Final Project

Face and Digit Classification

Author:
Brian Y. Mahesh A.
brianyi14@gmail.com
ma1700@scarletmail.rutgers.edu

Supervisors:
Abdeslam Boularias
Aravind S.

CS520: Artificial Intelligence
Rutgers University

April 29, 2020

1 Import and Data Processing

For this project, we used NumPy and Pandas modules to import, process, and transform the raw image data into usable information for model training. Below is an example of the the raw digit image data along with the converted image using our convert() method; the raw face image data is similar in construction. We chose to equate '#' and '+' pixels with a value of 1 while any empty pixels were assigned a value of 0. We storeed most of our data in Panda's Series and DataFrame data structures along with Python's native list structure.

```
array([['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                +++++##+    ', '0000000000000000111111110000'],
       ['        +++++######+###+    ', '0000000011111111111111110000'],
       ['       +##########+++++     ', '0000000111111111111111100000'],
       ['        #######+##          ', '0000000011111111110000000000'],
       ['        +++###  ++          ', '0000000011111100110000000000'],
       ['           +#+              ', '0000000000011100000000000000'],
       ['           +#+              ', '0000000000011100000000000000'],
       ['            +#+             ', '0000000000001110000000000000'],
       ['            +##++           ', '0000000000001111100000000000'],
       ['             +###++         ', '0000000000000111111000000000'],
       ['              ++##++        ', '0000000000000011111100000000'],
       ['                +##+        ', '0000000000000000111100000000'],
       ['                 ###+       ', '0000000000000000011110000000'],
       ['              +++###        ', '0000000000000011111100000000'],
       ['           ++######+        ', '0000000000011111111100000000'],
       ['         ++#######+         ', '0000000001111111111000000000'],
       ['       ++#######+           ', '0000000111111111100000000000'],
       ['      +#######+             ', '0000001111111110000000000000'],
       ['    ++#######+              ', '0000111111111100000000000000'],
       ['    +####++                 ', '0000111111100000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000'],
       ['                            ', '0000000000000000000000000000']],
      dtype=object)
```

2 Naive Bayes Model

2.1 Algorithm

For our Naive Bayes model, we predict the classification of an image through the following Naive Bayes assumption: $P(Class|Data) = \frac{P(Data|Class)P(Class)}{P(Data)}$. For each observation, we calculate $P(Class|Data)$ for each possible class, and choose the class with the highest probability as our prediction. Since $P(Data)$ is a constant, we can ignore it and focus on evaluating $P(Data|Class)P(Class)$ when estimating for $P(Class|Data)$. $P(Class)$ is the probability that a certain class appears in a data set and $P(Data|Class)$ is the probability that a certain feature value appears within a certain class. Since we are calculating $P(Data|Class)$ through a discrete methodology, there may be many values for a particular feature that do not appear even once in a class; this causes $P(Data|Class) = 0$ and messes up our estimation of $P(Class|Data)$. As such, we employ a smoothing method over our model parameters such that any $P(Data|Class)$ that equal zero are assigned a very small value of 1×10^{-10}.

2.2 Method

We use four methods to train and test our Naive Bayes model, partition(), feature_ext(), train_nb(), and test_nb(). Partition() serves to divide a single image of 0s and 1s and divides the image into the respective feature count and returns an array of features for that particular image. Feature_ext() takes these arrays of features returned from partition() and aggregates how often each feature value appears for each feature. Feature_ext() returns a dataframe where the columns represent each unique feature and the rows represent the total count of feature values. At that point, it is a simple matter to calculate P(Data | Class) from the dataframe by dividing each feature value aggregate count by the number of observations in that class. After we have P(Class) and P(Data | Class), we simply take the product for each class and find the highest product and return that as our prediction. We use the same training and testing method for both digit and face classification since our method is robust in feature input and adjusts the feature selection algorithm accordingly.

2.3 Digit Classification

We will first look at our model's performance on digit classification for digits of 0 through 9. Our feature selection is obtained by dividing a digit image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition is considered as one feature. Since each digit image is 28 x 28 pixels, we divided the image into 7 x 7 partitions for a total of 16 features. Unfortunately, using 100% of the training set, we got an accuracy of 63.8%. Being a ways off from the 70% cutoff, we decided to narrow the size of each feature to 4 x 4 for a new total of 49 features. By creating smaller partitions, our features become more precise in capturing the nuances that differ between each digit. This turned out to be the better choice and increased our model's overall accuracy from 63.8% to 74.7%. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. As you can see from the graph below, the accuracy of the model steadily improves the greater the percentage of the training set is used. This trend makes sense since the more data the model has on the various ways to draw each digit, the more accurate the model is when predicting a new test digit the model hasn't seen before. With regards to time efficiency, the majority of the time is spent testing the dataset as opposed to training due to method design. Even though the amount of training data varies throughout our testing, the test data remains the same at 1000 observations every time. This is why it takes about 130 seconds on average to fully train and test our model regardless of how much training data is used.

Table 2.1: Accuracy and Time of Naive Bayes Digit Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Time (sec)
10%	58.60	1.77	127.52
20%	65.12	1.85	105.45
30%	69.28	0.99	128.76
40%	71.06	1.08	120.70
50%	72.58	0.98	102.83
60%	73.14	0.63	115.49
70%	73.94	0.69	111.45
80%	73.80	0.43	114.23
90%	74.46	0.47	124.79
100%	74.70	0.00	114.15

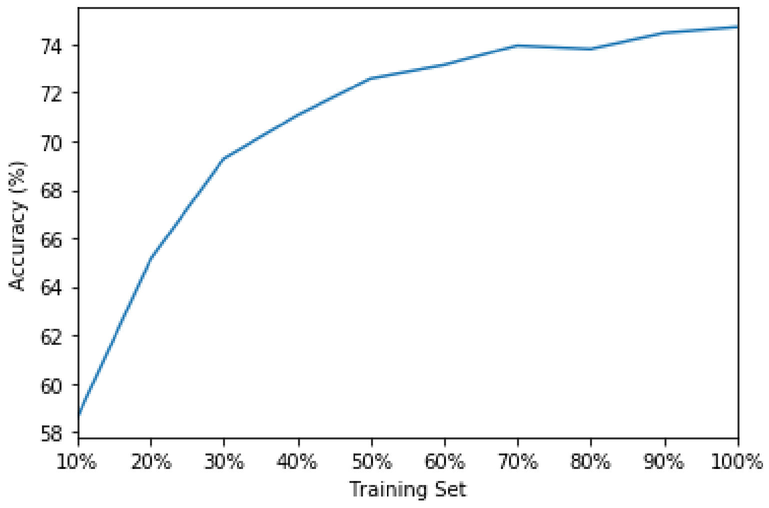

Figure 2.1: Accuracy at Various Training Set Divisions

2.4 Face Classification

For face classification, our goal is to determine whether an image is a face or not. Our feature selection for face classification is similar to that of digit classification in that we divied a face image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition is considered as one feature. Since each face image is 60 x 70 (width x height) pixels, we divided the image into 2 x 2 partitions for a total of 1050 features. Using 100% of the training set, we immediately got an accuracy of 68.7%. For a Naive Bayes model, we felt that this accuracy, being 1% off 70%, was sufficient in predicting our face data. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. Like our Naive Bayes model for digit classification, the accuracy of this model also improves as more of the training set is used to train our model. The time efficiency is constant for the same reasons as explained in digit classification since the same method is used. This time our testing set is much smaller at 150 observations so the average time it takes to fully train and test our model is 27 seconds.

Table 2.2: Accuracy and Time of Naive Bayes Face Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Time (sec)
10%	51.07	0.68	24.91
20%	59.20	1.29	26.03
30%	63.73	1.50	25.56
40%	66.67	0.73	26.00
50%	66.93	1.37	25.90
60%	67.73	2.09	27.35
70%	68.40	0.80	27.51
80%	68.53	0.65	27.39
90%	68.93	0.33	27.84
100%	68.67	0.00	27.73

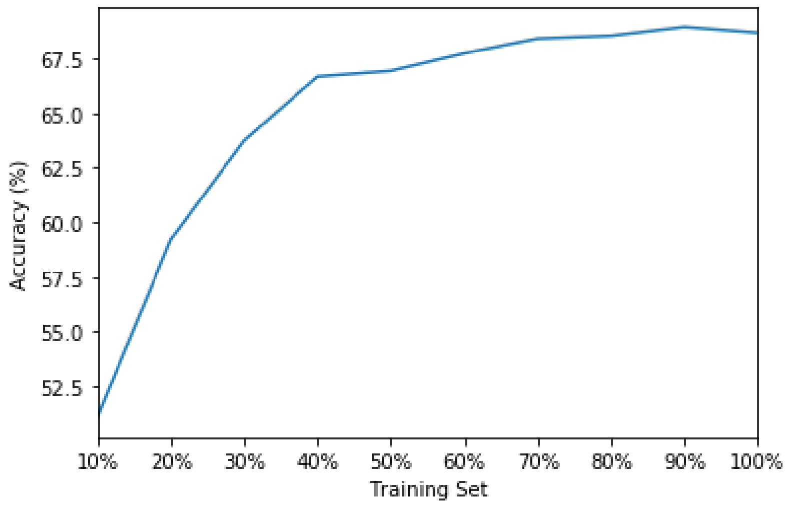

Figure 2.2: Accuracy at Various Training Set Divisions

3 Perceptron Model

Table 3.1: Accuracy and Time of Perceptron Digit Classification

Training	Accuracy (%)	Standard Deviation	Time (sec)
10%	68.28	3.86%	4.67
20%	74.18	4.38%	7.04
30%	75.96	0.79%	9.34
40%	78.16	1.71%	11.65
50%	76.30	3.76%	14.06
60%	77.26	2.67%	16.65
70%	77.04	1.23%	19.00
80%	77.66	2.04%	21.05
90%	79.34	1.73%	24.00
100%	80.40	1.78%	26.38

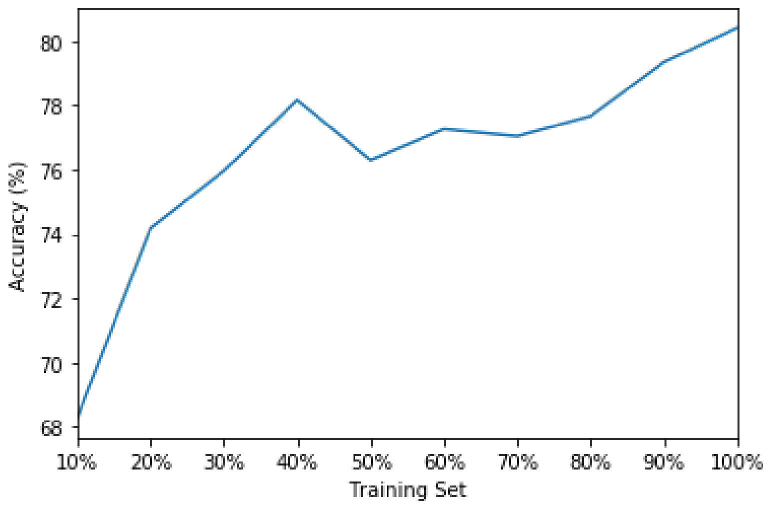

Figure 3.1: Accuracy at Various Training Set Divisions

Table 3.2: Accuracy and Time of Perceptron Face Classification

Training	Accuracy (%)	Standard deviation	Time (sec)
10%	68.40	5.03%	1.16
20%	78.67	3.6%	1.52
30%	81.60	2.0%	1.88
40%	83.33	3.13%	2.16
50%	84.40	3.34%	2.49
60%	84.13	3.99%	2.87
70%	84.93	2.88%	3.20
80%	87.73	1.61%	3.65
90%	85.73	1.72%	3.94
100%	87.20	2.04%	4.31

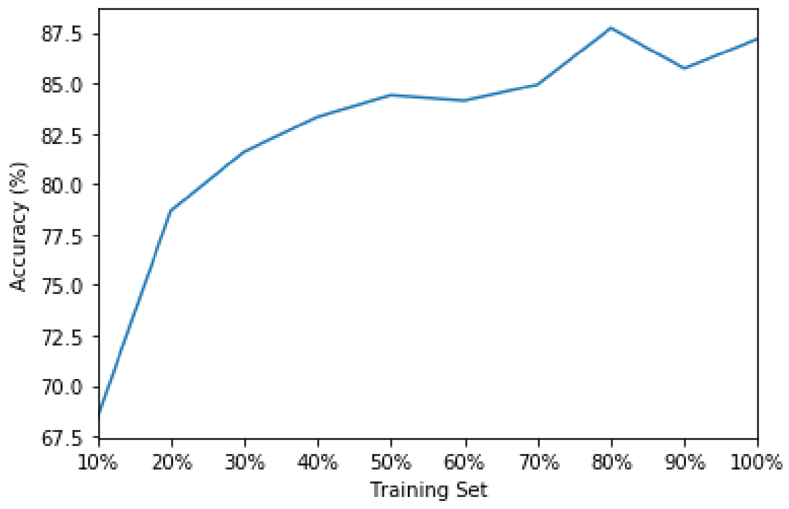

Figure 3.2: Accuracy at Various Training Set Divisions

4 KNN Model

4.1 Algorithm

For our third model, we chose to use the K-Nearest Neighbor (KNN) model to predict the classification of an image. KNN is also a form of supervised learning where the model is trained on a data set with labels. KNN predicts the class of an observation by finding the K nearest neighbors to that observation by a designated "distance" metric. Of these K nearest neighbors, we will select the majority class as our final prediction for that observation. The distance metric for our model will be the sum of the difference in pixels between two images. For example, if we have two different images of the digit 3. After converting both images into 0s and 1s, consider the converted image as a matrix of 0s and 1s. We then subtract one matrix from the other, take the squared value to avoid negative values, and sum up all the remaining 1s as our final distance value. So if any two images are identical, the distance between the two images would be zero. Our K nearest neighbors will be the K images that have the smallest distance with respect to our test image. Usually, an odd number K is selected for a model with an even number of classes. Since both our digit and image data have an even number of classes of 10 and 2 respectively, we chose K=7 as our model parameter.

4.2 Method

The methodology for KNN is much simpler than that of our previous models. Consider one single test observation. If we had to find the K=7 nearest neighbors to this test observation by our distance metric, we need to find the distance between this test image and every image in the training set. Our training/testing procedure is wrapped up in one method that finds the distance between any test image and all the training images and then chooses the majority class of the 7 classes with the smallest distance. Note that since a tie is much more likely to occur in this case as compared to our earlier model metrics, our model handles tie breakers through Python's list's inherent ordering since we order our 7 nearest neighbors and then select the class in the first index. We use a separate training/testing method for digit and face classification although the key difference lies only within feature selection in the form of data transformation.

4.3 Digit Classification

We will first look at our model's performance on digit classification for digits of 0 through 9. As explained earlier, our features for this model is the distance metric that is calculated by comparing two images pixel by pixel. We initially chose k=7 for the number of neighbors when testing our model. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. Unfortunately, the model takes a very long time to train and test since each test observation has to be compared to every single training observation pixel by pixel; thus, we only took an average accuracy over three iterations and used a test set of only 50 observations. Do note that while our training set is randomized each time for every partition, we chose the same 50 observations to test our model each time for more consistency. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. Once again, the accuracy of the model improves as more of the training set is used to train our model with a peak accuracy of 90% when we use the entire training set. Looking at the time efficiency of our model, using only 10% of the model takes X seconds while using the whole training set takes the model a full X hours to train!

Table 4.1: Accuracy and Time of KNN Face Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Time (sec)
10%	79.33	4.11	172.71
20%	87.33	1.89	1017.01
30%	86.00	1.63	1574.47
40%	86.67	1.89	1376.63
50%	88.67	2.49	1050.52
60%	87.33	0.94	1526.17
70%	87.33	2.49	3963.81
80%	87.33	1.89	2196.54
90%	89.33	1.89	1514.02
100%	92.00	0.00	1673.93

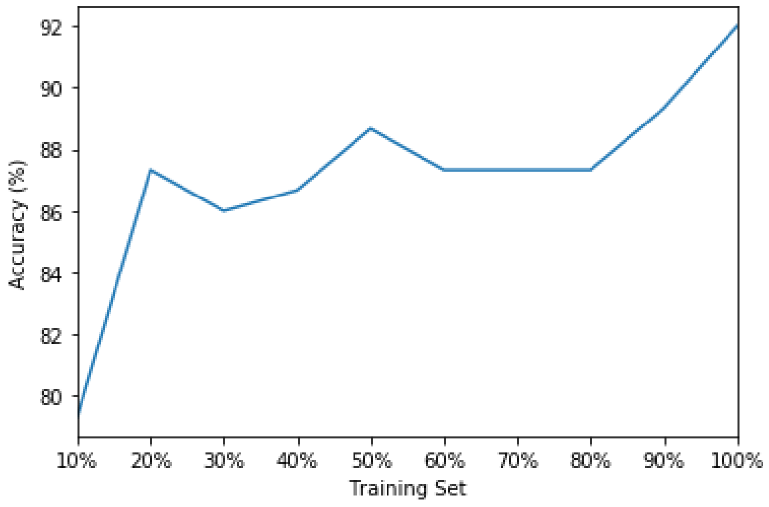

Figure 4.1: Accuracy at Various Training Set Divisions

4.4 Face Classification

As explained earlier, our features for the face classification KNN model is the distance metric that is calculated by comparing two images pixel by pixel. We initially chose k=7 for the number of neighbors for this model due to its efficacy in digit classification. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. We once again only took an average over three iterations due to the nature of KNN training and testing times, although we opted for a larger test set of size 150 since our training set of 451 data points is significantly less than the 5000 used in digit classification. Initially, our model only had an accuracy of 58.67% when trained upon the full training set. After looking through the image data again, we noticed that the deciding features of a face was far more consistent in the eye, nose, and mouth region. We decided to narrow the focus of our model to only the middle 25 x 30 pixel region and retrained our model accordingly. This time, our model's accuracy shot up to 74% from 58.67% while trained on the full training set. The following dataframe shows the accuracy and time it takes to train and test our model for various divisions of our training set. Unlike the Naive Bayes model, this graph shows an unexpected decrease in accuracy between 40 and 50% before improving constantly afterwards. This can be explained by taking a look at the standard deviations of the accuracy measurements between the first 10-30% training set iterations that tells us whether the mean accuracy measurement may have been influenced by outlier accuracy iterations. First, note that the standard deviations during the "accuracy dip" are 5.45% and 4.91% for 40%

and 50% (training set partitions) respectively. Now compare those standard deviations to the standard deviations measured before the accuracy dip, which are 8.82%, 9.02% and 8.24%. This allows us to infer that the accuracy measurements before the accuracy dip may be assumedly higher than it should be and inflated by an outlier measurement during one of the iterations. The standard deviation starts to decrease around 70%, which makes sense that there is less variation as more of the training set is sampled, leading to a steady increase in accuracy from there onwards. Overall, the standard deviations range from 5% to a shocking 10% when using 60% or less of the training set, which can be explained by the fact that our method only ran three iterations per training set partition due to runtime restrictions. Looking at the time efficiency of our model, the average runtime per iteration is 10 minutes for a total of 5 hours over thirty iterations.

Table 4.2: Accuracy and Time of KNN Face Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Time (sec)
10%	56.89	8.82	161.53
20%	63.11	9.02	293.37
30%	66.22	8.24	411.93
40%	64.89	5.45	631.28
50%	63.11	4.91	713.95
60%	64.44	6.89	857.38
70%	67.56	3.00	1012.93
80%	69.78	2.06	902.04
90%	72.00	1.96	434.50
100%	74.00	0.00	483.65

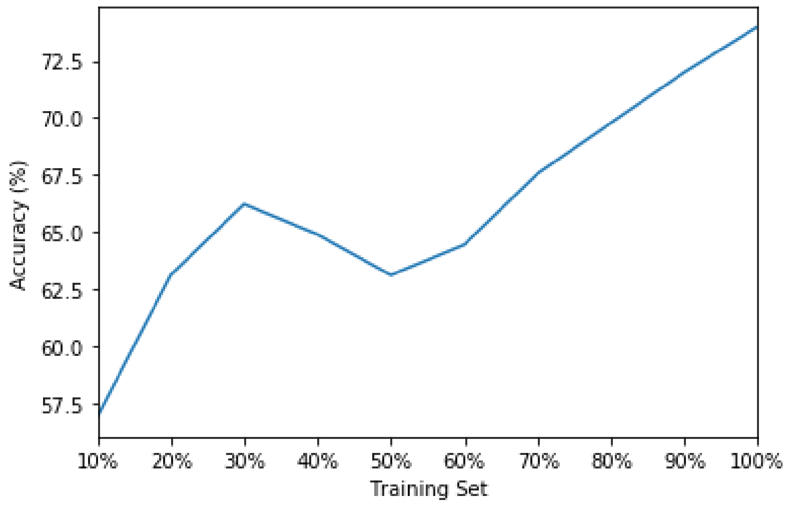

Figure 4.2: Accuracy at Various Training Set Divisions

5 Old Methods