HOMEWORK 7: HIDDEN MARKOV MODELS

10-301/10-601 Introduction to Machine Learning (Spring 2022)
http://www.cs.cmu.edu/~mgormley/courses/10601/
OUT: Friday, April 1st
DUE: Tuesday, April 12th
TAs: Brendon, Mukund, Tara, Shubham, Abu

Summary In this assignment you will implement a new named entity recognition system using Hidden Markov Models. You will begin by going through some multiple choice and short answer warm-up problems to build your intuition for these models and then use that intuition to build your own HMM models.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.9.6) and versions of permitted libraries (e.g. `numpy` 1.21.2 and `scipy` 1.7.1) match those used on Gradescope. You have 10 free Gradescope programming submissions. After 10 submissions, you will begin to lose points from your total programming score. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

7 Programming (80 points)

7.1 The Task

In the programming section you will implement a named entity recognition system using Hidden Markov Models (HMMs). Named entity recognition (NER) is the task of classifying named entities, typically proper nouns, into pre-defined categories, such as person, location, or organization. Consider the example sequence below, where each word is appended with a tab and then its tag:

“	O
Rhinestone	B-ORG
Cowboy	I-ORG
”	O
(	O
Larry	B-PER
Weiss	I-PER
)	O
-	O
3:15	O

`Rhinestone` and `Cowboy` are labeled as an organization (`ORG`), while `Larry` and `Weiss` is labeled as a person (`PER`). Words that aren't named entities are assigned the `O` tag. The `B-` prefix indicates that a word is the beginning of an entity, while the `I-` prefix indicates that the word is inside the entity.

NER is an incredibly important task for a machine to analyze and interpret a body of natural language text. For example, when designing a system that automatically summarizes news articles, it is important to recognize the key subjects in the articles. Another example is designing a trivia bot. If you can quickly extract the named entities from the trivia question, you may be able to more easily query your knowledge base (e.g. type a query into Google) to request information about the answer to the question.

On a technical level, the main task is to implement an algorithm to learn the HMM parameters given the training data and then implement the forward-backward algorithm to perform a smoothing query which we can then use to predict the hidden tags for a sequence of words.

7.2 The Dataset

WikiANN is a “silver standard” dataset that was generated without human labelling. The English Abstract Meaning Representation (AMR) corpus and DBpedia features were used to train an automatic classifier to label Wikipedia articles. These labels were then propagated throughout other Wikipedia articles using the Wikipedia's cross-language links and redirect links. Afterwards, another tagger that self-trains on the existing tagged entities was used to label all other mentions of the same entities, even those with different morphologies (prefixes and suffixes that modify a word in other languages). Finally, the amassed training examples were filtered by “commonness” and “topical relatedness” to pick more relevant training data.

The WikiANN dataset provides labelled entity data for Wikipedia articles in 282 languages. We will be primarily using the English subset, which contains 14,000 training examples and 3,300 test examples, and the French subset, which contains around 7,500 training examples and 300 test examples.

7.3 File Formats

The contents and formatting of each of the files in the handout folder is explained below.

1. **train.txt** This file contains labeled text data that you will use in training your model in the Learning problem (Section 7.4). Specifically, the text contains one word per line that has already been preprocessed, cleaned and tokenized. Every sequence has the following format:

   `<Word0>\t<Tag0>\n<Word1>\t<Tag1>\n ... <WordN>\t<TagN>\n`

   where every `<WordK>\t<TagK>` unit token is separated by a newline. Between each sequence is an empty line. If we have two three-word sequences in our data set, the data will look like so:

   ```
   <Word0>\t<Tag0>\n
   <Word1>\t<Tag1>\n
   <Word2>\t<Tag2>\n
   \n
   <Word0>\t<Tag0>\n
   <Word1>\t<Tag1>\n
   <Word2>\t<Tag2>
   ```

   Note: Word 2 of the second sequence does not end with a newline because it is the end of the data set.

2. **validation.txt**: This file contains labeled validation data that you will use to evaluate your model. This file has the same format as **train.txt**.

3. **index_to_word.txt, index_to_tag.txt**: These files contain a list of all words or tags that appear in the data set. The format is simple:

index_to_word.txt	**index_to_tag.txt**
`<Word0>\n`	`<Tag0>\n`
`<Word1>\n`	`<Tag1>\n`
`<Word2>\n`	`<Tag2>\n`
⋮	⋮

   In your functions, you will convert the string representation of words or tags to indices corresponding to the location of the word or tag in these files. For example, if *Austria* is on line 729 of **index_to_word.txt**, then all appearances of *Austria* in the data sets should be converted to the index 729. This index will also correspond to locations in the parameter matrices. For example, the word *Austria* corresponds to the parameters in column 729 of the matrix stored in **hmmemit.txt**. This will be useful for your forward-backward algorithm implementation (see Section 7.5).

4. **predicted.txt**: This file contains labeled data that you will use to debug your implementation. The labels in this file are generated by running a reference implementation using the features from **train.txt**. This file has the same format as **train.txt**.

5. **metrics.txt**: This file contains the metrics you will compute for the validation data. The first line should contain the average log likelihood, and the second line should contain the prediction accuracy. There should be a single space after the colon preceding the metric value; see the reference output file for more detail.

6. **hmmtrans.txt, hmmemit.txt, hmminit.txt**: These files contain pre-trained model parameters of an HMM that you can use to test your implementation of the Learning and Evaluation and Decoding problems (Sections 7.4, 7.5). The formats of the first two files are the same; each line in these files

consists of a conditional probability distribution. In the case of transition probabilities, this distribution corresponds to the probability of transitioning into another state, given a current state. Similarly, in the case of emission probabilities, this distribution corresponds to the probability of emitting a particular symbol, given a current state. Elements in the same row are separated by a space. Each row corresponds to a line of text, using `\n` to create new lines.

hmmtrans.txt:

```
<ProbS1S1> <ProbS1S2> ...  <ProbS1SN>\n
<ProbS2S1> <ProbS2S2> ...  <ProbS2SN>\n...
```

hmmemit.txt:

```
<ProbS1Word1> <ProbS1Word2> ...  <ProbS1WordN>\n
<ProbS2Word1> <ProbS2Word2> ...  <ProbS2WordN>\n...
```

The format of **hmminit.txt** is similarly defined except that it only contains a single probability distribution over starting states. Therefore, each row only has a single element.

hmminit.txt:

```
<ProbS1>\n
<ProbS2>\n...
```

7.4 Learning

Your first task is to write a program `learnhmm.py` to learn the Hidden Markov Model parameters needed to apply the forward-backward algorithm (See Section 7.5). There are three sets of parameters that you will need to estimate: the initialization probabilities $\boldsymbol{\pi}$, the transition probabilities $\mathbf{B}$, and the emission probabilities $\mathbf{A}$. For this assignment, we model each of these probabilities using a multinomial distribution with parameters $\pi_j = P(Y_1 = s_j)$, $B_{jk} = P(Y_t = s_k \mid Y_{t-1} = s_j)$, and $A_{jk} = P(X_t = k \mid Y_t = s_j)$. These can be estimated using maximum likelihood, which results in the following parameter estimates:

1. $P(Y_1 = s_j) = \pi_j = \frac{N_{Y_1=s_j}+1}{\sum_{p=1}^{J}(N_{Y_1=s_p}+1)}$, where $N_{Y_1=s_j}$ equals the number of times state s_j is associated with the first word of a sentence in the training data set.
2. $P(Y_t = s_k \mid Y_{t-1} = s_j) = B_{jk} = \frac{N_{Y_t=s_k,Y_{t-1}=s_j}+1}{\sum_{p=1}^{J}(N_{Y_t=s_p,Y_{t-1}=s_j}+1)}$, where $N_{Y_t=s_k,Y_{t-1}=s_j}$ is the number of times state s_j is followed by state s_k in the training data set.
3. $P(X_t = k \mid Y_t = s_j) = A_{jk} = \frac{N_{X_t=k,Y_t=s_j}+1}{\sum_{p=1}^{M}(N_{X_t=p,Y_t=s_j}+1)}$, where $N_{X_t=k,Y_t=s_j}$ is the number of times that the state s_j is associated with the word k in the training data set.

Note we add 1 to each count to make a pseudocount. This is slightly different from pure maximum likelihood estimation, but it is useful in improving performance when evaluating unseen cases during evaluation of your validation set.

Your implementation should read in the training data set (**train.txt**), and then estimate $\boldsymbol{\pi}$, $\mathbf{B}$, and $\mathbf{A}$ using the above maximum likelihood solutions with pseudocounts.

Your outputs should be in the same format as **hmminit.txt**, **hmmtrans.txt**, and **hmmemit.txt** (including the same number of decimal places to ensure there are no rounding errors during prediction). The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python3 learnhmm.py [args...]
```

Where `[args...]` is a placeholder for six command-line arguments: `<train_input>` `<index_to_word>` `<index_to_tag>` `<hmminit>` `<hmmemit>` `<hmmtrans>`. These arguments are described below:

1. `<train_input>`: path to the training input `.txt` file (see Section 7.2)
2. `<index_to_word>`: path to the `.txt` file that specifies the dictionary mapping from words to indices. The tags are ordered by index, with the first word having index of 0, the second word having index of 1, etc.
3. `<index_to_tag>`: path to the `.txt` file that specifies the dictionary mapping from tags to indices. The tags are ordered by index, with the first tag having index of 0, the second tag having index of 1, etc.
4. `<hmminit>`: path to output `.txt` file to which the estimated initialization probabilities ($\boldsymbol{\pi}$) will be written. The file output to this path should be in the same format as the handout `hmminit.txt` (see Section 7.2).
5. `<hmmemit>`: path to output `.txt` file to which the emission probabilities ($\mathbf{A}$) will be written. The file output to this path should be in the same format as the handout `hmmemit.txt` (see Section 7.2)
6. `<hmmtrans>`: path to output `.txt` file to which the transition probabilities ($\mathbf{B}$) will be written. The file output to this path should be in the same format as the handout `hmmtrans.txt` (see Section 7.2).

As an example, the following command would run your program on the toy dataset provided in the handout.

```
$ python3 learnhmm.py toy_data/train.txt toy_data/index_to_word.txt \
toy_data/index_to_tag.txt toy_data/hmminit.txt toy_data/hmmemit.txt \
toy_data/hmmtrans.txt
```

After running the command above, the `<hmminit>`, `<hmmemit>`, and `<hmmtrans>` output files should match the reference files provided in the `toy_output` directory.

7.5 Evaluation and Decoding

7.5.1 Forward Backward Algorithm and Minimal Bayes Risk Decoding

Your next task is to implement the forward-backward algorithm. Suppose we have a set of sequence consisting of T words, $x_1, \ldots, x_T$. Each word is associated with a label $Y_t \in \{1, \ldots, J\}$. In the forward-backward algorithm we seek to approximate $P(Y_t \mid x_{1:T})$ up to a multiplication constant. This is done by first breaking $P(Y_t \mid x_{1:T})$ into a "forward" component and a "backward" component as follows:

$$\begin{aligned} P(Y_t = s_j \mid x_{1:T}) &\propto P(Y_t = s_j, x_{t+1:T} \mid x_{1:t}) \\ &\propto P(Y_t = s_j \mid x_{1:t}) P(x_{t+1:T} \mid Y_t = s_j, x_{1:t}) \\ &\propto P(Y_t = s_j \mid x_{1:t}) P(x_{t+1:T} \mid Y_t = s_j) \\ &\propto P(Y_t = s_j, x_{1:t}) P(x_{t+1:T} \mid Y_t = s_j) \end{aligned}$$

where $P(Y_t = s_j \mid x_1, \ldots, x_t)$ and $P(x_{t+1}, \ldots, x_T \mid Y_t = s_j)$ are computed by bottom-up dynamic programming approach.

Forward Algorithm

Define $\alpha_t(s_j) = P(Y_t = s_j, x_{1:t})$. This can be rearranged into the following expression (the full derivation can be found in the lecture notes):

$$\alpha_t(s_j) = A_{jx_t} \sum_k B_{kj} \alpha_{t-1}(k) \tag{3}$$

Using this definition, the α's can be computed using the following dynamic programming procedure:

```
for t = 1,...,T:
    for j = 1,...,k:
        if t == 1:
            α_1(s_j) = π_j * A_{j,x_1}
        else:
            α_t(s_j) = A_{j,x_t} * α_{t-1}(s_k) * B_{k,j}
```

Backward Algorithm

Define $\beta_t(s_j) = P(x_{t+1:T} \mid Y_t = s_j)$. This can be rearranged into the following expression:

$$\beta_t(s_j) = \sum_{k=1}^{J} A_{kx_{t+1}} \beta_{t+1}(s_k) B_{jk} \tag{4}$$

Just like the α's, the β's can also be computed using the following dynamic programming procedure:

```
for t = T,...,1:
    for j = 1,...,k:
        if t == T:
            β_T(s_j) = 1
        else:
            β_t(s_j) = A_{k,x_{t+1}} β_{t+1}(s_k) B_{j,k}
```

Forward-Backward Algorithm As stated above, the goal of the Forward-Backward algorithm is to compute $P(Y_t = s_j \mid x_{1:T})$. This can be done using the following equation:

$$P(Y_t = s_j \mid x_{1:T}) \propto P(Y_t = s_j, x_{1:t}) P(x_{t+1:T} \mid Y_t = s_j)$$

After running your forward and backward passes through the sequence, you are now ready to estimate the conditional probabilities as:

$$P(Y_t \mid x_{1:t}) \propto \alpha_t \circ \beta_t$$

where $\circ$ is the element-wise product.

Minimum Bayes Risk Prediction We will assign tags using the minimum Bayes risk predictor, defined for this problem as follows:

$$\hat{Y}_t = \underset{j \in \{1,\ldots,J\}}{\operatorname{argmax}} P(Y_t = s_j \mid x_{1:T})$$

To resolve ties, select the tag that appears earlier in the `<index_to_tag>` input file.

Computing the Log Likelihood of a Sequence When we compute the log likelihood of a sequence, we are interested in the computing the quantity $\log(P(x_{1:T}))$. We can rewrite this in terms of values we have already computed in the forward-backward algorithm as follows:

$$\begin{aligned} \log P(x_{1:T}) &= \log \big(\sum_j P(x_{1:T}, Y_t = s_j) \big) \\ &= \log \big(\sum_j \alpha_T(s_j) \big) \end{aligned}$$

7.5.2 Implementation Details

You should now write a program `forwardbackward.py` that implements the forward-backward algorithm. The program will read in validation data and the parameter files produced by `learnhmm.py`. The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python3 forwardbackward.py [args...]
```

Where `[args...]` is a placeholder for eight command-line arguments: `<validation_input>` `<index_to_word>` `<index_to_tag>` `<hmminit>` `<hmmemit>` `<hmmtrans>` `<predicted_file>` `<metric_file>`. These arguments are described in detail below:

1. `<validation_input>`: path to the validation input `.txt` file that will be evaluated by your forward backward algorithm (see Section 7.2)
2. `<index_to_word>`: path to the `.txt` file that specifies the dictionary mapping from words to indices. The tags are ordered by index, with the first word having index of 0, the second word having index of 1, etc. This is the same file as was described for `learnhmm.py`.
3. `<index_to_tag>`: path to the `.txt` file that specifies the dictionary mapping from tags to indices. The tags are ordered by index, with the first tag having index of 0, the second tag having index of 1, etc. This is the same file as was described for `learnhmm.py`.
4. `<hmminit>`: path to input `.txt` file which contains the estimated initialization probabilities ($\boldsymbol{\pi}$).
5. `<hmmemit>`: path to input `.txt` file which contains the emission probabilities ($\mathbf{A}$).
6. `<hmmtrans>`: path to input `.txt` file which contains transition probabilities ($\mathbf{B}$).
7. `<predicted_file>`: path to the output `.txt` file to which the predicted tags will be written. The file should be in the same format as the `<validation_input>` file.
8. `<metric_file>`: path to the output `.txt` file to which the metrics will be written.

As an example, the following command would run your program on the toy dataset provided in the handout.

```
$ python3 forwardbackward.py toy_data/validation.txt \
toy_data/index_to_word.txt toy_data/index_to_tag.txt \
toy_data/hmminit.txt toy_data/hmmemit.txt \
toy_data/hmmtrans.txt toy_data/predicted.txt \
toy_data/metrics.txt
```

After running the command above, the `<predicted_file>` output should be:

```
fish    D
eat     C
you     D
```

And the <metric_file> output should be:

```
Average Log-Likelihood: -3.0438629330222424
Accuracy: 0.3333333333333333
```

where average log-likelihood and accuracy are evaluated over the validation set.

Take care that your output has the exact same format as shown above. There should be a single space after the colon preceding the metric value (e.g. a space after `Average Log-Likelihood:`). Each line should be terminated by a Unix line ending `\n`.

7.5.3 Log-Space Arithmetic for Avoiding Underflow

Handling underflow properly is a critical step in implementing an HMM. The most generalized way of handling numerical underflow due to products of small positive numbers (like probabilities) is to calculate everything in log-space, i.e., represent every quantity by their logarithm.

For this homework, using log-space starts with transforming Eq.(3) and Eq.(4) into logarithmic form - you may find the recitation handout helpful. Please use base e (natural log) for logarithm calculation.

After transforming the equations into log form, you may discover calculations of the following type:

$$\log \sum_i \exp\left(v_i\right)$$

This may be programmed as is, but $\exp\left(v_i\right)$ may cause underflow when v_i is large and negative. One way to avoid this is to use the log-sum-exp trick. We provide the pseudocode for this trick in Algorithm 1:

Algorithm 1 Log-Sum-Exp Trick

procedure LOGSUMEXPTRICK($(v_1, v_2, \cdots, v_n)$)
  $m = \max(v_i)$ for $i = \{1, 2, \cdots, n\}$
  return $m + \log(\sum_i \exp(v_i - m))$

Note: The autograder test cases account for numerical underflow using the Log-Sum-Exp Trick. If you do not implement forwardbackward.py with the trick, you might only receive partial credit.

7.6 Gradescope Submission

You should submit your `learnhmm.py`, `forwardbackward.py`, and `utils.py` (if using the functions in the file) to Gradescope. Please do not use other file names. This will cause problems for the autograder to correctly detect and run your code. Please go through the appendix at the end for information on starter-code.

Some additional tips: Make sure to read the autograder output carefully. The autograder for Gradescope prints out some additional information about the tests that it ran. For this programming assignment we've specially designed some buggy implementations that you might implement and will try our best to detect those and give you some more useful feedback in Gradescope's autograder. Make wise use of autograder's output for debugging your code.

Note: For this assignment, you have 10 submissions to Gradescope before the deadline, but only your last submission will be graded.

8 Appendix

8.1 Starter Code

In this handout, we provide you with some starter code to help you get started, primarily in the form of a `utils.py` file, along with more skeleton code in the `learnhmm.py` and `forwardbackward.py` files. The following functions are provided to you in `utils.py` to help with file I/O:

1. `make_dict`: This function takes in a file path and returns a dictionary that maps words or tags to their line numbers (0-indexed).
2. `parse_file`: This function takes in a file (train or otherwise), parses it, and returns two lists of lists, one containing sentences and another containing tags.
3. `write_predictions`: This function takes in a file path, list of sentences, list of predicted tags, and list of true tags, and writes your computed tags to the given file in the desired format. The function also calculates and returns the accuracy.
4. `write_metrics`: This function takes in a file path and relevant metrics, and writes the metrics to the file in the required format.
5. `get_matrices`: This function loads your initialization, emission, and transition matrices.

For detailed documentation on these functions, please read the function descriptions in the file. You are also encouraged to look at the outputs of these functions to further understand what they're doing.