Hunter Betz (hhb8)

Perceptron & Naive Bayes commands:

How to use:

Navigate to the directory the files are located and run the following command:
python dataClassifier.py -c classifier -d dataset -t #### -s ####

-c *classifier* refers to the type of algorithm to test on. In this case it is either naivebayes or the perceptron algorithm
-d *dataset* refers to the data set to run your classification algorithm against. The default is digits, but you can specify faces
-t #### is the number data points to train the algorithm on from the training set. The default is 100. #### refers to the number of data points which need to be specified (i.e. -t 1000 uses 1000 training data points)
-s #### is the number of data point to test the algorithm on from the training. The default is 100

Example:

```
ImageClassifier$ python dataClassifier.py -c perceptron -t 500 -s 2000
```

This trains the perceptron on 500 digit data points before testing it on 2000 digit data points in the data set

```
ImageClassifier$ python dataClassifier.py -c perceptron -d faces -t 450 -s 100
```

This trains the perceptron on the entire faces data set before testing it on 100 of the faces data points in the test data set.

Perceptron Algorithm:

You can add an extra command, -i ####, to specify the number of iterations, formally known as epochs, that the perceptron iterates over. The default is 3.

Example:

```
ImageClassifier$ python dataClassifier.py -c perceptron -t 5000 -s 500 -i 7
```

In this case, we are training the whole data set and testing over 500 test data points. The number of epochs we have is 7.

Our perceptron keeps a weight vector w_y for each label, y (the digits from 0 to 9) in our training data set. Given a feature vector *f*, the perceptron calculates the label *y* that is most similar to the input vector *f*. For each feature vector *f* we score each label as:

score = $\Sigma f * Wy$

Where Wy == w_y

We choose a our predicted label by choosing the score with the highest value for that data instance.It then compares the calculated label with the actual label. If the labels are equal, then the perceptron classified correctly, otherwise it classified incorrectly and updates its weights accordingly:

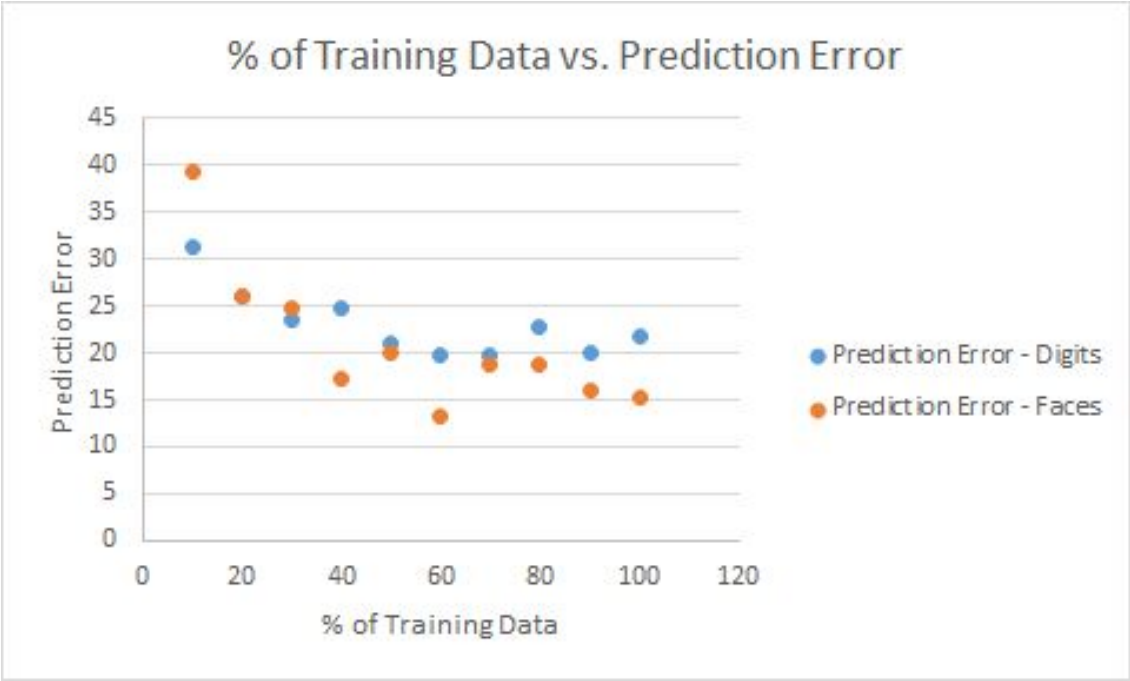
weights [y] = weights[y] + *f*

weight[y_predicted] = weight[y_predicted] - *f*

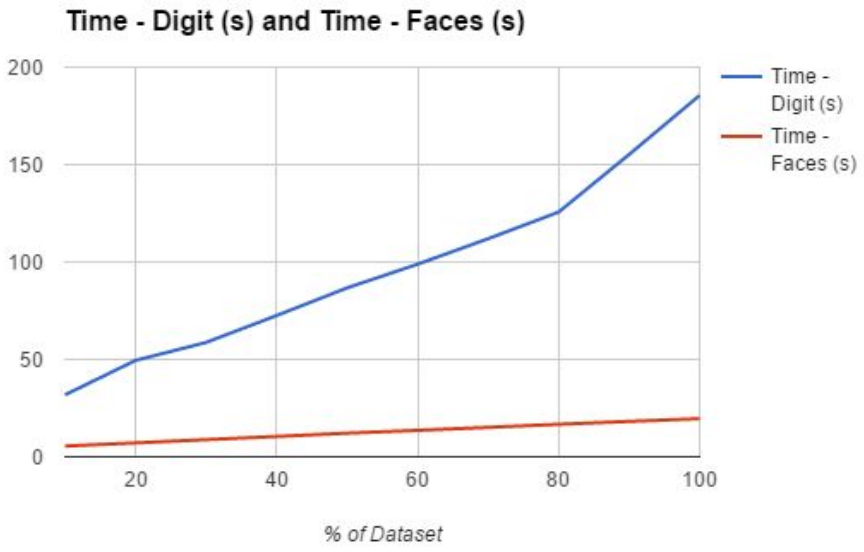
weights [y] should have scored *f* higher
weight[y_predicted] should have scored *f* less

With each epoch, the perceptron passes through the training data and updates the weight vector for each label based on any classification errors it comes across. Ideally the more epochs the more fine-tuned the weights become.

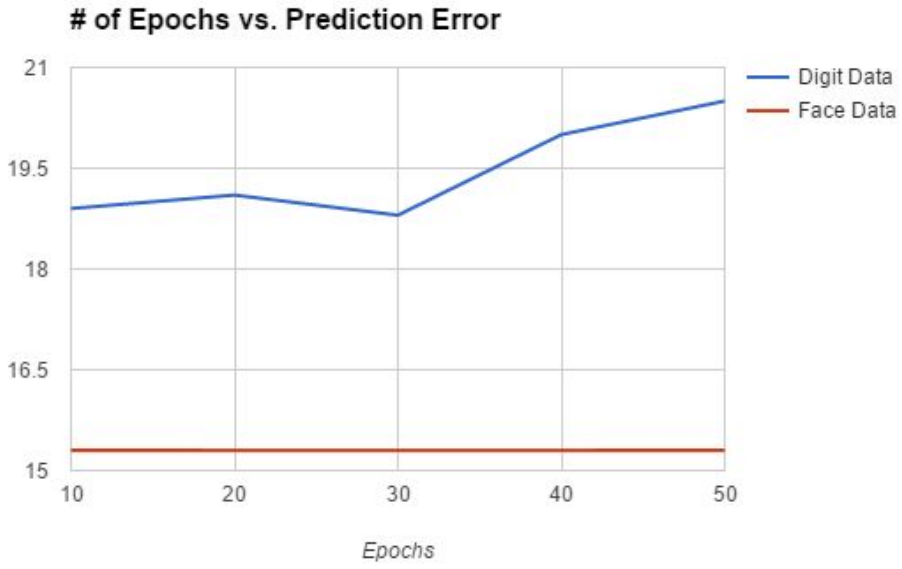
Data:
Using the default epochs of 3, we compared the percentage of the test data that was classified correctly with the percentage of the training data set used. We also compared the time it took with the percentage of the training data set used.



Standard deviation digits: 3.6316
Standard deviation faces: 7.5688



Due to the long periods of time it takes for the perceptron to iterate through the training data set, we only compared the number of epochs vs. prediction error for the entire training data set and incremented epochs by 10



Results:

It's interesting to note the differences in our results between the digit data and the face data. For both faces and digit data, our perceptron algorithm has a sharp decrease in prediction error when we increase the training data set from 10% to 20%. However, after this the gains are practically negligible for the digit data and very small for the faces data.

Both digit and face data appear to have a positive linear relation between % of Training Dataset Used vs. the time it takes for the program to finish executing. This isn't surprising considering we are incrementing the training data set in a positive linear fashion and the faces data set is smaller than the digit data set.
What is the most interesting is the relation between the number of epochs and the prediction error. For the faces data, the number of epochs had no effect on the prediction error - it remained constant. For the digit data, it appears that there is a positive relationship between the number of epochs and the prediction error. This is surprising considering that with each epoch, the weights of our perceptron become more fine tuned to changes. This might suggest that a perceptron could become "too tuned" to the training data where it fails to recognize/classify anything that might even be slightly different than what it was trained on. It's also possible that it reaches a plateau but considering the amount of time it takes for increasing number of epochs, we decided to halt our testing at 50 epochs.

Naive Bayes:

Example:

```
ImageClassifier$ python dataClassifier.py -a -c naiveBayes --autotune -t 500 -s 1000
```

To classify a feature set, we find the most probable label given the feature values of each pixel by using Baye's Theorem:

$$P(y \mid f_1 \dots f_m) = P(f_1 \dots f_m \mid y) * P(y) / P(f_1 \dots f_m)$$
$$= P(y) * \Pi^m_{i=1} P(f_i \mid y) / P(f_1 \dots f_m)$$

But as noted here, multiplying multiple probabilities may result in underflow so we instead compute the log probabilities.

$$P(y \mid f_1 \dots f_m) = \log(P(y)) + \Sigma^m_{i=1} \log(P(f_i \mid y))$$

Prior distribution:

We estimate the prior distribution for each label P(Y) by doing

$$P`(y) = C(y) / n$$

Where C(y) is the number of training instances with label y and n is the size of the training data

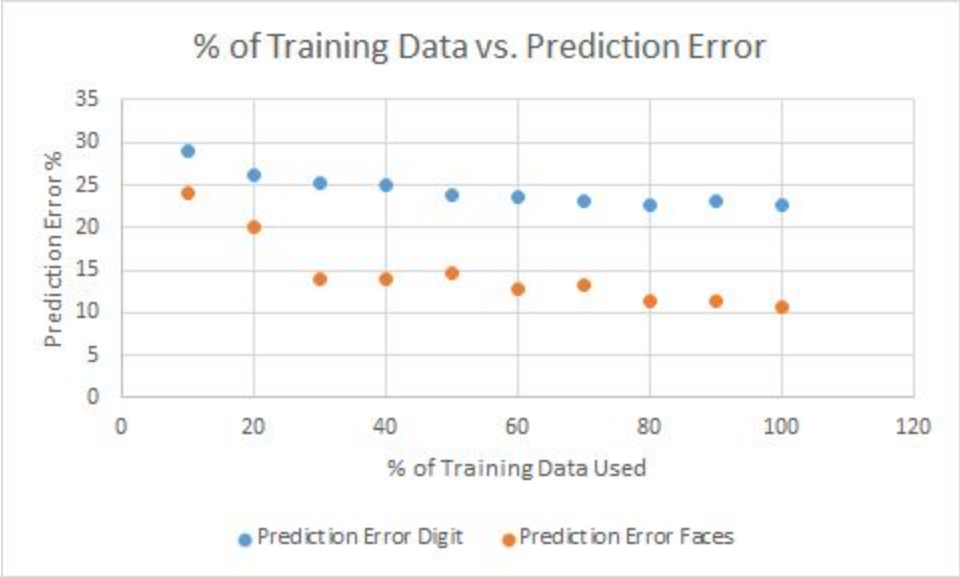
Conditional Probabilities:
We calculate the conditional probabilities of our features given each label y ($P(F_i \mid Y = y)$) for each feature value (0, 1)

$$P`(F_i = f_i \mid Y = y) = C(f_i , y) / [ \Sigma_{f_i \in \{0, 1\}} C(f_i` , y)]$$

$C(f_i , y)$ is the number of times pixel F_i had the value of f_i in the training examples of label y.

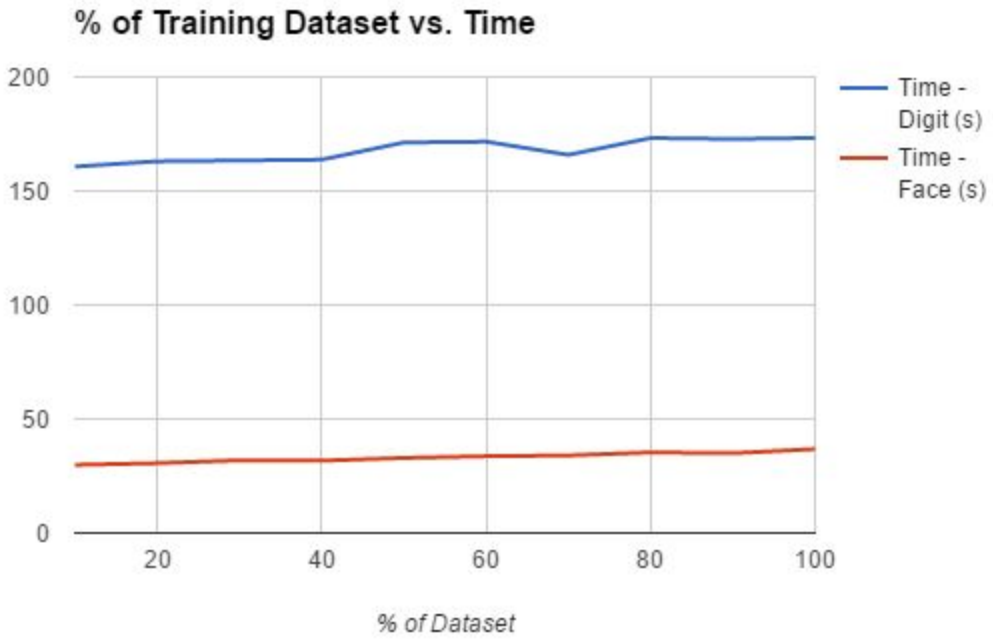
To ensure that we don't have a case where $P(F_i = f_i \mid Y = y) = 0$, we add a smoothing parameter which adds a value to every possible observation value:

$$P(F_i = f_i \mid Y = y) = [C(f_i , y) + k] / [\Sigma_{f_i \in \{0, 1\}} C(f_i` , y) + k]$$



Standard deviation digit data = 2.0274

Standard deviation faces data = 4.2234



Results:

It appears that the Naive Bayes algorithm is far more accurate in identifying faces than it is at classifying digits. Similar to the perceptron algorithm, there is a sudden decrease in prediction error when increasing sample size from 10% to 20% but this decrease levels off from 20% onwards. It is also interesting to note that the execution time for Naive Bayes steadily increases but that sample size has a very minimal effect on time. It is also interesting to note that compared to the perceptron algorithm, Naive Bayes takes longer to execute until the data set reaches a certain size. From that point on, the perceptron algorithm takes longer to execute and this gap will only increase given that execution time has a positive linear growth with respect to data set size.

K-Nearest Neighbors:

K-Nearest Neighbors commands:

How to use:

Navigate to the directory the files are located and run the following command:
python knnRun.py *-f* #### ####

-f is an optional command. It specifies to use the faces data. Omit this command if you wish to test for the digit data
refers to the amount of the training data set you wish to use. A value must be specified.
refers to the k value for the number of nearest neighbors you wish to consider. Again, this value needs to be specified.

Example:

```
ImageClassifier$ python knnRun.py -f 450 2
```

This example will run the entire faces training data set while considering the 2 nearest neighbors.

```
ImageClassifier$ python knnRun.py 750 5
```

This example will use the first 750 data points from the digit data training set while considering the 5 nearest neighbors.

Algorithm:

For our custom classification algorithm we implemented the K-Nearest Neighbors algorithm. To measure similarity between our features we implemented the Euclidean distance metric. The Euclidean distance is the square root of the sum of the squared differences between two numbers; in our case it's the square root of the sum of the squared differences between two features.
Euclidean Distance:

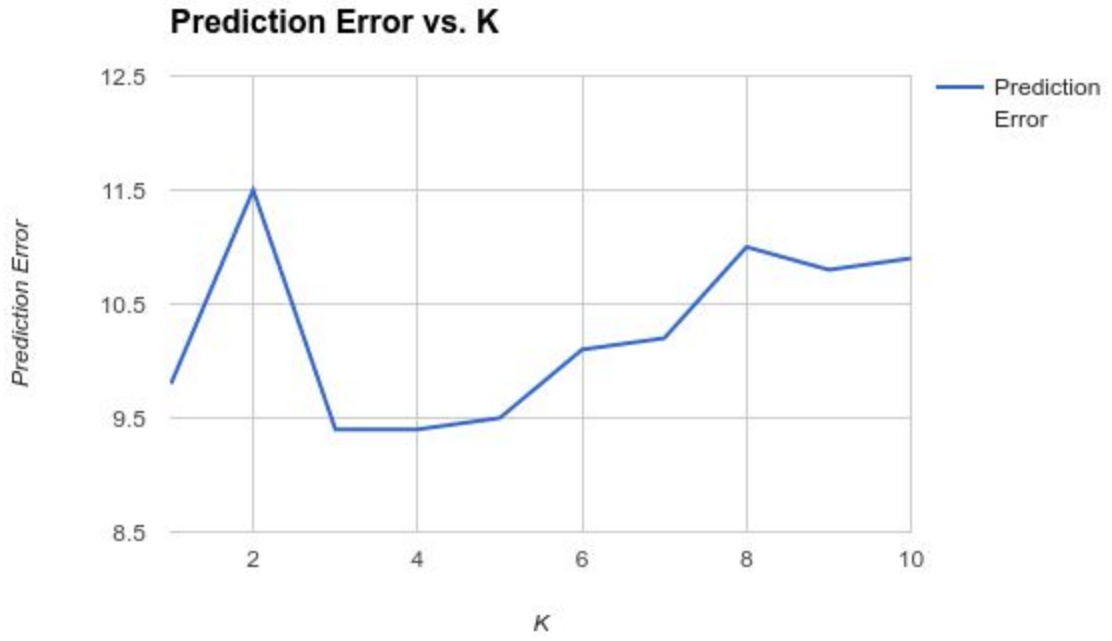
distance = $\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2 + \ldots + (a_n - b_n)^2}$

In our classifier we have a heapq to select the k smallest elements (which is determined by our Euclidean distance function) of our training data. We then create a list of the labels corresponding to the points returned by our heapq. Then, to make our classification complete, we select the maximum of the labels in our list with each label weight is its frequency in the list.

Prediction Error as a function of Training Data:

Standard deviation digit: 5.12796
Standard deviation faces: .843389

Time as a function of Training Data:



Results:

Similar to the other algorithms, the K-Nearest Neighbor algorithm classifies the faces data much faster than it classifies the digit data. However, the algorithm has a very poor success rate in correctly classifying images as faces or not. The time it takes for the algorithm to classify the digit data as a function of percentage of the training data set used, has a positive linear relation. Furthermore, the number of neighbors considered had a negligible effect on time for both faces and digit data. It was interesting to see that as k (the number of nearest neighbors considered) increased, the accuracy of the algorithm was more than 90% for k between 3 and 5 but began to get worse as more neighbors were considered.