Machine Learning for Signals (CS 545)
Homework 3

Due date: Nov. 3, 23:59 PM (Central)

Instructions

- Please don't submit your homework.
- The solution will be distributed on the due date.
- Please still do your homework, as the quizzes and exams will be based on the homework assignments (especially the exams).
- Unless mentioned otherwise, you are expected to implement all the functions instead of using existing toolboxes, e.g., from `sklearn`.
- Avoid using toolboxes.

P1: kNN Classification on LSH

1. `eeg_x_train.npy`, `eeg_y_train.npy`, `eeg_x_test.npy`, and `eeg_y_test.npy` contain the training and testing samples and their labels. Use them to replicate the STFT, subband selection, and reshaping part of my EEG classification experiments in the "Bayesian Classification" lecture (not the entire lecture, but from S3 to S8 and S37).
2. But, instead of PCA dimension reduction, we're going to use locality sensitive hashing to extract binary features. Also, instead of naïve Bayes, we'll do kNN classification.
3. For the kNN classification, for every test example you have to find the K nearest neighbors from 112 training samples. You're lucky. This kNN search is not a big deal, because your training set is tiny. However, as your professor I have to help you guys be prepared for your future big data projects at your fancy new job after your graduation (don't forget me). So, I'm giving you this homework assignment that will help you come up with a speed-up technique for the kNN search. It's not that complicated.
4. Come up with a random projection matrix $\boldsymbol{A} \in \mathbb{R}^{L\times M}$, where M denotes the number of dimensions of your data samples (5 rows of your magnitude STFT over a few time frames and three channels, and then vectorized)[1], and L is the new dimension after this random projection. So, you're replacing PCA with this random projection. Note that L doesn't always have to be smaller than M. Although $\boldsymbol{A}$ is a matrix with random values, it's safer to make sure that the row vectors (the projection vectors) are unit vectors, i.e. $||\boldsymbol{A}_{i,:}||_2 = 1$.
5. Do the random projection and take the sign of the result (element-wise), so that your random projection produces a bunch of +1's and -1's:

$$\boldsymbol{Y} = \text{sign}(\boldsymbol{A}\boldsymbol{X}_{1:M,:}) \qquad (1)$$

[1]Note that the final dimension 255 comes from the calculation [# channels] $\times$ [# frames] $\times$ [# subbands], while # frames can slightly vary depending on your STFT implementation. Mine on the slide resulted in $3 \times 17 \times 5 = 255$.

6. Use this $\boldsymbol{A}$ matrix and the sign function wrapper to convert your M-dimensional TRAINING samples into L-dimensional bipolar binary hash codes. Ditto for the TEST samples (use the same $\boldsymbol{A}$ matrix). If you're not comfortable with bipolar binaries, you can turn them into 0-1 binaries by replacing -1's with 0's. It's up to you.

7. Do your kNN classification using these binary version of your data samples. Note that you can compare the test bit string with the bit strings of the training data by using the Hamming distance, instead of the Eucleadian distance between the original real-valued vectors. You know your computer prefers binary values, right? This way you can speed up the comparison, and eventually the kNN search (although you might not be able to see the actual speed-up due to the too small dataset size and your script language that doesn't take advantage of binary variables).

8. Since your classification results vary by different choice of K and L, try to repeat your experiments with different hyperparameters. Also, the projection matrix is stochastic, which also makes some difference whenever you run it.

9. Compare your results with mine in S37. Your classifier is based on binary variables and bitwise operations, so you must have expected a big performance drop. How do you like it eventually?

P2: When to applaud?

1. `Piano_Clap.wav` is an audio signal simulating a music concert, particularly at the end of a song. As some of the audience haven't heard of the song before, they started to applaud before the end of the song (at around 1.5 seconds). Check out the audio.

2. Since I'm so kind, I did the MFCC conversion for you, which you can find in `mfcc.npy`. If you load it, you'll see a matrix `X`, which holds 958 MFCC vectors, each of which has 12 coefficients.

3. You can find the mean vectors and covariance matrices in `mu.npy` and `sigma.npy` that I kindly provide, too. Once you load it, you'll see the matrix `mX`, which has two column vectors as the mean vectors. The first column vector of `mX` is a 12-dimensional mean vector of MFCCs of the piano-only frames. The second vector holds another 12-dimensional mean vector of MFCCs of the claps. In addition to that, `Sigma`, has a 3D array (12×12×2), whose first 2D slice (12×12) is the covariance matrix of the piano part, and the upper 2D slice is for the clap sound.

4. Since you have all the parameters you need, you can calculate the p.d.f. of an MFCC vector in `X` for the two multivariate normal (Gaussian) distribution you can define from the two sets of means and cov matrices. Go ahead and calculate them. Put them in a 2×958 matrix:

$$\boldsymbol{P} = \begin{bmatrix} P(\boldsymbol{X}_1|\mathcal{C}_1) & P(\boldsymbol{X}_2|\mathcal{C}_1) & \cdots & P(\boldsymbol{X}_{958}|\mathcal{C}_1) \\ P(\boldsymbol{X}_1|\mathcal{C}_2) & P(\boldsymbol{X}_2|\mathcal{C}_2) & \cdots & P(\boldsymbol{X}_{958}|\mathcal{C}_2) \end{bmatrix}. \tag{2}$$

   Note that $\mathcal{C}_1$ is for the piano frames while $\mathcal{C}_2$ is for the applause. Normalize this matrix, so that you can recover the posterior probabilities of belonging to the two classes given an MFCC frame:

$$\tilde{\boldsymbol{P}}_{c,t} = \frac{P(\boldsymbol{X}_t|\mathcal{C}_c)}{\sum_{c=1}^{2} P(\boldsymbol{X}_t|\mathcal{C}_c)} \tag{3}$$

Plot this 2×958 matrix as an image. This is your detection result. If you see a frame at t' where $\tilde{\boldsymbol{P}}_{1,t'} < \tilde{\boldsymbol{P}}_{2,t'}$, it could be the right moment to start clapping.

5. You might not like the this result, because it is sensitive to the wrong claps in the middle. You want to smooth them out. For this, you may want to come up with a transition matrix with some dominant diagonal elements:

$$\boldsymbol{T} = \begin{bmatrix} 0.9 & 0.1 \\ 0 & 1 \end{bmatrix}. \tag{4}$$

What it means is that if you see a $\mathcal{C}_1$ frame, you'll want to stay at that status (no clap) in the next frame with a probability 0.9, while you want to transit to $\mathcal{C}_2$ (clap) with a probability of 0.1. On the other hand, you absolutely want to stay at $\mathcal{C}_2$ once you observe a frame with that label (i.e., you don't want to get back if you start clapping).

Apply this matrix to your $\tilde{\boldsymbol{P}}$ matrix in a recursive way:

$$\bar{\boldsymbol{P}}_{:,1} = \tilde{\boldsymbol{P}}_{:,1} \tag{5}$$
$$b = \text{argmax}_c \bar{\boldsymbol{P}}_{c,t} \tag{6}$$
$$\bar{\boldsymbol{P}}_{:,t+1} = \boldsymbol{T}_{b,:}^{\top} \odot \tilde{\boldsymbol{P}}_{:,t+1} \tag{7}$$
$$\tag{8}$$

First, you initialize the first column vector of your new posterior prob matrix $\bar{\boldsymbol{P}}$ (you have no previous frame to work with at that moment). For a given time frame of interest, $t+1$, you first need to see its previous frame to find which class the frame belongs to (by using the simple max operation on the posterior probabilities at t). This class index b is going to be used to pick up the corresponding transition probabilities (one of the row vectors of the $\boldsymbol{T}$ matrix). Then, the transition probabilities will be multiplied to your existing posterior probabilities at $t+1$.

In the end, you may want to normalize them so that they can serve as the posterior probabilities:

$$\begin{aligned} \bar{\boldsymbol{P}}_{1,t+1} &= \bar{\boldsymbol{P}}_{1,t+1} / \left(\bar{\boldsymbol{P}}_{1,t+1} + \bar{\boldsymbol{P}}_{2,t+1}\right) \\ \bar{\boldsymbol{P}}_{2,t+1} &= \bar{\boldsymbol{P}}_{2,t+1} / \left(\bar{\boldsymbol{P}}_{1,t+1} + \bar{\boldsymbol{P}}_{2,t+1}\right). \end{aligned} \tag{9}$$

Repeat this procedure for all the 958 frames.

6. Plot your new smoothed post prob matrix $\bar{\boldsymbol{P}}$. Do you like it?

7. If you don't like the naïve smoothing method, there is another option, called the Viterbi algorithm. This time, you'll calculate your smoothed post prob matrix in this way:

$$\bar{\boldsymbol{P}}_{:,1} = \tilde{\boldsymbol{P}}_{:,1} \tag{10}$$

First, initialize the first frame post prob as usual. Then, for $(t+1)$-th frame, we will construct $\bar{\boldsymbol{P}}_{c,t}$. For this, we need to see t-th frame, but this time we check on all paths to pick up the best one by calculating all the probabilities of state transitions from c at t to c' at $t+1$. For example, from (c,t) to $(c'=1,t+1)$:

$$b = \text{argmax}_c \boldsymbol{T}_{c,1} \bar{\boldsymbol{P}}_{c,t}. \tag{11}$$

This will tell you which of the two previous states $c=1$ and $c=2$ at t was the best transition to the current state $\mathcal{C}_1$ at $t+1$ (we're seeing only $\mathcal{C}_1$ for now). If $b=1$, it's more probable that the previous state at t was $\mathcal{C}_1$ rather than $\mathcal{C}_2$. We keep a record in our $\boldsymbol{B}$ matrix for this best choices:

$$\boldsymbol{B}_{1,t+1} = b \tag{12}$$

Eventually, we use this best transition probability to construct the post prob of $\mathcal{C}_1$ at $t+1$:

$$\bar{\boldsymbol{P}}_{1,t+1} = \boldsymbol{T}_{b,1} \bar{\boldsymbol{P}}_{b,t} \boldsymbol{P}_{1,t+1} \tag{13}$$

$$\tag{14}$$

In other words, the prior probability (the transition probability and the accumulated post prob in the t-th frame), $\boldsymbol{T}_{b,1}\bar{\boldsymbol{P}}_{b,t}$, is multiplied to the likelihood $\boldsymbol{P}_{1,t+1}$ to form the new post prob.

We keep this best previous state sequences. For example, $\boldsymbol{B}_{1,102}$ will hold the more likely previous state at 102-th frame if that frame's state is $\mathcal{C}_1$, while $\boldsymbol{B}_{2,102}$ records its corresponding 101-th state that could have transit to $\mathcal{C}_2$.

8. Don't forget to repeat this for $\bar{\boldsymbol{P}}_{2,t+1}$.
9. Don't forget to normalize $\bar{\boldsymbol{P}}$ as in (9).
10. Once you construct your new post prob matrix from the first frame to the 958-th frame, you can start backtracking. Choose the larger one from $\bar{\boldsymbol{P}}_{1,958}$ and $\bar{\boldsymbol{P}}_{2,958}$. Let's say $\bar{\boldsymbol{P}}_{2,958}$ is larger. Then, refer to $\boldsymbol{B}_{2,958}$ to decide the state of the 957-th frame, and so on.
11. Plot this series of states as your backtracking result. This will be the hidden state sequence you wanted to know. Is the start of the applause making more sense to you?

P3: Multidimensional Scaling

1. `pdist.npy` holds a 4,037×4,037 square matrix, which holds squared pairwise Euclidean distances that I constructed from a set of map locations.
2. I won't share the original map with you to intrigue you.
3. Instead, you'll write a code for multidimensional scaling so that you can recover the original map out of $\boldsymbol{L}$. If your MDS routine is successful, you should be able to see what the original map was. The original map was in the 2D space, so you may want to see two largest eigenvectors.
4. Scatterplot the eigenvector coefficients on a new 2D space to see what the original map looks like: dots represent the locations on the map.
5. Note that the MDS result can be rotated and shifted.

P4: oncentric circles classification using kernel PCA

1. `concetric_x.npy` contains 152 data points each of which is a 2-dimensional column vector. If you scatter plot them on a 2D space, you'll see that they form the concentric circles you saw in class. `concetric_y.npy` contains their labels.

2. Do kernel PCA on this data set to transform them into a new 3-dimensional feature space, i.e. $\boldsymbol{X} \in \mathbb{R}^{2\times 152} \Rightarrow$ Kernel PCA $\Rightarrow \boldsymbol{Z} \in \mathbb{R}^{3\times 152}$.

3. In theory, you have to do the centering and normalization procedures in the feature space, but for this particular data set, you can forget about it.

4. You remember how the after-transformation 3D features look like, right? Now your data set is linearly separable. Train a perceptron that does this linear classification. In other words, your perceptron will take the transformed version of your data (a 3-dimensional vector) and do a linear combination with three weights $\boldsymbol{w} = [w_1, w_2, w_3]^\top$ plus the bias term b:

$$y_i = \sigma\left(\boldsymbol{w}^\top \boldsymbol{Z}_{:,i} + b\right), \tag{15}$$

   where y_i is the scalar output of your perceptron, σ is the activation function you define. If you choose logistic function as your activation, then you'd better label your samples with 0 or 1.

5. You'll need to write a backpropagation algorithm to estimate your parameters $\boldsymbol{w}$ and b, which minimize the error between the prediction y_i and the ground-truth label t_i. There are a lot of choices, but you can use the binary cross entropy error: $t_i \log(y_i) + (1 - t_i)\log(1 - y_i)$, whose derivative w.r.t. $\boldsymbol{w}$ is $(y_i - t_i)\boldsymbol{Z}_{:,i}$, for example (see L8-S39). But a simple sum-of-squared loss should also work.

6. Note that this simple perceptron doesn't have a hidden layer but it should still work thanks to the kernel PCA's nonlinear transformation.

7. Note that you may want to initialize your parameters with some small (uniform or Gaussian) random numbers centered around zero.

8. Your training procedure will be sensitive to the choice of the learning rate and the initialization scheme (the range of your random numbers). So, you may want to try out a few different choices. But at any rate, this should give you a 100% accuracy on the training set. Good luck!