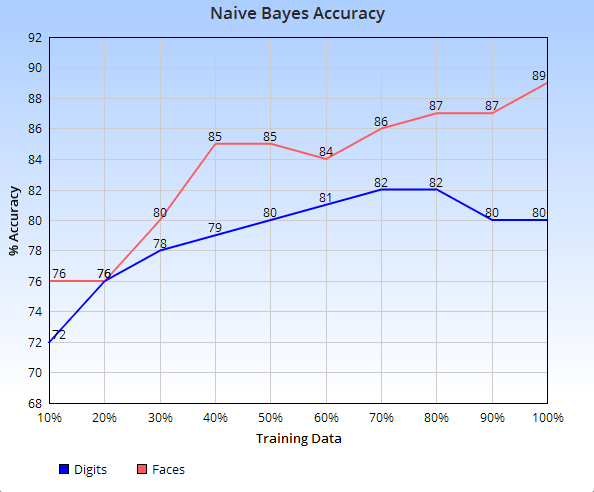
**Project: Face and Digit Classification**

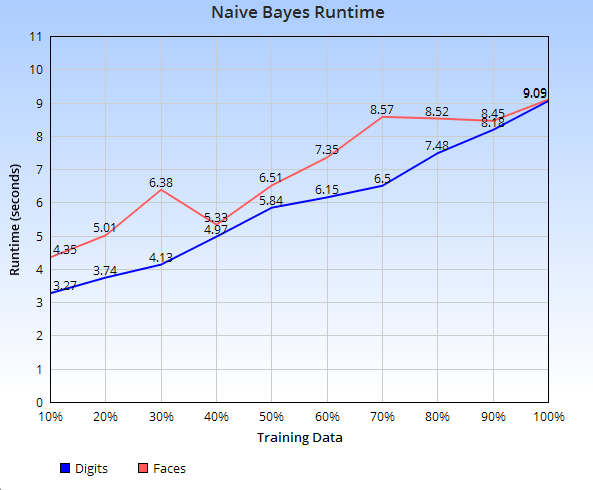
Kevin Nehrbauer RUID: 169005809

Christopher Woolley RUID: 170007112

Naive Bayes

Using the code skeleton provided by Berkeley, Naïve Bayes is implemented under the trainAndTune and calculateLogJointProbabilities methods. In trainAndTune, our code creates three counters to calculate the prior probability, conditional probability, and counts (which is the total data set used to calculate conditional probability). Using a for loop through the training labels gets us the prior probability, and a nested for loop going through each feature contained in training data, adding one to counts([feature, label]) each time, and adding one to conditional probability([feature, label]) if the label is equal to 1 for that particular feature until everything in the dataset has been iterated through. Once the prior is normalized (number of training instances with label y over the number of training instances) it then uses another nested for loop that goes through each value [(feature, label)] in counts and conditional probability, adds 1 to each conditional value and 2 to each counts value (since the only possible feature values can be 0 or 1. To get the final conditional probability values, a for loop sets each conditional probability[i] is set to (conditionalProb[i] / counts[i]). Once this is done, the Berkeley code calls calculateLogJointProbabilitites, which takes for each item i the log(prior probability) + the sum of the logs of each feature’s conditional probability. Overall, Naive Bayes resulted in somewhat accurate results using little of the training data as shown in the graphs below. As the amount of training data increases, the accuracy for both digit and face classifying increases, displaying a learning curve. From our results, Naive Bayes produced more accurate results when executed on Faces. For Digits, we found that in our case, using 70% or 80% training data resulted in 2% more accurate results than using 100%.





Perceptron

Hi chris

MIRA

We implemented MIRA following the code skeleton provided by Berkeley. Similar to Perceptron, MIRA keeps a weight vector for each label y. Then each label from trainingLabels is compared to their predictedLabel. If they are equal, then the instance is correct and nothing is done, else, we need to update the weight vectors. To update them, we calculate tau by taking the min of {C (where c is 50), ((self.weights[predictedLabel] - self.weights[label]) \* datum + 1) / (2 \* (datum \* datum))])} where datum is all the features of y and y\*. The variable diff (differences) is used to update the weight vectors which is calculated by taking all the features and multiplying their values by tau. Finally, the weights are updated using +diff and -diff. These weights are then used by the Berkeley code to classify the data.

As shown in the graphs below, MIRA produces accurate results even at 10% training data. Overall, both digits and faces produced similar learning curves. The large difference between faces and digits is the runtime. On 100% training data MIRA takes 132 seconds to run on digits, whereas on faces, it takes 22 seconds. Since there are 5000 items in digits vs 451 in faces, there are just more items to run through.

