
f = open("classes.txt","a")
for unigram in nonMappedUnigrams:
    f.write("0 0 " + unigram + " +unigram + "\n")
f.write("1")
f.close()
```

Then, using these files, we can create the transducer that maps into the classes with
and then, to create the LM, we first read the training sentences, compose them with the Class Mapper transducer and project them into the output labels. Thus, we have now a far file with the edited sentences.

Finally, we can build the language model with the classes as follows

The construction of the class-bases trigram model is analogous.

(c) Finally, we evaluate the class-based model by computing its perplexity. In order to do this, we need to preprocess the test sentences, by composing them with the ClassMapper as before. Then, we compute the perplexity just as we did in part 5. The results for these class-based models are:

<table>
<thead>
<tr>
<th>Model</th>
<th>Perplexity (w &lt; /s &gt;)</th>
<th>Perplexity (w/o &lt; /s &gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigram</td>
<td>156.432</td>
<td>218.456</td>
</tr>
<tr>
<td>Trigram</td>
<td>95.329</td>
<td>118.718</td>
</tr>
</tbody>
</table>

As we can see, grouping bigrams with large mutual information into classes helped to significantly improve the perplexity in all cases.

3. Maxent Models

After an intricate and complicated installation of the packages liblbfgs, srilm, and its extension for maxent models, we train a Maxent model with bigram features with the following code
The output is the following:

```plaintext
Iteration 99
  No of NaNs in logZs: 0, No infs: 0
dual is 4.75908
  regularized dual is 4.96105
  norm of gradient =0.00805258
  norm of regularized gradient =0.00804419
No of NaNs in logZs: 0, No infs: 0
dual is 4.75903
  regularized dual is 4.96081
  norm of gradient =0.00368955
  norm of regularized gradient =0.00367062
Iteration 100
  OWL-BFGS terminated with the stopping criterion
Duration: 11 seconds
```

From here we can see that the LGFBS optimization method has met a maximum iteration (100) criterion.

Similarly, we build the maxent model for trigram features.

```plaintext
ngram-count --text ../train.txt --maxent-lm ...
  MaxEntTrigram --order 3
```

The process, which takes significantly more time to run, now actually reaches convergence, although close to the 100 iteration limit.

```plaintext
Iteration 97
  No of NaNs in logZs: 0, No infs: 0
dual is 3.27721
  regularized dual is 3.70278
  norm of gradient =0.000727621
  norm of regularized gradient =0.000451379
Iteration 98
  OWL-BFGS resulted in convergence
Duration: 68 seconds
```

Now, we compute perplexities on the same test set as for part A. The code to implement this is

```plaintext
ngram --maxent --lm MaxEntBigram --ppl ../test.txt ...
  --debug 2
```
The result for the bigram model is the following:

```
15000 sentences, 335409 words, 5257 OOVs
0 zeroprobs, logprob= -786593 ppl= 190.097 ppl1= 241.278
```

As we did in the previous section, `ngram` reports the perplexity in the sentences with stopping signs ($ppl$) and without them ($ppl1$). On the other hand, for the trigram model we have

```
15000 sentences, 335409 words, 5257 OOVs
0 zeroprobs, logprob= -717485 ppl= 119.881 ppl1= 149.004
```

For completion, we now try another possible model that the `maxent` patch allows to create. This time, we create an interpolated mixture of the `maxent` bigram and trigram models.

```
ngram -maxent -lm MaxEntTrigram -mix-maxent -mix-lm ...
MaxEntBigram -ppl ../test.txt -bayes 0
```

The results, however, show that this mixture model does not perform as well as the pure trigram model:

```
15000 sentences, 335409 words, 5257 OOVs
0 zeroprobs, logprob= -720797 ppl= 122.559 ppl1= 152.485
```

Comparing to the results obtained in the first part, we see that the perplexities for both methods are very similar, with a slight but constant advantage for the `ngram` back-off models. In terms of efficiency, it seems like computing perplexities for the `maxent` model is faster than for the usual `ngrams`, but not faster than for the pruned `ngram` models. It must be noted also that the method LFBGS is known converge very fast, so the core of the `maxent` is already close to top-of-the-art in terms of efficiency.