In [62]:
```
import random
from base64 import b64decode
from json import loads
import numpy as np
import matplotlib.pyplot as plt
# set matplotlib to display all plots inline with the notebook
%matplotlib inline
```

In [63]:
```
def parse(x):
    """
    to parse the digits file into tuples of
    (labelled digit, numpy array of vector representation of digit)
    """
    digit = loads(x)
    array = np.fromstring(b64decode(digit["data"]),dtype=np.ubyte)
    array = array.astype(np.float64)
    return (digit["label"], array)
```

In [64]:
```
# read in the digits file. Digits is a list of 60,000 tuples,
# each containing a labelled digit and its vector representation.
with open("digits.base64.json","r") as f:
    digits = map(parse, f.readlines())
```

In [65]:
```
# pick a ratio for splitting the digits list into a training and a validation set.
ratio = int(len(digits)*0.25)
validation = digits[:ratio]
training = digits[ratio:]
```

In [66]:
```
def display_digit(digit, labeled = True, title = ""):
    """
    graphically displays a 784x1 vector, representing a digit
    """
    if labeled:
        digit = digit[1]
    image = digit
    plt.figure()
    fig = plt.imshow(image.reshape(28,28))
    fig.set_cmap('gray_r')
    fig.axes.get_xaxis().set_visible(False)
    fig.axes.get_yaxis().set_visible(False)
    if title != "":
        plt.title("Inferred label: " + str(title))
```

In [73]:
```
# writing Lloyd's Algorithm for K-Means clustering.
# (This exists in various libraries, but it's good practice to write by hand.)
def init_centroids(labelled_data,k):
    """
    randomly pick some k centers from the data as starting values for centroids.
    Remove labels.
    """
    return map(lambda x: x[1], random.sample(labelled_data,k))

def sum_cluster(labelled_cluster):
    """
    from http://stackoverflow.com/questions/20640396/quickly-summing-numpy-arrays-element-wise
    element-wise sums a list of arrays. assumes all datapoints in labelled_cluster are labelled.
    """
    # assumes len(cluster) > 0
    sum_ = labelled_cluster[0][1].copy()
    for (label,vector) in labelled_cluster[1:]:
        sum_ += vector
    return sum_

def mean_cluster(labelled_cluster):
    """
    computes the mean (i.e. the centroid at the middle) of a list of vectors (a cluster).
    take the sum and then divide by the size of the cluster.
    assumes all datapoints in labelled_cluster are labelled.
    """
    sum_of_points = sum_cluster(labelled_cluster)
    mean_of_points = sum_of_points * (1.0 / len(labelled_cluster))
    return mean_of_points
```

In [68]:
```
def form_clusters(labelled_data, unlabelled_centroids):
    """
```

```
    given some data and centroids for the data, allocate each datapoint
    to its closest centroid. This forms clusters.
    """
    # enumerate because centroids are arrays which are unhashable,
    centroids_indices = range(len(unlabelled_centroids))

    # initialize an empty list for each centroid. The list will contain
    # all the datapoints that are closer to that centroid than to any other.
    # That list is the cluster of that centroid.
    clusters = {c: [] for c in centroids_indices}

    for (label,Xi) in labelled_data:
        # for each datapoint, pick the closest centroid.
        smallest_distance = float("inf")
        for cj_index in centroids_indices:
            cj = unlabelled_centroids[cj_index]
            distance = np.linalg.norm(Xi - cj)
            if distance < smallest_distance:
                closest_centroid_index = cj_index
                smallest_distance = distance
        # allocate that datapoint to the cluster of that centroid.
        clusters[closest_centroid_index].append((label,Xi))
    return clusters.values()

def move_centroids(labelled_clusters):
    """
    returns a list of centroids corresponding to the clusters.
    """
    new_centroids = []
    for cluster in labelled_clusters:
        new_centroids.append(mean_cluster(cluster))
    return new_centroids

def repeat_until_convergence(labelled_data, labelled_clusters, unlabelled_centroids):
    """
    form clusters around centroids, then keep moving the centroids
    until the moves are no longer significant, i.e. we've found
    the best-fitting centroids for the data.
    """
    previous_max_difference = 0
    while True:
        unlabelled_old_centroids = unlabelled_centroids
        unlabelled_centroids = move_centroids(labelled_clusters)
        labelled_clusters = form_clusters(labelled_data, unlabelled_centroids)
        # we keep old_clusters and clusters so we can get the maximum difference
        # between centroid positions every time. we say the centroids have converged
        # when the maximum difference between centroid positions is small.
        differences = map(lambda a, b: np.linalg.norm(a-b),unlabelled_old_centroids,unlabelled_centroids)
        max_difference = max(differences)
        difference_change = abs((max_difference-
previous_max_difference)/np.mean([previous_max_difference,max_difference])) * 100
        previous_max_difference = max_difference
        # difference change is nan once the list of differences is all zeroes.
        if np.isnan(difference_change):
            break
    return labelled_clusters, unlabelled_centroids
```

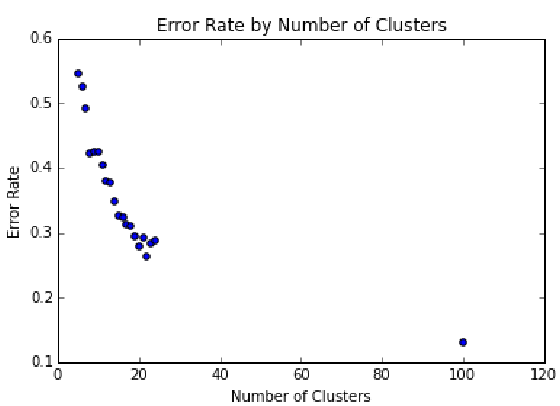
In [69]:
```
def cluster(labelled_data, k):
    """
    runs k-means clustering on the data. It is assumed that the data is labelled.
    """
    centroids = init_centroids(labelled_data, k)
    clusters = form_clusters(labelled_data, centroids)
    final_clusters, final_centroids = repeat_until_convergence(labelled_data, clusters, centroids)
    return final_clusters, final_centroids
```

In [70]:
```
def assign_labels_to_centroids(clusters, centroids):
    """
    Assigns a digit label to each cluster.
    Cluster is a list of clusters containing labelled datapoints.
    NOTE: this function depends on clusters and centroids being in the same order.
    """
    labelled_centroids = []
    for i in range(len(clusters)):
        labels = map(lambda x: x[0], clusters[i])
        # pick the most common label
        most_common = max(set(labels), key=labels.count)
        centroid = (most_common, centroids[i])
```

```
        labelled_centroids.append(centroid)
    return labelled_centroids
```

In [71]:
```
def classify_digit(digit, labelled_centroids):
    """
    given an unlabelled digit represented by a vector and a list of
    labelled centroids [(label,vector)], determine the closest centroid
    and thus classify the digit.
    """
    mindistance = float("inf")
    for (label, centroid) in labelled_centroids:
        distance = np.linalg.norm(centroid - digit)
        if distance < mindistance:
            mindistance = distance
            closest_centroid_label = label
    return closest_centroid_label

def get_error_rate(digits,labelled_centroids):
    """
    classifies a list of labelled digits. returns the error rate.
    """
    classified_incorrect = 0
    for (label,digit) in digits:
        classified_label = classify_digit(digit, labelled_centroids)
        if classified_label != label:
            classified_incorrect +=1
    error_rate = classified_incorrect / float(len(digits))
    return error_rate
```

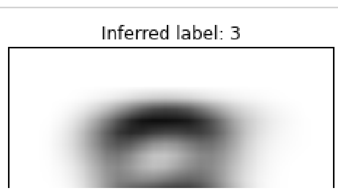
In [26]:
```
error_rates = {x:None for x in range(5,25)+[100]}
for k in range(5,25):
    trained_clusters, trained_centroids = cluster(training, k)
    labelled_centroids = assign_labels_to_centroids(trained_clusters, trained_centroids)
    error_rate = get_error_rate(validation, labelled_centroids)
    error_rates[k] = error_rate

# Show the error rates
x_axis = sorted(error_rates.keys())
y_axis = [error_rates[key] for key in x_axis]
plt.figure()
plt.title("Error Rate by Number of Clusters")
plt.scatter(x_axis, y_axis)
plt.xlabel("Number of Clusters")
plt.ylabel("Error Rate")
plt.show()
```

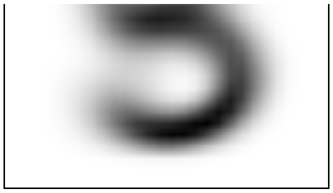


In [74]:
```
k = 16
trained_clusters, trained_centroids = cluster(training, k)
labelled_centroids = assign_labels_to_centroids(trained_clusters, trained_centroids)
```

```
-c:52: RuntimeWarning: invalid value encountered in double_scalars
```

In [78]:
```
for x in labelled_centroids:
    display_digit(x, title=x[0])
```

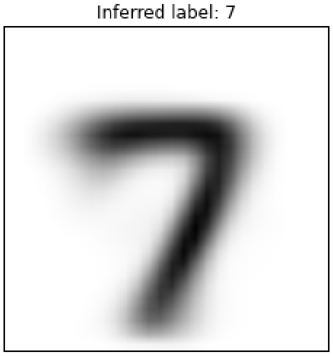




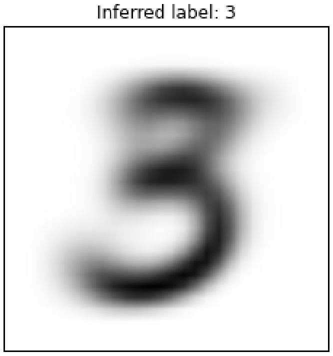
Inferred label: 9

Inferred label: 1

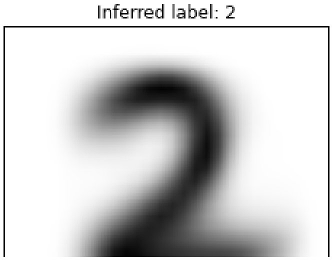
Inferred label: 7



Inferred label: 3



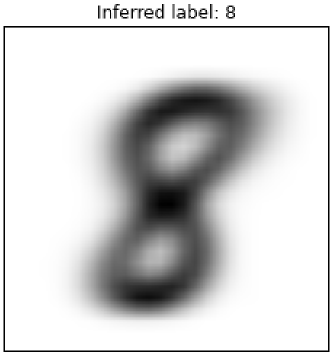
Inferred label: 2



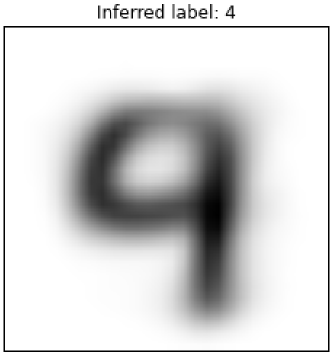
Inferred label: 1

Inferred label: 7

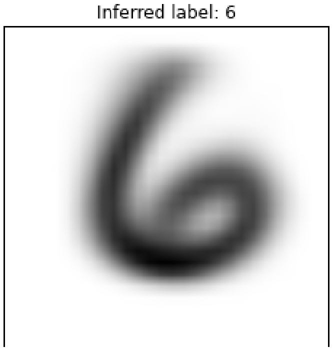
Inferred label: 8



Inferred label: 4



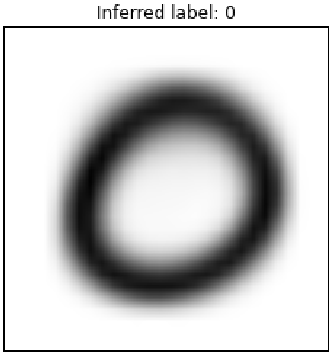
Inferred label: 6



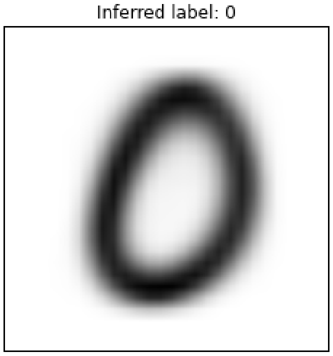
Inferred label: 2

Inferred label: 6

Inferred label: 0



Inferred label: 0



Inferred label: 0