Preprocessing:

The initial step in working with data involves gaining a comprehensive understanding of it. In our case, we have a dataset comprising 10,000 black and white images, with each image consisting of 28 by 28 pixels, resulting in an array of 784 values for each corresponding row.

```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      2       0       0       0       0       0       0       0       0
1      9       0       0       0       0       0       0       0       0
2      6       0       0       0       0       0       0       0       5
3      0       0       0       0       1       2       0       0       0
4      3       0       0       0       0       0       0       0       0

   pixel9  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0       0  ...         0         0         0         0         0         0
1       0  ...         0         0         0         0         0         0
2       0  ...         0         0         0        30        43         0
3       0  ...         3         0         0         0         0         1
4       0  ...         0         0         0         0         0         0

   pixel781  pixel782  pixel783  pixel784
0         0         0         0         0
1         0         0         0         0
2         0         0         0         0
3         0         0         0         0
4         0         0         0         0

[5 rows x 785 columns]
```

Given the nature of our dataset, it is likely that many of the pixel values are 0 or close to 0. Consequently, due to the high dimensionality and the prevalence of small values, employing Principal Component Analysis (PCA) holds significant potential for our analysis.

Subsequently, we conducted a thorough examination of our data to identify any null values, which fortunately yielded none. However, upon closer inspection, we discovered the presence of duplicate values within the dataset. To ensure data integrity and avoid any bias in our analysis, we took the necessary steps to eliminate these duplicate entries.

In [3]:
```
# Check if any rows have null values
null_rows = trainData.isnull().any(axis=1)
print('Number of rows with null values:', sum(null_rows))
#  There were no null values

# Check if there is any duplicate rows
duplicate_rows = trainData[trainData.duplicated()]
print("Number of duplicate rows: ", len(duplicate_rows))
```

```
Number of rows with null values: 0
Number of duplicate rows:  43
```

In [4]:
```
# getting rid of duplicate rows
trainData = trainData.drop_duplicates()

# Check if problem is solved
duplicate_rows = trainData[trainData.duplicated()]
print("Number of duplicate rows: ", len(duplicate_rows))
```

```
Number of duplicate rows:  0
```

In our preprocessing phase, we deemed it essential to assess the balance of our dataset. Upon careful evaluation, we observed that the distribution of classes within the data was relatively balanced. This aspect is crucial as it helps to mitigate any potential biases that may arise during model training and evaluation.

In [5]:
```
#check for imbalancement
class_counts = trainData['label'].value_counts()

# Printing the number of members in each class
print(class_counts)

#our data is balanced
```

```
5    6000
8    6000
9    5998
0    5998
3    5997
7    5996
1    5996
4    5995
6    5989
2    5988
Name: label, dtype: int64
```

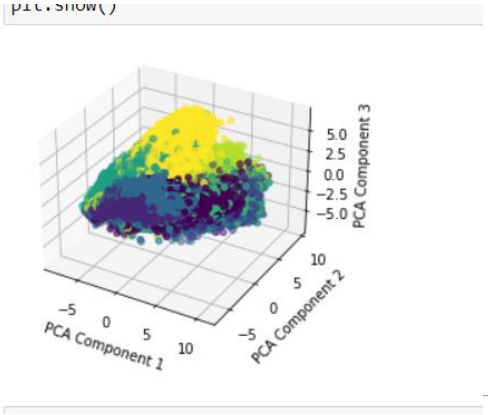
Additionally, it is worth mentioning that each column in our dataset corresponds to a pixel, with pixel values ranging from 0 to 255. Recognizing the significance of data optimization, we deemed it necessary to normalize the data. By employing normalization techniques, specifically the x-min/max-min technique, we were able to scale the pixel values to a standardized range. This normalization process plays a vital role in enhancing the accuracy and effectiveness of our analysis.

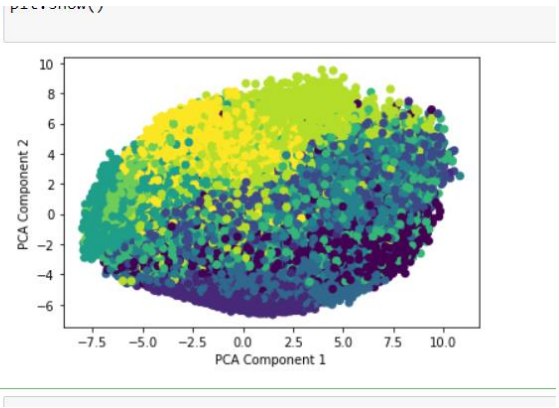
By considering these steps in our data preprocessing, we can ensure that our data is appropriately understood, prepared, and ready for subsequent analysis and model training.

PCA

Now, let's discuss the implementation of the PCA (Principal Component Analysis) technique. As a preliminary step, we begin by normalizing our data using its mean. Subsequently, we calculate the covariance matrix utilizing libraries such as NumPy. By finding the eigenvectors and their corresponding eigenvalues, we are able to sort the eigenvectors in descending order based on their eigenvalues. The resulting matrix, obtained by multiplying our data with the first k rows of the sorted eigenvectors, represents our PCA projection with k components. This approach allows us to effectively reduce the dimensionality of our data while retaining the most significant information captured by the principal components.

It is important to note that when visualizing data, our human perception is limited to comprehending information in two or three dimensions. Therefore, for visual analysis purposes, we typically rely on 2D or 3D visualizations. By leveraging such visualizations, where each color represents a specific class, we can observe a notable pattern: instances of the same class tend to cluster together. This clustering phenomenon is a positive indication, as it suggests that the dimensionality reduction achieved retains the essential information necessary for accurate classification. Despite reducing the dimensionality of the data, the inherent structure and discriminative properties among classes are effectively preserved, enabling us to potentially achieve high accuracy in classification tasks.





What are PCA components and who contributes the most?

Upon conducting PCA on the training data, a set of principal components is obtained. These components, which are linear combinations of the original features, are ordered in terms of the variance they explain in the data. The first principal component captures the highest amount of variance, followed by the second component, and so forth. Importantly, the principal components are orthogonal to one another, indicating that they are uncorrelated.

To understand the contribution of each original feature to a principal component, one can examine the corresponding eigenvector, also known as the loadings. The eigenvector signifies the direction in the original feature space that aligns with the principal component. The magnitude of the values within the eigenvector signifies the strength of each feature's contribution to the principal component. Features with higher absolute values make a more significant contribution to that specific principal component.

To identify the data features with the most substantial contributions, we determine the eigenvector with the greatest magnitude for each PCA component. We then employ a methodology to select the top five contributors and tally their occurrences. This process enables us to identify the features that consistently appear as significant contributors across the principal components. Subsequently, we present these features as the final result, providing valuable insights into the data's key contributing factors.

```
Top 5 Features
Feature: pixel16
Feature: pixel295
Feature: pixel42
Feature: pixel13
Feature: pixel270
```

Was sklearn's PCA same as ours?

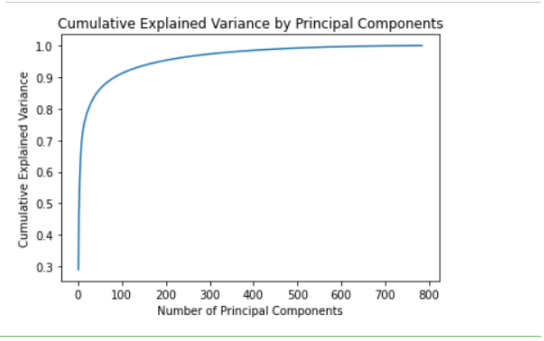
In our analysis, we implemented our custom PCA algorithm and compared it with scikit-learn's PCA implementation. We observed a sign difference in the extracted principal components between the two approaches, which is a common occurrence due to the arbitrary sign convention in eigenvector extraction. However, despite the sign difference, the core results obtained from both implementations were consistent. The explained variance, feature contributions, and captured patterns exhibited similarity, demonstrating that our custom PCA aligns with scikit-learn's PCA. We verified this through numerical comparison in our code, using a tolerance of 1e-4. Based on the observed consistency and similarity, we can confidently use either implementation for model training and analysis tasks.

```
              0         1         2
0      3.686147  4.960575  0.049794
1     -4.376604  3.859403  1.949315
2      7.265298  1.569918 -2.315191
3      3.228979 -2.325054  0.649587
4      4.148112 -4.039208  1.513174
...         ...       ...       ...
59995  0.017142  5.187802  5.057856
59996 -3.113472 -5.212777  0.104461
59997  2.181811  0.451604  0.558415
59998  1.469619  6.063090  0.625137
59999 -3.571684  4.769195  0.263435

[59957 rows x 3 columns]
[[ 3.68614723  4.96057532 -0.04979422]
 [-4.37660407  3.8594032  -1.94931515]
 [ 7.26529844  1.56991795  2.31519066]
 ...
 [ 2.18181149  0.45160356 -0.558415  ]
 [ 1.46961945  6.06308951 -0.62513733]
 [-3.57168421  4.76919486 -0.26343514]]
Are the results equal? True
```

Explained variance

In addition, we conducted an analysis of the explained variance to gain insights into the amount of information captured by each principal component. This analysis helps in determining the significance of each component and identifying the optimal number of components for future modeling tasks. By plotting the explained variance, we can visualize the cumulative contribution of the principal components and identify the point at which most of the information is preserved. This information is valuable for selecting the most effective and efficient components for subsequent modeling and analysis.



Modelling

SVM

Due to computational constraints, training the SVM model on the entire dataset of 10,000 images was not feasible. To overcome this limitation, a sampling technique called stratified sampling was employed. This technique ensured that each sample contained an equal number of instances from each class, allowing for a representative subset of the data.

In the SVM model, various hyperparameters were explored to achieve the best performance. The primary hyperparameter investigated was the choice of kernels, including 'linear', 'poly', and 'rbf'. Additionally, different numbers of PCA components were considered to optimize the model's performance. The best result is as mentioned below.

```
Best Hyperparameters: {'kernel': 'rbf'}
Best Sample_size: 9000
Best Score: 0.8647777777777778
Best dimension for this model 500
```

Bayes

The Naive Bayes model does not have hyperparameters to tune. Therefore, in our analysis, the focus was primarily on selecting the optimal number of PCA components to enhance the model's performance.

```
Best Score 0.7674666509666551 using 100 PCA components
```

KNN

In K-Nearest Neighbors (KNN), we experimented with different numbers of PCA components and explored hyperparameters such as weights and the number of neighbors (n-neighbors). We specifically considered odd values for the number of neighbors to facilitate majority voting. By systematically evaluating these factors, we aimed to optimize the performance of the KNN model for our dataset.

```
100
200
Best Score 0.8633520545679441 with 5 neighbors using 160 PCA components and weights='distance'
```

MLP

Given the computationally intensive nature of MLP models and hardware limitations, we selected a subset of hyperparameters to optimize. Although there are numerous hyperparameters to consider, we focused on key factors such as hidden layer size, learning rate and activation function. By carefully tuning these hyperparameters, we achieved the best results for our model. It is important to note that for faster computations, we utilized PCA with 10 components.

```
best Hidden layer size:  (100, 25)
best Learning rate:  adaptive
best Activation:  relu
best score:  0.8282833333333333
```

Different activation functions

1. Rectified Linear Unit (ReLU): The ReLU activation function effectively sets negative input values to zero while preserving positive values. It offers computational efficiency and addresses the vanishing gradient problem commonly encountered in deep neural networks. With its ability to facilitate faster convergence during training, ReLU has gained significant popularity and is extensively used in diverse applications.

2. Logistic (Sigmoid): The logistic or sigmoid activation function transforms input values into a range of 0 to 1, typically used in binary classification tasks to estimate probabilities. Despite its usefulness in such scenarios, the sigmoid function can encounter vanishing gradients, limiting its effectiveness in deep neural networks where gradients can diminish exponentially as they propagate through multiple layers.

3. Hyperbolic Tangent (Tanh): The tanh activation function operates by mapping input values to a range spanning from -1 to 1. It shares similarities with the sigmoid function but is centered around zero, providing a more balanced representation. Tanh is commonly utilized in binary and multi-class classification tasks, as it effectively captures both positive and negative input values. When compared to the sigmoid function, tanh demonstrates an enhanced ability to model and capture negative values.

4. Identity: The identity activation function simply returns the input value as the output. It is commonly used in regression tasks where the model's output directly represents the predicted values.

Final results

SVM

```
Confusion Matrix (Training Data):
[[5422    2   55  126    9    2  354    0   27    1]
 [  10 5855    9  105    6    0   10    0    1    0]
 [  52    1 5218   49  400    1  260    0    7    0]
 [  95    8   34 5621  138    0   90    0   11    0]
 [   6    4  311  149 5280    0  238    0    7    0]
 [   0    0    0    1    0 5847    0  119   10   23]
 [ 655    4  402  114  290    0 4498    0   25    1]
 [   0    0    0    0    0   62    0 5821    5  108]
 [   8    1   12   24   17    4   24   10 5900    0]
 [   0    0    0    1    0   28    0  159    1 5809]]
Confusion Matrix (Test Data):
[[864   0  11  25   0   2  88   0  10   0]
 [  2 975   2  15   0   1   5   0   0   0]
 [  9   0 832  14  80   0  63   0   2   0]
 [ 25   7   7 921  21   0  18   0   1   0]
 [  0   0  55  26 872   0  45   0   2   0]
 [  0   0   0   0   0 943   0  39   4  14]
 [150   0  75  25  55   0 687   0   8   0]
 [  0   0   0   0   0  17   0 950   0  33]
 [  3   0   6   2   1   2   7   2 977   0]
 [  0   0   0   0   0   6   0  32   0 962]]
```

During the evaluation of the MLP model, it was observed that overfitting was not a major concern. In fact, the model exhibited impressive performance even surpassing the training set results. However, it is important to note that the misclassifications observed in the training set were consistently present in the test set as well. This suggests that the model may have encountered underfitting issues, failing to capture certain patterns and complexities present in the data. Further analysis is required to better understand and address these mispredictions in future iterations.

Naïve Bayes

```
Confusion Matrix (Training Data):
[[4550    1  103  381   24   33  421    0  483    2]
 [  33 5448   84  235    9   18   59    0  110    0]
 [ 104    0 3783   35  769   42  868    0  387    0]
 [ 371   53   58 4813  175   33  295    2  197    0]
 [  39    2  616  296 3962   14  837    1  228    0]
 [  16    0    0    3    0 4603  163  923  203   89]
 [ 977    1  631  202  436   47 3115    0  580    0]
 [   2    0    0    0    0  341   14 5265   16  358]
 [  68    0   33   49   59  124  215  159 5287    6]
 [   4    0    0    0    0  106   23  443   83 5339]]
Confusion Matrix (Test Data):
[[725   0  23  77   5   2  71   2  95   0]
 [  2 913  15  37   2   1  13   0  17   0]
 [ 25   0 645   5 126   5 122   0  72   0]
 [ 56   8   9 815  30   3  40   0  39   0]
 [  6   0  85  38 704   1 135   0  31   0]
 [  4   0   1   0   0 772  12 160  30  21]
 [190   0  90  44  63   6 519   2  86   0]
 [  0   0   0   0   0  49   1 868   5  77]
 [ 18   0   4   5  10  18  31  28 884   2]
 [  2   0   0   0   0  21   6  84  12 875]]
Accuracy: 0.772
Precision: 0.776069815798567
Recall: 0.772
F1 Score: 0.7715691936240091
```

The evaluation of the Naive Bayes classifier indicated that overfitting was not a major concern. However, certain classes displayed a higher rate of mispredictions compared to others. This discrepancy suggests that additional examples of those classes may be required to enhance the classifier's performance. Alternatively, it is possible that Naive Bayes may not be the most suitable classifier for this particular dataset.

KNN

```
Confusion Matrix (Training Data):
[[5998    0    0    0    0    0    0    0    0    0]
 [   0 5996    0    0    0    0    0    0    0    0]
 [   0    0 5988    0    0    0    0    0    0    0]
 [   0    0    0 5997    0    0    0    0    0    0]
 [   0    0    0    0 5995    0    0    0    0    0]
 [   0    0    0    0    0 6000    0    0    0    0]
 [   0    0    0    0    0    0 5989    0    0    0]
 [   0    0    0    0    0    0    0 5996    0    0]
 [   0    0    0    0    0    0    0    0 6000    0]
 [   0    0    0    0    0    0    0    0    0 5998]]
Confusion Matrix (Test Data):
[[854   1  13  15   5   0 104   2   6   0]
 [  3 971   3  15   1   0   7   0   0   0]
 [ 16   0 782  12 108   0  78   0   4   0]
 [ 23   7  12 898  36   0  23   0   1   0]
 [  2   0  72  23 824   0  77   0   2   0]
 [  0   0   0   1   0 868   4  67   5  55]
 [195   1  89  19  76   0 611   0   9   0]
 [  0   0   0   0   0   5   0 944   0  51]
 [  2   0   9   1   5   1   5   3 972   2]
 [  0   0   0   0   0   4   0  34   0 962]]
Accuracy: 0.8686
Precision: 0.8694192173710786
Recall: 0.8686
F1 Score: 0.8680613759833808
```

The performance of the K-Nearest Neighbors (KNN) classifier was flawless in the training set as we could have guested. However, in the test set, we observed some mispredictions. Despite these mispredictions, the overall accuracy of the classifier remained high, suggesting that KNN is a suitable algorithm for our data. This observation aligns with the visualizations obtained through PCA, where we observed clusters of similar classes. The ability of KNN to leverage the proximity of data points makes it effective in capturing these clusters and achieving good classification accuracy. It is important to note that the time-consuming nature of KNN, particularly with a large number of dimensions, should be taken into consideration when working with this classifier.

MLP

```
Confusion Matrix (Training Data):
[[5014    6   67  196   31    1  652    0   31    0]
 [  23 5769    9  142   19    0   31    0    3    0]
 [  51    0 4406   64  659    3  783    0   22    0]
 [ 219   57   27 5288  223    1  160    0   21    1]
 [  12    8  662  252 4406    0  632    0   23    0]
 [   1    0    0    1    0 5577    0  293   25  103]
 [ 698   12  582  161  461    2 4032    0   41    0]
 [   0    0    0    0    0  137    0 5642    6  211]
 [  26    1   20   33   20   19   67   16 5794    4]
 [   0    0    0    1    0  117    0  281    2 5597]]
Confusion Matrix (Test Data):
[[804   5   8  27   6   4 138   0   8   0]
 [  3 964   2  21   2   0   6   1   1   0]
 [ 12   0 681  11 127   0 159   0  10   0]
 [ 40  18   9 869  39   1  20   0   4   0]
 [  2   1 112  38 754   0  90   0   3   0]
 [  0   0   0   2   0 883   0  71  10  34]
 [156   4  90  24  81   0 636   0   9   0]
 [  0   0   0   0   0  37   0 908   2  53]
 [  3   1  11   4   4   3  17   5 950   2]
 [  0   0   0   0   0  27   0  57   1 915]]
Accuracy: 0.8364
Precision: 0.8377627035284592
Recall: 0.8364
F1 Score: 0.8367724396797558
```

The performance of the MLP model is discussed in the subsequent question.

How does the performance of the MLP model compare to the other models used in the assignment?

The performance of the MLP model compared to the other models used in the assignment is subject to certain considerations. MLPs, like other neural network models, typically require a large dataset and considerable computational resources to achieve optimal performance. However, due to hardware limitations and time constraints, we were only able to explore a limited set of hyperparameters and utilize 10 PCA components in our experiments.

Despite these limitations, the MLP model demonstrated promising results with an accuracy of 83%. It is important to note that with a larger dataset, more extensive hyperparameter tuning, and the inclusion of additional PCA components, we can expect even higher accuracy levels. Thus, it is reasonable to anticipate that the MLP model has the potential to outperform the other models used in the assignment if these constraints were not a factor.