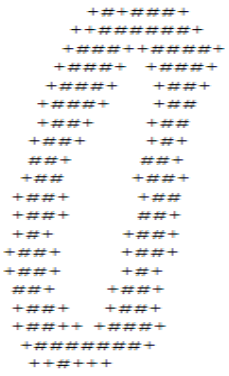
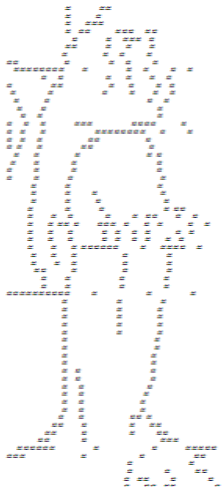
Jared Wyce

AI Final Project Report: Digit and Face Classification

The Problem

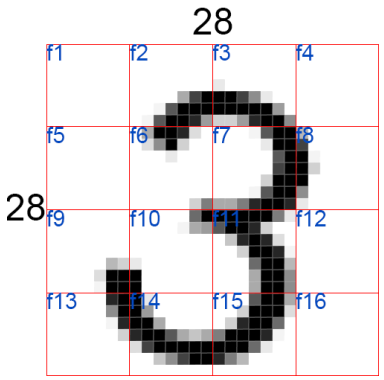
Given a massive amount of sample data, we would like to classify similar data which isn't 100% like anything seen before. Here we consider classifying digits of various handwriting and detecting whether an image is a face or not. Example sample points are shown below.





Features

To train both the digit and the face classification problems, it is very convenient to create a vector made up of various features from the training data. A natural choice of features for these problems would simply be to divide the image into a grid of a certain fineness. Each position in the grid f_i corresponds to a boolean feature which is either 1 if that part of the grid is marked or 0 otherwise. For example if X is the image to the right, then the feature vector f(X) = <0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0>. Then we can assign vector f the label 3 and use it as a training point for the various classification algorithms.

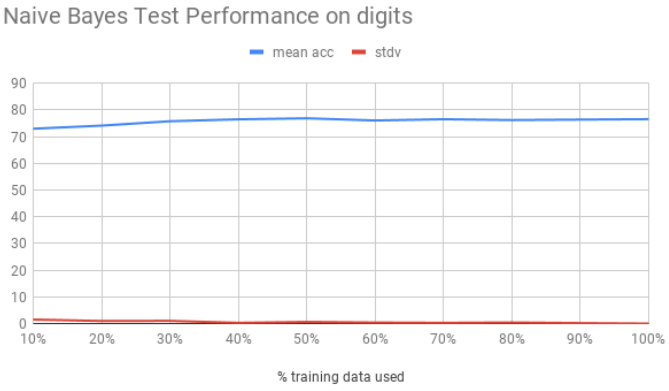


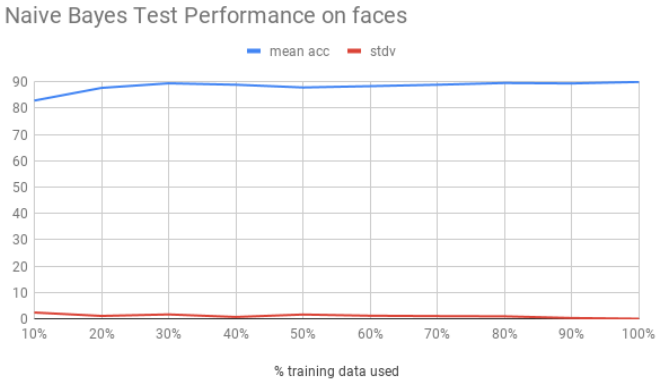
Jared Wyce

Algorithms

Naive Bayes: We implement Naive Bayes by selecting a random portion of the training data to be mapped to a feature vector paired with its label. In this implementation the best features are 2x2 (note: lower resolution) for both digits and faces and they are stored in an array of hashmaps where the index in the array is in the range of possible answers for the particular problem. For example, at the 4th index, all training points with the label 4 are stored there. This is simply to speed up classifying. Time to train here is linear in the number of training points.

To classify, we create a vector of probabilities also indexed by the range of answers for the problem. So for digits, we have the probabilities for the given data point of being any of the possible digits and we classify it according to its highest probability. The probability that data point X is a certain value j is (assuming independence i.e. naive bayes assumption) given by $P(X=j) = \prod_{i}^{l} P(f_i \mid y=j)$ for y ∈ Y where Y = training set and f_i is the i^{th} factor in f(X) and l is the length (dimension) of f(X). Also $P(f_i \mid y=j) = \frac{\#\ of\ f_i=f(y)_i}{\#\ of\ y=j}$. In this implementation, since multiplying probabilities results in underflow we take log(probabilities) $P(X=j) = \sum_{i}^{l} log\,(P(f_i \mid y=j))$. Formally to classify we take the argmax(P(X = j)) for j ∈ Range(answers to the specific problem).





An interesting note here is that the learning curves aren't really much curves as they did pretty much the same on all stages of training. Also, in some cases more training data led to "unlearning". Over all the algorithms this one did the worst, especially with digits.

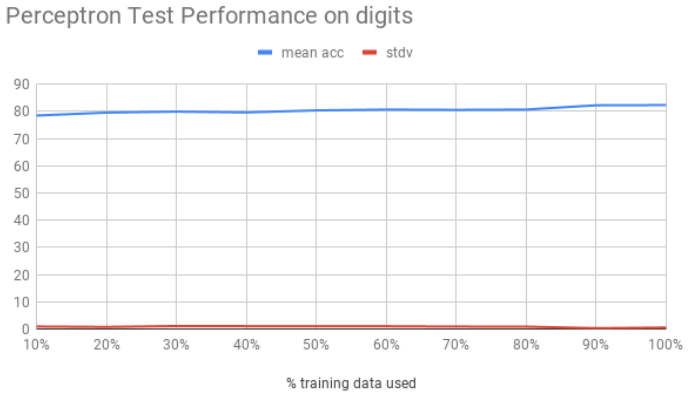
%digit training data used	mean acc	stdv
10%	73.02	1.507846146
20%	74.18	0.9641576635
30%	75.82	1.014692072
40%	76.54	0.2727636339
50%	76.92	0.6241794614
60%	76.1	0.3847076812
70%	76.58	0.2712931993
80%	76.26	0.4127953488
90%	76.44	0.1854723699
100%	76.6	0

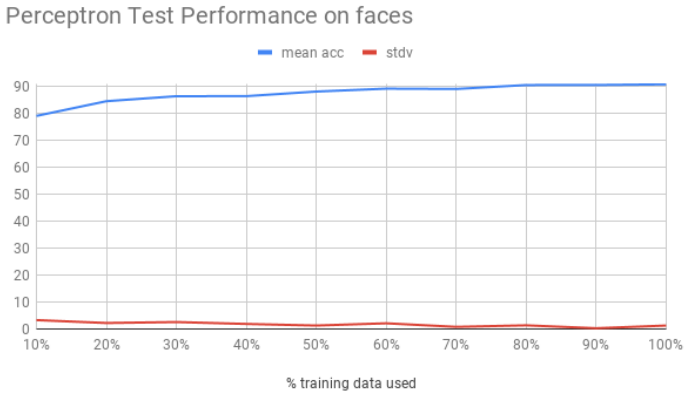
%face training data used	mean acc	stdv
10%	82.92	2.354909765
20%	87.74	1.068831137
30%	89.48	1.637559159
40%	88.94	0.6711184694
50%	87.9	1.6
60%	88.4	1.147170432
70%	88.94	1.00518655
80%	89.6	0.9143303561
90%	89.44	0.28
100%	90	0

Jared Wyce

Perceptron: We implement Perceptron by selecting a random portion of the training data to be mapped to a feature vector paired with its label. In this implementation the best features are 1x1 (i.e. every "pixel" or character) for both digits and faces and they are stored in a hashmap. The crux of this algorithm is in the training phase. We want to construct a linear function approximator so that we can make a decision boundary to split the training data into 2 categories. For digits we have to train 10 perceptrons, one for each digit. To train a perceptron, we store the weights for the linear function and for each training data point check to see if it is on the correct side of the line and if it isn't we update the weights and effective nudge the line in a direction that separates the training data better. Since we implemented a linear approximator, there could be cases where there is no line that exactly separates the training data into 2 perfect categories, in this case we set a "time limit" of maximum 50 iterations before we say our decision boundary is good enough. The training time here is O(range * # iterations * # training points * 2l) where l is the length (dimension) of the feature vectors. This is still linear in the number of training points but has a huge coefficient which is $O(10*50*2*28*28) = O(10^6)$ in the case of digits.

After the perceptron(s) are trained, classification is easy. We take the given data point vector and calculate its value at each function and take the function that gave the max and return whatever that function was measuring.





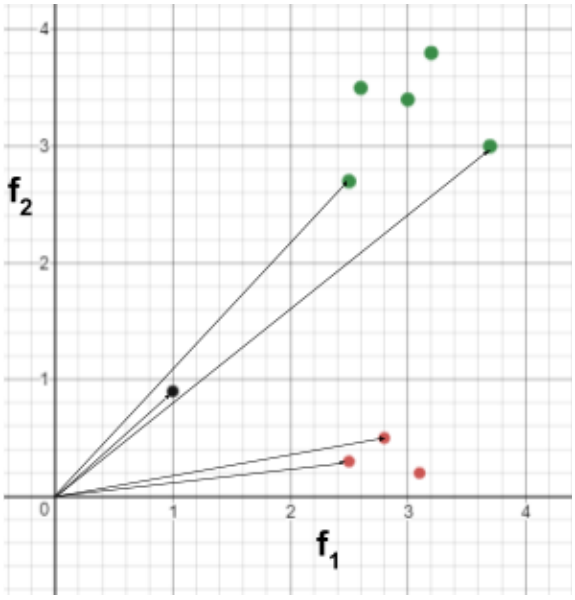
Perceptron did fairly well on both digits and faces. After trying multiple values for the time limit to find a good decision boundary, I found that smaller values were actually better, which went against my initial intuition.

%digit training data used	mean acc	stdv
10%	78.52	0.9108238029
20%	79.6	0.7615773106
30%	79.96	1.098362417
40%	79.66	1.055651458
50%	80.42	1.024499878
60%	80.72	1.010742301
70%	80.62	0.9019977827
80%	80.72	0.8634813258
90%	82.3	0.3033150178
100%	82.38	0.5418486874

%face training data used	mean acc	stdv
10%	79.08	3.30236279
20%	84.54	2.227644496
30%	86.4	2.583795658
40%	86.42	1.874459922
50%	88.12	1.287478155
60%	89.2	2.114710382
70%	89.08	0.793473377
80%	90.54	1.35735036
90%	90.56	0.28
100%	90.8	1.279062156

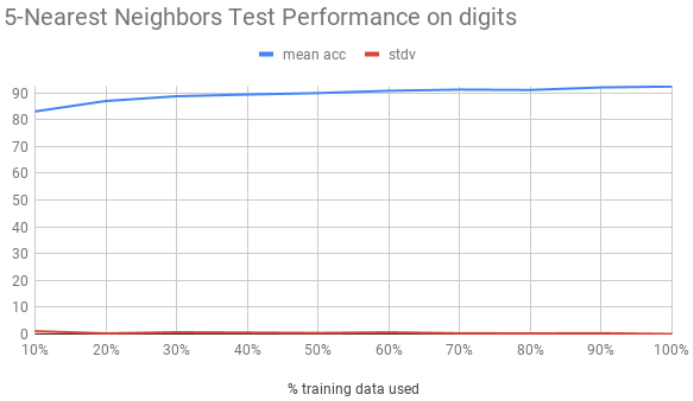
Jared Wyce

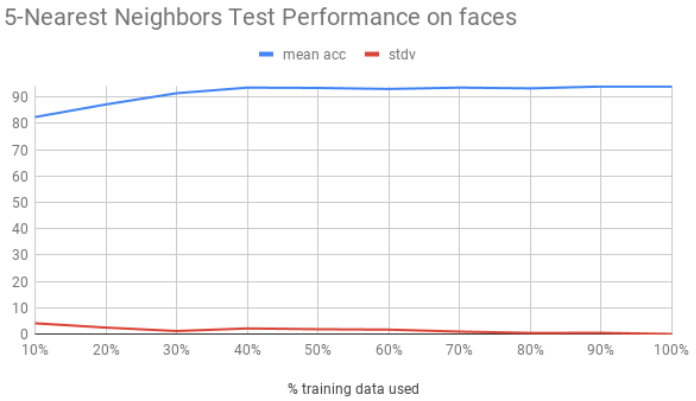
K-Nearest Neighbors: We implement K-Nearest Neighbors by, first, selecting a random portion of the training data and mapping each particular data point to a feature vector matched with a label. In this implementation the best features are 1x1 (i.e. every "pixel" or character) for both digits and faces and they are stored in a hashmap. In this training phase, the time it takes to train is linear in the number of training data used.



To classify new data, we map it to a feature vector and measure its distance from previously encountered points from the training phase. Then we take the most frequent neighbor of the k nearest neighbors. The concrete measure of distance between feature vectors u and v we use is the angle between them given by $\theta = arccos\left(\frac{\sum_i u_i \cdot v_i}{|u||v|}\right)$.

In this implementation, we find the most frequent of the 5 nearest neighbors.





Using 100% of the training data, nearest neighbors was able to classify 1000 digits never seen before with 92.4% accuracy and 150 faces with 94% accuracy which was the best overall.

%digit training data used	mean acc	stdv
10%	83.1	1.060188662
20%	87	0.228035085
30%	88.78	0.6734983296
40%	89.42	0.5418486874
50%	89.96	0.3929376541
60%	90.82	0.6337191807
70%	91.26	0.28
80%	91.12	0.2315167381
90%	92.08	0.2925747768
100%	92.4	0

%face training data used	mean acc	stdv
10%	82.4	4.078234912
20%	87.2	2.447856205
30%	91.46	1.132431013
40%	93.6	2.125088233
50%	93.48	1.82252572
60%	93.08	1.670209568
70%	93.6	0.9143303561
80%	93.32	0.4118252056
90%	94	0.4427188724
100%	94	0