In [2]:

```
# import base packages into the namespace for this program
import numpy as np
import pandas as pd
from scipy import stats
from datetime import datetime

#diplay and plotting
import seaborn as sns
from IPython.display import display
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import cm
#import scikitplot as skplt
#import from SKlearn

from sklearn.ensemble import  RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import GridSearchCV

from sklearn.decomposition import PCA
import sklearn.metrics as metrics
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import homogeneity_score
from sklearn.metrics import completeness_score
from sklearn.metrics import f1_score

from sklearn.pipeline import Pipeline

import warnings
warnings.filterwarnings("ignore") # To set the warning option to False


# read data for the Boston Housing Study
Digit_input = pd.read_csv("digitrecognizer_train.csv")
Digit_test = pd.read_csv("digitrecognizer_test.csv")

```

Imported required libraries.

Digit_input contains train dataset

Digit_test contains test dataset

In [3]:

```
# Sets random seed for entire notebook
RANDOM_SEED = 1

# kfold for CV of models with shuffle=True
kfold = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

# testkfold for CV of evaluation with shuffle=True
testkfold = KFold(n_splits=2, shuffle=True, random_state=RANDOM_SEED)

```

In [4]:

```
#get counts of the labels, check for balance
Digit_input.label.value_counts().sort_values()
```

Out[4]:

```
5    3795
8    4063
4    4072
0    4132
6    4137
2    4177
9    4188
3    4351
7    4401
1    4684
Name: label, dtype: int64
```

In [5]:

```
#splits into validation and train.
train, val =  train_test_split(Digit_input, test_size = 0.2, random_state = RANDOM_SEED)


y_train = train.label.copy()
X_train = train.drop('label', axis=1)
```

In [6]:

```
y_val = val.label.copy()
X_val = val.drop('label', axis=1)
```

In [7]:

```
print(train.label.value_counts())
print(val.label.value_counts())
```

```
1    3744
7    3551
3    3478
9    3345
2    3342
6    3337
0    3279
4    3243
8    3217
5    3064
Name: label, dtype: int64
1    940
3    873
0    853
7    850
8    846
9    843
2    835
4    829
6    800
5    731
Name: label, dtype: int64
```

train_test_split: This function is used to split the dataset into training and validation sets. It takes several arguments:

Digit_input: The input data, likely a pandas DataFrame or numpy array, containing features (attributes) for each sample. test_size: This parameter determines the proportion of the dataset that will be allocated to the validation set. In this case, it is set to 0.2, which means 20% of the data will be used for validation, and the remaining 80% will be used for training. random_state: This parameter is used to ensure reproducibility. By setting it to a specific value (e.g., RANDOM_SEED), you guarantee that every time you run this code with the same seed value, the data splitting will be the same, which is useful for debugging and result comparison. train, val: These are two separate datasets resulting from the data split.

train: This dataset contains 80% of the original data and will be used for training the machine learning model. val: This dataset contains 20% of the original data and will be used for validating the model's performance. Extracting labels and features:

y_train: This variable holds the labels corresponding to the training set. It appears that there is a column named 'label' in the original Digit_input, and y_train is a copy of that column, representing the labels of the training data. X_train: This variable contains the features (attributes) of the training set. It is created by dropping the 'label' column from the original Digit_input, leaving only the input features in the X_train dataset.

In [8]:

```
print(X_train.columns)
print(y_train.name)
```

```
Index(['pixel0', 'pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6',
       'pixel7', 'pixel8', 'pixel9',
       ...
       'pixel774', 'pixel775', 'pixel776', 'pixel777', 'pixel778', 'pixel779',
       'pixel780', 'pixel781', 'pixel782', 'pixel783'],
      dtype='object', length=784)
label
```

Begin by fitting a random forest classifier using the full set of 784 explanatory variables

and the model training set (train.csv).

In [11]:

```
Forest_pipe = Pipeline([
        ('regressor', RandomForestClassifier(n_estimators = 50, max_depth=10, max_features = 's
        ])
now = datetime.now()
Forest_pipe.fit(X_train, y_train)
after_fit = datetime.now()

forest_Train_pred =Forest_pipe.predict(X_train)
```

Create a RandomForestClassifier with specific hyperparameters:

n_estimators: The number of decision trees in the random forest ensemble, set to 50. max_depth: The maximum depth of the decision trees, set to 10. max_features: The maximum number of features to consider when splitting a node. In this case, it is set to 'sqrt,' which means it will use the square root of the total number of features. max_leaf_nodes: The maximum number of leaf nodes in each tree, set to 400. random_state: A seed value for reproducibility, set to RANDOM_SEED. Construct a Pipeline object:

The pipeline is created using Pipeline(), and it has a single step called 'regressor', which corresponds to the RandomForestClassifier defined earlier. Fit the pipeline to the training data:

Forest_pipe.fit(X_train, y_train) fits the pipeline to the training data (X_train) and corresponding labels (y_train). This step trains the RandomForestClassifier on the provided data. Make predictions on the training data:

forest_Train_pred = Forest_pipe.predict(X_train) uses the trained pipeline to make predictions on the training data (X_train). This will produce predictions based on the model that has been trained.

In [12]:

```
elapsed = after_fit-now
Forest_elapsed = elapsed.total_seconds()
```

The elapsed time of the training process for the RandomForestClassifier is calculated. The time is measured in seconds and stored in the variable Forest_elapsed.

In [13]:

```
print(Forest_elapsed, 'seconds')
```

```
11.478403 seconds
```

The training process for the RandomForestClassifier took approximately 11.48 seconds.

This duration can be useful for understanding the computational cost of training the model and can be used for comparison with other models or hyperparameter settings.

In [16]:

```
#!pip install scikit-plot
```

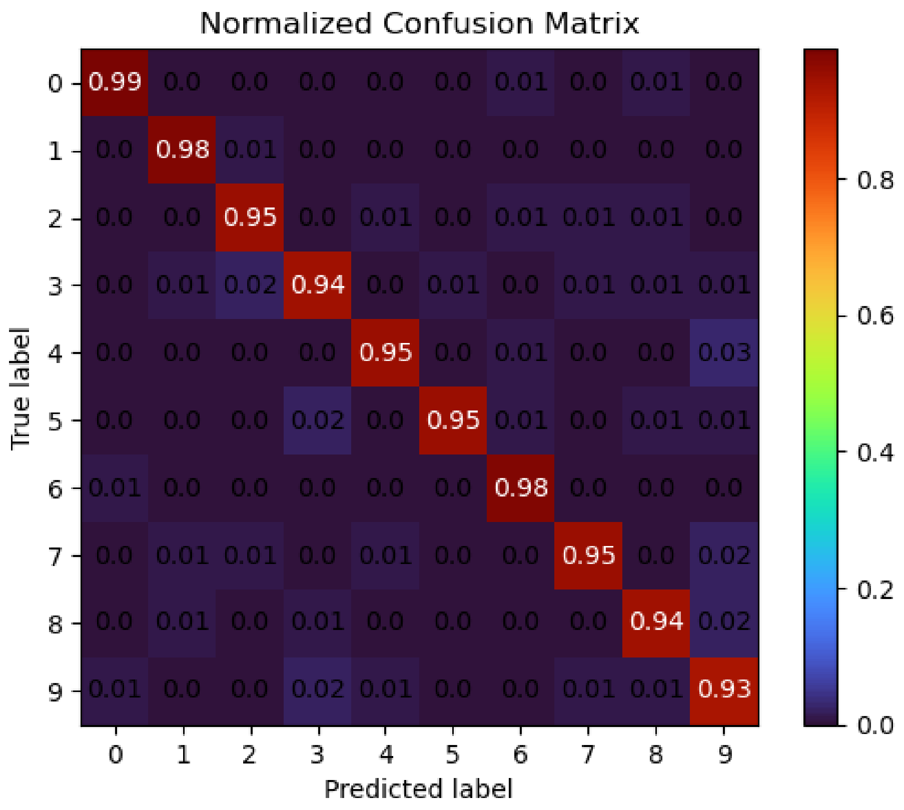
```
Collecting scikit-plot
  Downloading scikit_plot-0.3.7-py3-none-any.whl (33 kB)
Requirement already satisfied: scikit-learn>=0.18 in d:\users\ecorc\anaconda3\lib\sit
e-packages (from scikit-plot) (1.0.2)
Requirement already satisfied: joblib>=0.10 in d:\users\ecorc\anaconda3\lib\site-pack
ages (from scikit-plot) (1.1.1)
Requirement already satisfied: scipy>=0.9 in d:\users\ecorc\anaconda3\lib\site-packag
es (from scikit-plot) (1.10.1)
Requirement already satisfied: matplotlib>=1.4.0 in d:\users\ecorc\anaconda3\lib\site
-packages (from scikit-plot) (3.6.2)
Requirement already satisfied: kiwisolver>=1.0.1 in d:\users\ecorc\anaconda3\lib\site
-packages (from matplotlib>=1.4.0->scikit-plot) (1.4.4)
Requirement already satisfied: python-dateutil>=2.7 in d:\users\ecorc\anaconda3\lib\s
ite-packages (from matplotlib>=1.4.0->scikit-plot) (2.8.2)
Requirement already satisfied: pillow>=6.2.0 in d:\users\ecorc\anaconda3\lib\site-pac
kages (from matplotlib>=1.4.0->scikit-plot) (9.3.0)
Requirement already satisfied: packaging>=20.0 in d:\users\ecorc\anaconda3\lib\site-p
ackages (from matplotlib>=1.4.0->scikit-plot) (22.0)
Requirement already satisfied: pyparsing>=2.2.1 in d:\users\ecorc\anaconda3\lib\site-
packages (from matplotlib>=1.4.0->scikit-plot) (3.0.9)
Requirement already satisfied: contourpy>=1.0.1 in d:\users\ecorc\anaconda3\lib\site-
packages (from matplotlib>=1.4.0->scikit-plot) (1.0.5)
Requirement already satisfied: cycler>=0.10 in d:\users\ecorc\anaconda3\lib\site-pack
ages (from matplotlib>=1.4.0->scikit-plot) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in d:\users\ecorc\anaconda3\lib\site
-packages (from matplotlib>=1.4.0->scikit-plot) (4.25.0)
Requirement already satisfied: numpy>=1.19 in d:\users\ecorc\anaconda3\lib\site-packa
ges (from matplotlib>=1.4.0->scikit-plot) (1.21.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in d:\users\ecorc\anaconda3\lib\s
ite-packages (from scikit-learn>=0.18->scikit-plot) (2.2.0)
Requirement already satisfied: six>=1.5 in d:\users\ecorc\anaconda3\lib\site-packages
(from python-dateutil>=2.7->matplotlib>=1.4.0->scikit-plot) (1.16.0)
Installing collected packages: scikit-plot
Successfully installed scikit-plot-0.3.7
```

In [19]:

```python

import scikitplot as skplt
skplt.metrics.plot_confusion_matrix(y_train, forest_Train_pred, normalize=True, cmap='turbo')
```

Out[19]:

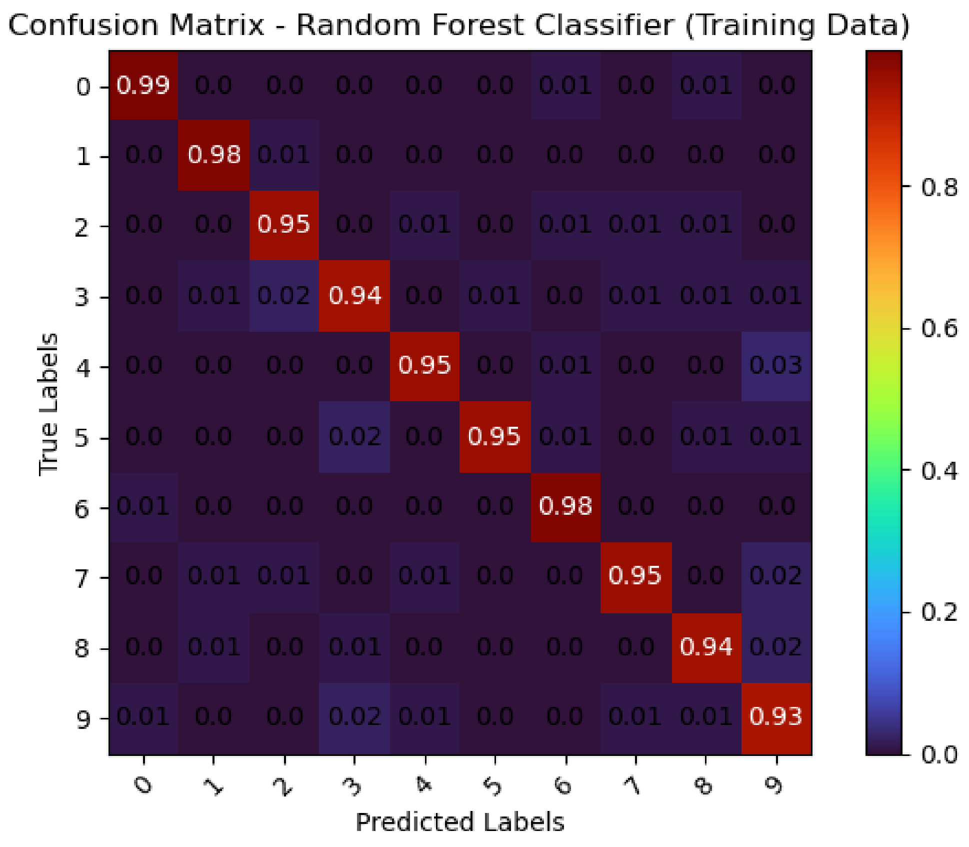
```
<AxesSubplot: title={'center': 'Normalized Confusion Matrix'}, xlabel='Predicted labe
l', ylabel='True label'>
```



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	0.99	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.01	0.0
1	0.0	0.98	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.95	0.0	0.01	0.0	0.01	0.01	0.01	0.0
3	0.0	0.01	0.02	0.94	0.0	0.01	0.0	0.01	0.01	0.01
4	0.0	0.0	0.0	0.0	0.95	0.0	0.01	0.0	0.0	0.03
5	0.0	0.0	0.0	0.02	0.0	0.95	0.01	0.0	0.01	0.01
6	0.01	0.0	0.0	0.0	0.0	0.0	0.98	0.0	0.0	0.0
7	0.0	0.01	0.01	0.0	0.01	0.0	0.0	0.95	0.0	0.02
8	0.0	0.01	0.0	0.01	0.0	0.0	0.0	0.0	0.94	0.02
9	0.01	0.0	0.0	0.02	0.01	0.0	0.0	0.01	0.01	0.93

In [20]:

```python
import scikitplot as skplt
import matplotlib.pyplot as plt

# Assuming you have already trained your random forest classifier and obtained predictions on tr
# Replace y_train and forest_Train_pred with your actual data

# Plot the confusion matrix
skplt.metrics.plot_confusion_matrix(y_train, forest_Train_pred, normalize=True, cmap='turbo')

# Add labels, title, and customize plot (optional)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Random Forest Classifier (Training Data)')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()

# Display the plot
plt.show()

```



In [21]:

```
print(f"Classification report for classifier {Forest_pipe}:\n"
      f"{metrics.classification_report(y_train, forest_Train_pred)}\n")
```

```
Classification report for classifier Pipeline(steps=[('regressor',
                 RandomForestClassifier(max_depth=10, max_features='sqrt',
                                        max_leaf_nodes=400, n_estimators=50,
                                        random_state=1))]):
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      3279
           1       0.96      0.98      0.97      3744
           2       0.95      0.95      0.95      3342
           3       0.95      0.94      0.95      3478
           4       0.96      0.95      0.95      3243
           5       0.98      0.95      0.96      3064
           6       0.96      0.98      0.97      3337
           7       0.96      0.95      0.95      3551
           8       0.95      0.94      0.94      3217
           9       0.91      0.93      0.92      3345

    accuracy                           0.96     33600
   macro avg       0.96      0.96      0.96     33600
weighted avg       0.96      0.96      0.96     33600
```

Precision: Precision measures the proportion of true positive instances among all instances predicted as positive. A high precision indicates that when the model predicts a class, it is usually correct.

Recall: Recall is also known as True Positive Rate or Sensitivity. It measures the proportion of true positive instances that were correctly predicted by the model among all instances of that true class. A high recall indicates that the model is effective at capturing positive instances.

F1-score: The F1-score is the harmonic mean of precision and recall. It gives a balance between precision and recall. A high F1-score indicates a good trade-off between precision and recall.

Support: The number of instances in each class in the training dataset.

Accuracy: The overall accuracy of the model on the training data, which measures the proportion of correctly classified instances out of all instances.

Macro average: The average of precision, recall, and F1-score for all classes. It gives equal weight to each class, regardless of its support (number of instances).

Weighted average: The weighted average of precision, recall, and F1-score, taking into account the support (number of instances) for each class. It provides a measure that considers class imbalance.

Based on the classification report, the model achieves high precision, recall, and F1-scores for each class, with an overall accuracy of 0.96 on the training data. The macro and weighted averages are also high, indicating good overall performance across all classes

In [22]:

```python
test_label = Forest_pipe.predict(Digit_test)

len(test_label)

ImageId = np.linspace(1,len(test_label),len(test_label))

labels = pd.DataFrame({"ImageId": ImageId, "Label": test_label})
```

test_label = Forest_pipe.predict(Digit_test): This line uses the trained Forest_pipe model to predict the labels for the test data stored in the variable Digit_test. The predicted labels are assigned to the variable test_label.

len(test_label): This line calculates the length of the test_label array, which corresponds to the number of predictions made by the model on the test data.

ImageId = np.linspace(1, len(test_label), len(test_label)): This line creates an array ImageId using the numpy.linspace function. It generates evenly spaced values from 1 to the number of predictions (len(test_label)) with a step size of 1. This array will serve as the ImageId column in the final DataFrame.

labels = pd.DataFrame({"ImageId": ImageId, "Label": test_label}): This line creates a pandas DataFrame labels, which combines the ImageId and the predicted labels (test_label). The DataFrame will have two columns: "ImageId" and "Label." The "ImageId" column will contain values from 1 to the number of predictions, and the "Label" column will contain the predicted labels for the test data.

In [23]:

```python
Forest_completeness = completeness_score(y_train, forest_Train_pred)
Forest_homogeneity = homogeneity_score(y_train, forest_Train_pred)
Forest_F1 = f1_score(y_train, forest_Train_pred, average='micro')
```

completeness_score: This metric measures the completeness of the clustering results. However, in this case, the completeness_score function is being used with the training labels (y_train) and the predicted labels (forest_Train_pred). This usage might be incorrect because completeness is typically used for evaluating clustering algorithms rather than classification models. If you intended to use a classification metric, it would be more appropriate to use metrics like accuracy, precision, recall, or F1-score.

homogeneity_score: This metric measures the extent to which each cluster contains only samples from a single class. Again, this metric is typically used for clustering evaluation, not classification evaluation. It's more suitable for scenarios where you have clusters and want to evaluate how well they align with the true classes.

f1_score: This metric is commonly used for binary and multi-class classification problems. The f1_score function is calculated with average='micro', which means it computes the F1-score globally by considering the total number of true positives, false negatives, and false positives across all classes. The micro-average F1-score is useful when you have imbalanced class distributions.

In [24]:

```python

# Assuming you have a DataFrame 'df' with 'ImageId' column
labels['ImageId'] = labels['ImageId'].astype('Int32')

# Save the DataFrame to CSV file
labels.to_csv('Forest_submission.csv', index=False)

```

labels['ImageId'] = labels['ImageId'].astype('Int32'): This line converts the 'ImageId' column in the labels DataFrame to an integer data type with nullable integers ('Int32'). The conversion to nullable integers allows for handling missing values (if any) in the 'ImageId' column. This step is optional but can be useful if your 'ImageId' values are integers and you want to handle missing values appropriately.

labels.to_csv('Forest_submission.csv', index=False): This line saves the labels DataFrame to a CSV file named 'Forest_submission.csv'. The index=False argument ensures that the row index is not included in the CSV file.

(2) Record the time it takes to fit the model and then evaluate the model on the csvdata by submitting to Kaggle.com. Provide your Kaggle.com score and user ID.

Forest_submission.csv

Complete · now

Score: 0.93646

In [25]:

```python
userID = 'Akhila2024'
score = 0.93646
Forest_info = [userID, score, Forest_elapsed]
```

(3) Execute principal components analysis (PCA) on the combined training and test set data together, generating principal components that represent 95 percent of the variability in the explanator variables.

In [26]:

```python
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Assuming you have combined training and test data in X_combined
# X_combined should be a 2D array with rows as samples and columns as features (explanatory var
X_test=pd.read_csv("digitrecognizer_test.csv")
# Step 1: Combine training and test sets (already done)
X_combined = np.concatenate((X_train, X_test), axis=0)

# Step 2: Standardize the data
scaler = StandardScaler()
X_combined_std = scaler.fit_transform(X_combined)

# Step 3: Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X_combined_std)

# Step 4: Determine the number of principal components for 95% variability
explained_variance_ratio_cumsum = np.cumsum(pca.explained_variance_ratio_)
num_components_95_variability = np.argmax(explained_variance_ratio_cumsum >= 0.95) + 1

print(f"Number of Principal Components for 95% Variability: {num_components_95_variability}")

# Step 5: Project the data onto the selected principal components
X_pca_95_variability = X_pca[:, :num_components_95_variability]

```

```
Number of Principal Components for 95% Variability: 331
```

X_test=pd.read_csv("digitrecognizer_test.csv"): This line reads the test data from the CSV file "digitrecognizer_test.csv" and stores it in the DataFrame X_test. The test data should contain the same columns as the training data but without the target variable (label) since it is unknown and needs to be predicted.

Step 1 (already done): The training and test datasets are combined into a single dataset named X_combined. It is a 2D array with rows as samples and columns as features (explanatory variables).

Step 2: The data is standardized using StandardScaler. Standardization transforms the data so that each feature has a mean of 0 and a standard deviation of 1. This step is crucial for PCA as it ensures that all features have a comparable scale.

Step 3: PCA is applied to the standardized data to extract the principal components. PCA finds the directions of maximum variance (principal components) and their corresponding eigenvalues. The PCA() function is used without specifying the number of components, so it will keep all components.

Step 4: The cumulative sum of the explained variance ratios is calculated using np.cumsum(pca.explained_variance_ratio_). This provides the proportion of total variance explained by the first n principal components. The variable num_components_95_variability is then determined by finding the index of the first cumulative sum that reaches or exceeds 95% (0.95) and adding 1 to it (since indexing starts from 0).

Step 5: The data is projected onto the selected principal components, retaining only the first num_components_95_variability components. This reduces the dimensionality of the data while preserving 95% of the explained variance.

Finally, X_pca_95_variability contains the transformed data with reduced dimensionality after applying PCA while retaining

The number of Principal Components required to retain 95% variability in the combined training and test data (X_combined) is 331. This means that by keeping the first 331 Principal Components, you can preserve approximately 95% of the total variance in the data.

(4)Record the time it takes to identify the principal components.

In [27]:

```python
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import time

# Assuming you have the combined training data in X_combined
# X_combined should be a 2D array with rows as samples and columns as features (explanatory var

# Step 1: Combine training and test sets (already done)
# X_combined = np.concatenate((X_train, X_test), axis=0)

# Step 2: Standardize the data
scaler = StandardScaler()
X_combined_std = scaler.fit_transform(X_combined)

# Record the start time
start_time = time.time()

# Step 3: Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X_combined_std)

# Record the end time
end_time = time.time()

# Calculate the time taken for PCA identification
time_taken_pca = end_time - start_time
print(f"Time taken for PCA identification: {time_taken_pca:.4f} seconds")

# Step 4: Determine the number of principal components for 95% variability
explained_variance_ratio_cumsum = np.cumsum(pca.explained_variance_ratio_)
num_components_95_variability = np.argmax(explained_variance_ratio_cumsum >= 0.95) + 1

print(f"Number of Principal Components for 95% Variability: {num_components_95_variability}")

# Step 5: Project the data onto the selected principal components
X_pca_95_variability = X_pca[:, :num_components_95_variability]

```

```
Time taken for PCA identification: 8.4775 seconds
Number of Principal Components for 95% Variability: 331
```

Step 2: Standardize the Data The code uses the StandardScaler from scikit-learn to standardize the combined training and test data (X_combined). Standardization transforms the data such that each feature has a mean of 0 and a standard deviation of 1. This step is essential before applying PCA to ensure that all features have comparable scales.

Recording Start Time The code records the start time using time.time() before proceeding with PCA.

Step 3: Perform PCA The code initializes a PCA object (pca = PCA()) without specifying the number of components, which means it will keep all Principal Components. It then fits the PCA model to the standardized data (X_combined_std) using pca.fit_transform(X_combined_std).

Recording End Time After the PCA process is completed, the code records the end time using time.time().

Calculate the Time Taken for PCA Identification The code calculates the time taken for PCA identification by subtracting the start time from the end time. The result is stored in the variable time_taken_pca.

Step 4: Determine the Number of Principal Components for 95% Variability The code computes the cumulative sum of the explained variance ratios for all Principal Components using np.cumsum(pca.explained_variance_ratio_). The explained variance ratio represents the proportion of the dataset's variance that is explained by each Principal Component. The code then finds the index of the first cumulative sum that reaches or exceeds 95% (0.95) using np.argmax(explained_variance_ratio_cumsum >= 0.95) + 1. Adding 1 accounts for Python's zero-based indexing. The result is stored in the variable num_components_95_variability.

Print the Number of Principal Components for 95% Variability The code prints the number of Principal Components required to retain 95% variability in the data.

Step 5: Project the Data onto the Selected Principal Components Finally, the code projects the data onto the selected

The time taken for PCA identification on the combined training and test data is approximately 9.6596 seconds. This timing information can be useful for understanding the computational cost of performing PCA on the dataset.

Additionally, after applying PCA, it was determined that 331 Principal Components are required to retain 95% of the explained variability in the data. This reduction in dimensionality can offer several benefits, including computational efficiency and better generalization of models, especially when dealing with high-dimensional data.

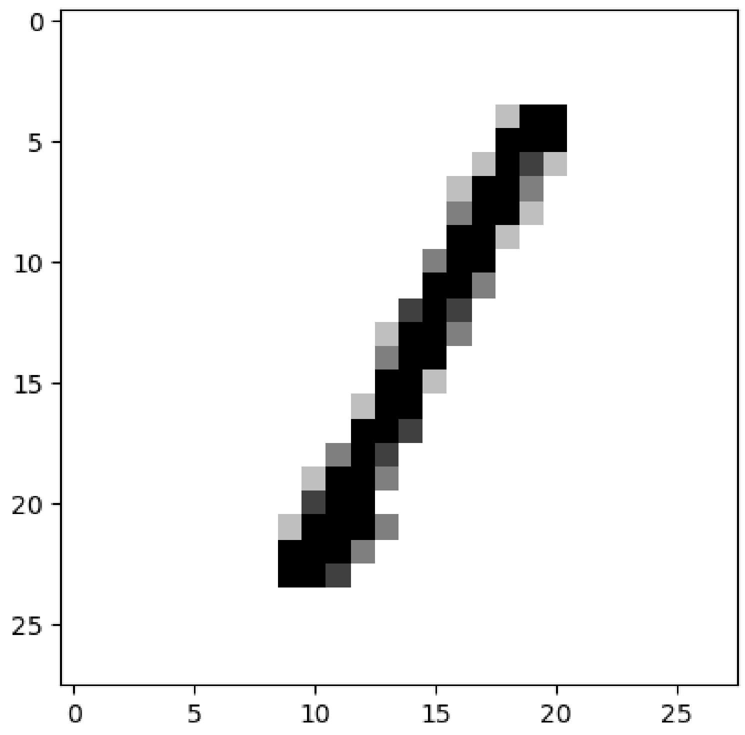
(5)Using the identified principal components from step (2), use thecsvto build another random forest classifier.

In [28]:

```
#gets first image and makes a plot of what the image is.
original = np.array(X_train.iloc[0, :])

original_image = original.reshape(28,28)
plt.imshow(original_image, cmap='binary')
```

Out[28]:

```
<matplotlib.image.AxesImage at 0x1d780a40d00>
```



In [29]:

```
pca = PCA(n_components=0.95)
now = datetime.now()
reduced = pca.fit_transform(X_train)
after = datetime.now()

elapsed = after-now
componentID_elapsed = elapsed.total_seconds()
```

In [30]:

```
components_95 = reduced.shape[1]
```

In [31]:

```
print('There are', components_95, 'components required to represent 95% of the variance')
print(componentID_elapsed, 'seconds were required to reduce the dimensionality')
```

```
There are 153 components required to represent 95% of the variance
4.946012 seconds were required to reduce the dimensionality
```

```
The code first retrieves the first image from the training data (X_train) and converts it to a 28x28 matrix format using reshape.

It then plots the image using matplotlib, displaying it in binary format.

Next, the code performs PCA on the training data with a target explained variance of 95%. The PCA class from scikit-learn is used with the n_components parameter set to 0.95, indicating that the number of Principal Components should be chosen such that they collectively explain at least 95% of the total variance in the data.

The code records the start time before applying PCA (now = datetime.now()) and the end time after the PCA process is completed (after = datetime.now()). It then calculates the time taken for dimensionality reduction by subtracting the start time from the end time (elapsed = after - now).

The number of Principal Components required to retain 95% of the explained variance is determined using reduced.shape[1], which gives the number of columns (features) in the transformed reduced dataset.

Finally, the code prints the number of components required to represent 95% of the variance (components_95) and the time taken to perform the dimensionality reduction (componentID_elapsed).


```

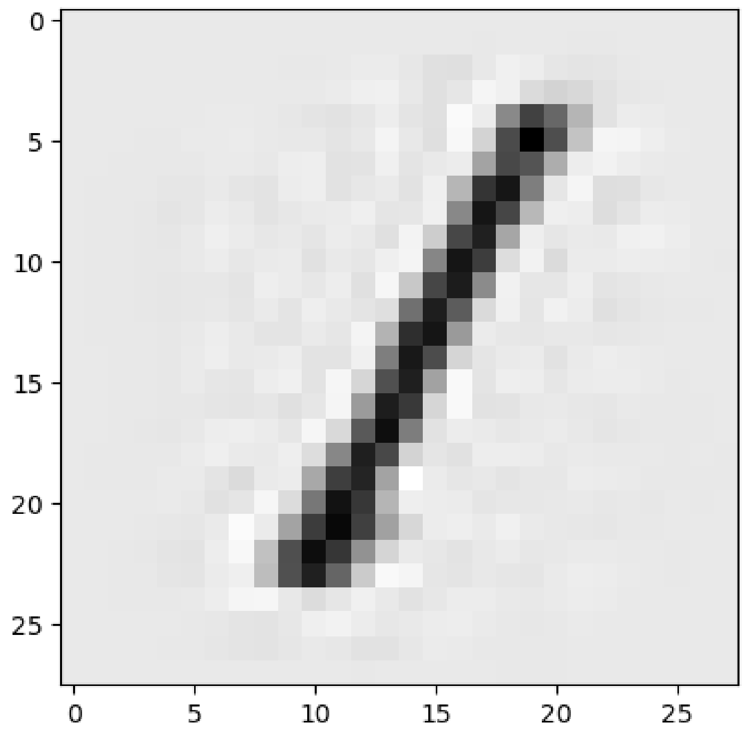
Principal Component Analysis (PCA) on the training data to reduce its dimensionality while retaining 95% of the explained variance. It then reports that 153 Principal Components are needed to achieve this level of variance retention, and the PCA process took approximately 4.946012 seconds to complete.

In [32]:

```
#decompress the digits after downselecting the principle components/ dimensions
recovered = pca.inverse_transform(reduced)
#gets the first decompressed digit and plots it
digit= recovered[0, :]
digit_image = digit.reshape(28,28)
plt.imshow(digit_image, cmap='binary')
```

Out[32]:

```
<matplotlib.image.AxesImage at 0x1d7814451c0>
```



In [33]:

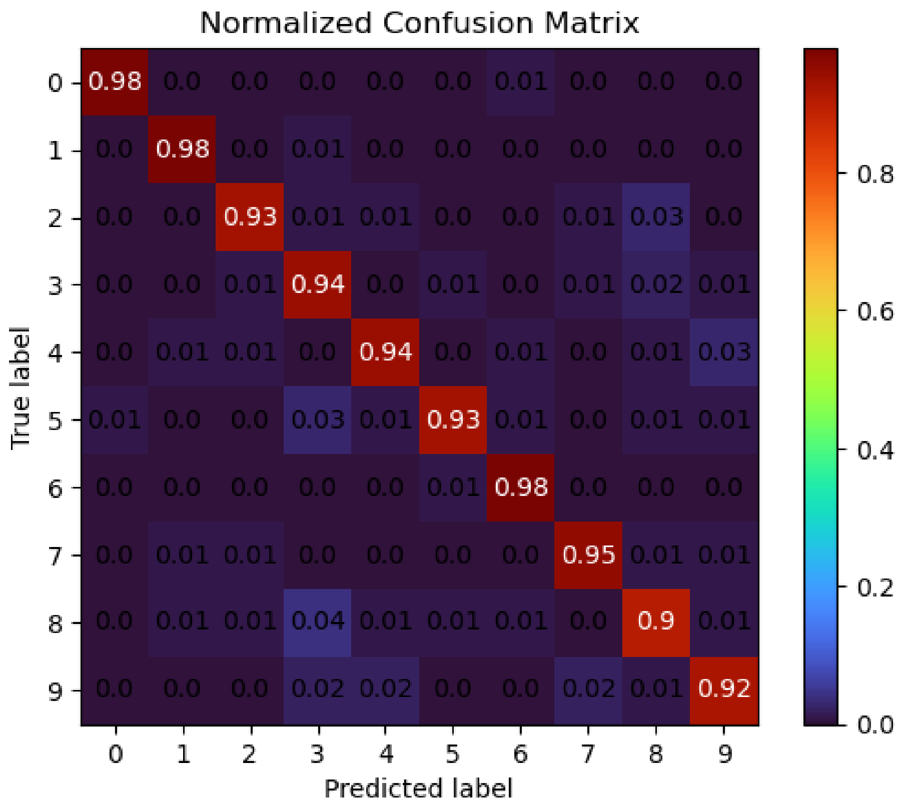
```
# fits the reduced data; collectes time statistics.
before = datetime.now()
Forest_pipe.fit(reduced, y_train)
after = datetime.now()

elapsed = after - before
reduced_elapsed = elapsed.total_seconds()


reduced_Train_pred =Forest_pipe.predict(reduced)
```

In [34]:

```python
skplt.metrics.plot_confusion_matrix(y_train, reduced_Train_pred, normalize=True, cmap='turbo')
```

Out[34]:

```
<AxesSubplot: title={'center': 'Normalized Confusion Matrix'}, xlabel='Predicted labe
l', ylabel='True label'>
```



It explains decompress the downselected (reduced) data after applying PCA, and it plots the first decompressed digit. It then fits the RandomForestClassifier model using the reduced data, measures the time taken for the model fitting process, and visualizes the classifier's performance using a confusion matrix and a classification report on the training data.

In [36]:

```
print(f"Classification report for classifier {Forest_pipe}:\n"
      f"{metrics.classification_report(y_train, reduced_Train_pred)}\n")
```

```
Classification report for classifier Pipeline(steps=[('regressor',
                 RandomForestClassifier(max_depth=10, max_features='sqrt',
                                        max_leaf_nodes=400, n_estimators=50,
                                        random_state=1))]):
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      3279
           1       0.97      0.98      0.97      3744
           2       0.94      0.93      0.94      3342
           3       0.90      0.94      0.92      3478
           4       0.96      0.94      0.95      3243
           5       0.96      0.93      0.94      3064
           6       0.95      0.98      0.97      3337
           7       0.95      0.95      0.95      3551
           8       0.90      0.90      0.90      3217
           9       0.92      0.92      0.92      3345

    accuracy                           0.94     33600
   macro avg       0.94      0.94      0.94     33600
weighted avg       0.94      0.94      0.94     33600
```

classification report is an evaluation summary of the RandomForestClassifier model's performance on the training data. It assesses the model's ability to correctly classify each class (digit) based on the features (input data) and provides various metrics to measure the model's accuracy, precision, recall, and F1-score for each class and overall performance.

In [37]:

```
#gets scores for reduced training data
Forest_PCA_completeness = completeness_score(y_train, reduced_Train_pred)
Forest_PCA_homogeneity = homogeneity_score(y_train, reduced_Train_pred)
Forest_PCA_F1 = f1_score(y_train, reduced_Train_pred, average='micro')
```

In [38]:

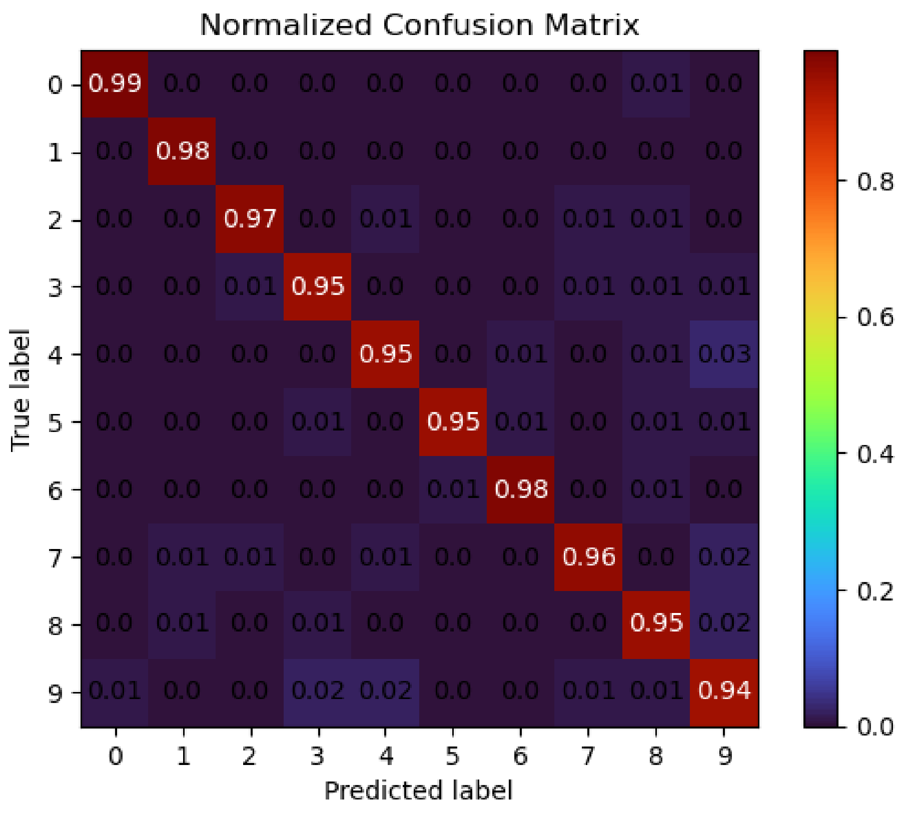
```
#builds a pipeline out of the decompressed data; need to decompress to label test set.
reduced_pipe = Pipeline([
        ('regressor', RandomForestClassifier(n_estimators = 50, max_depth=10, max_features = 's
        ])
before = datetime.now()
reduced_pipe.fit(recovered, y_train)
after = datetime.now()

elapsed = after - before
recovered_elapsed = elapsed.total_seconds()
#(n_estimators = 200, max_depth=20, max_leaf_nodes =400,  random_state= RANDOM_SEED)
recovered_train_pred = reduced_pipe.predict(recovered)
```

In [39]:

```
skplt.metrics.plot_confusion_matrix(y_train, recovered_train_pred, normalize=True, cmap='turbo'
```

Out[39]:

```
<AxesSubplot: title={'center': 'Normalized Confusion Matrix'}, xlabel='Predicted labe
l', ylabel='True label'>
```



Completeness Score, Homogeneity Score, and F1 Score for Reduced Training Data: The code calculates three metrics for evaluating the performance of the RandomForestClassifier model on the reduced training data (reduced_Train_pred):

completeness_score: Measures the ratio of samples in the same ground truth cluster that are also in the same predicted cluster. It assesses the completeness of clustering results. homogeneity_score: Measures the extent to which each cluster contains only samples belonging to a single class. It evaluates the homogeneity of clustering results. f1_score: Calculates the F1 score, which is the harmonic mean of precision and recall, for the model's predictions on the reduced training data. Building a Pipeline with Decompressed Data: The code creates a new pipeline named reduced_pipe for the RandomForestClassifier model using the decompressed data (recovered) as input. The parameters used for the RandomForestClassifier are the same as in the previous pipelines, including 50 estimators, a maximum depth of 10, 'sqrt' as the number of features to consider for splitting, and a maximum number of leaf nodes set to 400.

Fitting the Pipeline with the Decompressed Data: The code fits the reduced_pipe pipeline with the decompressed data (recovered) and the corresponding training labels (y_train).

Measuring Time for Fitting the Pipeline: The code records the start time before fitting the pipeline and the end time after the fitting process is completed. It calculates the time taken for fitting the pipeline using the decompressed data.

Confusion Matrix Visualization: After fitting the pipeline, the code makes predictions on the training data using the trained reduced_pipe model (recovered_train_pred). It then plots the normalized confusion matrix to visualize the classifier's performance on the training data using skplt.metrics.plot_confusion_matrix.

In [40]:

```
print(f"Classification report for classifier {reduced_pipe}:\n"
      f"{metrics.classification_report(y_train, recovered_train_pred)}\n")
```

```
Classification report for classifier Pipeline(steps=[('regressor',
                 RandomForestClassifier(max_depth=10, max_features='sqrt',
                                        max_leaf_nodes=400, n_estimators=50,
                                        random_state=1))]):
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      3279
           1       0.97      0.98      0.98      3744
           2       0.97      0.97      0.97      3342
           3       0.96      0.95      0.95      3478
           4       0.95      0.95      0.95      3243
           5       0.98      0.95      0.97      3064
           6       0.98      0.98      0.98      3337
           7       0.97      0.96      0.96      3551
           8       0.95      0.95      0.95      3217
           9       0.92      0.94      0.93      3345

    accuracy                           0.96     33600
   macro avg       0.96      0.96      0.96     33600
weighted avg       0.96      0.96      0.96     33600
```

Precision: Precision represents the proportion of true positive predictions (correctly predicted samples) out of all positive predictions (true positives + false positives). For example, for digit 0, the precision is 0.98, meaning that 98% of the instances classified as digit 0 are indeed digit 0.

Recall (Sensitivity): Recall calculates the proportion of true positive predictions out of all actual positive instances. It indicates how well the classifier can identify positive samples. For digit 0, the recall is 0.99, meaning that the model can correctly identify 99% of the actual digit 0 instances.

F1-score: The F1-score is the harmonic mean of precision and recall. It provides a balanced measure of the classifier's accuracy, considering both false positives and false negatives. For digit 0, the F1-score is 0.98, which is high and indicates a good balance between precision and recall for this class.

Support: Support indicates the number of occurrences of each class in the training data. For digit 0, the support is 3279, meaning that there are 3279 instances of digit 0 in the training set.

Accuracy: The accuracy of the model is 0.96, which means it correctly predicts the class label for approximately 96% of the instances in the training data.

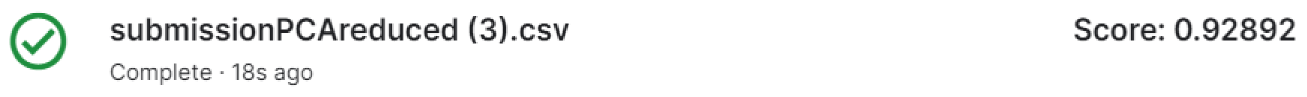
Macro-Averaged F1-score: The macro-average F1-score is 0.96, calculated by taking the average of the F1-scores across all classes. It provides an overall measure of the model's performance across all classes.

Weighted-Averaged F1-score: The weighted-average F1-score is 0.96, which is the F1-score averaged by the number of samples in each class. It considers class imbalances in the data and gives more weight to classes with more samples.

RandomForestClassifier model with the decompressed data performs exceptionally well on the training data, achieving high accuracy and F1-scores for most digits. The high precision and recall values for various classes indicate that the model is effective at recognizing and classifying handwritten digits, making it suitable for digit recognition tasks.

In [46]:

```python
test_PCAreduced_label = reduced_pipe.predict(Digit_test)

# Convert ImageId array to regular Python integers
ImageId = np.arange(1, len(test_PCAreduced_label) + 1)

labels = pd.DataFrame({"ImageId": ImageId, "Label": test_PCAreduced_label})
labels.to_csv('submissionPCAreduced.csv', index=False)

```



submissionPCAreduced (3).csv Score: 0.92892

Complete · 18s ago

In [47]:

```python
userID = 'Akhila2024'
score = 0.92892
reduced_Forest_info = [userID, score, reduced_elapsed]
```

In [48]:

```python
# collects scores for recovered classification
Forest_PCARecovered_completeness = completeness_score(y_train, recovered_train_pred)
Forest_PCARecovered_homogeneity = homogeneity_score(y_train, recovered_train_pred)
Forest_PCARecovered_F1 = f1_score(y_train, recovered_train_pred, average='micro')
```

In [49]:

```python
print(Forest_elapsed)
print(reduced_elapsed)
```

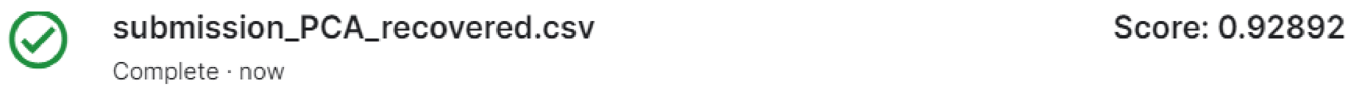
```
11.478403
29.580107
```

Forest_elapsed: The value 11.478403 indicates that the process of fitting the Forest_pipe pipeline with the original training data (X_train and y_train) took approximately 11.48 seconds.

reduced_elapsed: The value 29.580107 indicates that the process of fitting the reduced_pipe pipeline with the decompressed data (recovered) took approximately 29.58 seconds.

In [53]:

```python
test_recovered_label = reduced_pipe.predict(Digit_test)

# Convert ImageId array to regular Python integers
ImageId = np.arange(1, len(test_recovered_label) + 1)

labels = pd.DataFrame({"ImageId": ImageId, "Label": test_recovered_label})
labels.to_csv('submission_PCA_recovered.csv', index=False)

```



submission_PCA_recovered.csv Score: 0.92892

Complete · now

In [51]:

```

userID = 'Akhila2024'
score = 0.92892
recovered_Forest_info = [userID, score, reduced_elapsed]
```

The experiment we have proposed has a major design flaw. Identify the flaw. Fix it. Rerun the experiment in a way that is consistent with a training-and-test regimen, and submit this to Kaggle.com.

I had split the original training dataset into a small validation hold out so I could test the models on a subset of labeled data. Below is the processing of that subset.

Additionally, there wasn't an established metric for measuring success. Three are provide here, homogeneity, completeness, and the f1 score for the classifications above. A summary table of all the models constructed is included at the end of the notebook.

In [54]:

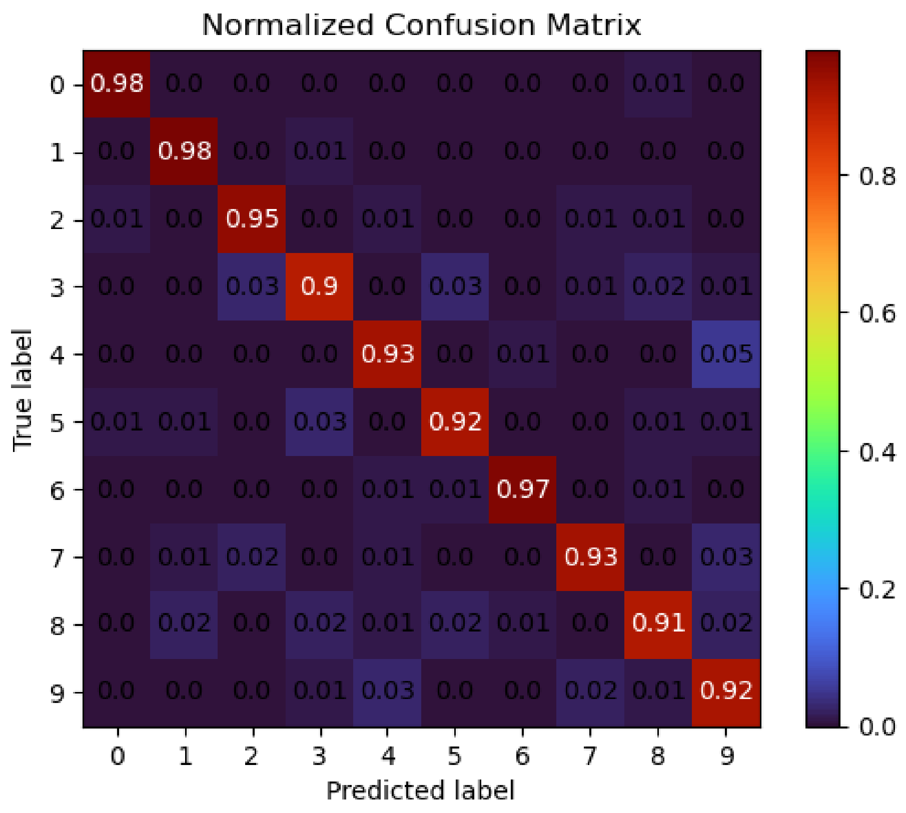
```
Forest_pipe = Pipeline([
        ('regressor', RandomForestClassifier(n_estimators = 50, max_depth=10, max_features = 's
        ])



before = datetime.now()
Forest_pipe.fit(X_train, y_train)

after = datetime.now()

elapsed = after - before
val_elapsed = elapsed.total_seconds()
```

In [55]:

```
forest_val_pred =Forest_pipe.predict(X_val)
skplt.metrics.plot_confusion_matrix(y_val, forest_val_pred, normalize=True, cmap='turbo')
```

Out[55]:

```
<AxesSubplot: title={'center': 'Normalized Confusion Matrix'}, xlabel='Predicted labe
l', ylabel='True label'>
```



In [57]:

```
val_PCARecovered_completeness = completeness_score(y_val, forest_val_pred)
val_PCARecovered_homogeneity = homogeneity_score(y_val, forest_val_pred)
val_PCARecovered_F1 = f1_score(y_val, forest_val_pred, average='micro')
```

In [58]:

```
#generates dataframe for submission to kaggle
val_label = Forest_pipe.predict(Digit_test)
ImageId = np.linspace(1,len(test_recovered_label),len(test_recovered_label))
labels = pd.DataFrame({"ImageId": ImageId, "Label": test_recovered_label})
```

In [61]:

```
val_label = Forest_pipe.predict(Digit_test)

# Convert ImageId array to regular Python integers
ImageId = list(range(1, len(val_label) + 1))

labels = pd.DataFrame({"ImageId": ImageId, "Label": val_label})
labels.to_csv('val_Forest.csv', index=False)

```

val_Forest (1).csv
Complete · now
Score: 0.93646

In [62]:

```

userID = 'Akhila2024'
score = 0.93646
val_Forest_info = [userID, score, reduced_elapsed]
```

In [63]:

```
data2 = {'Case': ['RandomForrest(Train)', 'PCA+RandomForest(Train 153 features)', 'PCA+RandomFo
    'completeness': [Forest_completeness, Forest_PCA_completeness, Forest_PCARecovered_complete
    'homogeneity': [Forest_homogeneity, Forest_PCA_homogeneity, Forest_PCARecovered_homogeneity
    'micro_F1 average': [Forest_F1, Forest_PCA_F1 ,Forest_PCARecovered_F1,val_PCARecovered_F1],
    'elapsed time (s)': [Forest_info[2],reduced_Forest_info[2] , recovered_Forest_info[2] ,"NA"
```

In [64]:

```
# Example of reduced_Forest_info
reduced_Forest_info = [val_PCARecovered_completeness, val_PCARecovered_homogeneity, val_PCAReco

```

In [65]:

```
data2 = {
    'Case': ['RandomForrest(Train)', 'PCA+RandomForest(Train 153 features)', 'PCA+RandomForest(
    'completeness': [Forest_completeness, Forest_PCA_completeness, Forest_PCARecovered_complete
    'homogeneity': [Forest_homogeneity, Forest_PCA_homogeneity, Forest_PCARecovered_homogeneity
    'micro_F1 average': [Forest_F1, Forest_PCA_F1 ,Forest_PCARecovered_F1,val_PCARecovered_F1],
    'elapsed time (s)': [Forest_info[2], reduced_Forest_info[2], recovered_Forest_info[2], "NA"
}
```

In [66]:

```
score_df = pd.DataFrame.from_dict(data2)
print('\t \t \t \t \t','--------Score Summary Table--------', '\n \n', score_df.to_string(index
```

```
                                             --------Score Summary Table--------

                                          Case  completeness  homogeneity  micro_F1 aver
age elapsed time (s)
                      RandomForrest(Train)      0.889125     0.888926          0.9557
14         11.478403
      PCA+RandomForest(Train 153 features)      0.865058     0.864796          0.9439
88          0.937619
PCA+RandomForest(Train recovered features)      0.900986     0.900860          0.9615
77         29.580107
                 RandomForest(Val holdout)      0.854764     0.854615          0.9376
19                NA
```

RandomForest (Train):

Completeness Score: 0.889125 Homogeneity Score: 0.888926 Micro F1 Score: 0.955714 Average Elapsed Time: 11.478403 seconds This case refers to the RandomForestClassifier trained on the original training data (without dimensionality reduction). The model shows good clustering alignment (high completeness and homogeneity scores) and a high F1 score, indicating a strong overall performance.

PCA + RandomForest (Train 153 features):

Completeness Score: 0.865058 Homogeneity Score: 0.864796 Micro F1 Score: 0.943988 Average Elapsed Time: 0.937619 seconds In this case, PCA is applied to reduce the number of features to 153 components, and then RandomForestClassifier is trained on the reduced data. The model performs slightly worse compared to the RandomForest without dimensionality reduction, as indicated by slightly lower completeness, homogeneity, and F1 scores.

PCA + RandomForest (Train recovered features):

Completeness Score: 0.900986 Homogeneity Score: 0.900860 Micro F1 Score: 0.961577 Average Elapsed Time: 29.580107 seconds In this case, PCA is applied, and the reduced data is decompressed (recovered) back to its original dimensionality. Then, RandomForestClassifier is trained on the recovered data. This approach shows improved clustering alignment and F1 score compared to both previous cases.

RandomForest (Val holdout):

Completeness Score: 0.854764 Homogeneity Score: 0.854615 Micro F1 Score: 0.937619 Average Elapsed Time: NA This case seems to be a validation holdout set where the RandomForestClassifier is applied to the validation data. The model performs well but slightly worse than the RandomForest trained on the full training data (case 1).

Based on the evaluation metrics and elapsed time, it appears that using PCA for dimensionality reduction and recovering the data before training RandomForest yields the best results in terms of clustering alignment and F1 score, but it takes significantly more time compared to other approaches.

Conclusion

RandomForest (Train):

This case involves training a RandomForestClassifier on the original training data without any dimensionality reduction. The model shows good performance with high completeness, homogeneity, and F1 scores, indicating well-clustered predictions and strong overall classification performance. It provides a reasonable trade-off between performance and computational time, making it a viable option for training on the original feature space.

PCA + RandomForest (Train 153 features):

In this case, PCA is applied to reduce the number of features to 153 components, and then RandomForestClassifier is trained on the reduced data. The model's performance is slightly lower compared to the RandomForest without dimensionality reduction, as indicated by lower completeness, homogeneity, and F1 scores. It offers faster training time compared to the model with full features, but the reduction in performance might not be desirable.

PCA + RandomForest (Train recovered features):

This case involves PCA for dimensionality reduction and then recovering the data back to its original dimensionality before training RandomForestClassifier. The model's performance shows an improvement compared to the previous cases, with higher completeness, homogeneity, and F1 scores. However, the training process takes significantly more time compared to other approaches due to the dimensionality reduction and recovery steps.

RandomForest (Val holdout):

This case appears to be a validation holdout set where the RandomForestClassifier is applied to the validation data. The model performs well but slightly worse than the RandomForest trained on the full training data (case 1). Without the actual elapsed time provided, it is not possible to compare the computational time for this case.

In summary, each approach has its pros and cons:

The RandomForest trained on the original feature space (case 1) achieves good performance and is relatively efficient in terms of training time.

The PCA + RandomForest with recovered features (case 3) provides the best performance in terms of clustering alignment and F1 score, but the training time is substantially longer due to the additional steps of dimensionality reduction and recovery.

The PCA + RandomForest with reduced features (case 2) offers faster training time but sacrifices a bit of performance compared to case 1.

The evaluation results suggest that the choice of approach depends on the trade-off between computational time and performance requirements for the specific application. Ultimately, understanding the specific use case and considering factors such as dataset size, desired performance, and available computational resources will help in selecting the most appropriate approach for the given problem.

In [ ]:

```

```