MODULE 6: Digit Recognizer

Claire Markey, Julia Granito, Manny Hurtado, and Steve Desilets

MSDS 422: Practical Machine Learning

May 7th, 2023

**Introduction**

Classification of handwritten digits may be accomplished through the use of classification methods. Random forests and K-means clustering are methods that may be utilized to build classifiers that can assign the correct label to the images. To that end, we sought to explore how those classification methods may be best used or modified to correctly classify digit images.

**Method**

Kaggle data containing images of hand-drawn digits from zero through nine were downloaded and analyzed using Jupyter Notebooks (AstroDave and Cukierski, 2012). Random Forest Classification methods were employed using the original pixel data and its principal components to construct models that classified digits given the input pixel data. Principal components were determined using principal component analysis (PCA). KMeans clustering methods were also explored to predict hand-drawn digits using the original pixel data. Model performance metrics were examined and compared.

**Results and Insights**

First, examination of the training and test data confirmed that there was data missingness to handle using data imputation methods. An exploratory data analysis (EDA) revealed that the training and testing datasets contain numeric data for 784 pixels (28x28 images) for each of the hand-drawn digits in the data. Pixel-values ranged from 0 to 255, where each value indicates the lightness or darkness of that pixel.

A Random Forest classifier was built that utilized training data only. The training data was first split into training and validation sets, then hyperparameter tuning was conducted using five-fold cross-validation. The runtime for this Random Forest classifier was 1 minute and 3 seconds. This approach yielded an accuracy of 0.99 and received a score of 0.964 on Kaggle.

A Random Forest analysis was conducted using the principal components from these digit datasets. To do so, the pixel data from the training and test datasets were first combined, the predictors were then scaled via standardization, and a PCA was conducted to identify 334 principal components that collectively accounted for 95% of the variation in the predictors. The runtime for this PCA was 10.9 seconds.

The training dataset digit labels and these 334 principal components were then concatenated, or combined, so that a random forest model could be conducted. The training dataset was split into training and validation datasets and then the model's hyperparameters (number of trees, maximum features considered when splitting nodes, and maximum tree depth) were tuned using five-fold cross-validation. The runtime for the creation of this random forest model was 3 minutes and 18 seconds. Application of the best model from the hyperparameter tuning process to the validation and testing datasets yielded accuracies of 91.12% and 91.14%, respectively. Subsequently, another random forest model (via similarly structured five-fold cross validation) was created that included the 334 principal components and the original 784 pixel variables. The runtime for the creation of this random forest model was 2 minutes and 45 seconds. Application of the best random forest model from the hyperparameter tuning process to the validation and testing datasets both independently yielded accuracies of 94.56%.

It is imperative to address a method design flaw encountered at this point in the analysis; components are extracted from the combined training and testing data and then used to build another classifier on the

already-seen training data. This process is inefficient, because extracting components from high-dimensional data can be resource intensive (hence the emphasis in the problem on compute time). In theory, a model is trained on, or inferences are drawn from, a smaller set of data from which predictive metrics can be generalized to (potentially a larger amount of) new unseen data. Through training and testing the model, appropriate hyperparameter tuning can be performed to improve model robustness. With large scale data, training on less data saves computational time relative to training on more data, and using this technique in conjunction with a good train-test regime can give an estimate of model performance. Since this week's assignment uses a dataset that contains larger scale data, these considerations become more critical. Another consideration may be the 95% explained variance. Depending on the problem, 95% explained variance may not be necessary if a model using fewer components is more robust or can address the problem needs with a more optimal tradeoff accuracy and compute resources.

KMeans clustering was then utilized to group MNIST observations and then assign labels. The predictor variables were first normalized and next functions were defined to predict which digit corresponds to each cluster (Salaria, 2022). These functions were necessary because KMeans clustering is an unsupervised learning method so the labels assigned by the algorithm correspond to the cluster, not the predicted digit. The KMeans model was tuned to find the optimal number of clusters that capture the underlying data structure by plotting inertia within cluster (sum of squares) versus k (number of clusters), and silhouette scores versus k. K values of 10, 36, 64, 144, 256, and 400 were examined.

The elbow in the inertia plot suggested that 144 and 256 were the optimal number of clusters for the data. Since only 10 digits are represented by the data, a model with 10 clusters was constructed. Application of these models which grouped training data into 10, 144, and 256 clusters yielded accuracies of 59.4%, 88.1%, and 92.1%, respectively. The KMeans model with 256 clusters also had the highest homogeneity score (0.837) out of the three KMeans models that were tested. This indicated that the generated clusters were homogeneous, meaning that the model did a good job of grouping samples belonging to the same true class together in the same cluster. These metrics suggest that the model effectively captured the underlying structure of the data. To further demonstrate this model's performance, it yielded a 90.8% accuracy score on the testing data in Kaggle. Overall, this KMeans cluster analysis demonstrates that although there are 10 digits represented in the data, 10 clusters do not adequately capture the underlying structure of the data. Differences in the style of identical digits or oddly shaped clusters could explain why more clusters were needed to adequately capture intra-class differences among the hand-written digits in the data.

This comparative analysis of classification methods demonstrates that random forest and KMean models may be used as effective tools to classify images of digits. While some practical limitations presented themselves, such as long lengths of time to conduct a random forest classification analysis using principal components, these applications yielded high accuracy scores across the board. Using unsupervised learning methods, supervised learning methods, and combinations of both methods yielded good, accurate classification of digit data.

**References**

AstroDave, and Will Cukierski. 2012. “Digit Recognizer.” *Kaggle.* https://www.kaggle.com/c/digit-recognizer

Salaria, Sajjad. 2022. “K Means Clustering for Imagery Analysis.” *Medium*. DataDrivenInvestor. https://medium.datadriveninvestor.com/k-means-clustering-for-imagery-analysis-56c9976f16b6#:~:text=Preprocessing.

# Appendix 1 - Python Code and Outputs

## Data Preparation

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

## Import Training Data

In [2]:
```
import numpy as np
import pandas as pd
# load training data
digit_training_data = pd.read_csv('train.csv')

# show first rows of the data
digit_training_data.head(100)
# show number of columns and rows
digit_training_data.shape
```

Out[2]:

| | label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | ... | pixel774 | pixel775 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **3** | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **95** | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **96** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **97** | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **98** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **99** | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |

100 rows × 785 columns

Out[2]:
```
(42000, 785)
```

## Investigation of Missing Data and Outliers in Training Data

In [3]:
```
# find null counts, percentage of null values, and column type
null_count = digit_training_data.isnull().sum()
null_percentage = digit_training_data.isnull().sum() * 100 / len(digit_training_data)
column_type = digit_training_data.dtypes
```

```
# show null counts, percentage of null values, and column type for columns with more t
null_summary = pd.concat([null_count, null_percentage, column_type], axis=1, keys=['Mi
null_summary_only_missing = null_summary[null_count != 0].sort_values('Percentage Miss
null_summary_only_missing
```

Out[3]:

| | Missing Count | Percentage Missing | Column Type |
|---|---|---|---|

The above analysis displays that there is no missing data in the digit recognizer training dataset.

## Import Testing Data

In [4]:
```
# import test dataset
digit_testing_data = pd.read_csv('test.csv')

# show first ten rows of the data
digit_testing_data.head(10)
# show number of columns and rows
digit_testing_data.shape
```

Out[4]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | pixel774 | pixel775 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |

10 rows × 784 columns

Out[4]:
```
(28000, 784)
```

## Investigation of Missing Data and Outliers in Training Data

In [5]:
```
# find null counts, percentage of null values, and column type
null_count = digit_testing_data.isnull().sum()
null_percentage = digit_testing_data.isnull().sum() * 100 / len(digit_training_data)
column_type = digit_testing_data.dtypes

# show null counts, percentage of null values, and column type for columns with more t
null_summary = pd.concat([null_count, null_percentage, column_type], axis=1, keys=['Mi
```

```
null_summary_only_missing = null_summary[null_count != 0].sort_values('Percentage Miss
null_summary_only_missing
```

Out[5]:

| | Missing Count | Percentage Missing | Column Type |
|---|---|---|---|

The above analysis displays that there is no missing data in the digit recognizer test dataset.

## Apply Principal Components Analysis (PCA) to Combined Training and Test Data

First, we will combine the training and test dataframes

In [6]:

```
# Create a copy of the training dataframe
pca_train_df = digit_training_data.copy(deep=True)

# Drop the label column from the copy of the training dataframe
pca_train_df.drop(['label'], axis=1, inplace=True)

# Concatenate the training and test dataframes
pca_df = pd.concat([pca_train_df, digit_testing_data])

# show first rows of the data
pca_df.head(10)
# show number of columns and rows
pca_df.shape
# Describe the dataframe
pca_df.describe()


# find null counts, percentage of null values, and column type
null_count = pca_df.isnull().sum()
null_percentage = pca_df.isnull().sum() * 100 / len(digit_training_data)
column_type = pca_df.dtypes

# show null counts, percentage of null values, and column type for columns with more t
null_summary = pd.concat([null_count, null_percentage, column_type], axis=1, keys=['Mi
null_summary_only_missing = null_summary[null_count != 0].sort_values('Percentage Miss
null_summary_only_missing
```

Out[6]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | pixel774 | pixel775 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 |

10 rows × 784 columns

Out[6]:
```
(70000, 784)
```

Out[6]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | ... | 70 |
| **mean** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **std** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **min** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **25%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **50%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **75%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **max** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |

8 rows × 784 columns

Out[6]:

| | Missing Count | Percentage Missing | Column Type |
|---|---|---|---|

## Construct a Random Forest Model Using the full training model

First let's load the required packages:

In [10]:
```
#Import required Modules
#pip install graphviz

import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_
```

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from sklearn.tree import export_graphviz
from IPython.display import Image
import graphviz
```

Next, the training and validation datasets were utilized to conduct hyperparameter tuning to find the best hyperparameters for random forest modeling.

In [11]:
```
# Start a timer for the Random Forest
rf_start = datetime.datetime.now()

# Import Required Modules
#pip install graphviz
#import pandas as pd
#import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from scipy.stats import randint
from sklearn.tree import export_graphviz
from IPython.display import Image
import graphviz

# Create a copy of the training dataframe
rf_train_df = digit_training_data.copy(deep=True)

# Drop the label column from the copy of the training dataframe
rf_train_df.drop(['label'], axis=1, inplace=True)


# Split the training dataset into predictor and outcome components
rf_train_x = rf_train_df
rf_train_y = digit_training_data['label']

# Split the Kaggle training data into training and validation components
rf_x_train, rf_x_validation, rf_y_train, rf_y_validation = train_test_split(rf_train_x
                                                                            rf_train_y,
                                                                                  test_size=
                                                                                 random_stat


# Conduct hyperparameter tuning for random forest models
param_dist = {'n_estimators': randint(10,100),
              'max_depth': randint(1,100),
             'max_features': randint(1,20)}

rf = RandomForestClassifier()

#This approach uses 5-fold cross-validation
rand_search = RandomizedSearchCV(rf,
                                 param_distributions = param_dist,
                                 n_iter=5,
                                 cv=5)

rand_search.fit(rf_train_x, rf_train_y)

# Create a variable for the best model
best_rf = rand_search.best_estimator_
```

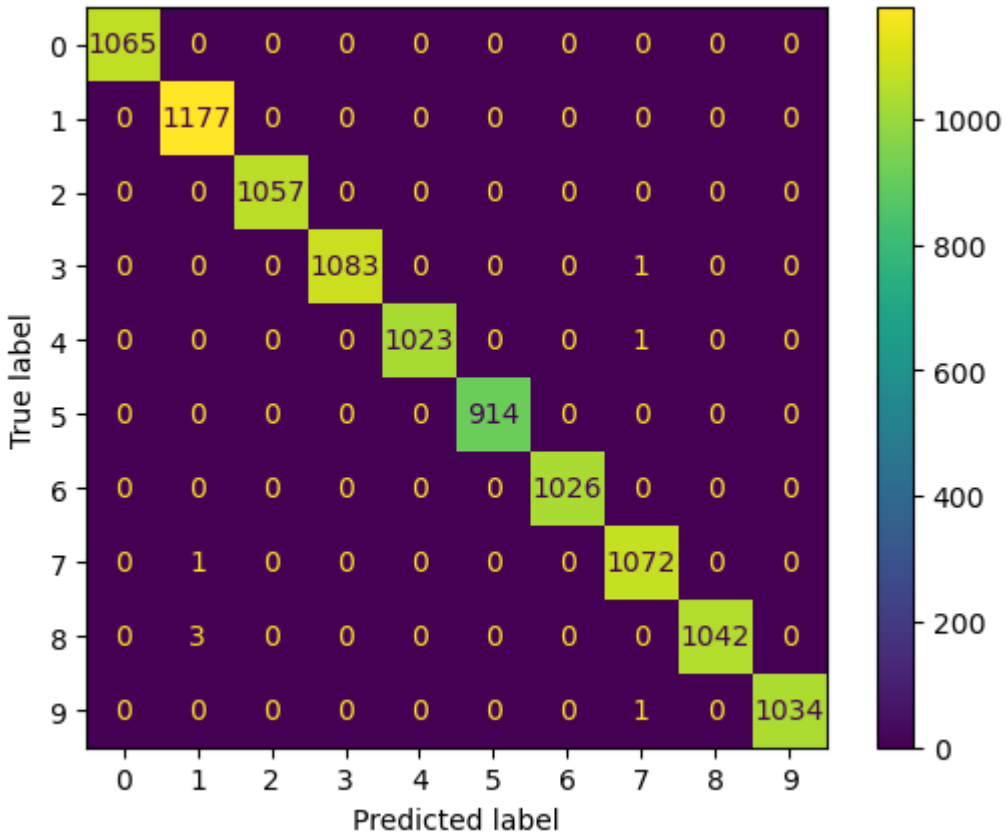
```
# Print the best hyperparameters
print('Best hyperparameters:',  rand_search.best_params_)

# Print the run time for Python to complete the Random Forest
rf_end = datetime.datetime.now()
rf_runtime = rf_end - rf_start
print(f"The total run time for the Random Forest Model using the training dataset was
```

Out[11]:
```
RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(), n_iter=5,
                   param_distributions={'max_depth': <scipy.stats._distn_infrastructu
re.rv_discrete_frozen object at 0x000001DC14C4A910>,
                                        'max_features': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DB9292ED30>,
                                        'n_estimators': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DB9292E580>})
Best hyperparameters: {'max_depth': 18, 'max_features': 15, 'n_estimators': 33}
The total run time for the Random Forest Model using the training dataset was 0:01:0
3.473638.
```

Next, let's examine the strength of the random forest model associated with the optimal hyperparameters by applying the model to the validation dataset and examining the resulting confusion matrix, accuracy, precision, and recall.

In [12]:
```
# Generate predictions with the best model
y_predictions_rf = best_rf.predict(rf_x_validation)

# Create the confusion matrix associated with the best random forest model
cm = confusion_matrix(rf_y_validation, y_predictions_rf)

ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Calculate the accuracy associated with the predictions of the best random forest mod

accuracy_rf_validation = accuracy_score(rf_y_validation, y_predictions_rf)

print("Accuracy:", accuracy_rf_validation)
```

Out[12]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1db929f4100>

Accuracy: 0.9993333333333333
```




In [13]:
```
from sklearn.metrics import classification_report
# print classification report
print(classification_report(rf_y_validation, y_predictions_rf))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1065
           1       1.00      1.00      1.00      1177
           2       1.00      1.00      1.00      1057
           3       1.00      1.00      1.00      1084
           4       1.00      1.00      1.00      1024
           5       1.00      1.00      1.00       914
           6       1.00      1.00      1.00      1026
           7       1.00      1.00      1.00      1073
           8       1.00      1.00      1.00      1045
           9       1.00      1.00      1.00      1035

    accuracy                           1.00     10500
   macro avg       1.00      1.00      1.00     10500
weighted avg       1.00      1.00      1.00     10500
```

Apply the Random Forest Model to the Test Dataframe

In [14]:
```
# Create a copy of the training dataframe
rf_testing_x = digit_testing_data.copy(deep=True)

# Drop the label column from the copy of the training dataframe
#rf_testing_x.drop(['label'], axis=1, inplace=True)

# Apply the Random Forest model to the test dataset
y_test_predictions_rf = best_rf.predict(rf_testing_x)
```

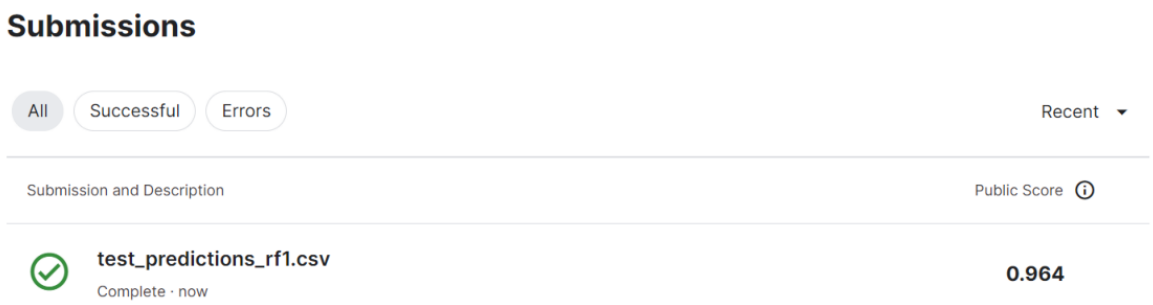
```
# Put the random forest predictions into a Pandas dataframe
prediction_df_rf = pd.DataFrame(y_test_predictions_rf, columns=['Label'])

# Add the ID column to the front of the random forest predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_rf.insert(0, 'ImageId', ImageId_series)

#output predictions to csv
#prediction_df_rf.to_csv('test_predictions_rf1.csv', index=False)
```

In [15]:
```
import matplotlib.pyplot as plt
# Display the kaggle results associated with the Random Forest Model
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Digit_Random_Forest1_Kaggle_Results_v1.png')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[15]:
```
<Figure size 1500x1500 with 0 Axes>
```

Out[15]:
```
<matplotlib.image.AxesImage at 0x1dbd069b070>
```

Out[15]:
```
(-0.5, 1455.5, 414.5, -0.5)
```




# Next, we scale the data to prepare it for our principal components analysis

In [7]:
```
# Scale PCA dataframe's data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
pca_scaled = sc.fit_transform(pca_df) # normalizing the features

# Convert scaled data from numpy array into dataframe
pca_features = list(pca_df.columns.values)
pca_scaled_df = pd.DataFrame(pca_scaled, columns=pca_features)

# Confirm scaling transformation was a success
pca_scaled_df.shape
pca_scaled_df.head(10)
pca_scaled_df.describe()
```

Out[7]:
```
(70000, 784)
```

Out[7]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | pixel774 | pixel77 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **1** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **2** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **3** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **4** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **5** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **6** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **7** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **8** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **9** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |

10 rows × 784 columns

Out[7]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | 70000.0 | ... | 7.( |
| **mean** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **std** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 1.( |
| **min** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **25%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **50%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **75%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **max** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 4.2 |

8 rows × 784 columns

We also apply this scaling to our test dataframe for later use as we progress through the construction of our Principal Component Analysis and Random Forest model creation processes.

In [8]:
```
# Apply the standard scaling to the test dataframe
pca_test_scaled = sc.transform(digit_testing_data)

# Convert scaled data from numpy array into dataframe
pca_test_features = list(digit_testing_data.columns.values)
pca_test_scaled_df = pd.DataFrame(pca_test_scaled, columns=pca_test_features)

# Confirm scaling transformation was a success
```

```
pca_test_scaled_df.shape
pca_test_scaled_df.head(10)
pca_test_scaled_df.describe()
```

Out[8]:
```
(28000, 784)
```

Out[8]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | pixel774 | pixel77 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **1** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **2** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **3** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **4** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **5** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **6** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **7** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **8** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |
| **9** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | -0.032951 | -0.02338 |

10 rows × 784 columns

Out[8]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | 28000.0 | ... | 28 |
| **mean** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **std** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **min** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **25%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **50%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **75%** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |
| **max** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | |

8 rows × 784 columns

Next, we will conduct a Principal Components Analysis to identify principal components that account for at least 95% of the variation in the data.

In [9]:
```
# Start a timer for the Principal Components Analysis
import datetime
```

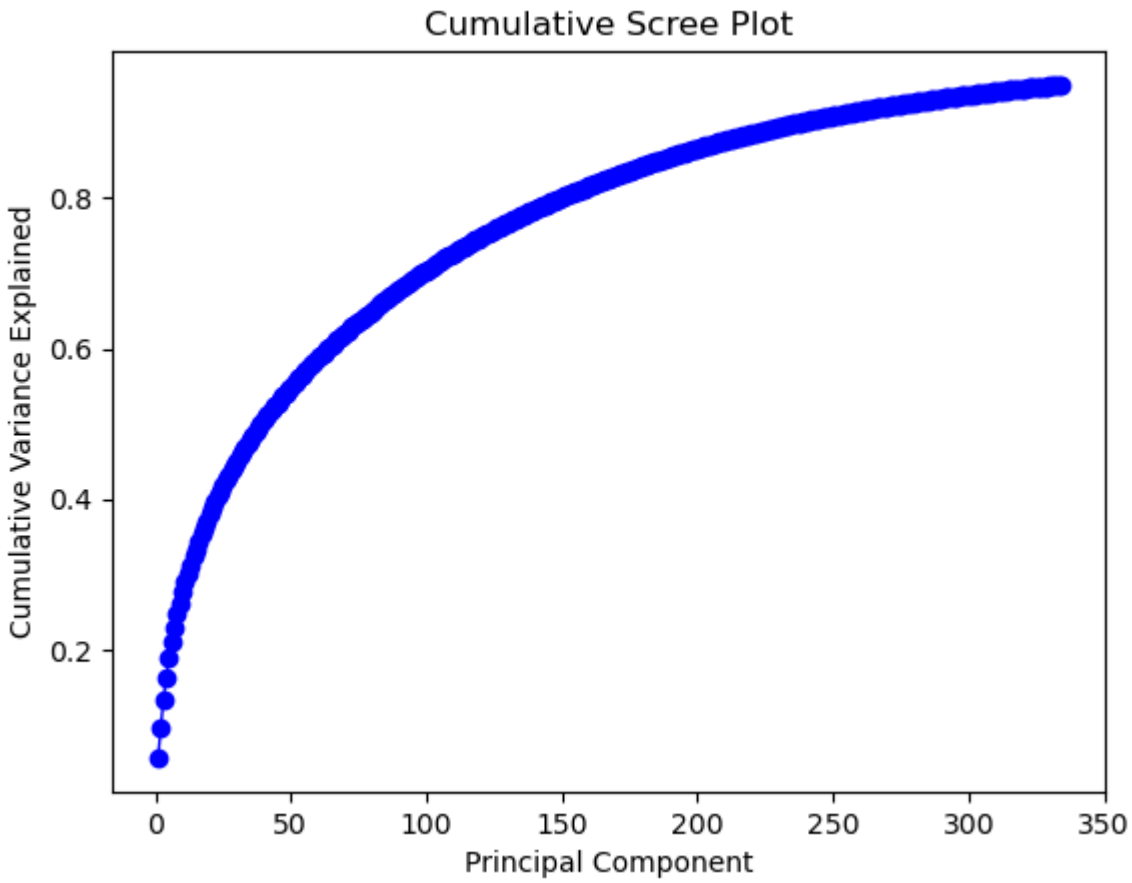
```
pca_start = datetime.datetime.now()

# Applying PCA function on training and testing set of X component
from sklearn.decomposition import PCA
pca_digits_train_test = PCA(n_components=334)
principal_components_digits = pca_digits_train_test.fit_transform(pca_scaled_df)


# Create a Cumulative Scree plot to help us determine how many principal components to
import matplotlib.pyplot as plt
import numpy as np

PC_values = np.arange(pca_digits_train_test.n_components_) + 1
cumulative_explained_variance_pca = np.cumsum(pca_digits_train_test.explained_variance

plt.plot(PC_values, cumulative_explained_variance_pca, 'o-', linewidth=1, color='blue'
plt.title('Cumulative Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Variance Explained')
plt.show()

# Create a dataframe to display the information in the cumulative scree plot in a diff
scree_df = pd.DataFrame({'Principal Component':PC_values, 'Variance Explained':cumulat
scree_df

# Create a dataframe that contains the principal component values for each of the obse
pca_column_list = []
for num in range(1, 335):
    pca_column_list.append("PC_" + str(num))

pca_digits_df = pd.DataFrame(data = principal_components_digits , columns = pca_columr

pca_digits_df


# Print the run time for Python to complete the Principal Components Analysis
pca_end = datetime.datetime.now()
pca_runtime = pca_end - pca_start
print(f"The total run time for the Principal Components Analysis was {pca_runtime}.")
```

Out[9]: [<matplotlib.lines.Line2D at 0x1dbd06a7280>]

Out[9]: Text(0.5, 1.0, 'Cumulative Scree Plot')

Out[9]: Text(0.5, 0, 'Principal Component')

Out[9]: Text(0, 0.5, 'Cumulative Variance Explained')




Out[9]:

| | Principal Component | Variance Explained |
|---|---|---|
| **0** | 1 | 0.056427 |
| **1** | 2 | 0.096839 |
| **2** | 3 | 0.134222 |
| **3** | 4 | 0.163152 |
| **4** | 5 | 0.188360 |
| **...** | ... | ... |
| **329** | 330 | 0.948805 |
| **330** | 331 | 0.949143 |
| **331** | 332 | 0.949480 |
| **332** | 333 | 0.949808 |
| **333** | 334 | 0.950135 |

334 rows × 2 columns

Out[9]:

| | PC_1 | PC_2 | PC_3 | PC_4 | PC_5 | PC_6 | PC_7 | PC_8 | PC_ |
|---|---|---|---|---|---|---|---|---|---|
| **0** | -5.230192 | -4.904646 | 4.175498 | -0.753746 | 4.991252 | 1.873491 | 4.739370 | -4.818814 | 0.20922 |
| **1** | 19.376064 | 5.924937 | 1.124527 | -2.236678 | 3.154725 | -1.899992 | -3.861523 | 0.291863 | -4.06420 |
| **2** | -7.675868 | -1.518335 | 2.369636 | 2.392773 | 4.809067 | -4.330499 | -0.993471 | 1.809950 | 0.31114 |
| **3** | -0.360917 | 5.988875 | 1.676212 | 4.312827 | 2.388172 | 2.129843 | 4.456385 | -0.344041 | 0.78353 |
| **4** | 26.628547 | 5.805648 | 0.833779 | -2.676026 | 9.565533 | -2.676311 | -6.303765 | -1.579776 | -4.07854 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **69995** | -1.099783 | 8.956724 | -2.928516 | -0.816439 | -5.882169 | -0.554970 | 2.339101 | -4.793652 | -2.05095 |
| **69996** | -3.590883 | 9.075696 | -5.882224 | 0.284067 | 2.110737 | -3.145613 | 7.328235 | 3.542143 | -3.87092 |
| **69997** | -2.978092 | 1.570972 | 5.616925 | -9.443330 | -0.177769 | -2.517486 | -1.220313 | 0.480684 | -2.48177 |
| **69998** | -3.978432 | 2.909071 | -3.836933 | -1.475158 | -6.942173 | -2.689870 | 1.417715 | -0.608694 | 2.04261 |
| **69999** | 8.792242 | -4.948765 | -1.306122 | 3.664094 | 0.247188 | 7.865749 | 3.700556 | -1.285305 | 0.41090 |

70000 rows × 334 columns

## Construct a Random Forest Model Using the Principal Components Identified

Let's fit a Random Forest Model to predict digits using the principal components just identified. We will use our training and validation datasets to conduct hyperparameter tuning to find the best hyperparameters for random forest modeling.

In [16]:
```
# Start a timer for the Random Forest

pca_rf_start = datetime.datetime.now()

# Create the Random Forest Model

# Import Required Modules
#pip install graphviz
#import pandas as pd
#import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from scipy.stats import randint
from sklearn.tree import export_graphviz
from IPython.display import Image
import graphviz

# Split the training dataset into predictor and outcome components
rf_train_validation_x = pca_digits_df.copy(deep=True)
rf_train_validation_x.drop(rf_train_validation_x.tail(28000).index, inplace = True)
rf_train_validation_y = digit_training_data['label']
```

```
# Split the Kaggle training data into training and validation components
rf_x_train, rf_x_validation, rf_y_train, rf_y_validation = train_test_split(rf_train_v
                                                                            rf_train_validat
                                                                                  test_size=
                                                                                random_stat


# Conduct hyperparameter tuning for random forest models
param_dist = {'n_estimators': randint(10,100),
              'max_depth': randint(1,100),
             'max_features': randint(1,20)}

rf = RandomForestClassifier()

rand_search = RandomizedSearchCV(rf,
                                 param_distributions = param_dist,
                                 n_iter=5,
                                 cv=5)

rand_search.fit(rf_x_train, rf_y_train)

# Create a variable for the best model
best_rf = rand_search.best_estimator_

# Print the best hyperparameters
print('Best hyperparameters:',  rand_search.best_params_)

# Print the run time for Python to complete the Random Forest
pca_rf_end = datetime.datetime.now()
pca_rf_runtime = pca_rf_end - pca_rf_start
print(f"The total run time for the Random Forest Model using the principal components
```

Out[16]:

```
RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(), n_iter=5,
                   param_distributions={'max_depth': <scipy.stats._distn_infrastructu
re.rv_discrete_frozen object at 0x000001DC14C42A90>,
                                        'max_features': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DC24AA5760>,
                                        'n_estimators': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DB92674310>})
Best hyperparameters: {'max_depth': 77, 'max_features': 18, 'n_estimators': 96}
The total run time for the Random Forest Model using the principal components was 0:1
0:11.179878.
```

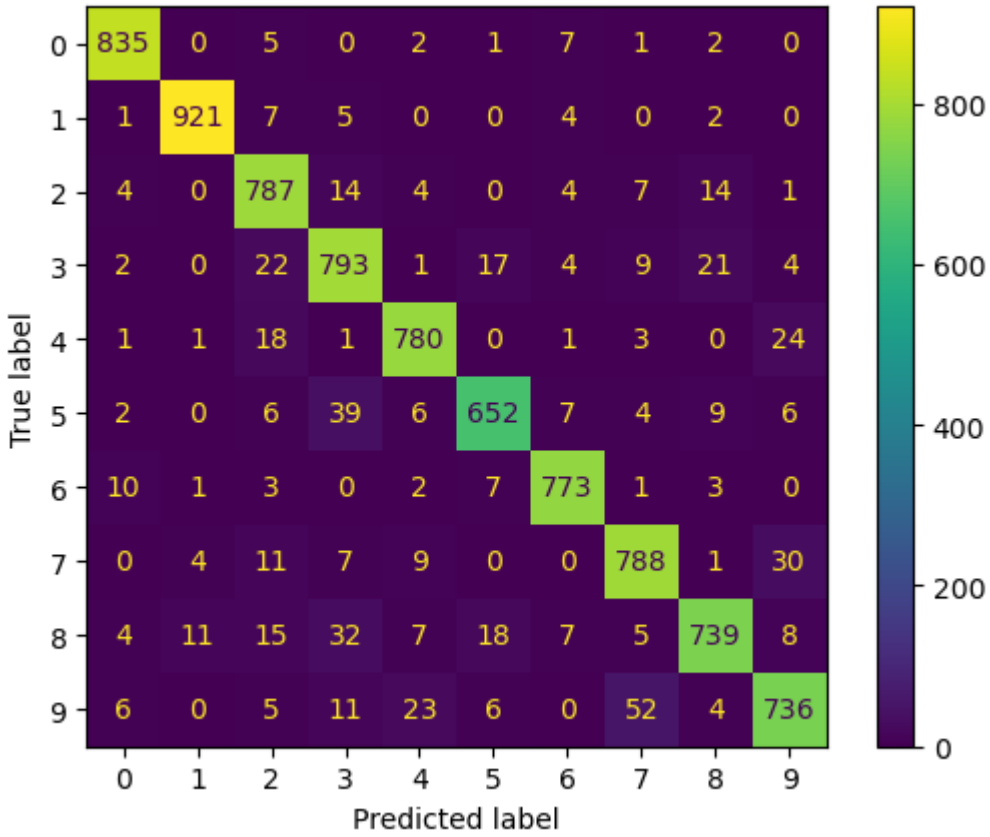
Next, we will assess the strength of the random forest model associated with the optimal hyperparameters by applying the model to the validation dataset and observing the resulting confusion matrix and accuracy.

In [17]:

```
# Generate predictions with the best model
y_validation_predictions_rf = best_rf.predict(rf_x_validation)

# Create the confusion matrix associated with the best random forest model
cm = confusion_matrix(rf_y_validation, y_validation_predictions_rf)

ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Calculate the accuracy, precision, and recall associated with the predictions of the

accuracy_rf_validation = accuracy_score(rf_y_validation, y_validation_predictions_rf)
#precision_rf_validation = precision_score(rf_y_validation, y_validation_predictions_r
#recall_rf_validation = recall_score(rf_y_validation, y_validation_predictions_rf)
```

```
print("Accuracy:", accuracy_rf_validation)
#print("Precision:", precision_rf_validation)
#print("Recall:", recall_rf_validation)
```

Out[17]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1dbd06a7250>

Accuracy: 0.929047619047619
```




Apply the Random Forest Model to the Test Dataframe

In [18]:
```
# Create a dataframe for predictor variables in the test dataframe for random forest m
#rf_testing_x = rf_testing_df.drop(columns=['PassengerId'])
rf_testing_x = pca_digits_df.copy(deep=True)
rf_testing_x.drop(rf_testing_x.head(42000).index, inplace = True)

# Apply the Random Forest model to the test dataset
y_test_predictions_rf = best_rf.predict(rf_testing_x)

# Put the random forest predictions into a Pandas dataframe
prediction_df_rf = pd.DataFrame(y_test_predictions_rf, columns=['Label'])

# Add the ID column to the front of the random forest predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_rf.insert(0, 'ImageId', ImageId_series)

#output predictions to csv
#prediction_df_rf.to_csv('test_predictions_pca_random_forest_v1.csv', index=False)
```

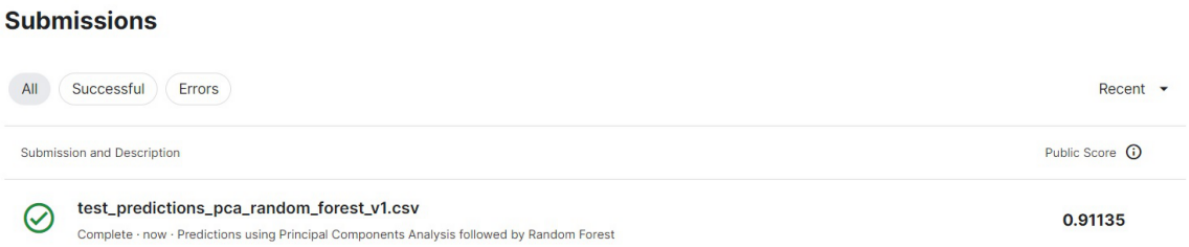
Let's display the Kaggle results from the application of the random forest model using principal components to the test dataset

In [19]:
```
# Display the kaggle results associated with the Random Forest Model
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Digit_PCA_Random_Forest_Kaggle_Results_v1.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[19]:
```
<Figure size 1500x1500 with 0 Axes>
```

Out[19]:
```
<matplotlib.image.AxesImage at 0x1dc14cc38e0>
```

Out[19]:
```
(-0.5, 1502.5, 339.5, -0.5)
```




## Construct a Random Forest Model Using the Principal Components Identified and the Original Data

Let's fit a Random Forest Model to predict digits using the principal components and the original underlying data. We will use our training and validation datasets to conduct hyperparameter tuning to find the best hyperparameters for random forest modeling.

In [20]:
```
# Start a timer for the Random Forest

pca_rf_v2_start = datetime.datetime.now()



# Split the training dataset into predictor and outcome components
rf_train_validation_x = pca_digits_df.copy(deep=True)
rf_train_validation_x.drop(rf_train_validation_x.tail(28000).index, inplace = True)
rf_train_validation_x = pd.concat([rf_train_validation_x, pca_train_df], axis=1)
rf_train_validation_y = digit_training_data['label']

# Split the Kaggle training data into training and validation components
rf_x_train, rf_x_validation, rf_y_train, rf_y_validation = train_test_split(rf_train_v
                                                                            rf_train_validat
                                                                                  test_size=
                                                                                 random_stat


# Conduct hyperparameter tuning for random forest models
param_dist = {'n_estimators': randint(10,100),
              'max_depth': randint(1,100),
             'max_features': randint(1,20)}

rf = RandomForestClassifier()

rand_search = RandomizedSearchCV(rf,
                                 param_distributions = param_dist,
```

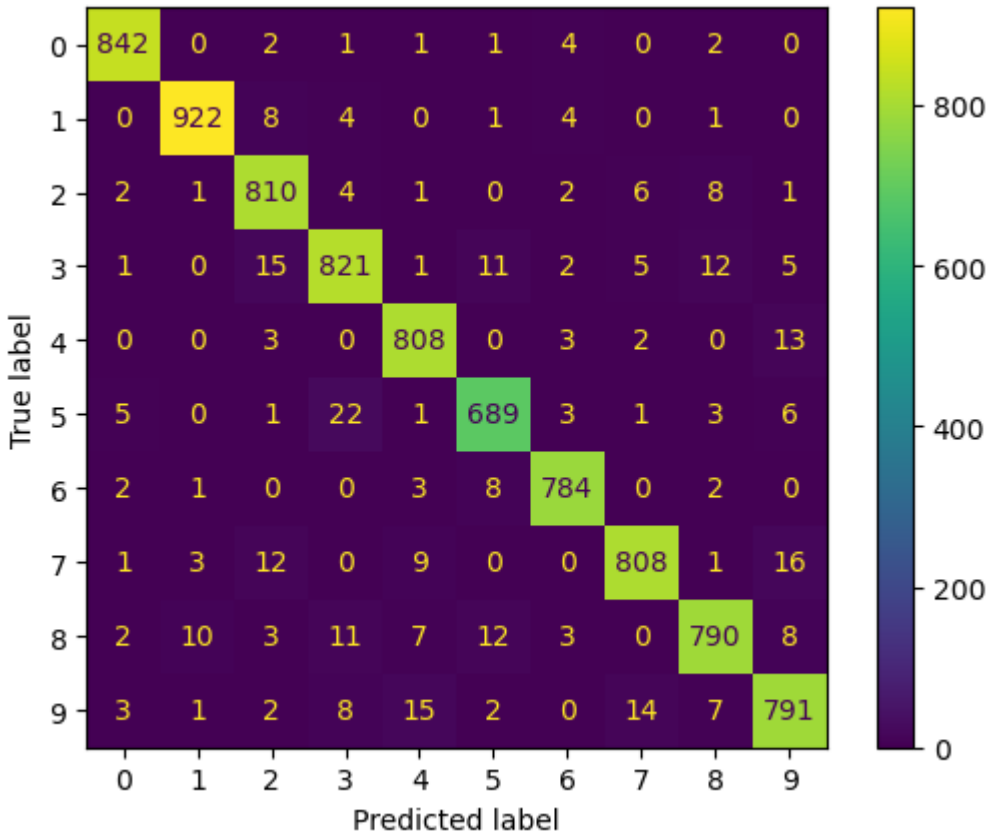
```
                                   n_iter=5,
                                   cv=5)

rand_search.fit(rf_x_train, rf_y_train)

# Create a variable for the best model
best_rf = rand_search.best_estimator_

# Print the best hyperparameters
print('Best hyperparameters:',  rand_search.best_params_)


# Print the run time for Python to complete the Random Forest
pca_rf_v2_end = datetime.datetime.now()
pca_rf_v2_runtime = pca_rf_v2_end - pca_rf_v2_start
print(f"The total run time for the Random Forest Model using the principal components
```

Out[20]:
```
RandomizedSearchCV(cv=5, estimator=RandomForestClassifier(), n_iter=5,
                   param_distributions={'max_depth': <scipy.stats._distn_infrastructu
re.rv_discrete_frozen object at 0x000001DC14CD3FD0>,
                                        'max_features': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DC14CB0DF0>,
                                        'n_estimators': <scipy.stats._distn_infrastru
cture.rv_discrete_frozen object at 0x000001DC14C42130>})
Best hyperparameters: {'max_depth': 40, 'max_features': 18, 'n_estimators': 83}
The total run time for the Random Forest Model using the principal components and ori
ginal pixel features was 0:02:45.850764.
```

Next, we will assess the strength of the random forest model associated with the optimal hyperparameters by applying the model to the validation dataset and observing the resulting confusion matrix and accuracy.

In [21]:
```
# Generate predictions with the best model
y_validation_predictions_rf = best_rf.predict(rf_x_validation)

# Create the confusion matrix associated with the best random forest model
cm = confusion_matrix(rf_y_validation, y_validation_predictions_rf)

ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Calculate the accuracy, precision, and recall associated with the predictions of the

accuracy_rf_validation = accuracy_score(rf_y_validation, y_validation_predictions_rf)
#precision_rf_validation = precision_score(rf_y_validation, y_validation_predictions_r
#recall_rf_validation = recall_score(rf_y_validation, y_validation_predictions_rf)

print("Accuracy:", accuracy_rf_validation)
#print("Precision:", precision_rf_validation)
#print("Recall:", recall_rf_validation)
```

Out[21]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1dbd08462b0>

Accuracy: 0.9601190476190476
```




Apply the Random Forest Model to the Test Dataframe

In [22]:
```
# Create a dataframe for predictor variables in the test dataframe for random forest m
rf_testing_x = pca_digits_df.copy(deep=True)
rf_testing_x.drop(rf_testing_x.head(42000).index, inplace = True)
rf_testing_x.reset_index(drop=True, inplace=True)
digit_testing_data.reset_index(drop=True, inplace=True)
rf_testing_x = pd.concat([rf_testing_x, digit_testing_data], axis=1)

# Apply the Random Forest model to the test dataset
y_test_predictions_rf = best_rf.predict(rf_testing_x)

# Put the random forest predictions into a Pandas dataframe
prediction_df_rf = pd.DataFrame(y_test_predictions_rf, columns=['Label'])

# Add the ID column to the front of the random forest predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_rf.insert(0, 'ImageId', ImageId_series)

#output predictions to csv
#prediction_df_rf.to_csv('test_predictions_pca_random_forest_v2.csv', index=False)
```

Let's display the Kaggle results from the application of the random forest model using principal components and the original underlying data features to the test dataset.

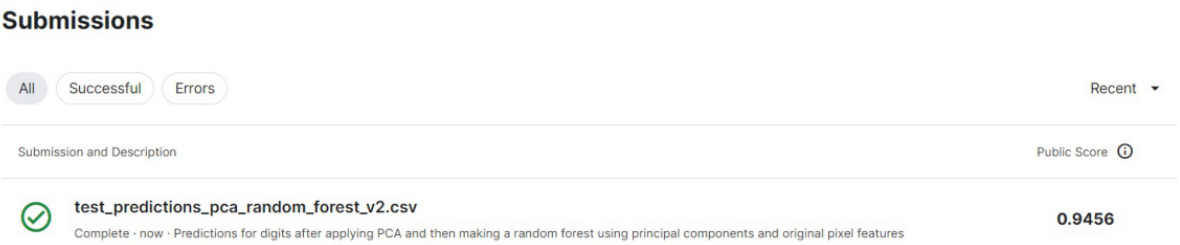
In [23]:
```
# Display the kaggle results associated with the Random Forest Model
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Digit_PCA_And_Original_Features_Random_Forest_Kaggle_Resu
plt.imshow(kaggle_results)
```

```
plt.axis("off")
plt.show()
```

Out[23]:
```
<Figure size 1500x1500 with 0 Axes>
```

Out[23]:
```
<matplotlib.image.AxesImage at 0x1dc15044100>
```

Out[23]:
```
(-0.5, 1510.5, 338.5, -0.5)
```




In [24]:
```
# mitigate design flaw
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

sc = StandardScaler()
train = digit_training_data.drop(columns = 'label')
train_label = digit_training_data['label']
scaled_train = sc.fit_transform(train)

pca = PCA(n_components=334)
pca_train = pca.fit_transform(scaled_train)

# Split the Kaggle training data into training and validation components
rf_x_train, rf_x_validation, rf_y_train, rf_y_validation = train_test_split(pca_train,

rf = RandomForestClassifier()
rf.fit(rf_x_train, rf_y_train)
predictions = rf.predict(rf_x_validation)
```

Out[24]:
```
RandomForestClassifier()
```

## Deploy K-Means Clustering

Let's use K-means clustering to predict digits using original features. First let's create our training and testing data and plot the digits in the dataset

In [25]:
```
import sys
import sklearn
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Split the training dataset into predictor and outcome variables
kmeans_x_train = digit_training_data.copy(deep=True)
kmeans_x_train.drop(['label'], axis=1, inplace=True)
kmeans_y_train = digit_training_data['label']
```

```
kmeans_x_train = np.array(kmeans_x_train)
kmeans_y_train = np.array(kmeans_y_train)


print('Training Data: {}'.format(kmeans_x_train.shape))
print('Training Labels: {}'.format(kmeans_y_train.shape))

# reshape array to 3-dimensional array so we can plot the numbers
kmeans_x_train_plot = kmeans_x_train.reshape(42000, 28, 28)

# Plot the digits in the dataset
fig, axs = plt.subplots(3, 3, figsize = (12, 12))
plt.gray()

for i, ax in enumerate(axs.flat):
    ax.matshow(kmeans_x_train_plot[i])
    ax.axis('off')
    ax.set_title('Number {}'.format(kmeans_y_train[i]))
    fig.show()
```

```
Training Data: (42000, 784)
Training Labels: (42000,)
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc1598d070>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 1')
```

```
C:\Users\mhurt\AppData\Local\Temp\ipykernel_22820\2378760689.py:32: UserWarning: Matp
lotlib is currently using module://matplotlib_inline.backend_inline, which is a non-G
UI backend, so cannot show the figure.
  fig.show()
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159e9310>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 0')
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159e9340>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 1')
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159e99d0>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 4')
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159e98e0>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 0')
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159fd220>
```

Out[25]:
```
(-0.5, 27.5, 27.5, -0.5)
```

Out[25]:
```
Text(0.5, 1.0, 'Number 0')
```

Out[25]:
```
<matplotlib.image.AxesImage at 0x1dc159fd490>
```

Out[25]: (-0.5, 27.5, 27.5, -0.5)

Out[25]: Text(0.5, 1.0, 'Number 7')

Out[25]: <matplotlib.image.AxesImage at 0x1dc159fd4f0>

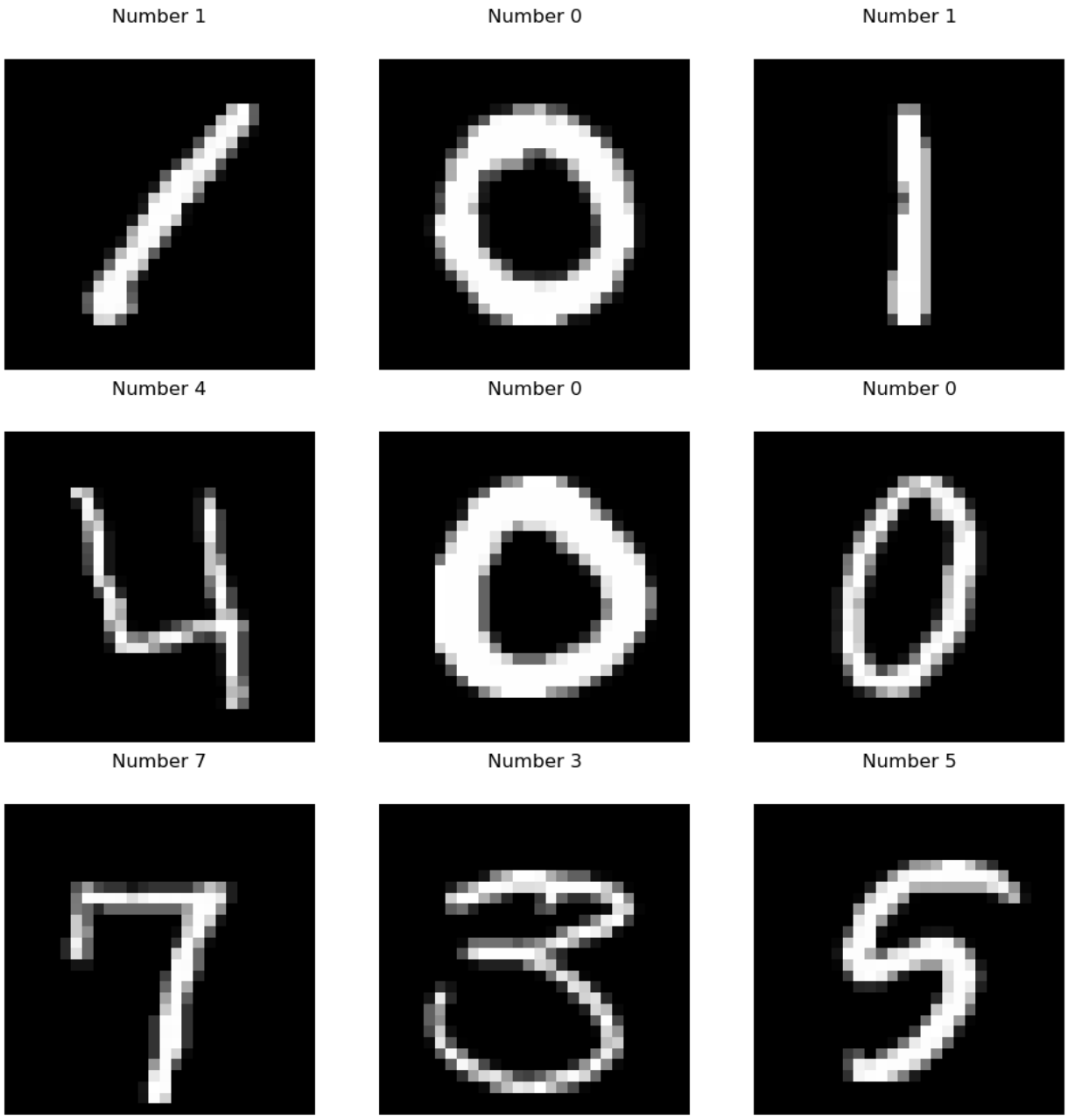
Out[25]: (-0.5, 27.5, 27.5, -0.5)

Out[25]: Text(0.5, 1.0, 'Number 3')

Out[25]: <matplotlib.image.AxesImage at 0x1dc159fdb20>

Out[25]: (-0.5, 27.5, 27.5, -0.5)

Out[25]: Text(0.5, 1.0, 'Number 5')




Normalize the training data before applying k-means clustering

In [26]:
```
from sklearn import preprocessing
kmeans_x_train_norm = preprocessing.normalize(kmeans_x_train)
```

The MNIST dataset contains images of the integers 0 to 9. Because of this, let's start by setting the number of clusters to 10, one for each digit

Compute the silhouette coefficients kmeans models with different numbers of clusters. This can vary between –1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary; finally, a coefficient close to –1 means that the instance may have been assigned to the wrong cluster.

reference: Geron, Aurelien. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 2nd ed. Sebastopol, CA: O'Reilly.

In [27]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples
from sklearn.cluster import MiniBatchKMeans
# minibatchkmeans has a memory leak warning that we can ignore
import warnings
warnings.filterwarnings('ignore')

# create k-means models with K clusters.
K = clusters=[10,16,36,64,144,256,400] # test listed cluster numbers

# Store within-cluster-sum of squares and silhouette scores for clusters
wss = []
sil_score = []

# loop though cluster values and save inertia and silhouttee values
for i in K:
    kmeans=MiniBatchKMeans(n_clusters=i, random_state=1)
    kmeans=kmeans.fit(kmeans_x_train_norm)
    # within-cluster-sum-squares
    wss_iter = kmeans.inertia_
    wss.append(wss_iter)
    # silhouttee score
    score = silhouette_score(kmeans_x_train_norm, kmeans.labels_)
    sil_score.append(score)
    print ("Silhouette score for k(clusters) = "+str(i)+" is "+str(score))
```

```
Silhouette score for k(clusters) = 10 is 0.08523799414147828
Silhouette score for k(clusters) = 16 is 0.07839942003565271
Silhouette score for k(clusters) = 36 is 0.07031085560863824
Silhouette score for k(clusters) = 64 is 0.058089283987564244
Silhouette score for k(clusters) = 144 is 0.05081383335242342
Silhouette score for k(clusters) = 256 is 0.04716730485359939
Silhouette score for k(clusters) = 400 is 0.04189046249619273
```

In [28]:
```
import seaborn as sns
# elbow and silhouttee scores in dataframe with number of clusters
cluster_sil_scores = pd.DataFrame({'Clusters' : K, 'WSS' : wss, 'Sil Score' : sil_scor
cluster_sil_scores
```

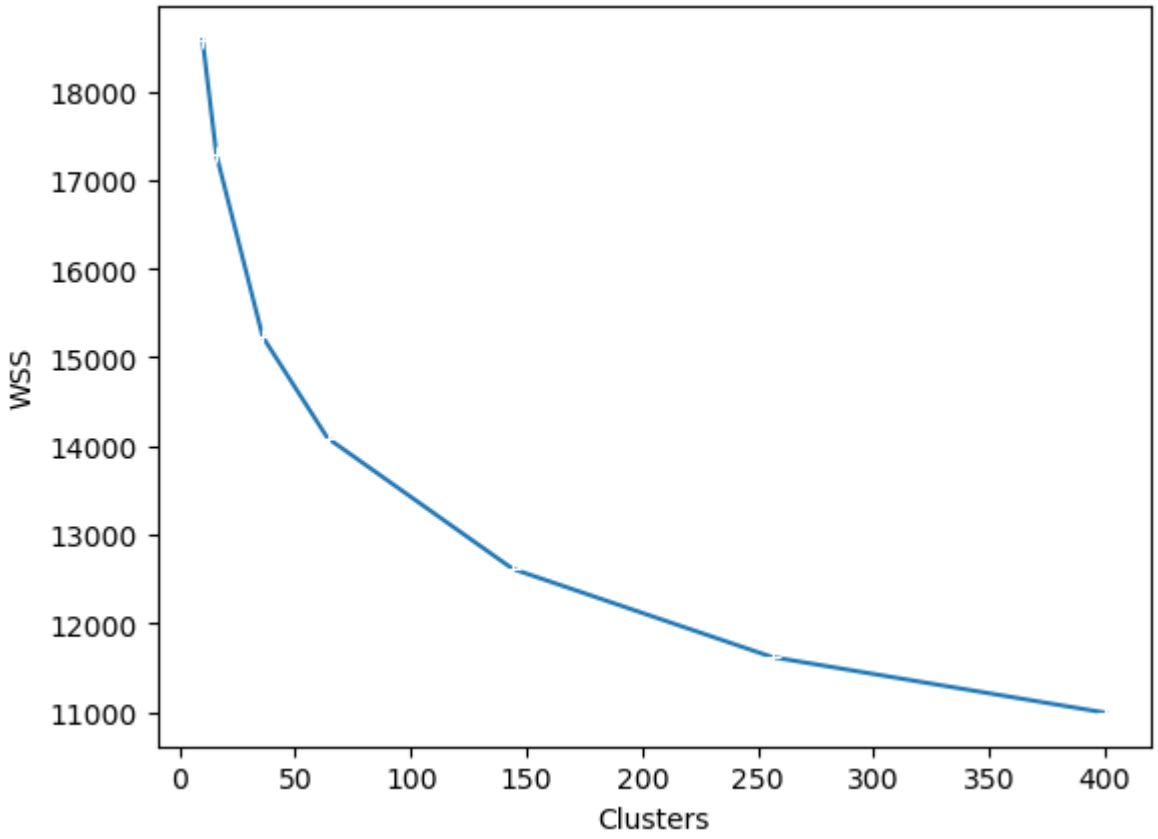
```
# plot the elbow scores
sns.lineplot(x = 'Clusters', y = 'WSS', data = cluster_sil_scores, marker="+")
```

Out[28]:

| | Clusters | WSS | Sil Score |
|---|---|---|---|
| **0** | 10 | 18579.941430 | 0.085238 |
| **1** | 16 | 17282.276208 | 0.078399 |
| **2** | 36 | 15230.436883 | 0.070311 |
| **3** | 64 | 14089.445126 | 0.058089 |
| **4** | 144 | 12613.518954 | 0.050814 |
| **5** | 256 | 11619.579117 | 0.047167 |
| **6** | 400 | 10991.675080 | 0.041890 |

Out[28]:

```
<AxesSubplot:xlabel='Clusters', ylabel='WSS'>
```




Based on the elbow plot, the inertia drops very quickly as we increase k up to 50, but then it decreases a bit more slowly as we keep increasing k. This curve has a distinct elbow shape, we also a more gradual decline around 250.

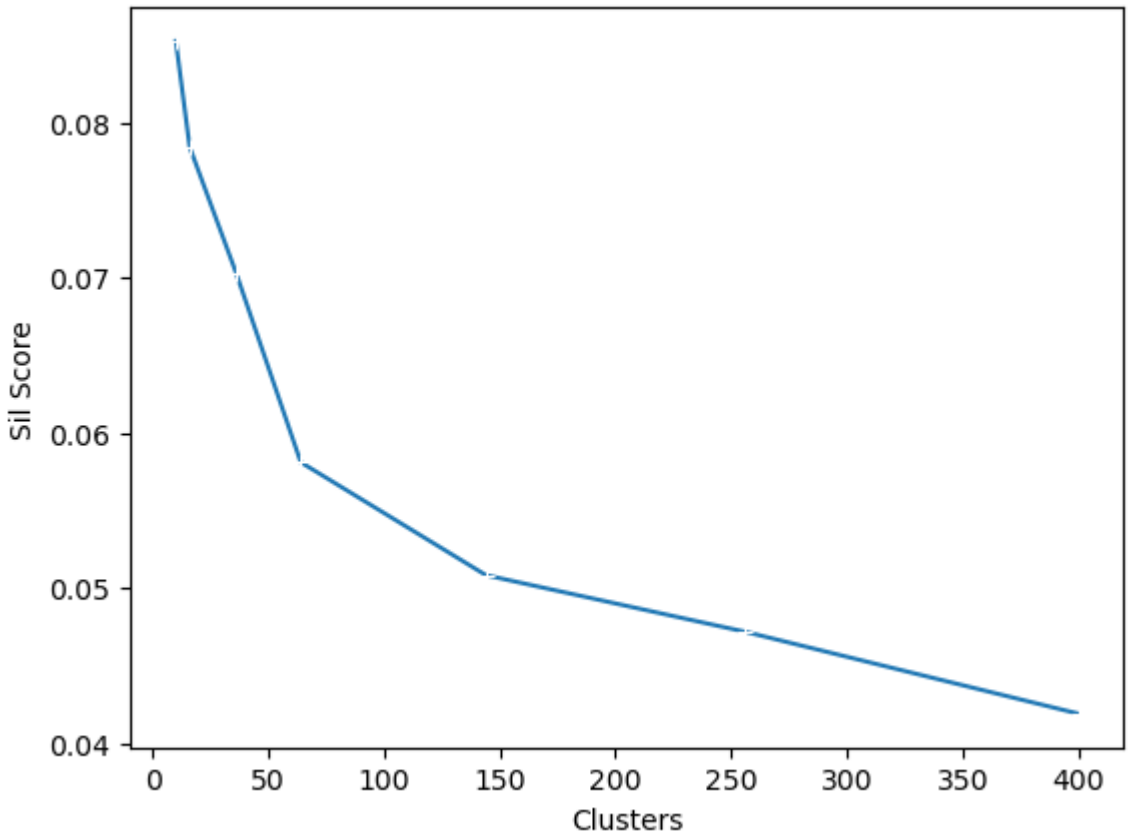
This indicates that 144 and 256 could be optimal cluster numbers.

In [29]:

```
# plot the silhouttee scores
sns.lineplot(x = 'Clusters', y = 'Sil Score', data = cluster_sil_scores, marker="+")
```

Out[29]:

```
<AxesSubplot:xlabel='Clusters', ylabel='Sil Score'>
```




Based on the plot, silhouette scores decline as the number of clusters increases. Scores close to 0 suggest that the clusters are overlapping, and the model with more clusters may not able to distinguish them well.

This isn't what we observe with the inertia plot, so we will still test models with 144 and 256 clusters. We also know there are 10 digits that are represented in the dataset so this could also be an optimal cluster number. We will build three models using these cluster numbers and compare performance metrics.

K-means clustering is an unsupervised machine learning method so the labels assigned by our KMeans algorithm refer to the cluster each array was assigned to, not the actual target integer. This section defines functions that predict which integer corresponds to each cluster. reference: https://medium.datadriveninvestor.com/k-means-clustering-for-imagery-analysis-56c9976f16b6#:~:text=Preprocessing

In [30]:
```
def infer_cluster_labels(kmeans, actual_labels):
    inferred_labels = {}

    for i in range(kmeans.n_clusters):

        # find index of points in cluster
        labels = []
        index = np.where(kmeans.labels_ == i)

        # append actual labels for each point in cluster
        labels.append(actual_labels[index])
```

```
        # determine most common label
        if len(labels[0]) == 1:
            counts = np.bincount(labels[0])
        else:
            counts = np.bincount(np.squeeze(labels))

        # assign the cluster to a value in the inferred_labels dictionary
        if np.argmax(counts) in inferred_labels:
            # append the new number to the existing array at this slot
            inferred_labels[np.argmax(counts)].append(i)
        else:
            # create a new array in this slot
            inferred_labels[np.argmax(counts)] = [i]

        #print(labels)
        #print('Cluster: {}, label: {}'.format(i, np.argmax(counts)))

    return inferred_labels

def infer_data_labels(X_labels, cluster_labels):
  # empty array of len(X)
    predicted_labels = np.zeros(len(X_labels)).astype(np.uint8)

    for i, cluster in enumerate(X_labels):
        for key, value in cluster_labels.items():
            if cluster in value:
                predicted_labels[i] = key

    return predicted_labels
```

Let's build models with 10, 144, and 256 clusters based on our knowledge of the data and the elbow and silhouette plot analysis.

In [31]:

```
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn import metrics

############ Initialize KMeans model with 10 clusters ##############
# Initialize KMeans model
kmeans = MiniBatchKMeans(n_clusters = 10, random_state=1)

# Fit the model to the training data
kmeans.fit(kmeans_x_train_norm)

# Predict the cluster assignment
X_clusters = kmeans.predict(kmeans_x_train_norm)
print(X_clusters[:20])

# predict labels for kmeans model with 10 clusters
cluster_labels=infer_cluster_labels(kmeans,kmeans_y_train)
predicted_labels = infer_data_labels(X_clusters, cluster_labels)

# print first 20 predicted labels and actual y-values
print(predicted_labels[:20])
print(kmeans_y_train[:20])

# Create the confusion matrix
cm = confusion_matrix(kmeans_y_train, predicted_labels)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
# Calculate the accuracy, inertia, and homogeneity scores
accuracy_kmeans = accuracy_score(kmeans_y_train, predicted_labels)
inertia_kmeans = kmeans.inertia_
homogeneity_kmeans = metrics.homogeneity_score(kmeans_y_train,predicted_labels)
print("Accuracy of K=10:", accuracy_kmeans)
print("Inertia of K=10:", inertia_kmeans)
print("Homogeneity of K=10:", homogeneity_kmeans)

########### Initialize KMeans model with 144 clusters ##############
kmeans = MiniBatchKMeans(n_clusters = 144, random_state=1)

# Fit the model to the training data
kmeans.fit(kmeans_x_train_norm)

# Predict the cluster assignment
X_clusters = kmeans.predict(kmeans_x_train_norm)
print(X_clusters[:20])

# predict labels for kmeans model with 144 clusters
cluster_labels=infer_cluster_labels(kmeans,kmeans_y_train)
predicted_labels = infer_data_labels(X_clusters, cluster_labels)

# print first 20 predicted labels and actual y-values
print(predicted_labels[:20])
print(kmeans_y_train[:20])

# Create the confusion matrix
cm = confusion_matrix(kmeans_y_train, predicted_labels)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Calculate the accuracy scores
accuracy_kmeans = accuracy_score(kmeans_y_train, predicted_labels)
inertia_kmeans = kmeans.inertia_
homogeneity_kmeans = metrics.homogeneity_score(kmeans_y_train,predicted_labels)
print("Accuracy of K=144:", accuracy_kmeans)
print("Inertia of K=144:", inertia_kmeans)
print("Homogeneity of K=144:", homogeneity_kmeans)


########### Initialize KMeans model with 256 clusters ##############
# Initialize KMeans model
kmeans = MiniBatchKMeans(n_clusters = 256, random_state=1)

# Fit the model to the training data
kmeans.fit(kmeans_x_train_norm)

# Predict the cluster assignment
X_clusters = kmeans.predict(kmeans_x_train_norm)
print(X_clusters[:20])

# predict labels for kmeans model with 256 clusters
cluster_labels = infer_cluster_labels(kmeans,kmeans_y_train)
predicted_labels = infer_data_labels(X_clusters, cluster_labels)

# print first 20 predicted labels and actual y-values
print(predicted_labels[:20])
print(kmeans_y_train[:20])

# Create the confusion matrix
cm = confusion_matrix(kmeans_y_train, predicted_labels)
```

```
ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Calculate the accuracy scores
accuracy_kmeans = accuracy_score(kmeans_y_train, predicted_labels)
inertia_kmeans = kmeans.inertia_
homogeneity_kmeans = metrics.homogeneity_score(kmeans_y_train,predicted_labels)
print("Accuracy of K=256:", accuracy_kmeans)
print("Inertia of K=256:", inertia_kmeans)
print("Homogeneity of K=256:", homogeneity_kmeans)
```

Out[31]:
```
MiniBatchKMeans(n_clusters=10, random_state=1)
```

```
[5 8 3 0 8 8 2 4 4 4 1 9 3 4 4 3 4 8 9 4]
[1 0 1 4 0 0 9 3 3 3 8 7 1 3 3 1 3 0 7 3]
[1 0 1 4 0 0 7 3 5 3 8 9 1 3 3 1 2 0 7 5]
```

Out[31]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1dc153cb280>
```

```
Accuracy of K=10: 0.5937619047619047
Inertia of K=10: 18579.941429667917
Homogeneity of K=10: 0.5069942742752722
```

Out[31]:
```
MiniBatchKMeans(n_clusters=144, random_state=1)
```

```
[125  48  46  91  22  43  32  93 122  80  63 112  44  93 129  31  81 134
  69 102]
[1 0 1 4 0 0 7 3 5 3 8 9 1 3 3 1 2 0 7 8]
[1 0 1 4 0 0 7 3 5 3 8 9 1 3 3 1 2 0 7 5]
```

Out[31]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1dc15af1520>
```

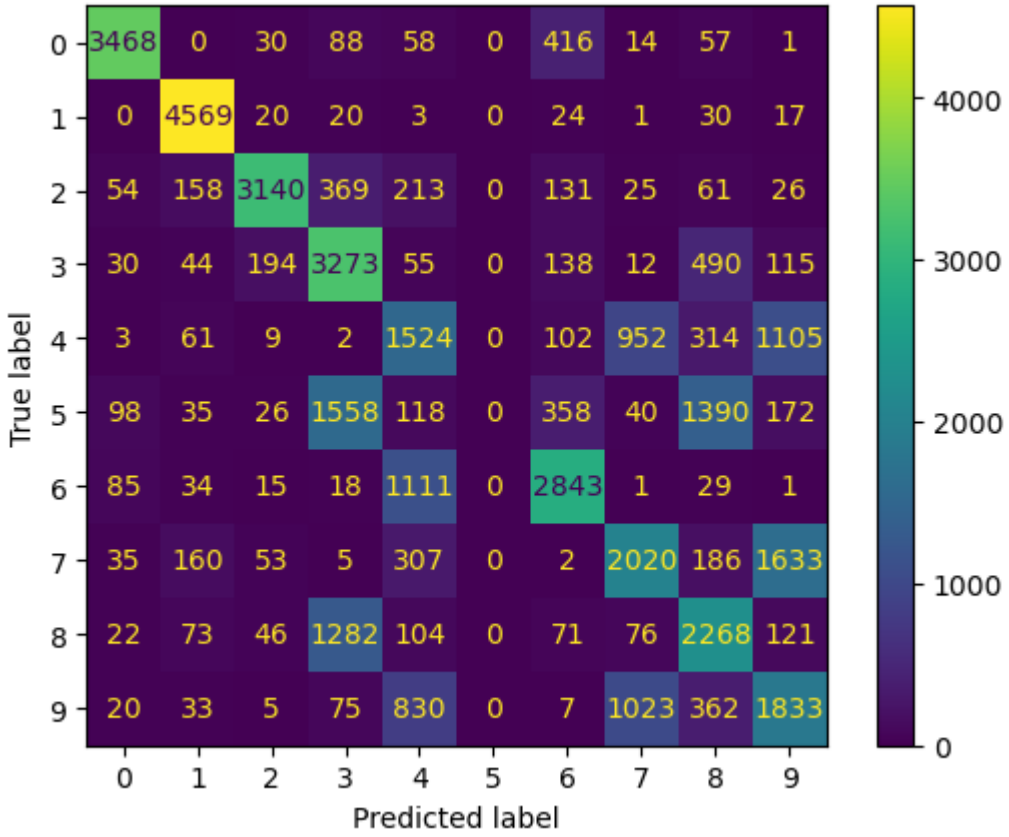
```
Accuracy of K=144: 0.8812142857142857
Inertia of K=144: 12613.518954331545
Homogeneity of K=144: 0.7806143306123215
```

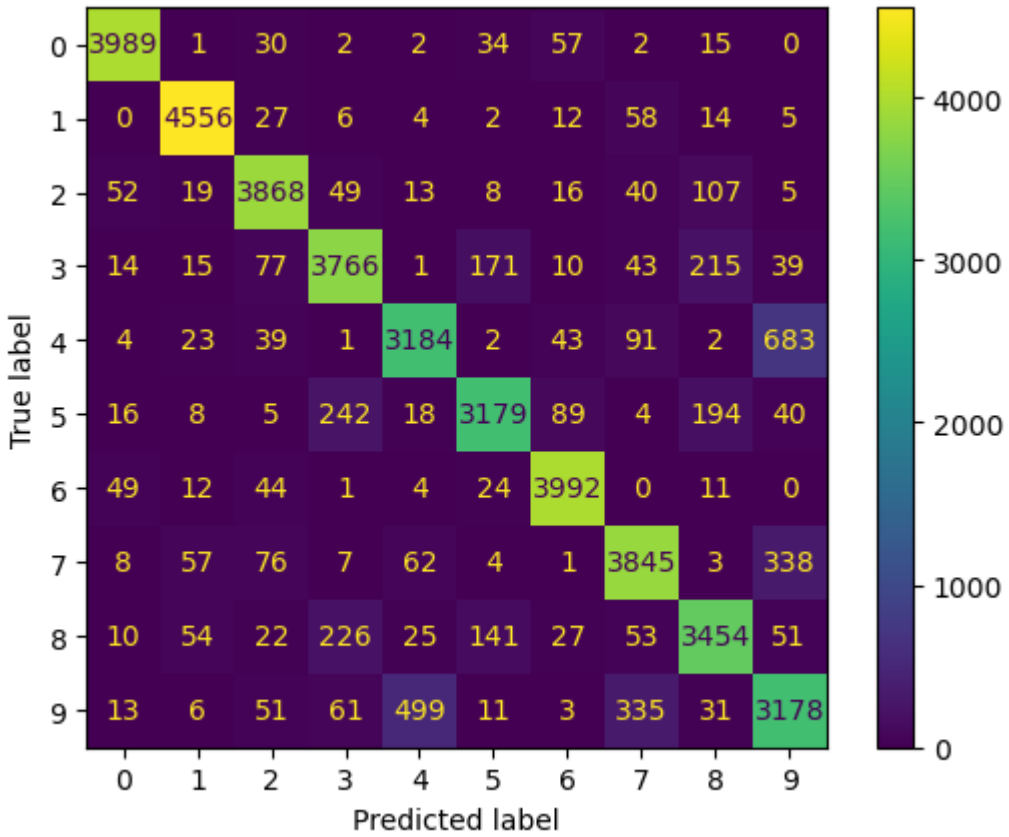
Out[31]:
```
MiniBatchKMeans(n_clusters=256, random_state=1)
```

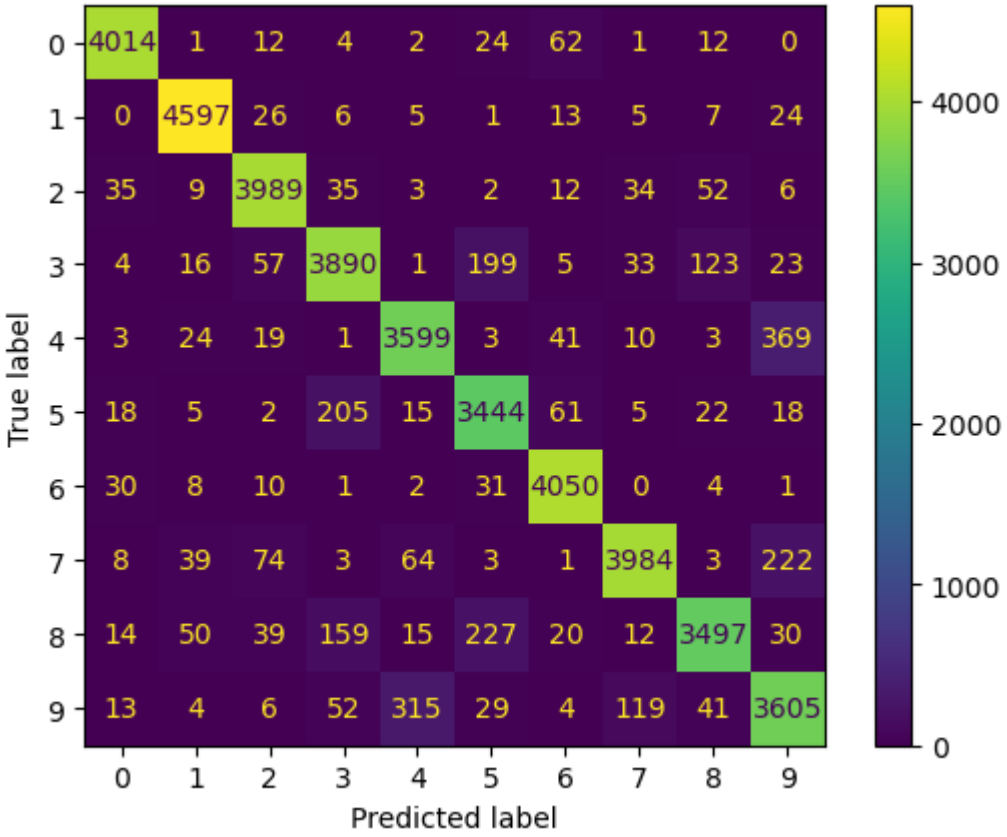
```
[147 143 226  46 143   7  37 253  25  60 167 146 226  65  24   1  21 250
  92 126]
[1 0 1 4 0 0 7 5 5 3 8 9 1 3 3 1 2 0 7 5]
[1 0 1 4 0 0 7 3 5 3 8 9 1 3 3 1 2 0 7 5]
```

Out[31]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1dc176d0370>
```

```
Accuracy of K=256: 0.9206904761904762
Inertia of K=256: 11619.579117068391
Homogeneity of K=256: 0.8374197295074177
```


0 3468 0 30 88 58 0 416 14 57 1
1 0 4569 20 20 3 0 24 1 30 17
2 54 158 3140 369 213 0 131 25 61 26
3 30 44 194 3273 55 0 138 12 490 115
4 3 61 9 2 1524 0 102 952 314 1105
5 98 35 26 1558 118 0 358 40 1390 172
6 85 34 15 18 1111 0 2843 1 29 1
7 35 160 53 5 307 0 2 2020 186 1633
8 22 73 46 1282 104 0 71 76 2268 121
9 20 33 5 75 830 0 7 1023 362 1833
0 1 2 3 4 5 6 7 8 9
True label
Predicted label
4000
3000
2000
1000
0


0 3989 1 30 2 2 34 57 2 15 0
1 0 4556 27 6 4 2 12 58 14 5
2 52 19 3868 49 13 8 16 40 107 5
3 14 15 77 3766 1 171 10 43 215 39
4 4 23 39 1 3184 2 43 91 2 683
5 16 8 5 242 18 3179 89 4 194 40
6 49 12 44 1 4 24 3992 0 11 0
7 8 57 76 7 62 4 1 3845 3 338
8 10 54 22 226 25 141 27 53 3454 51
9 13 6 51 61 499 11 3 335 31 3178
0 1 2 3 4 5 6 7 8 9
True label
Predicted label
4000
3000
2000
1000
0




We observe accuracy scores of

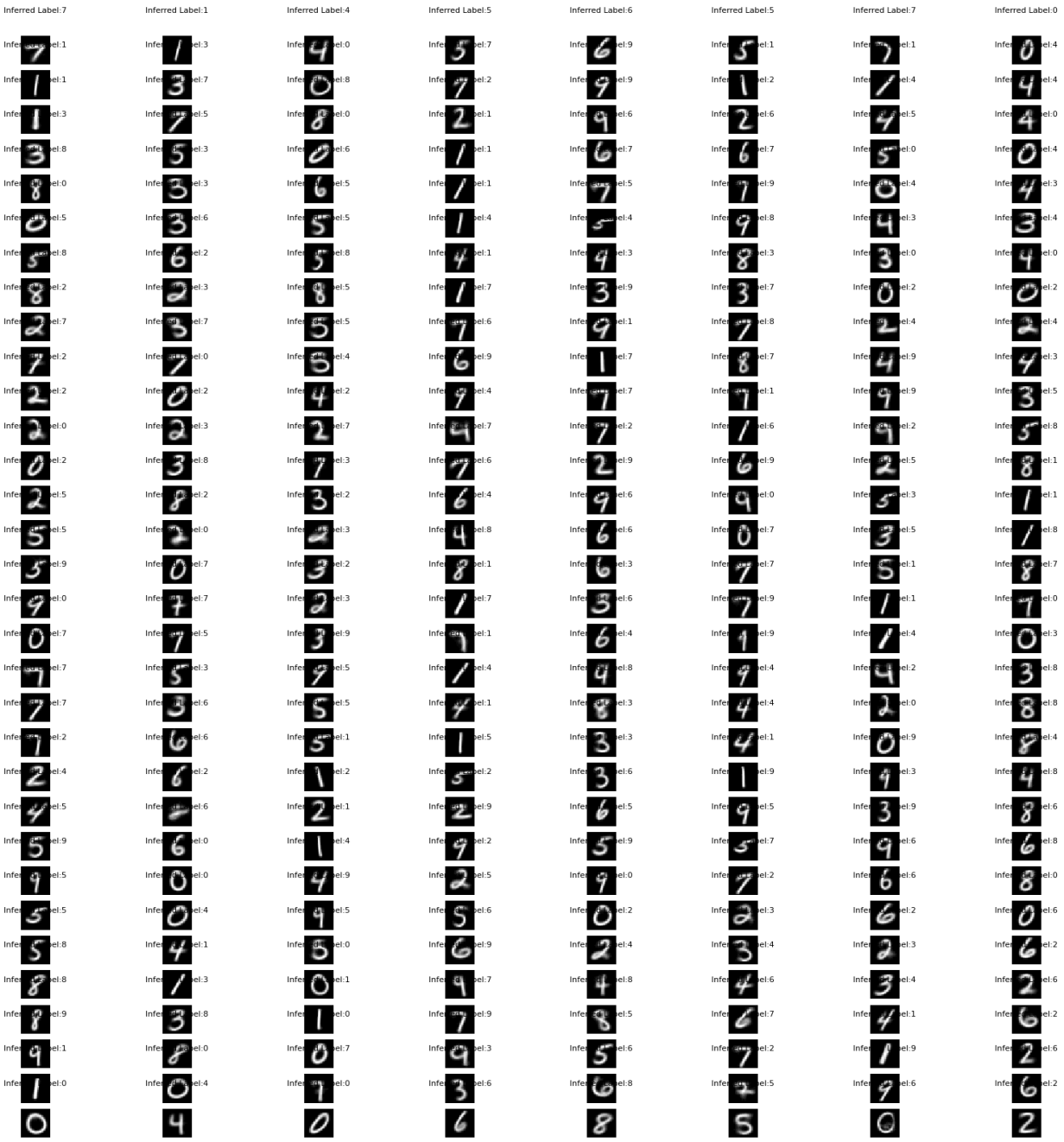
- 0.594 for the k-means model with 10 clusters
- 0.881 for the k-means model with 144 clusters
- 0.921 for the k-means model with 256 clusters.

Visualizing Cluster Centroids

Let's display the most representative image for each cluster.

In [32]:
```
# Initialize KMeans model with 256 clusters
kmeans = MiniBatchKMeans(n_clusters = 256, random_state=1)

# Fit the model to the training data
kmeans.fit(kmeans_x_train_norm)

# record centroid values
centroids = kmeans.cluster_centers_

# reshape centroids into images
images = centroids.reshape(256, 28, 28)
images *= 255
images = images.astype(np.uint8)

# determine cluster labels
cluster_labels = infer_cluster_labels(kmeans, kmeans_y_train)

# create figure with subplots using matplotlib.pyplot
fig, axs = plt.subplots(32, 8, figsize = (20, 20))
plt.gray();
```

```
# loop through subplots and add centroid images
for i, ax in enumerate(axs.flat):

    # determine inferred label using cluster_labels dictionary
    for key, value in cluster_labels.items():
        if i in value:
            ax.set_title('Inferred Label:{}'.format(key), fontsize=8)

    # add image to subplot
    ax.matshow(images[i]);
    ax.axis('off');

# display the figure
fig.show();
```




Apply the K-means Clustering Model to the Test Dataframe

In [33]:
```
# Create a dataframe for predictor variables in the test dataframe for kmeans model
kmeans_testing_x = digit_testing_data.copy(deep=True)
#kmeans_testing_x.drop(['Label'], axis=1, inplace=True)

# Apply the kmeans model to the test dataset
y_test_prediction_clusters_kmeans = kmeans.predict(kmeans_testing_x)

# predict labels for kmeans model
kmeans_predictions = infer_data_labels(y_test_prediction_clusters_kmeans, cluster_labe

# Put the kmeans predictions into a Pandas dataframe
prediction_df_kmeans = pd.DataFrame(kmeans_predictions, columns=['Label'])

# Add the ID column to the front of the kmeans predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_kmeans.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
#prediction_df_kmeans.to_csv('test_predictions_kmeans_v1.csv', index=False)
```

Let's display the Kaggle results from the application of the kmeans model on the test dataset

In [34]:
```
# Display the kaggle results associated with the Random Forest Model
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Digit_Kmeans_v1.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[34]:
```
<Figure size 1500x1500 with 0 Axes>
```

Out[34]:
```
<matplotlib.image.AxesImage at 0x1dc5be90fd0>
```

Out[34]:
```
(-0.5, 1161.5, 565.5, -0.5)
```