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

May 2, 2020

1 Import and Data Processing

For this project, we use NumPy and Pandas modules to import, process, and transform the raw image data into usable information for model training. The figue below is an example of the raw digit data (left) along with the converted image (right) created through our convert() method; we use the same convert() method to translate the face data as well. We choose to equate '#' and '+' pixels with a value of 1 and empty pixels with a value of 0. We store most of our data in Pandas Series and DataFrame 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'],
       ['            ++#####+        ', '0000000000001111111100000000'],
       ['          ++######+         ', '0000000000111111111000000000'],
       ['        ++######+           ', '0000000011111111100000000000'],
       ['       +######+             ', '0000000111111110000000000000'],
       ['    ++######+               ', '0000111111111000000000000000'],
       ['    +####++                 ', '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 test observation, we calculate $P(Class|Data)$ for each possible class and choose the class with the highest probability as our prediction. When estimating for $P(Class|Data)$, we can focus on evaluating $P(Data|Class)P(Class)$ and ignore $P(Data)$ because $P(Data)$ remains the same for a given test observation. $P(Class)$ 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 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 $P(Class|Data)$. To counteract this, we employ a smoothing method over our model parameters such that any $P(Data|Class)$ values 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(). First consider the converted digit images that consist of 0s and 1s. Partition() divides this single image of 0s and 1s into its features and returns these features as an array. 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 unique features and the rows represent the total counts for feature values. From here on, 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 calculating $P(Class)$ and $P(Data|Class)$ for all unique classes, we find the class with the highest $P(Class)P(Data|Class)$ value 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.

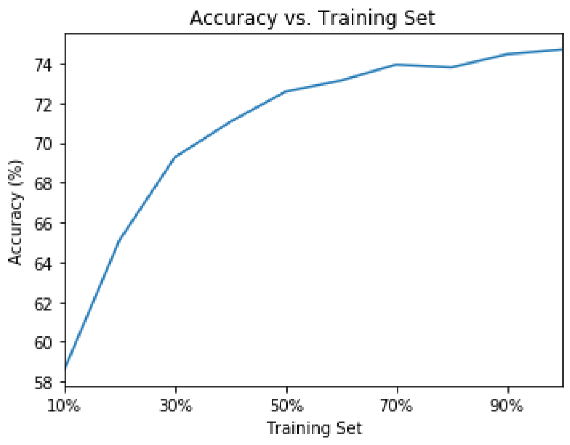
2.3 Digit Classification

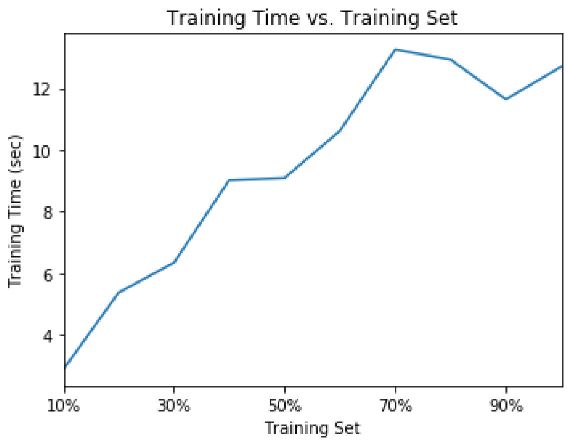
We first looked at our model's performance on digit classification for digits 0 through 9. Our feature selection is obtained by dividing a digit image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition is considered as one feature. Since each digit image is 28 x 28 pixels, we initially divided digits into 7 x 7 partitions for a total of 16 features per digit. After we trained this model 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 a 4 x 4 partition 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% when trained upon the entire training set.

Table 2.1 shows the accuracy and time it took to train and test our model using various divisions of our training set over 5 iterations. The accuracy of our model steadily improved when more of the training set was used; when our model had more data on various ways to draw each digit, the model became more accurate when predicting any new test observations. Note that as more training data was used, the standard deviation of accuracy measurements decreased over time as well. This decreased variation in accuracy was the result of using larger training sets that minimized the chance of any one iteration containing biased data that skewed the mean accuracy. Overall, the small standard deviations in accuracy over five testing iterations per training set told us that our mean accuracy measurements were pretty accurate, which was reflected in the smoothness of the curve in our Accuracy vs Training Set graph. With regards to time efficiency, our model took longer to train when more training data was used; the training times spanned from 3 to 13 seconds when trained on 10% to 100% of the training data respectively.

Table 2.1: Accuracy and Time of Naive Bayes Digit Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Training Time (sec)	Testing Time (sec)	Total Time (sec)
10%	58.60	1.77	2.87	111.61	114.49
20%	65.12	1.85	5.37	144.22	149.58
30%	69.28	0.99	6.34	214.24	220.58
40%	71.06	1.08	9.02	224.34	233.36
50%	72.58	0.98	9.09	206.67	215.75
60%	73.14	0.63	10.62	267.15	277.77
70%	73.94	0.69	13.26	289.50	302.75
80%	73.80	0.43	12.93	221.92	234.85
90%	74.46	0.47	11.64	224.00	235.65
100%	74.70	0.00	12.70	201.94	214.64



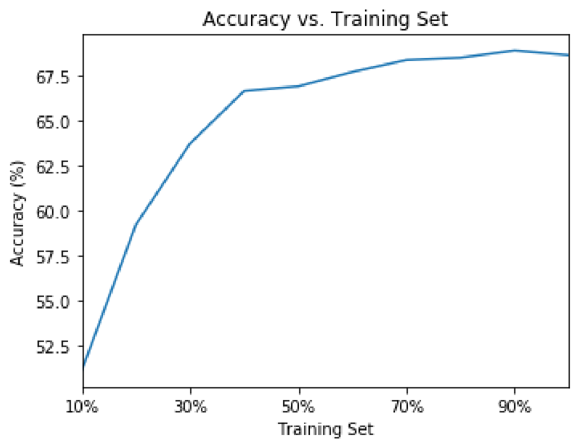


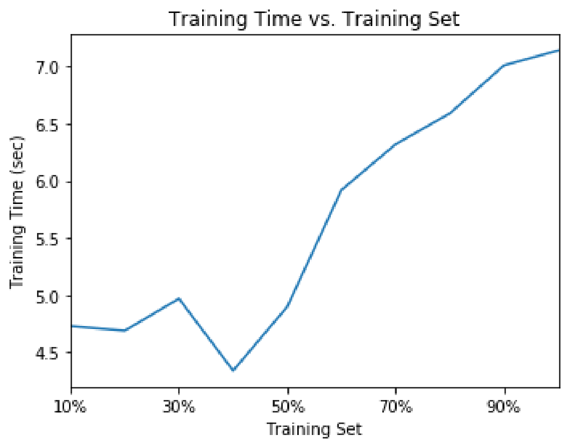
2.4 Face Classification

For face classification, our goal was to determine whether any observation was an image of a human face. Our feature selection was similar to that of digit classification in that we divided a face image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition was considered as one feature. Since each face image was 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. Table 2.2 shows the accuracy and time it took to train and test our model using various divisions of our training set over five iterations. Like our Naive Bayes model for digit classification, the accuracy of this model increased while the standard deviation decreased as more of the training set is used to train our model. The time it took to train our model also increased as more training data was used. Overall, our face classification model followed the same characteristics as the digit model with respect to accuracy and training time.

Table 2.2: Accuracy and Time of Naive Bayes Face Classification

	Mean(Accuracy) (%)	Std(Accuracy) (%)	Training Time (sec)	Testing Time (sec)	Total Time (sec)
Training					
10%	51.07	0.68	4.73	27.64	32.37
20%	59.20	1.29	4.69	27.00	31.69
30%	63.73	1.50	4.97	27.22	32.18
40%	66.67	0.73	4.34	23.29	27.63
50%	66.93	1.37	4.90	23.59	28.49
60%	67.73	2.09	5.92	24.27	30.19
70%	68.40	0.80	6.32	23.98	30.30
80%	68.53	0.65	6.59	23.88	30.46
90%	68.93	0.33	7.01	23.83	30.84
100%	68.67	0.00	7.14	23.31	30.46





3 Perceptron Model

3.1 Algorithm

The perceptron is a single layer neural network and is a classification algorithm that makes its predictions based on a linear predictor function that combines a set of weights with a feature vector. For our perceptron model, we predict the classification of an image using the following method:

- Extract features from the training dataset and store them in a feature set $\phi(x)$.
- Initialize function weights $\{w_j\}$; note that they may be initialized to 0.
- For each data point i.e image, (x_i, y_i), in our training dataset:
  - For each class label y^j, calculate $f(x_i, w) = w_0 + w_1\phi_1(x_i) + w_2\phi_2(x_i) + w_3\phi_3(x_i) + . . . + w_l\phi_l(x_i)$.
  - Select the class label y^j with the highest $f(x_i, w)$ value.
    - If y^j=y_i, then nothing needs to be done and we move to the next example (x_{i+1}, y_{i+1}).
    - Else, update the weights $\{w_{y_i}, w_{y^j}\}$:
      - For y_i class label, $w_{y_i} = w_{y_i} + \phi(x_i)$
      - For y^j class label, $w_{y^j} = w_{y^j}$ - $\phi(x_i)$
  - Repeat this process three more times on the training dataset.

After training our model, we return the final weights of each class label. Using these weights, we calculate the $f(x_i, w)$ value for each image in our test dataset and predict the y_i as the class label with the highest $f(x_i, w)$.

3.2 Method

We use five methods to train and test our perceptron model: partition(), feature_ext(), return_high_label(), train_perceptron(), and test_perceptron(). The partition() and feature_ext() methods are the same exact methods used in our Naive Bayes model. We have an additional method, return_high_label(), to return the class label with the highest$f(x_i, w)$ value. Our train_perceptron() method trains our model, returns the trainining time, and computes weights for various training sets. Note that train_perceptron() trains our model three times for one training set because any more training iterations may cause overfitting and decrease our testing accuracy. With the help of test_perceptron() method, we can compute the accuracy of our trained perceptron model and return the time taken to test the dataset. 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.

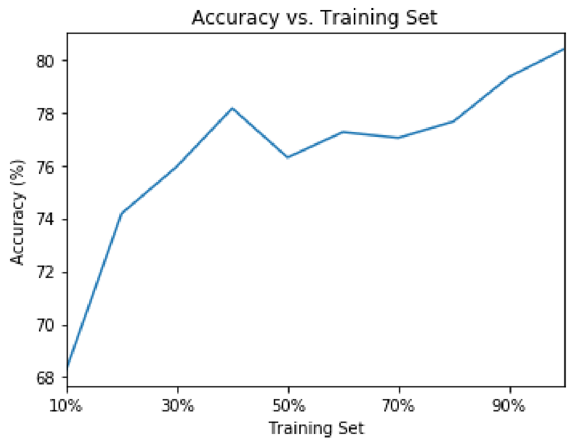
3.3 Digit Classification

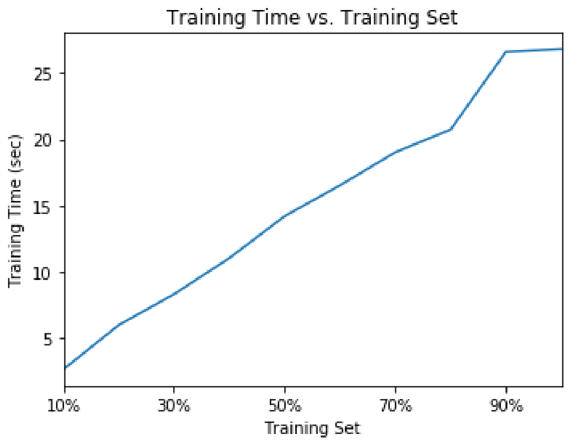
Our feature selection followed the same format as Naive Bayes by dividing a digit image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition was considered as one feature. Since each digit image was 28 x 28, we divided the images into 7 x 7 partitions for a total of 16 features similar to our Naive Bayes classification. Unfortunately, using 100% of the training set, we only got an accuracy of 52.74%. We decided to narrow the size of each feature to a 4 x 4 for a total of 49 features and improved our accuracy to 76.32%. Finally, we used 1 x 1 partitions for a total of 784 as features to get an accuracy of 80.40%. This turned out to be the best option out of our feature testing and increased our model's overall accuracy from 52.74% to 80.40%.

Table 3.1 shows the accuracy and time it takes to train and test our model for various divisions of our training set over five iterations. In the graph below, both training time and accuracy strictly increased with an increase in the number of training images. There was a slight dip in accuracy when using 50% of training data, which can be explained by the spike in standard deviation of the accuracy measurement when using 50% of the training data. After this slight dip, the accuracy continued on an upwards trend before peaking at 80.40% when using the 100% of the training data. Overall, the standard deviation values were larger than that of our Naive Bayes models, which was evidenced by a less smooth curve in the Accuracy vs Training Set graph. With regards to time efficiency, our model took longer to train when more training data was used; the training times spanned from 3 to 27 seconds when trained on 10% to 100% of the training data respectively.

Table 3.1: Accuracy and Time of Perceptron Digit Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Training Time (sec)	Testing Time (sec)	Total Time (sec)
10%	68.28	3.86	2.63	2.14	4.77
20%	74.18	4.38	5.98	2.56	8.53
30%	75.96	0.79	8.31	2.39	10.70
40%	78.16	1.71	11.03	2.47	13.50
50%	76.30	3.76	14.20	2.42	16.62
60%	77.26	2.67	16.52	2.37	18.89
70%	77.04	1.23	19.02	2.25	21.27
80%	77.66	2.04	20.73	2.29	23.02
90%	79.34	1.73	26.59	2.46	29.05
100%	80.40	1.78	26.81	2.27	29.08





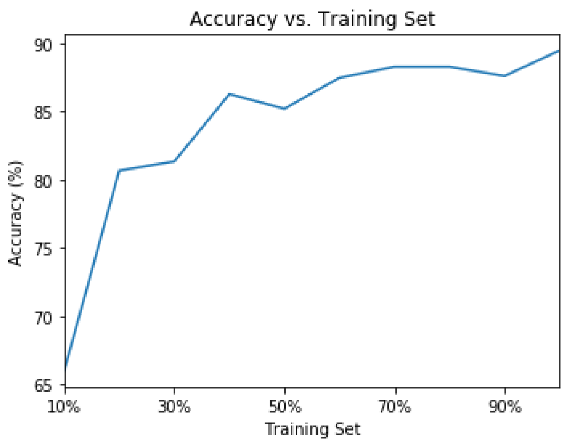
3.4 Face Classification

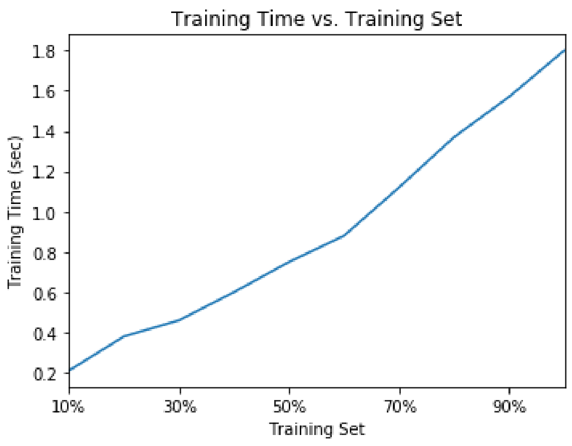
Our feature selection again divided a face image into n x n dimension partitions such that the sum of colored pixels (1s) within a partition was considered as one feature. Since each face image was 60 x 70 (width x height), we divided the image into 1 x 1 partitions (each pixel as a feature) for a total of 4200 features; using the full training set, we got an accuracy of 87.20%. We experimented with 2 x 2 partitions for a total of 1050 features and got an increased accuracy of 89.47% using the 100% training dataset.

Table 3.2 shows the accuracy and time it took to train and test our model for various divisions of our training set over five iterations. Like our perceptron model for digit classification, the accuracy and training time increased with the increase in the number of training images. Again note that the unexpected dip in accuracy at 50% was a product of the high standard deviation of 5.10% for the mean accuracy for that measurement. After that dip in accuracy, the accuracy increased much more consistently before reaching the max accuracy at 89.47%. The standard deviation values were again larger than that of our Naive Bayes models, which was evidenced by a less smooth curve in the Accuracy vs Training Set graph. With regards to time efficiency, our model took longer to train when more training data was used; the training times spanned from 0.21 to 1.8 seconds when trained on 10% to 100% of the training data respectively.

Table 3.2: Accuracy and Time of Perceptron Face Classification

Training	Mean(Accuracy) (%)	Std(Accuracy) (%)	Training Time (sec)	Testing Time (sec)	Total Time (sec)
10%	66.00	6.76	0.21	0.39	0.60
20%	80.67	6.23	0.38	0.40	0.78
30%	81.33	4.15	0.46	0.39	0.85
40%	86.27	2.82	0.60	0.38	0.98
50%	85.20	5.10	0.75	0.38	1.14
60%	87.47	3.19	0.88	0.39	1.27
70%	88.27	2.59	1.12	0.40	1.52
80%	88.27	1.24	1.37	0.45	1.82
90%	87.60	3.64	1.57	0.45	2.02
100%	89.47	2.04	1.80	0.46	2.25





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 of that observation through a designated "distance" metric. The distance metric for our model is the sum of the difference in pixels between two images. Consider two different images of the digit 3 that have been converted into 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. In the ideal case where the two images are identical, the calculated distance between the two images would be zero. Our K nearest neighbors are the K images that have the smallest distance with respect to our test image. Of these K nearest neighbors, we select the majority class as our final prediction for that observation. 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. Since KNN is instance-based learning, our training/testing procedure is wrapped up in one method that finds the distance between any test image and all the training images; it then chooses the majority class of the K images with the smallest distance. Note that since a tie is much more likely to occur between majority classes in KNN as compared to our earlier model; our method handles tie breakers through Python's list's inherent ordering by sorting our K nearest neighbors by distance and selecting the class in the first index. We use a separate training/testing method for digit and face classification (train_test_knn and train_test_knn2) since there are differences in feature selection for our two models.

4.3 Digit Classification

As explained earlier, our features for this model is the distance metric that is calculated by comparing two images pixel by pixel. We initially choose k=7 for the number of neighbors when testing our model. 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 instead of five 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 test observations each time for more consistency.

Table 4.1 shows the accuracy and time it takes to train and test our model for various divisions of our training set over three iterations. Figure 4.1 shows two unexpected dips in accuracy when using only 30% and 60% of the training set. This can be explained through the standard deviation in accuracy measurements between 10-50% that range between 1 and 2. There must have been iterations where a more extreme accuracy measurement pulled the mean accuracy away from it's supposed value. Do note that due to time restrictions, the lack of training/testing iterations and a smaller testing set of 50 observations plays a major contribution to this less stable accuracy trend as well. A clear example of this is the exact same three mean accuracies when using 60-80% of the training set due to a combination of high model accuracy and lack of testing set observations. Our model's peak accuracy when using 100% of the training data is 92%.

Since any calculations are done while testing observations, our KNN model does not have a "training time", and we look at the overall time efficiency of our model by looking at the training/testing step wrapped together. One would expect that as more training data is used, there would be an increase in training/testing time for our model since each test observation is compared to more training points. Unfortunately, there doesn't seem to be a clear pattern to our total training/testing times for our model. Our intuition on the cause of this phenomena is that the sorting component of our model may vary in that certain testing images may have a distance array that takes less time to sort in comparison to others. Overall, the average runtime per iteration is 27 minutes for a total of 13.3 hours over thirty iterations. This long training/testing time supports our algorithm explanation how KNN needs to iterate through each training observation for each new test observation to find the K nearest neighbors.

Table 4.1: Accuracy and Time of KNN Digit 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

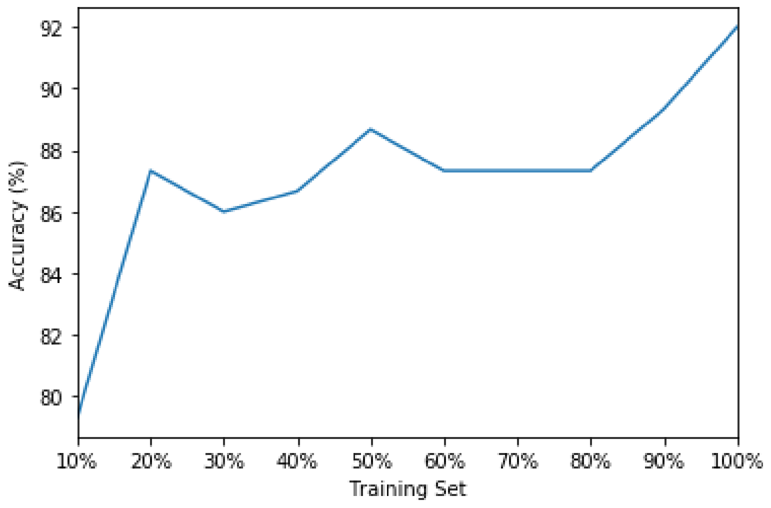


Figure 4.1: Accuracy at Various Training Set Divisions

4.4 Face Classification

We used the same distance metric that compares two images pixel by pixel for face classification as well. We chose k=7 for the number of neighbors for this model due to its efficacy in digit classification. 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 were 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% when trained on the full training set.

Table 4.2 shows the accuracy and time it takes to train and test our model for various divisions of our training set over three iterations. Figure 4.2 shows an unexpected decrease in accuracy when using 40% and 50% of the training set. 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 tell 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% as more of the training set is sampled, leading to a steady increase in accuracy from there onwards. Our model’s peak accuracy when using 100% of the training set is 74%. 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.

With respect to the time efficiency of our model, we again see an uneven training/testing time as we increase training data partitions. Similar to digit classification, we believe the sorting component of our algorithm plays a part to this randomness to the training/testing time of our KNN model. The average runtime per iteration is 10 minutes for a total of 5 hours over thirty iterations. This long training/testing time again supports our algorithm explanation how KNN needs to iterate through each training observation for each new test observation to find the K nearest neighbors.

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

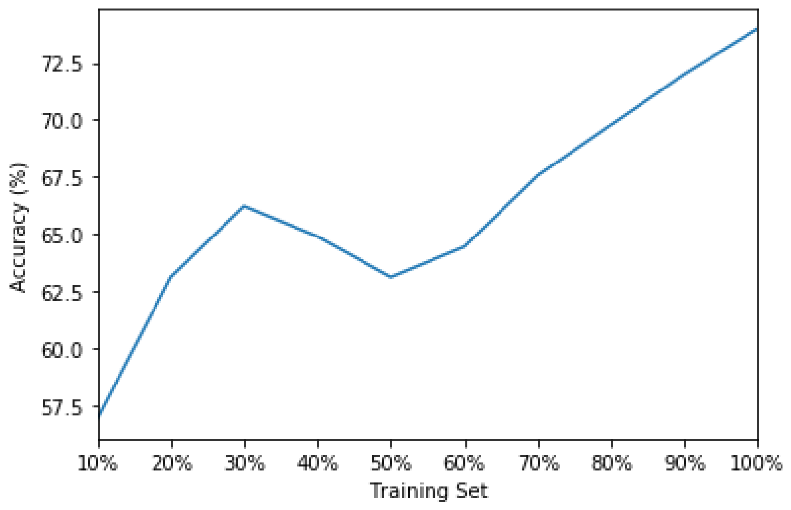


Figure 4.2: Accuracy at Various Training Set Divisions