# Project : Face and Digit Classification

Fatima AlSaadeh Ashley Dunn

May 21, 2020

## Description

In this project, we designed three classifiers: a naive Bayes classifier, a perceptron classifier and a multilayer perceptron classifier. These three classifiers were tested on two image data sets: a set of scanned handwritten digit images and a set of face images in which edges have already been detected. The project code followed the instructions from the programming assignments of Berkeley's CS188 course. [1]

## Data Processing, Splitting and Commands

After reading and preparing the data for processing and classifying, we have training and testing datasets and labels, the faces data contains 451 training data points and 150 test data points. The digits data contains 5000 training and 1000 test data points.

The training data points were picked randomly in groups of 10%, 20%, 30%,...100% to train, tune and compare the accuracy of the classifiers for each percentage used of it.
The project provides the option of choosing the dataset to be used - faces or digits, the size of the training and testing dataset, and to autotune the data or not. The following commands are samples to what we used to test the project and get the results :

(run the perceptron classifier with 5000 training images and 1000 test images on digits data)

```
python dataClassifier.py -c perceptron -t 5000 -s 1000 -d digits
```

(run the mlp classifier with default settings)

```
python dataClassifier.py -c mlp
```

(run the naiveBayes classifier on faces data with autotune)

```
python dataClassifier.py -c naiveBayes -t 300  -s 100 -d faces -a
```

## Perceptron

The perceptron classifier uses a binary classifier to determine if the features of the input match the characteristics of the class. There is a binary classifier for each of the classes (one class per digit, one class for face). The image will be classified in the class that produces that maximum score when the feature vector is multiplied by that classes weight vector.

### Algorithm Pseudocode

**Algorithm 1** Perceptron

Function perceptron(F)
**Input:** F = training dataset features [f1,...,fm], L= training dataset labels
**Output:** List of weight vectors for each class

1. Read the training dataset
2. For each training image with feature vector f, get the score for each class y with:

$$score(f, y) = \sum_i f_i w_i^y$$

3. Compute the max score (y') for f to detmine the closest matching class
4. If $y' \neq y$, adjust the weights accordingly:

$$w^y = w^y + f$$

$$w^{y'} = w^{y'} - f$$

### Results

| | Faces | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 45 | 4.647 | 79.3% | 4.582 |
| 20% | 90 | 5.732 | 80.0% | 4.0 |
| 30% | 135 | 8.185 | 88.0% | 1.0 |
| 40% | 180 | 10.495 | 87.3% | 1.414 |
| 50% | 225 | 13.018 | 87.3% | 0 |
| 60% | 270 | 16.505 | 87.3% | 0 |
| 70% | 315 | 17.110 | 87.3% | 0 |
| 80% | 360 | 19.655 | 87.3% | 0 |
| 90% | 405 | 22.176 | 87.3% | 0 |
| 100% | 451 | 24.783 | 87.3% | 0 |

| | Digits | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 500 | 14.055 | 80.1% | 37.467 |
| 20% | 1000 | 35.018 | 78.8% | 14.197 |
| 30% | 1500 | 54.898 | 79.7% | 18.276 |
| 40% | 2000 | 60.905 | 78.8% | 13.457 |
| 50% | 2500 | 50.160 | 80.6% | 5.657 |
| 60% | 3000 | 86.550 | 82.8% | 12.247 |
| 70% | 3500 | 62.675 | 81.3% | 6.325 |
| 80% | 4000 | 75.353 | 82.1% | 8.307 |
| 90% | 4500 | 90.971 | 80.7% | 5.385 |
| 100% | 5000 | 94.643 | 81.5% | 4.583 |

# Naive Bayes Classifier

This classification method is built based on Bayes theorem where we assume that the features are independent. We are using the log joint probability to avoid probability values approaching to zero when multiplying many probabilities together.

$$P\left(y|\, f1, ..., fm\right) = \frac{P\left(\, f1, ..., fm|\, y\right) P\left(y\right)}{P\left(f1, ..., fm\right)}$$

$$= \underset{y}{\text{argmax}}(\log P\left(y\right) + \sum_{i=1}^{m} \log\left(\, fi|\, y\right) P\left(y\right))$$

## Algorithm Pseudocode

**Algorithm 2** Naive Bayes Classifier

Function naiveBayesClassifier(F)
**Input:** F = training dataset features [f1,...,fm], L= training dataset labels
**Output:** $P\,(\,f1,...,fm|\,y)$

1. Read the training dataset
2. Count the occurrence for each label in the training labels L
3. Normalize prior probability p(y)

$$p(y) = numberoflabel_1/totallabels$$

4. Count Black and White Features in F Labels : number of times a feature is black or white pixel in all images

$$count(black\_feature\_label)$$

$$count(white\_feature\_label)$$

$$count(total\_features\_labels)$$

5. Smooth the features counts by adding k value
6. Calculate the conditional probability

$$P\,(\,f1,...,fm|\,y)$$

After this for the test data we calculate the posterior by selecting the maximum value out of calculated log joint probability the ($\log P\,(y) + \sum_{i=1}^{m} \log\,(\,fi|\,y)\,P\,(y)$) followed by the last step of finding the correctness percentage of our overall classification by comparing the calculated guessed label with the original test labels.

## Autotune Smoothing

For the smoothing step we allowed the option to autotune by selecting the -a option where the classification will run over different k values:

$$kgrid = [0.001, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 20, 50]$$

for each percentage of the training data it will run over all the k values, find the best correctness in the validation data and use this k for training the data that will be used to classifay the test data.

### Results

The best K for faces and digits data in most of the iterations was 0.001, the data points in each iteration below were picked randomly, run for 5 iterations per percentage and the recorded values for the training time and accuracy are the average of the total values in these 5 iteration, the results may vary each time we run the classifier.

| | Faces | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 45 | 0.797 | 68.7% | 8.774 |
| 20% | 90 | 1.397 | 80.7% | 14.387 |
| 30% | 135 | 2.096 | 81.3% | 13.341 |
| 40% | 180 | 2.791 | 87.3% | 2.449 |
| 50% | 225 | 3.305 | 87.3% | 1.414 |
| 60% | 270 | 5.522 | 86.7% | 1.0 |
| 70% | 315 | 5.606 | 88.7% | 3.0 |
| 80% | 360 | 5.461 | 87.3% | 2.236 |
| 90% | 405 | 6.091 | 88.0% | 1.414 |
| 100% | 451 | 6.756 | 89.3% | 0.0 |

| | Digits | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 500 | 0.809 | 72.6% | 18.138 |
| 20% | 1000 | 1.686 | 75.0% | 6.324 |
| 30% | 1500 | 2.374 | 75.6% | 13.928 |
| 40% | 2000 | 3.038 | 76.8% | 7.416 |
| 50% | 2500 | 3.867 | 76.2% | 3.0 |
| 60% | 3000 | 4.646 | 76.5% | 7.141 |
| 70% | 3500 | 5.403 | 76.6% | 4.0 |
| 80% | 4000 | 6.253 | 76.6% | 3.872 |
| 90% | 4500 | 6.975 | 76.7% | 2.449 |
| 100% | 5000 | 7.630 | 76.9% | 0 |

## Neural Network

This neural network implementation is a multilayer perceptron, which has one hidden layer of 150 perceptrons. The feature vector from the input layer gets input to each of the hidden perceptrons. The output of the hidden perceptrons then becomes the input for the output layer, which will determine the class of the image. Backward propagation is used to adjust the weights at each perceptron based on the error.

## Algorithm Pseudocode

**Algorithm 3** Multilayer Perceptron

Function mlp(F)
**Input:** F = training dataset features [f1,...,fm], L= training dataset labels
**Output:** List of weight vectors for each class
1. Read the training dataset
2. Initialize the weights for the input layer and hidden layer as vectors of dimensions m by 150 and 150 by number of classes for vectors w0 and w1, respecitively. Set the learning rate = 1
3. Training
**for** each training image **do**
  **Forward propagation:**
    Take the dot product of the input vector and w0, and apply the sigmoid function
    Take the dot product of the previous result and w1, and apply the sigmoid function
  **Backward Propagation:**
    Compute the overall error, then the error caused by the output layer weights and error caused by the hidden layer weights
    Update the weights by taking the dot product of the weight vectors and their corresponding error vectors
**end for**

## Results

| | Faces | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 45 | 4.646 | 61.3% | 7.746 |
| 20% | 90 | 10.064 | 72.0% | 3.871 |
| 30% | 135 | 14.701 | 72.0% | 3.162 |
| 40% | 180 | 24.416 | 78.0% | 3.162 |
| 50% | 225 | 18.473 | 82.0% | 3.00 |
| 60% | 270 | 30.939 | 85.3% | 2.449 |
| 70% | 315 | 22.008 | 87.3% | 4.000 |
| 80% | 360 | 30.972 | 87.3% | 3.317 |
| 90% | 405 | 35.530 | 90.7% | 5.099 |
| 100% | 451 | 33.871 | 88.0% | 2.449 |

| | Digits | | | |
|---|---|---|---|---|
| percentage | training data size | training time(s) | accuracy(%) | standard deviation |
| 10% | 500 | 1.712 | 83.0% | 16.971 |
| 20% | 1000 | 3.475 | 85.1% | 17.205 |
| 30% | 1500 | 5.755 | 85.9% | 8.000 |
| 40% | 2000 | 7.632 | 87.9% | 7.681 |
| 50% | 2500 | 9.917 | 88.9% | 5.385 |
| 60% | 3000 | 12.681 | 89.9% | 9.220 |
| 70% | 3500 | 13.690 | 87.9% | 4.796 |
| 80% | 4000 | 15.635 | 89.9% | 13.490 |
| 90% | 4500 | 17.548 | 89.9% | 14.107 |
| 100% | 5000 | 18.476 | 88.9% | 5.196 |

# References

[1] D. Klein and J. DeNero. Project 5: Classification.