Instructions: This assignment is intended to introduce you to Hidden Markov Models (HMMs) where you will explore more dynamic programming algorithms such as the forward-backward algorithm and viterbi. You will also gain experience with some of the numerical details that need to be addressed when putting together such a system. You can verify the quality of the MFCC features used in the previous assignments further, and explore the performance difference between DTW-based and HMM-based single-digit speech recognition. Please ask for help if you feel stuck!
Please submit answers to all underlined questions below in the writeup as a PDF file via the course Canvas site. The tex file used to create this PDF has also been provided so that you can enter your answers directly into this document. Additionally, please run through and execute all of the cells of the ipython notebook (including the parts you added which are marked by `# TODO`) and submit the notebook via Canvas.
Lateness policy: Late submissions submitted up to Monday 5/11/20 7:00 pm (the "late deadline") will receive a 20% reduction in credit. Submissions that are later than this will receive some non-zero credit, but we cannot guarantee how much. We cannot guarantee that we will carefully grade or give feedback on submissions after the "late deadline". In addition, you have four free "late days" that you can use throughout the term to extend homework (not project) deadlines without penalty. If you are using any of your "late days" for this assignment, mention that in the comment while submitting this assignment. You should state how many "late days" you are using and how many remaining "late days" you have. The number of late days used for any given homework must be an integer.
Collaboration: You are encouraged to discuss assignments and any aspect of the course material with others, but any material you submit (writeup, code, figures) should be produced on your own.

*** Before starting this homework, you'll need to make sure you have python3 and the following packages on your system: `numpy, scipy, jupyter, matplotlib, pysoundfile`. You may use any installer of your choice. Alternatively, you an follow the steps from HW2 to create a python3 environment and install the required packages using Miniconda.

Once everything is set up, you can open a notebook with the command: `jupyter notebook hw3.ipynb`. Make sure that the kernel is set to Python 3 in the notebook. Please let us know if you need help!

0. Please fill out this questionnaire. This is purely to help us calibrate assignment load and let us know what may need to be made clearer in class. Your responses will receive a small amount of credit, independent of the actual answers.

   (i) Did you collaborate on this assignment, and if so, with whom? **Nope.**

   (ii) Approximately how many hours did this assignment take to complete? **10 - 12 hours.**

   (iii) On a scale of 1 to 5 (where 1 = trivial and 5 = impossible), what was the difficulty of this assignment?
   4. Working with all the probabilities with `logsumexp` was a little tricky.

   (iv) On a scale of 1 to 5 (where 1 = useless and 5 = essential), how useful has this assignment been to your understanding of the material?
   5. I've implemented HMMs once in vanilla Python and once in numpy. This is my first time working with HMM/GMM and it's pretty cool.

1. Hidden Markov Models (HMMs)

   (a) Show that if any elements of the parameters π or $\mathbf{A}$ for an HMM are initially set to zero, then those elements will remain zero in all subsequent updates of the EM algorithm.

   Answer:

   Suppose initially $\pi_i = 0$. By the definition of the forward algorithm, $\alpha_1(i) = \pi_i b_i(o_1) = 0$. Hence, during EM,

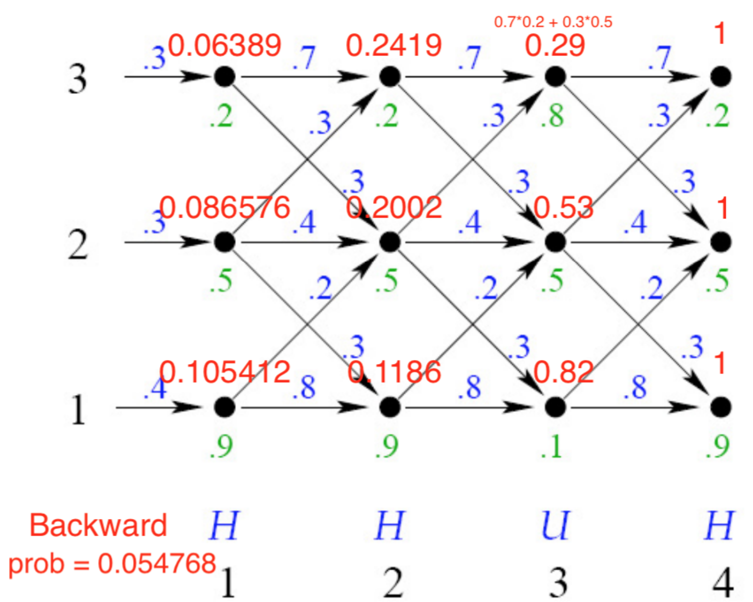
$$\hat{\pi}_i = \gamma_1(i) = \frac{\alpha_1(i)\beta_1(i)}{p(O|\lambda)} = 0$$

   Now suppose initially $a_{ij} = 0$. During EM,

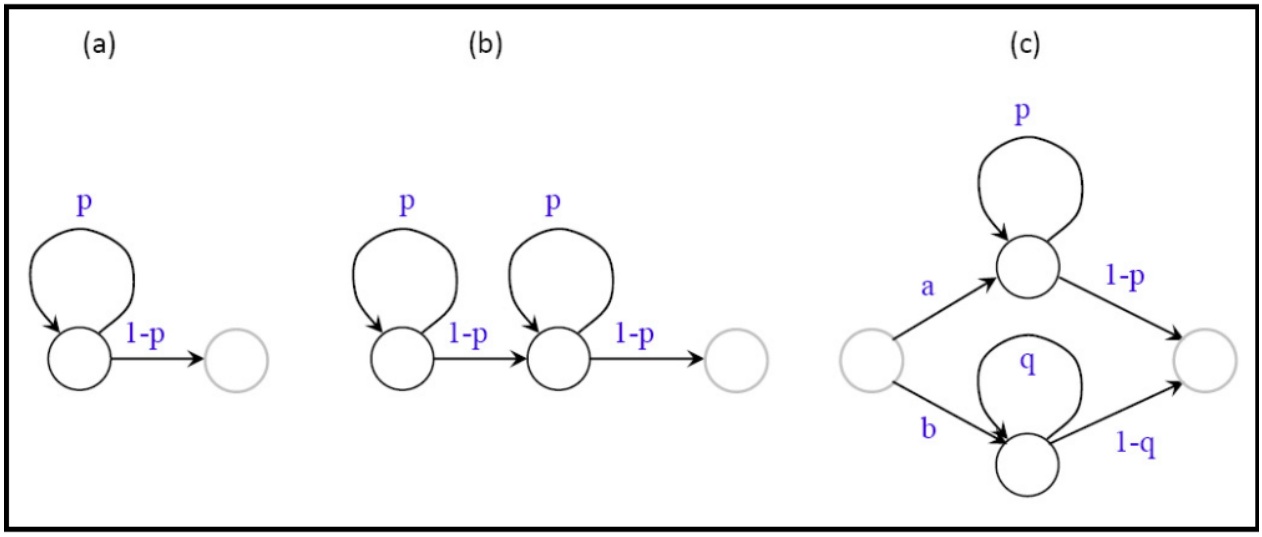
$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} = \frac{\frac{\sum_{t=1}^{T-1} \alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{p(O|\lambda)}}{\sum_{t=1}^{T-1} \gamma_t(i)} = 0$$

   (b) For the weather-mood HMM example shown in class, we manually ran the forward algorithm on a trellis for the observation sequence $O = HHUH$. Run the backward algorithm in the same way and submit your marked-up trellis, with the value of β indicated at each point,

   and final value for $p(O|\lambda)$ (which should be the same as that found in class with the forward algorithm).



   (c) As discussed in lecture, one of the limitations of HMMs is their geometric state duration distributions. One way of getting around this problem is to expand the state transition diagram in certain ways. In particular, for a given state whose duration distribution we wish to modify, we can expand it into multiple "tied" states, i.e. states with identical output distributions, with specific transition structures depending on the desired duration distribution. To see this, consider the 3 HMM transition diagrams shown below. The states with black outlines all have the same observation (emission) distributions, while the initial/final states in gray are non-emitting. HMM (a) has a single state, with probability p of staying in that state and probability 1-p of exiting the state (and therefore exiting the HMM). HMM (b) has two such states in series, and HMM (c) has two parallel paths with different transition probabilities (but still the same emission distributions).



Give an expression for the probability mass function of duration for each model.
That is, if X is the number of time steps from the start state to the exit state,
what is the PMF of X, $p(X)$, for each case?
Optional: Generalize the result to any number N of states connected in parallel or in series,
and/or plot some distributions of such models in python or another tool.

Answer:

(a) $X \sim \text{Geometric}(1-p)$,

$$P(X = k) = p^{k-1}(1-p)$$

(b) Let X_1 denote the number of steps from the first state to the second state and let X_2 denote the number of steps from the second state to the third state. $X = X_1 + X_2$, X_1 and X_2 are independent Geometric$(1-p)$ random variables.

$$\begin{aligned}
P(X = k) &= P(X_1 + X_2 = k) \\
&= \sum_{m=1}^{k-1} P(X_1 = m) P(X_1 + X_2 = k | X_1 = m) \\
&= \sum_{m=1}^{k-1} P(X_1 = m) P(X_2 = k - m | X_1 = m) \\
&= \sum_{m=1}^{k-1} P(X_1 = m) P(X_2 = k - m) \\
&= \sum_{m=1}^{k-1} p^{m-1}(1-p) p^{k-m-1}(1-p) \\
&= \sum_{m=1}^{k-1} (1-p)^2 p^{k-2} \\
&= (k-1)(1-p)^2 p^{k-2}
\end{aligned}$$

To generalize this to N states connected in series, each having self-transitioning probability p, recall that the sum of iid Geometric(p) random variables is a Negative Binomial random variable.
For $X \sim \text{NegativeBinomial}(N, 1-p)$, we need in total N "successes" (transitions to the next state in series) to get to the exit state.

$$P(X = k) = \binom{k-1}{N-1} (1-p)^N p^{k-N}$$

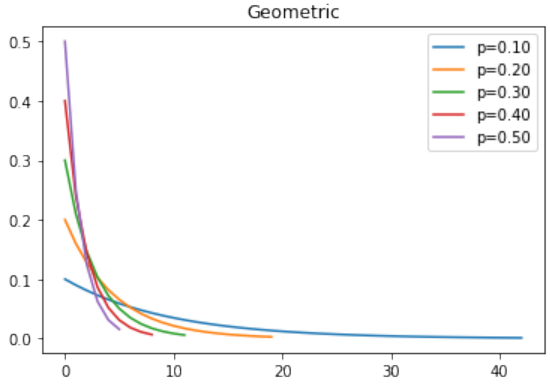
(c) Note that $a + b = 1$. Let A be the event that the start state transitions to the upper state, and B be the event that it transitions to the bottom state. $P(A) = a, P(B) = b = 1 - a$. Let X_1 denote the number of time steps from the upper state to the exit state, and X_2 the number of steps from the bottom state to the exit. $X_1 \sim \text{Geometric}(1-p)$, $X_2 \sim \text{Geometric}(1-q)$.

$$\begin{aligned} P(X = k) &= P(X_1 = k|A)P(A) + P(X_2 = k|B)P(B) \\ &= ap^{k-1}(1-p) + (1-a)q^{k-1}(1-q) \end{aligned}$$

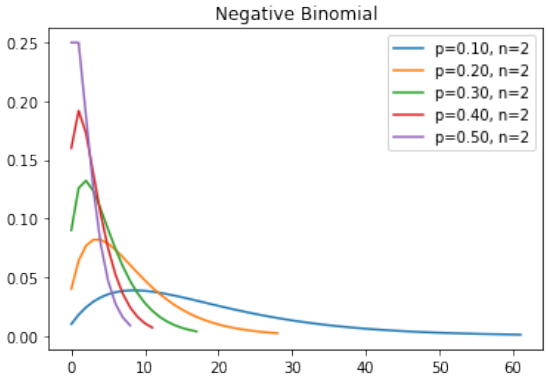
To generalize this to N states connected in parallel, denote the events of transitioning from the start state to each of the states tied in parallel as $A_i, i = 1, ..., N, \quad P(A_i) = a_i, \quad \sum_{i=1}^{N} a_i = 1$. Denote each state's self-transitioning probability as p_i and the time steps it take from state i to the exit state as X_i. $X_i \sim \text{Geometric}(1-p_i)$. The result is essentially a weighted sum of different but independent Geometric random variables.

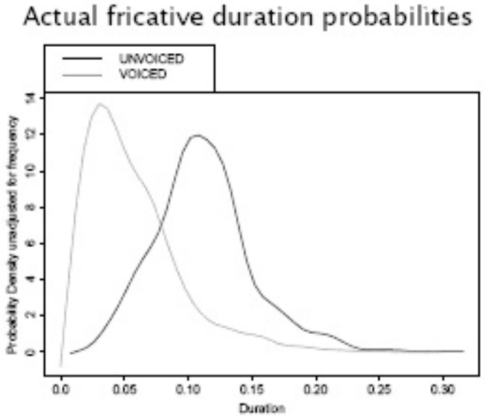
$$\begin{aligned} P(X = k) &= \sum_{i=1}^{N} P(X_i = k|A_i)P(A_i) \\ &= \sum_{i=1}^{N} a_i p_i^{k-1}(1-p_i) \end{aligned}$$

The following plots are visualizations of the PMFs of Geometric distributions, Negative Binomial distributions, and multiple tied states. Note that p denote the probability of transitioning to the next state instead of the self-transitioning probability. Please refer to plot.ipynb for the code.



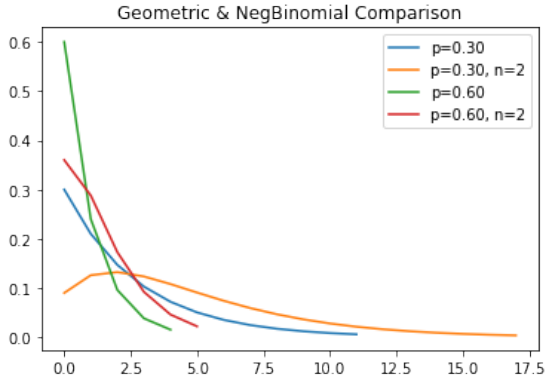
(a) From the Geometric distribution plot, if the probability of transitioning to another state is as high as 0.5, we'd expect the duration of staying in the current state to be no more than 5 time steps. On the other hand, if p is as low as 0.1, we'd expect the duration to be 0 to 10 steps.



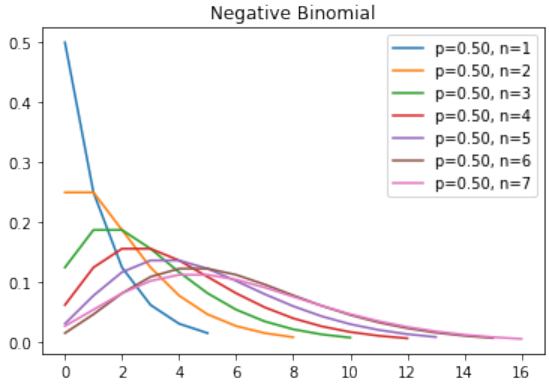


(b) The plot above depicts the expected number of steps before exiting in the case where we have 2 tied states in series, each with probability $p = 0.1, 0.2, \ldots, 0.5$ of transitioning to the next state. Unlike the strictly decreasing

trend we observe in the Geometric distribution plot, we actually see a peak in the probability of staying in these tied states for longer before exiting. This actually looks very similar to the fricative duration probability we saw in the slides (reproduced above).

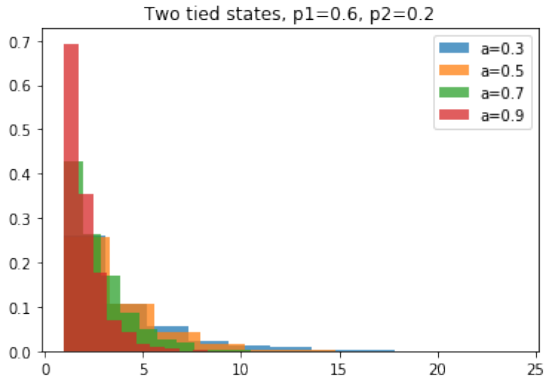


Overlaying the PMF of Geometric vs. Negative Binomial distributions reveals that the expected number of steps before exiting is higher in two-tied states than a single state, for the same p.



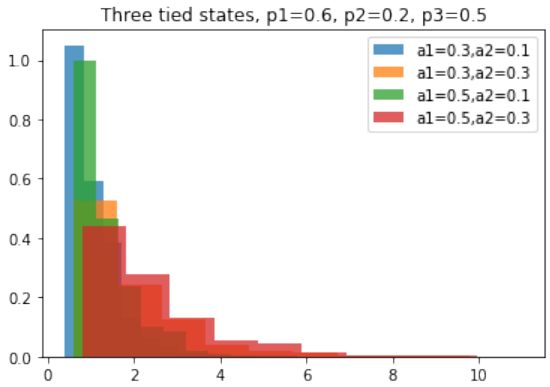
The plot above depicts the scenario where we have $n = 1, 2, 3, \ldots, 7$ tied states in series, each with probability $p = 0.5$ of transitioning to the next state. Note that $n = 1$ is just the special case, a Geometric distribution. We see that as the number of tied states increase, the expected number of steps before exiting as well as the corresponding probability stretch out over a wider range.

(c) The following plots depict parallel configuration. Since the configurations could be very diverse, we only present some selected cases.



The plot above shows two states tied in parallel. The start state has a probability $a = 0.3, 0.5, \ldots$ of transitioning into state No.1 with $p1 = 0.6$ exit probability. In parallel, the start state has a probability of $1 - a$ to transition into

state No.2 with $p2 = 0.2$ exit probability. We observe that different values of a can produce PMF of very diverse shapes. The same is depicted in the plot below with three states tied in parallel.



Overall, these plots lend some insights into the motivation of state tying: We may be able to model more complicated duration probabilities with serial or parallel state tying.

2. Single Digit Recognition with HMMs

For this part you will use a basic HMM-based recognizer to do single-digit recognition. You will use a data set consisting of training and test utterances containing isolated digits (55 each of 0-9, where '0" is always pronounced "oh" and not "zero"). The data and scripts are to be downloaded from the canvas site.

(a) Complete the implementations, in the ipynb, for the following (Note: all `# TODO`'s are in the code and can be completed in a single line):

- forward, backward, and Viterbi algorithms by filling in the 'forward', 'backward', and 'viterbi' functions
- log probability of the observed sequence (denoted `log_prob`)
- state posteriors (denoted `gamma`)
- state transition probabilities (i to j at time t) (denoted `xi`)
- means and (diagonal) covariances (denoted `self.mu` and `self.sigma`)

(b) Experiment with HMM single-digit recognition and see if you can improve the error rate. At least, experiment with different values for the number of states. Optionally, try changing other aspects: parameter initialization, convergence criteria, HMM topology, or any other aspect that you find interesting to experiment with. Run the corresponding added/modified cells in the notebook before submitting.

(c) Describe in 1-2 paragraphs what you did, the results you obtained, how they compared to your results with DTW, and any other comments about your experiments (e.g. the trade-off between performance and efficiency, number of iterations required, types of errors, ...). There will be some extra credit for the best final performance. (Note that, like in HW2, we are again "cheating" by tuning on the test set...)

Answer:

I experimented with 15 states/15 iterations, 25/25, and 50/50 and achieved accuracy close to the expected one in the instructions.

Some benchmarks on the accuracy:

	15/15	25/25	25/25 w/o convergence criteria	50/50	50/50 w/o convergence criteria
Forward	0.9607	0.9839	0.9786	0.9857	0.9875
Viterbi	0.9625	0.9839	0.9786	0.9857	0.9875

In addition, I experimented with the convergence criteria. I used the sum of the negative log probabilities `-log_prob`. Call this `plog` for positive log probabilities. We expect to see decreasing plogs as long as the model is still improving. My motivation for using plogs as a convergence criterion came from the observation that plogs actually increase in later iterations in the 50/50 configuration.

```
Starting iteration 46...
plog: 201686.96106066398
Starting iteration 47...
plog: 201691.90985687368
Starting iteration 48...
plog: 201695.7104363686
Starting iteration 49...
plog: 201701.6403190393
```

An increase in plogs indicates that the probabilities assigned to our training data by the model has decreased. In other words, our model becomes less adequate. This might be the result of overfitting the GMM. Therefore, instead of relying on the iteration numbers, I terminate my training as soon as the total plogs of the entire data set start to increase. (Alternatively, I could also terminate as soon as the plogs are not decreasing significantly, because that shows our model isn't improving significantly any more.)
In my training, I found that the 25/25 configuration terminated at iteration 21 for digit 2, iteration 16 for digit 5, and so on. In the 50/50 configuration, the training terminated around the 45th iterations or earlier for most of the digits. Please see the last page of this writeup for the complete training result summary.

Confusion matrix comparison with DTW:

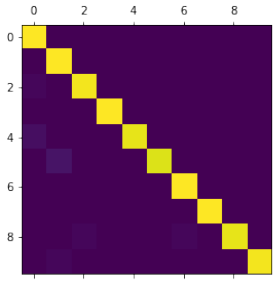


Figure 1: 98% accuracy with HMM/GMM

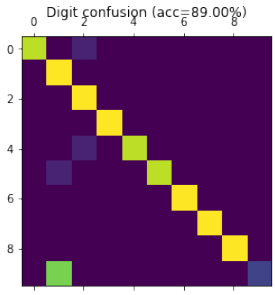


Figure 2: 89% accuracy with DTW

Remarks on the results:

- Accuracy in comparison with DTW: The 96%+ accuracy is much better than the 89% I achieved with DTW+Manhattan distance.
- Performance and efficiency trade-off: However, there is certainly some trade-off: DTW needs no training and is relatively fast to evaluate. On the other hand, even the least complex 15/15 HMM takes at least 5 minutes to train in Google Colab, and to evaluate the test set, we need to run the forward and Viterbi algorithms again. Moreover, HMM only improves marginally with a more complex configuration: the more complex 50/50 HMM takes at least 15 minutes to train but does only about 1% better than the 15/15 configuration on an absolute scale.
- Number of iterations and number of states: My experiments showed that the accuracy generally increases with more complexity: an increasing number of states and iterations, up to the point where the plogs started to increase. In the 25/25 with/without convergence criteria pair, the one that terminates training early saw an accuracy rate almost comparable to the 50/50 configuration, much better than the one without convergence criteria. Curiously, this doesn't hold for the 50/50 case. The one without convergence criteria actually achieved the best accuracy out of all five configurations in my experiment. This might just be some chance variations on the test set.
- Accuracy difference among different digits: Some digits achieve very high accuracy in all different configurations. Please refer to the tables on the next page. For example, digits 1 and 3 both achieved 100% forward and Viterbi accuracy in all my experiments. This could be because they have a more unique pronunciation than the rest of the digit class. In both the 25/25 and the 50/50 configuration, digit 5 and 6 achieve perfect accuracy. Perhaps this many number of states can capture some uniqueness in their pronunciation that a fewer number of states cannot. We also observe a significant increase in accuracy for digit 9 from the 15/15 configuration to the 25/25 configuration. This might be explained as well by the previous reason.

Additional remarks:
The slowness in training is also due to the fact that our code is not optimized as much as possible. For one, slicing `log_alpha[:, t]` is not as efficient as slicing `log_alpha[t]` because of memory layout and caching. If we have GPU available, it might also be cost-effective to zero-pad all training sequences to the same length, wrap them in a tensor, and adapt our forward, backward, EM algorithms to the tensor version to make full use of numpy's broadcasting features.

A detailed summary of the training results of different configurations (number of states, iterations, whether to terminate early based on plogs) starts on the next page.

Table 1: Accuracy, 15 states, 15 iterations, with convergence criteria

digit	plogs	forward	Viterbi
0	210,920.21	0.96	0.96
1	225,027.99	1.00	1.00
2	210,238.74	0.96	0.96
3	231,951.47	1.00	1.00
4	228,970.04	0.96	0.96
5	246,257.64	0.96	0.96
6	288,855.31	0.96	0.96
7	252,754.37	0.96	0.96
8	213,001.77	0.98	0.98
9	242,917.41	0.84	0.86

Table 2: 25/25 with convergence criteria

digit	plogs	forward	Viterbi
0	207,182.95	1.00	1.00
1	223,080.70	1.00	1.00
2	203,514.80	0.98	0.98
3	221,792.61	1.00	1.00
4	227,258.91	0.96	0.96
5	244,598.53	0.95	0.95
6	260,531.77	1.00	1.00
7	232,856.75	1.00	1.00
8	205,456.76	0.96	0.96
9	229,491.23	0.98	0.98

Table 3: 50/50 without convergence criteria

digit	plogs	forward	backward
0	212,800.00	0.98	0.98
1	218,846.60	1.00	1.00
2	200,624.37	0.98	0.98
3	219,848.58	1.00	1.00
4	236,116.60	0.95	0.95
5	235,505.60	1.00	1.00
6	253,358.52	1.00	1.00
7	232,374.02	1.00	1.00
8	205,026.74	0.98	0.98
9	230,825.49	0.98	0.98