CS440: Intro to Artificial Intelligence, Fall 2017, Homework 3

Name: Nestor Alejandro Bermudez Sarmiento (nab6)

Worked individually

Assignment 3 focuses on the Naive Bayes classification method. This report shows the performance of such method against different datasets and different feature extraction methods. I'm using Python **3.6** for my implementation.

Part 1

In this section we will take ASCII representations of digits to train our model and then evaluate its performance with a different of examples in the same format.

Implementation details

My code is split in the following pieces:

Parser: this class takes care of interfacing with the text file. It takes the paths for the files that contain the data and labels and generates tuples formed with a matrix representation of an example and its corresponding label. The class itself is a Python generator to prevent blowing up the memory (not that it will happen with the given dataset).

Feature Extractors: I created multiple of these classes that I call 'Feature extractors', they basically take an item and generate an array of features that will then be fed into the classifier. For Part 1.1 I created the **SinglePixelFeatureExtractor** which simply maps every pixel into its value (0, 1). For Part 1.2 I created the **PixelGroupFeatureExtractor** that accepts arguments to specify the dimensions of the group and whether or not they should be disjoint.

Visualizations: I use matplotlib[1] for the color maps. The implementation is rather simple and can be found in the **HeatMap** class. I use the *pcolormesh* function with a *jet* color schema because it seems to be the closest to the colors shown in the sample visualizations.

Probability Distributions: this is generic class that counts how many times a given value has appeared and provides methods to retrieve the probability and the log of the probability of a given value. It also takes care of implementing Laplacian smoothing [2].

Classifier: this class has a couple of data structures to hold the different probability distributions per class and per feature. It also provides methods to train the model and

[1] https://matplotlib.org/

[2] https://en.wikipedia.org/wiki/Laplacian_smoothing

exposes functions to classify a new example or evaluate the model performance given a sequence of examples. It also implements **Maximum a posteriori**[3] to make the predictions. Finally, it provides methods that calculate the confusion, class likelihood and log odd ratios matrices. For log odd ratios I use numpy[4] to take two matrices, divide them and then take the log of the resulting matrix.
Util: under this class you will find helpful methods and functions to print out matrices, examples and find the pair of classes we need to use for the visualizations (high confusion rates).

Part 1.1.

I tried different smoothing constants between 0.2 and 10 with increments of 0.2. The maximum accuracy was achieved using 0.2 as the constant and it seems to be inversely proportional: as the smoothing constant increased the accuracy overall decreased. It did have a local maximum around 0.8-1.0. For the sake of being pragmatic the following table does not include all the intervals but the remaining can be found in the zip file accompanying this report.

Classifier accuracy	
0.2	77.3%
0.4	77.2%
0.6	77.0%
0.8	77.1%
1	77.1%
2	76.6%
3	76.3%
4	76.2%
5	76.1%
6	76.0%

Table 1: Accuracy for different smoothing constants.

Confusion Matrix

As expected, the classifier was not perfect. Some of the examples were misclassified. The statistics of how the model behaved follows.

[3] https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation
[4] https://docs.scipy.org/doc/numpy-1.13.0/

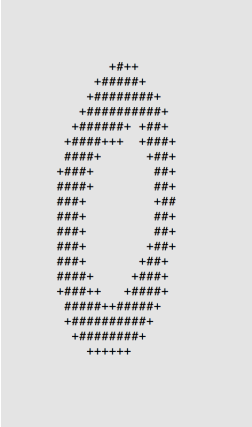
		Predicted class									
		0	1	2	3	4	5	6	7	8	9
Class	0	84.44	0	1.11	0	1.11	5.56	3.33	0	4.44	0
	1	0	96.3	0.93	0	0	1.85	0.93	0	0	0
	2	0.97	2.97	77.67	3.88	1.94	0	6.8	0.97	4.85	0
	3	0	1	0	80	0	3	2	7	1	6
	4	0	0	0.93	0	75.7	0.93	2.8	0.93	1.87	16.82
	5	2.17	1.09	1.09	13.04	3.26	68.48	1.09	1.09	2.17	6.52
	6	1.1	4.4	4.4	0	4.4	6.59	76.92	0	2.2	0
	7	0	5.66	2.83	0	2.83	0	0	72.64	2.83	13.21
	8	0.97	0.97	2.91	13.59	2.91	7.77	0	0.97	60.19	9.71
	9	1	1	0	3	10	2	0	2	1	80

Table 2: Confusion matrix using smoothing constant 0.2. Values are percentages.

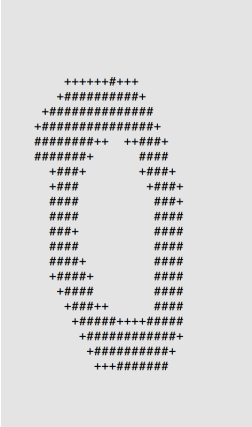
The classification rates are simply the values in the diagonal of the previous table.
Overall accuracy achieved: **77.3%**

Examples with highest and lowest probability

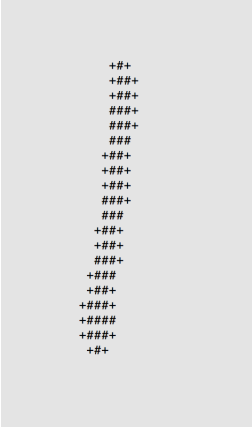
Class 0 (Highest)



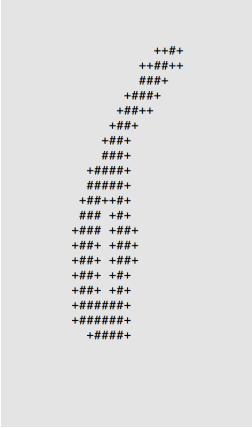
Class 0 (Lowest)



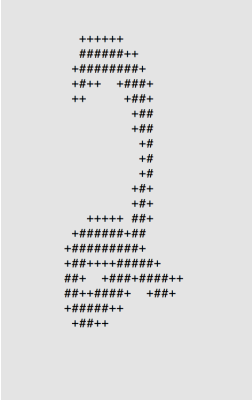
Class 1 (Highest)



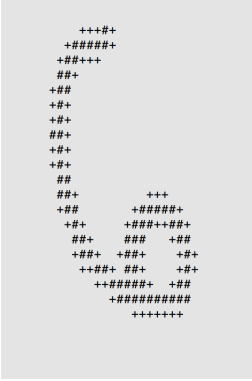
Class 1 (Lowest)



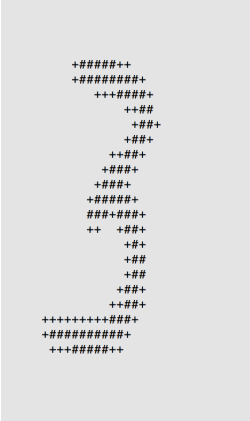
Class 2 (Highest)



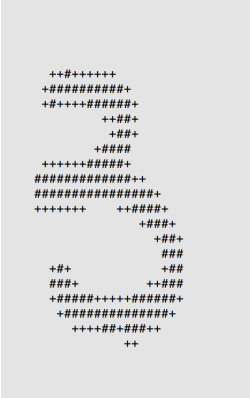
Class 2 (Lowest)



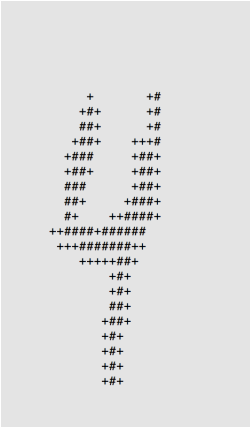
Class 3 (Highest)



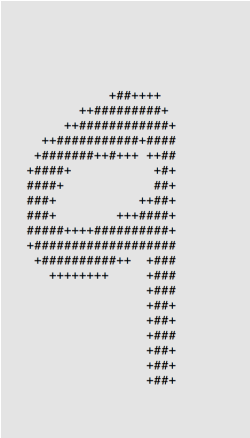
Class 3 (Lowest)



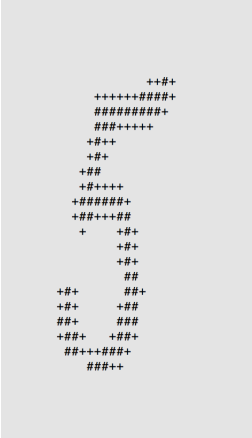
Class 4 (Highest)



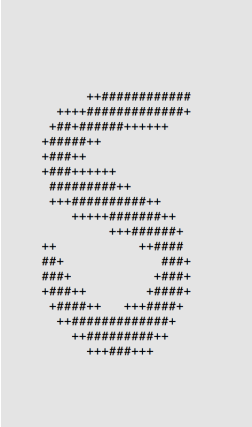
Class 4 (Lowest)



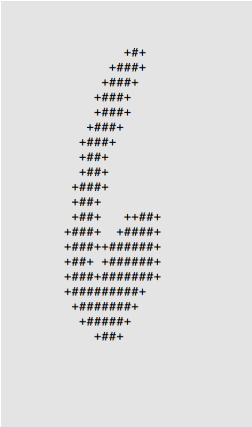
Class 5 (Highest)



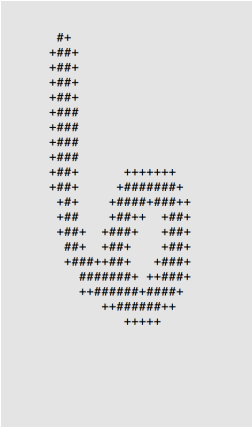
Class 5 (Lowest)



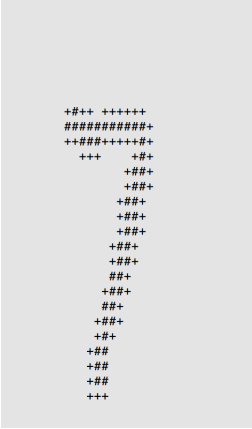
Class 6 (Highest)



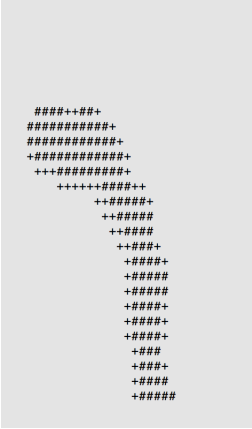
Class 6 (Lowest)



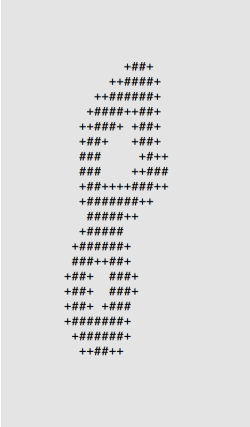
Class 7 (Highest)



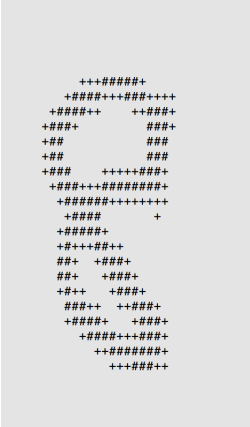
Class 7 (Lowest)



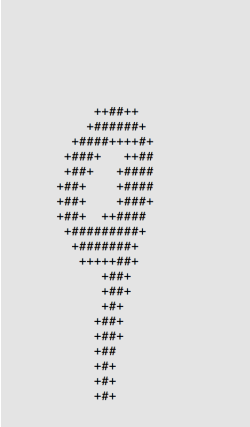
Class 8 (Highest)



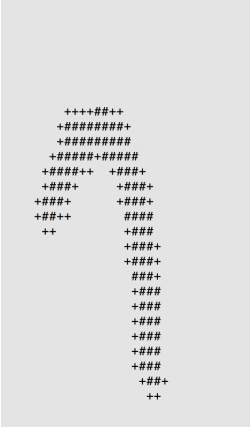
Class 8 (Lowest)



Class 9 (Highest)



Class 9 (Lowest)



Model likelihood and log odd ratios

Now lets look at the model likelihood for each of the classes with highest confusion rates.

Class 4 vs Class 9

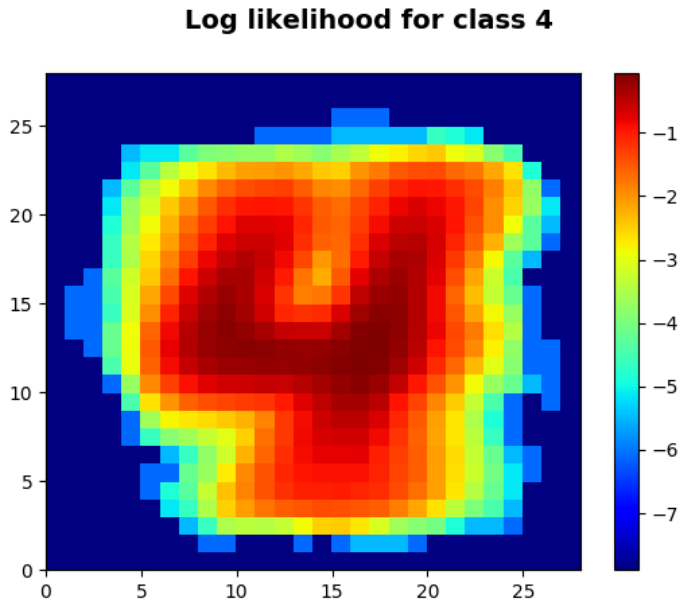


Figure 1: Log likelihood of class 4

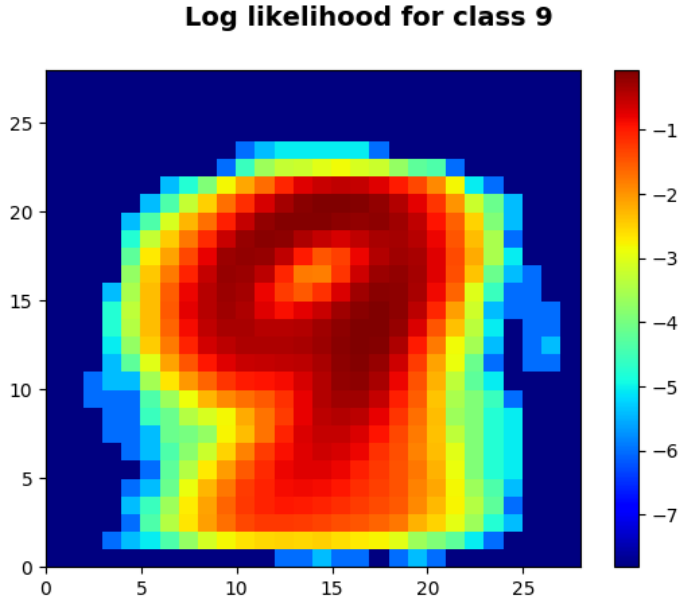


Figure 2: Log likelihood of class 9

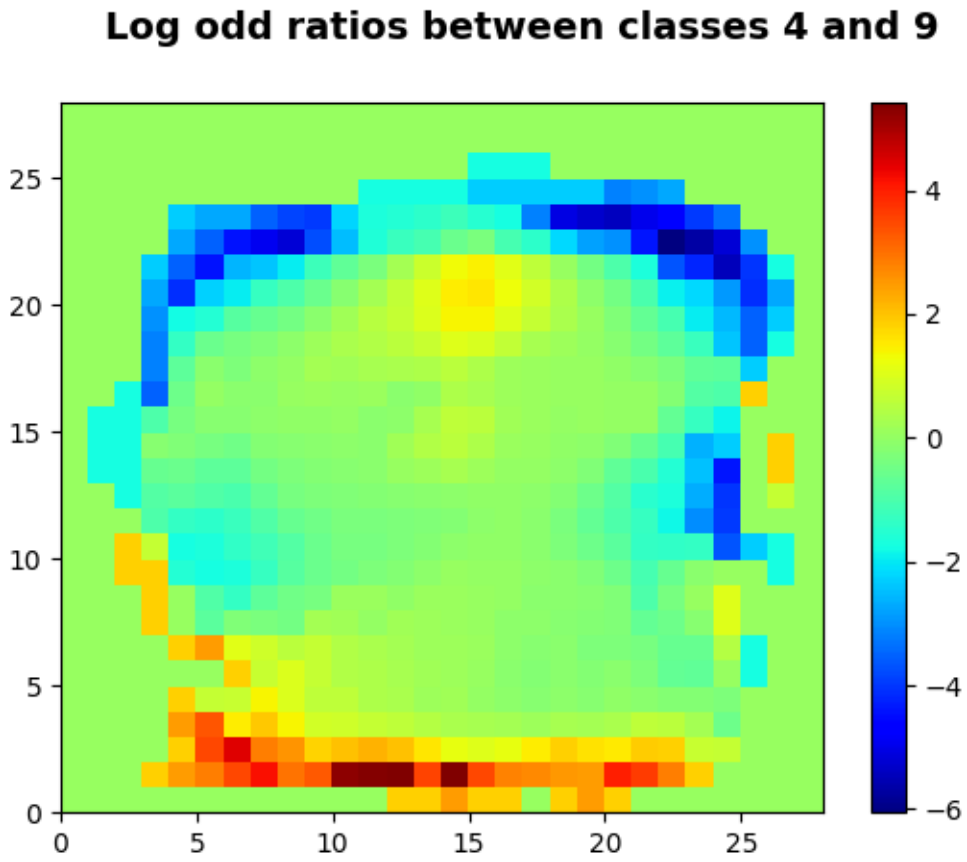


Figure 3: Log odd ratio of 4 over 9

Class 5 vs Class 3

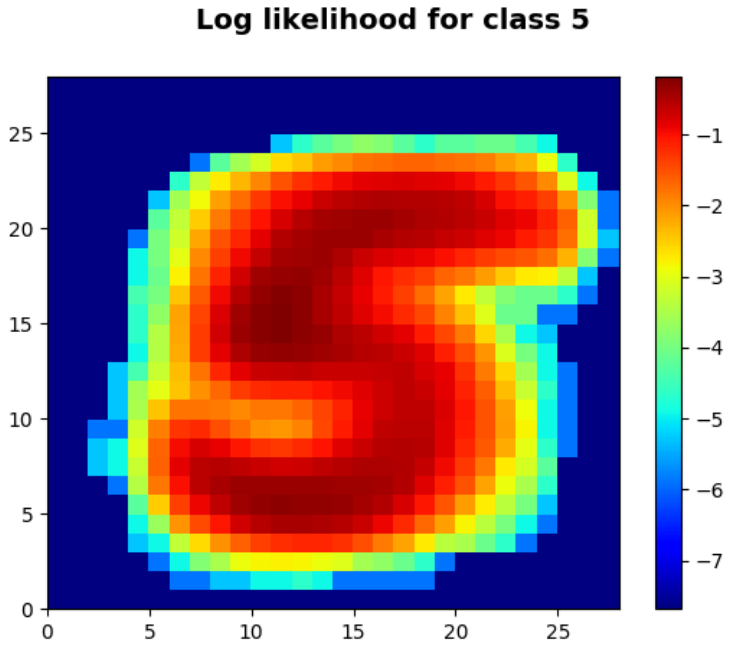


Figure 4: Log likelihood of class 5

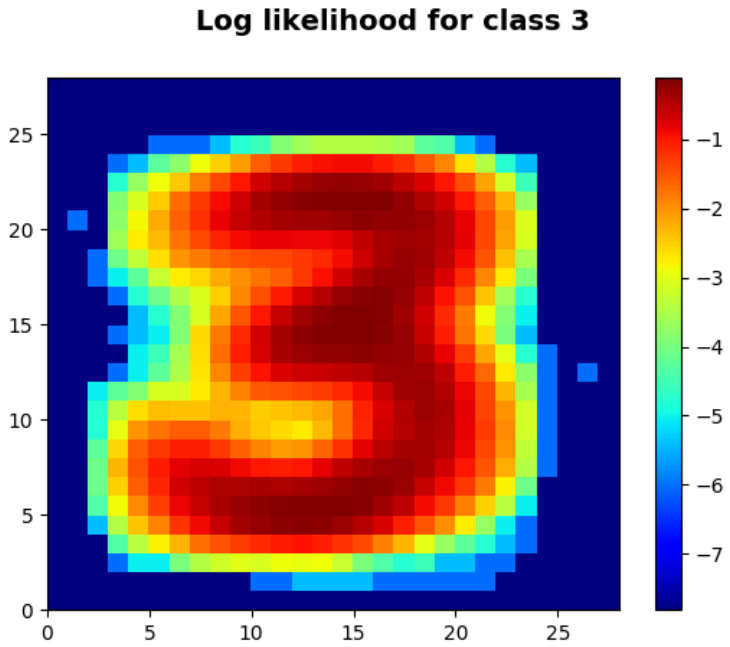


Figure 5: Log likelihood of class 3

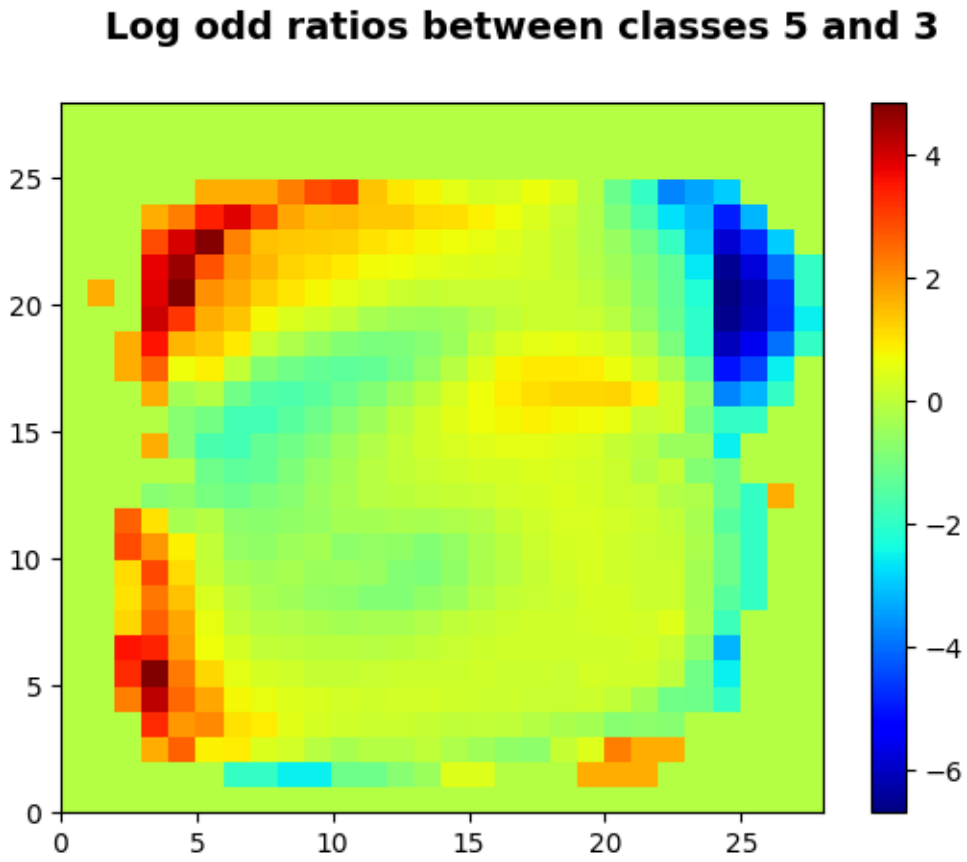


Figure 6: Log odd ratio of 5 over 3

Class 7 vs Class 9

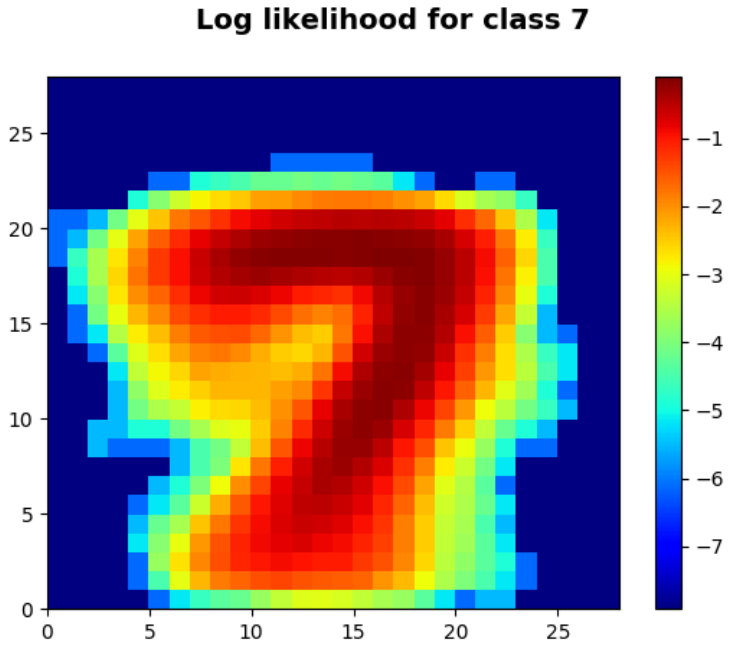


Figure 7: Log likelihood of class 7

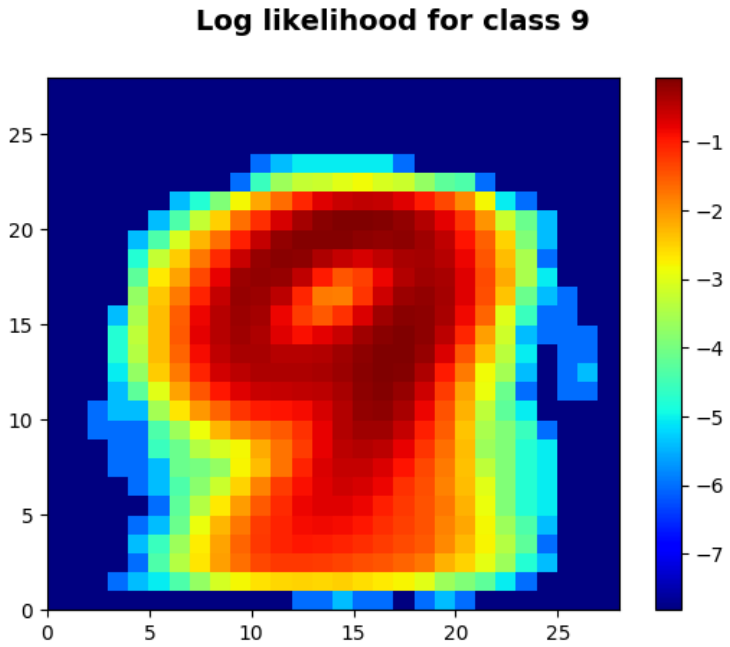


Figure 8: Log likelihood of class 9

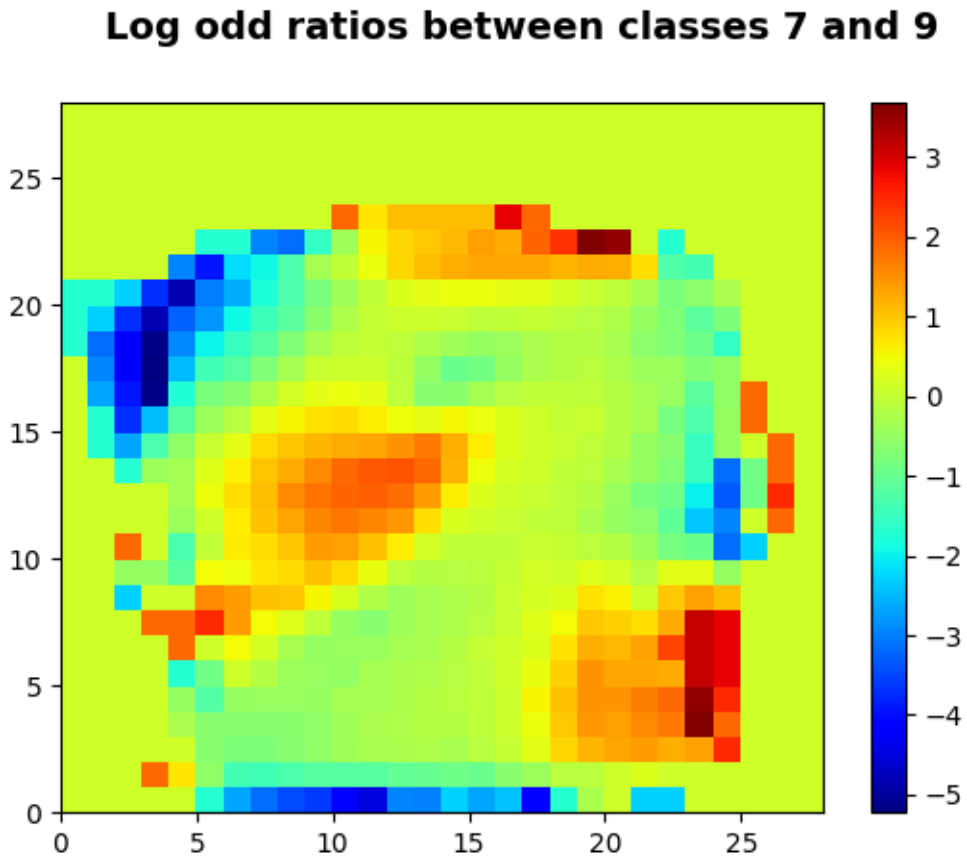


Figure 9: Log odd ratio of 7 over 9

Class 8 vs Class 3

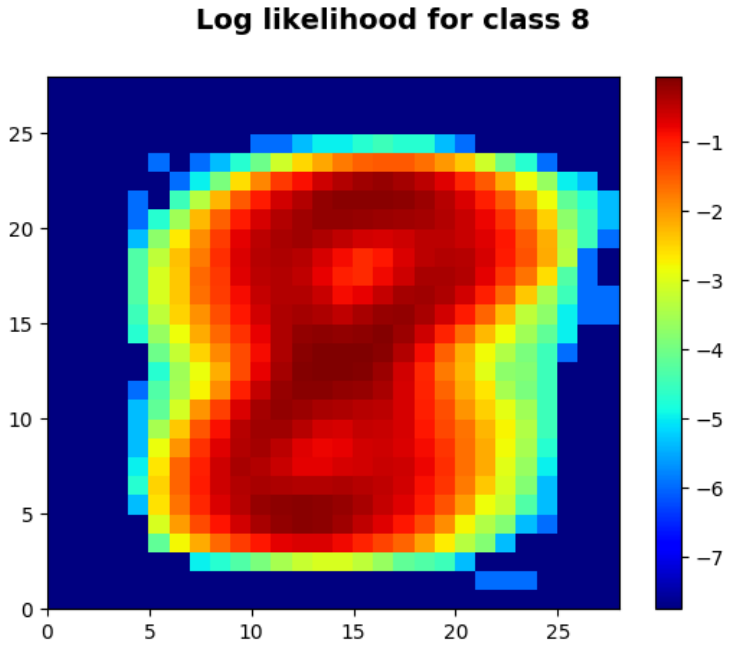


Figure 10: Log likelihood of class 8

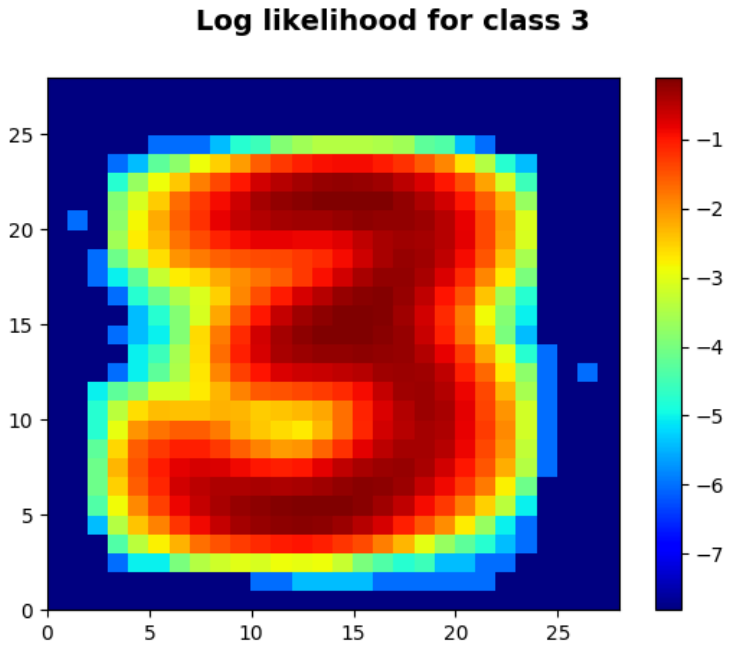


Figure 11: Log likelihood of class 3

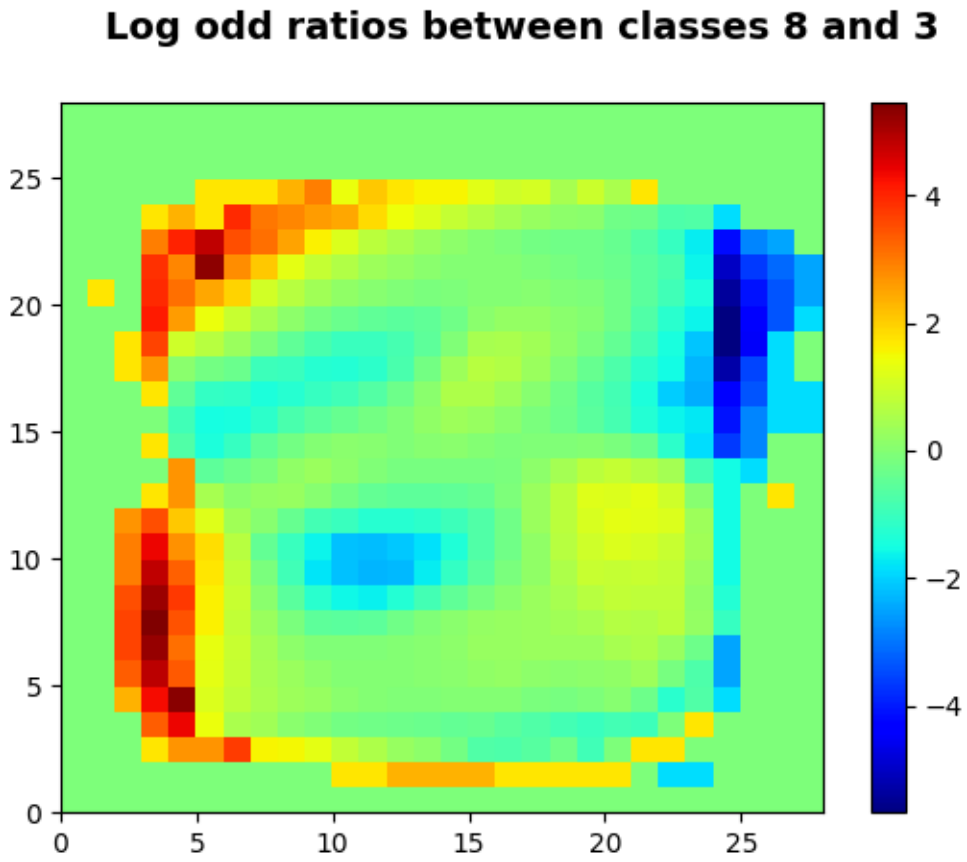


Figure 12: Log odd ratio of 8 over 3

Part 1.2.

For this part I set the smoothing constant to 0.001 because, based on part 1.1, it seems like the lower the value the better the accuracy. For the selected group we would have a small matrix, the feature is the result of traversing the matrix from left to right, top to bottom and create a string with the binary values in that order.

Disjoint groups of pixels

I trained my classifier using disjoint groups of pixels of size 2x2, 2x4, 4x2 and 4x4. Here are the results.

		Measure			
		Accuracy	Train time	Eval time	Feature count
Size	1x1	77.4%	13.08s	11.21s	784
	2x2	86.7%	5.68s	3.52s	196
	2x4	89.1%	3.55s	1.95s	98
	4x2	89.6%	3.9s	2.04s	98
	4x4	87.3%	3.1s	1.23s	49

Table 3: Summary of performance for different pixel group sizes.

Overlapping groups of pixels

First I tried multiple group size of overlapping group size and measured the accuracy, training and evaluation time. The results are presented in the following table.

		Measure			
		Accuracy	Train time	Eval time	Feature count
Size	1x1	77.4%	13.02s	11.89s	784
	2x2	89.1%	18.98s	14.14s	729
	2x4	90.2%	24.35s	15.05s	675
	4x2	90.8%	26.70s	15.97s	675
	4x4	91.2%	34.61s	17.09s	625
	2x3	90.4%	21.26s	14.12s	702
	3x2	90.4%	23.87s	15.12s	702
	3x3	91.6%	27.01s	16.01s	676

Table 4: Summary of performance for different pixel group sizes.

Now lets look at some of the trends on these results.

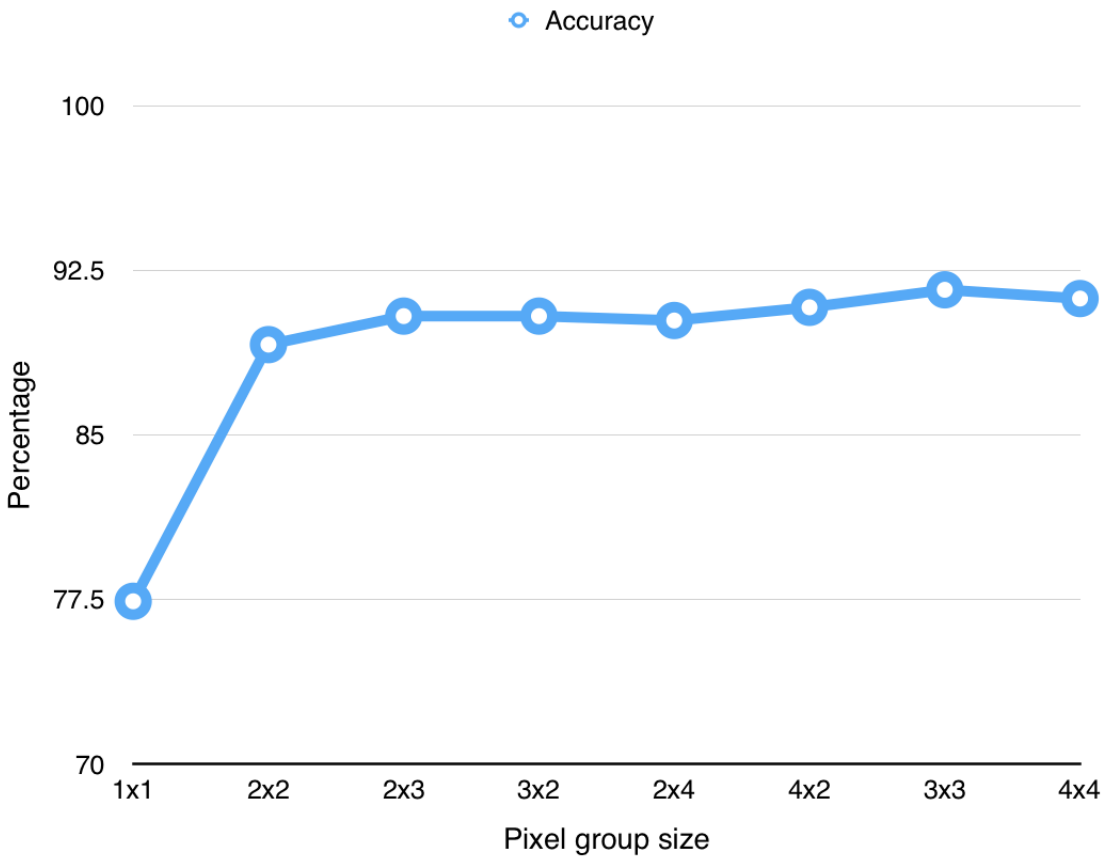


Figure 13: Line chart for the accuracy of different group sizes.

The group sizes were sorted increasingly based on the number of pixels in the group. Overall, it seems that the accuracy is positively correlated to the size of the group. Which makes sense but if we were to keep increasing the size we may end up over-fitting our data. For our particular data, 3x3 seems to be a good size.

Now, lets see the training time.

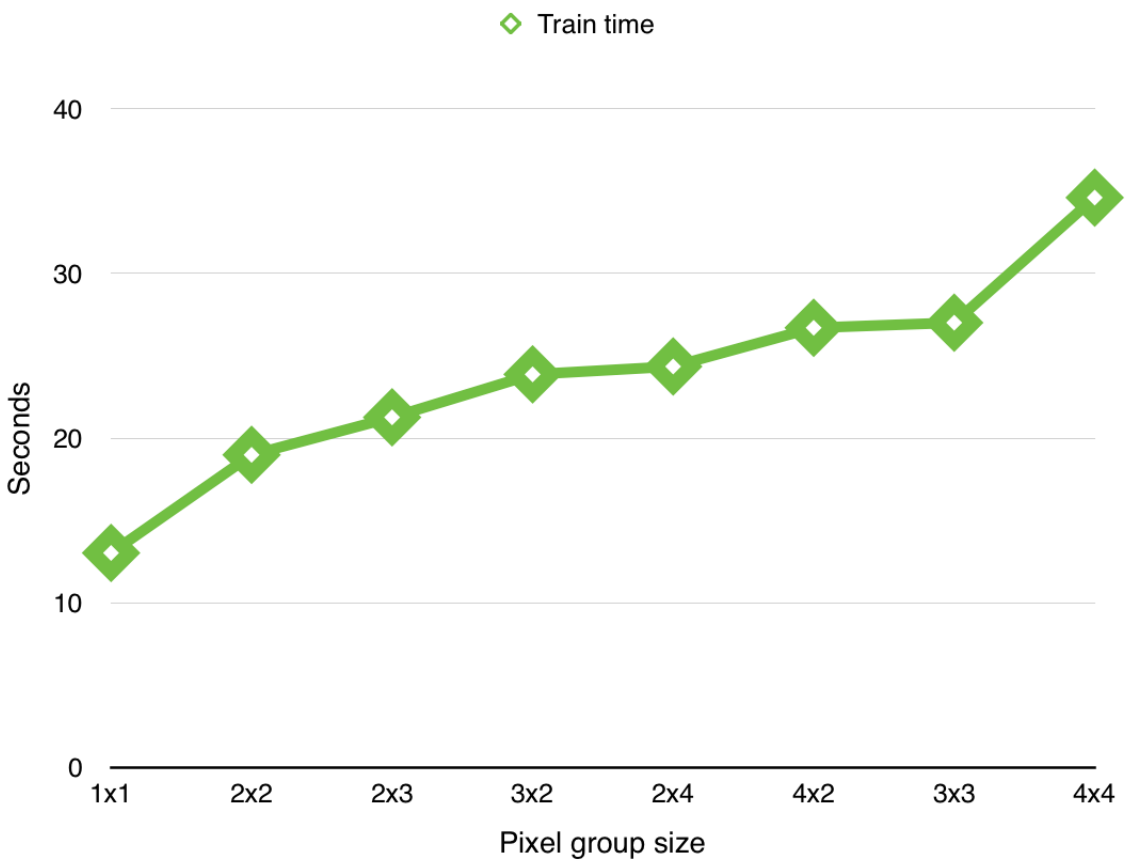


Figure 14: Line chart for the training time of different group sizes.

Again, it seems to be a positive correlation between group size and the time it takes the model to train. This makes sense because, as explained in the assignment description, when using a nxm group size we would have 2^{nm} different values.
Finally, this is how the test time is impacted.

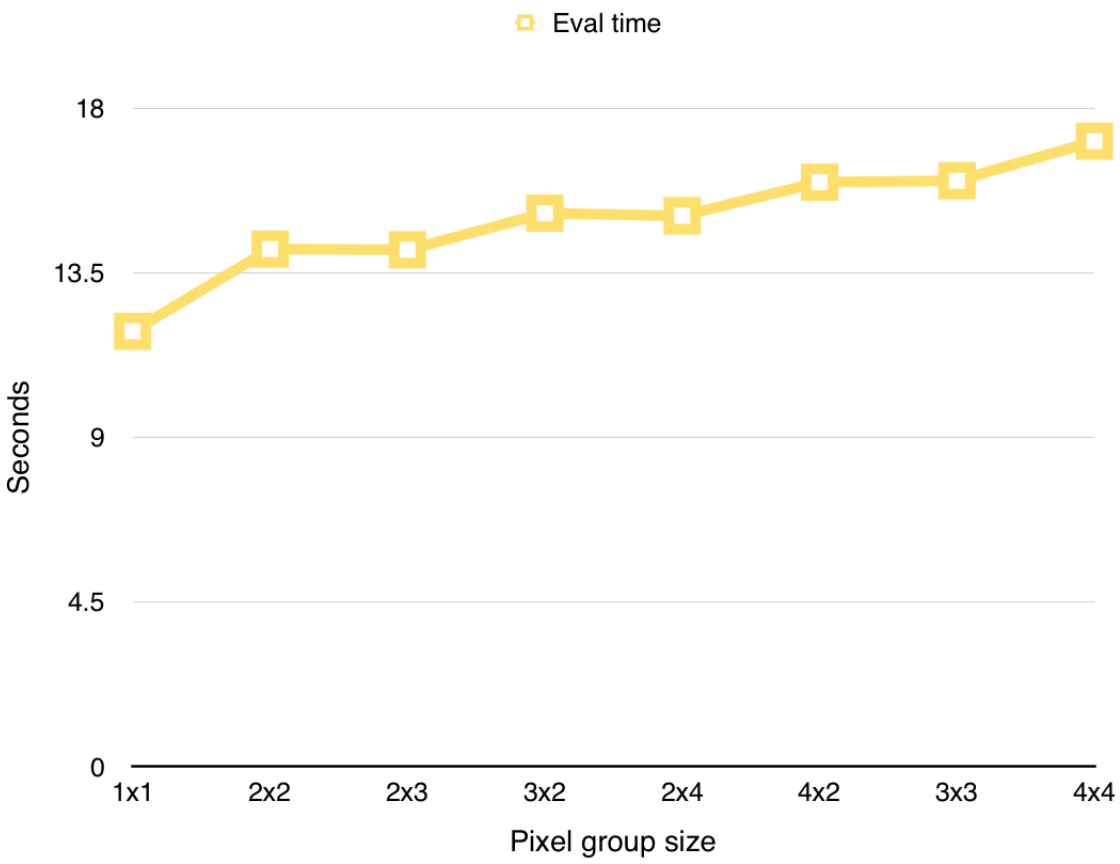


Figure 15: Line chart for the evaluation time of different group sizes.

The evaluation time does increase as the size of the group increased because there are more features to evaluate but note that its growth seems to be slower, probably linear, compared to the previous two measurements.

In all cases, going from single pixel to 2x2 represents a very significant improvement.

Why this works?
Having a bigger group size means that the feature can encode information like the convexity of a portion of the image (the bottom part of digit 8 for example) and the existence of line segments (strokes for digits 4 and 7 for instance). This creates more ways to differentiate between similar digits.

Note: more details on each of the trials can be found under the *results* folder. This includes the most and least prototypical digits per class and the confusion matrix.

Disjoint vs Overlapping comparison

Now lets see how performance differs depending on which type of grouping we use.

This is a plot of the accuracy for both types:

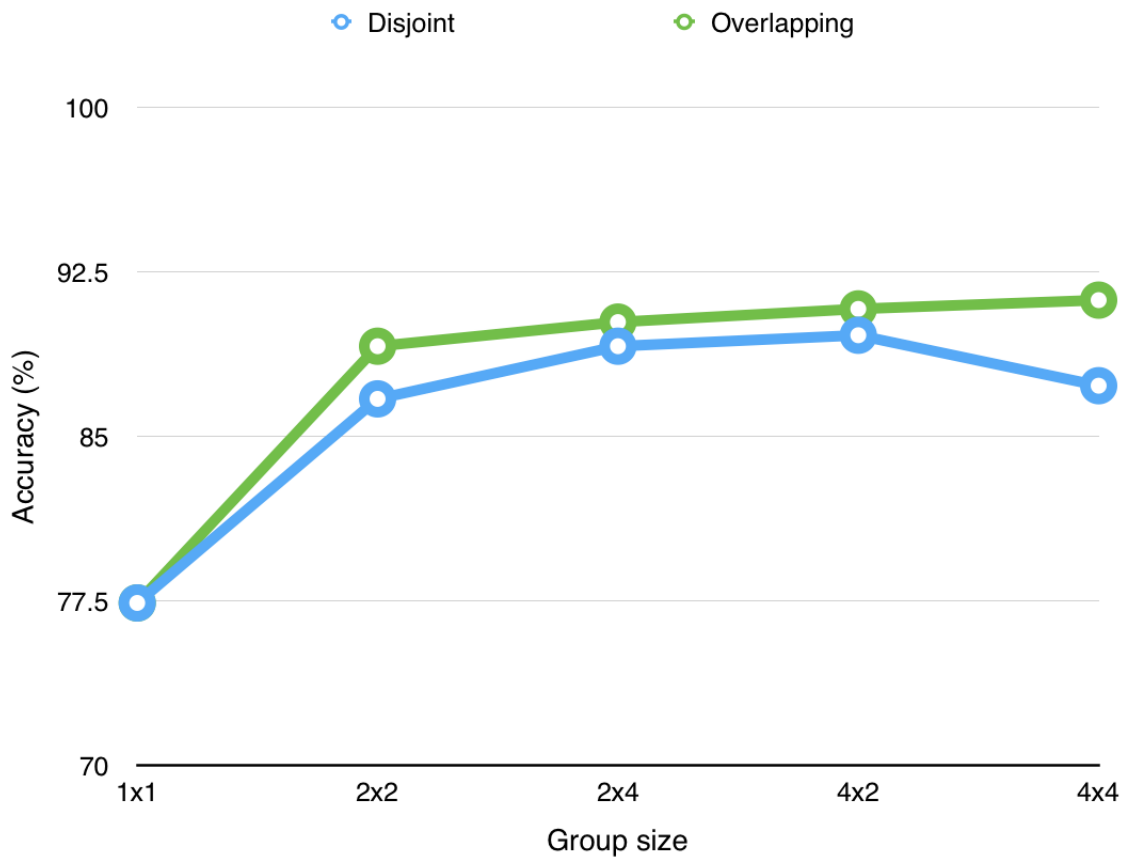


Figure 16: Line chart comparing accuracy of overlapping vs disjoint grouping.

We can see that using overlapping groups outperforms disjoint groups on every size. Not by much though. I think this happens for two reasons: (1) by overlapping we are relaxing the independence condition of Naive Bayes and (2) the number of features is greater.

And now lets look at the training time.

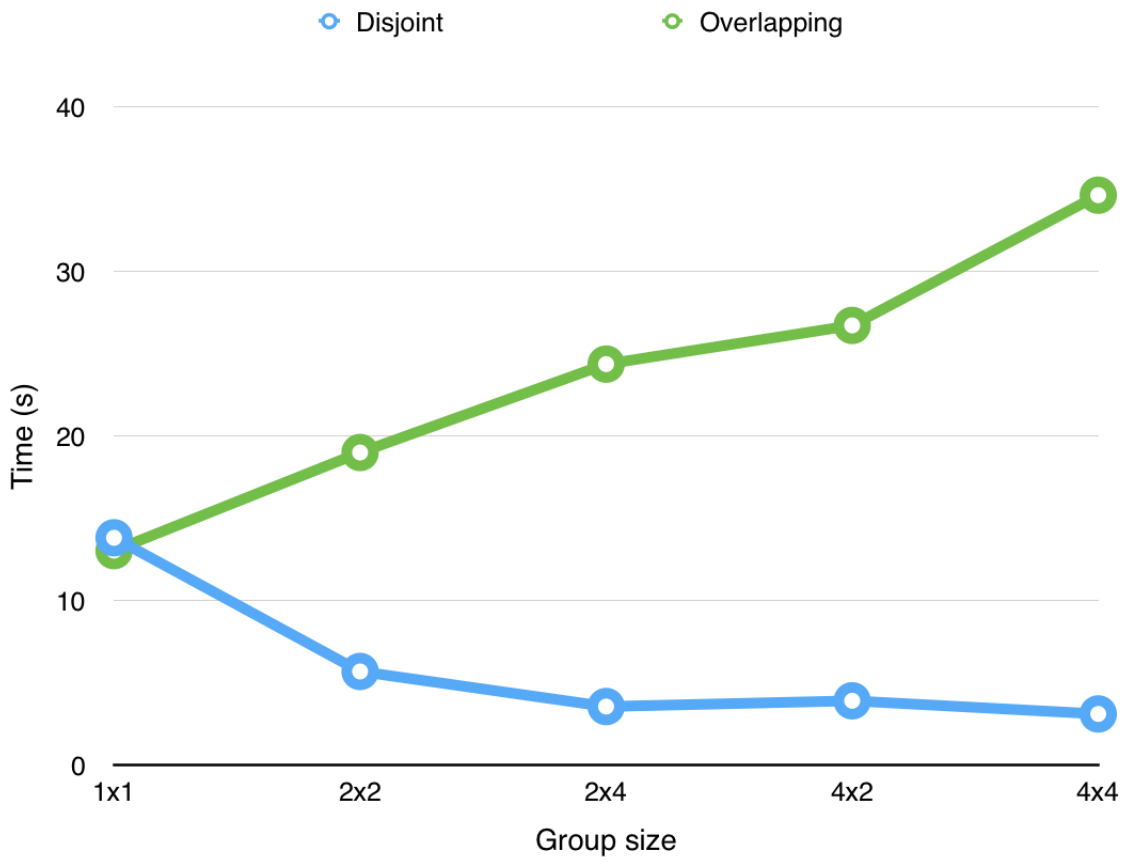


Figure 17: Line chart comparing training time of overlapping vs disjoint grouping.

As expected, the training time for the disjoint groups is way lower than for the overlapping ones. The reason is that the greater the size of the group, the less disjoint groups there will be.

At the end of the day, it is always a trade-off between speed and accuracy. If I had to choose one for this dataset I'd chose the disjoint ones because it runs as much as 10 times faster and the accuracy is still within acceptable ranges.

Time vs feature count

Finally, lets have a look at how the training time relates to the number of features.
For overlapping groups:

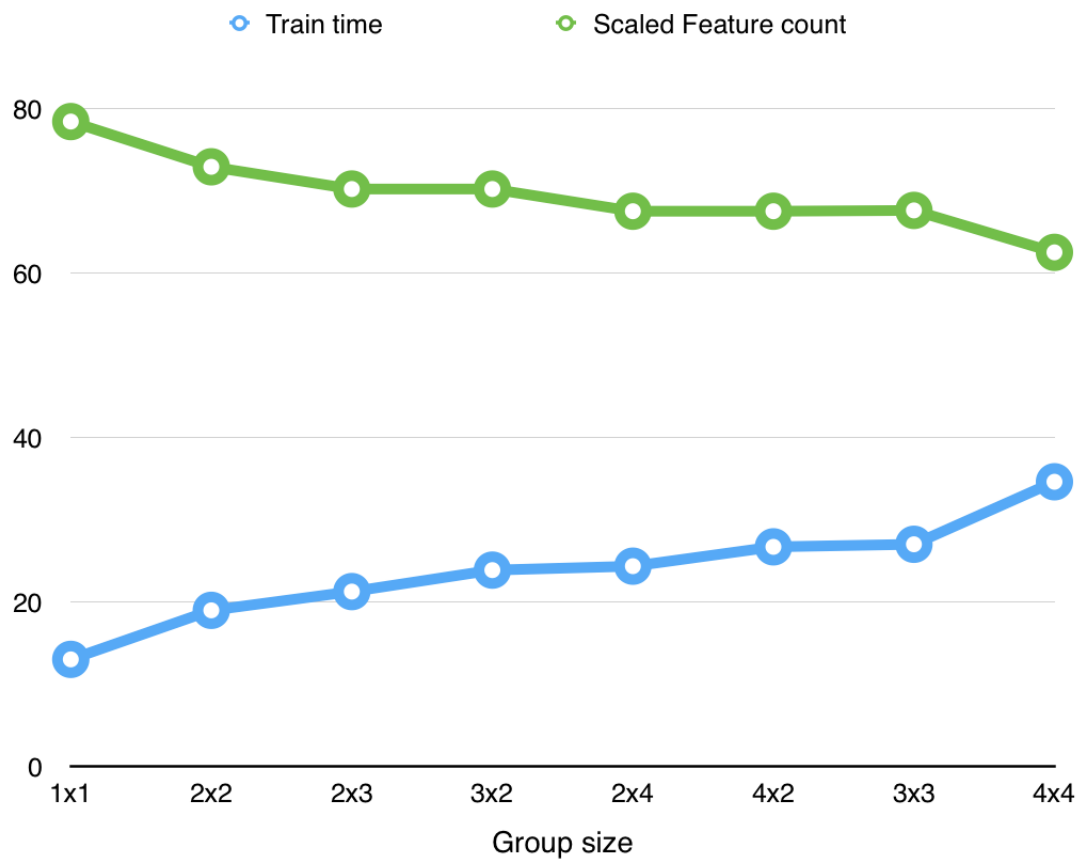


Figure 18: Line chart showing the relation between training time and feature count. Note: feature count has been scaled by 0.1 so that the chart will be more readable.

The reasoning for this relation is that even though the number of features is decreasing, it doesn't decrease too fast and, in the meanwhile, the information each feature encodes is greater so it takes longer to create the feature.
For disjoint groups:
For overlapping groups:

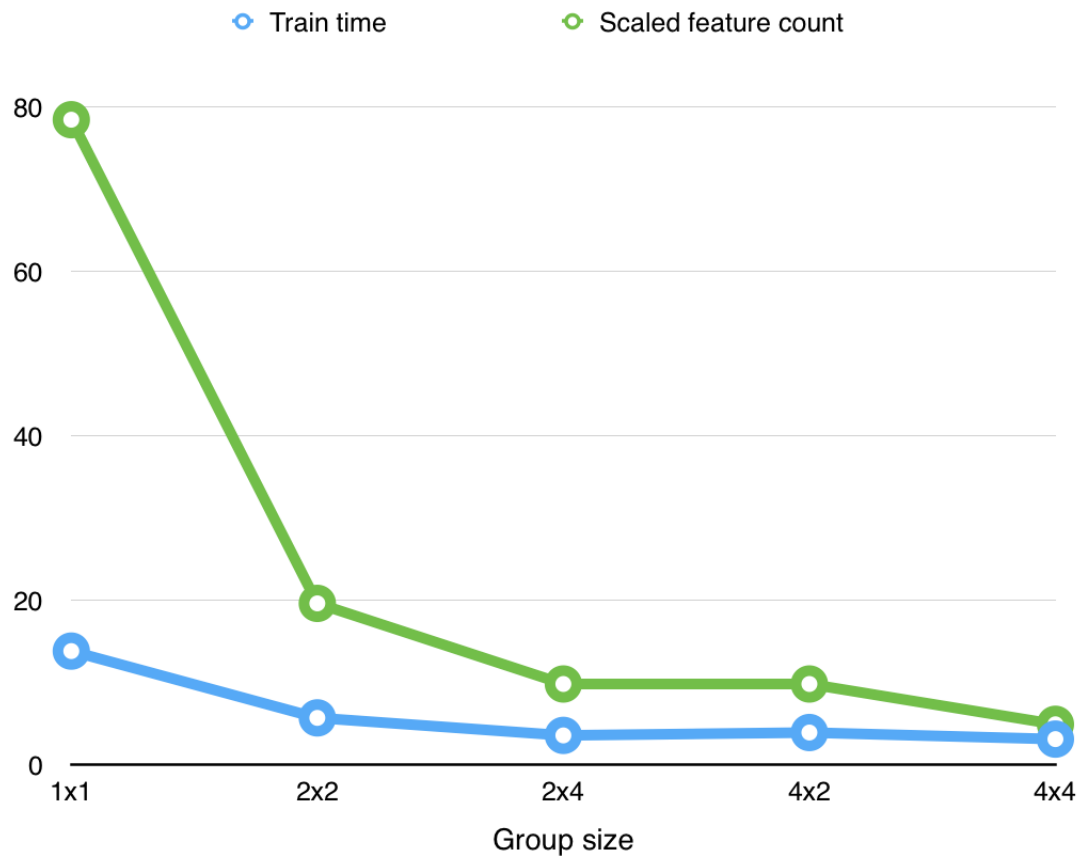


Figure 19: Line chart showing the relation between training time and feature count. Feature count has been scaled by 0.1 so that the chart will be more readable.

Just like for overlapping groups, the information each feature encodes is greater but in this case the number of features decreases very quickly compared to the overlapping case and therefore the training time decreases.

Extra Credit

Ternary feature

For the first extra credit I implemented the recommended ternary features.
For this I created a new class called **TernaryFeatureExtractor** which uses the three different values found on each example (+, # and space). It also support pixel grouping. The statistics shown here are the result of using a 3x3 pixel group size and a smoothing constant of 0.001. These values were chosen based on the results obtained from the previous sections.

I tried two variations of my **TernaryFeatureExtractor**:
Ternary only: when traversing the pixel group I just take the pixel type and concatenate to create the feature.

Using this feature I could never get the accuracy over 89%, tried the same group sizes from part 1.2 and different smoothing constants but it never went higher.
Ternary and binary: instead of just using the new ternary features I use double amount of features and add back the binary features from part 1.1 and 1.2. Using this finally got the accuracy to 90.2%.

The performance was better than part 1.1 but not better than part 1.2. This is the confusion matrix.

		Predicted class									
		0	1	2	3	4	5	6	7	8	9
Class	0	96.7	0	0	0	0	0	1.11	0	2.22	0
	1	0	96.3	0.93	0	0.93	0	0.93	0	0.93	0
	2	0.97	0	91.26	0.97	0	0	1.94	0.97	3.88	0
	3	0	0	1	92	0	3	0	1	1	2
	4	0	0	0	0	94.39	0	1.87	0.93	0	2.8
	5	1.09	0	1.09	4.35	0	86.96	0	1.09	4.35	1.09
	6	1.1	1.1	0	0	0	5.49	91.21	0	1.1	0
	7	0	2.83	3.77	0	0.94	0.94	0	79.25	1.89	10.38
	8	0.97	0	2.91	7.77	0.97	0.97	0	0.97	85.44	0
	9	1	0	0	2	3	1	0	0	4	89

Table 5: Confusion matrix using smoothing constant 0.001. Values are percentages.

Face Data

For the Face Data, I created a new **Parser** class that reads the files (data and labels) and generates tuple ready for the classifier training. I decided to first try the best setup found for the previous sections: smoothing constant=0.001 and pixel group size of 3x3.

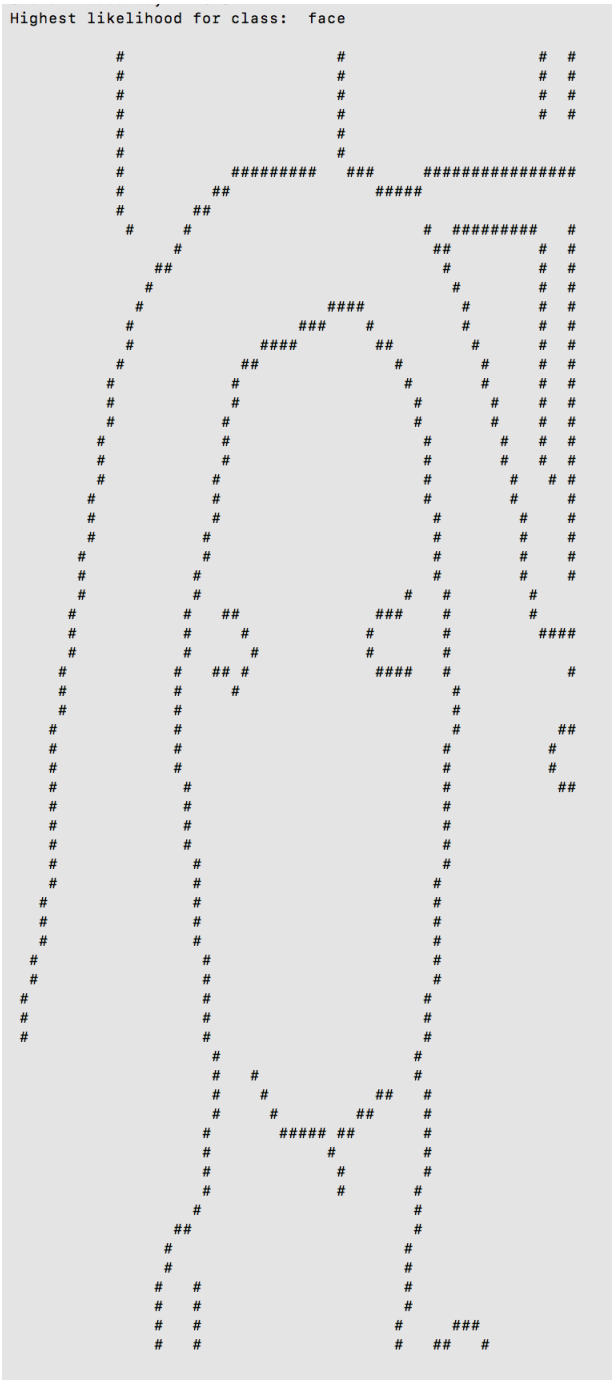
Turns out that with this setup I could achieve **96%** accuracy on the face data. This is the confusion matrix.

		Predicted class	
		face	non-face
Class	face	93.15	6.85
	non-face	1.3	98.7

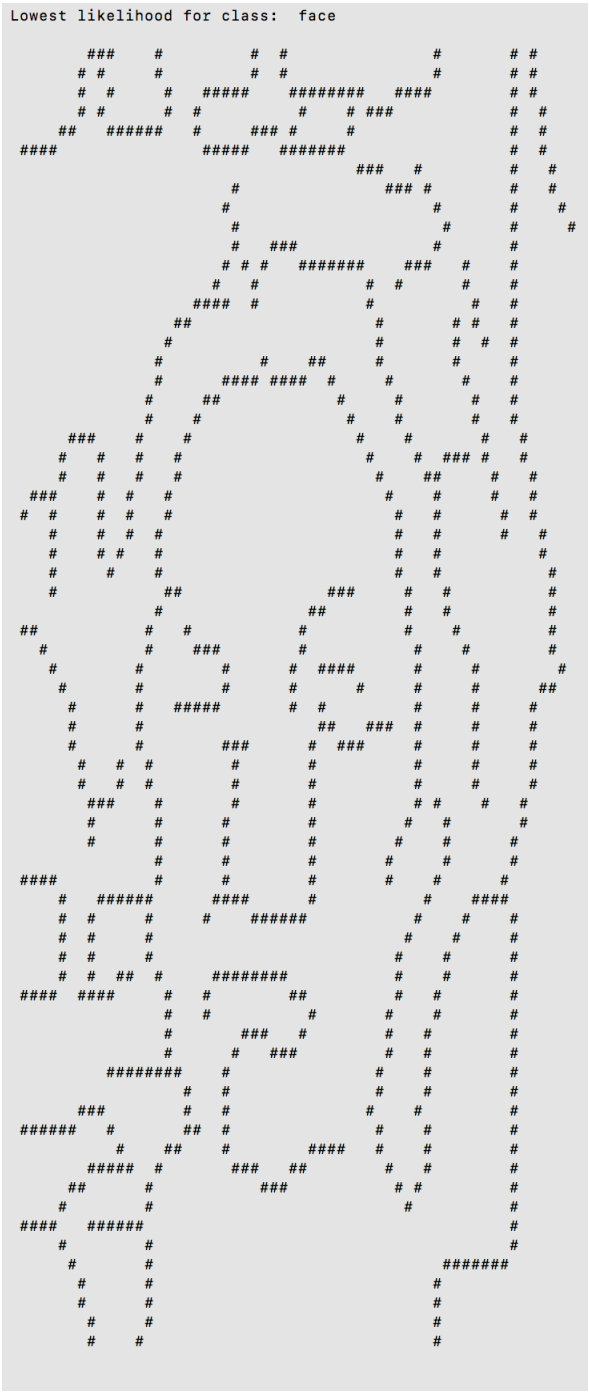
Table 6: Confusion matrix for Face Data.

And these are the most and least prototypical examples from the test data:

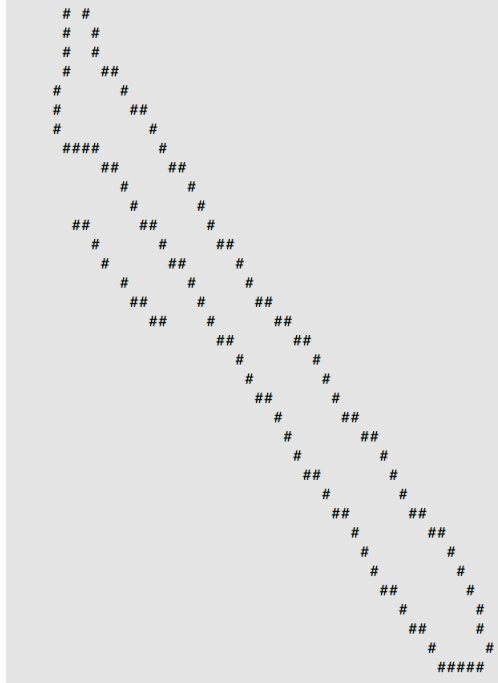
```
Highest likelihood for class:  face
```



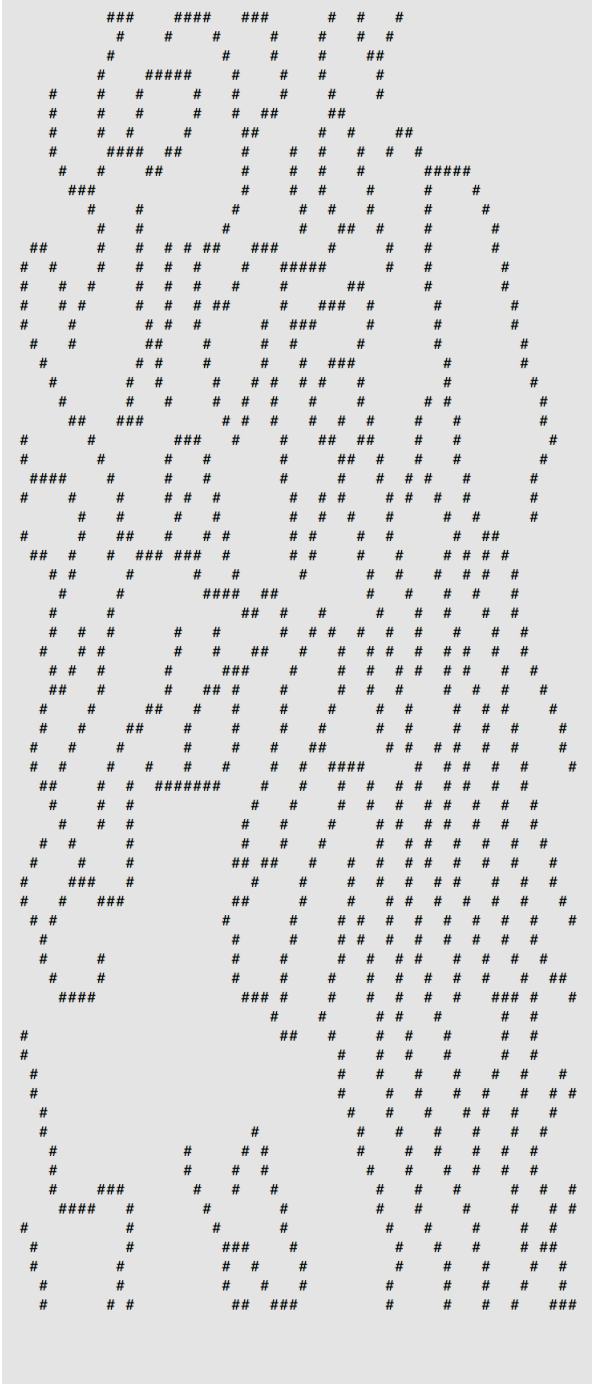
```
Lowest likelihood for class:  face
```



```
Highest likelihood for class:  non-face
```



```
Lowest likelihood for class:  non-face
```



Part 2

This part deals with classifying ASCII representations of spoken words, first Hebrew for yes and no and then digits.

SingleBlockFeatureExtractor is almost identical to **SinglePixelFeatureExtractor** used in Part 1, except for the character-binary mapping.

Part 2.1

Most of the details for this are borrowed from the work for Part 1. Basically, the only changes where on how to process the input files and feed the features into the classifier.

Implementation details

The classifier is exactly the same as the one used for Part 1, except for some some unused code that was removed.

I created a new parser class to read from the input files and generate the (features, label) tuples in the format expected by my classifier.
This parser class takes care of ignoring the 3 blank lines separating each entry in the data files.

My **ProbabilityDistribution** and **SingleBlockFeatureExtractor** classes are also the same as for Part 1.

Confusion matrix

		Predicted class	
		yes	no
Class	yes	96	4
	no	4	96

Table 7: Confusion matrix for Hebrew yes/no data.

Overall accuracy : **96%**

For this dataset it seems like the smoothing constant is not so relevant. I tried values between 0.001 and 5 and the accuracy remaining at 96%.

Part 2.2

Implementation

Again, just like in the previous section, I used the same classifier and implemented the appropriate parser class (**DigitAudioParser**).

Confusion matrix

		Predicted class				
		1	2	3	4	5
Class	1	87.5	0	12.5	0	0
	2	0	100	0	0	0
	3	0	0	100	0	0
	4	0	37.5	0	62.5	0
	5	0	0	12.5	0	87.5

Table 8: Confusion matrix for audio input of digits between 1 and 5. Values are percentages

Overall accuracy: **87.5%**

Full output can be found under the results folder and it contains the most and least prototypical examples for both Part 2.1 and 2.2.

Note: all the **Parser** and **FeatureExtractor** classes have main blocks that perform a few tests to ensure the output of the functions are the ones I was expecting. To execute it you just need to run that particular file.

Extra Credit

Three possible extra credits were available for implementation. In this section I'll explain the implementation details of all 3, the corresponding confusion matrix and overall accuracy.

Unsegmented data for training

For this extra credit the trickiest part was to figure out how to segment the data on each file. For this I created a new Parser file called **UnsegmentedDataParser**, what it does is:

1. converts the content of the file into a big matrix
2. takes the transpose of that matrix
3. over the transpose of the matrix it iterates over each row and takes the percentage of high energy pixels in it.
4. when that percentage is lower than a given threshold and the percentage on the next row is greater than the previous row then that row is chosen as the first real data row (which will be part of the first spectogram).

5. once the first row has been chosen, the 80 rows from that point forward are split into 8 sections of size 10.

6. the parser also takes the name of the file and splits it to map each 0 or 1 to the 'y' or 'n' label for the corresponding data.

As for the threshold mentioned before. I decided to use $\frac{23}{25}$ just because it seemed accurate based on the already segmented data we used for Part 2.1.

Note that the provided dataset did not include a testing set for this so I decided to use the same testing samples provided for Part 2.1. Which turns out to be convenient to compare the results with the ones obtained previously.

As with the previous sections, I tried different smoothing constants but settled with a value of **2** because it yielded the best accuracy.
I also tried different block group sizes (same sizes as for part 1.2) but they didn't improve the accuracy so I decided to stick with single block features.

The maximum accuracy achieved was **92%** and this is the confusion matrix obtained:

		Predicted class	
		yes	no
Class	yes	92	8
	no	6	94

Table 9: Confusion matrix for Hebrew yes/no data using the unsegmented data.

As a final note, the training time was greater than the previous scenarios. It took **505ms** to train. Which makes sense considering the additional work of figuring out the segmentation and the greater number of examples.

Fisher's Linear Discriminant Analysis

As suggested, I implemented the Fisher's Linear Discriminant as my second classifier. I understood the theory behind it by following Prof. Veksler[5] from Western University in Canada.

Most of the linear algebra operations, like taking a dot product or the inverse of a matrix was done using NumPy[6].
The implementation itself lives under the **FisherLDAClassifier** class and it has the same methods and function signatures as my Naive Bayes classifier to easily switch from one to another.

I confirmed my implementation with the example described in Prof. Veksler slides but when I tried to train it with the yes/no dataset it would not work because the *Within class scatter matrix* would be singular and I could not take the inverse.

[5] http://www.csd.uwo.ca/~olga/Courses/CS434a_541a/Lecture8.pdf
[6] http://www.numpy.org/

After further reading (here and here) I learned that it is a rather common problem when using Linear Discriminant Analysis when the number of examples is too small compared to the number of features.
As per one of the answers in the previous links, I decided to use a pseudo inverse (Moore-Penrose) to work around this problem.

After making that change the classifier could be trained and evaluated. Its accuracy was **81%**: lower than the Naive Bayes classifier. And its confusion matrix is shown below.

		Predicted class	
		yes	no
Class	yes	82	18
	no	20	80

Table 10: Confusion matrix for Hebrew yes/no data using Fisher's LDA classifier.

Column average features

The third bullet in the write-up asks us extract features from the data by taking the average of each of the columns and have a vector formed by them as our feature vector.

To do this I just created a new feature extractor class which I called **AverageVectorFeatureExtractor**. It is rather simple, it iterates over the matrix representation of the example and keeps a count of high energy values on each column. Then it divides by the column size (25).

The rest of the code remained the same except for my choice of smoothing constant. I tried multiple values ranging from 0.1 to 10 and the best accuracy was achieved when using **0.25**.

Overall accuracy: **88%** (8% lower than my implementation for part 2.1)

Now, as hinted in the assignment write-up, the computational complexity was definitely improved.
For Part 2.1 the average training time was **129.26ms** while the new feature extractor caused the training time to drop to **24.8ms**. That is a **500%** improvement!

Finally, this is the confusion matrix:

		Predicted	
		yes	no
Expected	yes	88	12
	no	12	88

Table 11: Confusion matrix for Hebrew yes/no data using column average features.