P3 and P4: PCA and Classification of MNIST Images

Will Koehrsen wjk68

April 9, 2018

Contents

1 Introduction **1**
1.1 Dataset: MNIST Images . . . 2
1.1.1 Example of Single Digit . . . 4
1.2 Plot of Original Digits . . . 5

2 MNIST Digits PCA **6**
2.1 First 25 Principal Components as Images . . . 6
2.2 Plot of Explained Variance . . . 7

3 Plot of PC1 vs PC2 **8**

4 Classifying Digits Using EM Algorithm **10**
4.1 Run the Algorithm on 2 PC . . . 11
4.2 Visualize Resulting Estimated Gaussians . . . 13
4.3 Plot the True Gaussians . . . 15
4.4 Accuracy of Classifications . . . 17
4.4.1 Plot of Actual Values and Predicted Values . . . 18

5 Classification with 3 Principal Components **20**

6 Conclusions **24**

1 Introduction

In this notebook, we will explore classifying the MNIST handwritten image dataset. We will use principal components analysis for dimensionality reduction, a Gaussian Mixture model as the generating function, and the expectation maximization algorithm to assign class responsibility to images as well as find the parameters of the distributions.

In [1]:
```
# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# Matplotlib for visualization
import matplotlib.pyplot as plt
%matplotlib inline

import matplotlib
matplotlib.rcParams['font.size'] = 18

import matplotlib.mlab as mlab
import matplotlib.lines as mlines

# Utilities
import pprint
from collections import Counter
from bokeh.palettes import Category10_10
```

In [2]:
```
# MNIST digits are included in sklearn
from sklearn.datasets import load_digits

# PCA for  dimensionality reducton
from sklearn.decomposition import PCA
```

In [3]:
```
# Expectation Maximization algorithm developed in previous notebook
from em import EM_algorithm
```

1.1 Dataset: MNIST Images

In [4]:
```
digits = load_digits(n_class=10)
pprint.pprint(digits.DESCR)
```

```
('Optical Recognition of Handwritten Digits Data Set\n'
 '===================================================\n'
 '\n'
 'Notes\n'
 '-----\n'
 'Data Set Characteristics:\n'
 '    :Number of Instances: 5620\n'
 '    :Number of Attributes: 64\n'
 '    :Attribute Information: 8x8 image of integer pixels in the range 0..16.\n'
 '    :Missing Attribute Values: None\n'
 "    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)\n"
 '    :Date: July; 1998\n'
```

```
 '\n'
 'This is a copy of the test set of the UCI ML hand-written digits datasets\n'
 'http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits\n'
 '\n'
 'The data set contains images of hand-written digits: 10 classes where\n'
 'each class refers to a digit.\n'
 '\n'
 'Preprocessing programs made available by NIST were used to extract\n'
 'normalized bitmaps of handwritten digits from a preprinted form. From a\n'
 'total of 43 people, 30 contributed to the training set and different 13\n'
 'to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of\n'
 '4x4 and the number of on pixels are counted in each block. This generates\n'
 'an input matrix of 8x8 where each element is an integer in the range\n'
 '0..16. This reduces dimensionality and gives invariance to small\n'
 'distortions.\n'
 '\n'
 'For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.\n'
 'T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.\n'
 'L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,\n'
 '1994.\n'
 '\n'
 'References\n'
 '----------\n'
 '  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their\n'
 '    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of\n'
 '    Graduate Studies in Science and Engineering, Bogazici University.\n'
 '  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.\n'
 '  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.\n'
 '    Linear dimensionalityreduction using relevance weighted LDA. School of\n'
 '    Electrical and Electronic Engineering Nanyang Technological University.\n'
 '    2005.\n'
 '  - Claudio Gentile. A New Approximate Maximal Margin Classification\n'
 '    Algorithm. NIPS. 2000.\n')
```

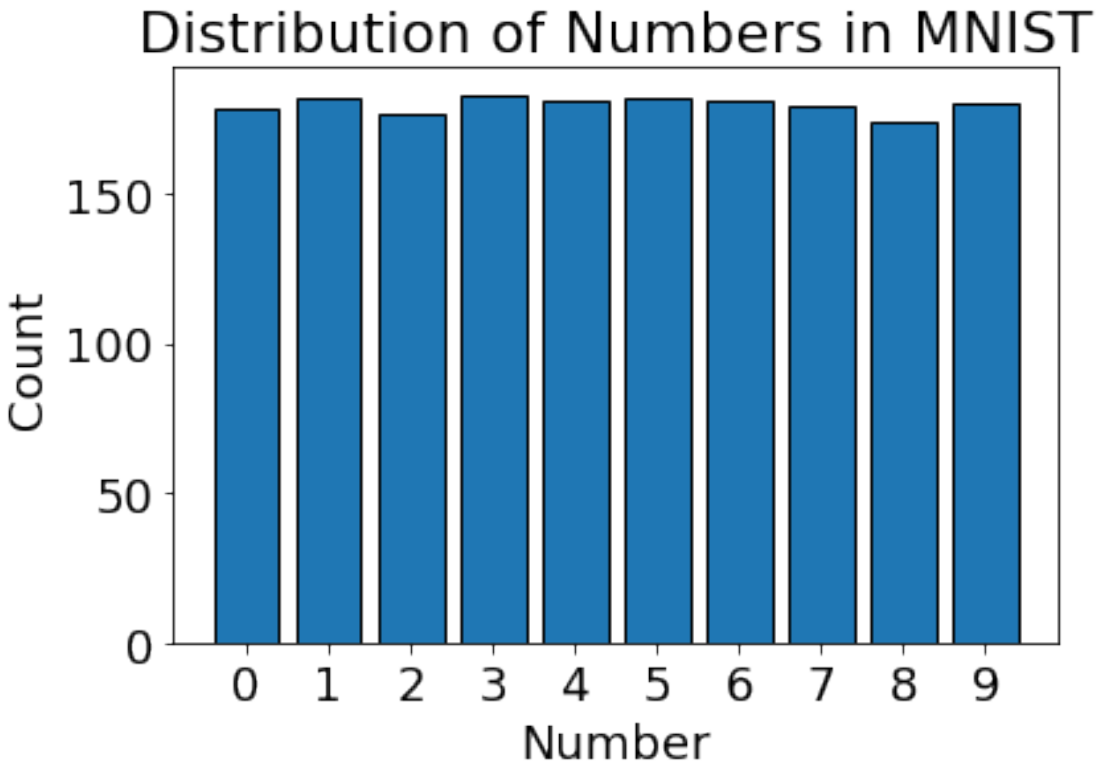
In [5]:
```
# Images as flattened vectors
features = digits.data
features.shape
```

Out[5]:
```
(1797, 64)
```

In [6]:
```
labels = digits.target
labels.shape
```

Out[6]:
```
(1797,)
```

```
In [7]: digits_counter = Counter(labels)
        plt.bar(digits_counter.keys(), digits_counter.values(), edgecolor = 'k');
        plt.xlabel('Number'); plt.ylabel('Count'); plt.title('Distribution of Numbers in
        plt.xticks(range(10), range(10));
```



1.1.1 Example of Single Digit

```
In [8]: features[1].reshape((8, 8))
```

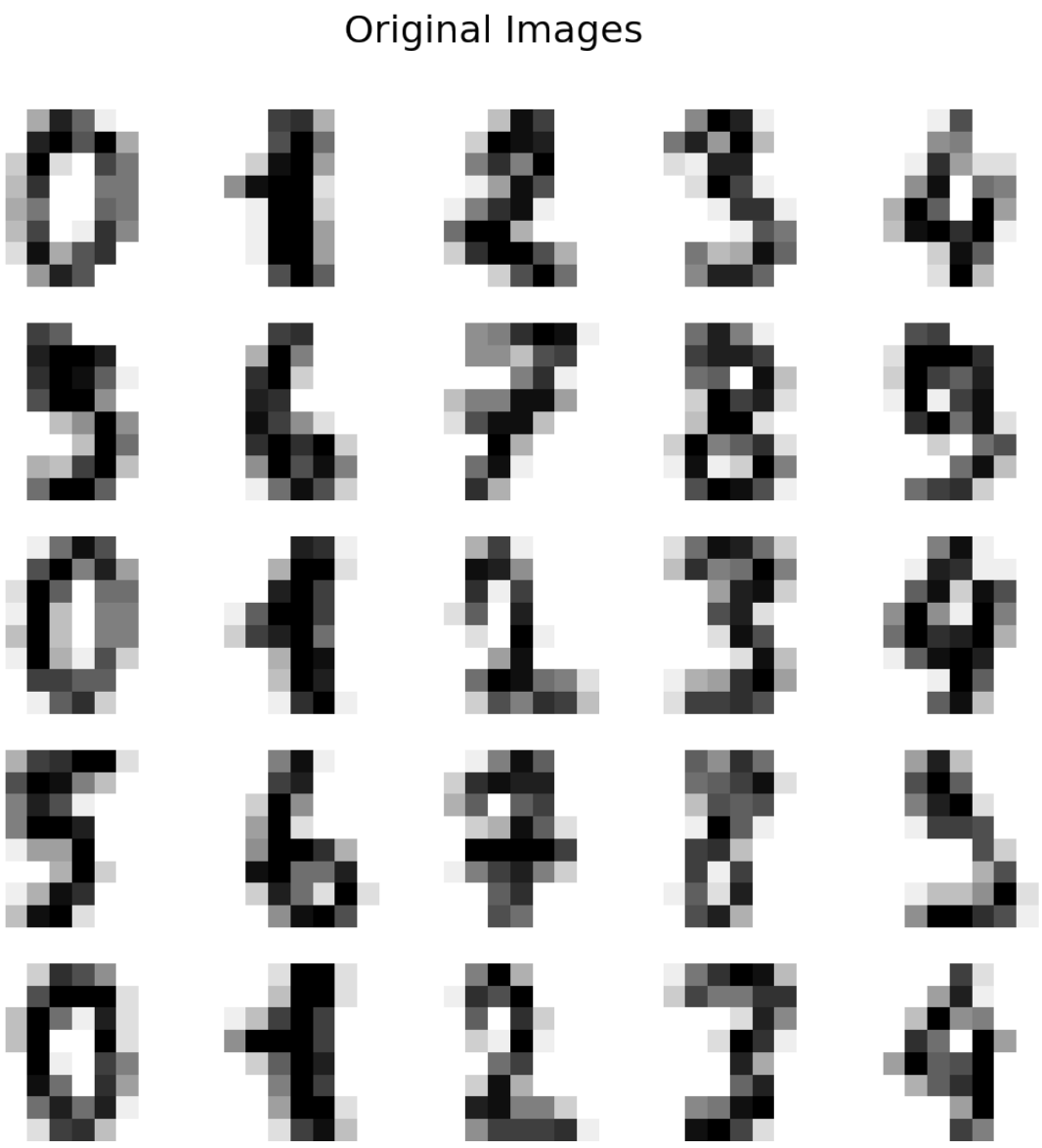
```
Out[8]: array([[ 0.,  0.,  0., 12., 13.,  5.,  0.,  0.],
               [ 0.,  0.,  0., 11., 16.,  9.,  0.,  0.],
               [ 0.,  0.,  3., 15., 16.,  6.,  0.,  0.],
               [ 0.,  7., 15., 16., 16.,  2.,  0.,  0.],
               [ 0.,  0.,  1., 16., 16.,  3.,  0.,  0.],
               [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
               [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
               [ 0.,  0.,  0., 11., 16., 10.,  0.,  0.]])
```

There are 1797 images in the MNIST dataset, each of which is represented by an 8 x 8 grid of pixel intensities. The features represent the digits flattened to a 1-D vector with 64 values. The labels are the true value (target) for each image. The distribution of digits is nearly uniform with no prominent skew to one digit.

1.2 Plot of Original Digits

```
In [9]: def plot_digits(features, title):
            plt.figure(figsize = (16, 16))
            for i in range(25):
                img = features[i].reshape((8,8))

                plt.subplot(5, 5, i + 1)
                plt.imshow(img, cmap = 'binary', interpolation='nearest')
                plt.axis('off')
            plt.suptitle(title, y = 0.95, size = 32)
            plt.show()
```

```
In [10]: plot_digits(features, 'Original Images')
```



2 MNIST Digits PCA

We will perform Principal Components Analysis on the digits to reduce the dimensions for visualization purposes. We will retain 99% of the variance to see how many components this requires, but then we will actually use only the first two principal components to make the classifications.

```
In [11]: # Perform PCA with 99% of the variance explained
         pca = PCA(n_components = 0.99, random_state = 50)
         pca_features = pca.fit_transform(features)
         pca_features.shape
```

```
Out[11]: (1797, 41)
```

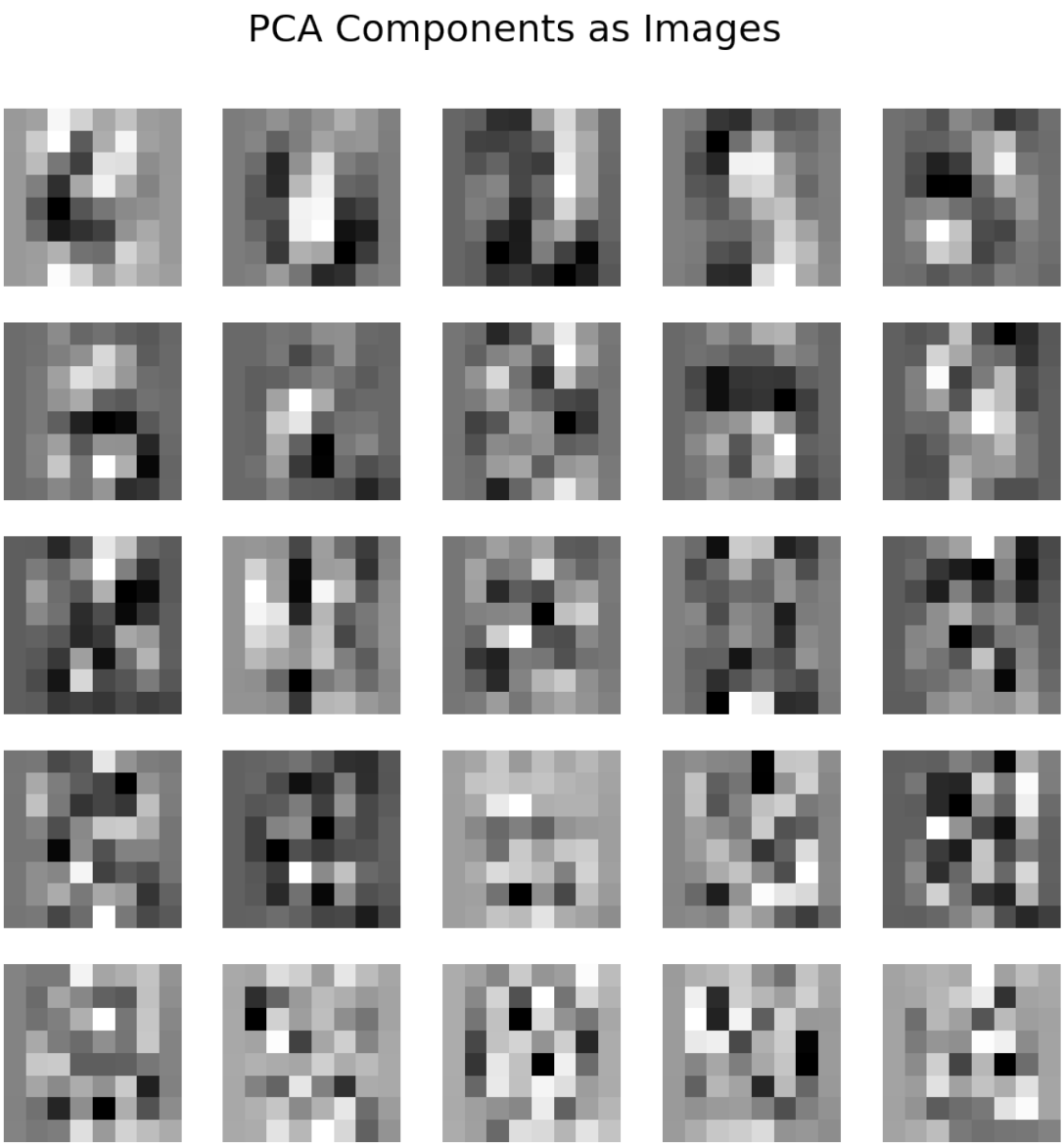
```
In [12]: pca.components_.shape
```

```
Out[12]: (41, 64)
```

The digits are each now represented by 41 features instead of 64. Each of these features represents the images' weight for the corresponding principal component. The 41 principal components in turn have a weight associated with each of the original 64 features. We can visual the first 25 principal components as images.

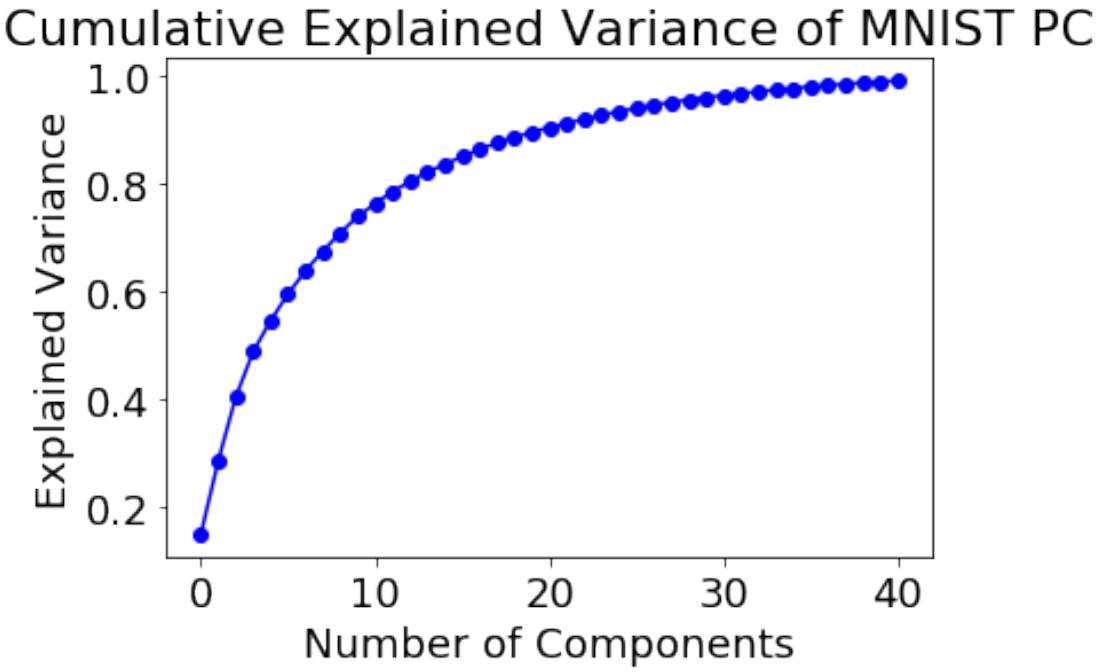
2.1 First 25 Principal Components as Images

```
In [13]: plot_digits(pca.components_, 'PCA Components as Images');
```



2.2 Plot of Explained Variance

```
In [14]: plt.plot(np.cumsum(pca.explained_variance_ratio_), 'bo-')
         plt.xlabel('Number of Components'); plt.ylabel('Explained Variance');
         plt.title('Cumulative Explained Variance of MNIST PC');
```




```
In [15]: print('The first 2 PC explain {:.2f}% of the variance in the MNIST digits.'.for
```

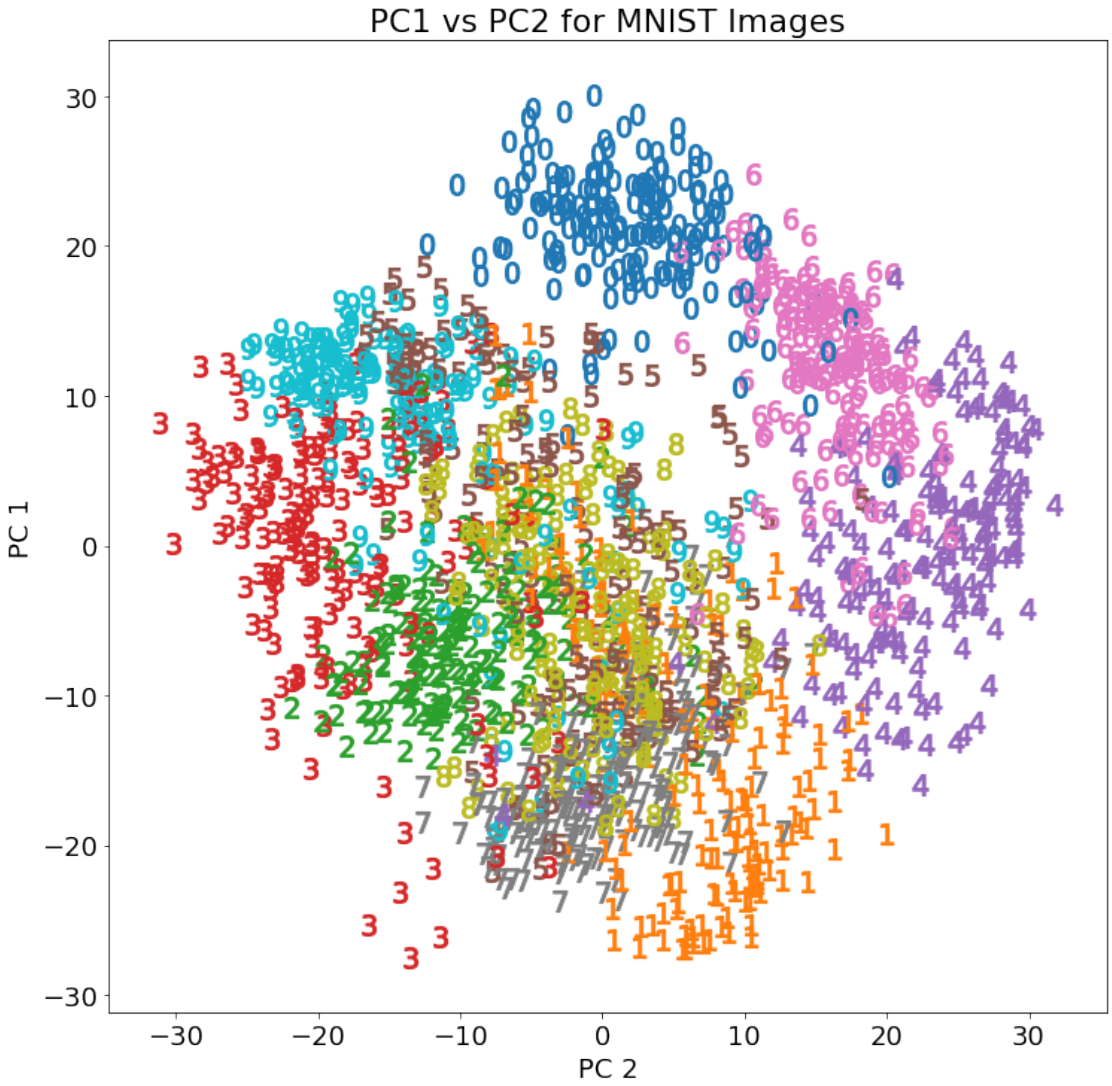
```
The first 2 PC explain 28.51% of the variance in the MNIST digits.
```

3 Plot of PC1 vs PC2

We can plot the first versus the second principal component to see if there are separable clusters. The number of each data point is double encoded in both the color and the marker.

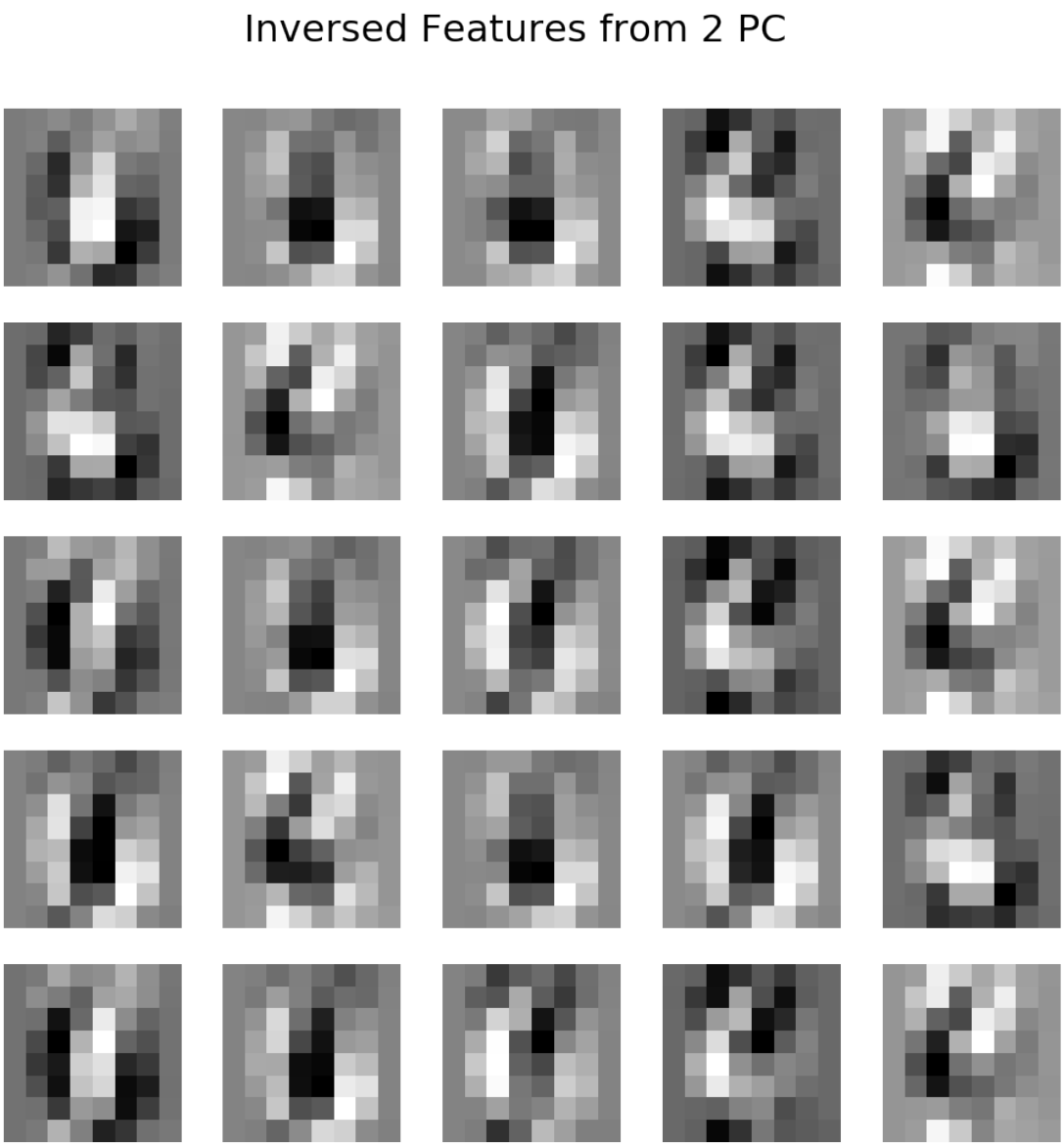
```
In [16]: # Dictionary mapping digit to a color
         color_dict = {number: color for number, color in zip(set(labels), Category10_10
```

```
In [17]: plt.figure(figsize = (12, 12))

         # Plot each of the data points
         for x, y, label in zip(pca_features[:, 0], pca_features[:, 1], labels):
             # Marker and color represents actual number
             marker = '$%d$' % label
             color = color_dict.get(label)
             plt.scatter(x, y, marker = marker, color = color, s = 200)

         plt.xlabel('PC 2'); plt.ylabel('PC 1');
         plt.title('PC1 vs PC2 for MNIST Images');
```



From the PC1 vs PC2 plot, the digits are not clearly separable into 10 classes but do appear to form groups. Using more principal components would likely separate the digits better, but then we would lose the ability to visualize the data.

```
In [18]: inv_features = np.dot(pca_features[:, :2], pca.components_[:2, :])
         plot_digits(inv_features, 'Inversed Features from 2 PC')
```



It is difficult to identify the digits obtained through reconstructing the images using only the first 2 principal components. With only 28% of the variance explained, the first 2 PC do not capture enough information to allow for easy human identification of the reconstructed digits.

4 Classifying Digits Using EM Algorithm

Now we will move on to classifying the digits using a Gaussian Mixture Model and the Expectation Maximization algorithm. We will use the first 2 PC of the MNIST images as our data and the functions developed in the previous notebook. A Gaussian Mixture

Model assigns fractional responsibilities for each data point to each cluster rather than a hard (single) classfication. This expresses uncertainty in classification, but we can also use the fractional responsibilities to find the most likely class and calculate an accuracy based on the known targets.

```
In [19]: digits_two_pc = pca_features[:, :2]
         digits_two_pc.shape
```

```
Out[19]: (1797, 2)
```

4.1 Run the Algorithm on 2 PC

```
In [20]: def initialize_and_run(data, n_cluster):
             # Select random points for initial cluster means
             selected = np.random.choice(len(data), n_cluster, replace = False)
             initial_means = [data[i] for i in selected]

             # All the covariances are set to the covariance of the entire dataset
             initial_covs = [np.cov(data, rowvar=0)] * n_cluster

             # Uniform initial weights
             initial_weights = [1./n_cluster] * n_cluster
             initial = {'data': data, 'weights': initial_weights,
                        'means': initial_means, 'covs': initial_covs}

             # Run the algorithm, reduce the max number of iterations
             results = EM_algorithm(data, initial_means, initial_covs, initial_weights,
                                    maxiter=100, thresh = 1e-2)

             return initial, results

         initial, results = initialize_and_run(digits_two_pc, 10)
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 25
Iteration 30
Iteration 35
Iteration 40
Iteration 45
Iteration 50
Iteration 55
```

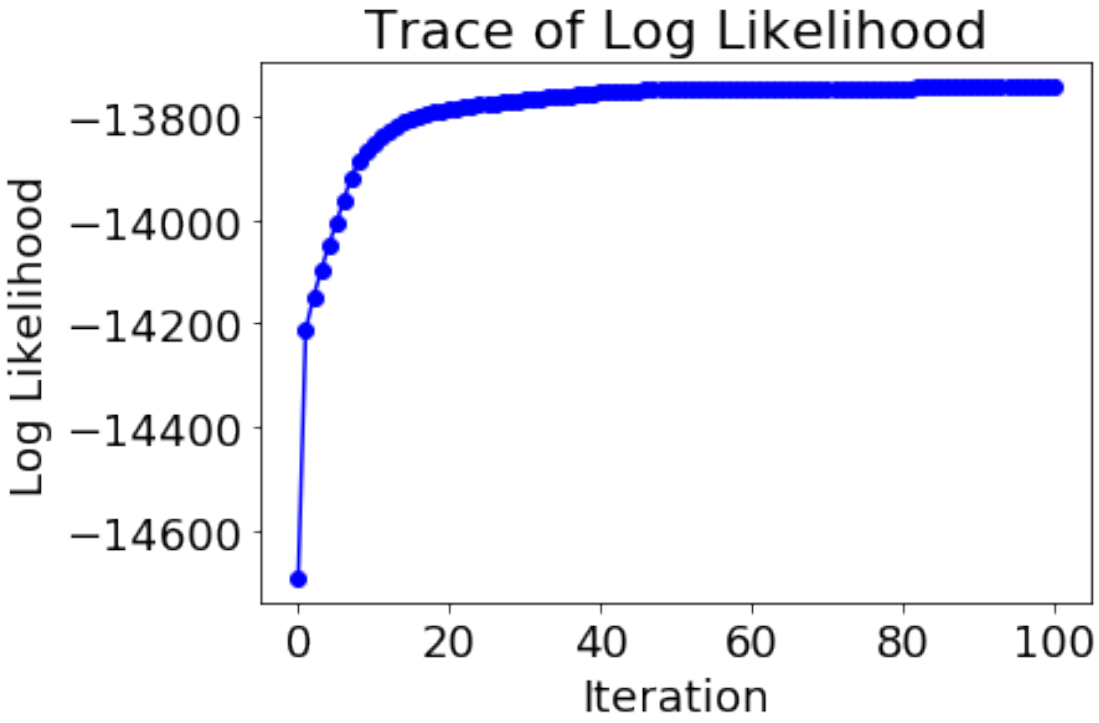
```
Iteration 60
Iteration 65
Iteration 70
Iteration 75
Iteration 80
Iteration 85
Iteration 90
Iteration 95
```

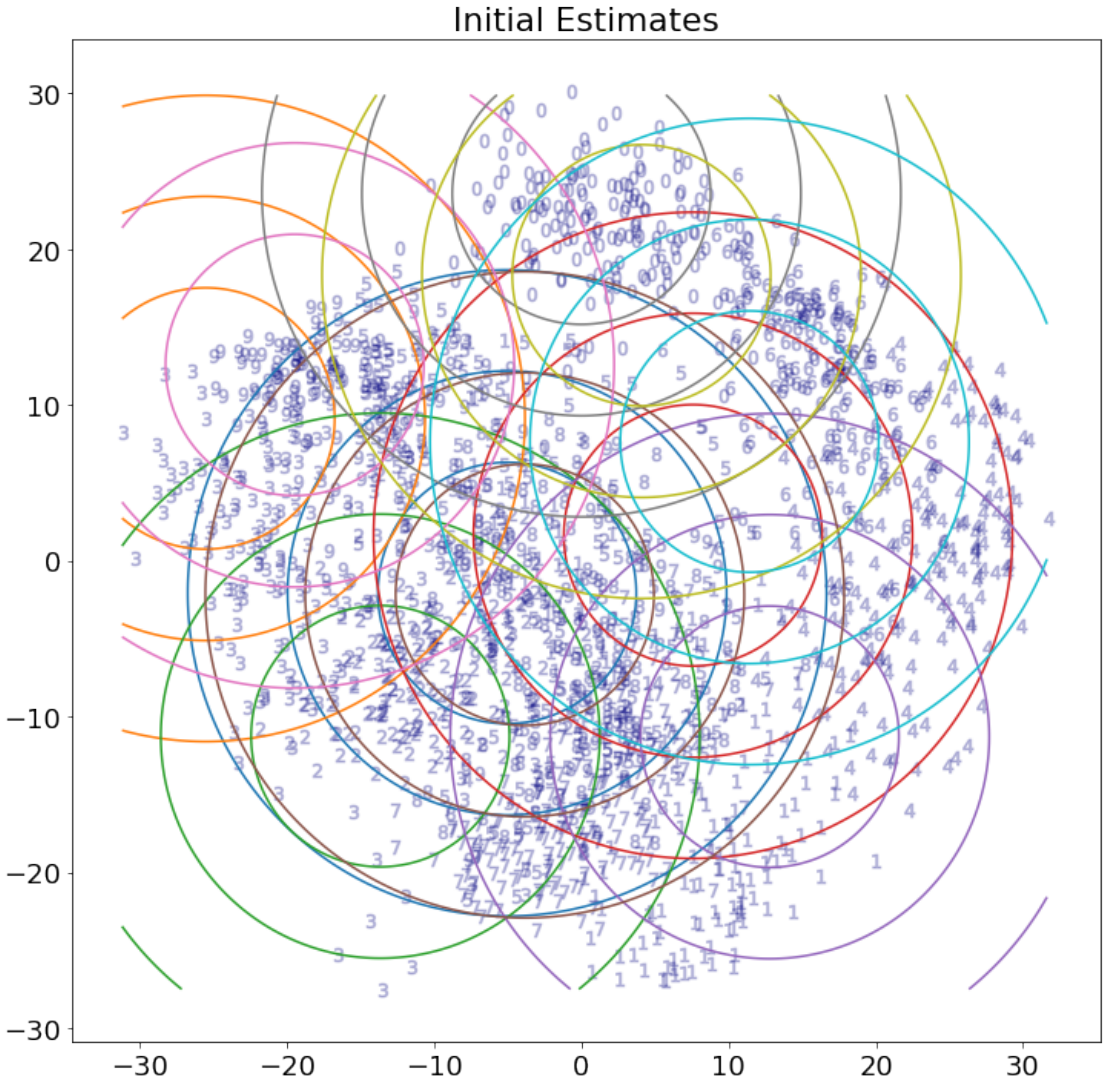
In [21]:
```
# Plot the log likelihood trace.
# Plot the data points with the initial parameters
# Plot the data points with the final parameters after EM algorith,
def plot_results(initial, results):
    data = initial['data']

    plt.figure(figsize = (6, 4))
    plt.plot(results['loglike'], 'bo-')
    plt.xlabel('Iteration'); plt.ylabel('Log Likelihood')
    plt.title('Trace of Log Likelihood'); plt.show();

    delta = 0.2
    x = np.arange(np.min(data[:, 0]), np.max(data[:, 0]), delta)
    y = np.arange(np.min(data[:, 1]), np.max(data[:, 1]), delta)
    X, Y = np.meshgrid(x, y)

    col = Category10_10

    initial_data = True
    # Plot the initial estimates and then the final results
    for d in [initial, results]:
        plt.figure(figsize = (12, 12))

        # Plot the data as numbers
        for xp, yp, label in zip(data[:, 0], data[:, 1], labels):
            plt.scatter(xp, yp, c = 'navy', s = 80, marker = '$%d$' % label, al

        # Iterate through the Gaussians
        for k in range(len(d['means'])):
            # Extract information
            mean = d['means'][k]
            cov = d['covs'][k]
            sigmax = np.sqrt(cov[0][0])
            sigmay = np.sqrt(cov[1][1])
            sigmaxy = cov[0][1]
```

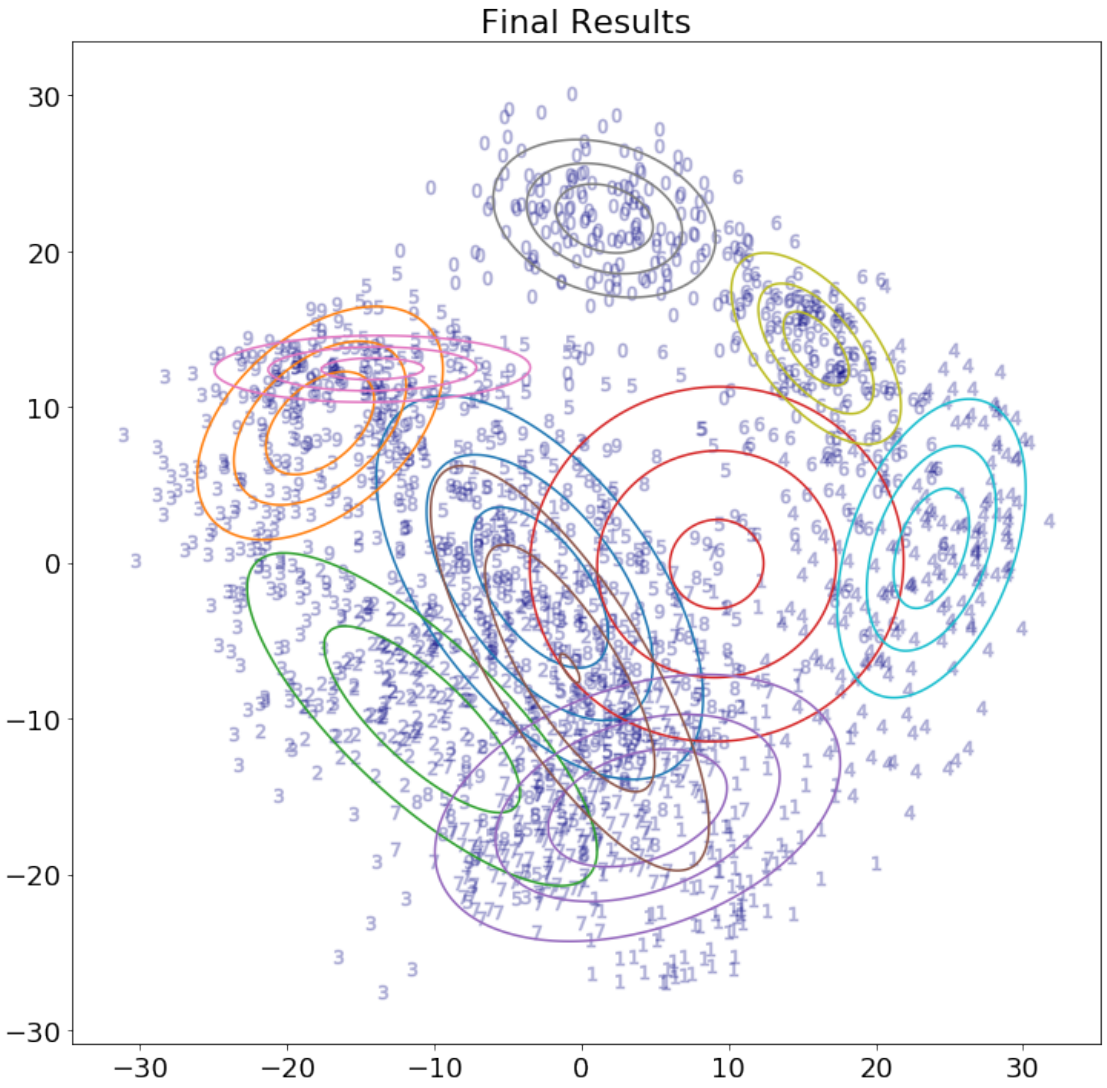
```
            Z = mlab.bivariate_normal(X, Y, sigmax, sigmay,
                                      mean[0], mean[1], sigmaxy)

            plt.contour(X, Y, Z, 3, colors = col[k])

    if initial_data:
        plt.title('Initial Estimates')
        initial_data = False
        plt.show()
    else:
        plt.title('Final Results')
        plt.show()
```

4.2 Visualize Resulting Estimated Gaussians

In [22]: `plot_results(initial, results);`



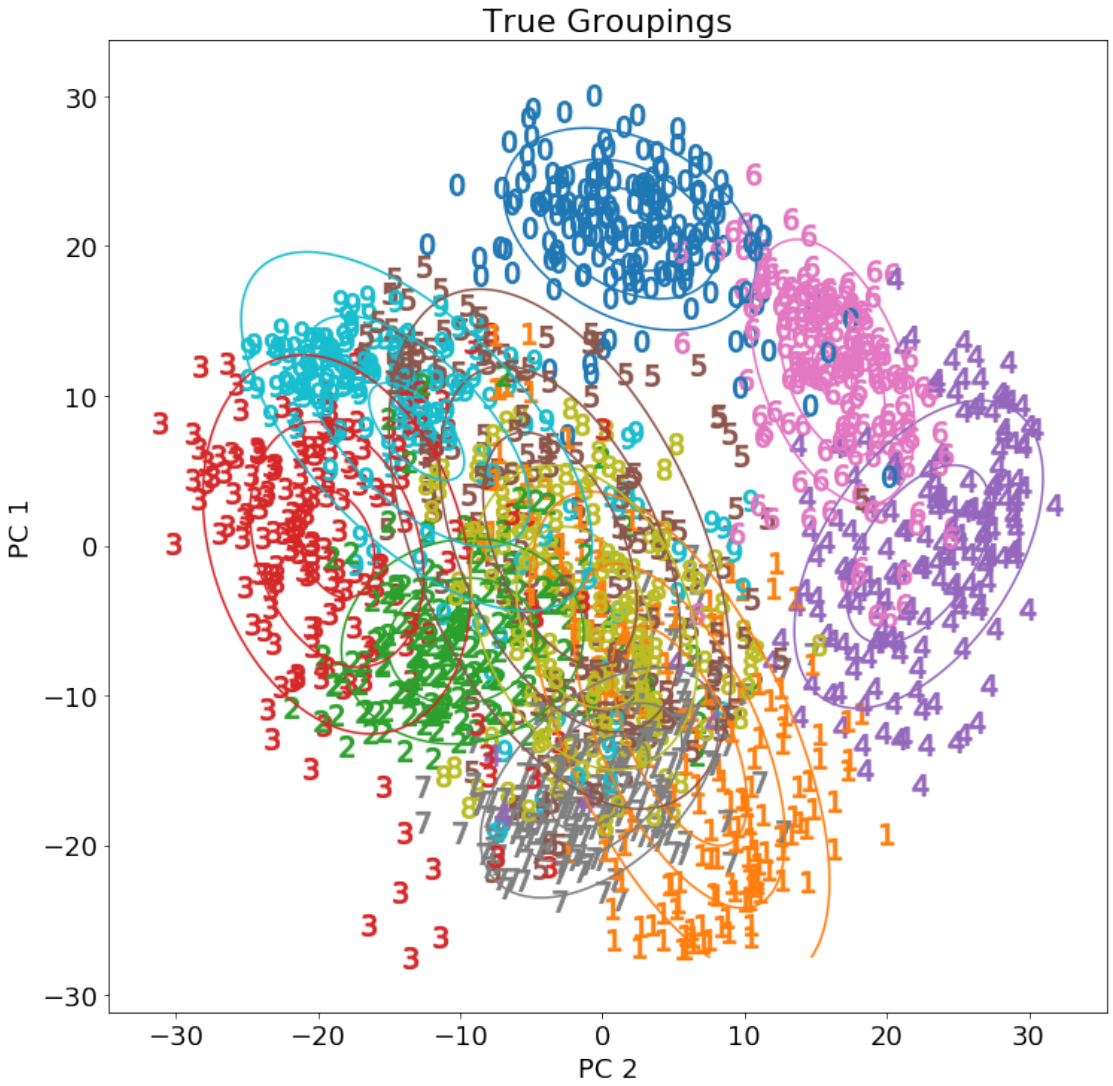

Initial Estimates
30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30



4.3 Plot the True Gaussians

In [23]:
```
true_means = []
true_covariances = []

# Extract the features corresponding to each number
for i in set(labels):
    subset = digits_two_pc[labels == i]

    # Find the mean and covariances
    true_means.append(np.mean(subset, axis=0))
    true_covariances.append(np.cov(subset, rowvar=0))
```

```
true_means = np.array(true_means)
true_covariances = np.array(true_covariances)

# Dictionary of true means and covariances
true_dict = {'means': true_means, 'covs': true_covariances}

plt.figure(figsize = (12, 12))

# Plot the data points with numbers encoded as markers and colors
for x, y, label in zip(digits_two_pc[:, 0], digits_two_pc[:, 1], labels):
    marker = '$%d$' % label
    color = color_dict.get(label)
    plt.scatter(x, y, marker = marker, color = color, s = 200)

plt.xlabel('PC 2'); plt.ylabel('PC 1');

delta = 0.2
x = np.arange(np.min(digits_two_pc[:, 0]), np.max(digits_two_pc[:, 0]), delta)
y = np.arange(np.min(digits_two_pc[:, 1]), np.max(digits_two_pc[:, 1]), delta)
X, Y = np.meshgrid(x, y)
col = Category10_10

# Iterate through the Gaussians
for k in range(len(true_dict['means'])):
    # Extract information
    mean = true_dict['means'][k]
    cov = true_dict['covs'][k]
    sigmax = np.sqrt(cov[0][0])
    sigmay = np.sqrt(cov[1][1])
    sigmaxy = cov[0][1]
    Z = mlab.bivariate_normal(X, Y, sigmax, sigmay,
                              mean[0], mean[1], sigmaxy)

    plt.contour(X, Y, Z, 3, colors = col[k])

plt.title('True Groupings');
```



4.4 Accuracy of Classifications

The Gaussian Mixture Model assigns fractional responsibilities to each data point for each of the classes. We can find the maximium responsibility for each data point to determine the most likely class. Then, for each of the estimate classes, we find the true label that is assigned to each class most often to serve as the predicted label. Once we have the predicted labels, we can then determine an accuracy.

```
In [25]: group_dict
```

```
Out[25]: {0: 8, 1: 3, 2: 2, 3: array([5, 6]), 4: 1, 5: 1, 6: 9, 7: 0, 8: 6, 9: 4}
```

```
In [26]: # Maximum responsibility
         groupings = np.argmax(results['resp'], axis = 1)
         classes = pd.DataFrame({'grouping': groupings, 'target': labels, 'pc1': digits_

         # Most common target for each grouping
         group_series = classes.groupby('grouping')['target'].agg(pd.Series.mode)

         # Mappig of assigned group to target
         group_dict = {group: target for group, target in zip(group_series.index, group_

         for key, value in group_dict.items():
             if type(value) != np.int32:
                 group_dict[key] = value[0]

         # 'Predicted' class
         classes['predicted'] = classes['grouping'].replace(group_dict)

         # Colors for plotting
         g_colors_dict = {number: color for number, color in
                          zip(set(classes['predicted']), Category10_10)}

         accuracy = np.mean(np.equal(classes['target'], classes['predicted']))
         print('Accuracy using 2 principal components: {:.2f}%.'.format(100 * accuracy))
```

```
Accuracy using 2 principal components: 50.31%.
```

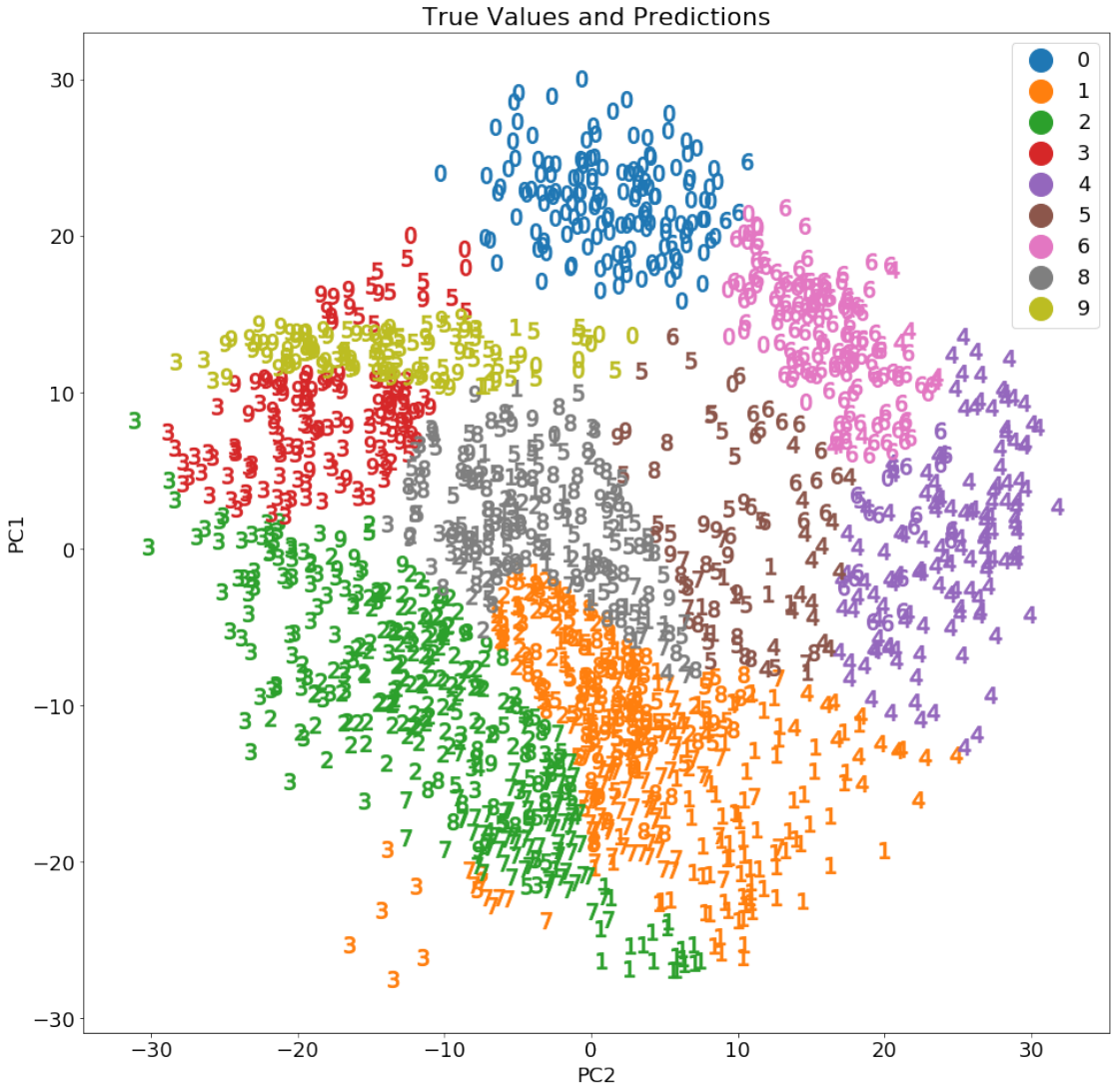
4.4.1 Plot of Actual Values and Predicted Values

In the following plot, the marker shows the actual number and the color corresponds to the predicted class.

```
In [27]: classes = classes.sort_values('predicted')
         legend_handles = [mlines.Line2D([], [], color = g_colors_dict.get(i), markersiz
                                         marker = 'o', label = i, linestyle = 'None') fo
```

```
In [28]: plt.figure(figsize = (16, 16))

         for x, y, label, predicted in zip(classes['pc1'], classes['pc2'],
                                           classes['target'], classes['predicted']):

             # Marker references true value
             marker = '$%d$' % label
             # Color references predicted value
             color = g_colors_dict.get(predicted)
```

```
        plt.scatter(x, y, marker = marker, c = color, s = 200)
        plt.scatter(x, y, c = color, label = predicted, alpha = 0.0)

    plt.legend(handles = legend_handles)
    plt.xlabel('PC2'); plt.ylabel('PC1')
    plt.title('True Values and Predictions');
```



The predicted classes and the actual classes are in close agreement as the model achieves greater than 50% accuracy. We can also make a confusion matrix to see the types of classification mistakes made by the model.

```
In [29]: from sklearn.metrics import confusion_matrix

In [30]: cm = pd.DataFrame(confusion_matrix(y_true = classes['target'], y_pred = classes
         cm.index.name = 'actual'
```

```
In [30]: cm.columns.name = 'predicted'
         cm
```

Out[30]:

predicted	0	1	2	3	4	5	6	7	8	9
actual										
0	150	0	0	3	1	1	16	0	1	6
1	0	123	21	0	0	9	0	0	24	5
2	0	31	123	3	0	0	0	0	19	1
3	0	10	84	67	0	0	0	0	14	8
4	0	15	3	0	138	16	8	0	1	0
5	0	51	22	14	1	18	0	0	44	32
6	5	0	0	0	19	18	139	0	0	0
7	0	88	79	0	0	5	0	0	7	0
8	0	67	21	0	0	12	0	0	74	0
9	0	7	15	58	0	8	0	0	20	72

By inspecting the confusion matrix, we can see the most common mistakes made by the model. For example, the model misclasses many 1, 2, and 5 as 8. The model does well on identifying 0, 4, 6, 7, 8, and 9. Using this information, we could weight the prior probabilities to construct a more accurate model. Another approach to improving the performance is including more information about the digits by retaining additional principal components.

5 Classification with 3 Principal Components

We can repeat the entire process using three prinicipal components to see how that affects the classification accuracy. To show the results, we will only plot the first two prinicipal components with the marker corresponding to the true number and the color representing the prediction. We can also look at the accuracy and the confusion matrix to see how including more information (a greater percentage of the variance) improves model performance.

```
In [43]: digits_three_pc = pca_features[:, :3]
         digits_three_pc.shape
```

```
Out[43]: (1797, 3)
```

```
In [44]: print('The first 3 principal components explain {:0.2f}% of the variance in the
```

```
The first 3 principal components explain 40.30% of the variance in the MNIST digits.
```

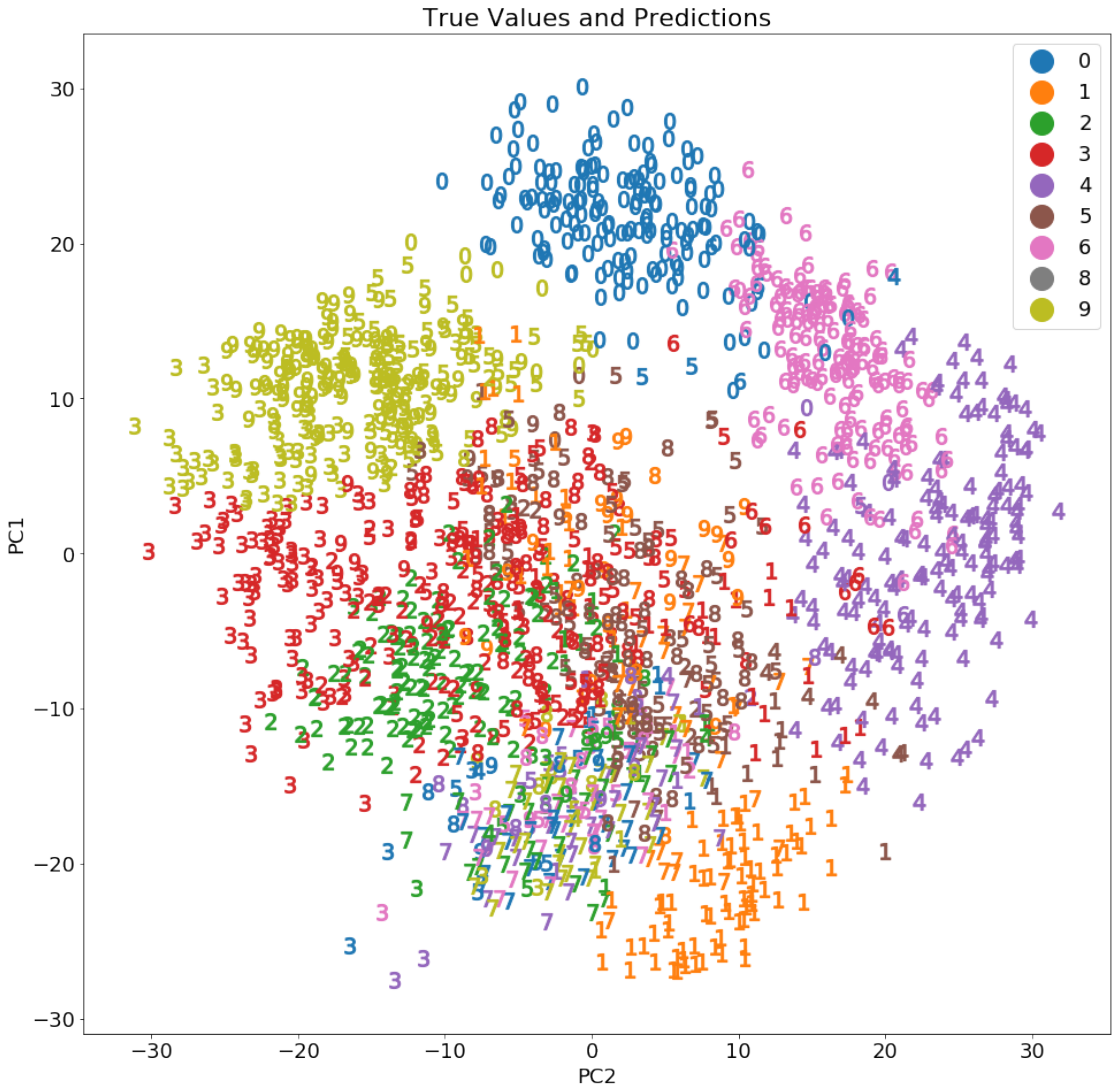
```
In [45]: initial_three, results_three = initialize_and_run(digits_three_pc, 10)
```

```
Iteration 0
Iteration 5
Iteration 10
Iteration 15
Iteration 20
Iteration 25
Iteration 30
Iteration 35
Iteration 40
Iteration 45
Iteration 50
Iteration 55
Iteration 60
Iteration 65
Iteration 70
Iteration 75
Iteration 80
Iteration 85
Iteration 90
Iteration 95
```

In [46]:
```
# Maximum responsibility
groupings = np.argmax(results_three['resp'], axis = 1)
classes = pd.DataFrame({'grouping': groupings, 'target': labels, 'pc1': digits_

# Most common target for each grouping
group_series = classes.groupby('grouping')['target'].agg(pd.Series.mode)

# Mappig of assigned group to target
group_dict = {group: target for group, target in zip(group_series.index, group_

for key, value in group_dict.items():
    if type(value) != np.int32:
        group_dict[key] = value[0]

# 'Predicted' class
classes['predicted'] = classes['grouping'].replace(group_dict)

plt.figure(figsize = (16, 16))

for x, y, label, predicted in zip(classes['pc1'], classes['pc2'],
                                  classes['target'], classes['predicted']):

    # Marker references true value
```

```
            marker = '$%d$' % label
            # Color references predicted value
            color = g_colors_dict.get(predicted)

            plt.scatter(x, y, marker = marker, c = color, s = 200)
            plt.scatter(x, y, c = color, label = predicted, alpha = 0.0)

        plt.legend(handles = legend_handles)
        plt.xlabel('PC2'); plt.ylabel('PC1')
        plt.title('True Values and Predictions');

        accuracy = np.mean(np.equal(classes['target'], classes['predicted']))

        print('Accuracy Using 3 Principal Components: {:.2f}%.'.format(100 * accuracy))
```

```
Accuracy Using 3 Principal Components: 64.27%.
```



In [47]:
```
cm = pd.DataFrame(confusion_matrix(y_true = classes['target'], y_pred = classes
cm.index.name = 'actual'
cm.columns.name = 'predicted'
cm
```

Out[47]:

predicted	0	1	2	3	4	5	6	7	8	9
actual										
0	165	0	0	0	2	2	1	0	0	8
1	0	104	7	29	0	14	0	28	0	0
2	0	0	117	45	0	7	0	4	0	4
3	0	0	1	100	0	1	0	12	0	69
4	1	0	0	0	168	7	0	5	0	0
5	2	2	0	43	1	60	0	25	0	49

6	3	0	0	12	1	0	165	0	0	0
7	0	32	0	0	0	4	0	143	0	0
8	0	3	0	79	1	59	0	30	0	2
9	0	23	0	10	0	5	0	9	0	133

The accuracy using 3 prinicipal components is noticeably improved over that attained using 2. This suggests the groupings among numbers are easier to separate in the higher-dimensional space. We would expect this to be the case because we are including more of the variance in the data when we retain additional principal components. Accounting for a greater percentage of the variance should allow for cleaner separation between the digits.

6 Conclusions

In this notebook we looked at classifying the MNIST digits using Principal Components Analysis, Gaussian Mixture Models, and the Expectation Maximization Algorithm. Each of these techniques serve the following purpose:

1. PCA: For dimensionality reduction. Using PCA, we can reduce the 64 dimensions of the original data (8 x 8 images) to any lower number. Here we looked at using 2 and 3 principal components which explained about 28% and 40% of the variance respectively. Reducing the number of dimensions by retaining only those with the most variance allows us to visualize the data and can remove potentially noisy features that do not help to separate the classes.

2. Gaussian Mixture Models: the data is represented as being generated from a mixture of Gaussians. In the MNIST case, the data would come from 10 Gaussians, each of which produces a different digit. We can then try to find the parameters of these distributions and assign probabilities that a given observation comes from a given distribution. GMMs, unlike other clustering methods like K-Means, results in soft classifications, where each observation is assigned a probability from each cluster, representing uncertainty about the classifications.

3. Expectation Maximization: used to find the most probable parameters of the Gaussians given the data. The EM algorithm first assigns a probability for each data point to each cluster (expectation), then updates the cluster parameters based on the assignments (maximization). These two steps are iteratively repeated until the log likelihood of the data given the clusters no longer increases. Therefore, the final result of the algorithm is the most likely Gaussian distributions for the dataset.

We explored using both 2 and 3 principal components to classify the digits and observed a significant increase in accuracy by including the additional component. This is to be expected as the more pcs we retain, the greater percentage of the variation is represented. With 3 principal components, the model was able to classify the data points with greater than 60% accuracy. We also plotted the first versus the second principal component to visually determine if there are identifiable clusters within the data. Even with

only two pcs, there are noticeable clusters corresponding to the numbers which allows for separation and classification. Although the Gaussian Mixture Model was successful in hard classifying the digits (by assigning the most probable class as the label), it is perhaps best suited for problems where data points can belong to multiple classes, such as in document classification. Nonetheless, this notebook showed that even dealing with data that is not necessarily Gaussian, and using only a few principal components, the Gaussian Mixture Model solved with the Expectation Maximization algorithm is a capable method for classification.