# COMPGI07: Assignment 2

Antonio Remiro Azócar, MSc Machine Learning

25 January 2017

## Note

This assignment has been undertaken individually. A 9 day extension was granted due to my former partner's abandonment of the course and coursework.

## README - Files

### Exercise 1

**1.1.1:** `new_kmeans.r`, `chooseCentroids.r`, `squared_Distance.r`
**1.1.2:** `generate_data.r, new_kmeans.r, new_kmeans2.r, chooseCentroids.r, squared_distance.r`, `compute_occe.m`, `compute_simple_error.m`, `gen2data.m`, `new_kmeans.m`, `compGI07_A1_2.m`
**1.1.3:** `iris.csv`, `compGI07_A1_3.m`, `new_kmeans.m`, `compute_simple_error.m`, `compute_occe.m`

### Exercise 2

`my_pca.m`, `iris.csv`, `new_kmeans_clust.m`, `compGI07_A2_3.m`,`compGI07_A2_4.m`, `compGI07_A2_5.m`, `compute_occe.m`, `compute_simple_error.m`.

### Exercise 3

`compGI07_A3.m`, `dtest123.dat`, `dtest123.dat`, `extendedtestperceptron.m`, `poly_kernel.m`, `ziptest.dat`, `compGI07_A3_2.m`, `ziptrain.dat`, `testperceptron.m`, `trainperceptron.m`, `gaussian_kernel.m`

# 1 K-means

## 1.1 Practical

Programming language: R
**(1.)**
The $k$-means clustering algorithm is implemented as the function `NewKmeans`, saved in the file `new_kmeans.r` (see Appendix for code). This function intakes a series of data points and the number of clusters. The series of data points are presented as a matrix (data frame) where each row represents a data point. The default number of clusters is four. The program is initialised by choosing random centres $\mathbf{c}_1, ..., \mathbf{c}_k$. In order to do this, the function `randCentroids` is implemented.

The `randCentroids` function, saved in the file `choose_centroids.r`, takes as inputs the data frame with the series of data points and the number of clusters. It firstly initialises an empty matrix. As many centroids as clusters as we want to have in our algorithm are created. In order to do this, random points are initialised at each iteration, $k$ are selected, and the $x$, $y$ co-ordinates of each centroid in the array are stored. The function `randcentroids` returns the array with all the $x$, $y$ coordinates of the initially specified centroids.

We now return to our main `NewKmeans` implementation. Once the centroids have been chosen randomly, a matrix is created to keep track of which data point belongs to each cluster. A boolean variable `has_changed`, which keeps track of the changes in any of the points, is initialised. Subsequently, the clusters that a data point belongs to are updated using the following assignment step: each data point is assigned to the closest cluster centre i.e. $r_{ij} = 1$ if $j = \text{argmin}_{1 \leq s \leq k} ||\mathbf{x}_i - \mathbf{c}_s||^2$, $r_{ij} = 0$ otherwise. In this step's implementation, the distance metric `my_distance` is initialised to infinity. This variable stores the distance of an $i$-th point to a $j$-th point. The variable `my_new_index` updates the index of the cluster that the $i$-th observation

belongs to. To compute the distance metric, the function `squaredDistance` is implemented (saved in the file `squared_Distance.r`). This function calculates the 2-norm (Euclidean norm) distance between two input vectors.

After the assignment step, an update step is performed where the new centroids of the clusters are computed (using $\mathbf{c}_j = \frac{\sum_{i=1}^{l} r_{ij}\mathbf{x}_i}{\sum_{i=1}^{l} r_{ij}}$, $j = 1, ..., k$). The assignment and update procedures are repeated until there is no further change in the assignments of the data points to the clusters. i.e. `has_changed = FALSE`. Subsequently, the program `NewKmeans` returns a data frame with the values of the datapoints.

**(2.)** The file `generate_data.r`, replicates the MATLAB file provided for data generation `genData2.mat`. The random seed is fixed at 8 (`set.seed(8)`) in order to replicate results. Once the data has been generated, the K-means algorithm (`NewKmeans`) is tested on the three sets of data produced. This can be seen in file `generate_data.r`. The clustered data is plotted in Figure 1.1, using a different colour/point shape for the points in each cluster. Additional larger markers are used to indicate both centres of the clusters. To compute both the original and new cluster centroids, we implement the function `NewKmeans2`, saved in the file `new_kmeans2.r`. This is very similar to `NewKmeans`, but returning the position of the cluster centroids instead of the data points. In `generate_data.r`, the position and cluster assignment of the data points is saved as `kmm` and that of the cluster centroids is saved as `df`.

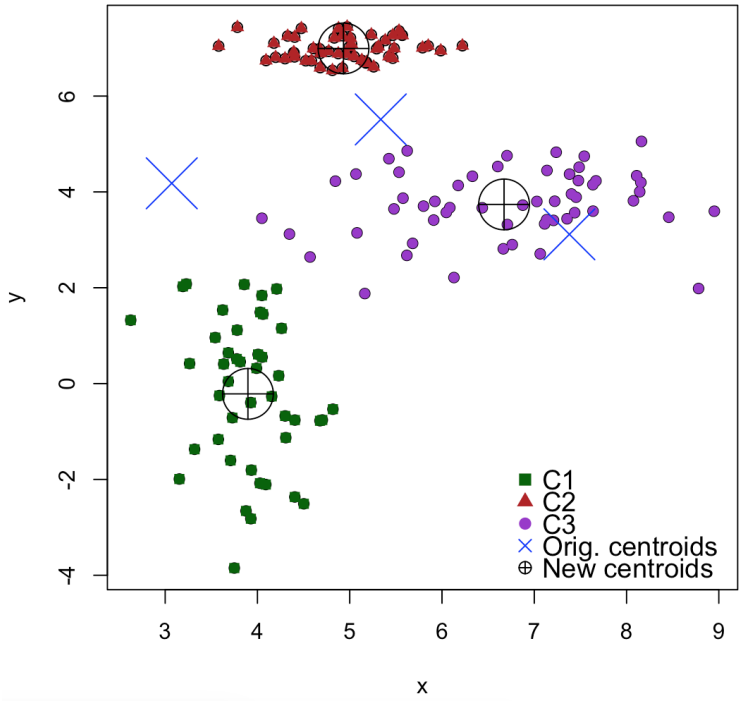


Figure 1.1: Plot of the clustered data for Exercise 1.2

In Figure 1, Cluster 1 (representative of $S_1$) is characterised by green square points, Cluster 2 (representative of $S_2$) is characterised by red triangular points and Cluster 3 (representative of $S_3$) is characterised by purple circular points. Whereas the smaller markers are data points, the larger markers are cluster centres. The blue crosses correspond to the original random centroids, whose coordinates are $(x, y) = (3.07, 4.18)$ for C1, $(7.38, 3.11)$ for C2 and $(5.34, 5.51)$ for C3. The black 'bull's eyes' correspond to the recalculated cluster centroids, whose coordinates are $(x, y) = (3.90, -0.22)$ for C1, $(4.93, 6.99)$ for C2 and $(6.67, 3.74)$ for C3. These values are stored in data frame `df` (with column names Original.1, Original.2, Centroids.1, Centroids.2). We can observe how the initialised clusters converge into their mean positions in Figure 1. The algorithm appears to work quite well, classifying data points from different 'true' partitionings into different clusters. Note that we have plotted the graph for `rng(8)`. Where as this setup gives good results, more misclassified points arise with other random seeds.

For the rest of this exercise, the programming language used is MATLAB.
**(2. - continued)** For the `occe` computation refer to the file `compute_occe.m` For the simple error computation, refer to `compute_simple_error.m`. This function is designed to work within the `compute_occe.m` environment.

**Computing the occe:** The function `compute_occe.m` takes as input a cell array of cluster group allocations labelled `batches`. In this case, each batch contains coordinate information for its corresponding cluster. The function also inputs the variable `true_splits`. This is a cell array containing the coordinate information of the elements in each true partitioning. Firstly, the function computes all the permutations of the input cluster assignments. The provided method of permuting the `batches` has drawn inspiration from the MATLAB `perms` function, particularly in its use of a helper for recursion `Psmall`. The permutations are stored as the variables `permutes`. Bare in mind that a permutation of a vector of $n$ rows ($n$ data points) will result in a matrix of $n!$ rows and $n$ columns with all possible permutations of $n$ elements. The variables `simple_error`, `CLUSTER`, `ERROR` are initialised. `simple_error` keeps track of the simple error computation for each iteration. `CLUSTER` is a cell array which keeps track of the cluster permutations (for each `run`, `CLUSTER{k} = permutesrun,k`, where `k` indexes the cluster). Recall that we use the following equation to compute the simple error $e$,

$$e = \frac{|\{\mathbf{x}|(\mathbf{x} \in C_1 \text{ and } \mathbf{x} \notin S_1) \vee ... \vee (\mathbf{x} \in C_k \text{ and } \mathbf{x} \notin S_k)\}|}{l},$$

The cell array `ERROR`, computed via `ERROR{k} = !ismember(CLUSTER{k},true_splits{k},'rows');`, effectively calculates the number of times an element is in an indexed permutation but not in the true partitioning (row mismatches). i.e. the sum of these mismatches gives the count of $\{\mathbf{x} \in C_i \text{ and } \mathbf{x} \notin S_i\}$, for a given $i = 1, 2, 3$. This corresponds to the numerator in our expression for the simple error calculation. Subsequently, the implemented function `compute_simple_error.m` is called to keep track of the simple error for each run. This function works solely as a helper function and is designed to work within the `compute_occe.m` environment. It intakes the `ERROR` cell array and outputs `simple_error` for a given iteration. It computes `sum(ERROR{k})` for each $k = 1, ..3$ and sums such `error_component` to `tot_error`. `tot_error` corresponds to the numerator in the expression above. This is divided by $l$ (`tot_points`), the total number of points (150), to give $e$ for a given iteration. Returning to the `compute_occe.m` function, recall that the `occe` is given by,

$$occe = \min_{\mathbf{p} \in \mathbb{P}_k} \frac{|\{\mathbf{x}|(\mathbf{x} \in C_{p_1} \text{ and } \mathbf{x} \notin S_1) \vee ... \vee (\mathbf{x} \in C_{p_k} \text{ and } \mathbf{x} \notin S_k)\}|}{l}.$$

Hence, it can be computed keeping track of the minimum simple error. i.e. `if simple_error(run) < min_simple_error`, `min_simple_error = simple_error(run)`. After `tot_runs=n` iterations, the `occe` is given by the stored minimum simple error. Scripts with the utilised functions are submitted in the appendix. Note that the functions have been tuned for this specific set of problems; more could be done to make them more generalisable.

**Main program:** Data for the assignment is provided in the file `gen2data.m`. The main function has been saved as `compGI07_A1_2.m`. We make use of newly implemented function `new_kmeans.m` for the $k$-means computation. This function is very similar to `new_kmeans.r`, implemented for the previous subsection - it is essentially a 'translation' of such into MATLAB. A more in-depth description of the code is presented in the comments in the Appendix.

The following set of results (Figure 1.2) are displayed for the average `occe` and its standard deviation over 100 trials:

```
>> compGI07_A1_2
Mean after 100 trials: 0.079
Standard deviation after 100 trials: 0.122
```

Figure 1.2: Mean and standard deviation for 100 trials displayed on screen after running `compGI07_A1_2.m`

Over 100 runs, the mean `occe` is rather low, suggesting a good performance on the data provided by `gen2data.m`. Note however that the standard deviation is significantly higher than the mean. This corresponds to very high values of the `occe` for some trials (since we cannot have negative errors). This is due to $k$-means sometimes converging to 'lower-quality' centroids and is triggered by using initial cluster centroids with random positions.

**(3.)** The iris data was loaded as a .csv file. We followed the same procedure as for **1.1.2**. The main function has been saved as `compGI07_A1_3.m`. A more in-depth description of this code is presented in the comments in the appendix.

The following set of results (Figure 1.3) are obtained for the average `occe` and its standard deviation over 100 trials:

```
>> COMPGI07_A1_3
Mean after 100 trials: 0.186
Standard deviation after 100 trials: 0.142
```

Figure 1.3: Mean and standard deviation for 100 trials displayed on screen after running `compGI07_A1_3.m`

## 1.2 Questions

**1.)** Let $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_l\}$ be a set of points. The centroid $\mathbf{c}_j$ of all the data points belonging to cluster $j$ is computed as:

$$\mathbf{c}_j = \frac{\sum_{i=1}^{l} r_{ij}\mathbf{x}_i}{\sum_{i=1}^{l} r_{ij}}, \qquad\qquad j = 1, ..., k,$$

where the denominator is equal to the number of points assigned to the cluster $j$. For a given cluster $j$:

$$\sum_{i=1}^{l} |\mathbf{x}_i - \mathbf{x}|^2 = \sum_{i=1}^{l} |\mathbf{x}_i + \mathbf{c}_j - \mathbf{c}_j - \mathbf{x}|^2 = \sum_{i=1}^{l} |\mathbf{x}_i + \mathbf{c}_j|^2 + 2(\mathbf{c}_j - \mathbf{x}) \cdot \sum_{i=1}^{l} (\mathbf{x}_i - \mathbf{c}_j) + l|\mathbf{c}_j - \mathbf{x}|,$$

where $\mathbf{x}$ is any point in the cluster. Now, $\mathbf{c}_j$ is the centroid of the cluster; therefore $\sum_{i=1}^{l}(\mathbf{x}_i - \mathbf{c}_j) = 0$. Hence,

$$\sum_{i=1}^{l} |\mathbf{x}_i - \mathbf{x}|^2 = \sum_{i=1}^{l} |\mathbf{x}_i - \mathbf{c}_j|^2 + l||\mathbf{c}_j - \mathbf{x}||.$$

i.e. the sum of squared distances from $\mathbf{x}_i$ to a point $\mathbf{x}$ is equivalent to the sum of squared distances from $\mathbf{x}_i$ to the centroid added to the number of points in the set multiplied by the distance between the centroid and point $\mathbf{x}$. $l||\mathbf{c}_j - \mathbf{x}||$ is positive without exception. Therefore, the left hand side above is minimised when we set $\mathbf{x} = \frac{\sum_{i=1}^{l} \mathbf{x}_i}{n}$; the definition of the centroid. Consequently, $\mathbf{x}$ being a centroid minimises the sum of squared distances, $\sum_{i=1}^{l} |\mathbf{x}_i - \mathbf{x}|^2$.

**2.)** Firstly, we prove that $k$-means converges. In other to prove this we refer to a physical analogy and use a Lyapunov function. If a problem has a Lyapunov function, it is "well-behaved" in the sense that when iterating, its objective function decreases monotonically to a certain minimum value (converges). In this case, the optimisation objective is to minimise the non-negative distortion function $J = \sum_{\mathbf{x}} ||\mathbf{x} - \mathbf{c}_i||^2$ over $k$ clusters, where $\mathbf{x}$ is a given point in the dataset and $\mathbf{c}_i$ its corresponding centroid.

Alternatively, imagine coupling a spring between both points. A spring will have non-negative energy $E = \frac{1}{2}Kd^2$, where $K$ is a measure of the spring's stiffness and $d$ is the distance between the points. The energy is proportional to the length of spring joining both points squared. In the assignment step, each data point is assigned to its closest cluster centre. In the update step, the cluster means are recomputed to be the new centres of the clusters. Both assignment and update steps reduce the energy/distortion function. Why? A point is only assigned a cluster centroid given an energy decrease; hence each update step may only decrease the energy, minimising that of the spring. Therefore, the energy/distortion function is bounded from below and the condition for a Lypaunov function is satisfied. The energy converges to a local minimum as it is continually reduced. It is sufficient to set a given convergence threshold for the energy/distortion function for iterations to stop.

We now prove that $k$-means converges in a finite number of steps. We have $n$ data points. These can be partitioned into $k$ clusters in no more than $k^n$ ways. We call each partition a 'cluster assignment'. There may be a very large but always finite number of cluster assignments. For a given algorithm iteration, the energy/distortion function can either only decrease or reach convergence. If convergence is reached, the new cluster assignment will be the same as the previous one. On the other hand, if $J$ or $E$ have a lower cost, the new cluster assignment will differ. Now, since there are finitely many cluster assignments, an iteration must at some point move into a cycle. Such cycle cannot be greater than one i.e. a given cluster assignment configuration cannot be revisited. Why? If there is a cycle greater than one, a cluster assignment with a cost lower than itself would be encountered. Hence, the length of the cycle is exactly one and $k$-means converges in a finite number of steps.

## 1.3 Extensions

**1.)**

**Background:**

The 'hard' $k$-means algorithm uses only square Euclidean distance as a metric of discrepancy in the minimisation step. This presents a problem of robustness when dealing, for example, with non-quantitative data or with very large distances between an outlier and another point (the squared Euclidean distance gives the largest influence to greater distances). To resolve the problem, we may generalise by using the $p$-discrepancy $d_p$ as a measure of dissimilarity between points, where,

$$d_p(\mathbf{c}, \mathbf{x}) = \sum_{i=1}^{n} |c_i|^p - |x_i|^p - p(c_i - x_i)sign(x_i)|x_i|^{p-1},$$

with $d_p$ reduced to the Euclidean distance for $p = 2$. Both $k$-means and this modification are partitional, however whereas $k$-means attempts to minimise total squared error, the new method attempts to minimise $d_p$. A **medoid** is defined as the most centrally located point in the data set. $\{c_1, ..., c_n\}$ may correspond to the medoids of the currently designated clusters. It is an actual member of the set of data points, (note 'hard' $k$-means centroids are not necessarily within the set) which has minimal average discrepancy to all data points.

**Algorithm:**

**I. Initialisation:** One can randomly select $k$ of the $n$ data points in the cluster as supposed medoids $\{c_1, ..., c_k\}$.

**II. Assignment/Build step:** A given cluster designation $C(i)$, $i = 1, ..., k$ is associated to a potential medoid. Essentially, each remaining data point is assigned to the cluster with the closest medoid. For a given cluster assignment $C(i)$, find:

$$i_j^* = \text{argmin}_{\{i:C(i)=j\}} \sum_{C(i')=j} d_p(\mathbf{x}_i, \mathbf{x}_{i'}),$$

where $i_j^*$ indexes the observation minimising the $p$-discrepancy from point $\mathbf{x}_i$ to other points in the cluster $\mathbf{x}_{i'}$. Note that there is now no necessity to compute cluster centres, we simply need to track the indices. In this step, $\mathbf{c}_i = \mathbf{x}_{i_i^*}$, $i = 1, ..., k$, make up the current estimates of cluster medoids.

**III. Update step:** We proceed to swap each medoid $\mathbf{c}_i$ with each data point $\mathbf{x}_i$ associated to it, calculating:

$$C(i) = \text{argmin }_{1 \leq j \leq k} d_p(\mathbf{c}_j, \mathbf{x}_i).$$

This corresponds to minimising the total cost ($p$-discrepancy) given a set of cluster medoids $\{\mathbf{c}_i, ..., \mathbf{c}_k\}$. The medoid $\mathbf{c}_j$ with lowest configuration cost is selected.

**IV. Repeat until convergence:** Repeat alternating steps 2 and 3 until the medoid assignments are invariant. **Correctness:** The medoids are updated by moving clusters to positions incurring least cost by their already assigned points. Recall that in Step 2, the penalty is minimised by associating each data point with the 'closest' medoid (that minimising $p$-discrepancy). Therefore, for each cycle $t$ we have, $d_p^{(t)} \leq d_p^{(t-1)}$. We can use the reasoning employed in Exercise 1.2.2 (we have a finite number of data points $n$; these can be partitioned into $k$ clusters in no more than $k^n$ ways and we have a finite number of cluster assignments) to deduce that eventually $d_p^{(t)} = d_p^{(t-1)}$ and the algorithm converges. **Complexity:** The complexity of the assignment step for each standard $k$-means cluster scales with the number of observations assigned to it. On the other hand, for $(p, k)$-means, the assignment step has time complexity $\mathcal{O}(n^2)$ for each cluster, where $n$ is the cluster's number of data points. This is evident since for each point we need to a discrepancy with all other points in the cluster. Therefore, computing new cluster assignments $C(i)$ in the update step requires time complexity $\mathcal{O}(nk)$, finding a minimiser over $1 \leq j \leq k$.

**2.)**

We need to segment the sequence of data points $\{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_l\} \subset \mathbb{R}^n$ into $k$ segments $\mathbf{s}_j = (i_{j-1} + 1, i_j), j = 1, ..., k$. Each line segment is a straight line joining $x_{i_{j-1}+1}$ and $x_{i_j}$. We need to find the global minima of the following expression:

$$\sum_{j=1}^{k} \sum_{p=i_{j-1}+1}^{i_j} ||\mathbf{x}_p - \mathbf{c}_j||^2.$$

The expression above corresponds to the 'loss' function of all the $k$ segments; the global problem. The loss function for an individual segment is,

$$\sum_{p=i_{j-1}+1}^{i_j} ||\mathbf{x}_p - \mathbf{c}_j||^2.$$

where $\mathbf{c}_j, j = 1, ..., k$ is the centroid of a given segment. We observe that the global problem can be solved as a dynamic programming problem[1]. It satisfies the conditions required to do so. Firstly, it has an optimal substructure i.e. the global optimal solution can be constructed from optimal solutions to subproblems. Secondly, it has overlapping subproblems. i.e. the subproblems can be reused multiple times to find an optimal solution. We explain our reasoning below.

**Optimal substructure:** To illustrate this property, find a recursion relation by considering successive values of $1 < \kappa < k$ via induction. Note $k$ corresponds to the number of segments. Lets denote the segmentation error over the sequence $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_l\} \subset \mathbb{R}^n$ with $\kappa$ segments as $e_{seg}[i_k, \kappa]$. We will denote the error representing the points $\{\mathbf{x}_{i_{\kappa-1}+1}, ..., \mathbf{x}_{i_\kappa}\}$ using just the mean of the data as $E[i_{\kappa'-1} + 1, i_\kappa]$. The $\kappa = 1$ case is trivial. It simply represents $e_{seg}[i_k, 1] = E[1, i_k]$ i.e fitting one segment as a mean over all data points. If we add a second segment and $\kappa = 2$, we can calculate the segmentation error via,

$$e_{seg}[i_k, 2] = \min_{1 \leq i_{\kappa-1} \leq l} \Big( E[1, i_{\kappa-1}] + E[i_{\kappa-1} + 1, i_k] \Big).$$

More generally, we obtain the following recursion association between $e_{seg}[i_\kappa, \kappa]$ and $e_{seg}[i_{\kappa-1}, \kappa - 1]$,

$$e_{seg}[i_\kappa, \kappa] = \min_{1 \leq i_{\kappa-1} \leq i_\kappa} \Big( e_{seg}[i_{\kappa-1}, \kappa - 1] + E[i_{\kappa-1} + 1, i_\kappa] \Big).$$

This recursion relation highlights the optimal substructure of the problem. It is evident that to find an optimal solution for $\kappa$ segmentation, one must find a solution for $\kappa - 1$ segmentation beforehand. We can make use of this recursion relation to avoid descending through the recursion tree when an answer is required.

**Overlapping subproblems:** To illustrate this property, consider the previous equation. Both expressions $e_{seg}[i_{\kappa-1}, \kappa-1]$ and particularly, $E[i_{\kappa-1}+1, i_\kappa]$, are required many times for a particular $(i_{\kappa-1}+1, i_\kappa)$.

**Algorithm**: For this procedure, consider the loss function of an individual line segment:

$$L = \sum_{p=i_{j-1}}^{i_j} ||\mathbf{x}_p - \mathbf{c}_j||^2.$$

Expanding the expression gives,

$$L = \sum_{p=i_{j-1}}^{i_j} \mathbf{x}_p^T \mathbf{x}_p - 2 \cdot \mathbf{x}_p^T \mathbf{c}_j + \mathbf{c}_j^T \mathbf{c}_j$$

From the ideas drawn before, we note the loss function for a segment can be computed iteratively, via,

$$L = \mathbf{x}_{i_{j-1}}^T \mathbf{x}_{i_{j-1}} + \sum_{p=i_{j-1}+1}^{i_j} \mathbf{x}_p^T \mathbf{x}_p + \frac{(\mathbf{x}_{i_{j-1}} + \sum_{i_{j-1}+1}^{i_j} \mathbf{x}_p)^T (\mathbf{x}_{i_{j-1}} + \sum_{i_{j-1}+1}^{i_j} \mathbf{x}_p)}{i_j - i_{j-1} + 1}.$$

This constitutes the first step of our algorithm. **Step 1.** We loop over all $p = 1, ..., l$, and within this, loop over all $j = p + 1 \leq l$, to find the loss function of the segment between $p$ and $j$ using the above equation. This step has time complexity $\mathcal{O}(n^2 l^2)$. **Step 2.** Recall, our recursive relationship $e_{seg}[i_\kappa, 2] = \min_{1 \leq i_{\kappa-1} \leq l} \Big( E[1, i_{\kappa-1}] + E[i_{\kappa-1} + 1, i_k] \Big)$. For $(\kappa = 1)$, trivially for each $i \leq l$, $e_{seg}[i_k, 1] = E[1, i_k]$ (complexity $\mathcal{O}(nl)$. We can then start with $\kappa = 2$. Recall that we can compute the loss function for a given $\kappa$ using that of $\kappa - 1$. Once we have $(\kappa = 1)$, we have the elements to compute $e_{seg}[i_\kappa, 2]$ for $\kappa_2$, then similarly for $\kappa_3$, etc. This phenomenon produces $lk$ local minimisation problems. Why? We loop over each $\kappa$, with $\kappa = 2, ..., k$, then solve a minimisation problem for each segment $l$ where $i_\kappa = l$. Hence, the total complexity of this step is $\mathcal{O}(nl \times kl) = \mathcal{O}(nkl^2)$. Hence overall, the time complexity of the algorithm is $\mathcal{O}(n^2 l^2 k)$ and the algorithm is time polynomial.

[1] http://homepages.spa.umn.edu/ willmert/science/ksegments/

# 2 Principal Components Analysis

## 2.1 Practical

Programming language: MATLAB

**1.)** The PCA algorithm is implemented as the function `my_PCA`, saved in the file `my_pca.m` and displayed in the appendix. This function intakes as inputs a series of data points ($m \times n$ matrix - $m$ examples, $n$ dimensions) and the number of reduced dimensions $k$ i.e. the number of new principal components. Firstly, the covariance matrix,

$$C = \frac{1}{m} X'X,$$

is derived from the data. In the function, the covariance matrix is represented by variable `covar`. Subsequently, singular value decomposition is utilised to factorise the matrix with MATLAB built-in function `svd`. We have,

$$C = U\Sigma V'.$$

Since for the required feature map $\phi_k = (U'_1, ..., U'_k)$, and the transformed data is $\hat{X} = X\phi'_k$, we compute `transformed_data = X*u(:,1:pc_no)` in the program, where `pc_no` is the number of new principal components $k$.

**(2.)** We modify our original MATLAB `new_kmeans.m` function to the function `new_kmeans_clust.m` (see Appendix). This function has added functionality (lines 50-60 of the file) to compute the optimal cost of $k$-means. It does this once the algorithm has converged i.e. the `while` condition is no longer satisfied. To compute the optimal cost, we loop over each data point, then loop over each cluster assignment, calculating `cost(j) = norm(X(i,:) - centroids(j,:))`. Once, we have looped over the three cluster assignments, the 'optimum' cluster and its corresponding cost are stored via `[min_cost, ] = min(cost)`. `min_cost` is added to tot_cost as we iterate over the examples. The function returns `tot_cost` alongside the cell array containing cluster information.

**(3.)** We make use of the previously implemented functions `new_kmeans_clust.m` and `my_PCA.m` in the main code for this problem, `compGI07_A2_3.m` (see Appendix). For this exercise, we run $k$-means over 100 trials for different values of `pca_no` (variable indicating the number of principal components used in the PCA). The reported 3 smallest `occes`, alongside the computed value of their objective, and the means and standard deviations of the `occes` and their objectives are presented in Figures 2.1-2.5 for different cases. Note the first lines of the figures represent the mean and standard deviation of the `occe`. For `pca_no = 1`:

```
>> compGI07_A2_3
Mean after 100 trials: 0.162
Standard deviation after 100 trials: 0.018
Number of principal components: 1
Mean optimum cost is: 62.968
Standard deviation of optimum cost is: 2.883
Lowest three occes are:     0.1600    0.1600    0.1600

Corresponding lowest three costs are:    62.6794   62.6794   62.6794
```

Figure 2.1: Results displayed using one principal component.

For `pca_no = 2`:

```
>> compGI07_A2_3
Mean after 100 trials: 0.189
Standard deviation after 100 trials: 0.145
Number of principal components: 2
Mean optimum cost is: 90.073
Standard deviation of optimum cost is: 11.752
Lowest three occes are:     0.1133    0.1133    0.1133

Corresponding lowest three costs are:    83.8880   83.8880   83.8880
```

Figure 2.2: Results displayed using two principal components.

For `pca_no = 3`:

```
>> compGI07_A2_3
Mean after 100 trials: 0.184
Standard deviation after 100 trials: 0.138
Number of principal components: 3
Mean optimum cost is: 100.455
Standard deviation of optimum cost is: 11.579
Lowest three occes are:     0.1067    0.1067    0.1067

Corresponding lowest three costs are:    94.1705   94.1705   94.1705
```

Figure 2.3: Results displayed using three principal components.

For `pca_no = 4`;

```
>> compGI07_A2_3
Mean after 100 trials: 0.186
Standard deviation after 100 trials: 0.142
Number of principal components: 4
Mean optimum cost is: 103.452
Standard deviation of optimum cost is: 11.285
Lowest three occes are:     0.1067    0.1067    0.1067

Corresponding lowest three costs are:    97.3259   97.3259   97.3259
```

Figure 2.4: Results displayed using four principal components.

For the untransformed data,

```
>> compGI07_A2_3
Mean after 100 trials: 0.186
Standard deviation after 100 trials: 0.142
Mean optimum cost is: 103.452
Standard deviation of optimum cost is: 11.285
Lowest three occes are:     0.1067    0.1067    0.1067

Corresponding lowest three costs are:    97.3259   97.3259   97.3259
```

Figure 2.5: Results displayed using the untransformed data.

Figures 2.6-2.10 present bar charts where the occes are plotted as a function of the rank of the objective function for all five cases.

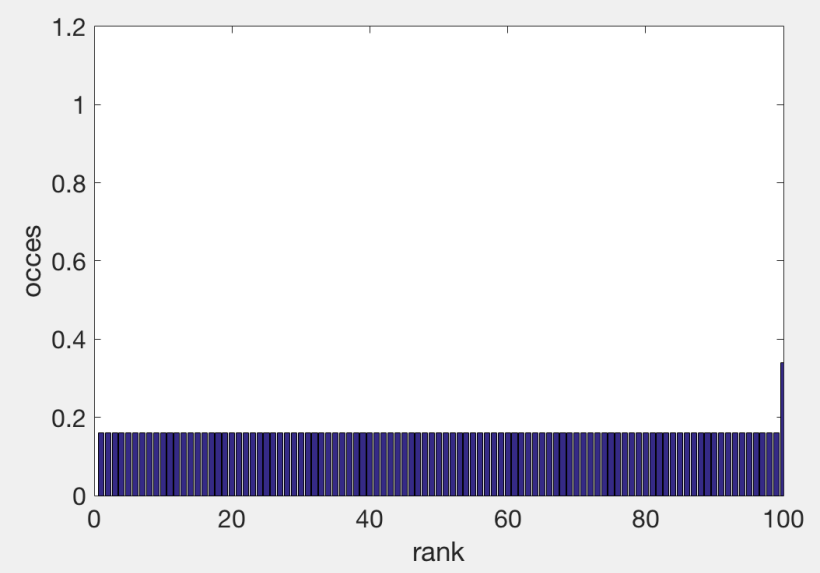


Figure 2.6: Bar chart plotting the occes as a function of rank for `pca_no = 1`

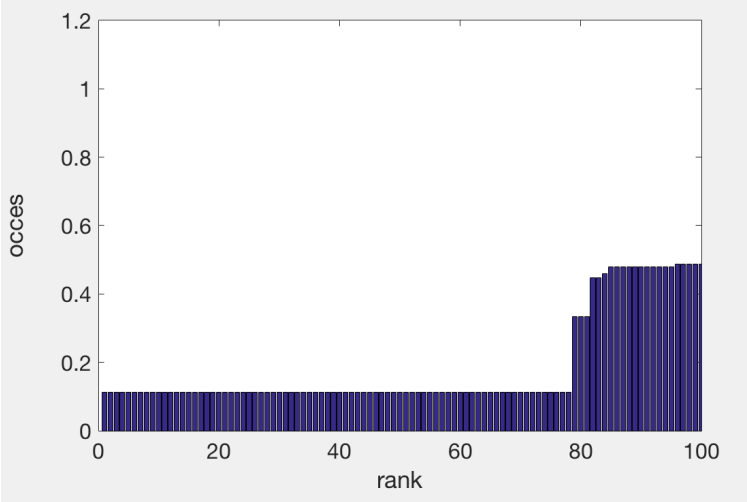


Figure 2.7: Bar chart plotting the occes as a function of rank for `pca_no = 2`

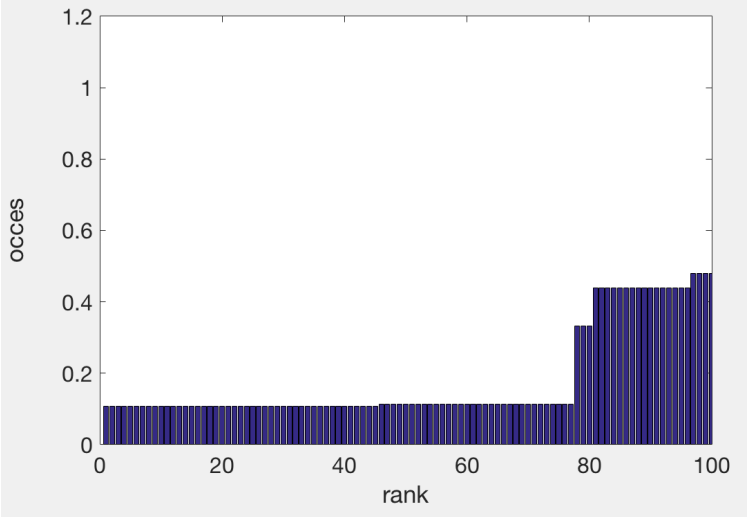


Figure 2.8: Bar chart plotting the occes as a function of rank for `pca_no = 3`

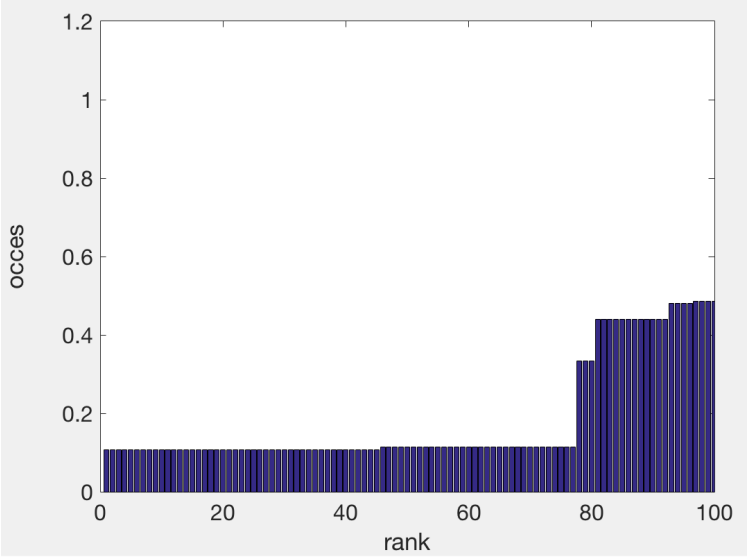


Figure 2.9: Bar chart plotting the occes as a function of rank for `pca_no = 4`

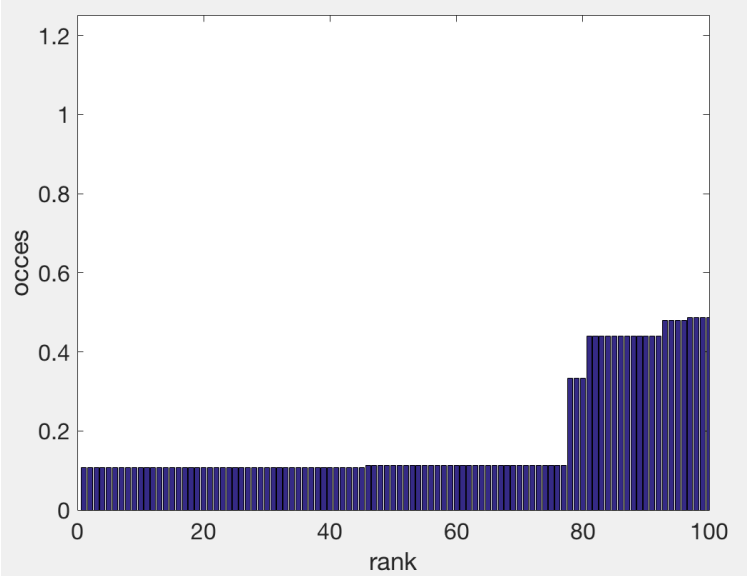


Figure 2.10: Bar chart plotting the occes as a function of rank for the untransformed data

From Figures 2.1-2.10, we can appreciate that as expected, the relationship between the `occe` and the penalty of a cluster assignment is strong. The relationship is harder to appreciate as the number of principal components is increased. We can see that most of the information is captured via two principal components. The 'spike' appears with $k = 2$ and the structure of the plot remains similar for higher numbers of components

**(4.)** A plot is created to visualise the clustered data when the dimensionality is reduced to 2D (`pca_no = 2`) via PCA. This plot is presented in Figure 2.11. The markers for each of the clusters are a different colour. A larger marker is used for the cluster centres. The program used to plot Figure 2.11 is `compGI07_A2_4.m`.

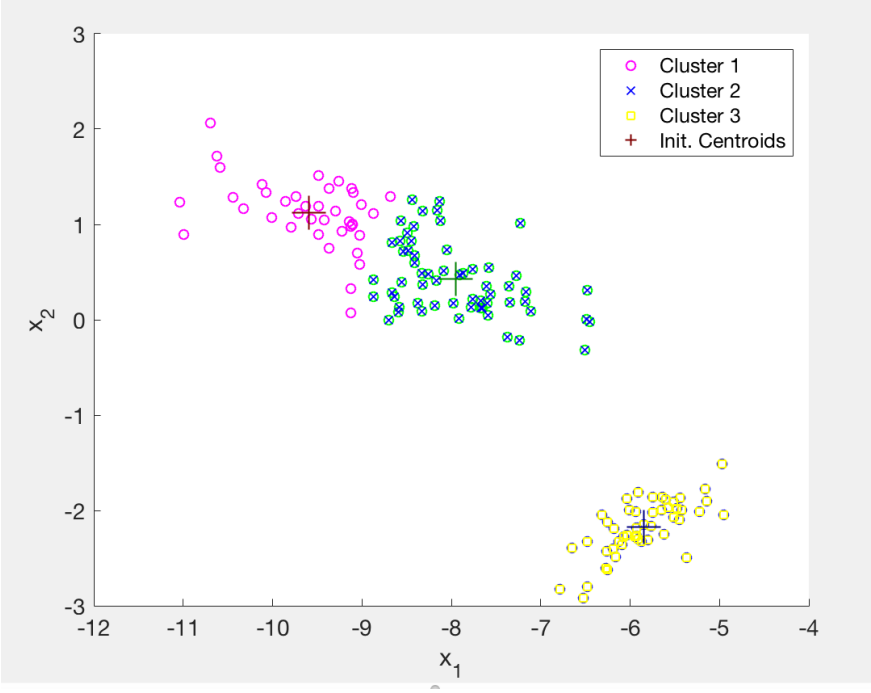


Figure 2.11: Clustered Iris data with dimensionality reduced to 2 via PCA.

Another plot is created to visualise the clustered data when the dimensionality is reduced to 3D (`pca_no = 3`) via PCA. This plot is presented in Figure 2.12. The markers for each of the clusters are a different colour. A larger marker is used for the cluster centres. The program used to plot Figure 2.12 is `compGI07_A2_5.m`.

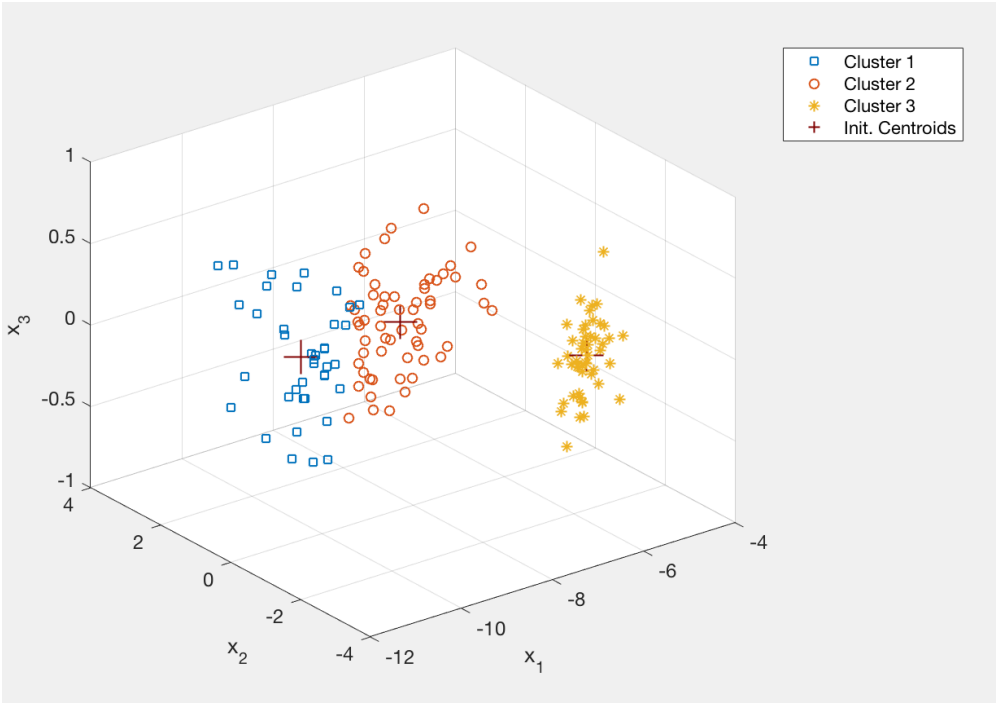


Figure 2.12: Clustered Iris data with dimensionality reduced to 3 via PCA.

## 2.2 Computing the `occe`

Consider a given cluster assignment/partitioning $C_1, ..., C_k$ and a true partitioning $S_1, ...S_k$. The error $e$ is given by,

$$e = \frac{|\{\mathbf{x}|(\mathbf{x} \in C_1 \text{ and } \mathbf{x} \notin S_1) \vee ... \vee (\mathbf{x} \in C_k \text{ and } \mathbf{x} \notin S_k)\}|}{l}. \tag{1}$$

The `occe` is given by,

$$occe = \min_{\mathbf{p} \in \mathbb{P}_k} \frac{|\{\mathbf{x}|(\mathbf{x} \in C_{p_1} \text{ and } \mathbf{x} \notin S_1) \vee ... \vee (\mathbf{x} \in C_{p_k} \text{ and } \mathbf{x} \notin S_k)\}|}{l}. \tag{2}$$

Computing the `occe` for $k$ classes requires naively $k!$ time. This is because there are $k \times k$ possible ways of assigning $k$ 'resources' to $k$ 'tasks'. In this case, 'resources' correspond to clusters $C_1, ...C_k$ (or centroids). These are assigned to partitions $S_1, ..., S_k$, which take on the role of the 'tasks'. In otherwords, naively calculating the `occe` is computationally expensive, since it requires minimising over all permutations of indices $C$. e.g. for $k = 3$, $\mathcal{P}_k = \{(1,2,3),(1,3,2)(2,1,3)(2,3,1),(3,1,2),(3,2,1)\}$. Lets now devise a less computationally expensive algorithm to compute the `occe`.

The `occe` can be redefined as,

$$occe = \sum_{i=1}^{k} \sum_{j=1}^{k} E_{ij} B_{ij},$$

where $E$ is a cost matrix, having as elements the corresponding error/cost for $C_i \to S_j$ and $B$ is a binary assignment matrix, having as elements the minimising indexes of an optimal cluster assignment. In our algorithm, $B$ is initialised as a matrix of zeros. Equation (1) determines the error matrix $E$. Subsequently, the following steps are performed in order to find the optimal arrangement of $C_i \to S_j$ mappings. These steps are characteristic of a combinatorial optimisation problem. As we explain each step of the algorithm, reference will be made to the time complexity of each.

1. **Row subtraction:** `for i=1:k`, search for the lowest error in the $i$-th row of $E$ i.e. search for $e^{(i)}_{min} = \min\{E(i,:)\}$. Consequently, for each row, subtract such minimum error from each element. i.e. `for i=1:k`, `for j=1:k` $e^{(i,j)}_{new} = e^{(i,j)} - e^{i}_{min}$.. We need to iterate over $k$ rows to find the minimum error in each row. Additionally, we need to iterate over $k$ columns to subtract $e^{(i)}_{min}$ from each element in the corresponding row. Hence, this step has time complexity $\mathcal{O}(k^2)$.

2. **Column subtraction:** `for j=1:k`, search for the lowest error in the $j$-th column of $E$ i.e. search for $e_{min}^{(j)} = \min\{E(:,j)\}$. Consequently, for each column, subtract such minimum error from each element. i.e. `for j=1:k`, `for i=1:k` $e_{new}^{(i,j)} = e^{(i,j)} - e_{min}^{j}$. We need to iterate over $k$ columns to find the minimum error in each column. Additionally, we need to iterate over $k$ rows to subtract $e_{min}^{(j)}$ from each element in the corresponding column. Hence, this step has time complexity $\mathcal{O}(k^2)$.

3. **Elimination and checking for optimality:** At this point, each row/column contains at least one column and rows/columns can be eliminated. Consider drawing the minimum number of straight lines on the error matrix to cover all zeros (lines eliminated). If the number of lines eliminated, $l$, is equal or greater to the number of rows or columns, $k$, an optimal assignment can be made and we can skip to Step 5. Alternatively, if $l < k$, the optimality condition is false and no optimal assignment can be made. In this case, we proceed to Step 4.

4. **Zero shifting:** At least one zero needs to be shifted to an uncovered position to increase the minimum number of lines required to cover all the zeros. In order to do this, one must first find the smallest uncovered value. This value must be subtracted from all the uncovered values and added to each value situated at the intersection of two lines. The lines are then removed and we return to Step 3.

   **Time complexity of Steps 3 and 4:** Step 3 covers zeros with $l$ lines. It requires visiting each zero, of which there are at most $k^2$ (the matrix has $k$ rows and $k$ columns, therefore $k^2$ elements), and covering it if not yet covered. Step 4 scans and adjusts at most $\mathcal{O}(k^2)$ values. Steps 3 and 4 also iterate while $l < k$. Each iteration causes the number of lines to increase by at least one. Therefore, for steps 3 and 4, there are at most $\mathcal{O}(k)$ iterations, each of which is $\mathcal{O}(k^2)$. Therefore steps 3 and 4 are $\mathcal{O}(k^3)$.

5. **Making the final assignment:** Choose $k$ zeros, where $k$ is the number of rows and columns of the error matrix, while ensuring that each row/column of the matrix contains only one zero. The chosen zeros represent the final assignment of clusters to partitions. We may find more than one way of choosing the zeros; all choices have the same total cost. Step 5 involves $\mathcal{O}(k)$, since it sums $k$ values of the original matrix. Note that in $B$, the binary assignment matrix, the elements set to ones correspond to the cluster to partition assignments we decide to make. The remaining elements are set to zeros.

Therefore, the overall complexity of the algorithm is $\mathcal{O}(k^3)$ as opposed to the naive $\mathcal{O}(k!)$. Note that this algorithm is a popular one, also known as the Munkres Assignment Algorithm or 'Hungarian Algorithm'. We have proved it runs in polynomial time.

We now provide a toy numerical example to show the correctness of the algorithm. Consider the following arbitrary $k = 3$ error matrix,

$$E = \begin{pmatrix} 30 & 25 & 10 \\ 15 & 10 & 20 \\ 25 & 20 & 15 \end{pmatrix}.$$

**Step 1: Row reduction**. The minimum values for each row are 10,10 and 15 from top to bottom. Subtracting these from their corresponding row elements yields the modified error matrix,

$$\hat{E} = \begin{pmatrix} 20 & 15 & 0 \\ 5 & 0 & 10 \\ 10 & 5 & 0 \end{pmatrix}.$$

**Step 2: Column reduction.** The minimum values for each column are 5,0 and 0 from left to right. Subtracting these from their corresponding column elements yields,

$$\hat{E} = \begin{pmatrix} 15 & 15 & 0 \\ 0 & 0 & 10 \\ 5 & 5 & 0 \end{pmatrix}.$$

**Step 3: Row/column elimination and checking for optimality.** To cover the zeros, we must cover row 2 and column 3 ($l = 2$). We have $k = 3$, hence $l < k$. The optimality condition is false and no optimal assignment can be made. We therefore proceed to Step 4.

**Step 4: Zero shifting.** We find the smallest uncovered value is 5. Subtracting such value from all uncovered entries and adding it to those at the intersection of two lines yields,

$$\hat{E} = \begin{pmatrix} 10 & 10 & 0 \\ 0 & 0 & 15 \\ 0 & 0 & 0 \end{pmatrix}.$$

Return to **Step 3.** We can cover the zeros by covering rows 1-3 or columns 1-3. We have $l = 3$, therefore $l = k$. The optimality condition is true. An optimal assignment can be made and we proceed to Step 5.

**Step 5: Making the final assignment.** We can now choose 3 zeros, provided they are each in a different row/different column. Examples of assignments correspond to the starred zeros below,

$$\hat{E} = \begin{pmatrix} 10 & 10 & 0^* \\ 0^* & 0 & 15 \\ 0 & 0^* & 0 \end{pmatrix} \quad \text{or} \quad \hat{E} = \begin{pmatrix} 10 & 10 & 0^* \\ 0 & 0^* & 15 \\ 0^* & 0 & 0 \end{pmatrix}.$$

Choosing the first (left) assignment, yields the binary assignment matrix,

$$B = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

This results in the following `occe` computation:

$$occe = \sum_{i=1}^{k} \sum_{j=1}^{k} E_{ij} B_{ij} = 15 + 20 + 10 = 45.$$

Similarly, for the second (right) assignment,

$$occe = \sum_{i=1}^{k} \sum_{j=1}^{k} E_{ij} B_{ij} = 25 + 10 + 10 = 45.$$

By inspecting the original error matrix, $E$, we can confirm that our value corresponds to the minimum 'cost' and our algorithm is correct.

# 3 Kernel Perceptron

## 3.1 Exercises

Programming language: MATLAB

### Polynomial Kernel

We must generalise the perceptron to use a polynomial kernel function $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q} + c)^d$ (we set $c = 0$). The function `poly_kernel.m` allows mapping the data to a higher dimensional space and is parametrised by a positive integer $d$, the degree/dimensions of the polynomial. `poly_kernel.m` takes as inputs $d$ and two matrices in the input space. The function returns the polynomial kernel mapping of the inputs.

### Perceptron Training:

.The training algorithm is implemented as the function `trainperceptron.m`. The function operates on one training example of $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$ (`train`) at a time. It intakes the training examples `train`, a polynomial kernel formed by the training inputs `train_kern` , a coefficient matrix labelled `alpha` (is preset to zeros but is updated as we cycle through the loops) and a variable `digitno`. `digitno` corresponds to the number of different hand written digits in the dataset. As highlighted in the assignment, the perceptron must be generalised into a majority network of perceptrons to separate `digitno` classes. In order to do this, `digitno` 2-classifiers must be trained.

**Training each classifier**. For each training example `i`, a single kernel function $K(\mathbf{x}_i, \cdot)$ is computed (`kern_row = train_kern(i,:)`). Each 2-classifier (for each `j`) is denoted as a weight vector $\mathbf{w}^{(j)}$ - `weight(j)` in the code. Each `weight` is set to zero when initialised and trained for a given `kern_row`. This kernel function is scaled by a dual term $\alpha_i$ (initialised to ones) and the product for each example is added to `sum`. i.e. `for k=1:length(alpha), sum=sum+ beta(k)*kern_row(k)`, where `weights(j) = sum` after iterating

(note the length of `alpha` is equivalent to the number of training examples). Also `beta` is a dummy variable with `beta = alpha(j,:)` These lines of code are equivalent to evaluating the sum $\mathbf{w}(\cdot) = \sum_{i=0}^{m} \alpha_i K(\mathbf{x}_i, \cdot)$, with `alpha*kern_row` contributions being added to the sum as we loop over training examples for a given classifier. Once the sum is assigned to a given `weight`, these can be used to classify the training instances. Those with the desired output class (`Y_true == l`), satisfy the condition $\mathbf{w}^T \mathbf{y}_{pred} > 0$ and are labelled as positive examples (`Y = 1`). Otherwise, if the predicted output does not match the target, the condition $\mathbf{w}^T \mathbf{y}_{pred} \leq 0$ is satisfied (`Y*weights(l)<=0`), and the examples are labelled negatively (`Y = -1`). If the prediction is correct, there is no change in the weights. Otherwise, we adjust the `alpha` by adding or subtracting `Y`. In this case, a value of $\texttt{Y} = -1$ corresponds to adding 1 to a particular `alpha`. This procedure can be seen from lines 37 to 43.

A given 2-classifier, in addition to predicting a class, also gives an associated magnitude called a confidence $\kappa$. The confidence $\kappa^{(i)}$ of the classifier $\mathbf{w}^{(i)}$ on a pattern $\mathbf{x}$ is identified as $\mathbf{w}^{(i)}(\mathbf{x})$. Subsequently, our $k$-classifier consists of predicting the classifier with most confidence in its prediction i.e. we predict class $\text{argmax}_{1 \leq i \leq k} \kappa^{(i)}(\mathbf{x})$ for all $k$ 2-classifiers. In the code, the variable `max_confidence` keeps track of this value. It is initially set to a dummy value of $-10^9$ and as we iterate over the classifiers for a training example, we check if `weights(l+1) > max_confidence`. If so, the new `max_confidence` is assigned the value `weights(l)`, and we keep track of the most confident classifier in its prediction $\texttt{max\_w} = \texttt{l}$ in its prediction. Finally, if the final prediction `max_w` does not equal the true value of the target `Y_true`, a mistake is added to the count of `errors` which determine the training error.

## Perceptron Testing

Once the $k$-classifier has been trained on the previous step, the perceptron can be tested. For this purpose, we implement the function `testperceptron.m`. It is very similar to `trainperceptron.m`, with only the prediction step for each example in the test set. The `testperceptron.m` function operates on one test example at a time, taking as inputs the test examples `test`, a polynomial kernel formed between the training inputs and the test inputs `train_test_kern`, a coefficient matrix labelled `alpha`, inherited from the training step, and the variable `digitno`. `digitno` corresponds to the number of different hand written digits in the dataset. As you can see, the function is nearly identical to the previous one, this time without an update step being performed. We loop over single kernel functions for each **test** example to later evaluate the sum $\mathbf{w}(\cdot) = \sum_{i=0}^{m} \alpha_i K(\mathbf{x}_i, \cdot)$ for each **train** example for a given classifier. We keep track of the most confident classifier and the mistake count; this value is returned by the function.

## Mistake Analysis

**Testing Perceptron:** To implement the hold out method/choose a classifier, the following functionality is added to our testing perceptron function. The modified function is called `extendedtestperceptron.m`.

A confusion matrix `error_matrix` keeps track of the number of times a given digit in the validation set has been misclassified for another. Using `ziptrain.m` and `ziptest.m` (digits 0-9), the confusion matrix has 11 rows and 12 columns. The first column indexes the true labels while the first row indexes the digits the true labels have been misclassified for. The values in the middle (rows 2-11, columns 2-11) indicate the number of times the label corresponding to a given row has been misclassified for the column label. The rightmost column (column 12) indicates the total number of times a true label has been misclassified. The mistake counts in `error_matrix` are initialised to zero. As we iterate over the test set examples, both total and specific misclassification counts in `error_matrix` are updated each time there is a mistake (`maxi!=Y_true`).

## Main Code:

The main program is attached in the appendix as `compGI07_A3.m`. It initially loads `ziptrain.dat` and `ziptest.dat`. Note these datasets have ten possible labels, corresponding to digits 0-9. Our code has been written to include the label 0 (`digitno=10`) and requires modification (looping over classes in the perceptron training and test functions) if working with `dtrain123.dat` and `dtest123.dat`. These datasets only include 1-3. The total number of epochs is set to four (`nepochs=4`). This value was worked out heuristically. We notice that at 4 epochs, the training perceptron has not completely converged; its training error (number of mistakes) decreases further with a greater number of epochs. However, this decrease does not necessarily result in a decrease of the validation error. In fact, if we increase the number of passes made over the training data, the algorithm overfits. `nepochs=4` is sufficiently 'good' as a 'sweet spot', with performance degrading

with more iterations due to overfitting. Additionally, a lesser number of total epochs is less computationally expensive.

We iterate over degrees $d = 2, ..., 7$, hence `max_degree=7`. The original training dataset is subdivided into sets `train_set` and `val_set`. `train_set` corresponds to $\frac{2}{3}$ of the original training set and `val_set` is the remaining third. For given parameters $d = 2, ..., 7$ we train on `train_set` and measure our performance on `val_set` as follows. Iterating over each `degree`, `train_kern`, the polynomial kernel between the training examples, and `val_kern`, the polynomial kernel between training and validation examples, are computed. It is worth noting that the kernels are only computed once for each polynomial degree value (outside the epoch iteration). For a given degree selection, the kernels is invariant and pre-computing them vastly diminishes computation time. We then commence iterating over each epoch, updating `alpha` as we call the `trainperceptron.m` function, displaying training and test errors on the screen (calling `extendedtestfunction.m`) and generating a confusion matrix for each perceptron test. Once we have iterated over all epochs for a given value of $d$, the current confusion matrix is displayed and the validation error is stored in the table `holdout_degree`, which keeps track of the validation error for different values of $d$.

## Choosing parameters (model selection):

We choose our optimal parameter $d$ as that corresponding to the classifer which had the best performance (fewest mistakes) on `val_set`. Note that, at this point, when running the aforementioned code, the following output is displayed. Figure 3.1 shows the displayed `error_matrix` corresponding to $d = 3$ (after four epochs). In this particular case, it looks like the most misclassified true label is 4, with the most common mix up being mistaking a four for a two. Ignore the leftmost and rightmost zeros in the top row.

```
0     0     1     2     3     4     5     6     7     8     9     0
0     0     0     3     2     0     0     0     0     2     0     7
1     0     0     0     0     0     0     0     0     1     0     1
2     0     0     0     0     0     0     0     1     1     0     2
3     0     0     0     0     0     3     0     2     2     0     7
4     1     6     7     0     0     0     4     0     2     2    22
5     0     1     3     6     0     0     3     1     2     0    16
6     2     0     1     0     0     0     0     0     0     0     3
7     0     3     1     1     4     0     0     0     0     3    12
8     1     1     3     3     0     2     2     0     0     0    12
9     1     1     1     2     1     0     0     1     1     0     8
```

Figure 3.1: `error_matrix` for $d = 3$, displayed after four epochs.

Figure 3.2 shows the table `holdout_degree`, displayed after iterating over all of $d = 2, ..., 7$. This table presents the validation errors (bottom row) for each degree (top row), with four epochs passed for each degree. Note that the validation error is presented as a percentage corresponding to the percentage of mistakes. In the code, this percentage is computed as `percentage_error = test_errors*100/mval`, where `mval` is the number of examples in the validation set.

```
2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
4.2798    3.7037    2.5103    4.2387    3.9918    4.5679
```

Figure 3.2: Table `holdout_degree` for $d = 2, ..., 7$, displayed after four epochs passed for every degree value.

The table in Figure 3.2 is presented as a graph in Figure 3.3, where the $y$-axis corresponds to the validation error (percentage of mistakes) and the $x$-axis corresponds to the value of the polynomial kernel degree, $d$. In Figure 3.3, We can see that $d = 4$ minimises the validation error ($err_{val} = 2.51\%$). Hence, it is considered the 'optimal' parameter value. We now retrain our classifier on the dataset `train_set` + `val_set` with the chosen parameter $d = 4$. This is known as *hold out* method.

## Final Test

Again, we refer to the main code `compGI07_A3.m`. The original training dataset `ziptrain.m` is given the name `original_train`. The original test set is labelled `test`. The polynomial kernel degree `optimal_degree` is fixed at its optimal value, 4. The classifier is retrained. Firstly, `original_train_kern`, the polynomial kernel between the original training examples and `test_kern`, the polynomial between the training and test examples, are calculated (using the `poly_kernel` function).

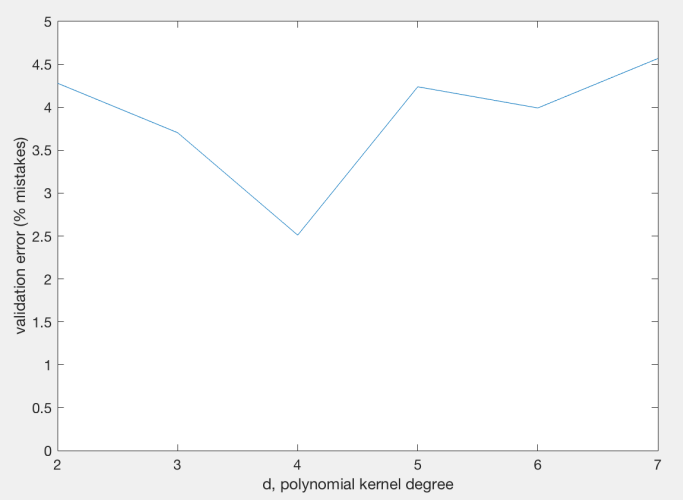


Figure 3.3: Polynomial kernel degree, $d$, plotted against its corresponding validation error.

Subsequently, we again iterate over each epoch, updating `alpha_final` as we call the `trainperceptron.m` function. Again, training and test errors are displayed on the screen. A confusion matrix is generated for each epoch when `extendedtestperceptron.m` is called. Once we have iterated over all epochs for $d = 4$, the final confusion matrix, `final_error_matrix`, is displayed. The final test error is displayed as a percentage and stored as `percentage_error_final`. Additionally, we utilise a variable `min_test_err` to keep track of the lowest test error as we iterate over epochs. We have assigned `nepochs=4`, but it could well be that a lower test error is attained earlier. `opt_epoch` keeps track of such 'optimal' epoch. `min_test_err` corresponds to such lower test error percentage. These values are stored and displayed. For an epoch, if `percentage_error_final < min_test_err`, `min_test_err = percentage_error_final` and `opt_epoch = epoch`. When running the aforementioned code, the following output is displayed (Figure 3.4).

```
The percentage test error after 4 epochs is 6.078724 percent.
A min test error of 5.879422 percent is obtained after 3 test epochs.

Confusion table for test set, d=4, 4 epochs:
     0     0     1     2     3     4     5     6     7     8     9     0
     0     0     1     1     0     1     0     2     1     0     0     6
     1     0     0     0     0     5     0     3     2     0     0    10
     2     4     0     0     3     4     1     0     2     5     0    19
     3     0     0     2     0     1    14     0     2     2     1    22
     4     0     2     5     0     0     1     1     1     0     4    14
     5     3     0     1     4     0     0     0     0     3     1    12
     6     0     1     1     0     3     1     0     0     2     0     8
     7     0     0     2     0     4     0     0     0     1     2     9
     8     3     0     2     2     0     4     0     1     0     2    14
     9     0     0     0     0     4     1     0     1     2     0     8
```

Figure 3.4: `final_error_matrix` for optimal $d = 4$ after four epochs. This follows a print out of the test percentage error after four epochs, and a lesser test error obtained earlier.

Note that the final test error is presented as a percentage corresponding to the percentage of mistakes (mistakes divided by number of test examples multiplied by 100). The test error after four epochs is $err_{test} = 6.08\%$. Note that after three testing epochs, the test error is even lower, $err_{test} = 5.88\%$. These values are significantly lower than the maximum final test error (9%) required by the exercise.

## Extended Analysis

The final confusion matrix can be utilised to answer the following questions for the test set:

- **What is the hardest/easiest digit to recognise?**
- **What are the two digits that are most often confused?**
- **What are the five most difficult to recognise scanned digits relative to our algorithm?**

In order to answer some of these questions, we must know the number of occurrences in the test set corresponding to each class. In the code, the number of occurrences is tracked by `occurrence_count`. We find there are 359 occurrences of 5,6 and 7, 177 occurrences of 0 and 9, 170 occurrences of 1, 3, 4 and 8, and 166 occurrences of 2. Consequently, the table `relative_error` is produced with the percentage error relative to each specific true label. The the output in Figure 3.5 is displayed. The first row of the matrix indexes each digit, the second row lists the number of mistakes for such digit and the third row lists the number of occurrences of such digit on the test set. The final row lists the percentage mistake error on a specific true label.

```
         0    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000    8.0000    9.0000
    6.0000   10.0000   19.0000   22.0000   14.0000   12.0000    8.0000    9.0000   14.0000    8.0000
  177.0000  170.0000  166.0000  170.0000  170.0000  359.0000  359.0000  359.0000  170.0000  177.0000
    3.3898    5.8824   11.4458   12.9412    8.2353    3.3426    2.2284    2.5070    8.2353    4.5198
```

Figure 3.5: `relative_error` table, highlighting test percentage errors for specific digits in the bottom row.

Upon inspection of `relative_error`, one notes that the hardest digit to recognise is three. The highest test percentage error (12.94%) corresponds to three, which also contributes the greatest number of mistakes on the test set (22). On the other hand, the easiest digit to recognise is six. It has the lowest test percentage error (2.23%), contributing only 8 mistakes while being one of the more frequent digits.

Upon inspection of the confusion matrix, the two digits that are most often confused are 3 and 5. In other words, the most frequent wrong classification is mistaking a true label of 3 for a 5. This incorrect classification contributes 14 mistakes.

To find the five most difficult to recognise scanned digits, we proceed as follows. The test set alone using $d = 4$ only can only provide records over 4 iterations (epochs). We decide to use the original training set over all parameters $d = 2, ..., 7$ to train our weights. Subsequently, we test on the full test set and count the number of times each individual scanned digit has been misclassified. Using all values of $d$ again, we now have 24 epochs in total and a more informative record of which the most misclassified digits are. The variable `ESD` is now introduced as an input into the perceptron testing function. It is initialised in the main program as a vector with length equal to the number of test examples. `ESD` has two columns. The first indexes the specific 'record number' of a scanned digit as we move through the dataset. The second column is updated if there is a mistake. i.e. if `maxi!=Y_true`, `ESD(i,2) = ESD(i,2)+1`. Once the function has iterated over all test examples, the matrix `ESD` is returned with the number of mistakes for each record using a given polynomial degree value. In the main code, as we iterate over different epochs and different polynomial degrees, the total values in `ESD` are recursively updated. Once these iterations have been completed, one can sort the entries in `ESD` in descending order using the MATLAB function `sort`. These entries are stored in a final 'table' labelled `hard2rec`. The top 5 hardest-to-recognise entries and their number of mistakes are stored in variable `top5` and displayed. Inspecting the variable `hard2rec`, we can observe that there are 42 entries with the maximum number of mistakes (24 - 4 epochs over 6 degrees). `top5` stores the five hardest-to-recognise entries 'highest up' the test set. These are displayed in Figure 3.6.

```
top5 =

    24    24    24    24    24
    18   123   135   199   234
```

Figure 3.6: `top5` table, recording the hardest-to-recognise entries and the their respective number of mistakes.

We proceed to print off the scanned digits corresponding to the hardest-to-recognise entries in the test set: 18, 123, 135, 199 and 234. When running the code, the true labels of the scanned digits are displayed in the command window. The print-offs are presented in Figure 3.7.

## Summary of Results

For the validation phase and as displayed earlier, a table of test errors for the trained classifiers for $d = 2, ...7$ is presented in Table 1.
Consequently, $d = 4$ is chosen as the optimal parameter. Retraining the classifier on `train_set` + `val_set` and testing on the full `ziptest.m` yields the results in Table 2 over each epoch.
Using $d = 4$,

Figure 3.7: 5 of the hardest-to-recognise digits. From to bottom, left to right, their corresponding true values are 6,3,6,8 and 1. We can observe these labels would be hard to recognise even by a human.

| Degree, $d$ | Validation error (%) |
|---|---|
| 2 | 4.28 |
| 3 | 3.71 |
| 4 | 2.51 |
| 5 | 4.24 |
| 6 | 3.99 |
| 7 | 4.57 |

Table 1: % validation error for polynomial kernel degree, $d = 2, ..., 7$, after four epochs.

| Epochs completed | Test error (%) |
|---|---|
| 1 | 7.03 |
| 2 | 6.23 |
| 3 | 5.88 |
| 4 | 6.08 |

Table 2: % test error for chosen optimal $d = 4$, after the pass of each epoch.

- The hardest digit to recognise (as a class) is three (12.94 % mistake rate).
- The easiest digit to recognise (as a class) is six (2.23 % mistake rate.)
- The most frequent mislabelling occurs misjudging a 3 as a 5 (contributing 14 mistakes).

Iterating over all $d = 2, ..., 7$,

- Among the hardest-to-recognise entries in the full data set are entries 18, 123, 135, 199 and 234, corresponding to true values of 6, 3, 6, 8 and 1, respectively.

## 3.2 Extension

Programming language: MATLAB.

**(1.)**

### Gaussian Kernel

We now generalise the perceptron to use a Gaussian kernel function $K(\mathbf{p}, \mathbf{q}) = e^{-c||p-q||^2}$. The function `gaussian_kernel.m` allows mapping the data to a higher dimensional space and is parametrised by a positive integer $c$. `gaussian_kernel.m` takes as inputs $c$ and two matrices in the input space. The function returns the Gaussian kernel mapping of the inputs.

The methods described in **3.1** are repeated. Bare in mind the `gaussian_kernel.m` function does not take as input the polynomial degree. It intakes parameter $c$ over the same range $c = 2, ..., 7$. To run the code using the Gaussian kernel simply replace the functions `poly_kernel` with `gaussian_kernel` in the main code. Note that the computation using the Gaussian kernel is slower. Additionally, note that the perceptron appears to over-fit much faster using a Gaussian kernel, with two epochs appearing to be enough to reach a 'sweet spot'. For both reasons, we set `nepochs=2` when using the Gaussian kernel. The file `compGI07_A3_2.m` runs these changes automatically.

## Summary of results

For the validation phase, a table of test errors for the trained classifiers for $c = 2, ...7$ is presented in Table 3.

| Degree, $d$ | Validation error (%) |
|---|---|
| 2 | 7.20 |
| 3 | 7.41 |
| 4 | 7.20 |
| 5 | 7.00 |
| 6 | 6.21 |
| 7 | 8.11 |

Table 3: % validation error for Gaussian kernel $c = 2, ..., 7$, after two epochs.

Consequently, $d = 6$ is chosen as the optimal parameter. Retraining the classifier on `train_set` + `val_set` and testing on the full `ziptest.m` yields the results in Table 4 over each epoch.

| Epochs completed | Test error (%) |
|---|---|
| 1 | 12.20 |
| 2 | 6.21 |

Table 4: % test error for chosen optimal $d = 6$, after the pass of each epoch.

Using $d = 6$, we obtain the following mistake rates for each class (Table 5):

| Class | Test error (%) |
|---|---|
| 0 | 3.39 |
| 1 | 5.88 |
| 2 | 16.87 |
| 3 | 11.76 |
| 4 | 12.35 |
| 5 | 6.13 |
| 6 | 3.62 |
| 7 | 4.18 |
| 8 | 15.29 |
| 9 | 8.47 |

Table 5: % Mistake rates for each class using a Gaussian kernel.

We record that,

- The hardest digit to recognise (as a class) is two (16.87 % mistake rate).
- The easiest digit to recognise (as a class) is zero (3.39 % mistake rate.)
- The most frequent mislabelling occurs misjudging an 8 as a zero (contributing 16 mistakes).

Iterating over all $d = 2, ..., 7$,

- Among the hardest-to-recognise entries in the full data set are entries 18, 28, 123, 146 and 165, corresponding to true values of 6, 3, 3, 2 and 0, respectively. The print-offs for these entries are presented in Figure 3.8.

Figure 3.8: 5 of the hardest-to-recognise digits for the Gaussian kernel. From to bottom, left to right, their corresponding true values are 6,3,3,2 and 0. We can observe that these classes would be hard to recognise even by a human.

**(2.)** We attempted to compare the Kernel Perceptron algorithm with another two algorithms. These were **(i)** A multi-class support vector machine **(ii)** the $k$-NN algorithm. **Support vector machine:** We started coding up a function but unfortunately, were unable to see it in action due to time constraints. **k-NN**. For $k$-NN, the built-in MATLAB function `fitcknn` was utilised. We iterated over different values of $k = 2, ..., 7$ (nearest neighbours) to find the optimal one in a similar fashion to the previous parts of this exercise i.e we find an optimal $k$ over the validation, then use the full test set. $k$-NN seemed to give better results for lower values of $k$, with test errors around 3%. We suggest two reasons for the kernel perceptron's worse performance. Firstly, the data may not be as linearly separable as we believe. Secondly, the nature of $k$-NN may capture better the irregular dynamic of written digits. The frequency of certain digits fluctuates greatly through time as some digits tend to be closer/further away from each other. This introduces noise into the system.

# Appendix (Code)

## Exercise 1

### 1.1.1

#### `new_kmeans.r` (main $k$-means implementation)

```
source("./squared_distance.R")
source("./choose_centroids.R")

NewKmeans <- function(X, n_clusters){
  # X is a matrix (represented by a Data Frame) R^n where each row represents
  # a data point
  # n_clusters is the number of clusters we want to divide the data in.
  m <- nrow(X)
  n <- ncol(X)
  # create matrix to keep track of which data point belongs to which cluster
  clustA <- matrix(0L, nrow = m, ncol = 2)
  # choose centers randomly
  centroids <- randCentroids(X, n_clusters)
  # keep the original for plotting in order to test
  original <- centroids

```

```
17    # has_changed is the boolean variable that keeps track of the changes in
          any of the points
18    has_changed <- TRUE
19    # update the clusters that a data point belongs to
20    # the algorithm ends where no more points are updated, has_changed = FALSE
21    while(has_changed){
22      has_changed <- FALSE
23      for(i in 1:m){
24        # my_distance, set the distance to inf, the first time it will be less
              than any distance calculated
25        # keeps the distance of the Ith point to the Jth centroid
26        my_distance <- 50000
27        my_new_index <- -1
28        # my_new_index is the variable that updates the index of the cluster
              that the
29        # ith observation belongs to
30        for(j in 1:n_clusters){
31          # print(X[i,])
32          # print(centroids[j,])
33          cal_dist <- sqrt(sum((X[i,] - centroids[j,])^2))
34
35          if(cal_dist < my_distance){
36            my_distance <- cal_dist
37            my_new_index <- j
38          }
39        }
40
41        if(clustA[i,1] != my_new_index){
42          has_changed <- TRUE
43        }
44
45        clustA[i,1] <- my_new_index
46        clustA[i,2] <- cal_dist^2
47      }
48      # calculate the new centroids of the clusters
49      for(cent in 1:n_clusters){
50        if(length(which(clustA[,1] == cent)) == 0){
51          for(colm in 1:n){
52            minJ <- min(X[,colm])
53            maxJ <- max(X[,colm])
54            rangeJ <- maxJ - minJ
55            centroids[cent,colm] <- minJ + rangeJ * runif(1)
56          }
57        }else{
58          clust_points <- X[which(clustA[,1] == cent),]
59          for(colm2 in 1:ncol(X)){
60            centroids[cent,colm2] = mean(clust_points[,colm2])
61          }
62        }
63      }
64    }
65
66    # create a data.frame with the values to return
67    df <- data.frame(Original = original, Centroids = centroids)
68    # return the data.frame with the Original centroids and the Centroids that
          resulted from the algorithm
69    return(clustA)
70  }
```

choose_centroids.R (chooses random centroids)

```
randCentroids <- function(X, n_centroids){
  # Return n_centroids amount of centroids from the Data Frame X

  # initialize an empty matrix
  centroids <- matrix(0, n_centroids, ncol(X))

  # create as many centroids as clusters we want to have in our algorithm
  # for(i in 1:n_centroids){
  #   # initialize the random points at each iteration
  #   cent <- NULL
  #   # select k random points
  #   cent <- (X[sample(nrow(X), n_centroids),])
  #   # store the x and y values of each centroid in the array
  #   for(j in 1:ncol(X)){
  #     centroids[i,j] <- sum(cent[,j])/n_centroids
  #   }
  # }

  n <- ncol(X)
  for(j in 1:n){
    minJ <- min(X[,j])
    maxJ <- max(X[,j])
    rangeJ <- maxJ - minJ
    centroids[,j] <- minJ + rangeJ * runif(n_centroids)
    # print(centroids[,j])
    # print(centroids)
  }
  # return the array with all the x and y coordinates of the initial
     centroids
  return(centroids)
}
```

squared_distance.r (calculates 2-norm distance for $k$-means)

```
squaredDistance <- function(vec1, vec2){
  # squaredDistance calculates the norm2 distance

  # calculate the squared distance between two vectors
  return(sqrt(sum(vec1-vec2)^2))
}
```

### 1.1.2

new_kmeans2.r (Similar to new_kmeans.r but returns centroid position)

```
source("./squared_distance.R")
source("./choose_centroids.R")

NewKmeans2 <- function(X, n_clusters){
  # X is a matrix (represented by a Data Frame) R^n where each row represents
  # a data point
  # n_clusters is the number of clusters we want to divide the data in.
  m <- nrow(X)
  n <- ncol(X)
  # create matrix to keep track of which data point belongs to which cluster
  clustA <- matrix(0L, nrow = m, ncol = 2)
  # choose centers randomly
  centroids <- randCentroids(X, n_clusters)
```

```
14  # keep the original for plotting in order to test
15  original <- centroids
16
17  # has_changed is the boolean variable that keeps track of the changes in
        any of the points
18  has_changed <- TRUE
19  # update the clusters that a data point belongs to
20  # the algorithm ends where no more points are updated, has_changed = FALSE
21  while(has_changed){
22    has_changed <- FALSE
23    for(i in 1:m){
24      # my_distance, set the distance to inf, the first time it will be less
            than any distance calculated
25      # keeps the distance of the Ith point to the Jth centroid
26      my_distance <- 50000
27      my_new_index <- -1
28      # my_new_index is the variable that updates the index of the cluster
            that the
29      # ith observation belongs to
30      for(j in 1:n_clusters){
31        # print(X[i,])
32        # print(centroids[j,])
33        cal_dist <- sqrt(sum((X[i,] - centroids[j,])^2))
34
35        if(cal_dist < my_distance){
36          my_distance <- cal_dist
37          my_new_index <- j
38        }
39      }
40
41      if(clustA[i,1] != my_new_index){
42        has_changed <- TRUE
43      }
44
45      clustA[i,1] <- my_new_index
46      clustA[i,2] <- cal_dist^2
47    }
48    # calculate the new centroids of the clusters
49    for(cent in 1:n_clusters){
50      if(length(which(clustA[,1] == cent)) == 0){
51        for(colm in 1:n){
52          minJ <- min(X[,colm])
53          maxJ <- max(X[,colm])
54          rangeJ <- maxJ - minJ
55          centroids[cent,colm] <- minJ + rangeJ * runif(1)
56        }
57      }else{
58        clust_points <- X[which(clustA[,1] == cent),]
59        for(colm2 in 1:ncol(X)){
60          centroids[cent,colm2] = mean(clust_points[,colm2])
61        }
62      }
63    }
64  }
65
66  # create a data.frame with the values to return
67  df <- data.frame(Original = original, Centroids = centroids)
68  # return the data.frame with the Original centroids and the Centroids that
        resulted from the algorithm
```

```
  return(df)
}
```

### generate_data.r (Code to generate Figure 1 (Exercise 1.2.2))

```
rm(list=ls(all=TRUE))

if(!require("combinat")) {
  install.packages("combinat")
}
library(combinat)

# set fixed random seed in order to replicate results
set.seed(8)

A1 <- t(matrix(c(0.5, 0.2, 0, 2), ncol = 2));
u1 <- c(4, 0);

A2 <- t(matrix(c(0.5, 0.2, 0, 0.3), ncol = 2));
u2 <- c(5, 7);

A3 <- t(matrix(c(0.8, 0, 0, 0.8), ncol = 2));
u3 <- c(7, 4);

data <- matrix(rnorm(150*2,mean=0,sd=1), 150, 2)

# test algorithm in 2D dataset where each dataset has a different
# mean and variance

# Group 1 (S1)
for(f in 1:50){
  data[f,] <- u1 + A1 %*% data[f,]
}

# Group 2 (S2)
for(s in 51:100){
  data[s,] <- u2 + A2 %*% data[s,]
}

# Group 3 (S3)
for(t in 101:150){
  data[t,] <- u3 + A3 %*% data[t,]
}

# data <- readMat('data.mat')
# my_data <- data[[1]]
#

# K-means algorithm testing
source("./new_kmeans.R")
source("./new_kmeans2.R")
my_data <- data
kmm <- NewKmeans(my_data, 3)
df <- NewKmeans2(my_data,3)


plot(my_data, xlab="x", ylab="y")
points(my_data[which(kmm[,1] == 1),], col = 'darkgreen', pch = 15)
points(my_data[which(kmm[,1] == 2),], col = 'darkorchid', pch = 16)
points(my_data[which(kmm[,1] == 3),], col = 'firebrick', pch = 17)
```

```
### Plot the original random centroids
points(df[c('Original.1', 'Original.2')], col = 'blue', pch=4, cex=5)
### Plot the re-calculated centroids
points(df[c('Centroids.1', 'Centroids.2')], col = 'black', pch = 10,cex=5, bg
    ='red')
legend ("bottomright", bty = "n", inset = c(0.1,0),legend = c('C1','C2', 'C3'
    , 'Orig. centroids',
        'New centroids'),cex=1.2,col = c("darkgreen", "firebrick", "
           darkorchid", "blue","black"),
                                                    pch = c(15, 17, 16,4,10))
```

`compute_occe.m` **(function to compute occe)**

```
% function compute_occe.m -> computes optimistic clustering
% classification error.
%
% inputs -> batches: these are the cluster groups output by the
% k-means algorithm i.e. partitionings C_1, C_2, C_3.
% true_splits contain the true partitionings S_1, S_2, S_3.
%
% outputs -> occe: the optimistic clustering classification error
% for C_i, S_i.
%
function occe = compute_occe(batches, true_splits)
    % this part computes permutations of cluster groups
    n = length(batches);
    permutes = 1;
    for nn=2:n
        % helper for recursion
        Psmall = permutes;
        m = size(Psmall,1);
        permutes = zeros(nn*m,nn);
        permutes(1:m, 1) = nn;
        permutes(1:m, 2:end) = Psmall;
        for i = nn-1:-1:1
            reorder = [1:i-1, i+1:nn];
            % assign the next m rows in permutes.
            permutes((nn-i)*m+1:(nn-i+1)*m,1) = i;
            permutes((nn-i)*m+1:(nn-i+1)*m,2:end) = reorder(Psmall);
        end
    end
    % inspiration from MATLAB perms function
    if isequal(batches, 1:n)
        permutes = cast(permutes, 'like', batches);
    else
        permutes = batches(permutes);
    end
    % permutes -> batches (vectors of n rows) have created a matrix
    % of n! rows and n columns with all possible permutations of n
    % elements
    tot_runs = size(permutes,1); % total number of iterations
    simple_error = zeros(tot_runs,1); % vector keeping track of simple error
    % initialise cell array keeping track of cluster permutations
    CLUSTER = cell(3,1);
    % initialise cell array keeping track of errors i.e. this
    % variable keeps track of the number of times x is an element
    % of C but not of S.
    ERROR = cell(3,1);
    min_simple_error = 100000000;
    for run = 1:tot_runs   % loop over each iterations
```

```
48          for k = 1:3   % for each k = 1,..3
49              CLUSTER{k} = permutes{run,k}; % compute cluster assignments
50              % compute number of times x is in C and not in S
51              % row/data point mismatches
52              ERROR{k} = ~ismember(CLUSTER{k},true_splits{k},'rows');
53          end
54          % call function for simple error computation
55          simple_error(run) = compute_simple_error(ERROR);
56          % keep track of minimum simple error
57          if simple_error(run) < min_simple_error
58                  min_simple_error = simple_error(run);
59          end
60      end
61      % occe is the optimum minimum error
62      occe = min_simple_error;
63  end
```

### `compute_simple_error.m` (helper function to compute simple error)

```
1  % function compute_simple_error.m -> helper function
2  % to compute simple error. works
3  %
4  % inputs -> ERROR. cell array with number of times a data point is
5  % in C but not in S.
6  %
7  % outputs -> simple_error for a given run
8  %
9  function simple_error = compute_simple_error(ERROR)
10     tot_error = 0; % initialise total error
11     tot_points = 150; % total number of points in data set
12     for k=1:3
13         error_component = sum(ERROR{k}); % error for a given i=1,...k
14         tot_error = tot_error + error_component;
15     end
16     simple_error = tot_error/tot_points; % returns simple error
17 end
```

### `gen2data.m` (data generation for 1.1.2)

```
1  %
2  % Data provided by Mark Herbster for Assignment 2
3  %
4  function [ data ] = genData2
5      % generate data
6      A1=[0.5  0.2; 0  2];
7      u1=[4  0];
8      A2=[0.5  0.2; 0  0.3];
9      u2=[5  7];
10     A3=[0.8  0; 0  0.8];
11     u3=[7  4];
12     data = randn(150,2) ;
13     for i=1:50
14         data(i,:)     = u1' + A1 * data(i,:)' ;
15     end
16     for i=51:100
17         data(i,:)  = u2' + A2 * data(i,:)' ;
18     end
19     for i=101:150
20         data(i,:) = u3' + A3 * data(i,:)' ;
21     end
```

```
end
```

### new_kmeans.m (MATLAB $k$-means implementation)

```
% The function new_kmeans.m models the k-means clustering algorithm.
%
% It takes as inputs a set of datapoints X and the specified number of
% clusters one wishes to assign to the data.
%
% It ouputs a cell array cluster_info which contains the centroids of
% each cluster and the data point coordinates corresponding to each cluster.
%
% cluster_info.centroids - centroid information
% cluster_info.clusters - data points assigned to each cluster
%
function [cluster_info] = new_kmeans(X, n_clusters)
    [m,~]=size(X); % m is the number of data points
    % randperm to randomly choose centroids instead of choose_centroids.r
    centroids = X(randperm(m,n_clusters),:);
    % initialise distance metrics
    my_distance = 50000; % dist_provisional - dist
    dist = zeros(m,n_clusters);
    % update clusters that a data point belongs to. the algorithm
    % ends where no more points are updated, dist is not lesser than
    % dist_provisional
    while norm(my_distance) > 0
        dist_provisional = dist; % distance metric for t-1
        dist = zeros(m, n_clusters);  % restart distance metric
        % track closest centroids
        for i = 1:m  % loop over each data point
            % recall centroids is coord vector of cluster centers
            [m_cent,~] = size(centroids);
            metric = ones(1, m_cent); % initialisation
            for j = 1:m_cent % loop over each centroid
                % euclidean distance between point and centroid
                metric(j) = norm(X(i,:) - centroids(j,:));
            end
            % assigned centroid to data point minimises distance
            [~, closest] = min(metric);
            dist(i, closest) = 1;
        end
        % calculate new cluster centroids
        for j = 1:n_clusters % for each cluster
            % create a matrix to keep track of which data point belongs to
            % each cluster
            matrix = X(dist(:,j) == 1,:);
            [mmat, ~] = size(matrix);
            centroids(j,:) = sum(matrix)/mmat;
        end
        my_distance = dist_provisional - dist;
    end
    % a cell array cluster_info is returned with coordinate information of
    % the centroids for each cluster and the data point coordinates
    % corresponding to each cluster.
    for j = 1:n_clusters
        cluster_info.centroids{j} = centroids(j,:);
        cluster_info.clusters{j} = X(dist(:,j) == 1,:);
    end
end
```

`compGI07_A1_2.m` **(main code for 1.1.2)**

```
% The function new_kmeans.m models the k-means clustering algorithm.
%
% It takes as inputs a set of datapoints X and the specified number of
% clusters one wishes to assign to the data.
%
% It ouputs a cell array cluster_info which contains the centroids of
% each cluster and the data point coordinates corresponding to each cluster.
%
% cluster_info.centroids - centroid information
% cluster_info.clusters - data points assigned to each cluster
%
function [cluster_info] = new_kmeans(X, n_clusters)
    [m,~]=size(X); % m is the number of data points
    % randperm to randomly choose centroids instead of choose_centroids.r
    centroids = X(randperm(m, n_clusters),:);
    % initialise distance metrics
    my_distance = 50000; % dist_provisional - dist
    dist = zeros(m, n_clusters);
    % update clusters that a data point belongs to. the algorithm
    % ends where no more points are updated, dist is not lesser than
    % dist_provisional
    while norm(my_distance) > 0
        dist_provisional = dist; % distance metric for t-1
        dist = zeros(m, n_clusters);  % restart distance metric
        % track closest centroids
        for i = 1:m  % loop over each data point
            % recall centroids is coord vector of cluster centers
            [m_cent,~] = size(centroids);
            metric = ones(1, m_cent); % initialisation
            for j = 1:m_cent % loop over each centroid
                % euclidean distance between point and centroid
                metric(j) = norm(X(i,:) - centroids(j,:));
            end
            % assigned centroid to data point minimises distance
            [~, closest] = min(metric);
            dist(i, closest) = 1;
        end
        % calculate new cluster centroids
        for j = 1:n_clusters % for each cluster
            % create a matrix to keep track of which data point belongs to
            % each cluster
            matrix = X(dist(:,j) == 1,:);
            [mmat, ~] = size(matrix);
            centroids(j,:) = sum(matrix)/mmat;
        end
        my_distance = dist_provisional - dist;
    end
    % a cell array cluster_info is returned with coordinate information of
    % the centroids for each cluster and the data point coordinates
    % corresponding to each cluster.
    for j = 1:n_clusters
        cluster_info.centroids{j} = centroids(j,:);
        cluster_info.clusters{j} = X(dist(:,j) == 1,:);
    end
end
```

### 1.1.3

`compGI07_A1_3.m` **(main code for 1.1.3)**

```
clear all;
% load iris dataset
iris_file = load('iris.csv');
% set random seed
rng(332);
k=3;  % cluster number
tot_iters = 100; % total iterations.
% initialise occe place-keeper
occes= zeros(1,tot_iters); % occe placekeeper
% partition limits
low_lim = 50;
mid_lim = 100;
high_lim =150;
% call Kmeans
for i = 1:tot_iters;
    part3data = iris_file; % data generation provided in question
    % returns information on cluster assignments as cell array
    % clusters.clusters composed data points assigned to each cluster
    clusters = new_kmeans(part3data, k);
    % clusters.clusters composed of data points assigned to each cluster
    % clusters.centroids composed of centroid information.
    true_splits{1} = part3data(1:low_lim,:);
    true_splits{2} = part3data(low_lim+1:mid_lim,:);
    true_splits{3} = part3data(mid_lim+1:high_lim,:);
    % occes computation -> assigned clusters vs true splits
    occes(i) = compute_occe(clusters.clusters,true_splits);
end
occe_mean = sum(occes)/length(occes);
occe_stdev = std(occes); % standard deviation of occe

fprintf('Mean after 100 trials: %.3f \n', occe_mean)
fprintf('Standard deviation after 100 trials: %.3f \n', occe_stdev)
```

## Exercise 2

### 2.1.1

`my_pca.m (PCA function)`

```
% function my_pca.m performs PCA on a dataset.
%
% Inputs: dataset X (mxn matrix), pc_no -> number of new principal
% components
%
% Output: Transformed coordinates using feature map
%
function transformed_data = my_pca(X, pc_no)
    % covariance matrix computations
    big_covar = X'*X;
    covar = big_covar/length(X);
    % apply singular value decomposition
    [u,~,~]=svd(covar);
    % data transformation
    transformed_data = X*u(:,1:pc_no);
end
```

### 2.1.2

#### new_kmeans_clust.m ($k$-means algorithm returning cost)

```
% The function new_kmeans_clust.m models the k-means clustering algorithm.
% This modification also returns a cost function alongside the cluster
% info.
%
% It takes as inputs a set of datapoints X and the specified number of
% clusters one wishes to assign to the data.
%
% It ouputs a cell array cluster_info which contains the centroids of
% each cluster and the data point coordinates corresponding to each cluster.
% It also outputs the optimum cost function for the algorithm
%
% cluster_info.centroids - centroid information
% cluster_info.clusters - data points assigned to each cluster
%
function [cluster_info, tot_cost] = new_kmeans_clust(X, n_clusters)
    [m,~]=size(X); % m is the number of data points
    % randperm to randomly choose centroids instead of choose_centroids.r
    centroids = X(randperm(m,n_clusters),:);
    % initialise distance metrics
    my_distance = 50000; % dist_provisional - dist
    dist = zeros(m,n_clusters);
    % update clusters that a data point belongs to. the algorithm
    % ends where no more points are updated, dist is not lesser than
    % dist_provisional
    while norm(my_distance) > 0
        dist_provisional = dist; % distance metric for t-1
        dist = zeros(m, n_clusters); % restart distance metric
        % track closest centroids
        for i = 1:m % loop over each data point
            % recall centroids is coord vector of cluster centers
            [m_cent,~] = size(centroids);
            metric = ones(1, m_cent); % initialisation
            for j = 1:m_cent % loop over each centroid
                % euclidean distance between point and centroid
                metric(j) = norm(X(i,:) - centroids(j,:));
            end
            % assigned centroid to data point that minimises distance
            [~, closest] = min(metric);
            dist(i, closest) = 1;
        end
        % calculate new cluster centroids
        for j = 1:n_clusters % for each cluster
            % create a matrix to keep track of which data point belongs to
            % each cluster
            matrix = X(dist(:,j) == 1,:);
            [mmat, ~] = size(matrix);
            centroids(j,:) = sum(matrix)/mmat;
        end
        my_distance = dist_provisional - dist;
    end
    cost = zeros(1,n_clusters);
    tot_cost = 0;
    % COST FUNCTION COMPUTATION
    for i=1:m
        for j = 1:m_cent % loop over each centroid
            % euclidean distance between point and centroid
```

```
57              cost(j) = norm(X(i,:) - centroids(j,:));
58          end
59          [min_cost, ~] = min(cost); % minimum cost for given example
60          tot_cost = tot_cost + min_cost; % keeps track of total cost.
61      end
62      % a cell array cluster_info is returned with coordinate information of
63      % the centroids for each cluster and the data point coordinates
64      % corresponding to each cluster.
65      for j = 1:n_clusters
66          cluster_info.centroids{j} = centroids(j,:);
67          cluster_info.clusters{j} = X(dist(:,j) == 1,:);
68      end
69  end
```

### 2.1.3

### compGI07_A2_3.m (bar charts)

```
1  clear all;
2  % set random seed
3  rng(332)
4  k=3;  % cluster number
5  pca_no=1; % number of components for PCA function.
6  tot_iters = 100; % total iterations.
7  % initialise occe place-keeper
8  occes= zeros(1,tot_iters); % occe placekeeper
9  % partition limits
10 low_lim = 50;
11 mid_lim = 100;
12 high_lim =150;
13 tot_tot_cost = zeros(1,tot_iters); % keeps track of cost during all
       iterations
14 % call Kmeans
15 for i = 1:tot_iters;
16     part2data = load('iris.csv'); % data generation provided in question
17     % returns information on cluster assignments as cell array
18     % clusters.clusters composed data points assigned to each cluster
19     part2data = my_pca(part2data,pca_no);
20     [clusters, tot_cost] = new_kmeans_clust(part2data, k);
21     % clusters.clusters composed of data points assigned to each cluster
22     % clusters.centroids composed of centroid information.
23     tot_tot_cost(i) = tot_cost;
24     true_splits{1} = part2data(1:low_lim,:);
25     true_splits{2} = part2data(low_lim+1:mid_lim,:);
26     true_splits{3} = part2data(mid_lim+1:high_lim,:);
27     % occes computation -> assigned clusters vs true splits
28     occes(i) = compute_occe(clusters.clusters,true_splits);
29 end
30 occe_mean = sum(occes)/length(occes);
31 occe_stdev = std(occes); % standard deviation of occe
32 mean_cost = mean(tot_tot_cost);
33 std_cost = std(tot_tot_cost);
34 occes_temp =  sort(occes,'ascend');
35 occes_low3 = occes_temp(1:3);
36 cost_temp = sort(tot_tot_cost, 'ascend');
37 cost_low3 = cost_temp(1:3);
38
39
40 fprintf('Mean after 100 trials: %.3f \n', occe_mean)
41 fprintf('Standard deviation after 100 trials: %.3f \n', occe_stdev)
```

```
fprintf('Number of principal components: %d \n', pca_no)
fprintf('Mean optimum cost is: %.3f \n', mean_cost)
fprintf('Standard deviation of optimum cost is: %.3f \n', std_cost)
fprintf('Lowest three occes are: ')
disp(occes_low3)
fprintf('Corresponding lowest three costs are: ')
disp(cost_low3)
% bar chart of occes to rank
b = bar(occes_temp);
```

#### 2.1.4

#### compGI07_A2_5.m (3D visualisation)

```
clear all;
% set random seed
rng(332)
k=3;  % cluster number
% partition limits
part2data_original = load('iris.csv'); % data generation provided in question
% returns information on cluster assignments as cell array
% clusters.clusters composed data points assigned to each cluster

% FOR 3 PRINCIPAL COMPONENTS
pca_no=2; % number of components for PCA function.
part2data = my_pca(part2data_original,pca_no);
[clusters, tot_cost] = new_kmeans_clust(part2data, k);
% retrieve individual cluster assignments
centroid1 = clusters.centroids{1};
centroid2 = clusters.centroids{2};
centroid3 = clusters.centroids{3};
cluster1 = clusters.clusters{1};
cluster2 = clusters.clusters{2};
cluster3 = clusters.clusters{3};
figure
x1=scatter(cluster1(:,1),cluster1(:,2),'m','o');
hold on
x2=scatter(cluster2(:,1),cluster2(:,2),'b','x');
x3=scatter(cluster3(:,1),cluster3(:,2),'y','s');
x4=scatter(centroid1(:,1),centroid1(:,2),375,[0.5,0,0],'+');
scatter(centroid2(:,1),centroid2(:,2),375,[0,0.5,0],'+');
scatter(centroid3(:,1),centroid3(:,2),375,[0,0,0.5],'+');
hold off

legend([x1 x2 x3 x4],'Cluster 1', 'Cluster 2', 'Cluster 3', 'Init. Centroids'
    )
```

#### compGI07_A2_4.m (2D visualisation)

```
clear all;
% set random seed
rng(332)
k=3;  % cluster number
% partition limits
part2data_original = load('iris.csv'); % data generation provided in question
% returns information on cluster assignments as cell array
% clusters.clusters composed data points assigned to each cluster

% FOR 3 PRINCIPAL COMPONENTS
pca_no=3; % number of components for PCA function.
part2data = my_pca(part2data_original,pca_no);
```

```
[clusters, tot_cost] = new_kmeans_clust(part2data, k);
% retrieve individual cluster assignments
centroid1 = clusters.centroids{1};
centroid2 = clusters.centroids{2};
centroid3 = clusters.centroids{3};
cluster1 = clusters.clusters{1};
cluster2 = clusters.clusters{2};
cluster3 = clusters.clusters{3};
% PLOTS
figure
x1 =scatter3(cluster1(:,1),cluster1(:,2),cluster1(:,3),'s');
hold on
x2 =scatter3(cluster2(:,1),cluster2(:,2),cluster2(:,3),'o');
x3 =scatter3(cluster3(:,1),cluster3(:,2),cluster3(:,3),'*');

x4 = scatter3(centroid1(:,1),centroid1(:,2),centroid1(:,3),375,[0.5,0,0],'+')
    ;
scatter3(centroid2(:,1),centroid2(:,2),centroid2(:,3),375,[0.5,0,0],'+')
scatter3(centroid3(:,1),centroid3(:,2),centroid3(:,3),375,[0.5,0,0],'+')
hold off
legend([x1 x2 x3 x4],'Cluster 1', 'Cluster 2', 'Cluster 3', 'Init. Centroids'
    )
```

## Exercise 3

### 3.1

#### poly_kernel.m - polynomial kernel mapping

```
% function poly_kernel - Polynomial Kernel mapping.
%
% Inputs - X1 and X2 are matrices in the input space i.e. matrices of
% features computed from training/test sets. c=0 in K(x,y) = (x'*y +c)^d so c
% is ignored. degree: the dimension of the polynomial.
%
% Output - [kernel], the polynomial kernel mapping of the inputs.
function [kernel] = poly_kernel(X1,X2, degree)
    kernel = (X1*X2').^degree;
end
```

#### trainperceptron.m - training perceptron algorithm

```
% trainperceptron.m - Implementation of the training perceptron algorithm.
%
% Inputs: train - a set of training inputs X (includes (x_1, y_1)...(x_m, y_m
% )). train_kern - a polynomial kernel formed by the training inputs
% (x_1,...,x_m). alpha - coefficient matrix updates as we cycle through the
% program. digitno - number of different digits available. We generalise
% into a majority network of perceptrons to separate digitno classes.
%
% Outputs: errors - this variable returns the total number of errors made
% during the training process i.e.the number of times the predicted output
% does not match the desired target for selected 2-classifiers.
% alpha - a trained/updated matrix of  coefficients for later use in
% test steps.

function [errors,alpha]= trainperceptron( train, train_kern, alpha, digitno)
    errors=0;
    [m,n] = size(train);
    for i=1:m % online algorithm operates on single example at a time
```

```
        Y_true = train(i,1);  % y_i and rest of train (X_i): input
        weights = zeros(digitno,1); % classifier initialisation
        % single kernel function K(x_t, .) added for each example
        kern_row = train_kern(i,:);
        max_confidence = -1000000000; % dummy value
        max_w=0; % dummy value
        % classifier training: algorithm generalised for digitno classes.
        for j=1:digitno % we train digitno 2-classifiers
            sum=0;
            beta = alpha(j,:); % dummy
            for k=1:length(alpha) % loop over training examples
                % single kernel function scaled by the term alpha
                sum=sum+ beta(k)*kern_row(k); % product added to sum
            end
            % trained weight for a 2-classifier for a training example
            weights(j) = sum;  %w_k = sum(alpha_i*K*x_i,...)%
        end
        for l=0:digitno-1
            if Y_true==l;
                Y=1;  % examples with desired output class given +ive label
            else
                Y=-1;  % otherwise are given negative label
            end
            if Y*weights(l+1)<=0 % if predicted output does not match  target
                % incorrect prediction leads to change in the coeff matrix
                if weights(l+1) <=0
                    alpha(l+1,i)=alpha(l+1,i)-(-1); % for Y=-1 add (in this
                        case).
                else
                    alpha(l+1,i)=alpha(l+1,i)-(1);  % for Y=1 subtract
                end
            end
            % check associated confidence of each 2-classifier/weights (k =w)
            if weights(l+1)>max_confidence % if argmax
                max_w=l; % new most confident 2-classifier
                max_confidence=weights(l+1); % new max_confidence
            end
        end
        if max_w~=Y_true % for selected 2-classifier add to mistake count
            errors=errors+1; % when output does not match target
    end
end
```

### testperceptron.m - testing perceptron algorithm

testperceptron.m

```
% testperceptron.m - Implementation of perceptron algorithm for testing.
% very similar to trainperceptron.m, but only requires the prediction
% step for each example in the test set. Update step is not performed.
%
% train -> a set of testing inputs X (includes (x_1, y_1)...(x_m, y_m
% )). test_kern - a polynomial kernel formed by the test inputs
% (x_1,...,x_m).  weight_mat - weight vector inherited from training phase
% into a majority network of perceptrons to separate digitno classes.
% digitno -> number of classes/ different digits available
%
% Outputs: errors - this variable returns the total number of mistakes made
% during the testing process i.e.the number of times the predicted output
% does not match the desired target.
function [errors] = testperceptron (test, test_kern, alpha, digitno)
```

```
    errors=0;
    [m,n] = size(test);
    for i=1:m
        Y_true=test(i,1);
        weights = zeros(digitno,1); % classifier initialisation
        max_confidence = -1000000000;
        maxi=0;
        kern_row=test_kern(i,:);
        for j=1:digitno % we test digitno 2-classifiers
            sum=0;
            beta = alpha(j,:); % dummy
            for k=1:length(alpha) % loop over test examples
                % single kernel function scaled by the term alpha
                sum=sum+ beta(k)*kern_row(k); % product added to sum
            end
            % test example weights
            weights(j) = sum;  %w_k = sum(alpha_i*K*x_i,...)%
        end
        for l=0:digitno-1
            % update step eliminated
            if weights(l+1)>max_confidence
                maxi=l;
                max_confidence=weights(l+1);
            end
        end
        if maxi~=Y_true;
            errors=errors+1;
        end
    end
end
```

### `extendedtestperceptron.m` - testing algorithm with error analysis functionality

```
% extendedtestperceptron.m - Implementation of perceptron algorithm
% for testing with error analysis functionality.
% very similar to trainperceptron.m, but only requires the prediction
% step for each example in the test set. Update step is not performed.
%
% train -> a set of testing inputs X (includes (x_1, y_1)...(x_m, y_m
% )). test_kern - a polynomial kernel formed by the test inputs
% (x_1,...,x_m).  weight_mat - weight vector inherited from training phase
% into a majority network of perceptrons to separate digitno classes.
% digitno -> number of classes/ different digits available
%
% Outputs: errors - this variable returns the total number of mistakes made
% during the testing process i.e.the number of times the predicted output
% does not match the desired target. error_matrix - this variable returns a
% confusion table presenting which digits have been misclassified with
% which others.
%
function [errors, error_matrix] = extendedtestperceptron(test,...
                                                         test_kern, alpha,
                                                            digitno)
    errors=0;
    % confusion table setup
    error_mat_left = [0  0:digitno-1]'; % leftmost column indexes true labels
        (0...9)
    % rightmost column tracks total number of times a true label (row) is
    % misclassified.
```

```
    error_mat_right = zeros(digitno+1,1);
    % error matrix initialised with 0...9 mid-columns keeping track of the
        number
    % of times a (row) true value has been misclassified for column value.
    error_matrix = [error_mat_left [0:digitno-1;zeros(digitno,digitno)] ...
                                                    error_mat_right];
    % for error_matrix ignore leftmost and righmost zero in top row
    [m,n] = size(test);
    for i=1:m
        Y_true=test(i,1);
        weights = zeros(digitno,1); % classifier initialisation
        max_confidence = -1000000000;
        maxi=0;
        kern_row=test_kern(i,:);
        for j=1:digitno % we test digitno 2-classifiers
            sum=0;
            beta = alpha(j,:); % dummy
            for k=1:length(alpha) % loop over test examples
                % single kernel function scaled by the term alpha
                sum=sum+ beta(k)*kern_row(k); % product added to sum
            end
            % test example weights
            weights(j) = sum;  %w_k = sum(alpha_i*K*x_i,...)%
        end
        for l=0:digitno-1
            % update step eliminated
            if weights(l+1)>max_confidence
                maxi=l;
                max_confidence=weights(l+1);
            end
        end
        if maxi~=Y_true;
            % if prediction wrong, add mistake
            errors=errors+1;
            % update count of times row value misclassified for column value
            error_matrix(Y_true+2,maxi+2)=error_matrix(Y_true+2,maxi+2)+1;
            % update count of total times given row value is misclassified
            error_matrix(Y_true+2,digitno+2)=error_matrix(Y_true+2,digitno+2)
                +1;

        end
    end
end
```

**comp_GI07_A3.m - main file for Exercise 3 Part 1.**

```
clear all
%
% COMPGI07 ASSIGNMENT 2 - QUESTION 3
%
% train = load('dtrain123.dat');
% test = load('dtest123.dat');
%
train = load('ziptrain.dat');
test = load('ziptest.dat');
%
% number of classes/nepochs
%
nepochs = 4;
digitno = 10;
```

```
15 max_degree = 7;
16 %
17 % divide train further into train and validation
18 %
19 [m,n] = size(train);
20 split = round((2/3)*m); % 2:1 train:val split
21 train_set = train(1:split,:);
22 val_set = train(split+1:end,:);
23 [mtrain, ntrain] = size(train_set);
24 [mval,nval] = size(val_set);
25 %
26 fprintf('Using hold-out to compute optimum polynomial degree: \n')
27 %
28 % test error table for different degrees
29 holdout_degree=[2:max_degree; ones(1,max_degree-1)];
30 ESD1 = zeros(mval,2);
31 %
32 %loop for different value of degree
33 for degree=2:max_degree
34     fprintf('Computing validation error for a degree of %d.\n', degree)
35     % good for performance - remove kernel computation from epoch iteration
36     train_kern=poly_kernel(train_set(:,2:end),train_set(:,2:end), degree);
37     val_kern=poly_kernel(val_set(:,2:end),train_set(:,2:end), degree);
38     alpha = zeros(digitno, mtrain);
39     % iterate over epochs
40     for epoch=1:nepochs
41         [training_errors, alpha]=trainperceptron(train_set,train_kern,alpha,
               digitno);
42         fprintf('Training - Epoch %i: %i mistakes out of %i examples:\n', ...
43                 [epoch training_errors mtrain])
44         [test_errors, error_matrix, ESD1]=extendedtestperceptron(val_set, ...
45                                                 val_kern,alpha, digitno,
                                                    ESD1);
46         percentage_error = test_errors*100/mval;
47         fprintf('Testing - Epoch %i: Test error is %f percent.\n\n', [epoch
               percentage_error])
48         % once the final epoch has been reached, print out confusion table
49         if epoch==nepochs
50             holdout_degree(2,degree-1)=percentage_error;
51             disp(error_matrix)
52         end
53     end
54 end
55
56 % print out table of test error vs degree
57 disp(holdout_degree)
58
59 % MODEL SELECTION
60
61 % graph of polynomial kernel degree plotted against validation error.
62 figure
63 plot(holdout_degree(1,:),holdout_degree(2,:))
64 % optimal polynomial kernel degree is d=4.
65 [opt_err optimal_degree]= min(holdout_degree(2,:));
66 optimal_degree = optimal_degree+1;
67 fprintf('Model selection completed. Optimal degree is %d\n', optimal_degree)
68 %
69 % We now retrain our classifier on the dataset train_set + val_set
70 % with the chosen parameter d=4. Hold out method.
```

```
%
holdout_degree_final=[2:max_degree; ones(1,max_degree-1)];
%
fprintf('Retrain classifier on train_set + val_set, d=4.\n')
original_train = train;
[mot,not] = size(original_train);
[mt,nt] = size(test);
min_test_err = 100000000;
%
% for d=4
ESD2 = zeros(mt,2);
fprintf('Computing test error with a polynomial kernel degree of 4.\n')
% Kernel computations
original_train_kern=poly_kernel(original_train(:,2:end),...
                                original_train(:,2:end), optimal_degree);
test_kern=poly_kernel(test(:,2:end),original_train(:,2:end), optimal_degree);
alpha_final = zeros(digitno, mot);
% iterate over epochs
for epoch=1:nepochs
    [tr_errors_final, alpha_final]=trainperceptron(original_train,...
                                    original_train_kern, alpha_final,
                                       digitno);
    fprintf('Training - Epoch %i: %i mistakes out of %i examples:\n', ...
                                [epoch tr_errors_final mot])
    [test_errors_final, final_error_matrix, ESD2]=extendedtestperceptron(test
        , ...
                                        test_kern,alpha_final, digitno,
                                           ESD2);
    percentage_error_final = test_errors_final*100/mt;
    fprintf('Testing - Epoch %i: Test error is %f percent.\n\n', [epoch ...
                                            percentage_error_final])
    if percentage_error_final < min_test_err
        min_test_err = percentage_error_final;
        opt_epoch = epoch;
    end
end
fprintf('The percentage test error after %d epochs is %f percent.\n', ...
                                epoch, percentage_error_final)
fprintf('A min test error of %f percent is obtained after %d test epochs.\n\n
    ', ...
                                min_test_err, opt_epoch)

disp('Confusion table for test set, d=4, 4 epochs: ')
disp(final_error_matrix)

y_true_classes = test(:,1);
occurrence_count = zeros(1,digitno);
for i=1:digitno
    occurrence_count(i) = sum(y_true_classes==y_true_classes(i));
end

relative_error = zeros(4,digitno);
relative_error(1,:) = 0:digitno-1;
relative_error(2,:) = (final_error_matrix(2:digitno+1,digitno+2))';
relative_error(3,:) = occurrence_count;
relative_error(4,:) = relative_error(2,:)./relative_error(3,:)*100;

disp('Test percentage errors for specific classes: ')
disp(relative_error)
```

```
126
127 %
128 % our aim is to find the most difficult to recognise scanned digits. we now
129 % loop over all polynomial degrees and epochs again, this time over the
130 % full training set and full test set. This final step is performed to
131 % keep track of the specific most misclassified scanned digits. in order to
132 % do this, we add variable ESD to the perceptron_test function.
133 %
134 % loop for different value of degree
135 ESD3 = zeros(mt,2);
136 for degree=2:max_degree
137     fprintf('Computing the most misclassified scanned digits for degree of %d
        .\n',...
138                                                                     degree)
139     % good for performance - remove kernel computation from epoch iteration
140     original_train_kern=poly_kernel(original_train(:,2:end),...
141                                     original_train(:,2:end), degree);
142     test_kern=poly_kernel(test(:,2:end),original_train(:,2:end), degree);
143     alpha_final = zeros(digitno, mot);
144     % iterate over epochs
145     for epoch=1:nepochs
146         [tr_errors_final, alpha_final]=trainperceptron(original_train,...
147                                       original_train_kern, alpha_final,
                                          digitno);
148         [test_errors_final, final_error_matrix, ESD3]=extendedtestperceptron(
            test, ...
149                                           test_kern,alpha_final, digitno,
                                              ESD3);
150     end
151 end
152 % ESD3 returned - table with all specific scanned digit indexes and their
153 % respective number of mistakes.
154
155 % top_number specifies the number of top entries that are displayed
156 top_number=5;
157 % sort ESD in descending order
158 [mistake_rank, hardest_to_recognize] = sort(ESD3(:,2),1,'descend');
159 fprintf('The 5 hardest-to-recognize digits over 6 degree iterations (4 epochs
        each) are: \n')
160 % table with scanned digit entries and their mistakes in descending order
161 hard2rec = [hardest_to_recognize'; mistake_rank'];
162 % table only featuring top 5 most misclassified scanned digit records.
163 top5 = hard2rec(:,1:top_number);
164 disp(top5);
165
166 % display printoffs
167 % the true label of the printoffs is displayed in the command window
168
169 % these are plotted after inspecting the top5 to find entries
170 figure;
171 subplot(2,3,1)
172 fprintf('True label of digit is %i\n\n', test(18,1))
173 imagesc(reshape(test(18,2:end), 16, 16)'); colormap 'gray';
174 subplot(2,3,2)
175 fprintf('True label of digit is %i\n\n', test(123,1))
176 imagesc(reshape(test(123,2:end), 16, 16)'); colormap 'gray';
177 subplot(2,3,3)
178 fprintf('True label of digit is %i\n\n', test(135,1))
179 imagesc(reshape(test(135,2:end), 16, 16)'); colormap 'gray';
```

```
180 subplot(2,3,4)
181 fprintf('True label of digit is %i\n\n', test(199,1))
182 imagesc(reshape(test(199,2:end), 16, 16)'); colormap 'gray';
183 subplot(2,3,6)
184 fprintf('True label of digit is %i\n\n', test(234,1))
185 imagesc(reshape(test(234,2:end), 16, 16)'); colormap 'gray';
```

**comp_GI07_A3_2.m - main file for Exercise 3 Part 2 (Gaussian).**

```
1  clear all
2  %
3  % COMPGI07 ASSIGNMENT 2 - QUESTION 3 (GAUSSIAN)
4  %
5  % train = load('dtrain123.dat');
6  % test = load('dtest123.dat');
7  %
8  train = load('ziptrain.dat');
9  test = load('ziptest.dat');
10 %
11 % number of classes/nepochs
12 %
13 nepochs = 2;
14 digitno = 10;
15 max_degree = 7;
16 %
17 % divide train further into train and validation
18 %
19 [m,n] = size(train);
20 split = round((2/3)*m); % 2:1 train:val split
21 train_set = train(1:split,:);
22 val_set = train(split+1:end,:);
23 [mtrain, ntrain] = size(train_set);
24 [mval,nval] = size(val_set);
25 %
26 fprintf('Using hold-out to compute optimum polynomial degree: \n')
27 %
28 % test error table for different degrees
29 holdout_degree=[2:max_degree; ones(1,max_degree-1)];
30 ESD1 = zeros(mval,2);
31 %
32 %loop for different value of degree
33 for degree=2:max_degree
34     fprintf('Computing validation error for a degree of %d.\n', degree)
35     % good for performance - remove kernel computation from epoch iteration
36     train_kern=gaussian_kernel(train_set(:,2:end),train_set(:,2:end), degree)
           ;
37     val_kern=gaussian_kernel(val_set(:,2:end),train_set(:,2:end), degree);
38     alpha = zeros(digitno, mtrain);
39     % iterate over epochs
40     for epoch=1:nepochs
41         [training_errors, alpha]=trainperceptron(train_set,train_kern,alpha,
               digitno);
42         fprintf('Training - Epoch %i: %i mistakes out of %i examples:\n', ...
43                 [epoch training_errors mtrain])
44         [test_errors, error_matrix, ESD1]=extendedtestperceptron(val_set, ...
45                                                 val_kern,alpha, digitno,
                                                       ESD1);
46         percentage_error = test_errors*100/mval;
47         fprintf('Testing - Epoch %i: Test error is %f percent.\n\n', [epoch
               percentage_error])
```

```
48          % once the final epoch has been reached, print out confusion table
49          if epoch==nepochs
50              holdout_degree(2,degree-1)=percentage_error;
51              disp(error_matrix)
52          end
53      end
54  end
55
56  % print out table of test error vs degree
57  disp(holdout_degree)
58
59  % MODEL SELECTION
60
61  % graph of polynomial kernel degree plotted against validation error.
62  figure
63  plot(holdout_degree(1,:),holdout_degree(2,:))
64  % optimal polynomial kernel degree is d=4.
65  [opt_err optimal_degree]= min(holdout_degree(2,:));
66  optimal_degree = optimal_degree+1;
67  fprintf('Model selection completed. Optimal degree is %d\n', optimal_degree)
68  %
69  % We now retrain our classifier on the dataset train_set + val_set
70  % with the chosen parameter d=4. Hold out method.
71  %
72  holdout_degree_final=[2:max_degree; ones(1,max_degree-1)];
73  %
74  fprintf('Retrain classifier on train_set + val_set, d=4.\n')
75  original_train = train;
76  [mot,not] = size(original_train);
77  [mt,nt] = size(test);
78  min_test_err = 100000000;
79  %
80  % for d=4
81  ESD2 = zeros(mt,2);
82  fprintf('Computing test error with a polynomial kernel degree of 4.\n')
83  % Kernel computations
84  original_train_kern=gaussian_kernel(original_train(:,2:end),...
85                                      original_train(:,2:end), optimal_degree);
86  test_kern=gaussian_kernel(test(:,2:end),original_train(:,2:end),
        optimal_degree);
87  alpha_final = zeros(digitno, mot);
88  % iterate over epochs
89  for epoch=1:nepochs
90      [tr_errors_final, alpha_final]=trainperceptron(original_train,...
91                                          original_train_kern, alpha_final,
                                                digitno);
92      fprintf('Training - Epoch %i: %i mistakes out of %i examples:\n', ...
93                                      [epoch tr_errors_final mot])
94      [test_errors_final, final_error_matrix, ESD2]=extendedtestperceptron(test
            , ...
95                                              test_kern,alpha_final, digitno,
                                                  ESD2);
96      percentage_error_final = test_errors_final*100/mt;
97      fprintf('Testing - Epoch %i: Test error is %f percent.\n\n', [epoch ...
98                                                  percentage_error_final])
99      if percentage_error_final < min_test_err
100         min_test_err = percentage_error_final;
101         opt_epoch = epoch;
102     end
```

```
103 end
104 fprintf('The percentage test error after %d epochs is %f percent.\n', ...
105                                     epoch, percentage_error_final)
106 fprintf('A min test error of %f percent is obtained after %d test epochs.\n\n
    ', ...
107                                     min_test_err, opt_epoch)
108
109 disp('Confusion table for test set, d=6, 2 epochs: ')
110 disp(final_error_matrix)
111
112 y_true_classes = test(:,1);
113 occurrence_count = zeros(1,digitno);
114 for i=1:digitno
115     occurrence_count(i) = sum(y_true_classes==y_true_classes(i));
116 end
117
118 relative_error = zeros(4,digitno);
119 relative_error(1,:) = 0:digitno-1;
120 relative_error(2,:) = (final_error_matrix(2:digitno+1,digitno+2))';
121 relative_error(3,:) = occurrence_count;
122 relative_error(4,:) = relative_error(2,:)./relative_error(3,:)*100;
123
124 disp('Test percentage errors for specific classes: ')
125 disp(relative_error)
126
127 %
128 % our aim is to find the most difficult to recognise scanned digits. we now
129 % loop over all polynomial degrees and epochs again, this time over the
130 % full training set and full test set. This final step is performed to
131 % keep track of the specific most misclassified scanned digits. in order to
132 % do this, we add variable ESD to the perceptron_test function.
133 %
134 % loop for different value of degree
135 ESD3 = zeros(mt,2);
136 for degree=2:max_degree
137     fprintf('Computing the most misclassified scanned digits for degree of %d
        .\n',...
138                                                                 degree)
139     % good for performance - remove kernel computation from epoch iteration
140     original_train_kern=gaussian_kernel(original_train(:,2:end),...
141                                 original_train(:,2:end), degree);
142     test_kern=gaussian_kernel(test(:,2:end),original_train(:,2:end), degree);
143     alpha_final = zeros(digitno, mot);
144     % iterate over epochs
145     for epoch=1:nepochs
146         [tr_errors_final, alpha_final]=trainperceptron(original_train,...
147                                     original_train_kern, alpha_final,
                                        digitno);
148         [test_errors_final, final_error_matrix, ESD3]=extendedtestperceptron(
            test, ...
149                                         test_kern,alpha_final, digitno,
                                            ESD3);
150     end
151 end
152 % ESD3 returned - table with all specific scanned digit indexes and their
153 % respective number of mistakes.
154
155 % top_number specifies the number of top entries that are displayed
156 top_number=5;
```

```
157 % sort ESD in descending order
158 [mistake_rank, hardest_to_recognize] = sort(ESD3(:,2),1,'descend');
159 fprintf('The 5 hardest-to-recognize digits over 6 degree iterations (2 epochs
        each) are: \n')
160 % table with scanned digit entries and their mistakes in descending order
161 hard2rec = [hardest_to_recognize'; mistake_rank'];
162 % table only featuring top 5 most misclassified scanned digit records.
163 top5 = hard2rec(:,1:top_number);
164 disp(top5);
165
166 % display printoffs
167 % the true label of the printoffs is displayed in the command window
168
169 % these are plotted after inspecting the top5
170 figure;
171 subplot(2,3,1)
172 fprintf('True label of digit is %i\n\n', test(18,1))
173 imagesc(reshape(test(18,2:end), 16, 16)'); colormap 'gray';
174 subplot(2,3,2)
175 fprintf('True label of digit is %i\n\n', test(28,1))
176 imagesc(reshape(test(28,2:end), 16, 16)'); colormap 'gray';
177 subplot(2,3,3)
178 fprintf('True label of digit is %i\n\n', test(123,1))
179 imagesc(reshape(test(123,2:end), 16, 16)'); colormap 'gray';
180 subplot(2,3,4)
181 fprintf('True label of digit is %i\n\n', test(146,1))
182 imagesc(reshape(test(146,2:end), 16, 16)'); colormap 'gray';
183 subplot(2,3,6)
184 fprintf('True label of digit is %i\n\n', test(165,1))
185 imagesc(reshape(test(165,2:end), 16, 16)'); colormap 'gray';
```