In [ ]:

```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()  # for plot styling
import numpy as np
import pandas as pd
from sklearn.cluster import k_means, kmeans_plusplus, KMeans, AgglomerativeClust
from sklearn.decomposition import PCA
import math
import matplotlib.pyplot as plt
import matplotlib as mpl
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import GridSearchCV, train_test_split, cross_val_sc
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
import math
from random import randint
```

Assignment 3 - Clustering

For this assignment you'll need to use some clustering knowledge to build a function that can generate handwritten numbers from a provided number.

The modelling parts of this assignment are not very complex, the application of the clustering algorithms is very similar to the examples from class. This will require a little more manipulation of data, and building a little bit of structure around the models, that's where some of the challenge lies.

Requirements

- Use clustering to take the X data (the features/pixels) of the MNIST dataset, and group it into clusters.
  - Do not use the targets from the dataset at all.
- Assign labels to your clusters, so there is now a label for each cluster. You'll need to manually do a little mapping here by eye.
- Use GMM to build a function that can generate a new digit from the information in the cluster.
- Write a function, writeNumber, that can take in an integer (you can assume it is between 1 and 20 digits, this is mostly for printing purposes, the modelling part isn't impacted by this at all) and print out that integer as a generated handwritten number.
- **When generating the handwritten numbers, each version of a digit should be different. I.e. If the number printed is 22222, there should not be 5 identical 2s, they should vary a bit like real writing.**

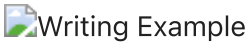
Deliverables

Submit your .ipynb file to the Moodle dropbox. PLEASE make sure that the file runs BY ITSELF outside of importing libraries. It should not reference any other files, either data or code.

Within your file, create a function called writeNumber, which takes a number as an input, and prints it out as a series of handwritten digits. The function should be callable like this, if I wanted to print the number 218201

```
writeNumber(218201)
```

This would produce something that looks, somewhat, like this:


Writing Example

The exact appearance of the generated writing can vary, it likely won't be as well defined as this picture. As long as I can read it as a number, it is fine.

Hint: the number of clusters may vary.

Grading

- 70% - Code works. This is mostly a yes/no thing, if it doesn't work I'll try to see if it was a small or large issue in the code, but it largely works or it doesn't.
- 20% - Numbers appearance. Is the writing OK? I'm not going to be overly picky, if they are reasonably legible, that is fine.
- 10% - Code legibility. Is a quick read over of the code clear? Sections, comments, etc...
-

In [ ]:

```
#Look at an image
def showDigit(digit, label, size=28):
    some_digit = digit
    #turn array into the correct shape
    some_digit_image = np.array(some_digit).reshape(size, size)
    #imshow displays an array like an image
    plt.imshow(some_digit_image, cmap=mpl.cm.binary)
    plt.title(label)
    plt.axis("off")
    plt.show()

#Display multiple digits
def showDigits(digits, labels, indexes, size=28):
    #Make a grid that is the right size
    pics = len(indexes)
    cols = 8
    rows = math.ceil(pics/cols)
    fig, axes = plt.subplots(rows, cols, figsize=(14,6))
    plt.axis("off")

    #loop through the list of indexes, grab images and labels, plot in the "next
    for i in range(0, pics):
```

```
        n = indexes[i]
        some_digit = digits[n:n+1]
        some_digit_image = np.array(some_digit).reshape(size, size)
        ax = axes[i//cols, i%cols]
        ax.axis("off")
        ax.imshow(some_digit_image, cmap=mpl.cm.binary)
        #ax.set_title('Ind: {} - Lbl: {}'.format(indexes[i], labels[n]))
    plt.tight_layout()
    plt.axis("off")
    plt.show()
```

Load Data

Please do not change this (substantially), probably outside of choosing between the full data and a subset. Don't load the target.

Note: testing will be much faster with a subset of records.

In [ ]:
```
#Load Data
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
#mnist = mnist[0:15000]
X = mnist["data"]
print(X.shape)
```

```
(70000, 784)
```

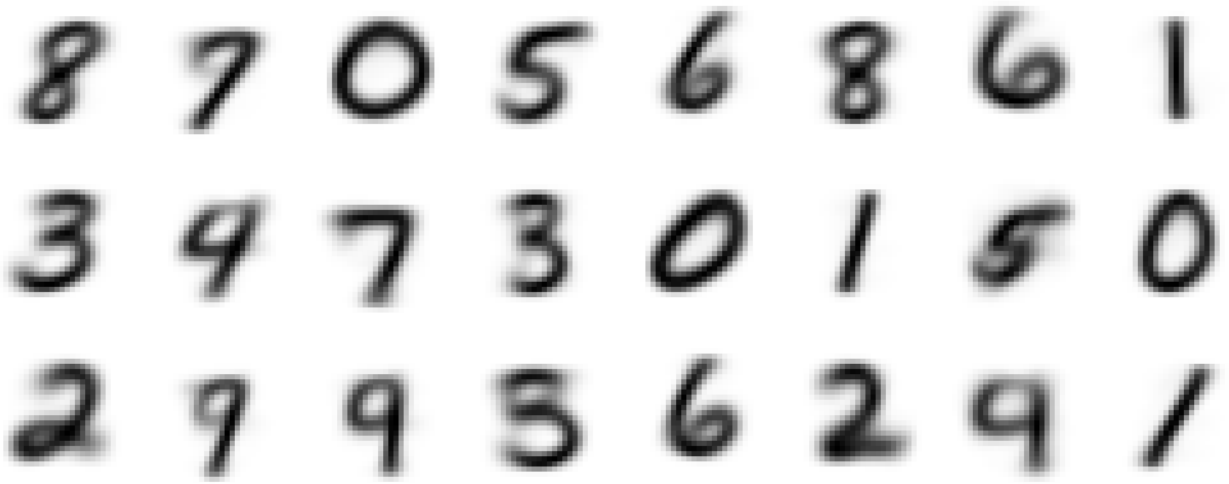
Cluster

We need to break the data into clusters first.

In [ ]:
```
y_null = [" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " "]
indexes = np.array(range(0,24))
```

In [ ]:
```
kmeans = KMeans(n_clusters=24, init="k-means++", random_state=12)
clusters = kmeans.fit_predict(X)
print(kmeans.cluster_centers_.shape)
centers = kmeans.cluster_centers_.reshape(24, 28, 28)
showDigits(digits=centers, labels=y_null, indexes=indexes, size=28)
```

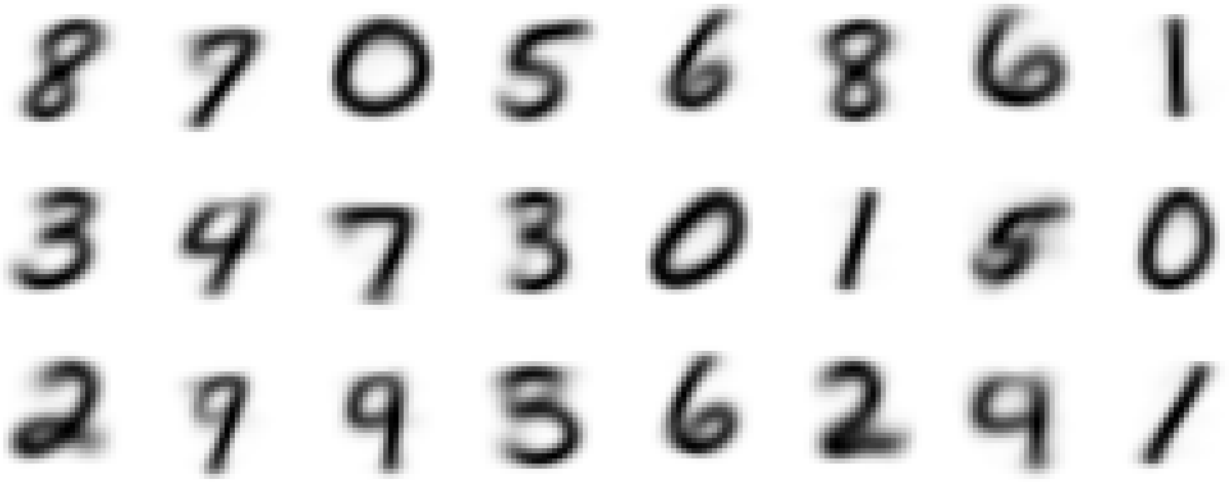
```
(24, 784)
```



Cluster with PCA

In [ ]:

```
kmeans = KMeans(n_clusters=24, init="k-means++", random_state=12)
clus_pca = PCA()
clus_trans = clus_pca.fit_transform(X)
clusters = kmeans.fit_predict(clus_trans)

centers = clus_pca.inverse_transform(kmeans.cluster_centers_).reshape(24, 28, 28
showDigits(digits=centers, labels=y_null, indexes=indexes, size=28)
```



Visually Match the Clusters

Note: The label here is not the actual value of that number, they're assigned sequentially by the clustering, which does not know what each number is.

In [ ]:

```
df = pd.DataFrame(X)
df["label"] = kmeans.labels_
df["label"].value_counts()
```

Out[ ]:

```
18    4221
17    3850
11    3645
```

```
9     3446
19    3339
7     3260
1     3247
14    3227
13    3200
22    3083
21    2936
8     2890
0     2865
20    2798
4     2782
16    2766
5     2735
10    2636
23    2433
2     2318
3     2317
12    2226
15    2031
6     1749
Name: label, dtype: int64
```

In [ ]:
```
map_dict = {0:8, 1:7, 2:0, 3:5, 4:6, 5:8, 6:6, 7:1, 8:3, 9:4, 10:7, 11:3, 12:0,
df["label"] = df["label"].map(map_dict)
df["label"].value_counts()
```

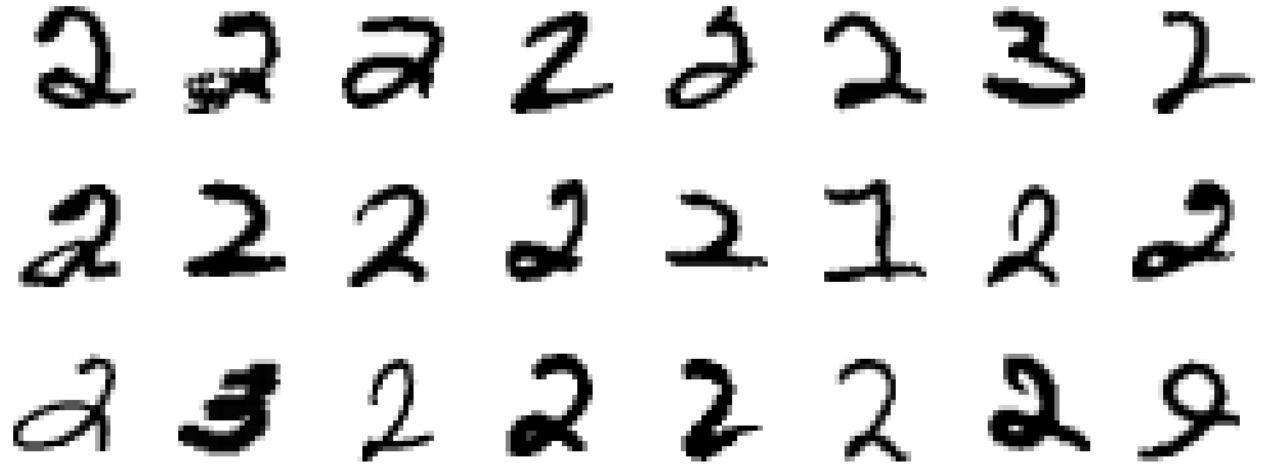
Out[ ]:
```
9    11154
1     8893
5     8883
6     7329
0     6575
3     6535
7     5883
2     5702
8     5600
4     3446
Name: label, dtype: int64
```

In [ ]:
```
for i in range(10):
    samp_ind = df[df["label"] == i].index.values
    rows = df.loc[samp_ind,:].sample(24)
    print_ind = rows.index.values
    y_tmp = np.full(10, i)
    ind_tmp = np.array(range(0,10))
    #print(print_ind)
    print(i)
    showDigits(df.drop(columns={"label"}).to_numpy(), y_tmp, print_ind)
```

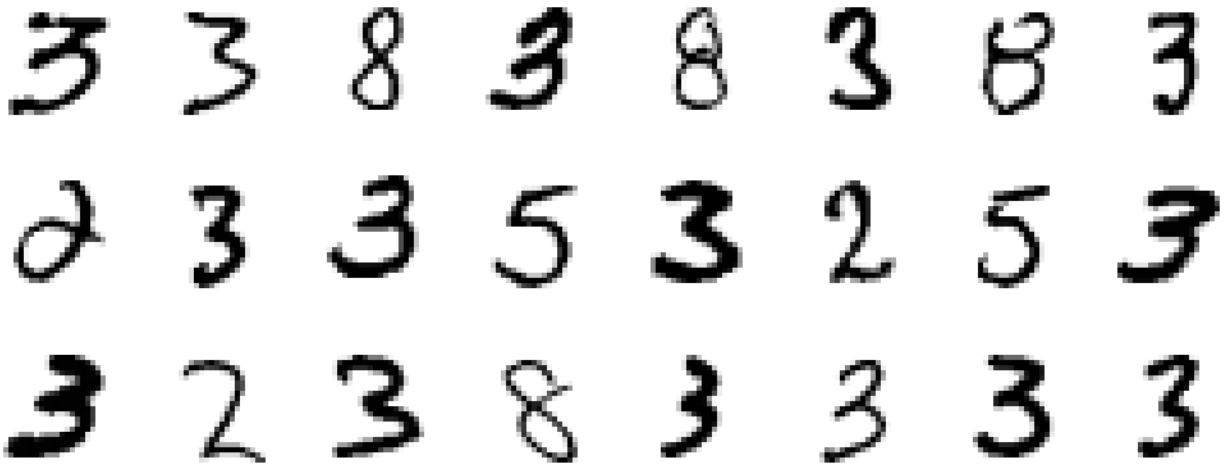
```
0
```

1

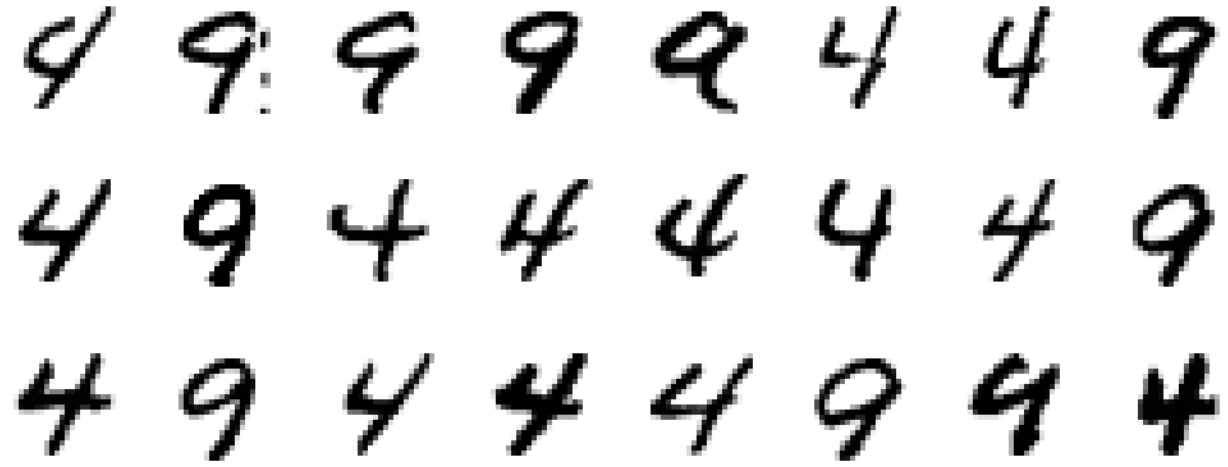
2



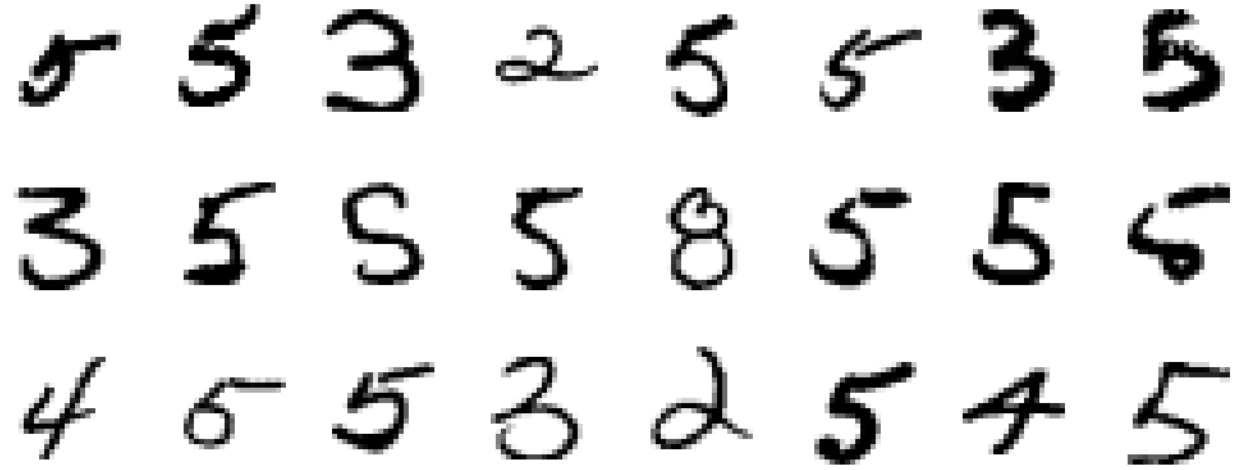
3



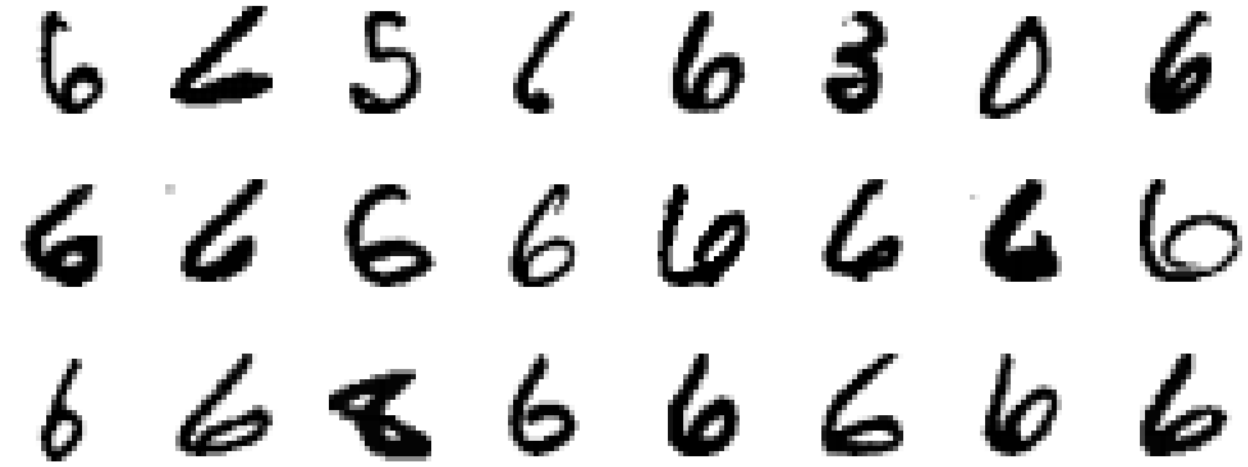
4



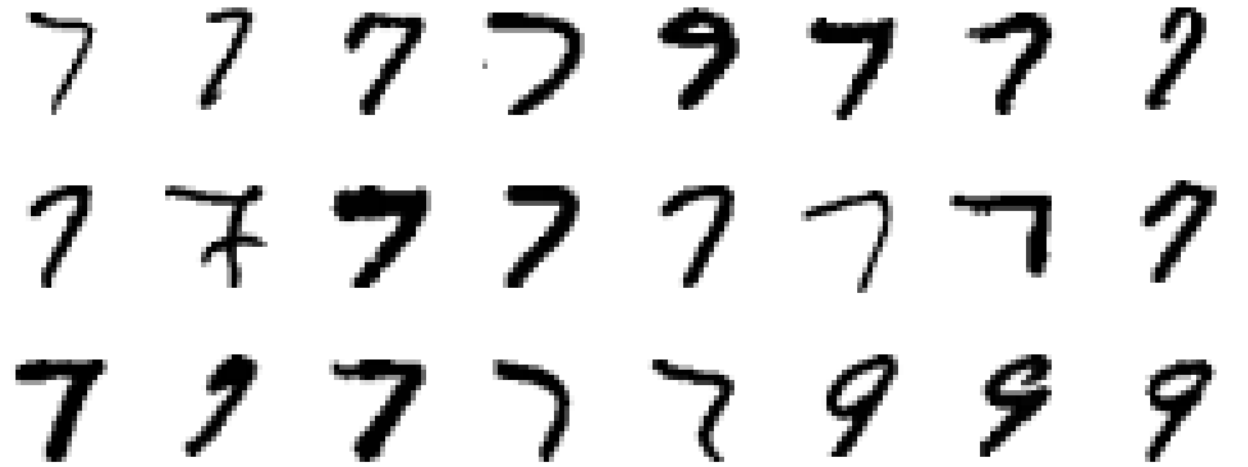
5



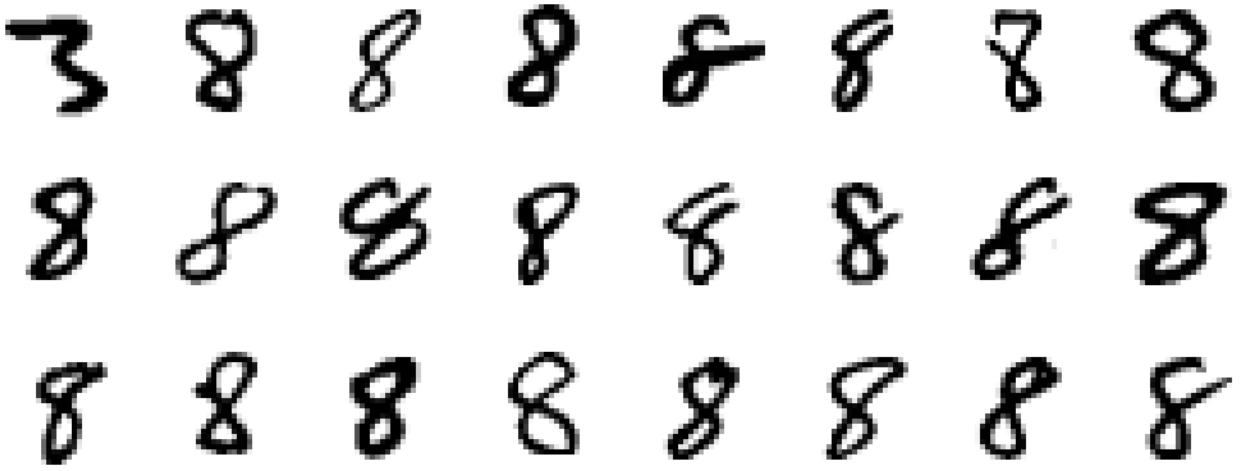
6



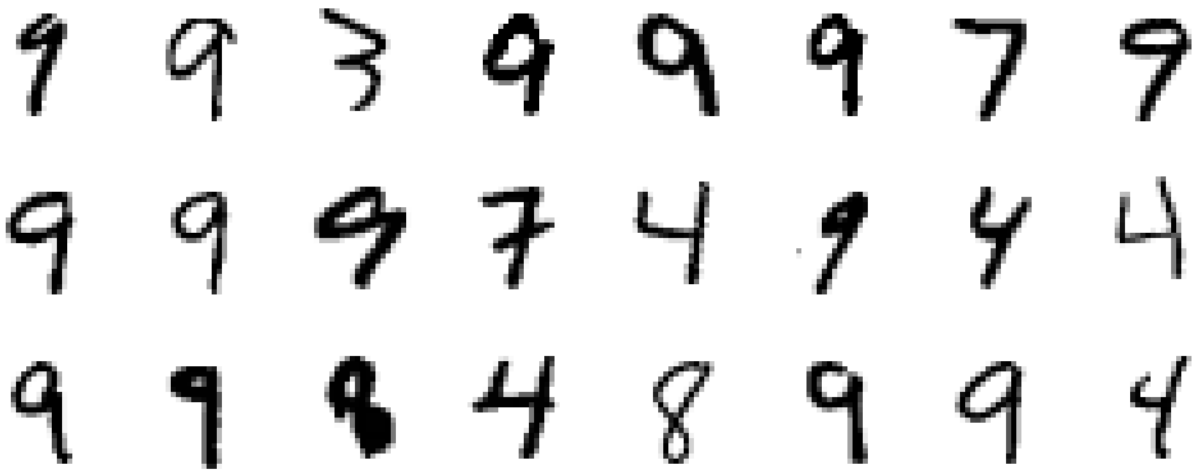
7



8



9



Data is Labeled

We now have a label for each cluster. They are moderately accurate by visual verification.

Create Generators with GMM

In [ ]:
```
np.array(df[df["label"]==1].drop(columns={"label"})).shape
```

Out[ ]:
```
(8893, 784)
```

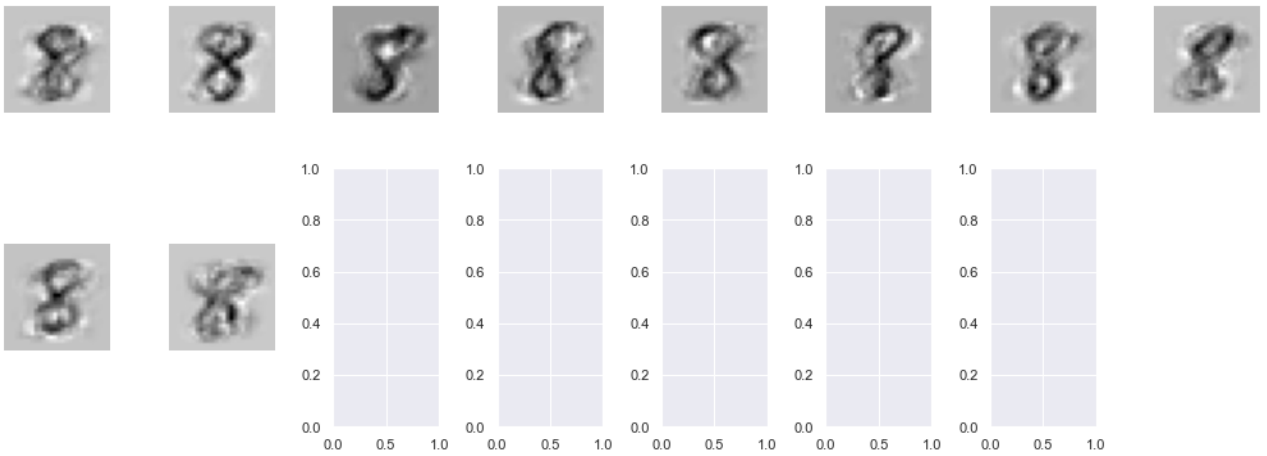
In [ ]:
```
generators = []

for i in range(10):
    gmm = GaussianMixture(covariance_type='full', random_state=0)
    tmp_data = np.array(df[df["label"] == i].drop(columns={"label"}))
    gmm.fit(tmp_data)
    generators.append(gmm)
    print(str(i), "Converged:", gmm.converged_)
```

```
0 Converged: True
1 Converged: True
2 Converged: True
3 Converged: True
4 Converged: True
5 Converged: True
6 Converged: True
7 Converged: True
8 Converged: True
9 Converged: True
```

In [ ]:
```
data_new = generators[8].sample(16)
data_new = data_new[0]
print(data_new.shape)
```

```
(10, 784)
(10, 784)
```

In [ ]:
```
showDigits(digits=data_new, labels=y_null, indexes=[1,2,3,4,5,6,7,8,9,0], size=2
```



Print a Number

In [ ]:

```
def get_digit(number, i):
    return number // 10**i % 10

def num_printer(number):
    cols = int(math.log10(number))+1
    rows = 1
    fig, axes = plt.subplots(rows, cols, figsize=(14,6))

    i = 0
    while i < cols:
        n = get_digit(number,i)
        dig = generators[n].sample(100)

        dig = dig[0][randint(0,100)]
        ax = axes[cols-1-i] #Need to find why it is backwards
        img = np.array(dig).reshape(28,28)
        ax.imshow(img, cmap=mpl.cm.binary, interpolation='nearest')
        #ax.imshow(img, cmap='gray_r')
        #title_str = "I:", str(i), "N:", str(n)
        #ax.set_title(title_str)
        i += 1

    plt.tight_layout()
    plt.setp(plt.gcf().get_axes(), xticks=[], yticks=[])
    plt.show()
```

In [ ]:

```
num_printer(222226666888)
```