## Part 1: Naive Bayes Classifiers on Digit classification

### 1-1:  Single pixels as features

**Implementation:**

**I first read the data and store them as ArrayList<ArrayList<String>> in the variable “dataset”. I then store each digit figure with an array with 1024 elements. I create a 3 dimensional array(10\*32\*32) called “probability” to store the figure into the corresponding digit(0~9). I also create a 2 dimension array called “count feature” and store the number of occurrence of the features(i,jth value is 1) in it, and then I normalize it with the total size of the dataset. Thus, I can get the likelihood function. I also calculate the prior probability as the number of the occurrence of each digit divided by the total size of the training set.**

**For the test set, I read the data into an ArrayList<String> as testset, where the size of the string is 1024(32\*32). And I use the logarithm of the prior probability calculated and sum it with other 1024 log likelihood function and get the final max apriori probabioity. I then decide that which number is most likely to be the result based on it.**

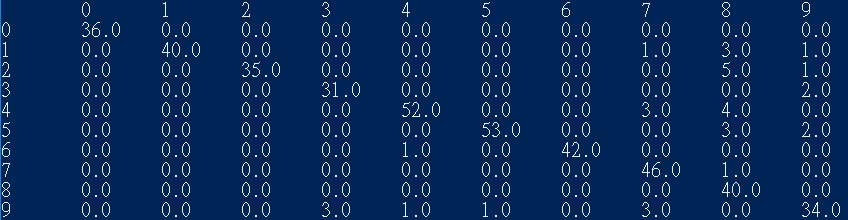
**With the smoothing factor chosen as 0.1, I am able to get the test set accuracy as 92.12%. I found that with no smoothing factor, the accuracy is around 85%, however, when the smoothing factor is larger than 0.1, the accuracy also keeps decreasing.**

**For the confusion matrix, I calculate all the test tokens and assign the real input as rows, and the classify result as the columns. With the confusion matrix, I am able to calculate the “precision” and “recall”.**

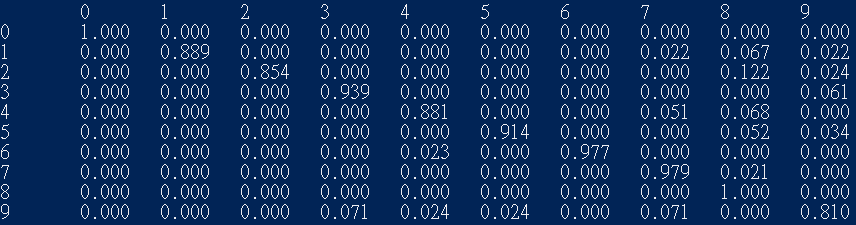
**For the odd ratio problem, I calculate the ratio of the conditional probability of “given class 1 and the feature value i,j is 1” and “given class 2 and the feature value i,j is 1”. With such method, I am able to tell that which symbols best separate the 2 similar tokens.**

****

**Confusion matrix total count:**

****

**Confusion matrix:**

****

**Defining precision as , and recall as**

**Then for each digit:**

0:precision:100%, recall:100%

1:precision:100%, recall:88.9%

2:precision:100%, recall:85.3%

3:precision:91.1%, recall:93.9%

4:precision:96.3%, recall:88.1%

5:precision:98.1%, recall:91.4%

6:precision:100%, recall:97.7%

7:precision:86.8%, recall:97.9%

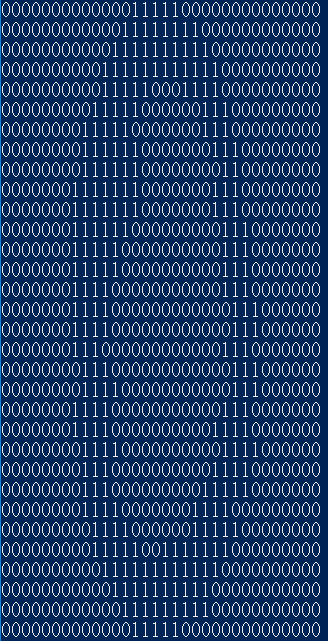
8:precision:71.4%, recall:100%

9:precision:85%, recall:81%

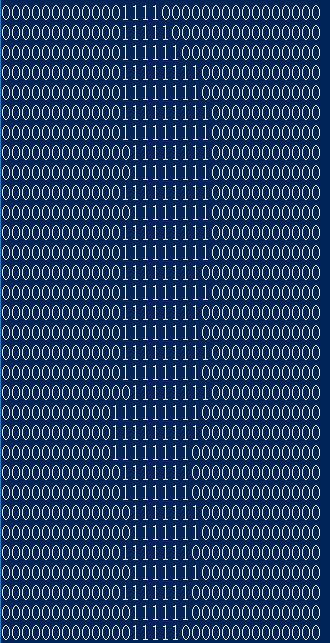
**Test tokens with the highest/the lowest probability for according to the classifier:**

**Highest:**

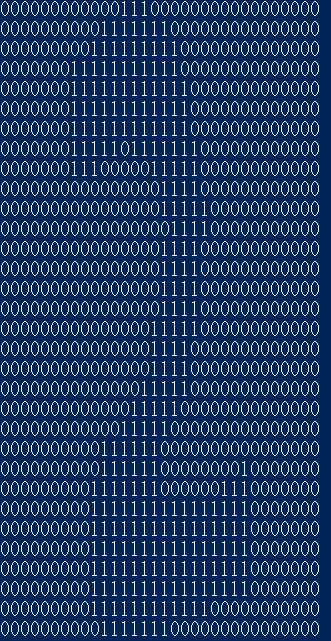
**0:**

****

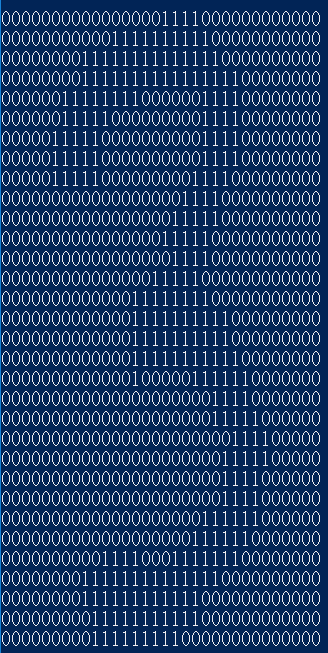
**1.**

****

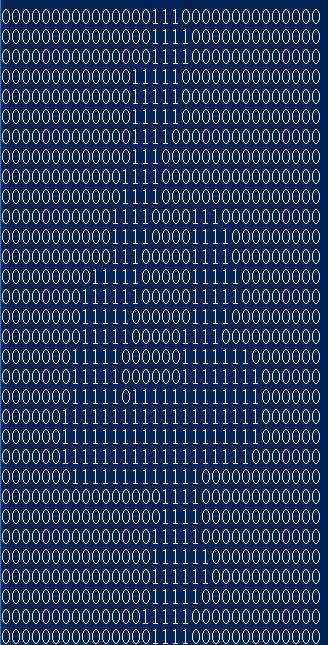
**2.**

****

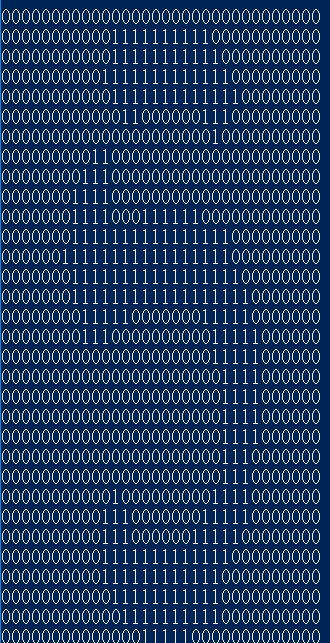
**3.**

****

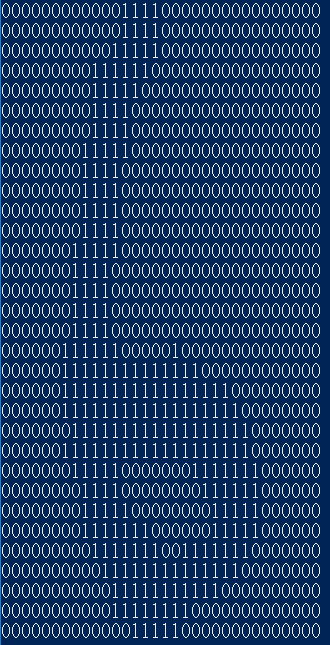
**4.**

****

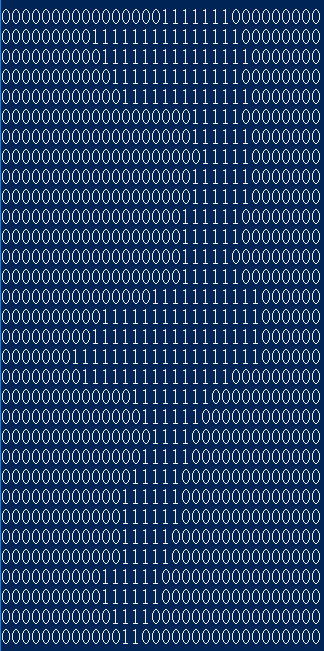
**5.**

****

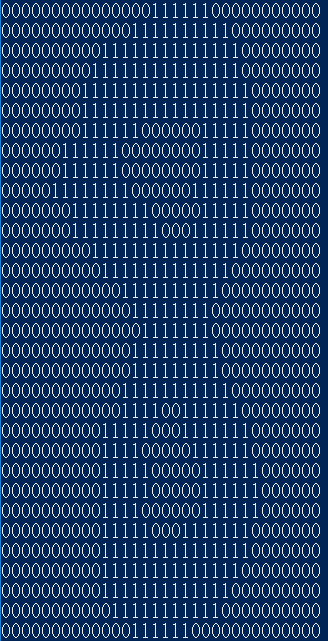
**6.**

****

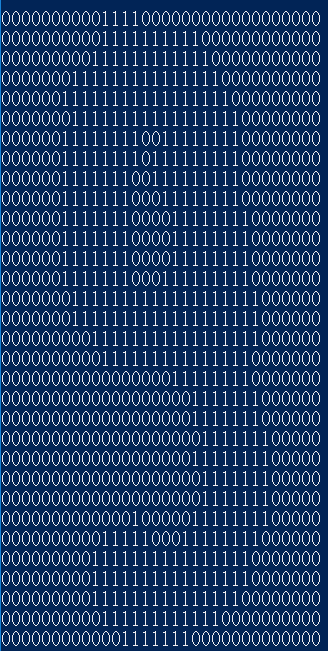
**7.**

****

**8.**

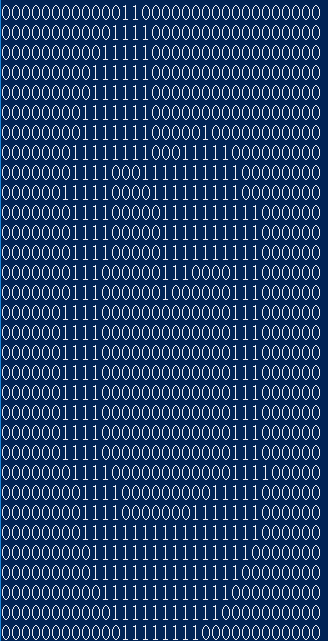
****

**9.**

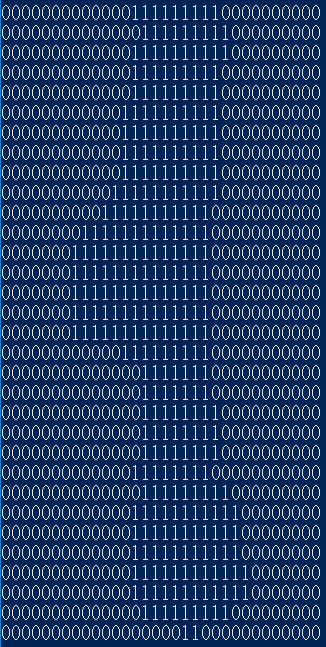
****

**Lowest:**

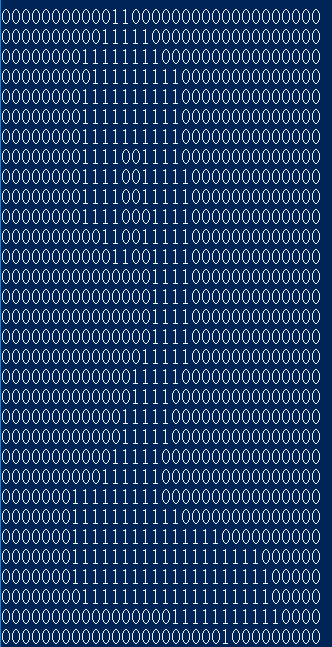
**0.**

****

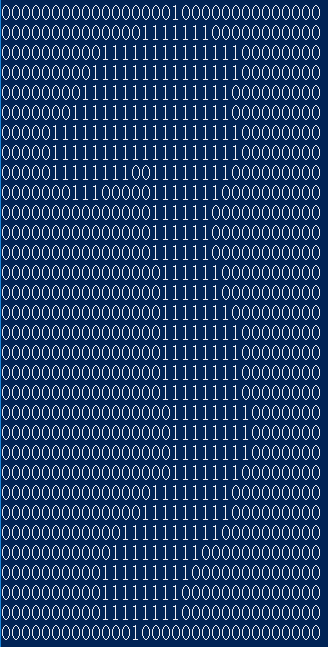
**1.**

****

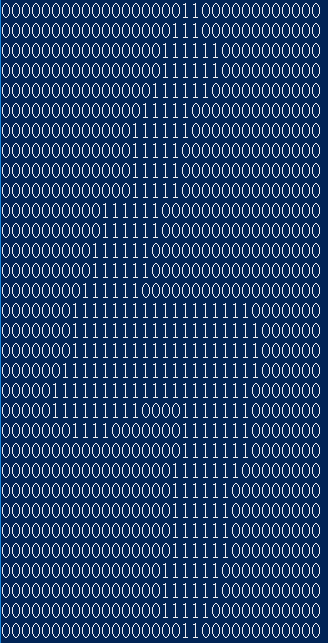
**2.**

****

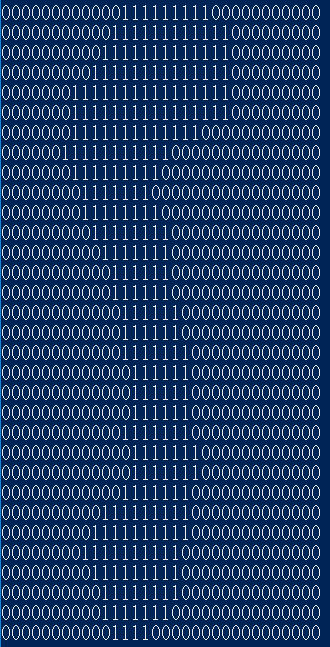
**3.**

****

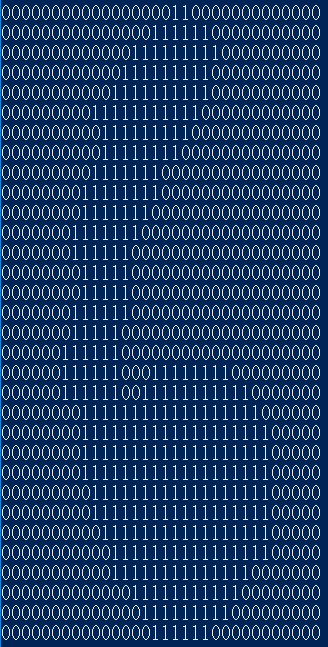
**4.**

****

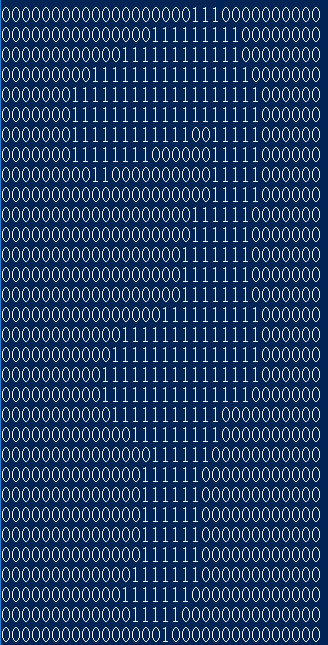
**5.**

****

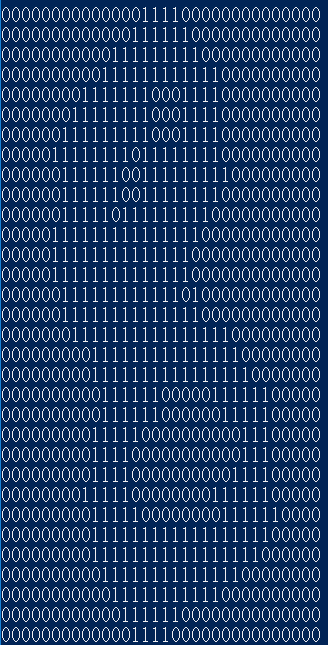
**6.**

****

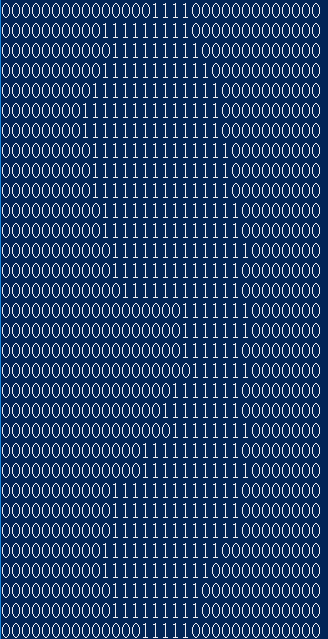
**7.**

****

**8.**

****

**9.**

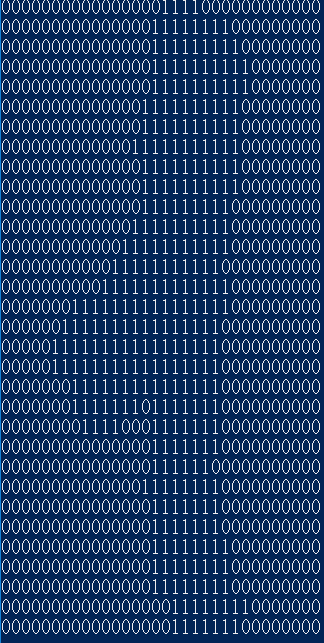
****

**4 pairs of digit types that have the highest confusion rates:**

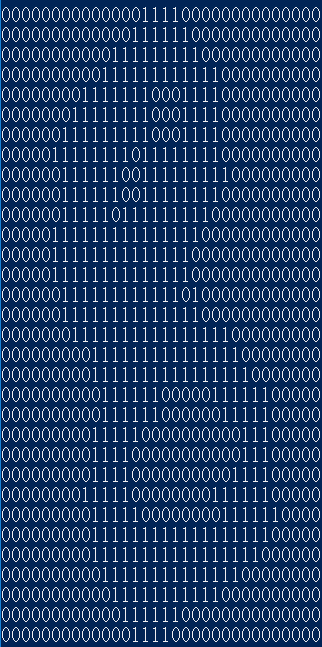
1. **1 and 8**
2. **2 and 8**
3. **3 and 9**
4. **7 and 9**

**(1).1 and 8**

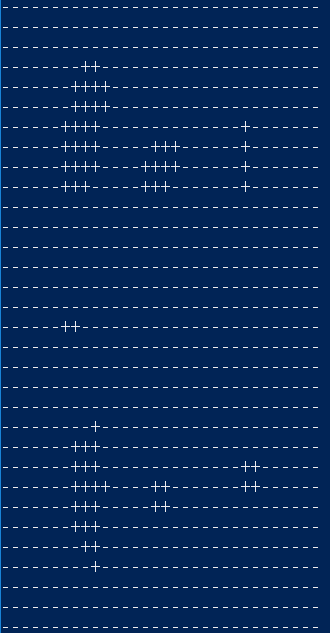
**1.**

****

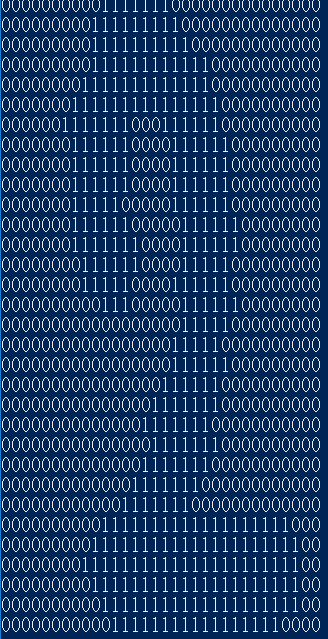
**8.**

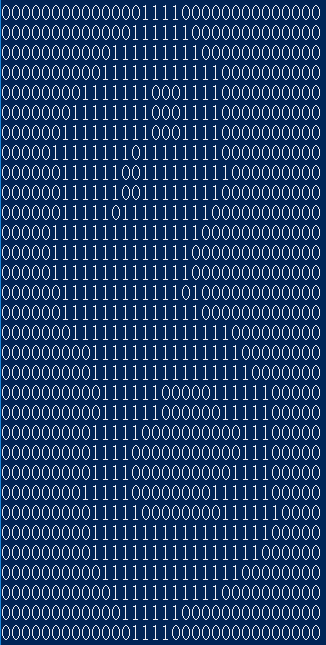
****

**Log odd ratio:**

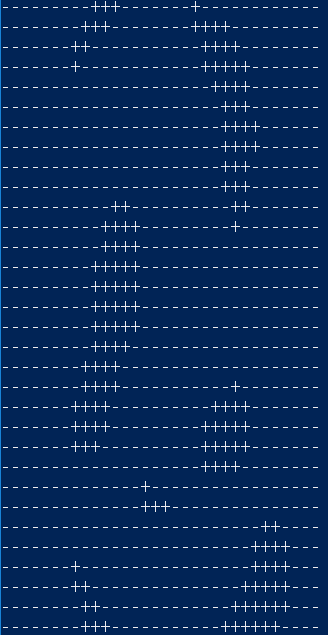
****

**(2).2 and 8**



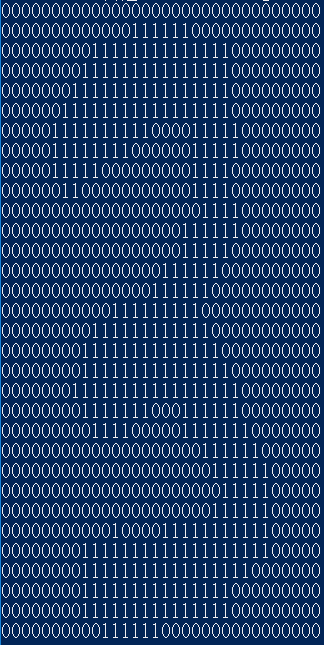


**Log odd ratio:**

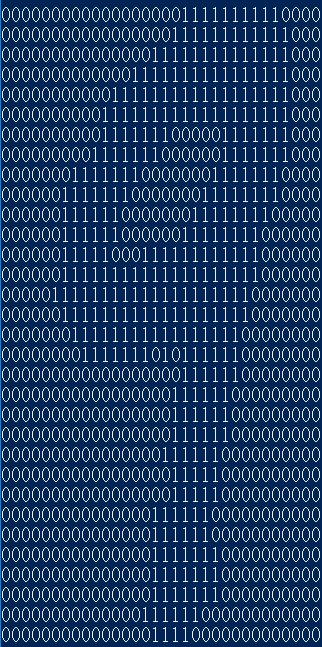


**(3)3 and 9**

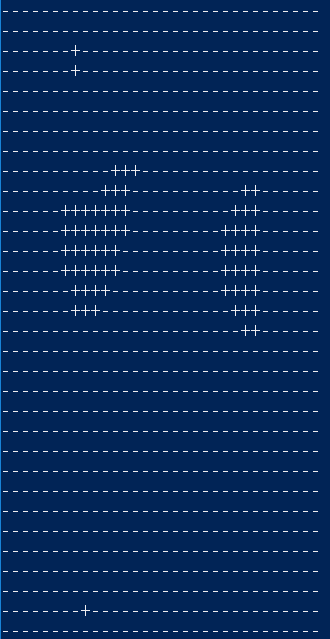
**3.**

****

**9.**

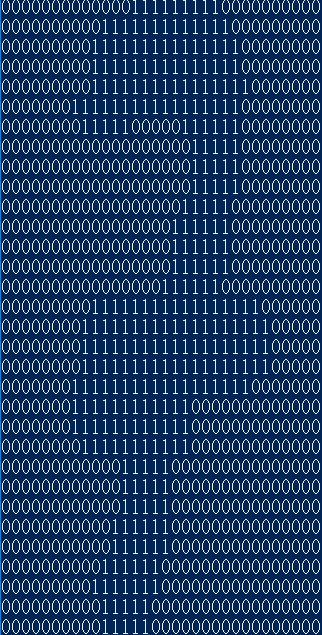
****

**Log odd ratio:**

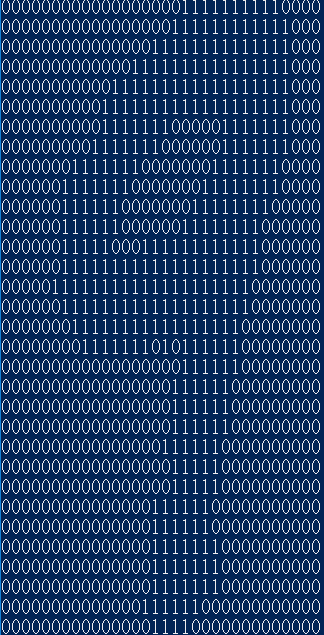
****

**(4).7 and 9**

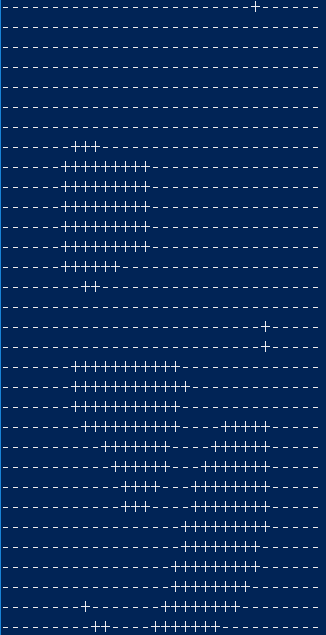
**7.**

****

**9.**

****

**Log odd ratio:**

****

**1.2**

**Implementation:**

For the pixel groups problem, I first store all the data in the format of ArrayList<ArrayList<String>>, and I then group the pixels. I select the smoothing factor as 0.1 and create a 5 dimensional array to store all the feature-value pairs.

For example, an 2\*2 disjoint patch, I will use a 10\*16\*16\*2\*2 array to store all the features(where 10 is corresponding to the ground truth digits), and 16 is the row and column, coming from 32/2(since it is disjoint), while 2\*2 is the distinct pixel level information corresponding to each feature. Note that for the 2\*2 distinct pixel level information, I can have 2^(2\*2) distinct values, which is a large overhead for storage space if the exponent gets larger.

And for the overlap 3\*3 patch, I will use a 10\*30\*30\*3\*3 array to store all the features(where 10 is corresponding to the ground truth digits), and 30 is the row and column, coming from 32-3+1(since it is overlapped), while 3\*3 is the distinct pixel level information corresponding to each feature. Note that for the 3\*3 distinct pixel level information, I can have 2^(3\*3) distinct values, which is a large overhead for storage space if the exponent gets larger.

disjoint patches:

(1)2\*2:

Correct count:409



Accuracy:92.117%



(2)2\*4:(same with 2\*2 coincidentally)

Correct count:409



Accuracy:92.117%



(3)4\*2:

Correct count:411

****

Accuracy:92.57 %



(4)4\*4: (same with 2\*2 and 2\*4 coincidentally)

Correct count:409



Accuracy:92.117%



overlapping patches:

(1)2\*2:

Correct count:410

****

Accuracy:92.34%



(2)2\*4:(same with 2\*2 coincidentally)

Correct count:410

****

Accuracy:92.34%



(3)4\*2: (same with 2\*2 and 2\*4 coincidentally)

Correct count:410

****

Accuracy:92.34%



(4)4\*4: (same with 2\*2 and 2\*4 and 4\*2 coincidentally)

Correct count:410

****

Accuracy:92.34%



(5) 2\*3: (same with 2\*2 and 2\*4 and 4\*2 and 4\*4 coincidentally)

Correct count:410

****

Accuracy:92.34%



(6)3\*2:

Correct count:409



Accuracy:92.117%



(7)3\*3: (same with 3\*2 coincidentally)

Correct count:409



Accuracy:92.117%



Discussion:

The trends I have observed for the different feature sets (including single pixels), in particular, why certain features work better than others for this task.

I found that with overlapping pixel groups, I can generally get better prediction accuracy. And I think that it is because the pixel of the digits am not only of single pixel levels, but a cluster of pixels. The pixel is highly correlated with its neighbors, so taking the nearby pixels into consider can generate better results, hence, the pixel groups generally have better performance than the single pixel.

What’s more, the overlapping patches utilize the windows with a smaller stride(the stride is one) compared to the disjoint one, which means that the overlapping one can generate more features. It also takes more information into consider. With the experiment results, I found that the overlapping patches can yield better results than disjoint ones, corresponding well with the our inference.

Running time comparison:

The running time for single pixel is the fastest, since there am only 1024 features with 2 values, so there am only 2048 different variable-value for each digit. And the second fastest is the disjoint patches, the 2\*2 one is faster than the rest, since it has only 16\*16\*2^(2\*2)=4096(space) different variable-value pair and 16\*16\*2\*2=1024 assignments should be made for each digit. The 4\*4 one is almost the same(slightly slower), since it has 8\*8\*2^(4\*4)=4194304 different variable-value pair and 8\*8\*4\*4=1024 assignments for each digit.

However, the overlapping patches take a longer time. Since there am more features and generally more variable-value pairs. For the 2\*2 one, it has 31\*31\*2^(2\*2)=15376(space) different variable-value pair and 31\*31\*2\*2=3844 assignments for each digit; while the 4\*4 one has 29\*29\*2^(4\*4)=55115776 different variable-value pairs and 29\*29\*4\*4=13456 assignments for each pair, which takes a longer time to execute.

With the calculation above, I found that if the size of the patches grow, then the execute time is going to grow linearly(almost proportional to the feature set size). However, the space needed grows exponentially.

**1-3 Extra credits:**

**Implementation:**

The extra credit problem is similar to the previous one, however, the number of the classes to be classified is reduced from 10 to 2. And the row number is changed from 32 to 70, while the column number is changed from 32 to 60. And with the parameter fine tune, I found that the smoothing factor as 0.07 yields the best result for test set accuracy.

I use the same method as naïve Bayes to calculate the prior probability with the training set first, and then use the result to get the MAP for the prediction of the test set.

We’ve tested with single pixel, and also use the features as overlapping features. Since the size of the face dataset is 70\*60, which is not as easy to be divided as 32\*32 does. And with the results from previous sections, I know that overlapping patches can yield better results than disjoint patches, so I use the various kinds of overlapping patches besides single pixel one. The result is as follows:

**Single pixel:**

Total correct count out of 150 test cases:131



The accuracy:87.33%



**Overlapping patches:**

**(1)2\*2**

Total correct count out of 150 test cases:129

****

The accuracy:86%



**(2)2\*4**(same with 2\*2 coincidentally)

Total correct count out of 150 test cases:129

****

The accuracy:86%



**(3)4\*2**

Total correct count out of 150 test cases:128

****

The accuracy:85.33%



**(4)4\*4**

Total correct count out of 150 test cases:130

****

The accuracy:86.67%



**(5)2\*3**(same with 4\*4 coincidentally)

Total correct count out of 150 test cases:130

****

The accuracy:86.67%



**(6)3\*2**(same with 4\*2 coincidentally)

Total correct count out of 150 test cases:128

****

The accuracy:85.33%



**(7)3\*3**(same with 4\*2 and 3\*2 coincidentally)

Total correct count out of 150 test cases:128

****

The accuracy:85.33%



## Part 2: Digit Classification using Discriminative Machine Learning Methods

**2.1**

Implementation:

The structure I use for the differential perceptron training is fully connection, they am 1024 input features(32\*32), so the first layer consists of 1024 neurons, and the second layer is the output layer(this is a single layer neural network), with 10 outputs(digits 0~9).

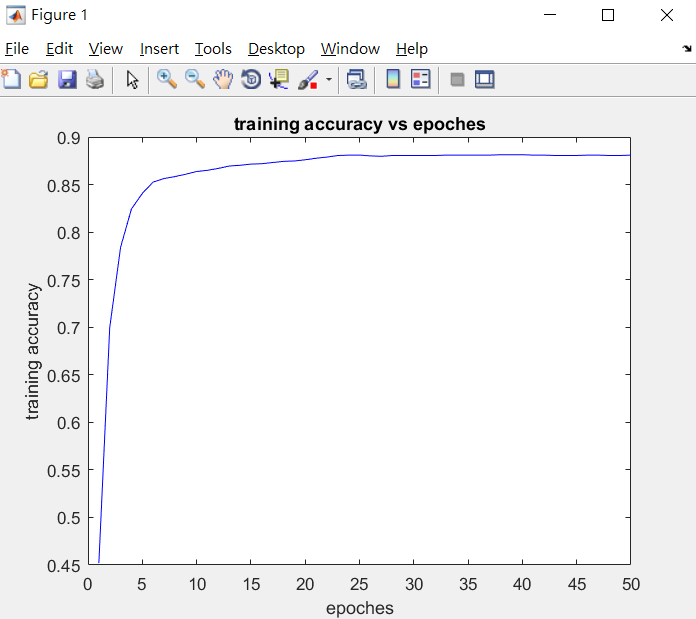
I read the dataset as an **ArrayList<ArrayList<String>> with the String consists of 1024 features. While the activation function I use is sigmoid, I introduce the nonlinearity for the prediction with this function.**

**I first set the weights to 0, and calculate the dot product of the weights and the features, and pass the result throught the sigmoid function and calculate the loss function using the labeled ground truth.**

**I then utilize it with the features to calculate the gradient for error back propagation and use the learning rate as 0.01, and normalize the gradient with the size of the data, and update the weights with the gradient obtained.**

**Training set accuracy vs epochs:**

**The accuracy is about to converge at the 26th epochs.**

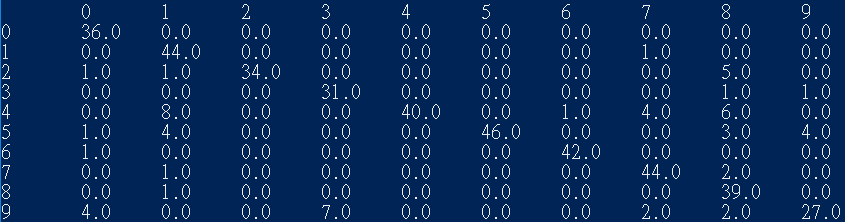
****

**weights initialized fixedly**

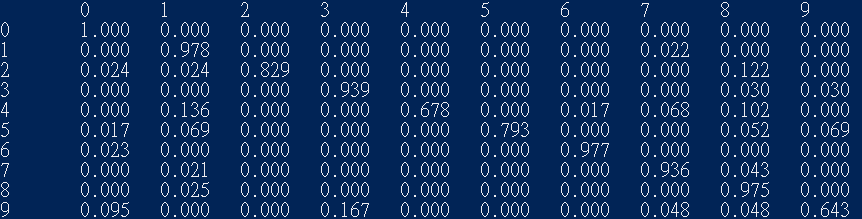
**Testing set accuracy with weights initialized fixedly:86.26%**

****

**Confusion matrix total count:**

****

**Confusion matrix:**

****

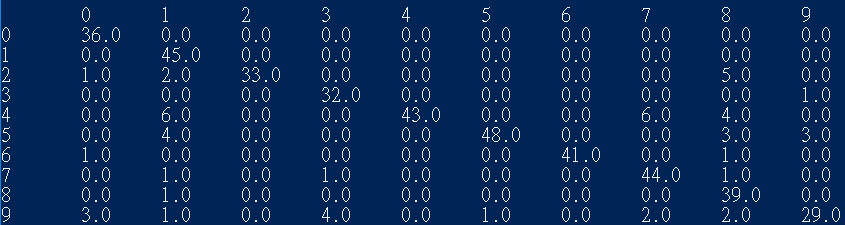
**Parameters tuning:**

I found that the accuracy converge at a epoch of 25, so I select the number of epochs as 50, which gives out the best results.

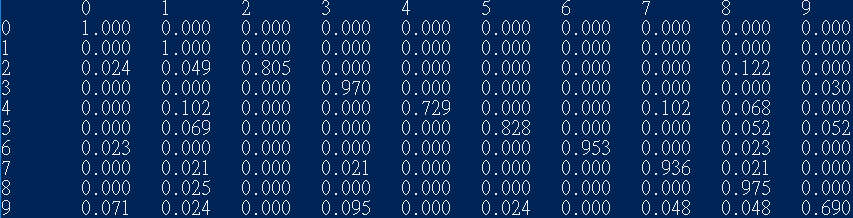
**weight initialized randomly**

**Test set accuracy with weight initialized randomly:87.84%**

**Confusion matrix total count:**

****

**Confusion matrix:**



I found that if the weight is initialized randomly, I can get slightly better results.

Parameter explanation:

The final results I found is that learning rate initially set as 0.01, and the decay rate set as , and with the bias present, and with the initialization of weights being random, and with the ordering of training examples being random, and with the number of epochs being 50 yields the best result.

I thought that with the initialization of weights being random, it is more likely to get the correct result without the result being wrongly shifted. And with the order of training examples being random, I am less likely to get a cluster of corrupted data and lead to a wrong shift of the prediction results. The bias is a constant which can be used to make the prediction result of the formula more robust.