Assignment 3

BIOENG 145/245

Due Monday 2/13 11:59PM

*****PLEASE DO NOT MODIFY THE FUNCTION NAMES AND THE FILE NAMES OR THE AUTOGRADER WILL BREAK!!*****

1 Expectation Maximization - Finding Motifs

In this section, we will implement the expectation maximization algorithm for finding motifs that we saw in lecture (look back at the lecture slides from lecture 3 for details). Implement the following functions in `q1.py` according to the specifications of each function.

- `init_p`: This function initializes a weight matrix representing the profile of the motif.
- `update_locations`: This function updates Z, a matrix representing the probability of the motif starting at each location in each sequence, based on the previous estimate of p, the motif profile.

  NOTE: Updating Z requires us to multiply a long chain of probabilities together, which results in very small numerical values. Python does not represent these tiny values very accurately, which leads to rounding errors. In order to avoid these rounding errors, we can scale each probability by some constant value (10 works well in this case). Since we normalize Z so that the sum of each row is equal to 1, these constants will be cancelled out later in the function.
- `update_locations_E_or_M`: Is `update_locations` the expectation or maximization step of the EM algorithm? Have this function return 'E' or 'M' in order to answer.
- `update_profile`: This function updates the motif profile, p, based on the previous estimate of Z, the location matrix.

  NOTE: Remember that in lecture we used pseudo-counts in order to avoid dividing by 0 and in order to account for the fact that rare events are not impossible simply because we haven't observed them in our limited number of samples. For this problem, use a pseudo-count value of one.
- `update_profile_E_or_M`: Is `update_profile` the expectation or maximization step of the EM algorithm? Have this function return 'E' or 'M' in order to answer.
- `run_EM`: Run the EM algorithm for motif-finding by putting together the other functions that you implemented: `init_p`, `update_locations`, and `update_motif`. Terminate the algorithm once the estimate of p effectively stops changing (the difference between the current p and the previous p is less than epsilon for each value in p).

2 K Nearest Neighbors - Optional

This question is OPTIONAL, and you are NOT expected to complete it!

In this section, you will implement K-Nearest Neighbors from scratch. The only package allowed available to you is NumPy. There will be 4 functions for you to implement, and the autograder will grade some functions individually and in the end the overall performance of the `KNearestNeighbors.class`.

As we learned in lecture, K-Nearest Neighbors is a powerful clustering algorithm that allows data without labels to be clustered into groups accordingly (is this supervised or unsupervised learning?). Additionally, this algorithm follows the abstraction of finding the expectation of some metric and maximizing that expectation (did we learn this in class? What is the abstraction of this type of algorithm called?). I recommend understanding this algorithm completely before writing code. The code for this question is located in `q2.py`.

3 functions you will implement:

- `euclidean`: This method calculates the euclidean distance between two data points, p & q. The euclidean distance is the distance we learned back in basic algebra: $d = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$ where p is a 2D point (p_1, p_2) and $q = (q_1, q_2)$. In ML where data almost always have more than 2 dimensions, the formula is as follows:
$$d = \sqrt{\sum_{i=1}^{D} (p_i - q_i)^2}$$
where D is the number of dimensions and i iterates through each feature. Since we are simply using the distance as a comparison metric, we do not need the square root operation. Thus, compute $d = \sum_{i=1}^{D} (p_i - q_i)^2$.
- `fit`: This is the core method of our K-Means class, and is the method that will run the entire K-Nearest Neighbors algorithm. The algorithm in a nutshell: grab k random data points and set them as the new centroids. Then, assign the data points closest to some centroids to that cluster. This is the expectation step, and the cluster assignments of the data points are the latent variables (e.g. the missing information in our data). For the maximization step, recalculate the mean of each newly formed cluster, and the mean will now be the new centroids (the parameters we are maximizing).
- `performance`: Calculate the performance of current clustering results. First, we add up each sample point's distance to its cluster center. Then, we average all the distances as a performance metrics. The lower the average distance the better the performance. This metric is also known as the *intertia*. Note that in this function, there is an extra bit of code that **you should NOT modify**. This code is just here to help improve the performance of K-Means for visualization purposes. All this snippet of code does is to enforce more distance between the centroids, thereby encouraging potentially better clusters. That said, please also **do not modify the `lmbda` hyperparameter!**

You can run this file to visualize the performance of your clustering algorithms. Please do not modify any other functions besides the 3 listed above!

3 Markov Model - Promoter Sequences

For the second question of the homework, you will write a Markov Model from scratch that will help you classify whether a genetic sequence is a promoter sequence or not. For sequences that are promoter sequences, we will call them *positive sequences* and *negative sequences* for those that aren't. In short, this Markov Model will keep track of the probability of seeing one kmer after another (kmers are substrings of k characters, so a 3-mer would be `ATC, GTA, CAT, etc`). The transition matrix for the Markov Model will be built based on a bunch of genetic sequences.

To use these Markov Models to perform binary classifications, We will build two transition matrices, one from the positive sequences and one from the negative sequences. Then, we use a log odds ratio to predict whether a new sequence is more likely to belong to class 1 or class 0.

As an example, let's look at the sequence `ATCGACTGCATCGACGACTGACT`, and set that it belongs to class 0 (negative). Let's say we set `K=3`, then we will see transitions such as `ATC, TCG, CGA, GAC, ...`, where `ATC, TCG, etc.` are different states of our Markov Model. For the first four letters ATCG, we say that we see a transition from ATC to TCG. That said, we will build a transition matrix where we will record the probabilities of transitioning from one kmer to the next (such as $\mathrm{P}(\texttt{TCG}|\texttt{ATC})$). However, notice that we can rewrite this as $\mathrm{P}(\texttt{TCG}|\texttt{ATC}) = \mathrm{P}(\texttt{G}|\texttt{ATC})$ (why?). To find this probability, we simply iterate through the sequences to find the following probability:

$$\mathrm{P}(\text{kmer}_j|\text{kmer}_i) = \frac{\text{\# of kmer j that transition after kmer i}}{\text{Total \# of occurrences of kmer i}}.$$

With the example used above, $\mathrm{P}(\texttt{TCG}|\texttt{ATC}) = \mathrm{P}(\texttt{G}|\texttt{ATC})$ would be obtained from finding the total number of `ATC` followed by a nucleotide `G` divided by the total number of `ATC` in the entire dataset. In the end, we will have a transition matrix for class 1 and a transition matrix for class 0 where $\mathbb{P}(\texttt{kmer\_i}, \texttt{kmer\_j})$ gives the probability of transitioning from `kmer_i` to `kmer_j`. These transition matrices are our Markov Models. (Sanity check: what should be the dimension of the transition matrix with $k = 3$? The answer is (64, 4), do you know why?)

Here are the functions you are required to implement:

- `read_data`: Read in `q3_data.csv` that includes 2 columns, one column containing the strings of sequences and one column containing the class it belongs, it could be either 0 or 1.
- `build_transition_matrix`: The main algorithm that will build our Markov Model. Refer to the code itself and the example above for a clearer explanation.
- `log_odds_ratio`: Calculates the log of probability ratio of a sequence being in class 0 or class 1. Use the following formula:

$$\texttt{log\_odds\_ratio} = \log\frac{\mathrm{P}(\text{sequence being in class 1})}{\mathrm{P}(\text{sequence being in class 0})}$$

- `classify`: takes a sequence and classifies whether the sequence is a positive class or a negative class. If log_odds_ratio > 0, (probability of positive $>$ negative) $\rightarrow$ classify as positive class! Else if log_odds_ratio $< 0 \rightarrow$ classify as negative class! (Why is this our decision rule?)

How do we determine whether a sequence belongs to class 1 or class 0? We can simply apply the chain rule of probabilities:

$$\mathrm{P}(\texttt{kmer}_1, \texttt{kmer}_2, ..., \texttt{kmer}_l) = \mathrm{P}(\texttt{kmer}_2|\texttt{kmer}_1)\mathrm{P}(\texttt{kmer}_3|\texttt{kmer}_2)...\mathrm{P}(\texttt{kmer}_l|\texttt{kmer}_{l-1}) = \prod_{i=1}^{l-1} \mathrm{P}(\texttt{kmer}_{i+1}|\texttt{kmer}_i)$$

where each $\mathrm{P}(\texttt{kmer}_{i+1}|\texttt{kmer}_i)$ can be found from the transition matrix.

Lastly, a short script has been provided for you to run the algorithm for debugging and to demonstrate your algorithm's performance. Additionally, running this file will generate **five** files, `q3_predictions_k=[1, 2, 3, 4, 5].npy`. **Please submit these 5 files along with the Python files!**

4 Submission

Here are the **8** files that need to be turned in to Gradescope:

- `q1.py, q2.py, & q3.py`
- `q3_predictions_k=[1, 2, 3, 4, 5].npy`

For the coding portion, *****PLEASE DO NOT MODIFY THE FUNCTION NAMES AND THE FILE NAMES OR THE AUTOGRADER WILL BREAK!!***** The autograder will run your program and test the code, and whatever score you see after the autograder finishes running is the score you receive for the coding portion.