

ENEE436 Project One

Yinjun Zhang

Intro to MNIST	1
What is MNIST?	1
Problem One: Naive Bayesian Classifier	2
What is a Naive Bayesian Classifier (with a Gaussian Assumption)?	2
Naive Bayesian Classifier Step-By-Step	2
Naive Bayesian Classifier Error Rates and Analysis	3
Problem Two: N Nearest Neighbors Classifier	4
What is the N Nearest Neighbor Classifier?	4
N Nearest Neighbor Classifier Step-By-Step	4
N Nearest Neighbor Error Rates, Plots, and Analysis	5
Problem Three: Fisher's LDA Classifier	7
What is Fisher's LDA Classifier?	7
Fisher's LDA Classifier Step-By-Step	9
Fisher LDA Error Rates and Analysis:	11
Problem Four: PCA Dimensionality Reduction	11
What is PCA Dimensionality Reduction?	11
PCA Dimensionality Reduction Step-By-Step	12
PCA Dimensionality Reduction Plots, Outputs, and Analysis	13

Intro to MNIST

What is MNIST?

The MNIST contains a publicly available database of handwritten digits labeled 0-9. This dataset is ideal for people learning to implement and analyze various machine learning and classification techniques. The dataset is composed of 60,000 training examples and 10,000 testing examples. Each example is a 28x28 matrix of pixels, with each element holding a value between 0-255. Furthermore, each example is labeled with a class value ranging from 0-9.

Problem One: Naive Bayesian Classifier

What is a Naive Bayesian Classifier (with a Gaussian Assumption)?

$$p(C_k | \mathbf{x}) = \frac{p(C_k) p(\mathbf{x} | C_k)}{p(\mathbf{x})}$$

Figure 1.1: Bayes' Theorem Applied to Naive Bayesian Classifier.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Figure 1.2: Gaussian Probability Density Function.

A Naive Bayesian Classifier works by leveraging Bayes' Theorem (Figure 1.1) to calculate the probability that a piece of data belongs to a certain class based on prior knowledge (previous data). According to Bayes' Theorem, the posterior probability that the data point belongs to class k , $P(C_k|x)$, is equal to the likelihood of the data given class k , $P(x|C_k)$, times the class prior probability, $P(C_k)$, divided by the evidence, $P(x)$. The classifier then assigns the new data point to the class k with the greatest posterior probability, $P(C_k|x)$. Since $P(x)$ will be the same when calculating each posterior probability, we can remove it when trying to find the class with the greatest posterior probability. Hence, all that's left is to calculate $P(C_k)$ and $P(x|C_k)$. Class prior probability $P(C_k)$ can be calculated by taking the ratio of data points of class k over the total number of data points. However, the likelihood, $P(x|C_k)$ is a bit more tricky, and we can only calculate this value by making an assumption regarding the data.

For this project, we assume the likelihood follows a Gaussian probability density function, and we can calculate $P(x|C_k)$ by plugging in our data into the equation seen in Figure 1.2. Note that in order to calculate the Gaussian probability, we need to find the mean and variance of the dataset for each class, as they are variables required in the equation. Furthermore, note that this classifier is called naive because each attribute is assumed to be conditionally independent, meaning the likelihood of each attribute is simply multiplied for each class to obtain the final

probability. However, this is a strong assumption to make, and attributes often aren't independent, hence the name naive Bayes.

Naive Bayesian Classifier Step-By-Step

The first step in performing a Naive Bayesian Classifier is to group the training data by class (or label). Then, for each class, we need to find the mean and covariance associated with that label. We will use these mean and covariance values later while computing the Gaussian probability (see Figure 1.2).

In the next step, we compute the posterior probability for each class k given a new input data vector x . The input data will be assigned to the class with the largest posterior probability. As defined in the previous section, for the purposes of classification, all we need to do is solve the equation $P(C_k|x) = P(C_k) * P(x|C_k)$. To calculate $P(C_k)$, we simply take the ratio of data points in class k over the total number of points. To calculate $P(x|C_k)$, we assume a Gaussian probability density function, and we plug each individual attribute of x as well as the associated mean and covariance for class k into the equation seen in Figure 1.2. Then, because we assume all attributes are independent, $P(x|C_k)$ is simply the product of all attributes of x after plugging it into the Gaussian probability density function, or $\prod P(x_i|C_k)$. Now that we have solved $P(C_k|x)$ for class k , we proceed to do this for each class, and we assign the input data vector x to the class which yields the highest posterior probability.

For this project, I used the sklearn python library to build my Naive Bayesian Classifier so I didn't have to implement it from scratch. Using the sklearn's Naive Bayesian Classifier, I just had to pass in the training and test data, as well as the input data that I wanted to predict, and it did all the fitting and processing described above for me. In the next section, we will analyze the output and error rates produced by the Naive Bayesian Classifier.

Naive Bayesian Classifier Error Rates and Analysis

```
(base) bawhun@ Project1 % python3 naive_bayesian.py
NAIVE BAYESIAN CLASSIFIER
TRAINING DATA - Num. mislabeled points: 26106 / 60000
TEST DATA - Num. mislabeled points: 4442 / 10000
```

Figure 1.3: Error Rate Output for Naive Bayesian Classifier

Running the training data through the Naive Bayesian Classifier, we get 26,106/60,000 mislabeled points, or an error rate of 43.51%. Running the test data through the Naive Bayesian Classifier, we get 4,442/10,000 mislabeled points, or an error rate of 44.42%. First, note that the error rate for the training data is lower than the error rate for the test data. This is expected because the Naive Bayesian Classifier is better fitted to the data it was trained on compared to new test data. Furthermore, we observe that the error rates for both training and test data are very close to each other, suggesting that the Naive Bayesian Classifier does a good job predicting new data compared to looking at data it's already seen. Finally, we note that the error rates for both training and test data hover around 45% which is considerably high compared to some of the

other classifiers we analyze later in this report. However, this is also expected because a naive Bayesian Classifier assumes attribute independence and simply takes the product of all the attribute likelihoods. This is a very strong assumption to make, and it tends to lead to higher error rates when the attributes are in fact dependent (which they are for MNIST data).

Problem Two: N Nearest Neighbors Classifier

What is the N Nearest Neighbor Classifier?

The N Nearest Neighbor Classifier follows a very simple algorithm: for each data point in the input data vector, locate the N closest points from the training data, and assign this new data point to the class with the most occurrences out of these N closest points. N is an arbitrary number chosen by the analyst, and in this project we run the Nearest Neighbor Classifier for varying values of N to analyze which value performs the best. In addition, the definition of “closest points” also depends on the data. Generally, Euclidean Distance is used to measure the distance between data points, and the N points with the smallest Euclidean Distance relative to the input data point are considered the N closest points. For this project, we assume distance is measured by Euclidean Distance.

N Nearest Neighbor Classifier Step-By-Step

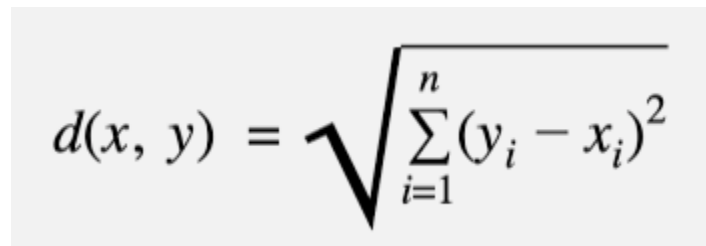

$$d(x, y) = \sqrt{\sum_{i=1}^n (y_i - x_i)^2}$$

Figure 2.1: Formula for n dimensional Euclidean Distance.

The first step in using the Nearest Neighbor Classifier is to define a convention for measuring distance between points. Then, using this convention, you can determine which points are “closest” to the input data point. Euclidean Distance is measured by taking the element-by-element difference between two data points, summing their squares, then square-rooting the summation.

The next step is to choose a number for N. N represents the number of closest neighbors you look for in the training data that have the shortest Euclidean Distance to the input data point. Then, after finding those N neighbors, you can classify the new input data by assigning it to the class with the most occurrences within those N points. For this project, we run the Nearest Neighbor Classifier on multiple values of N (1, 5, 10, 20, 50, 100), and plot the training and error rates for each N value to analyze the effect changing N has on prediction accuracy.

The last step is to run the Nearest Neighbor Classifier on the input data vector. For each input data point, the algorithm must go through every point of the training data, compute the Euclidean Distance between the training data point and input data point, and sort the results. Finally, the algorithm will take the first N points with the smallest Euclidean Distance and make a classification predicting the class of the input data point.

For this project, I leveraged the sklearn Nearest Neighbor classifier so I didn't have to write it from scratch. Using the sklearn Nearest Neighbor classifier, I just had to pass in the training data, the input data I wanted to predict, and a value for N, and the algorithm did all the processing and classification described above for me. Finally, I plotted the error rates for training and test data compared to selected values of N. In the next section, we will analyze the output, error rates, and plots produced by the Nearest Neighbor Classifier.

N Nearest Neighbor Error Rates, Plots, and Analysis

```
N NEAREST NEIGHBORS CLASSIFIER
NUM NEIGHBORS: 1
TRAINING DATA - Num. mislabeled points: 0 / 60000
TEST DATA - Num. mislabeled points: 309 / 10000
NUM NEIGHBORS: 5
TRAINING DATA - Num. mislabeled points: 1085 / 60000
TEST DATA - Num. mislabeled points: 312 / 10000
NUM NEIGHBORS: 10
TRAINING DATA - Num. mislabeled points: 1500 / 60000
TEST DATA - Num. mislabeled points: 335 / 10000
NUM NEIGHBORS: 20
TRAINING DATA - Num. mislabeled points: 1957 / 60000
TEST DATA - Num. mislabeled points: 375 / 10000
NUM NEIGHBORS: 50
TRAINING DATA - Num. mislabeled points: 2782 / 60000
TEST DATA - Num. mislabeled points: 466 / 10000
NUM NEIGHBORS: 100
TRAINING DATA - Num. mislabeled points: 3521 / 60000
TEST DATA - Num. mislabeled points: 560 / 10000
```

Figure 2.2: Error Rate Output for Nearest Neighbor Classifier

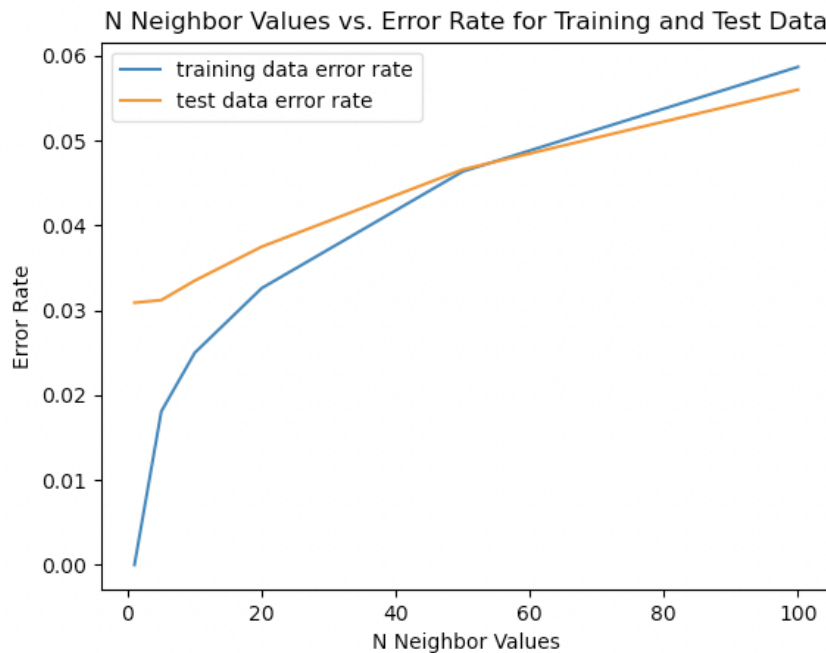


Figure 2.3: Plot of N Neighbor Values vs. Training and Test Data Error Rates.

Looking at the program output (Figure 2.2) as well as the plot (Figure 2.3), we can see that the number of mislabeled points for both training and test data is lowest when $N = 1$. From there, the number of mislabeled points continues to increase as N increases. Based on the data, this implies that the error rate of the Nearest Neighbor Classifier increases as the value of N increases. From here, let's analyze the results of the training data predictions and the test data predictions separately, as they have different expected behaviors.

First, looking at the training data error rates, we see that the error rate starts at 0%, then continues to grow rapidly as N increases. This makes sense because when $N = 1$, the closest neighbor to any point in the training data is that point itself, so the prediction will always be accurate. However, as you start increasing the value of N , other neighbors will start to pollute this value, yielding the blue curve we see in Figure 2.3.

Interestingly, the results for the test data error rates aren't what we would traditionally expect. Normally, a value of N that is too high or too low will yield higher error rates when trying to predict a new set of data. When a value of N is too low, we run into a phenomenon called overfitting, in which the model is fit too closely to the training data, such that it is no longer making generalized predictions for any new data. On the other hand, when the value of N is too high, the classifier is no longer looking at common attributes between neighbors in a specific area, it's just taking a large weighted average where the most common class wins. Generally, the most optimal value for N is the square root of the length of the training data. This guarantees enough values to build a solid sample size while not being too large to the extent where these points are no longer representative of the given input data point. However, in this case, the error rates for the test data are lowest when $N = 1$ and continue to grow linearly as N increases,

diverging from the expected behavior explained above. One plausible explanation for this behavior is that the test data is very similar to the training data. As such, the test data will follow a similar trend as the training data described in the previous paragraph. Because the test data and training data are so similar, generally the closest training data point to each test data point will accurately represent the class of that test data point. Adding more points will just pollute this value, increasing the error rate as N increases.

Finally, there are a couple more things worth noting about this classifier. First, even in the worst case, the Nearest Neighbor Classifier still had error rates below 10%. Compared to the ~45% error rate we saw in the Naive Bayesian Classifier, that's a substantial improvement in classification accuracy. Now, we can better understand why the first algorithm is considered naive and how Nearest Neighbors does a better job of accounting for dependencies between attributes. However, this better accuracy doesn't come without a cost: because Nearest Neighbor requires each input data point to be compared against every data point in the training data, the runtime for this algorithm is extremely slow. Running it on my laptop took multiple hours compared to the other algorithms which finished in a matter of minutes.

Problem Three: Fisher's LDA Classifier

What is Fisher's LDA Classifier?

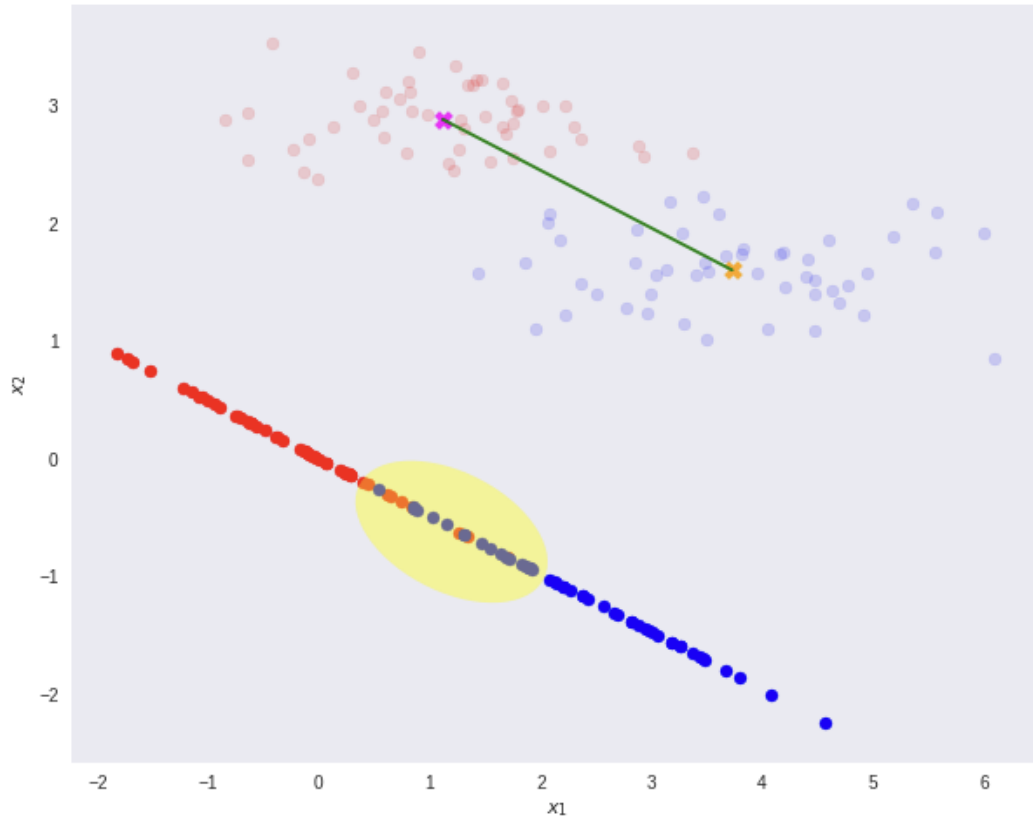


Figure 3.1: 2D Data Projected onto 1D Space.

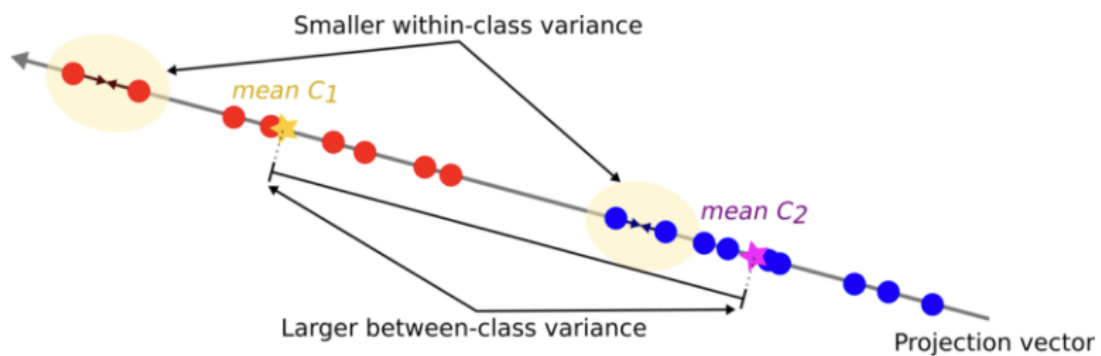


Figure 3.2: Two Classes of Data Transformed into 1D Space via Fisher's Classifier.

$$J(\mathbf{W}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}$$

Between-class variance

Within-class variance

$$s_k^2 = \sum_{n \in C_k} (y_n - m_k)^2 \quad y_n = \mathbf{W}^T x_n$$

$$J(\mathbf{W}) = \frac{\mathbf{W}^T S_B \mathbf{W}}{\mathbf{W}^T S_W \mathbf{W}}$$

$$\mathbf{W} \propto S_W^{-1} (m_2 - m_1)$$

Figure 3.3: Derivation of Transformation W that Produces Optimal Fisher Transformation.

An LDA Classifier, or Linear Discriminant Analysis Classifier, works by projecting a dataset of D dimensions onto D' dimensions (where D' < D) by multiplying the original dataset by a transformation matrix W. For the purposes of this project, we will be classifying input data as one of two digits, meaning we're dealing with a two-class classification problem. Then, we define a threshold that separates the two classes. Now, classifying a new input data vector is as simple as multiplying our input data vector by the same transformation matrix W and comparing the resulting value with the aforementioned threshold. If the output value is greater than the threshold, we classify it as the first class, otherwise we classify it as the second class. However, we cannot just choose an arbitrary linear transformation and transformation matrix W. For example, looking at Figure 3.1, if we choose an arbitrary transformation, it's possible that the points will overlap when projected onto D' dimensions, making it difficult to define a clear threshold. This is where Fisher's LDA Classifier comes into play.

The goal of Fisher's Classifier is to keep these two classes of data as far away from each other as possible when projected onto D' dimensions. That way, it's easier to define a clear threshold that separates the two classes, making classification much easier and predictions much more accurate. Formally, this means maximizing the between-class variance while minimizing the within-class variance (see Figure 3.2). Formally, Fisher's LDA can be defined by the first equation in Figure 3.3, where the transformation matrix W that produces the largest value $J(W)$ is the transformation matrix that should be applied to the training and input data. Looking at the right side of the equation, we can once again see that we're trying to maximize the between-class variance (numerator) while minimizing within-class variance (denominator). Looking at the derivations in Figure 3.3, we conclude that the optimal transformation matrix is proportional to the inverse of the within-class covariance matrix times the difference of the class means. Now that we've constructed the optimal transformation and separated the classes as much as possible, we can go ahead and perform the classification process outlined above on new data.

Fisher's LDA Classifier Step-By-Step

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n \quad m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n$$

Figure 3.4: Finding the Mean Vectors.

$$S_W = \sum_{k=1}^K S_k$$

$$S_k = \sum_{n \in C_k} (x_n - m_k)(x_n - m_k)^T$$

$$S_B = \sum_{k=1}^K N_k (m_k - m)(m_k - m)^T$$

$$W = \max_{D'} (eig(S_W^{-1} S_B))$$

Figure 3.5: Calculating S_B and S_W . Use Them to Solve for W .

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Figure 3.6: Gaussian Normal Distribution for Calculating Threshold.

First, we need to group our training data by class so we can perform class-based calculations. In this case, since we're doing two-class predictions, we just need to isolate the data that belong to either of these two classes. Then, we calculate the mean vector for both of these classes based on training data (see Figure 3.4). From there, we calculate the S_B and S_W matrices needed to maximize the difference between means of different classes while minimizing the variance within each class (see Figure 3.5). From there, we can construct our transformation matrix W according to the last equation in Figure 3.5. With the transformation matrix W , we take the first D' eigenvectors and eigenvalues to construct a transformation onto D' dimensions. After applying this transformation onto our training data, we can calculate a threshold using the Gaussian Normal Distribution (see Figure 3.6), allowing us to classify future input data. Finally, we just have to apply the same dimensional reduction transformation to any input data we want to predict, and compare the resulting value to our defined threshold. If it's greater than the threshold, we can classify the input data as class one. If it's less, we can classify the data as class two.

For this project, I used the sklearn library's LDA so I didn't have to implement the classifier from scratch. The sklearn LDA handled all of the processing, fitting, and classifying outlined above. I just had to do some preprocessing and formatting to create the desired training and test datasets that I wanted to pass into the LDA. This mostly just consisted of isolating the data that belonged to the two classes I was aiming to compare. Since the original dataset contains labels for values 0-9, but we're only using LDA to classify between two digits, I had to figure out which data points belonged to those two desired digits and discard the rest. From there, I passed the isolated data into the LDA as well as my input data, and the LDA returned the prediction regarding which digit each input data point represented. In the next section, we will analyze the error rates and outputs of the Fisher LDA Classifier.

Fisher LDA Error Rates and Analysis:

```
(base) bawhun@ Project1 % python3 lda.py
FISHER LDA TWO-CATEGORY CLASSIFIER
Digit: 0 vs. digit: 9
TRAINING DATA - Num. mislabeled points: 59 / 11872
TEST DATA - Num. mislabeled points: 23 / 1989
Digit: 0 vs. digit: 8
TRAINING DATA - Num. mislabeled points: 133 / 11774
TEST DATA - Num. mislabeled points: 20 / 1954
Digit: 1 vs. digit: 7
TRAINING DATA - Num. mislabeled points: 91 / 13007
TEST DATA - Num. mislabeled points: 23 / 2163
```

Figure 3.7: Error Rate Output for Fisher LDA Classifier.

Looking at the error rates for the two-digit Fisher LDA Classifier, we notice that the number of mislabeled points is extremely small, especially compared to previous classifiers. The smallest error rate is 0.5% while the largest error rate is 1.16%. That's just over one percent! Observing such small error rates suggests that the Fisher LDA transformation works really well for this dataset. The Fisher LDA successfully transformed the data such that the variance between each class was small while the variance between classes was large, creating a clear threshold to accurately classify new data.

Furthermore, note that this classifier was only used for two-class comparisons while the previous classifiers had to choose between multiple classes. Having less classes to choose from reduces the overall error rate because each additional class compounds the probability for error. For example, assuming a random uniform distribution, the probability for guessing the right class is 50% for 2 classes, but only 10% for 10 classes. Similarly, it makes sense that the error rates for the two-class Fisher LDA Classifier are lower than any of the other classifiers because this classifier is only predicting between two classes.

Problem Four: PCA Dimensionality Reduction

What is PCA Dimensionality Reduction?

PCA, or Principal Component Analysis, is a method for reducing the dimensionality of data, particularly large data sets. The aim of PCA is to reduce the dimensionality of large data sets while preserving as much meaningful data as possible. By reducing the dimensionality, large data sets can become easier to visualize, analyze, and process. For the purpose of this project, we will be applying PCA to the 784-attribute training and test data sets, running different classifiers on them, then observing the differences between classifying the original dataset compared to the PCA-reduced dataset.

PCA Dimensionality Reduction Step-By-Step

The first step in performing PCA Dimensionality Reduction is calculating the mean of every column of the input matrix. Next, we center the values in each column of the input matrix by subtracting the mean of that column. Next, we take the covariance matrix of our centered input matrix. Finally, we calculate the eigendecomposition of this covariance matrix. Sorting the eigenvectors in descending order by eigenvalue magnitude, we establish a ranking of the components or axes for the new subspace of the original input matrix. From here, we choose the first k eigenvectors and their corresponding eigenvalues to transform the input matrix in subspace k . Finally, we multiply the transpose of our k principal components and the original input matrix to obtain our projected matrix onto subspace k . Furthermore, it's important to note that eigenvalues close to zero indicate components or axes that may be discarded with little to no loss in the original dataset.

For this project, we perform PCA on the training and test data. Both of these datasets have 784 attributes, and we will be reducing them to different subspaces, ranging from 3-100. For the first part of the project, we perform PCA on the training data, projecting it onto subspace 2 and subspace 3, then plotting both of these graphs. From there, we perform PCA on both the training and test data for subspace = 5, 10, 20, 50, 100. Then, we run the naive Bayesian and Nearest Neighbor Classifiers on these PCA reduced datasets, plotting and logging the differences in error rates as we change the target subspace of the PCA.

Instead of implementing PCA from scratch, I leveraged sklearn's PCA library. Sklearn's PCA handles all of the processing and linear algebra computations described above. All I had to do was pass in the subspace I wanted the PCA to reduce down to, as well as the input data for the PCA to reduce. From there, I took the PCA-reduced data, plotted it, and ran it against two different classifiers, plotting and logging the resulting error rates. For more information on the classifiers used (Naive Bayesian and Nearest Neighbor), see the corresponding sections above. In the next section, we will analyze these plots and outputs.

PCA Dimensionality Reduction Plots, Outputs, and Analysis

```
(base) bawhun@ Project1 % python3 pca.py
PCA PLOTTING AND CLASSIFICATION
PCA Classifications for n reduced to 5
BAYESIAN TRAINING DATA - Num. mislabeled points: 21274 / 60000
BAYESIAN TEST DATA - Num. mislabeled points: 7496 / 10000
NEAREST NEIGHBOR TRAINING DATA - Num. mislabeled points: 11182 / 60000
NEAREST NEIGHBOR TEST DATA - Num. mislabeled points: 7469 / 10000
PCA Classifications for n reduced to 10
BAYESIAN TRAINING DATA - Num. mislabeled points: 13778 / 60000
BAYESIAN TEST DATA - Num. mislabeled points: 8531 / 10000
NEAREST NEIGHBOR TRAINING DATA - Num. mislabeled points: 2731 / 60000
NEAREST NEIGHBOR TEST DATA - Num. mislabeled points: 8519 / 10000
PCA Classifications for n reduced to 20
BAYESIAN TRAINING DATA - Num. mislabeled points: 9547 / 60000
BAYESIAN TEST DATA - Num. mislabeled points: 8713 / 10000
NEAREST NEIGHBOR TRAINING DATA - Num. mislabeled points: 1134 / 60000
NEAREST NEIGHBOR TEST DATA - Num. mislabeled points: 8805 / 10000
PCA Classifications for n reduced to 50
BAYESIAN TRAINING DATA - Num. mislabeled points: 7724 / 60000
BAYESIAN TEST DATA - Num. mislabeled points: 8304 / 10000
NEAREST NEIGHBOR TRAINING DATA - Num. mislabeled points: 850 / 60000
NEAREST NEIGHBOR TEST DATA - Num. mislabeled points: 8617 / 10000
PCA Classifications for n reduced to 100
BAYESIAN TRAINING DATA - Num. mislabeled points: 7840 / 60000
BAYESIAN TEST DATA - Num. mislabeled points: 7760 / 10000
NEAREST NEIGHBOR TRAINING DATA - Num. mislabeled points: 958 / 60000
NEAREST NEIGHBOR TEST DATA - Num. mislabeled points: 8608 / 10000
```

Figure 4.1: PCA-Reduced Classification Output.

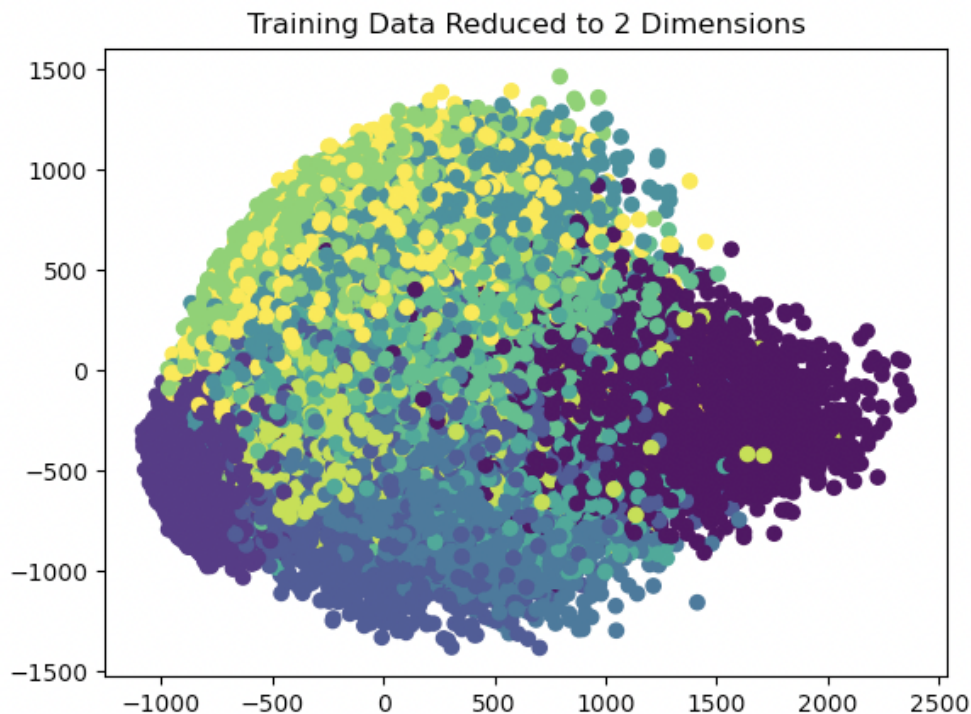


Figure 4.2: PCA-Reduced Training Data Plotted in 2D.

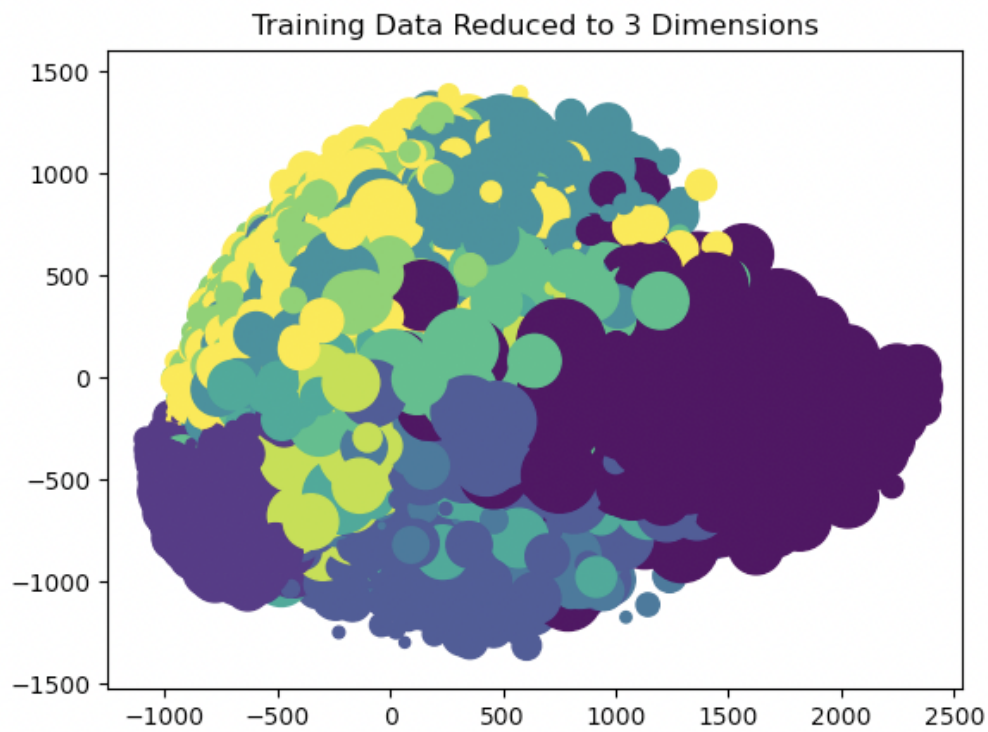


Figure 4.3: PCA-Reduced Training Data Plotted in 3D.

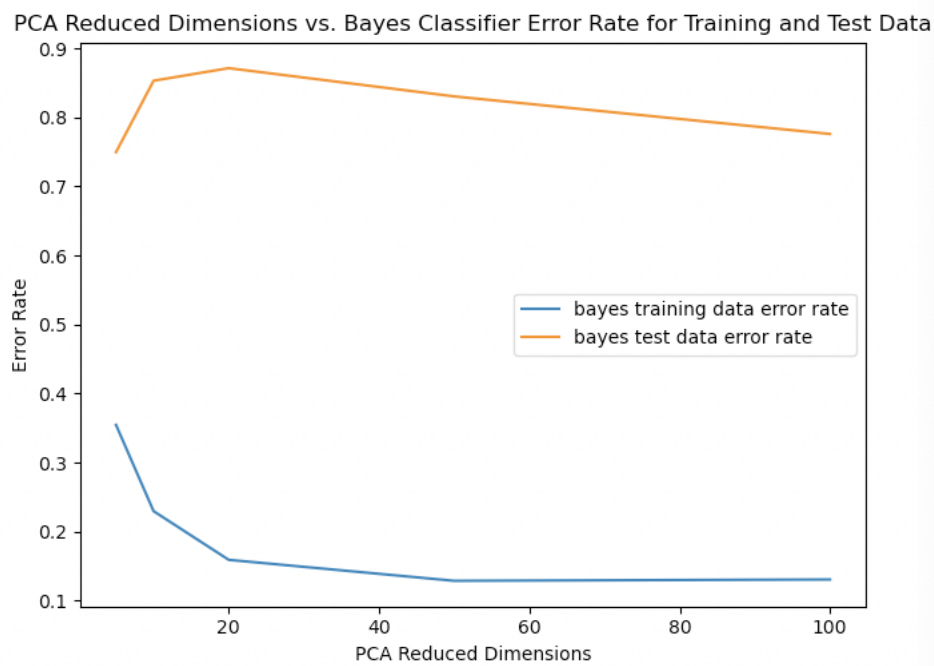


Figure 4.4: PCA-Reduced Data Error Rates After Naive Bayesian Classification.

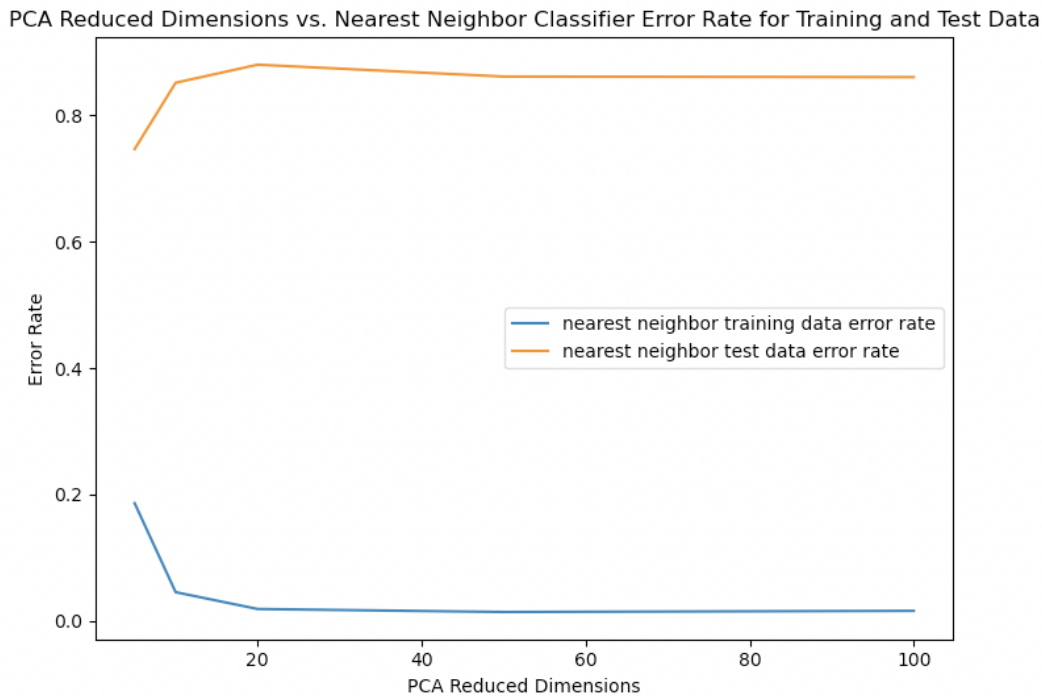


Figure 4.5: PCA-Reduced Data Error Rates After Nearest Neighbor Classification.

For the first part of the PCA assignment, we had to plot the training data reduced to two dimensions (Figure 4.2) and three dimensions (Figure 4.3). In both figures, the color of the point represents the class it belongs to. In the 3D plot, the size of the point represents the magnitude of the third dimension. While we can still make out some patterns and distinguishing features for each class, the data is largely clustered together and indistinguishable. This makes sense, as reducing a 784-attribute data point into two or three dimensions will inevitably lead to a lot of data loss, and these two plots reflect that.

For the next part of the PCA assignment, we had to run a Naive Bayesian Classifier and a Nearest Neighbor Classifier on PCA-reduced training and test datasets. From there, we altered the subspace the PCA was expected to reduce down to. We tested reduction values: 5, 10, 20, 50, 100, and plotted the corresponding error rates for both datasets against the PCA reduction value. To see the exact number of mislabeled points for each classifier under each reduction value, refer to Figure 4.1. However, Figures 4.4 and 4.5 showcase general trends of error rates for varying values of PCA reduction. Note that the Naive Bayesian Classifier (Figure 4.4) and the Nearest Neighbor Classifier (Figure 4.5), produced almost identical results, with the exception that the Naive Bayesian Classifier's error rates were higher by a small constant amount. However, based on our previous analysis of these two classifiers, this difference in error rates makes sense. More importantly, the two classifiers produced graphs with similar shapes, so we can analyze both of these graphs together.

First, looking at the training data error rate for both of these graphs, their curves drop dramatically as the PCA reduction value increases before tapering off. This makes sense, as we expect the error rate to go down as we restore more dimensionality to the original data set. Intuitively, the smaller we make the reduction subspace, the less representative the PCA-reduced dataset will be, and the higher the error rates will become, and the training dataset reflects that. Similarly, for the test datasets for both graphs, the error rates tend to go down as we increase the dimensionality of the PCA reduction. However, the test dataset curves have an anomaly: the error rates actually start lower for the smallest PCA reduction value tested: 5. The error rate then climbs for a short interval before dropping again, as expected. There could be many reasons for why this spike occurs before it resumes expected behavior, but one plausible explanation is that the test dataset simply lines up very well with a PCA-reduction value of 5. Looking at the graphs, we can see the rest of the curves follow our expected trajectory except for this value.

Finally, it's important to note how high the error rates are for the PCA-reduced test datasets. They generally fall in the 80-90% range! I think this makes sense, because the classifiers are fitted by PCA-reduced training data, which means the new training data is already not very representative of the original training data. From here, the test data is also PCA-reduced, losing a lot of its representation as well. As a result, you end up running a classifier on a reduced input that doesn't represent the original input very well, and your classifier is trained in reduced training data that also doesn't represent the original training data very well. Because of all these compound errors, we end up with very inaccurate predictions and error rates upwards of 80%.