AMATH 482: HOMEWORK 3

SOPHIA BARBER

Department of Applied Mathematics, University of Washington, Seattle, WA
sophiab3@uw.edu

Abstract. In this paper we will classify images of digits. We perform PCA on both the train and test data with k such that the low-dimensional reconstruction of the data from k-PC modes will retain 85% of the energy. Then we project the data into k-modes PCA space and use Ridge, KNN, and SVM classifiers to classify each image by digit, using cross-validation to tune hyper-parameters. We also restrict our data to only images of certain digits and apply the same process. We then evaluate the accuracy of each classifier on both the train and test data.

1. Introduction and Overview

In this report, our goal is to be able to take an image of a handwritten digit (0 through 9) and classify it by its number. Our data consists of both a training and a testing set. The training data consists of 60000 images of handwritten numbers 0 through 9, each with 784 (28 x 28) pixels. We have training labels, which identify each sample by digit. The testing set has 10000 images of the same format, and testing labels likewise. Our goal is to be able to predict the label (i.e. what digit) when given a particular image.

Our first step is to perform PCA on both the train and test data sets with k such that a low-dimensional reconstruction of the data from the first k PC modes will retain 85% of the energy in the original data. We then project both the train and test sets into k-modes PCA space and apply a classifier method, using cross validation to tune any hyper-parameters we have. We will use Ridge, KNN, and Naive-Bayes classifiers, and compare their efficacy. For all of these classifiers, we fit our model to our training data, and perform cross validation using only the training data. The testing data remains untouched until we have finished cross validation, and then we can test our model on the test set to determine how good our model is at classification of new images it has not seen.

We first perform this classification process on only a subset of the data (i.e. we only keep the images in both the train and test set whose labels are either of two particular digits) using the ridge classifier method. After doing this we perform classification on the full data set, with various classifiers.

2. Theoretical Background

Singular Value Decomposition (SVD) and Principal Component Analysis (PCA): To understand PCA we must understand SVD. Performing SVD on a matrix $A \in \mathbb{R}^{n\times m}$ is computing a factorization of A: $\exists$ orthogonal matrices $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{m\times m}$ and diagonal matrix $\Sigma \in \mathbb{R}^{n\times m}$ with positive entries for which $A = U\Sigma V^T$. The matrix U contains the left singular vectors of A as its columns, the matrix V^T contains the right singular vectors of A as its columns, and the matrix Σ has the singular values of A as its diagonal entries, and 0 elsewhere, with the largest singular value σ_1 in the (1,1) entry of Σ, and with $\sigma_2, ...$ placed in decreasing order as you go down the diagonal of Σ [1].

Now that we have performed SVD on A, which contains our data, we use it to perform PCA. To do PCA, we must center and scale our data, i.e. subtract the mean and divide by the variance, before we compute the SVD above. Assume that A was already pre-processed as such before we computed the SVD. The goal of PCA is to decompose the data into directions that maximize variance, and keep only so many of these directions. We want to build a low-rank projection of our data onto the directions of maximum variance that we kept [1]. Our left singular vectors (columns of U) are our PC modes, which we computed with SVD. These are the directions of most to least variance, with the first singular vector/PC mode being the direction of most variance, and the last singular vector/PC mode being the direction of least variance. Now we want to project our data (our matrix A) onto the first k of these vectors, i.e. into k-PCA space. Here, k is the number of singular vectors/PC modes we keep. This means that we are not keeping all the information,

Date: March 2, 2025.

only the information in the k directions of most variance, i.e. in the k directions that capture the most information. In effect what we are doing is truncating the SVD. From [1], if we write the SVD as

$$A = \sum_{i=1}^{min(m,n)} \sigma_i \vec{u_i} \vec{v_i}^T \tag{1}$$

where $\vec{u_i}$ is the i^{th} column of U and $\vec{v_i}$ is the i^{th} column of V^T, then the projection of A into k-mode PCA space (where $k \leq$min(m,n)) is the truncated SVD, which is effectively equation 1, but where $\sigma_{k+1}, ..., \sigma_{min(m,n)} = 0$.

$$A_k = \sum_{i=1}^{k} \sigma_i \vec{u_i} \vec{v_i}^T$$

Ridge Regression: Our goal is to find a model $f(\vec{x})$ that maps a point $\vec{x}$ into $\mathbb{R}^c$, where c will be the number of classes we have (explained below). Assume that we have m data points in our training set, $\{\vec{x_i}\}_{i=1}^m$, where each point is d-dimensional ($\in \mathbb{R}^d$). We have a set of labels $\{\vec{y_i}\}_{i=1}^m$, one for each data point, where each $\vec{y_i} \in \mathbb{R}^c$ (how these labels are generated is explained below). We want a function $f : \mathbb{R}^d \longrightarrow \mathbb{R}^c$ where $f(\vec{x_i}) \approx \vec{y_i} \quad \forall\, i \in \{1, ..., m\}$ [1]. Letting $Y \in \mathbb{R}^{m \times c}$ be the matrix whose rows are $\vec{y_i}^T$, and $X \in \mathbb{R}^{m \times d}$ be the matrix whose rows are $\vec{x_i}^T$, then we can write $f(X) : \mathbb{R}^{m \times d} \longrightarrow \mathbb{R}^{m \times c}$.

Our goal is to find f that gives the highest probability of having $f(\vec{x_i}) = \vec{y_i} \quad \forall\, i \in \{1, ..., m\}$, i.e. to find the maximum likelihood estimator (MLE) of f, which we denote f_{MLE} [1]. This is the function f such that we have the highest probability that our model (f) will yield Y when given X (i.e. f that maximizes $\mathbb{P}(Y|X)$). From [1], this is given by

$$f_{MLE} = \underset{f}{\operatorname{argmin}}(\frac{1}{2\sigma^2}||f(X) - Y||_2^2) = \sum_{i=1}^{m} ||f(\vec{x_i}) - \vec{y_i}||_2^2 = \sum_{i=1}^{m} \sum_{j=1}^{c} (f(\vec{x_i})_j - (\vec{y_i})_j))^2.$$

Since we are using ridge regression, we'll constrain f to be of the form $f(X) = A\beta$ where the ith row of $f(X)$ is $f(\vec{x_i}) = \vec{\beta_0} + \sum_{j=1}^{d} x_{ij} \vec{\beta_j}$ where each $\{\vec{\beta_j}\}_{j=0}^d \in \mathbb{R}^c$ [1]. The matrix $\beta \in \mathbb{R}^{(d+1) \times c}$ is the matrix whose rows are $\vec{\beta_j}^T$, and the matrix $A \in \mathbb{R}^{m \times (d+1)}$ is the matrix whose rows are $[1\ \vec{x_i}^T] \in \mathbb{R}^{1 \times (d+1)}$. Since we are using Ridge Regression to build our model, (from [1]) our problem reduces to finding the matrix

$$\beta_{MLE} = \underset{\beta}{\operatorname{argmin}}(||A\beta - Y||_2^2 + \alpha||\beta||_2^2). \tag{2}$$

Then our model is $f(X) = A\beta_{MLE}$, where $\beta_{MLE} = (A^T A + \alpha I)^{-1} A^T Y$. In vector form, this means that $f(\vec{x_i}) = [1\ \vec{x_i}^T]\beta_{MLE} = \vec{\beta_0} + \sum_{j=1}^{d} x_{ij} \vec{\beta_j}$ [1]. Thus we can build our model once we have obtained β_{MLE}. We use the training data points and their labels as our data set when finding β_{MLE}, but once we have built our model we can give it any data points we wish. The value α in equation 2 is the regularization parameter, which is actually a hyper-parameter (see below). If we choose any value of $\alpha > 0$ then we can guarantee that the matrix $(A^T A + \alpha I)^{-1}$ is always invertible, giving numerical stability in calculating β_{MLE}. Other benefits of regularizing include making our model more robust to distributional shifts, and constraining our model by reducing the size of the vectors $\vec{\beta_j}$ so that no one feature of our data will dominate [1].

Ridge Classifier Method: The Ridge Classifier method utilizes Ridge Regression to classify data points. Let's assume that we have c classes, and we want to classify each data point as belonging to one of these classes. Firstly, an encoding (or label) is assigned to each class. These encodings will be vectors in $\mathbb{R}^c$. One possible way to do this is with "one-hot vector encodings", where the encodings will be the standard basis vectors in $\mathbb{R}^c$. Python's built-in Ridge Classifier method will build the encodings differently, as vectors of 1s and -1s, but the principle is the same [1]. In our data set we want to classify images as digits 0 through 9, so $c = 10$ and each digit will be assigned an encoding. These encodings become the "labels" $\vec{y_i} \in \mathbb{R}^c$ in the explanation of Ridge Regression above. Secondly, Ridge Regression is performed, yielding β_{MLE}, from which we build our model, f (see above). Then, to classify a point $\vec{x'} \in \mathbb{R}^d$, the model finds the closest class encoding to $f(\vec{x'})$ in $\mathbb{R}^c$ space, using the euclidean distance, and classifies the point $\vec{x'}$ as belonging to the class whose encoding was closest to $f(\vec{x'})$ [1]. This is repeated for all data points we wish to classify.

K-Nearest Neighbor (KNN) Classifier Method: First, we choose a value of k, which will be the number of nearest neighbors we will look at. This will be determined by cross-validation (see below). To classify a test point using the KNN Classifier, the model selects the k points of the training set (the data the

model is trained on) which are closest to the test point. Then the test point is assigned to the class which is most prevalent among the neighboring k points [1]. For example, if k is 3, and 2 of the neighboring 3 points are from class 1 and one is from class 2, then the test point would be assigned to class 1. This is repeated for all data points we wish to classify. However, the number of neighbors, k, does not "define" a model. This is because as the size of the data set goes to ∞, the value of k doesn't converge. This means that KNN is a non-parametric model, and k is a hyper-parameter, which we will determine via cross validation [1].

Cross Validation: The purpose of cross validation is to determine the value of a hyper-parameter by comparing models constructed with different possible hyper-parameter values, without using the test set. First we will explain the difference between hyper-parameters and parameters. A parameter is a numerical value used to define our model. We obtain the value of parameters when we train our model, and they appear in the model itself. Hyper-parameters are values that are used to train the model, but do not appear in the model itself [1]. In the Ridge Classifier, α is a hyper-parameter; in the KNN classifier, the number of neighbors k is a hyper-parameter.

To determine the value of a hyper-parameter by cross validation, we choose a set of possible values for the hyper-parameter, and the number of folds (call this n here, to avoid confusion with the hyper-parameter k). There are built in packages that will perform cross validation; see Section 3. They will follow this format: The training data will be (randomly) divided into n folds, or groups. In my models I have chosen to have $n = 5$ folds, which is a common choice in machine learning models. You specify a list of possible values for a hyper-parameter, and for each, the following will happen: Fold 1 will be designated the "test" fold, and the model will be trained on the other 4 folds. Then the model is evaluated on the "test" fold, and the classification accuracy is recorded. This is repeated 4 more times, each time holding back a different fold as the "test" fold. The accuracies of all 5 classifications are averaged, producing the average classification accuracy with this value of the hyper-parameter, which is called a "cross validation score"; referred to as such hereafter. This will be repeated with every hyper-parameter value in the list, and the value of the hyper-parameter which has the highest cross validation score will be chosen and used in the final model [1].

3. Algorithm Implementation and Development

Task I: We compile our training and testing data into the matrices `X_train` and `X_test`, respectively. Each row of these matrices is one 784-pixel image. This means the data has 784 dimensions. We perform PCA analysis on the train set (see Section 2) using `PCA` from `sklearn` to get the first 16 PC modes [4].

Task II: To determine k, the number of PC modes needed such that a reconstruction of the data from k PC modes will retain at least 85% of the energy in the original data, we perform PCA analysis on the training set and can get out the amount of energy stored in each PC mode. We do this by initializing `PCA()` without specifying k and transforming our training data into PCA space using `.fit_transform(X_train)`, and then using the `explained_variance_ratio_` command to get out a list of the amount of energy stored in each PC mode (non-cumulative). Then we can easily construct a loop to find how many PC modes we need to approximate 85% of the energy. Once we have k, we initialize `PCA(k)` and transform both `X_train` and `X_test` into k-modes PCA space, with `PCA` fitted to `X_train`. We will use this reduced-dimensional ($d = 59$) version of our data for classification.

Task III: We write a function that will select all samples of a particular subset of digits from `X_train` and `X_test` and their corresponding labels. We use Python's built in `enumerate` capability within the function.

Task IV: Using the function we built in task III, we can reduce our reduced-dimensional test and train data to contain only images of the digits 1 and 8. Then we apply the Ridge Classifier Method, with cross validation, as explained in section 2. We do this by instantiating a Ridge Classifier with the class `RidgeClassifier()` from `sklearn.linear_model` [4]. We then instantiate the class `GridSearchCV()` from `sklearn.model_selection` [4], giving it our ridge classifier, a list of possible α values, and the number of folds (I chose 5). Then we fit the grid search object we created to our subset of the training data set (& its labels), and it will perform cross validation for our ridge classifier, as explained in Section 2. From this we then extract the cross validation scores for each α value we fed into `GridSearchCV`, the α that yielded the highest cross validation score, and our final Ridge classifier model which will use the best α obtained by cross-validation (note that this is just the "best" out of all α values we fed into it). Then we can use this finalized model to compute the classification accuracies when the model is asked to classify our subsets of both the training data and the testing data. We do this using the `.score` command.

Task V: We repeat task IV, except we use our function from task III to select the digits 3 and 8. Repeat again for the digits 2 and 7.

Task VI: To perform multi-class classification with the Ridge Classifier, we repeat the procedure of task IV with the entire reduced-dimensional data set (all digits included). To perform multi-class classification with the KNN Classifier, we instantiate a KNN classifier with the class `KNeighborsClassifier()` from `sklearn.neighbors` [4]. We follow the same procedure as in task IV, but here we give `GridSearchCV()` [4] our KNN classifier, possible values of the hyper-parameter k (number of neighbors), and number of folds (5). When we fit this instance of `GridSearchCV` to our reduced-dimension training data (& labels), it performs cross validation for our KNN classifier, as explained in Section 2. Similarly to the Ridge Classifier case, we can extract the cross validation scores for each k we fed into `GridSearchCV`, as well as the k value that yielded the highest cross validation score, and the final KNN classifier model which we use to compute the classification accuracy on both the (low-dimensional) training and testing data using the `.score` command.
Task VII: We now perform multi-class classification using a Naive-Bayes Gaussian (NB) classifier. Since it is Gaussian, our model will assume that features will follow a Gaussian distribution. We instantiate the classifier with the class `GaussianNB()` from `sklearn.naive_bayes` [4]. As before, we instantiate our grid search object with `GridSearchCV()` [4], but this time we give it our NB classifier, number of folds (5), and possible values of the hyper-parameter `var_smoothing`. This hyper-parameter adds a small value to the variance of each feature to improve stability in the model. Therefore we expect `var_smoothing` to have a small value (close to 0). When we fit our instance of `GridSearchCV` (i.e. our grid search object) to our low-dimensional training data (& labels), it performs cross validation for our NB classifier. Again, we can get the cross validation scores for each value of `var_smoothing` we fed into `GridSearchCV`, including the value of `var_smoothing` with the highest cross validation score, and our final NB model, and we compute the classification accuracy of the final NB model in classifying both the (low-dimensional) training and the testing data with `.score`. Note that in each classification we perform, we are creating an entirely new model; they are separate classification problems.

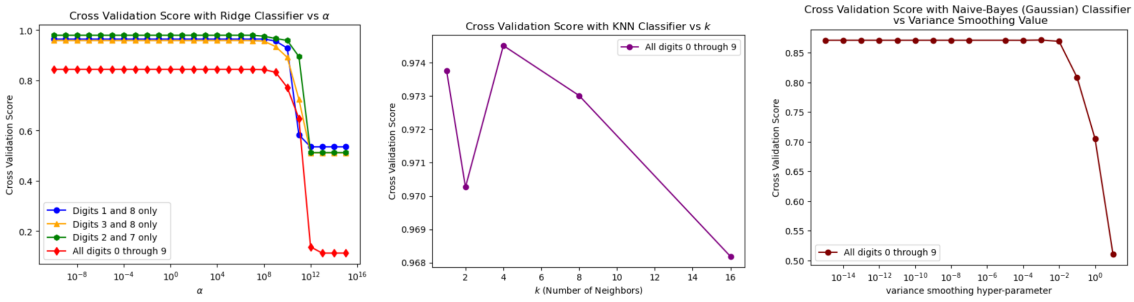
4. Computational Results

Task I: After performing PCA analysis on the images in the training set, we plotted the first 16 PC modes as 28 x 28 images (see Figure 1a) [3]. We can see that these images vaguely resemble numbers, telling us that performing PCA analysis is reasonable.
Task II: After inspecting the cumulative energy of the singular values, we determine that we need $k = 59$ PC modes to approximate 85% of the energy. Figure 1b shows the first 64 training images, and Figure 1c shows the first 64 training images reconstructed from $k = 59$ PC modes. We are easily able to tell what digit is in each of the reconstructed images, so we conclude that truncating our data to 59 dimensions is reasonable to use for classification.
Tasks IV, V, VI, VII: Figure 2a shows the cross validation scores vs α, when we performed cross validation for various Ridge Classifiers. We performed 4 ridge classification problems with cross validation, for various digit pairs and for all digits (see legend). We can see that in all cases, alpha had little effect on the cross validation score up to a point, when the cross validation score drops sharply (when α was about 10^{10} to 10^{12}). This is because, while regularization does not have a huge affect on the efficacy of the model, at some point regularization will be so high that the term $\alpha||\beta||_2^2$ in equation 2 will be so large that the data itself will have little effect on the model, which would in effect make the model blind to variation between data points, making it unable to classify points accurately. Figure 2b show the cross validation scores vs the number of neighbors k, when we perform cross validation for our multi-class KNN classifier. We see that $k = 4$ is the best out of all of the options we gave `GridSearchCV`. Finally, Figure 2c shows cross validation scores vs the value of `var_smoothing`. We see that as `var_smoothing` nears 1 the cross validation score drops sharply, presumably because we are adding too high a factor to the variance of each feature.

In Figure 3a, we see the classification accuracies when we classified our testing and training sub-data sets using our final Ridge Classifier models after hyper-parameter tuning via cross validation, for various digit pairs (Note: the classification & cross validation process was repeated for each of these sub-data sets, yielding 3 separate models). We see that ridge classification was more effective in distinguishing digits 1 from 8 and digits 2 from 7, and less effective in distinguishing digits 3 from 8. This makes sense logically, as the digits 3 and 8 are more similar-looking than the digits in the other pairs. However, in all cases, accuracy of both the training and the testing set was above 95%. This means our models performed quite well. Figure 3b shows the testing and training accuracy in classifying between all 10 digits with three different classifier methods: Ridge, KNN, and Naive-Bayes. We can see that the KNN classifier was the most effective in classifying both the training and testing sets, and outperformed the other two methods significantly.

(A) (B) (C)

FIGURE 1. (A): First 16 PC Modes as 28 x 28 images, (B): first 64 training images, and (C): first 64 training images reconstructed from k=59 truncated PC Modes [3].



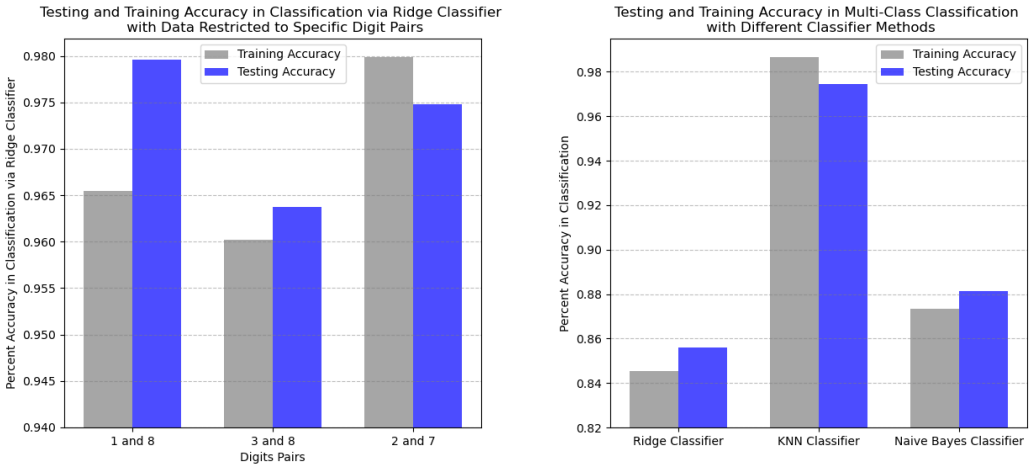
(A) (B) (C)

FIGURE 2. (A) Shows Cross Validation (CV) Scores vs α when performing CV for various Ridge Classifier models, with various subsets of the data (see legend). (B) Shows the CV Scores vs k (number of neighbors) when CV was performed for our multi-class K-Nearest Neighbors (KNN) Classifier, and (C) Shows the CV Scores vs variance smoothing hyper-parameter `var_smoothing` when CV was performed for our multi-class NB Classifier [3].

However, Naive-Bayes was still a bit better than the Ridge classifier. Notice that in all of the classifications we performed, with various data sets and classifier types, none of the training accuracies outperformed the testing accuracies significantly, so we conclude that no over-fitting occurred in any of these models. If we saw that for some classifier model, the training accuracy was much higher than the testing accuracy, this would indicate that our model has been fitted to some of the noise in the training set (over-fitting), making it worse at classification of new samples. We do not see this here.

Figure 4 consolidates the information given in Figure 3, and gives the hyper-parameter values we used in our final models for each classification problem we solved. We chose these values because they yielded the highest cross validation score when we performed cross validation. Notice that the testing and training accuracy for classifying all 10 digits with ridge classification were significantly lower (by $\sim 10\%$) than then accuracies when we classified between only two digits, for any of the digit pairs we looked at. This makes sense because clearly classifying between 10 classes is a harder problem than just classifying between two classes, but we didn't increase the complexity of our model at all. However, our KNN classifier still had really good accuracy when we classified between all digits, suggesting that this model would be preferable for multi-class classification over the Ridge Classifier model.

Figure 5 shows the confusion matrices for each classifier method in classifying between all 10 digits. Confusion matrices are generated by using a model to predict the labels of the test set and comparing them to the true labels in a matrix format. Then we can see common misclassifications that a model made when classifying the test set. Figure 5a gives the confusion matrix for our final multi-class ridge classifier that we



(A) (B)

FIGURE 3. (A) Displays the testing and training accuracies in classification via Ridge Classification when the data set was reduced to various digit pairs. (B) Shows the testing and training accuracies in multi-class classification with 3 different classifier methods [2], [3].

Training and Testing Accuracy

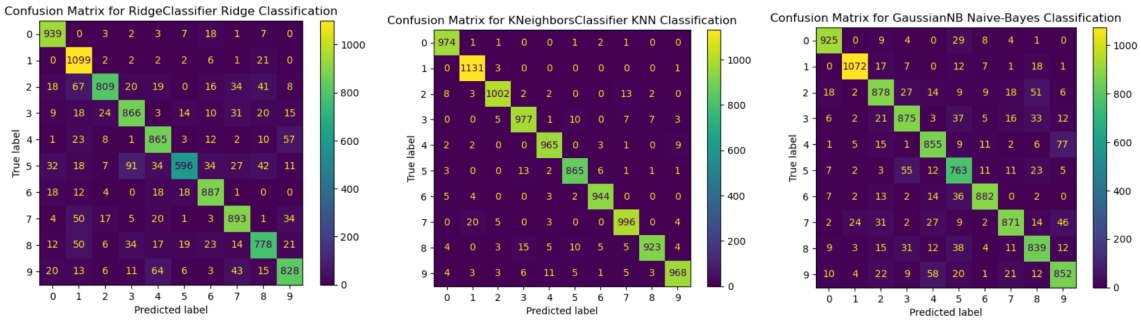
	Training Accuracy	Testing Accuracy	Hyperparameter Value
Ridge Classifier, digits 1 & 8 only	0.9654569999205908	0.979611190137506	alpha: 100000000.0
Ridge Classifier, digits 3 & 8 only	0.9601902854281422	0.9637096774193549	alpha: 100000.0
Ridge Classifier, digits 2 & 7 only	0.9798740080176715	0.974757281553398	alpha: 100000.0
Ridge Classifier, all digits	0.8451833333333333	0.856	alpha: 1e-10
KNN Classifier, all digits	0.9866166666666667	0.9745	Number of Neighbors: 4
Naive-Bayes Classifier, all digits	0.8733	0.8812	Variance Smoothing Value: 0.001

FIGURE 4. Information from each classifier model after cross-validation and testing [3].

obtained after cross validation. We see that 5 was the most misclassified digit; 91 images with the digit 5 were misclassified as the digit 3. This makes sense, as 5 and 3 are physically more similar than some other pairs of digits. Figure 5b gives the confusion matrix for our KNN classifier. We see that 5 was still the most misclassified digit, but was still classified correctly much more often than with the ridge classifier. The KNN classifier rarely misclassified digits. Figure 5c gives the confusion matrix for our Naive-Bayes classifier. We see that 5 was the most misclassified digit with this classifier as well. For all classifiers, the digit 1 was classified correctly the most often, which makes sense as 1 bears little resemblance to any of the other digits.

5. Summary and Conclusions

In this report, we used PCA analysis to reduce the dimensionality of our data from 784 to 59 dimensions. We then created a variety of classifier models to classify our data (or subsets of our data) by digit. We used cross validation to tune the hyper-parameters for each model we created, examined training and testing accuracies, and compared the efficacy of each model. We drew interesting conclusions about the common mistakes made by each model by looking at confusion matrices, and about hyper-parameters from looking at cross validation scores. In future, it would be interesting to try the SVM classifier and see how it performs.



(A) (B) (C)

FIGURE 5. Confusion matrices for multi-class classification with (A) the Ridge Classifier, (B) the KNN Classifier, and (C) the Naive-Bayes (Gaussian) Classifier [3].

ACKNOWLEDGMENTS

The author is thankful to Dr. Frank for her lectures on the principles of linear and ridge regression, machine learning, various classifier methods, and cross validation. We also thank Dr. Frank for providing us with helper code for loading and plotting the data used in this report, and for creating the confusion matrices. We are also thankful to Rohin Gilman for explaining both the concepts of cross validation and how it works computationally, and in answering a variety of other questions on topics related to this assignment.

REFERENCES

[1] N. Frank. Amath 482 lecture notes. *Department of Applied Mathematics, University of Washington*, Feb 2025.
[2] C. R. Harris, K. J. Millman, S. J. van der Walt, et al. Array programming with NumPy. *Nature*, 585:357–362, 2020.
[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
[4] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.