Mini Project 1

Fezile Manana, Dushan Terzikj

March 17, 2019

Contents

Chapter 1

Overview

1.1 Data Set

The data set is a matrix of size 2000×240 which encodes 2000 normalized handwritten digits, 0 - 9. The 240 features of the vector are the grayscale values, ranging from 0 - 6, of a 15×16 image panel of the digit. There are 200 samples for each digit ranging from 0 - 9.

1.2 Task Description

Pick one of the digits (e.g. the "ones"), which gives you a dataset of 200 image vectors. Carry out a K-means clustering on your chosen sample, setting K = 1, 2, 3, and 200 in four runs of this algorithm. Generate visualizations of the images that are coded in the respective codebook vectors that you get (for the K = 200 case, only visualize a few). Discuss what you see. Your discussion should include (but not be restricted to) answers to the questions (1) what is the mathematical nature of the codebook image for the case K = 1? (2) what is the mathematical nature of the codebook images for the case K = 200? (give formulas).

1.3 Task Objectives

The goal of this project is to implement K-means clustering on the data set and analyse the effects of the number of initial clusters on the codebook vectors.

Chapter 2

Procedure

2.1 Methods

For the purpose of this project the data set of "ones" was chosen for analysis. There is no particular reason why the "ones" were chosen. The way we worked is that we initialized K to 1, and then 2, 3 and 200. The results are analyzed below and the algorithm is explained below as well.

2.1.1 Algorithm Implementation - Initializations

Listing 2.1: Set up parameters

```
#[a, b) is the interval over which we obtain our training points
a = 200
b = 400
m = b - a # number of training examples
n = pixel_data.shape[1] # number of features, 240
K = 1 # set the number of clusters
```

In Listing 2.1 we define the interval over which we will obtain our training points. The interval $[200, 400)$ defines the training points for the set of "ones". The independent variable K is the number of clusters we wish to distinguish in the first run of the algorithm. As mentioned in section 2.1 the K is later changed to 2, then 3, and 200.

Listing 2.2: Alternating cluster initialization

```
clusters = defaultdict(list)
def alternating_bins_initialisation():
    for i in range(a, b): # selecting sevens as data set
        clusters[i \% K].append(pixel_data[i])
```

The method used for cluster initialisation is an alternating cluster assignment scheme, where each proceeding vector is assigned to neighbouring cluster above it.

Listing 2.3: In order initialization

```
def in_order_initialisation():
    i = 0
    for k in range(K):
        while(len(clusters[k]) < m/K and i < m):
            clusters[k].append(pixel_data[a + i])
            i += 1
```

In Listing 2.3 a different type of initialization is presented. This "in-order" type of initialization assigns the first $\frac{m}{K}$ vectors the to first cluster and then the second $\frac{m}{K}$ vectors (if they exists, i.e. if $K > 1$) to the second cluster, and so on.

Listing 2.4: Unbalanced initialization

```
def unbalanced_initialisation(offset):

    """
    :param offset: the first offset vectors are put in the first cluster
    :type offset: int
    """

    if(K > 1):
        for i in range(a, a + offset):
            clusters[0].append(pixel_data[i])
        # the remaining vectors are spread evenly in the remaining clusters
        j = a + offset
        for k in range(1, K):
            while(len(clusters[k]) < (m - offset)/(K-1) and j < b):
                clusters[k].append(pixel_data[j])
                j += 1
    else:
        print("cannot have unbalanced initialisation with one cluster")
```

In Listing 2.4 unbalanced initialization is presented. This type unevenly distributes vectors into clusters by placing $offset$ number of vectors in the first cluster and evenly spreads the remaining vectors in the remaining clusters. This type of initialization can only be used if $K > 1$ for trivial reasons.

These initialization types are used when running the K-means algorithm on our chosen data set. Only alternating and in-order initialization will

be discussed in this paper. If you want to see the effects of unbalanced initialization, please use our source code.

2.1.2 Algorithm Implementation - Codebook Vectors

Listing 2.5: Calculate Codebook Vectors

```
1  def calculate_cb_vecs():
2      # setup the codebook vectors
3      cb_vectors = np.zeros([n * K]).reshape(K, n)
4      # calculate codebook vectors
5      for i in range(K):
6          sum = np.zeros([n], dtype=np.uint).reshape(1, n)
7          for vector in clusters[i]:
8              sum += vector
9          # divide the sum of the vectors by the size of the cluster
10         cb_vectors[i] = np.divide(sum, len(clusters[i]))
11 return cb_vectors
```

The function *calculate_cb_vecs()* in Listing 2.5 returns a $K \times n$ matrix where K is the number of clusters and n is the feature length of the vectors.

The general formula for computing the codebook vectors takes the form:

$$\mu_j = |S_j|^{-1} \sum_{x \in S_j} x$$

where S_j is a cluster in the set of K clusters ($j = 1, ..., K$). For $K = 1$, μ_j is the average vector of the entire data set. However, for values of K greater than 1, K-means will produce codebook vectors for the average of K distinctive sets.

For brevity the algorithm for carrying out the K-means is omitted from this report (find attached python file).

2.2 Results

2.2.1 K=1

For $K = 1$, K-means returns the average vector for all the ones (i.e. what a "one" looks like on average). All types of initialization look the same in this case. Figure 2.1 illustrates the codebook vector for $K = 1$ plotted on a 15×16 grid.

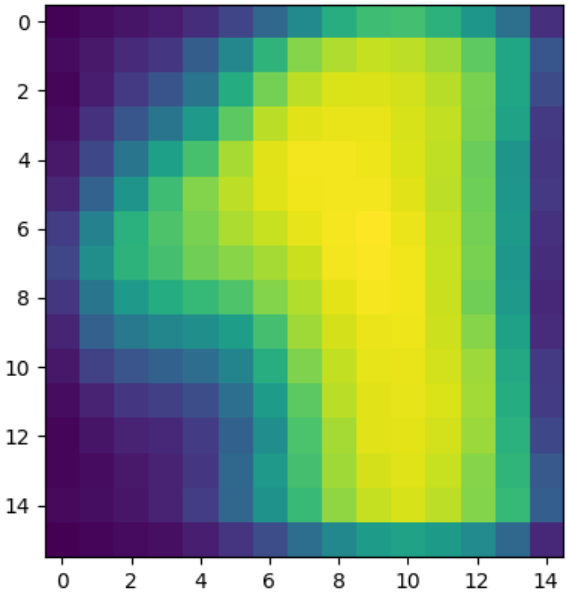

Figure 2.1: Codebook vector when K=1

2.2.2 K=2

There is not much improvement in distinguishing differently written 1s when $K = 2$ (check figures 2.2 to 2.5). You can see that using alternating initialization (figures 2.2 and 2.3) can give us better perspective to distinguish different written 1s. The algorithm here iteratively computes 2 centres, since we have 2 clusters. They approach convergence, but there is still a lot of noise. The in-order samples (figures 2.4 and 2.5) look very similar because our training points are ordered in a way that similar vectors precede and succeed each other. Due the the initialization approach in Listing 2.3 the means are very similar.

2.2.3 K=3

Comparing figures 2.6 and 2.7 with 2.2 and 2.3 respectively, we can clearly see how the plots become more "sharp". The reason for this is because we have 3 clusters instead of 2, i.e., the convergence can be spotted more clearly. Very similar argument can be given regarding in-order initialization as it was given in section 2.2.2. In other words, the approach of initializing the clusters makes the means of both codebook vectors very similar.

2.2.4 K=200

In this case the plots of the codebook vectors become very "sharp" (figures 2.10 and 2.11). Since we only have 200 training points for the 1s, one codebook vector is calculated for each training point. In this case, initializing the clusters with altering or in-order initializtion does not make a difference, because when $K = 200$ both algorithms yield the same result.

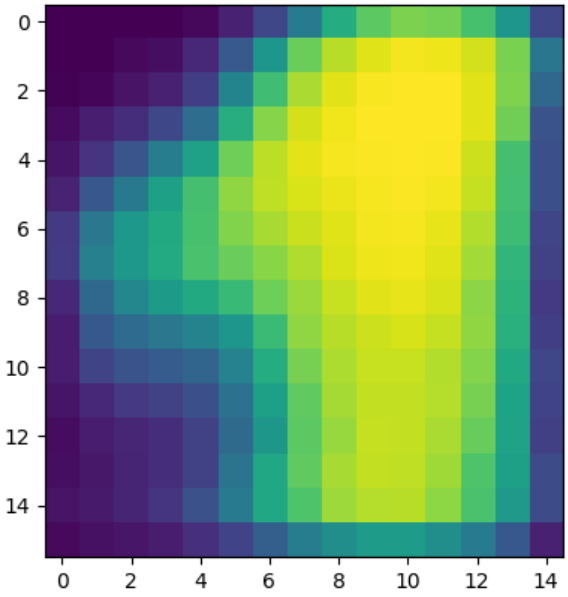

Figure 2.2: First codebook vector when K=2, using alternating initialization

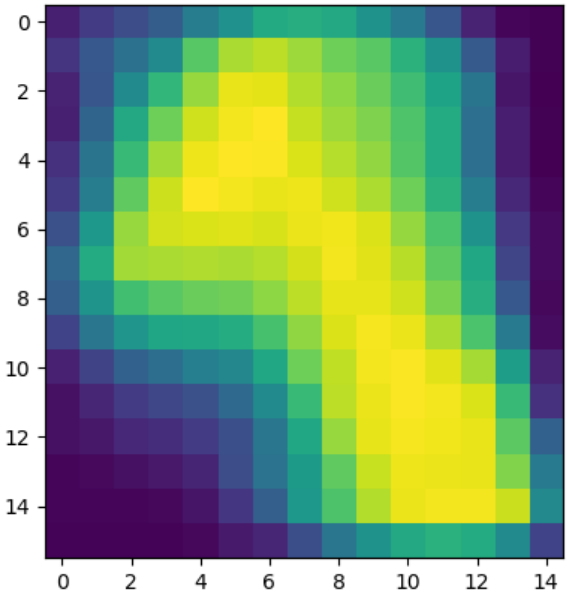

Figure 2.3: Second codebook vector when K=2, using alternating initialization

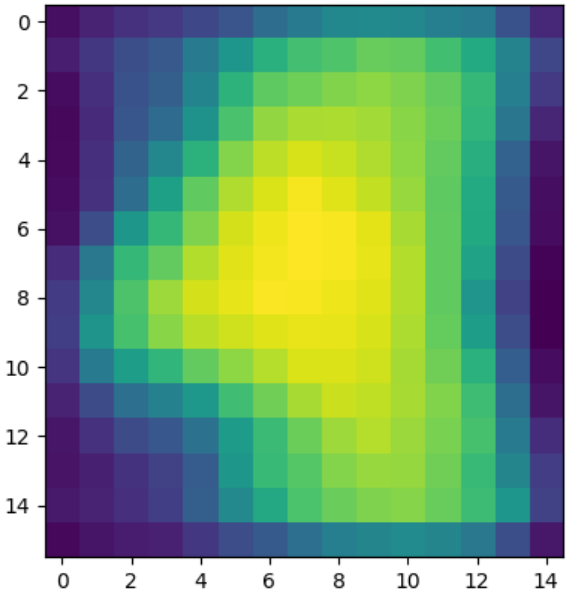

Figure 2.4: First codebook vector when K=2, using in-order initialization

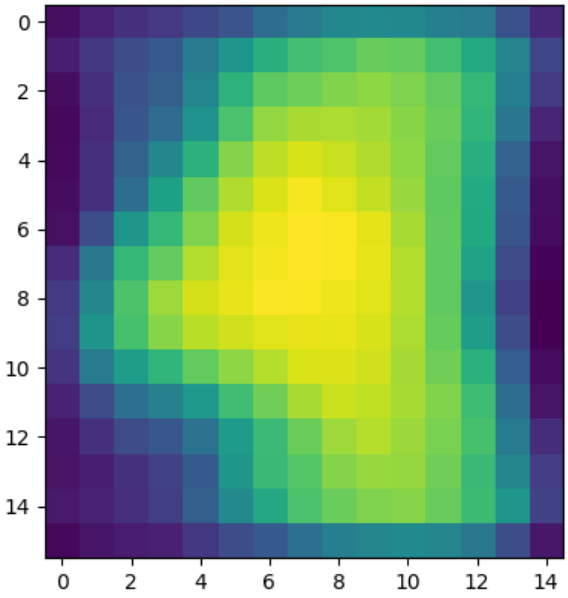

Figure 2.5: Second codebook vector when K=2, using in-order initialization

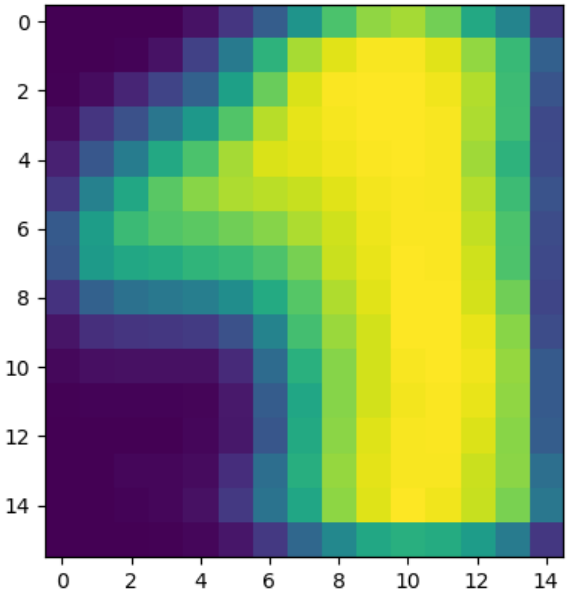

Figure 2.6: First codebook vector when K=3, using alternating initialization

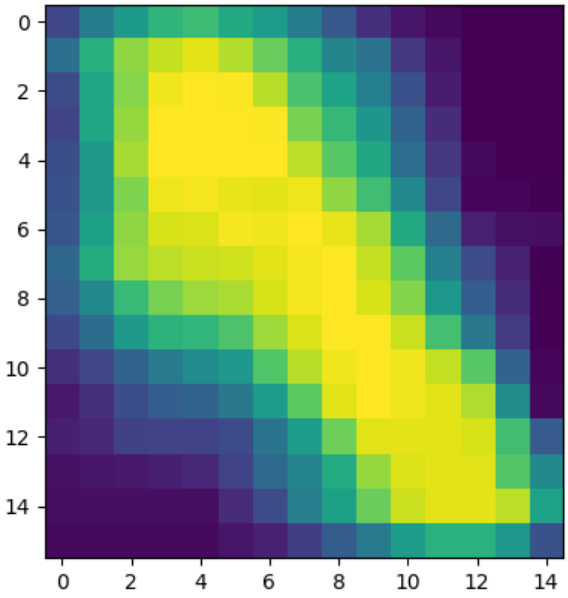

Figure 2.7: Second codebook vector when K=3, using alternating initialization

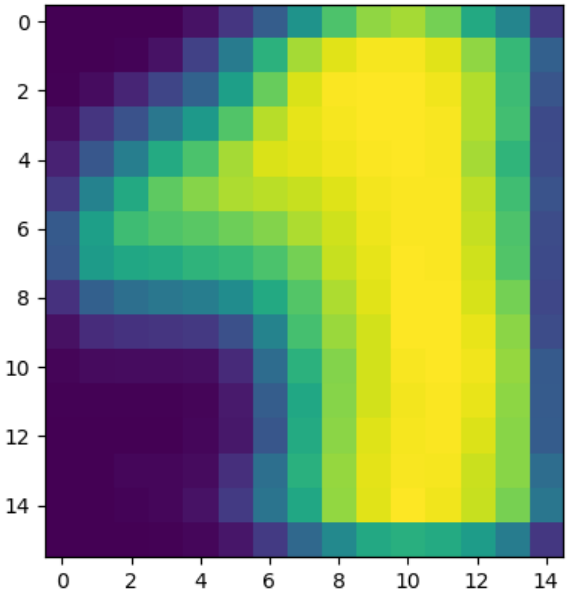

Figure 2.8: First codebook vector when K=3, using in-order initialization

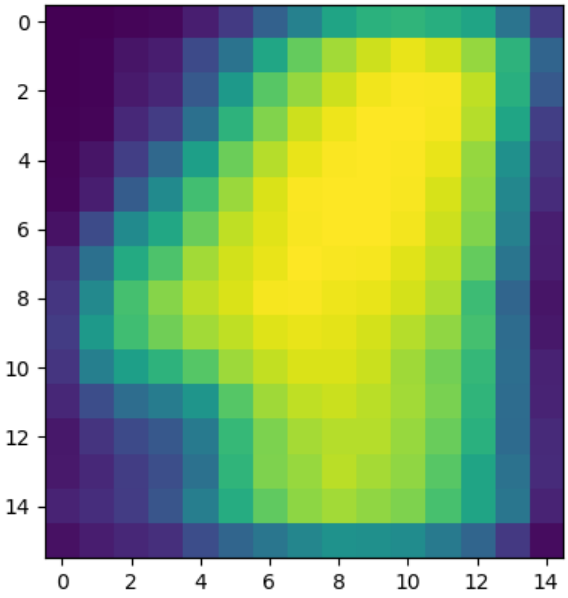

Figure 2.9: Second codebook vector when K=3, using in-order initialization

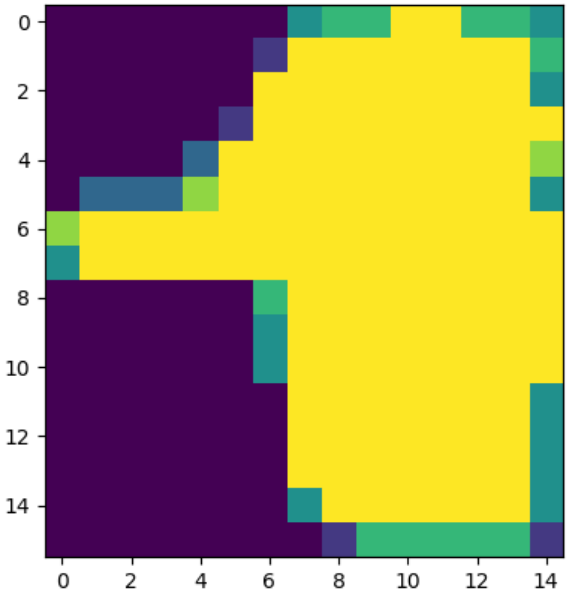

Figure 2.10: Random codebook vector when K=200, using alternating initialization

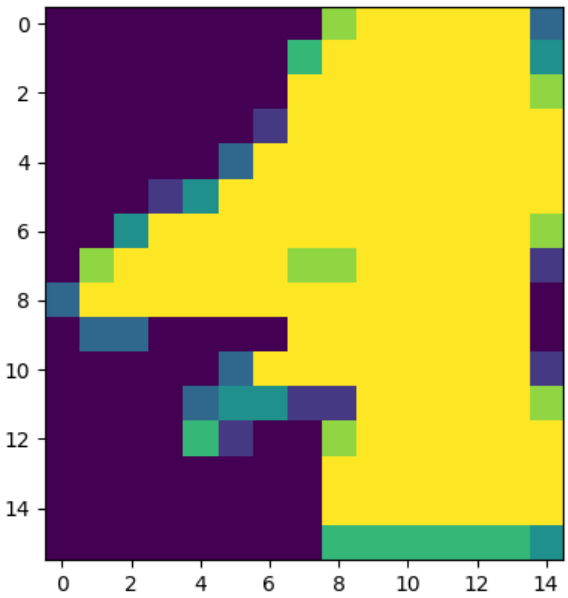

Figure 2.11: Random codebook vector when K=200, using in-order initialization

Chapter 3

Summary

3.1 Conclusion

This paper is just a glimpse of what is happening with the clusters when K and the initialization algorithms vary. Our training points set contains different type of handwritten 1s and when one tries to cluster those training points into more clusters, the more means of different types are yielded. In other words, as K approaches the size of the training set, we can see more clear categorization of 1s.

Additionally, different types of initialization in this case yield different clusters. The reason for this is that our training set is not that big (only 200). In the first three cases where $K << 200$ we can see different clusters for different initialization. But once $K = 200$, categorization of these 1s becomes more clear and clusters do not differ depending on the type of initialization.