David Elkind 306358425 Assaf Lehr 035885334

1 Descriptive stats

1) unigram segments instances count= 127884 (train) 11282 (gold)

2) unigram segments unique types= 15986 (train) 3171 (gold)

3) seg-tag pairs instance count= 127884 (train) 11282 (gold)

4) seg-tag pairs unique count= 18143 (train) 3424 (gold)

5) ambiguousness = average different tag types per segment=1.135 (train) 1.08 (gold)

6) discussion: The ambiguity is low on average, this means that a lot of segments have only one tag, and that a basic tagger (choose one of the seen tags, or choose the most frequent tag) will be quite accurate even if we ignore grammer information, as long as we saw that word before.

The gold file is smaller (10% of sentences) of the train file, and the unigram/tags instance count match this ratio.
The number of unique segments does not grow linearly which the number of instances. This matches the (external) knowledge on distribution of words in corpus. As the number of unique words grows slowly with the data, and our train corpus is not huge, there still be large numbers of words which are different between the corpuses (this claim requires looking at different size of corpuses to be certain, like in question 5,c)
Same idea about seg-tag pairs.

Double-click (or enter) to edit

2 Basic Tagger

(1) Parameter scheme: We calculate here the conditional probability of a tag given segment, thne apply ln on it: $\ln P(t_i|w_j)$ for i in 1..|tags| and j in 1..|words| and save only the most frequest:
$argmax_i P(t_i|w_j) = argmax_i \ln P(t_i|w_j)$ for i in 1..|tags| and j in 1..|words|
ln is applied for code reuse, as argmax result is the same.

parameter file structure (see train.train_word_prob): Format:

```
SEG POS1 logprob1 ...
SEG POS1 logprob1 ...
...
```

Note that although the format is similar to that of the lex file, its meaning is different.

(2) This is per tag,word (ignoring sequences)
for i in 1..|tags| and j in 1..|words| $\ln P(t_i|w_j) = \ln Count(t_i, w_j) - \ln Count(w_j)$
This expression is adaptaed from $P(t_i|w_j) = Count(t_i, w_j)/Count(w_j)$, by adding $\ln$ on both sides
Note that afterwards we apply argmax as in (1)

(3) Runtime complexity (for sentence of length N, with T unique categroies in corpus. For multiple sentences, assume N length of the flattened concat of them)
Train: $O(N * T)$: we tokenize by scanning the sentence once, then count t,w pairs and w , both are O(n) as we use hash-table/arrays with O(1) access. Then we choose max of the categories and save to file only that result. This is done in the worst case in O(T) on the number of categies
Decode: $O(N)$: get the saved value for that category from file.

(4) see results/1/heb-pos.test.eval macro-avg 0.8303 0.1080
1st value is sentence accuracy (proportion of correct tags in the corpus)
2nd is word accuracy (proportion of correct tags in the corpus)

3 First Order Tagger

1. target function : $t_1^n* = argmax_{t_1^n} \sum_{i=1}^{n} \ln P(w_i|t_i) + \ln P(t_i|t_{i-1})$

(It was changed from the book target $t_1^n* = argmax_{t_1^n} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1})$ due to converting to ln)

1. parameters equations:
   $\ln \hat{P}(w_i|t_i)$ - This is the ln of emission prob. and filled in the .lex file
   $\ln \hat{P}(t_i|t_{i-1})$ - These are the ln transitions prob. which are part of the .gram file
2. evaluator equations:
   $\ln \hat{P}(w_i|t_i) = \ln Count(w_i, t_i) - \ln Count(t_i)$
   $\ln \hat{P}(t_i|t_{i-1}) = \ln Count(t_{i-1}, t_i) - \ln Count(t_{i-1})$

1. Runtime complexity of train . for sentence of length N with T unique categroies in corpus (for corupus of multiple sentences, treat N as long sentence of their concatination)
   $O(N)$: we scan each the sentences and save the various counts for emissions into a table.
   In practice we implemented it with two numpy dense tables, for emissions NxT size and for transition into TxT , thus it is in our implementation $O(N * T)$
2. Runtime complexity Decode step:
   for single sentence: $O(N * T^2)$ in vitorbi we use dynamic-programming on a table with T*L cells. For each cell calculation, we scan T of the previous cells.
3. results: macro-avg 0.9035 0.2640 (see question 2 for meaning)
4. we implemented smooting for unkown words, based on Add-δ smooting with uniform prior, see train.py for concrete implementation and details.
   This provided significant improvment: macro avg 0.8884 0.2220.
   With addition of adaptation due to domain knowledge, we got 0.901 0.256. example to this rule is setting emission of P(UNK|'H') to 0, as we expect only 'H' segment to be emitted from 'H' tag
5. impelmenting smoothing for unkown transition with the same smoothing technique did not significally imporve the results (cherry-picking the smoothing parameters did not imporve by more than 0.05%). This is probably due to reasnable coverage of the state transitions.

4 process results

1. confusion matrix and frequent erros : m_{ij} a word was tagged m_i instead of m_j:
   - (('JJ', 'POS'), 1)
   - (('VB', 'NNP'), 1)
   - (('NN', 'NNP'), 1)
2. manual tagging, results are at results/isha/ , (manual) gold file at exps/isha/ accuracy: 0.8182 (words) sentence is ofcourse 0.

   ```
   AIFH    NN   correct
   NELH    POS   wrong! tagged manually as JJ
   NELH    NNP    wrong! tagged manually as VB
   NELH    NNP   wrong! tagged manually as NN
   yyCM    yyCM correct
   NELA    VB   correct
   AT    AT  correct
   H    H  correct
   DLT    NN  correct
   BPNI    IN  correct
   BELH    NN  correct
   ```

   potential problems in the model :
   - NELH did not appear in the training and does not have any emissions, thus only grammer (transitions) was used here.
     More data may have fixed this (we might have seen this word with at least few of it's meanings/tags,
     and/or morphological prediction on unkown words, would have helped to better estimate the emissions(at least for NEL/H/W meanings)
   - (to less effect) The model chose 'H' tag which is a very unlikely, as P('H'|'H') is close to 1. In other words, it would have been better if the smoothing result would have make it unlikely to be in 'closed groups' where the set of known words is small.

Note that even after solving this, we might still have multiple equally likely transitions, so this imporvment will not be as strong as the previous one.

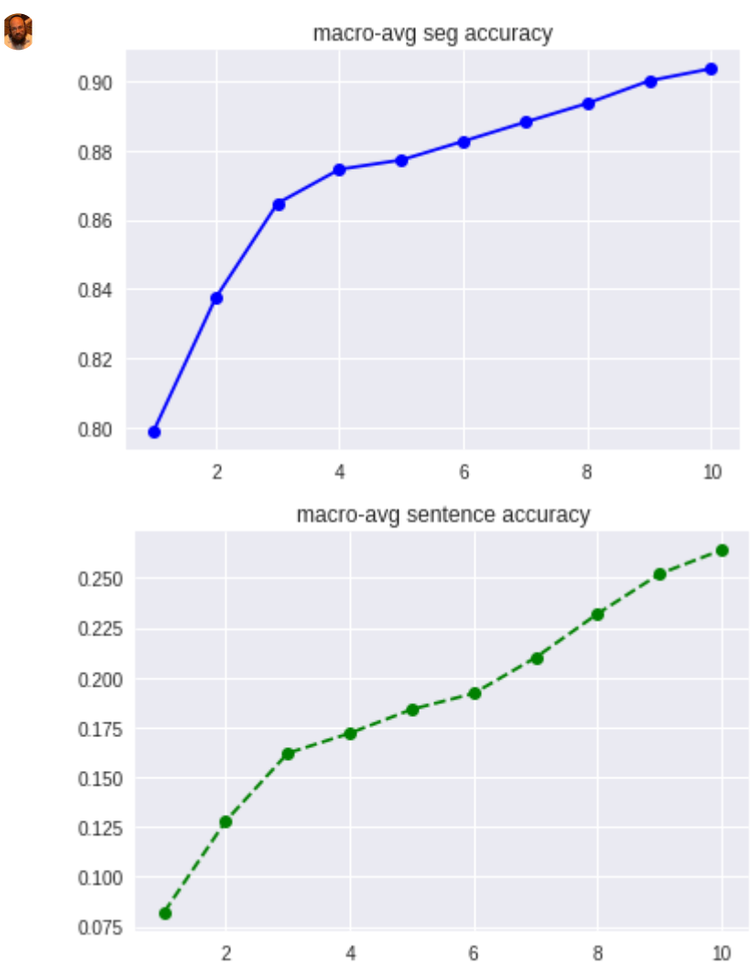
4,3 training on part of the data.

Please see the graphs below.
As we enlarge the train corpus size, the accuracy imporves, and it is clear we are far from saturating the model training ability (At least a 2-fold increase can still increase accuracy without saturation, probably more will be possible)
I also plotted the results with log2 of the X axis, as in terms of unique-words, it is reasonable to believe that the growth is dependent of log of the corpus size (although more data is needed for this conclusion. This graph is too small)

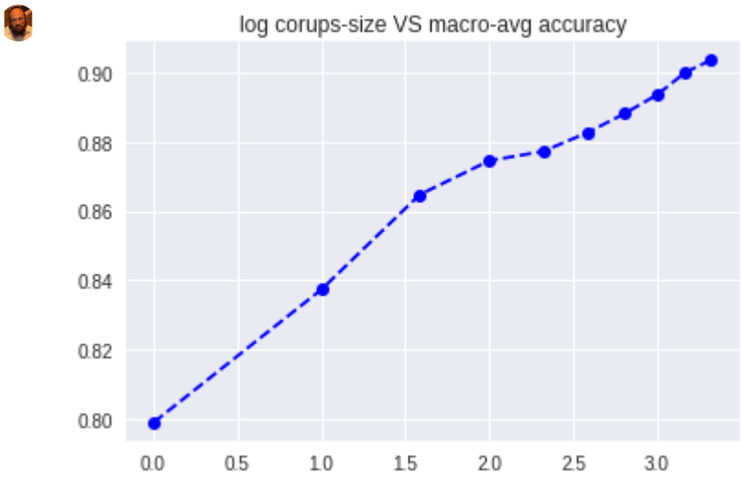
```
%matplotlib inline

import matplotlib
import numpy as np
import matplotlib.pyplot as plt

X = np.arange(1,11)
Y=[(0.7991, 0.0820),  #1
(0.8374,0.1280),  #2
(0.8647, 0.1620),
(0.8745, 0.1720),
(0.8772, 0.1840),
(0.8826, 0.1920), # 6
(0.8881, 0.2100),
(0.8935, 0.2320),
(0.9000, 0.2520),
(0.9035, 0.2640)] # 10
Y = np.array(Y)
plt.title('macro-avg seg accuracy')
plt.plot(X,Y[:,0],'bo-')
plt.show()
plt.title('macro-avg sentence accuracy')
plt.plot(X,Y[:,1],'go--')
plt.show()
```




```
plt.title('log corups-size VS macro-avg accuracy')
plt.plot(np.log2(X),Y[:,0],'bo--')
plt.show()
```



5 Bonus question: Random-order tagger

1. The parameters semantics does not change - we are still looking at emission (.lex) and transition (.gram) probabilities. However, since we are dealing with the higher order model, instead of bigrams we are dealing with trigrams, 4-grams etc., depending on the model order.
2. The same Viterbi algorithm is used for tagging. However, the number of states grows as each order increment adds additional dimension. The complexity for Viterbi algorithm generalized for arbitrary order is $O(N * T^{order+1})$.
3. The complexity for the transitions training per se was not affected since despite increase in the sliding window size, the number of strides is still the same up to a constant (the number of the start tags is the only thing that is dependent on the order) and this is because training complexity depends on the number of the observations and this does not change. However, since (for simplicity sake) we use arrays, the complexity deteriorates to $O(N * T^{order})$ and also transition probabilities smoothing is performed for each cell in the $|Tags|^{order} \times |Tags|$ matrix, the complexity increases by the factor of $|Tags|$. The emission probabilities training and smoothing does not depend on the order so it is not affected.
4. See (2) for decoding complexity analysis.
5. The running time for order 2 is around 10 minutes and the performance actually deteriorates. Our hypothesis is that this happens since Hebrew is relatively flexible in terms of grammar so the additional information provided by longer history is outweighed by data sparsity which fails to be adequately addressed due to a subpar transition smoothing strategy. On the "isha" part we actually see an improvement (macro-avg 0.8182 for trigrams vs 0.7273 for bigrams).

6 : High level review of the code

```
src folder:
taggre/decoder - methods to load param files (.lex,.gram), two decoders
majority_vote_decoder and viterbi_decoder
tagger/evaluator - methods to calcualte macro-avg
tagger/utils - method to load .test and .gold files
explore - data exploration for question 1
evaluate/train/decode python wrappers
train - also contains train code to fill .lex,.gram,.param(base decoder) files.

shell scripts to run each step by itself:
  train
  decode
  evaluate

shell  scripts which run full experiment (train,decode,evaluate and copy files to
destination directory)
  exp_1_n.sh  - base
  exp_2_y.sh  - viterbi
```

```
exp_2_n.sh - viterbi with smoothing
run.10.sh  - run exp_2_n.sh on 10%,20%... of the train data (question 5,c)
```

conclusions

HMM model and unknown-word smooting provide signficant imporvment over the baseline.
The 3 problems mentioned for implementing a tagger for Hebrew compared to English were:

- rich word mophology
- less-restrictive transitions (which means there are more possible transitions than in English)
- datasets are small - the effect of this is clearly shown in the graph at 4,3.

Improvements to the model:

- It is clear that more data will improve the accuracy of the model, via better emission proobabilities. Order of magnitude of increase will probably still won't saturate it (question 4,3 graph)
- Smarter smoothing strategy on emissions probabilities is important. For example, some categroies like pronoun,determiner have a small closed set of known words, while others like proper-noun have unlimited one. Additional tweaking here, should probably improve the results.
- In this work, we had only partial treatment of morphology of a word and treated it as one token. This means we may know a word like NEL, but fail to understand NELW . a classifier for unknown words, based on morphology, could imporve this.
- Higher order tagger does not improve grammar behavior (if not due to a bug then due to data sparsity and inferior smoothing strategy)

Additional possible improvements:

- Increasing Hebrew dataset for part-of-speech tagging is clearly needed. It can be done by manual/semi-manual work (which requires a large budget). Suggesion: If there exists a tagged english corpus with a high quality hebrew translation, it might be possible to write a model which will allign the words and, assuming the same tags are valid for both languages, allow for a larger corpus.