

ECE 448 Assignment 2 Report (3 credits)

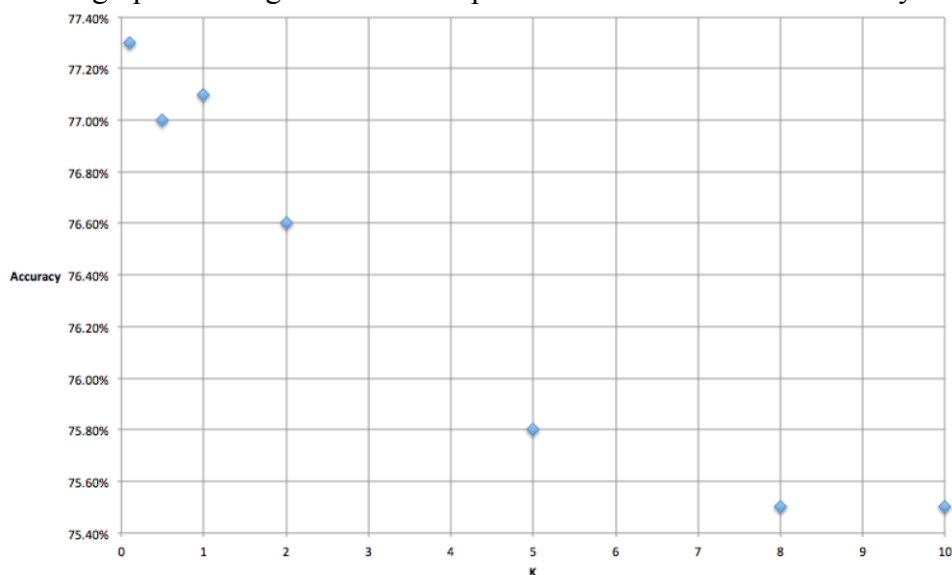
Renjie Fan, rfan8
Jiangtian Feng, jfeng18

Part 1: Digit Classification

1.1 Single pixels as feature. (part1.1.py)

Features: ‘#’ and ‘+’ (foreground): $F_{ij} = 1$; ‘ ’ (background): $F_{ij} = 0$

Training: Calculating the likelihoods for each pixel location for every digit class with 2 distinct feature values. V (#unique value) = 2, as F_{ij} could be 0 or 1; $k = 0.01$. Through a lot of experiments, 0.01 gives the highest accuracy rate of classification. Below is a scatter graph showing the relationship between value of k and accuracy of classification:



The graph clearly shows that the smaller the k value, the more accurate the Naïve Bayesian Classification. In fact, if k value were set to 0.01, the accuracy would be 78%.

Testing: Maximum A Posterior function, comparing the log value of posteriors of each digit class for each image.

Classification Results:

Total Classification Rate: 77.3 %. Out of 1000 images.
Classification Rate for 0 : 84.4 %. Out of 90 images.
Classification Rate for 1 : 96.3 %. Out of 108 images.
Classification Rate for 2 : 78.6 %. Out of 103 images.
Classification Rate for 3 : 80.0 %. Out of 100 images.
Classification Rate for 4 : 74.8 %. Out of 107 images.
Classification Rate for 5 : 68.5 %. Out of 92 images.
Classification Rate for 6 : 76.9 %. Out of 91 images.
Classification Rate for 7 : 72.6 %. Out of 106 images.
Classification Rate for 8 : 60.2 %. Out of 103 images.
Classification Rate for 9 : 80.0 %. Out of 100 images.

Confusion Matrix:

	0	1	2	3	4	5	6	7	8	9
0	0.0 %	0.0 %	1.1 %	0.0 %	1.1 %	5.6 %	3.3 %	0.0 %	4.4 %	0.0 %
1	0.0 %	0.0 %	0.9 %	0.0 %	0.0 %	1.9 %	0.9 %	0.0 %	0.0 %	0.0 %
2	1.0 %	2.9 %	0.0 %	3.9 %	1.9 %	0.0 %	5.8 %	1.0 %	4.9 %	0.0 %
3	0.0 %	1.0 %	0.0 %	0.0 %	0.0 %	3.0 %	2.0 %	7.0 %	1.0 %	6.0 %
4	0.0 %	0.0 %	0.9 %	0.0 %	0.0 %	0.9 %	3.7 %	0.9 %	1.9 %	16.8 %
5	2.2 %	1.1 %	1.1 %	13.0 %	3.3 %	0.0 %	1.1 %	1.1 %	2.2 %	6.5 %
6	1.1 %	4.4 %	4.4 %	0.0 %	4.4 %	6.6 %	0.0 %	0.0 %	2.2 %	0.0 %
7	0.0 %	4.7 %	3.8 %	0.0 %	2.8 %	0.0 %	0.0 %	0.0 %	2.8 %	13.2 %
8	1.0 %	1.0 %	2.9 %	13.6 %	2.9 %	7.8 %	0.0 %	1.0 %	0.0 %	9.7 %
9	1.0 %	1.0 %	0.0 %	3.0 %	10.0 %	2.0 %	0.0 %	2.0 %	1.0 %	0.0 %

Digit Class 2	Least Prototypical Index:	790
Most Prototypical Index:	795	

```
++++++
#####+
+#####+
++#+   +###+
++      +###+
          +##
          +##
          +#
          +#
          +#
          +#+
          +#+
      +++++ ##+
+#####+##
+#####+
+###+++++#####
###+   +#####++
###+#####   +###+
+#####+
+###+

```

```

+ + + + + ##### +
+ + + ##### +
+ ##### +
+ ##### + ##### +
+ ##### + ##### +
+ ##### + ##### +
##### + + + + + + ##### +
### + + + ##### +
# + + ##### +
+ + + ##### +
+ + #####
##### +
+ ##### +
+ ##### +
+ ##### +
+ ##### +
+ #####

```

Digit Class 3 Least Prototypical Index: 681
Most Prototypical Index: 205

[illegible]

```

      +#####++
    ++#####+
  #####+++++#####
+###+          +###+
#####          #####
+++            +###+
                +###
                +####+
                ++###+
                +####+
                +####+
                ###+
                +##
                ++###
                ++####+
                +####+
                +####+
                +#####+
                +#####+
                +#####+
                +#####+

```

```
Digit Class 4
Most Prototypical Index: 111
Least Prototypical Index: 253
```

+ +
+#+ +
##+ +
++#+ ++
++++# +++
+### ++
+##+ ++
++
##+ ++
#+ ++
+++++ ++
++++++ ++
++++++ ++
++++++ ++
++++++ ++
++++++ ++
++++++ ++

```

+##+
+###
+###      ++##+
+###      +###+
+####      +###+      +
+####      +###+      ++##+
+####      +###+      +##+
+####      +###+      +##+
+###+      +####      +###+
+###+      +##      ++####+
+###+      +###+#####+
+###+      ++++#####++
+#####++
+#####++
+++++++##
          ###+
          +##+
          +##+
          +##
          +#

```

Digit Class 5 Least Prototypical Index: 737
Most Prototypical Index: 471

```

      ++##+
    ++++###+
  #####+
  ###+++++
++#++
+#+
+##
++#+++
+#####+
+###+##
+      +##+
      +##+
      +##+
      ##
++#+      ##+
++#+      +##
###+      ###
++##+      ++#+
##++++###+
      #####

```

```

++###+
+#####+
#####+
#####+
#####+
#####+
+++++#####
      +++++###+
            +#####+
                  +++###+
                        ###+
                              +###+
                                    +###+
                                          ++++++
+#####+++++#####
+#####
+#####
      ++++++

```

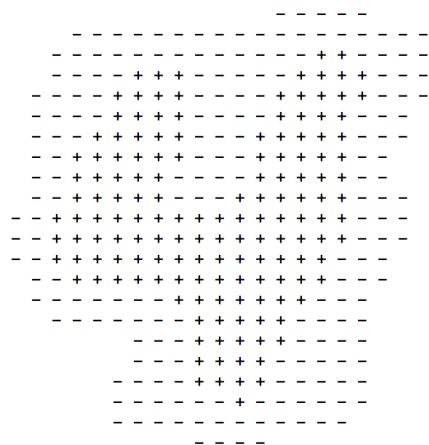

Odd Ratios:

For log likelihood maps: $\log \text{ value} > -1 \rightarrow '+'$
 $\log \text{ value} > -3 \rightarrow \text{'—'}$
 $\log \text{ value} < -3 \rightarrow \text{' '}$

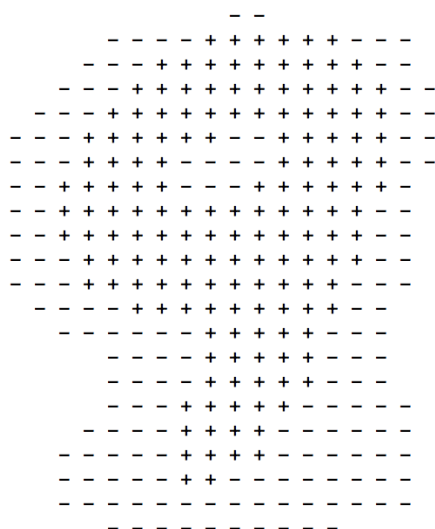
For log odds ratio maps: $\log \text{ odd} > 0 \rightarrow '+'$
 $\log \text{ odd} > -2 \rightarrow \text{'—'}$
 $\log \text{ odd} < -2 \rightarrow \text{' '}$

Four pairs of digit with highest confusion rate:

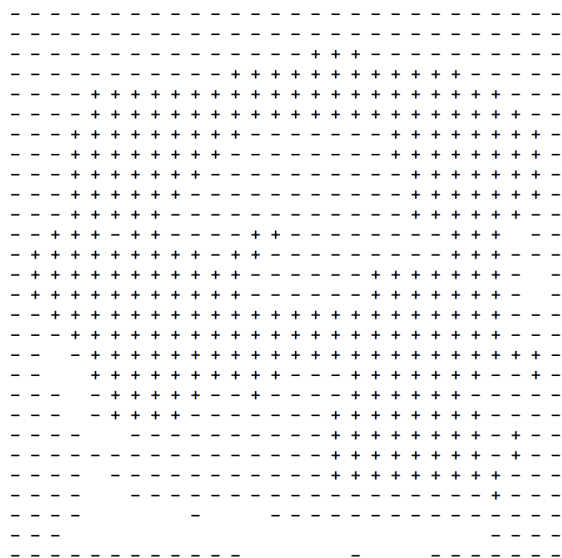
(4, 9) : 16.8%



log likelihood map for digit 4:

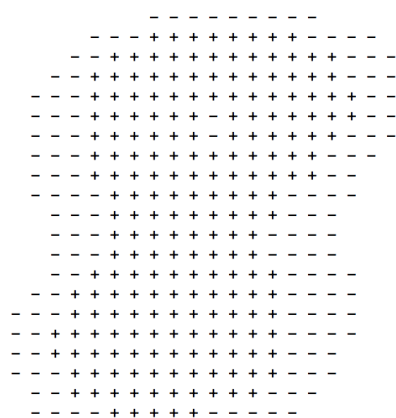


log likelihood map for digit 9:

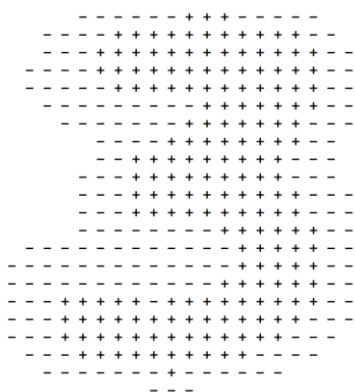


log ratio for 4 over 9:

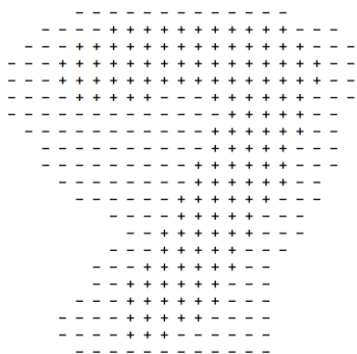
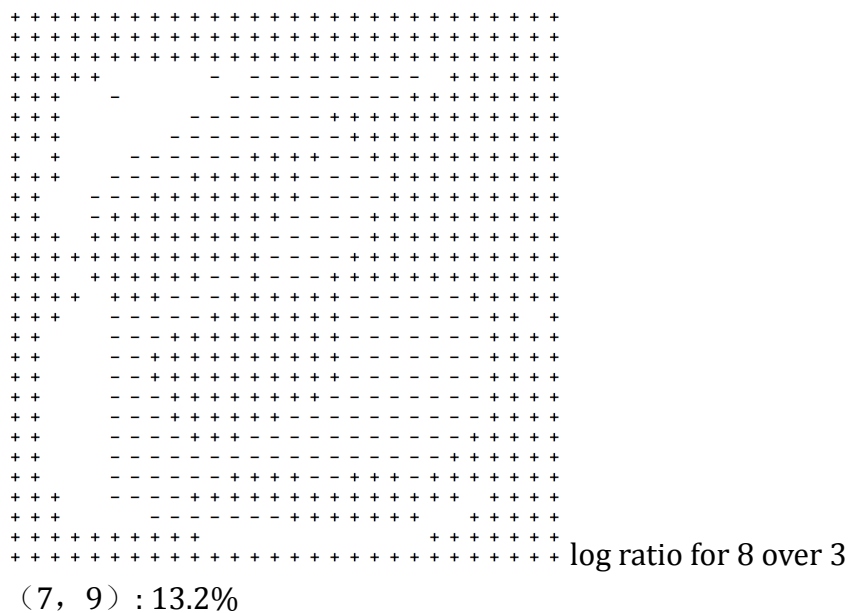
(8, 3) : 13.6%



log likelihood map for digit 8:

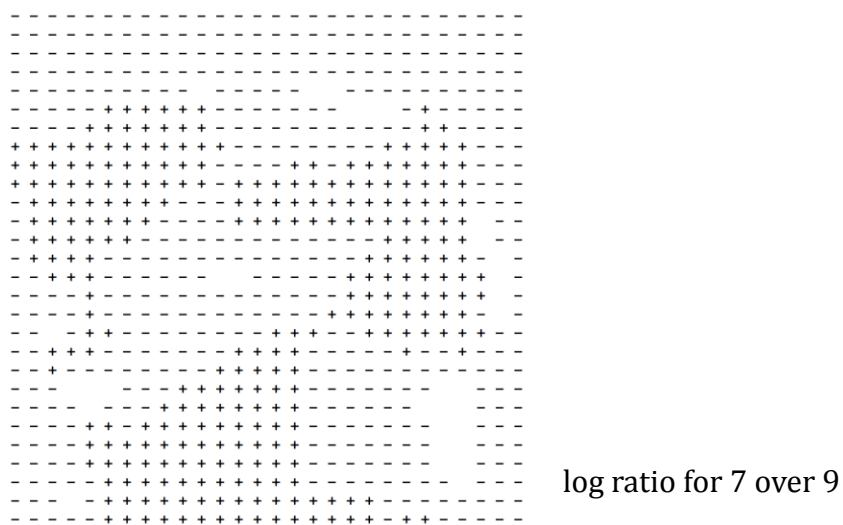


log likelihood map for digit 3:

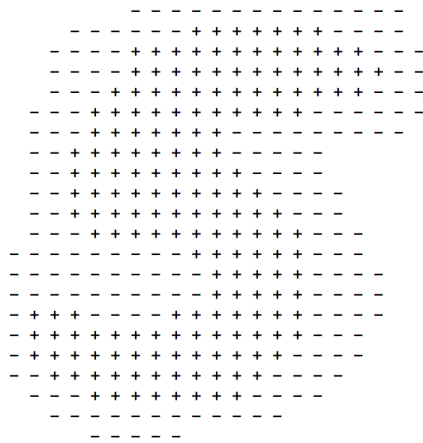


log likelihood map for digit 7

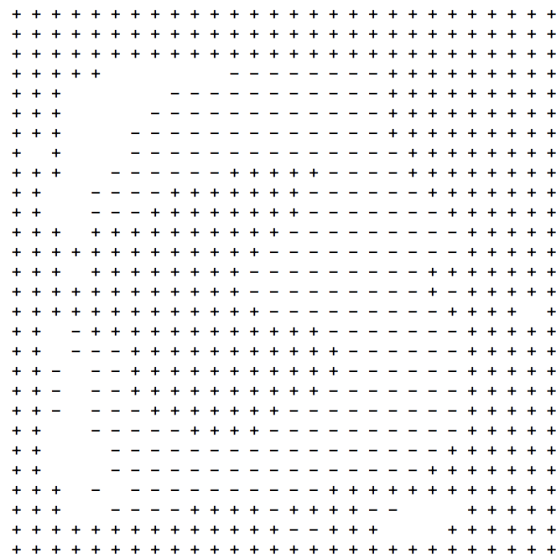
log likelihood map for digit 9 already reported.



(5, 3) : 13.0%



log likelihood map for digit 5



log ratio for 5 over 3

In Result part of the code, there are 4 parts, Basic Data, Confusion Matrix, Highest and Lowest Posteriors and Odd Ratios. Uncomment them to see output.

1.2 Pixel groups as feature. (part1.2.(n*mdis/over).py files)

Single Pixel: (2 distinct feature values, 784 features)

Accuracy: 77.3%

Training Running Time: 1.3s

Testing Running Time: 3.2s

Disjoint Patches of size 2*2: (16 distinct feature values, 196 features)

Accuracy: 85.8%

Training Running Time: 1.3s

Testing Running Time: 2.5s

Disjoint Patches of size 2*4: (256 distinct feature values, 98 features)

Accuracy: 84.9%

Training Running Time: 1.7s

Testing Running Time: 2.9s

Disjoint Patches of size 4*2: (256 distinct feature values, 98 features)

Accuracy: 85.8%

Training Running Time: 1.7s

Testing Running Time: 3.0s

Disjoint Patches of size 4*4: (65536 distinct feature values, 49 features)

Accuracy: 79.1%

Training Running Time: 37.4s

Testing Running Time: 2.1s

Overlapping Patches of size 2*2: (16 distinct feature values, 729 features)

Accuracy: **87.1% Over 80%.**

Training Running Time: 4.5s

Testing Running Time: 9.7s

Overlapping Patches of size 2*4: (256 distinct feature values, 675 features)

Accuracy: 79.0%

Training Running Time: 10.8s

Testing Running Time: 21.1s

Overlapping Patches of size 4*2: (256 distinct feature values, 675 features)

Accuracy: 79.5%

Training Running Time: 11.0s

Testing Running Time: 20.4s

Overlapping Patches of size 2*3: (64 distinct feature values, 702 features)

Accuracy: 78.8%

Training Running Time: 9.0s

Testing Running Time: 18.6s

Overlapping Patches of size 3*2: (64 distinct feature values, 702 features)

Accuracy: 79.4%

Training Running Time: 9.3s

Testing Running Time: 18.7s

Overlapping Patches of size 3*3: (512 distinct feature values, 676 features)

Accuracy: 77.1%

Training Running Time: 13.1s

Testing Running Time: 21.6s

Single pixel has 784 features, each with 2 distinct feature values. As the k value I choose for the single pixel feature is 0.1, I use $k=0.1$ for all pixel groups features.

For disjoint patches, the accuracy increases significantly as the size of pixel group increases. The reason is that disjoint patches actually increases the number of distinct feature values, for example, 2*2 disjoint patch evaluates 16 different feature values. However, the accuracy of 4*4 disjoint patch is unexpectedly lower than other 3 disjoint groups, less than 80%. For large disjoint patch size, number of feature value turns very big. For example, 4*4 pixel group has 65536 (2^{16}) different feature values, and it's rare to match a same feature value in test examples.

For overlapping patches, number of features does not decrease greatly, but they include more number of distinct feature values. For example, 2*2 overlapping patch evaluates 729 features, each with 16 feature values. So it achieves the highest accuracy, 87.1%. Accuracy of overlapping patches remains stable through different sizes of group.

An important conclusion is that the balance between number of features and number of feature values must be achieved to gain the highest classification accuracy.

Training running time increases as the size of pixel groups increases, because the number of distinct feature values ($\# = 2^{(n*m)}$) increases. So it needs to expand more nodes for each training image. Also, overlapping patch has longer training running time than disjoint patch with same group size.

Testing running time remains stable for both disjoint patch and overlapping patch.

Disjoint: 2~3 seconds.

Overlapping: 18~21 seconds.

Part1 Extra, Ternary Feature (part1.ternary.py)

Features: '+' \rightarrow Fij = 2; '#' \rightarrow Fij = 1; ' ' \rightarrow Fij = 0

In Laplace smoothing, set value of v to 3, as feature has 3 distinct values for ternary features. For value of k , 0.1 still achieves the highest classification accuracy.

Total Classification Rate: 77.8 %. Out of 1000 images.
Classification Rate for 0 : 83.3 %. Out of 90 images.
Classification Rate for 1 : 95.4 %. Out of 108 images.
Classification Rate for 2 : 76.7 %. Out of 103 images.
Classification Rate for 3 : 81.0 %. Out of 100 images.
Classification Rate for 4 : 76.6 %. Out of 107 images.
Classification Rate for 5 : 68.5 %. Out of 92 images.
Classification Rate for 6 : 81.3 %. Out of 91 images.
Classification Rate for 7 : 73.6 %. Out of 106 images.
Classification Rate for 8 : 60.2 %. Out of 103 images.
Classification Rate for 9 : 81.0 %. Out of 100 images.

Confusion Matrix:

	0	1	2	3	4	5	6	7	8	9
0	0.0 %	0.0 %	1.1 %	0.0 %	0.0 %	6.7 %	3.3 %	0.0 %	5.6 %	0.0 %
1	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	1.9 %	0.9 %	0.0 %	1.9 %	0.0 %
2	1.0 %	2.9 %	0.0 %	3.9 %	1.0 %	1.0 %	6.8 %	1.9 %	4.9 %	0.0 %
3	0.0 %	1.0 %	0.0 %	0.0 %	0.0 %	3.0 %	2.0 %	6.0 %	2.0 %	5.0 %
4	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.9 %	2.8 %	0.9 %	1.9 %	16.8 %
5	2.2 %	1.1 %	1.1 %	13.0 %	3.3 %	0.0 %	1.1 %	1.1 %	2.2 %	6.5 %
6	0.0 %	3.3 %	3.3 %	0.0 %	4.4 %	5.5 %	0.0 %	0.0 %	2.2 %	0.0 %
7	0.0 %	5.7 %	2.8 %	0.0 %	2.8 %	0.0 %	0.0 %	0.0 %	2.8 %	12.3 %
8	1.0 %	1.0 %	2.9 %	11.7 %	2.9 %	8.7 %	0.0 %	1.0 %	0.0 %	10.7 %
9	1.0 %	0.0 %	0.0 %	2.0 %	10.0 %	2.0 %	0.0 %	2.0 %	2.0 %	0.0 %

Classification using ternary feature approaches a higher accuracy than binary feature, but not a large improvement, because in training and testing images, number of '+' is way less than number of '#', so it's not significantly different to separate them in likelihood calculation.

Part1 Extra, Face Detection (part1.face.py)

Features: '#' \rightarrow Fij = 1; ' ' \rightarrow Fij = 0

In Laplace smoothing, set value of v to 2, as feature has 2 distinct values for binary features. For value of k , $k=1$ achieves the highest classification accuracy through a number of experiments.

Accuracy: 90.7% out of 150 images.

Part 2: Audio classification

Part 2.1:

Model:

1.load training and test files: I use the function named **get_data (filename)** to load data from four input files: no_test.txt, no_train.txt, yes_test.txt, yes_train.txt; this function reads each file by regarding ‘%’ as ‘0’ and ‘ ’(blank) as ‘1’. My implementation is to build a 3-D array which stores certain amounts of 2-D array; each 2-D array is regarded as a **spectrogram** and it has 28 rows that first 25 rows are data, the rest 3 rows are left blank. And in each 2-D array there are 250 characters (25*10) that consist of a spectrogram.

2.probability model:

Random variable (binary variable): amount: 250; domain: blank, “%”

Outcome (spectrogram): complete specification of the state of the spectrogram.

Specifically, all 250 random variables in spectrogram has a certain value(blank or %).

Outcome could belong to event(class) “yes” or “no”.

Event (class): yes and no.

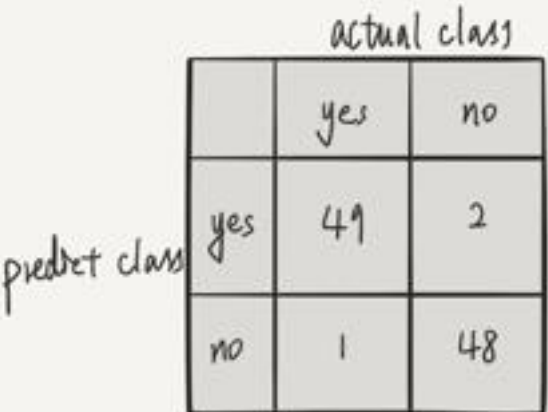
3.According to my probability model, now I need a method to predict whether a outcome belongs to “yes” or “no” by analyzing the training data. Specifically, **posteriori** $P(\text{class} | \text{character})$ is necessary(MAP: assign a spectrogram to the class with the highest posterior); thus, we are supposed to calculate the **prior** $P(\text{class})$ and **likelihood** $P(\text{character} | \text{class})$. I build the function named **likelihood(data)** to do so. It calculates the likelihood $P(\text{character} | \text{class})$ by estimating it as the proportion of the training data from that class. (simply get the sum of each random variable, and divide it by amount of spectrogram to get likelihood). Furthermore, in this function, I use the Laplace smooth to avoid probability 0(arithmetic exception) for log-likelihood and increase the classification accuracy. After experimenting all kinds of k value(0.1 - 10), I choose $k = 7.6$; and V

should be 2(the number of possible values each character can take on is 2). Prior of those two class is even simpler: divide the amount of spectrogram in each class by the total amount of spectrogram in those two class.

Testing: I build `naive_baes_classifier(yes_test, no_test)` function to classify yes and no spectrogram. In this function, it loads the test data first; then traverse the 3-D array built by `get_data()` function. By comparing the value of $P(\text{yes} \mid \text{character})$ with $P(\text{no} \mid \text{character})$, it classifies the spectrograms in file “no_test.txt” and “yes_test.txt” into corresponding class. The accuracy of classification and confusion matrix are listed below:

Accuracy: yes_test: 98%(49/50); no_test: 96%(48/50)

Confusion matrix:(horizontal direction: actual class; vertical direction: predicted class)



		actual class	
		yes	no
predet class	yes	49	2
	no	1	48

Part 2.2(my implementation is very similar to part2.1):

Model:

1.load training and test files: I use the function named **get_data_2(filename)** to load data from two input files: training_data.txt and testing_data.txt; this function reads each file by regarding ‘%’ as ‘0’ and ‘ ’(blank) as ‘1’. My implementation still is to build a 3-D array which stores certain amounts of 2-D array; each 2-D array is regarded as a **banalized MFCC(spectrogram)** and it has 33 rows that first 30 rows are data, the rest 3 rows are left blank. And in each 2-D array there are 390 characters(30*13) that consist of a spectrogram. Furthermore, each spectrogram corresponds to a label that belongs to class audio digit 1-5. In order to prepare the training and testing, I build the function named **get_label_2(filename)**; it reads the input file(training_labels.txt and testing_labels.txt) to generate two arrays that contains corresponding labels for testing and training data.

2.probability model:

Random variable(binary variable): amount: 390; domain: blank, “%”

Outcome(spectrogram): complete specification of the state of the spectrogram.

Specifically, all 390 random variables in spectrogram has a certain value(blank or %).

Outcome could belong to event(class): audio 1-5.

Event(class): audio 1, audio 2, audio 3, audio 4, audio 5.

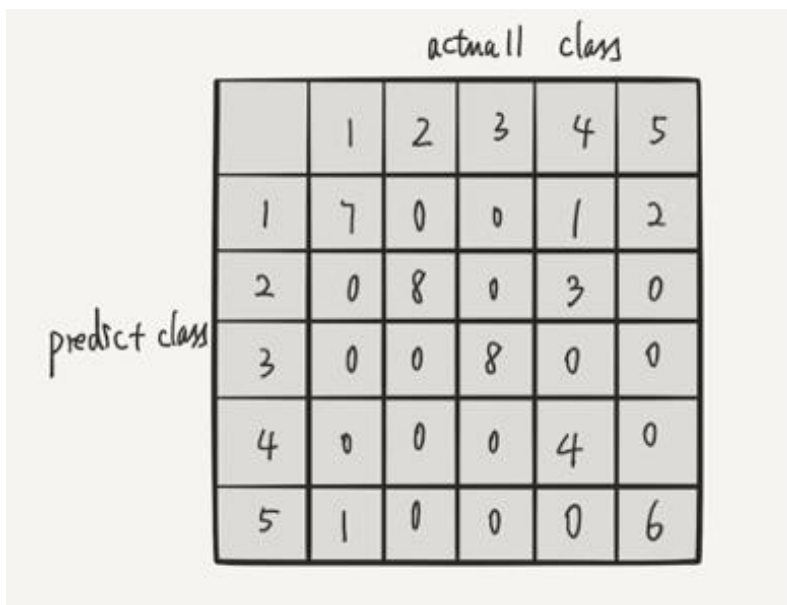
3.According to my probability model, now I need a method to predict which event(class) a outcome belongs to by analyzing the training data. Specifically, **posteriori** $P(\text{class} | \text{character})$ is necessary(MAP: assign a spectrogram to the class with the highest posterior); thus, we are supposed to calculate the **prior** $P(\text{class})$ and **likelihood** $P(\text{character} | \text{class})$. I build the function named **likelihood_2(data, label, labels)** to do so. Label represents the specific class to calculate.(e.g. label = 1, calculate the likelihood for audio 1) “Labels” is an array that contains all the corresponding labels for the “data”. Each spectrogram in “data” corresponds to label in “labels” by order. This function calculates the likelihood

$P(\text{character} \mid \text{class})$ by estimating it as the proportion of the training data from that class. (simply get the sum of each random variable, and divide it by amount of spectrograms(12 for training data) to get likelihood). Finally, in this function, I use the Laplace smooth to avoid the probability 0(arithmetic exception) for the log-likelihood and increase classification accuracy. After experimenting all kinds of k value(0.1 - 10), I choose $k = 0.8$; and V should be 2(the number of possible values each character can take on is 2). Prior for 5 audio is same because each class has same amount of spectrogram(12 for training data, 8 for testing data).

Testing: I build `naive_baes_classifier_2(yes_test, no_test)` function to classify those spectrograms that belong to audio 1-5. In this function, it loads the test data first; then traverse the 3-D array built by `get_data_2()` function. By comparing the log-based posterior * prior value (with `argmax()` function in “numpy” library), it classifies the spectrograms in file “testing_data.txt” into corresponding class(audio 1-5). Finally, according to the “testing_labels.txt”, it calculates the classification accuracy. The accuracy of classification and confusion matrix are listed below:

Accuracy: 82.5%(33/40)

Confusion matrix:(horizontal direction: actual class; vertical direction: predicted class)



	actual class				
	1	2	3	4	5
1	7	0	0	1	2
2	0	8	0	3	0
3	0	0	8	0	0
4	0	0	0	4	0
5	1	0	0	0	6

Part 2 extra credit 1: classification with unsegmented version

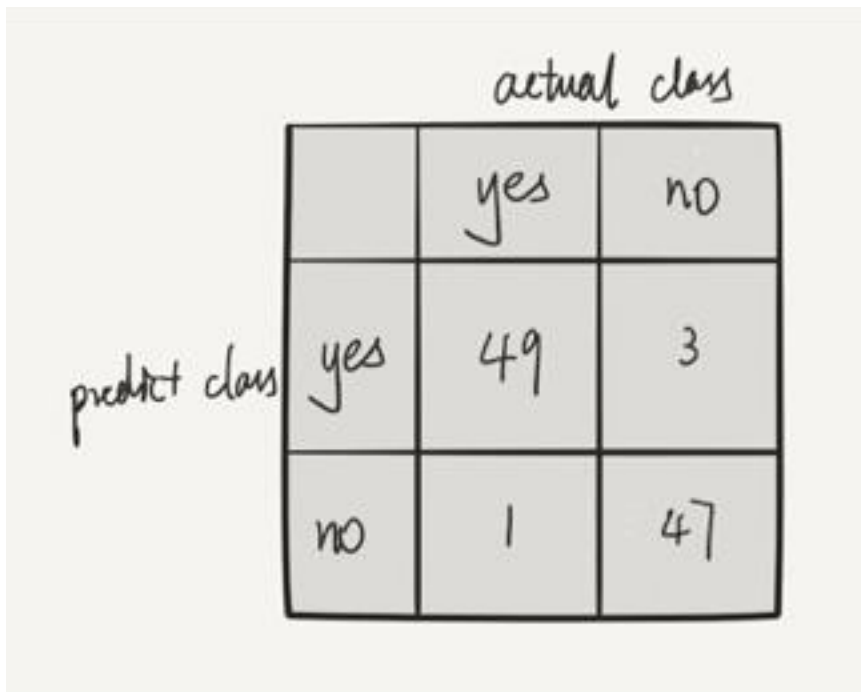
Specifically, this part is same as part2.1 except the training data is unsegmented, which contains several garbage data(maybe noise??). In “part2.ec1.py”, I use the same `model(naive_bayes_classifier(yes_test, no_test), likelihood(data))` and same probability model as part2.1 after I split the garbage data out. Finally, for the Laplace smooth constant, I choose $K = 0.4$, $V = 2$, which generate most accurate result.

Different `get_data` functions: because in this part, the training data are unsegmented and are not stored in a single file. Data are stored in a directory that contains 60 files. Thus, I build the function named `get_train_data(directory)`. This function read every file in the provided directory. Likewise, the useful data in each file are 8 spectrogram which belongs to yes or no class, which depends on filename. For example, “0_0_0_0_1_1_1_1.txt” means that in this file first 4 spectrograms belong to no class and rest 4 belong to yes class. After reading a file, this function generates a 2-D array (25*150). In this array, “%” is represented as 0; blank is represented as 1. Now I have to analyze this raw data to split the garbage out. Thus, I build another function named `throw_garbage_away(raw_dataset, high_ener)`. `Raw_dataset` contains the data read from each file; `High_ener` is an constant that generated by the amount of blank in `raw_dataset` divided by 8. This number represent the amount of blank in each spectrogram approximately. Specifically, this function traverse the `raw_dataset` by columns: when finding a column has at least one high energy(blank, 1) character, it records the subsequent 10 columns including this column. Then calculate the amount of blanks in these 10 columns; if it’s larger than (`high_ener - offset`), store this spectrogram as useful data. (There should be an offset because not all high energy characters belong to useful spectrogram data; By experimenting different offset value, I choose the one that gives me largest useful data among 60 files). Finally, this function returns the useful spectrogram it finds. Now we come back to `get_train_data()`, we have to judge the amount of spectrogram `throw_garbage_away()` provide. If the amount is eight, the data

are really useful; otherwise, those are still garbage. If it's really useful, divide those 8 spectrogram into yes training data or no training data by the filename. After reading 60 files, 44 of them are truly useful for training. On the other hand, for the test, the test data are also stored in a directory with several files. Thus, I build the function **get_test_data(directory)** that reads the test data.

Accuracy: yes_test: 98%(49/50); no_test: 94%(47/50)

Confusion matrix:(horizontal direction: actual class; vertical direction: predicted class)



A hand-drawn confusion matrix table. The horizontal axis is labeled 'actual class' and the vertical axis is labeled 'predict class'. The table has two columns for actual classes ('yes', 'no') and two rows for predicted classes ('yes', 'no'). The counts are: 49 correct 'yes' predictions, 3 incorrect 'yes' predictions (actual 'no'), 1 incorrect 'no' prediction (actual 'yes'), and 47 correct 'no' predictions.

		actual class	
		yes	no
predict class	yes	49	3
	no	1	47

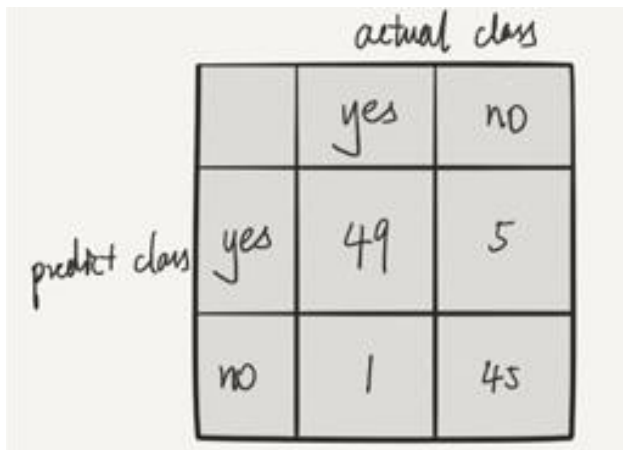
Part 2 extra credit 3: classify by 25 feature

This part use the same data loading function named **get_data(filename)**; different function for calculating likelihood and classification: **likelihood_3(data)**, **naive_bayes_classifier_3(yes_test, no_test)**.

Specifically, there is only one difference between **likelihood_3()** and **likelihood()** function used in part2.1. Each spectrogram has 25 rows and 10 columns. In part2.1, each character has 2 possibilities: blank and %(250 characters). Now, we regard the spectrogram as 25 “strings”; each row has 11 probabilities (0, 1/10, 2/10, 3/10,...,1), which are the amount of blank character divided by 10. Thus, in **likelihood_3** function, I build a 2-D array that sizes 25*11 to represent total likelihood; each row in the array has 11 position that store the likelihood for the corresponding spectrogram row. Therefore, for the classifier, when testing, it add corresponding log-based likelihoods stored in 2 2-D array(one for yes class, one for no class) and corresponding prior. Finally, comparing the value to classify to class. For the Laplace smooth constant, I choose $k = 4.5$ $v = 11$ (there are 11 possible values the feature can take on).

Accuracy: yes_test: 98%(49/50); no_test: 90%(45/50)

Confusion matrix: (horizontal direction: actual class; vertical direction: predicted class)



		actual class	
		yes	no
predict class	yes	49	5
	no	1	45

Individual Contribution:

Renjie Fan	-----	Part1.1, Part1.2, Part1 Extra Credits and related report.
Jiangtian Feng	-----	Part2.1, Part2.2, Part2 Extra Credits and related report.