Artificial Intelligence: Knowledge Representation and Planning

CM0472: Assignment 3 – Clustering

JABER RAHIMIFARD [875545]

AREF ENAYATI [882341]

Assignment

Perform classification of the Semeion Handwritten Digit data set using:

```
-Latent class analysis;
-Mean shift;
-Normalized cut.
```

Mean shift and **Normalized Cut** assume that the images are vectors in a 256 dimensional Euclidean space.

Provide the code and the extracted clusters as the number of clusters k varies **from 5 to 15**, for **LCA and normalized-cut**, while for **Mean shift** vary the kernel width. For each value of k (or kernel width) provide the value of the Rand index:

R=2(a+b)/(n(n-1))

where
n is the number of images in the dataset.

a is the number of pairs of images that represent the same digit and that are clustered together.

is the number of pairs of images that represent different digits and that are placed in different clusters.
Explain the differences between the three models.

Tip: Bernoulli models can be visualized as a greyscale image to inspect the learned model.

Overview

In this assignment, we perform classification by using 3 algorithms **LCA**, **Mean Shift**, and **Normalized Cut** on the Semeion Handwritten Digit dataset.
We modify the number of clusters in each cluster and the width band in case using mean shift algorithm.

For each cluster, we run the __rand score__, __adjusted rand score__ and compare them.

Also, Survey the averages by the __number of clusters__, and the __number of DR models__.
And in the final show, the **best ARI** achieved.

Clustering

We have Supervised learning, Unsupervised learning and Semi supervised learning which each of them has different method to do. Clustering is one of the unsupervised learning method that trained a machine using information are not labeled. The task of clustering algorithms is divide the collection of data points or objects into number of groups with similar role or characterstics.

Code Explaination

First we call data from dataset we downloaded before by the function get_data(). By the class NoDR() we make a fake class for no dimension reduction.

```
In [ ]:
def get_data():
    data = np.loadtxt('./semeion.data', dtype=np.int8)
    return data[:, :256], data[:, 256:]


def one_hot_decode(y: np.array):
    return np.argmax(y, axis=1)


def get_data_transformed():
    data = np.loadtxt('./semeion.data', dtype=np.int8)
    return data[:, :256], one_hot_decode(data[:, 256:])
```

```
In [ ]:
class NoDR:
    def __init__(self, n_components):
        pass

    def fit_transform(self, X):
        return X.copy()
```

ks is a vector containing the number of cluster we want for assignment. And, **ds** is a vector containing the dimension for dimensionality reduction.

```
In [ ]:
simplefilter(action='ignore', category=FutureWarning)

ks = range(5, 16)
ds = (2, 64, 128, 256)
```

1- LCA :

Latent Class Analysis or **LCA** is a way to uncover hidden groupings in data. More specifically, it's a way to to group subjects from multivariate data into "latent classes" — groups or subgroups with similar, unobservable, membership.

-Latent implies that the analysis is based on an error-free latent variable (Collins & Lanza, 2013).
-Classes are groups formed by uncovering hidden (latent) patterns in data.

LCA find the connections between the objects and group them like cluster analysis.

Now, A glance view to LCA code and briefly explanation the code, after that the explaining the performance and result.

The function **_log_no_underflow** computes log without underflow which has a parameter x which should be float.

```
In [ ]:
def _log_no_underflow(x):
    return np.log(np.clip(x, 1e-15, 1))
```

The function **estimate_bernoulli_parameters** estimate the bernouli distribution parameters. Parameter **X** is the input data array. **resp** array has the responsibility for each data sample in X. **nk** is the numbers of data samples in the current components and **means** is the centers of the current components.

```
In [2]:
def _estimate_bernoulli_parameters(X, resp):
    nk = resp.sum(axis=0) + 10 * np.finfo(resp.dtype).eps
    means = np.dot(resp.T, X) / nk[:, np.newaxis]
    return nk, means
```

The function **estimate_log_bernouli_prob** has the duty of estimate the log bernoulli probability. It has **X** and **means** parameters.

```
In [ ]:
def _estimate_log_bernouli_prob(X, means):
    return (X.dot(_log_no_underflow(means).T) +
            (1 - X).dot(_log_no_underflow(1 - means).T))
```

Since **Bernoulli_mixture class** has a lot of function, I decide to explain it inside the code.

```
In [ ]:
class Bernoulli_Mixture:
    def __init__(self, n_components, max_iter=500, tol=1e-3):
        self.n_components = n_components
        self.max_iter = max_iter
        self.tol = tol

    def __str__(self):
        return 'BMM'

    def fit(self, X, init_kmeans=True):

        """Estimate model parameters using X and predict the labels for X.
        Parameter X is a list of n_features-dimentional data points. each row corresponds to a single data poi
        """
        self.converged_ = False

        n_samples, _ = X.shape
        random_state = 42
        if init_kmeans:
            resp = np.zeros((n_samples, self.n_components))
            label = (
                KMeans(n_clusters=self.n_components, n_init=1, random_state=random_state)
                    .fit(X)
                    .labels_
            )
            resp[np.arange(n_samples), label] = 1
        else:
            resp = random_state.rand(n_samples, self.n_components)
            resp /= resp.sum(axis=1)[:, np.newaxis]

        self._initialize(X, resp)

        lower_bound = -np.inf

        for n_iter in range(1, self.max_iter + 1):
            prev_lower_bound = lower_bound

            log_prob_norm, log_resp = self._e_step(X)
            self._m_step(X, log_resp)
            lower_bound = log_prob_norm

            change = lower_bound - prev_lower_bound

            if abs(change) < self.tol:
                self.converged_ = True
                break

        if not self.converged_:
            raise ValueError('Not converged')

        _, log_resp = self._e_step(X)

        self.labels_ = log_resp.argmax(axis=1)

    def _e_step(self, X):
        """
        -log_prob_norm which is float value has duty to get Mean of the
         logarithms of the probabilities of each sample in X.

        -log_responsibility get the Logarithm of the posterior probabilities (or responsibilities) of
         the point of each sample in X.
        """
        log_prob_norm, log_resp = self._estimate_log_prob_resp(X)
        return np.mean(log_prob_norm), log_resp

    def _estimate_weighted_log_prob(self, X):
        """
        Estimate the weighted log-probabilities, log P(X | Z) + log weights.
        """
        return self._estimate_log_prob(X) + self._estimate_log_weights()

    def _estimate_log_prob_resp(self, X):
        """
        Estimate log probabilities and responsibilities for each sample.
        Compute the log probabilities, weighted log probabilities for
        component and responsibilities for each sample in X with respect to
        the current state of the model.
        """
        weighted_log_prob = self._estimate_weighted_log_prob(X)
        log_prob_norm = logsumexp(weighted_log_prob, axis=1)
        with np.errstate(under="ignore"):
            # ignore underflow
            log_resp = weighted_log_prob - log_prob_norm[:, np.newaxis]
        return log_prob_norm, log_resp

    def _initialize(self, X, resp):
        """Initialization of the Bernoulli mixture parameters.
        """
        n_samples, n_dim = X.shape

        weights, means = _estimate_bernoulli_parameters(X, resp)
        weights /= n_samples

        self.weights_ = weights
        self.means_ = means

    def _m_step(self, X, log_resp):
        """get the Logarithm of the posterior probabilities (or responsibilities) of
            the point of each sample in X.
        """
        n_samples, _ = X.shape
        self.weights_, self.means_ = (
            _estimate_bernoulli_parameters(X, np.exp(log_resp))
        )
        self.weights_ /= n_samples

    def _estimate_log_prob(self, X):
        return _estimate_log_bernoulli_prob(X, self.means_)

    def _estimate_log_weights(self):
        return np.log(self.weights_)
```

The **LCA** function read and split data and by using loop clustering model compute dimension, number of cluster, time, adjusted Rand Index and Rand Index and record and save them in a LCA.csv file.

```
In [ ]:
def LCA(p=False):
    columns = ('clustering_model', 'dim_red_model', 'd', 'k', 'time', 'ARI', 'RI')
    rows = []

    # read data and split
    X, y = get_data_transformed()
    # loop clusteting model
    for k in ks:
        for d in ds:
            if d != 256:
                embedding_model = PCA
            else:
                embedding_model = NoDR
            start_time = time()
            embedding = embedding_model(n_components=d)
            X_transformed = embedding.fit_transform(X)
            model = Bernoulli_Mixture(k)
            model.fit(X_transformed)
            elapsed = time() - start_time
            title = f'Results for {model} on {d}-{embedding_model.__name__} - k={k} ({elapsed:.02f}s)'
            print(title)
            ari, ri = print_rand(y, model.labels_)
            if p: plot2D(X_transformed, model.labels_, title)
            result = (str(model), embedding_model.__name__, d, k, elapsed, ari, ri)
            rows.append(result)
    df = pd.DataFrame(data=rows, columns=columns)
    print(df)
    df.to_csv('LCA.csv', index=False)
```

1.1 Explaining the performance and result

You can see all the results by using algorithm LCA in the left side and in the right side we compute the averages by the number of cluster, number of DR models and Best Rand index with more than 90%.

Results

Dimension	Cluster	ARI	RI	Time
2-PCA	5	16.07 %	72.41 %	0.064393
64-PCA	5	18.72 %	78.48 %	0.150597
128-PCA	5	17.35 %	78.45 %	0.307651
256-NoDR	5	25.26 %	79.95 %	0.178798
2-PCA	6	17.13 %	75.92 %	0.070811
64-PCA	6	24.76 %	81.9 %	0.185023
128-PCA	6	23.66 %	81.98 %	0.272484
256-NoDR	6	30.68 %	83.59 %	0.658985
2-PCA	7	17.06 %	76.77 %	0.078182
64-PCA	7	28.9 %	84.63 %	0.174242
128-PCA	7	26.31 %	84.1 %	0.331657
256-NoDR	7	34.04 %	84.05 %	0.251229
2-PCA	8	17.54 %	76.57 %	0.070812
64-PCA	8	31.89 %	86.24 %	0.173881
128-PCA	8	27.07 %	85.37 %	0.299007
256-NoDR	8	35.04 %	86.3 %	0.459920
2-PCA	9	17.16 %	75.87 %	0.122036
64-PCA	9	27.34 %	86.06 %	0.203799
128-PCA	9	31.28 %	86.77 %	0.255027
256-NoDR	9	41.09 %	88.44 %	0.254372
2-PCA	10	17.38 %	76.02 %	0.080753
64-PCA	10	33.41 %	87.86 %	0.143551
128-PCA	10	28.36 %	86.99 %	0.274869
256-NoDR	10	45.81 %	90.09 %	0.502467
2-PCA	11	18.33 %	78.96 %	0.093763
64-PCA	11	39.71 %	89.5 %	0.149052
128-PCA	11	34.04 %	88.52 %	0.333906
256-NoDR	11	46.38 %	90.59 %	0.285237
2-PCA	12	18.28 %	79.03 %	0.098734
64-PCA	12	32.41 %	88.68 %	0.140141
128-PCA	12	37.9 %	89.57 %	0.416136
256-NoDR	12	43.42 %	90.51 %	0.510908
2-PCA	13	17.63 %	76.68 %	0.109713
64-PCA	13	35.56 %	89.51 %	0.176716
128-PCA	13	31.48 %	88.87 %	0.386177
256-NoDR	13	42.54 %	90.67 %	0.515680
2-PCA	14	18.63 %	79.51 %	0.081914
64-PCA	14	30.47 %	89.03 %	0.243644
128-PCA	14	33.78 %	89.56 %	0.415308
256-NoDR	14	42.91 %	90.93 %	0.486732
2-PCA	15	18.27 %	79.96 %	0.110071
64-PCA	15	33.3 %	89.69 %	0.160934
128-PCA	15	26.73 %	88.61 %	0.436832
256-NoDR	15	41.44 %	90.89 %	0.398509

Averages By N. Clusters

K	Time	ARI	RI
11	0.215489	0.346162	0.868934
12	0.291480	0.330023	0.869482
13	0.297071	0.318029	0.864332
14	0.306899	0.314457	0.872589
10	0.250410	0.312417	0.852417
15	0.276586	0.299348	0.872843
9	0.208808	0.292170	0.842836
8	0.250905	0.278825	0.836186
7	0.208827	0.265759	0.823894
6	0.296826	0.240595	0.808474
5	0.175360	0.193511	0.773210

Averages By DR Models

DR Model	Time	ARI	RI
NoDR	0.409349	0.389655	0.878179
PCA	0.200358	0.256938	0.832752

Best ARI Achieved By LCA

DR Model	Dimension	N. Clusters	Time	ARI	RI
NoDR	256	11	0.285237	0.463818	0.905877
NoDR	256	10	0.502467	0.458088	0.900913
NoDR	256	12	0.510908	0.434214	0.905120

2- Mean Shift :

Mean shift clustering aims to discover "blobs" in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Seeding is performed using a binning technique for scalability.

*Class **Mean_Shift** using mean shift segmentation algorithm which has (int)Bandwindth parameter which is the number of neighbors considered.
we use n_jobs=1 because this will be used in nested calls under parallel calls to mean_shift_single_seed. so we don't need for future parallelism.
all_res execute iterations on all seeds in parallel and copy results in a dictionary.

post processing remove near duplicate points. If the distance between two kernels is less than the bandwidth, then we have to remove one because it is a duplicate. Remove the one with fewer points.

```
In [ ]:
class Mean_Shift:
    def __init__(self, bandwidth, max_iter=200, n_jobs=4):
        self.bandwidth = bandwidth
        self.max_iter = max_iter
        self.n_jobs = n_jobs

    def __str__(self):
        return 'MS'

    def fit(self, X):
        seeds = X
        n_samples, n_features = X.shape
        center_intensity_dict = {}
        nbrs = NearestNeighbors(radius=self.bandwidth, n_jobs=1).fit(X)
        all_res = Parallel(n_jobs=self.n_jobs)(
            delayed(_mean_shift_single_seed)
            (seed, X, nbrs, self.max_iter) for seed in seeds)

        for i in range(len(seeds)):
            if all_res[i][1]:  # i.e. len(points_within) > 0
                center_intensity_dict[all_res[i][0]] = all_res[i][1]

        self.n_iter_ = max([x[2] for x in all_res])

        if not center_intensity_dict:
            # nothing near seeds
            raise ValueError("No point was within bandwidth=%f of any seed."
                             " Try a different seeding strategy \
                             or increase the bandwidth."
                             % self.bandwidth)
        sorted_by_intensity = sorted(center_intensity_dict.items(),
                                     key=lambda tup: (tup[1], tup[0]),
                                     reverse=True)
        sorted_centers = np.array([tup[0] for tup in sorted_by_intensity])
        unique = np.ones(len(sorted_centers), dtype=bool)
        nbrs = NearestNeighbors(radius=self.bandwidth,
                                n_jobs=self.n_jobs).fit(sorted_centers)
        for i, center in enumerate(sorted_centers):
            if unique[i]:
                neighbor_idxs = nbrs.radius_neighbors([center],
                                                      return_distance=False)[0]
                unique[neighbor_idxs] = 0
                unique[i] = 1  # leave the current point as unique
        cluster_centers = sorted_centers[unique]

        # ASSIGN LABELS: a point belongs to the cluster that it is closest to
        nbrs = NearestNeighbors(n_neighbors=1,
                                n_jobs=self.n_jobs).fit(cluster_centers)
        labels = np.zeros(n_samples, dtype=int)
        idxs = nbrs.kneighbors(X, return_distance=False)
        labels = idxs.flatten()

        self.cluster_centers_, self.labels_, self.nlabels_ = cluster_centers, labels, len(np.unique(labels))
        return self
```

the **mean_shift_single_seed** function is a separate function for each seed iterative loop. For each seed, climb gradient until convergence. if converged or at max_iter, it add the cluster.

```
In [ ]:
def _mean_shift_single_seed(my_mean, X, nbrs, max_iter):
    bandwidth = nbrs.get_params()['radius']
    stop_thresh = 1e-3 * bandwidth
    completed_iterations = 0
    while True:
        i_nbrs = nbrs.radius_neighbors([my_mean], bandwidth,
                                       return_distance=False)[0]
        points_within = X[i_nbrs]
        if len(points_within) == 0:
            break
        my_old_mean = my_mean
        my_mean = np.mean(points_within, axis=0)
        if (np.linalg.norm(my_mean - my_old_mean) < stop_thresh or
                completed_iterations == max_iter):
            break
        completed_iterations += 1
    return tuple(my_mean), len(points_within), completed_iterations
```

The **MS** function read and split data and by using loop clustering model compute dimension, number of cluster, time, adjusted Rand Index and Rand Index and record and save them in a MS.csv file.

```
In [ ]:
def MS(p=False):
    columns = ('clustering_model','dim_red_model','d','k','time','ARI','RI')
    rows = []
    params = ((1, 2), (6.3, 64), (7.5, 128), (7, 256))
    X, y = get_data_transformed()
    for w,d in params:
        model = Mean_Shift(w)
        start_time = time()
        if d != 256:
            embedding_model = PCA
        else:
            embedding_model = NoDR
        X_transformed = embedding_model(n_components=d).fit_transform(X)
        ename = embedding_model.__name__
        model.fit(X_transformed)
        elapsed = time() - start_time
        title = f'Results for {model} on dim={d} ({elapsed:.02f}s)\n' +\
            f'number of centers: {model.nlabels_} - window size: {w}'
        labels = model.labels_
        print(title)
        ari, ri = print_rand(y, model.labels_)
        if p: plot2D(X_transformed, model.labels_, title)
        result = (str(model),ename,d, w, elapsed, ari, ri)
        rows.append(result)
    df = pd.DataFrame(data=rows, columns=columns)
    print(df)
    df.to_csv('MS.csv', index=False)
```

2.1 Explaining the performance and result

You can see all the results at the bottom by using algorithm Mean Shift. We compute the averages by the number of cluster, number of DR models and Best Rand index which it is less than LCA algoritm.

Results

Dim	N.Centers	Window size	ARI	RI	Time
2-PCA	8	1.0	0.206825	0.818561	6.913914
64-PCA	40	6.3	0.036450	0.515360	6.353009
128-PCA	13	7.5	0.000115	0.118547	16.814545
256-NoDR	250	7.0	0.064224	0.695134	31.511524

Averages By N.Centers

K	Time	ARI	RI
1	6.913914	0.206825	0.818561
7.0	31.511524	0.064224	0.695134
6.3	6.353009	0.036450	0.515360
7.5	16.814545	0.000115	0.118547

Averages By DR Models

DR Model	Time	ARI	RI
PCA	10.027156	0.081130	0.484156
NoDR	31.511524	0.064224	0.695134

Best ARI Achieved By MS

DR Model	Dimension	N.Centers	Time	ARI	RI
PCA	2	1.0	6.913914	0.206825	0.818561
NoDR	256	7.0	31.511524	0.064224	0.695134
PCA	64	6.3	6.353009	0.036450	0.515360

3- Normalized Cut :

The basic idea of this problem is that we want to find the weights in that graph that minimize the cut. In doing so we found a big big problem because to find the 2 clusters and the cut we do not take into consideration what happens in between the clusters and so we will find two unbalanced clusters. Normalized cut is the CUT problem that takes into consideration what happens in between the clusters. It does the CUT as before but takes into consideration the volume of A and the volume of B in such a way that now it creates the balanced clusters.

Class **N_Cut** and specially fit function inside has duty to find largest and smallest eigenvalues and nonzero eigenvalues.

```
In [ ]:
class N_Cut():
    def __init__(self, n_clusters):
        self.n_clusters = n_clusters


    def __str__(self):
        return 'NCUT'


    def fit(self, X, verbose=False):
        self.n_samples, self.n_features = X.shape

        A = rbf_kernel(X)

        np.fill_diagonal(A, 0.0)

        G = nx.from_numpy_matrix(A)
        L = nx.normalized_laplacian_matrix(G)
        w = np.linalg.eigvals(L.A)
        if verbose:
         print("Largest eigenvalue:", max(w))
         print("Smallest eigenvalue:", min(w))
         print(f"first k={self.n_clusters} nonzero eigenvalues:", w[1:self.n_clusters+1])

        w, v = np.linalg.eig(L.A)

        if not abs(w[0]) < 1e-10:
            raise ValueError(f'First eigenvalue is {w[0]}.\nMust be close to zero.')

        U = v[:, 1:self.n_clusters+1]

        kmeans = KMeans(n_clusters=self.n_clusters).fit(U)

        self.labels_ = kmeans.labels_
```

The **NCut_Result** function read and split data and by using loop clustering model compute dimension, number of cluster, time, adjusted Rand Index and Rand Index and record and save them in a NCut.csv file.

```
In [ ]:
def NCut_Result(p=False):
    columns = ('clustering_model', 'dim_red_model', 'd', 'k', 'time', 'ARI', 'RI')
    rows = []
    X, y = get_data_transformed()
    for k in ks:
        for d in ds:
            if d != 256:
                embedding_model = PCA
            else:
                embedding_model = NoDR
            start_time = time()
            embedding = embedding_model(n_components=d)
            X_transformed = embedding.fit_transform(X)
            model = N_Cut(k)  # clustering model
            model.fit(X_transformed)
            elapsed = time() - start_time
            title = f'Results for {model} on {d}-{embedding_model.__name__} - k={k} ({elapsed:.02f}s)'
            print(title)
            ari, ri = print_rand(y, model.labels_)
            if p: plot2D(X_transformed, model.labels_, title)
            result = (str(model), embedding_model.__name__, d, k, elapsed, ari, ri)
            rows.append(result)
    df = pd.DataFrame(data=rows, columns=columns)
    print(df)
    df.to_csv('NCut.csv', index=False)
```

3.1 Explaining the performance and result

You can see all the results by using algorithm **NCut** in the left side and in the right side we compute the averages by the number of cluster, number of DR models and Best Rand index with more than 86% which is almost similar to LCA.

Results

Dimension	Cluster	ARI	RI	Time
2-PCA	5	0.199028 %	0.767762 %	13.864685
64-PCA	5	0.242466 %	0.797952 %	12.649546
128-PCA	5	0.237510 %	0.796779 %	12.151481
256-NoDR	5	0.236645 %	0.796717 %	12.018857
2-PCA	6	0.213673 %	0.795057 %	12.137772
64-PCA	6	0.287075 %	0.826352 %	12.311077
128-PCA	6	0.302717 %	0.833327 %	12.623976
256-NoDR	6	0.296321 %	0.832270 %	11.930096
2-PCA	7	0.196427 %	0.789850 %	12.056757
64-PCA	7	0.332831 %	0.853275 %	12.052766
128-PCA	7	0.333116 %	0.853612 %	11.903321
256-NoDR	7	0.345580 %	0.855968 %	11.855295
2-PCA	8	0.214157 %	0.825093 %	12.412082
64-PCA	8	0.370219 %	0.869161 %	12.216820
128-PCA	8	0.356182 %	0.866370 %	12.219285
256-NoDR	8	0.366834 %	0.869493 %	12.084681
2-PCA	9	0.190103 %	0.832990 %	12.319028
64-PCA	9	0.347881 %	0.873466 %	11.895190
128-PCA	9	0.293015 %	0.865515 %	12.042794
256-NoDR	9	0.356792 %	0.876436 %	11.998960
2-PCA	10	0.200573 %	0.830980 %	12.412806
64-PCA	10	0.336301 %	0.876912 %	12.134358
128-PCA	10	0.341982 %	0.878455 %	11.960045
256-NoDR	10	0.337744 %	0.879858 %	11.970470
2-PCA	11	0.191930 %	0.843083 %	12.087674
64-PCA	11	0.327393 %	0.881370 %	12.299681
128-PCA	11	0.294857 %	0.877650 %	11.979205
256-NoDR	11	0.364965 %	0.888283 %	13.644511
2-PCA	12	0.206013 %	0.847036 %	14.133188
64-PCA	12	0.356897 %	0.889239 %	14.206896
128-PCA	12	0.314877 %	0.883678 %	13.555822
256-NoDR	12	0.308342 %	0.883836 %	13.001232
2-PCA	13	0.187552 %	0.852909 %	12.549751
64-PCA	13	0.366277 %	0.895123 %	13.061071
128-PCA	13	0.318798 %	0.888448 %	12.440696
256-NoDR	13	0.337868 %	0.892718 %	12.945152
2-PCA	14	0.182451 %	0.858124 %	12.685079
64-PCA	14	0.357432 %	0.898257 %	12.694050
128-PCA	14	0.341937 %	0.895827 %	13.089947
256-NoDR	14	0.334667 %	0.894904 %	13.076824
2-PCA	15	0.179252 %	0.860609 %	13.340178
64-PCA	15	0.343292 %	0.898642 %	12.389898
128-PCA	15	0.325645 %	0.895513 %	12.382390
256-NoDR	15	0.331618 %	0.896514 %	12.610276

Averages By N. Clusters

K	Time	ARI	RI
8	12.233217	0.326848	0.857529
10	12.119420	0.304150	0.866551
14	12.886475	0.304122	0.886778
13	12.749168	0.302624	0.882300
7	11.967035	0.301988	0.838176
9	12.063993	0.296948	0.862102
12	13.724284	0.296532	0.875947
15	12.680686	0.294952	0.887819
11	12.502768	0.294786	0.872597
6	12.250730	0.274947	0.821752
5	12.671142	0.228912	0.789803

Averages By DR Models

DR Model	Time	ARI	RI
NoDR	12.466941	0.328852	0.869727
PCA	12.553313	0.281511	0.854497

Best ARI Achieved By NCut

DR Model	Dimension	N. Clusters	Time	ARI	RI
PCA	64	8	12.216820	0.370219	0.869161
NoDR	256	8	12.084681	0.366834	0.869493
PCA	64	13	13.061071	0.366277	0.895123

Compare Algorithms :

As you can see by the results the best classifier used in this assignment is the **LCA** by looking the data. After that, second place by the results we export is **Normalized Cut** which almost near and similar to LCA. and the third place by the results belongs to **MeanShift**. After discussions, we agreed that we need more input data to make clustering better. Also, if we have to chance to choose one of the these algorithms, clearly we choose Normalized cut since it is really easy to understand how it works and also where are the problem.