Machine Learning
Homework #3

Daniel Ho Kwan Leung
104360098
CSIE, Year 3

Question 1:

The solution was computed using scikit-learn module or library of Python3. Since the internal parameters of the linear transformation matrix could not be obtained from the library, I used the entire dataset of 150 samples to find the corresponding independent components.

The following are the code implementation ICA for Iris dataset. Code can also be found via link:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW3_Q1.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.decomposition import FastICA
```

```
"""
Load UCI ML Iris data
Return: data(shape = (150, 4)) and labels(shape = (150, 1)) in numpy array(rank 2)
"""
def load_data(classes):
    data = pd.read_csv('Iris.csv', index_col=0).as_matrix()
    features = data[:, :-1]
    labels = data[:, -1].reshape(-1, 1)
    for class_id in classes:
        labels[labels == class_id[0]] = class_id[1]
    return features.astype(np.float32), labels.astype(np.float32)
```

```
"""
Perform Data Shuffling
Return: shuffled features and labels
"""
def data_shuffling(features, labels):
    shuffle_array = np.random.permutation(features.shape[0])
    features = features[shuffle_array]
    labels = labels[shuffle_array]
    return features, labels
```

```
"""
Splitting the data into train and test set
Variable: ratio = percentage of data for train set
Return: train set and test set
"""
def data_split(features, labels, ratio=0.7):
    train_end_index = np.ceil(features.shape[0] * 0.7).astype(np.int32)
    train_features = features[:train_end_index]
    train_labels = labels[:train_end_index]
    test_features = features[train_end_index:]
    test_labels = labels[train_end_index:]
    return train_features, train_labels, test_features, test_labels
```

```
"""
Perform standardization for features
Return: standardized features, standard deviation, mean
"""
def standardization(features, exist_params=False, std=None, mean=None, colvar=True):
```

```
    features = features.T if not colvar else features
    if not exist_params:
        std = np.std(features, axis=0, keepdims=True)
        mean = np.mean(features, axis=0, keepdims=True)
    standard_features = (features - mean) / std
    return standard_features, std, mean
```

```
"""
Preparation of data
Return: train_features, train_labels, test_features, test_labels
"""
def data_preparation(features, labels):
    features, labels = data_shuffling(features, labels)
    train_features = np.empty((0, 4))
    train_labels = np.empty((0, 1))
    test_features = np.empty((0, 4))
    test_labels = np.empty((0, 1))
    for each_class in classes:
        features_class = features[np.squeeze(labels == float(each_class[1]))]
        labels_class = labels[np.squeeze(labels == float(each_class[1]))]
        a, b, c, d = data_split(features_class, labels_class)
        train_features = np.append(train_features, a, axis=0)
        train_labels = np.append(train_labels, b, axis=0)
        test_features = np.append(test_features, c, axis=0)
        test_labels = np.append(test_labels, d, axis=0)
    train_features, train_labels = data_shuffling(train_features, train_labels)
    test_features, test_labels = data_shuffling(test_features, test_labels)
    train_features, std, mean = standardization(train_features)
    test_features, _, _ = standardization(test_features, exist_params=True, std=std,
mean=mean)
    return train_features, train_labels, test_features, test_labels
```

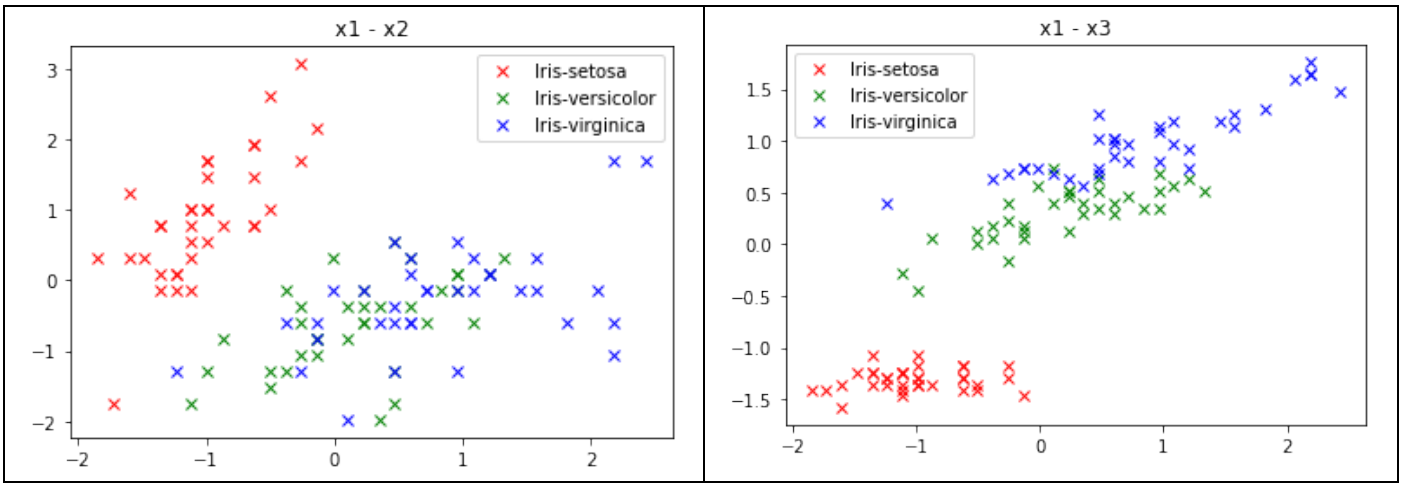
```
"""
Perform Independent Component Analysis
"""
def ica(features, colvar=True):
    features = features.T if not colvar else features
    ICA = FastICA(n_components=2)
    features_reduced = ICA.fit_transform(features)
    return features_reduced
```

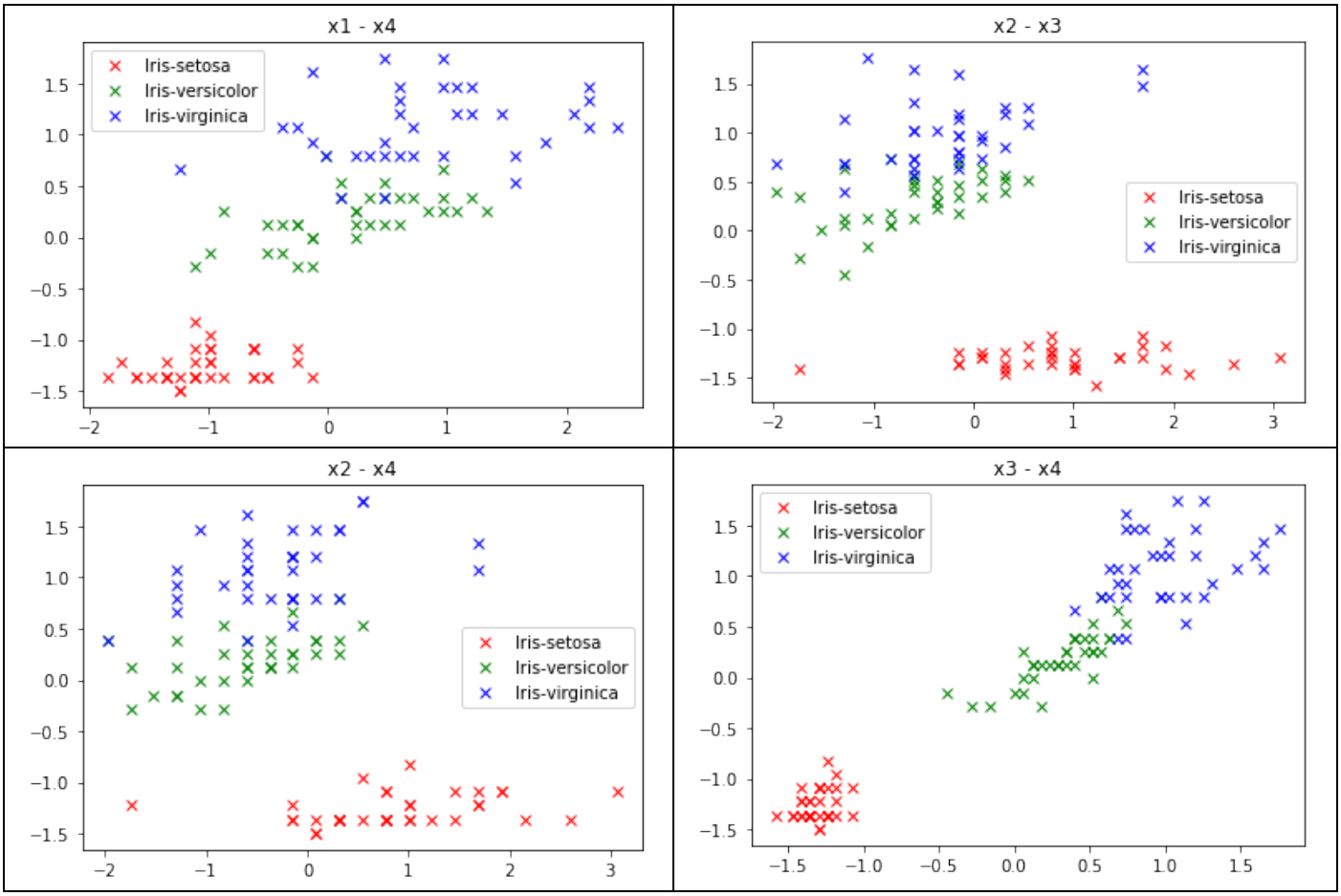
```
"""
k-NN inferening
Variable: k = k-nearest neightbour
Return: predictions to inference set
"""
def kNN_inference(train_features, train_labels, inf_features, k=3):

    #Properties#
    train_data_size = train_features.shape[0]
    train_features_size = train_features.shape[1]
    inf_data_size = inf_features.shape[0]

    #Define saver of predictions #
    predictions = np.empty((inf_features.shape[0], 1))

    #Compute the predictions for all in inference set#
    for i in range(inf_data_size):
        current_inf = np.tile(inf_features[i].reshape(1, -1), (train_data_size, 1))
        euclidean = np.linalg.norm(np.subtract(train_features, current_inf), axis=1)
        sort_index = np.argsort(euclidean)
        euclidean = euclidean[sort_index]
        k_labels = train_labels.reshape(-1)[sort_index][:k]
        pred_class, counts = np.unique(k_labels, return_counts=True)
        predict = pred_class[np.argmax(counts)]
        predictions[i] = predict
```

```
    return predictions
"""
Plot the scatter graphs
"""
def plot_scatter(features, labels):
    num_features = features.shape[1]
    for i in range(num_features - 1):
        for j in range(i + 1, num_features):
            plt.plot(features[np.squeeze(labels == float(0))][:, i],
features[np.squeeze(labels == float(0))][:, j], 'rx', label='Iris-setosa')
            plt.plot(features[np.squeeze(labels == float(1))][:, i],
features[np.squeeze(labels == float(1))][:, j], 'gx', label='Iris-versicolor')
            plt.plot(features[np.squeeze(labels == float(2))][:, i],
features[np.squeeze(labels == float(2))][:, j], 'bx', label='Iris-virginica')
            plt.title('x{} - x{}'.format(i + 1, j + 1))
            plt.legend(loc='best')
            plt.show()
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
cross_val_times = 10
```

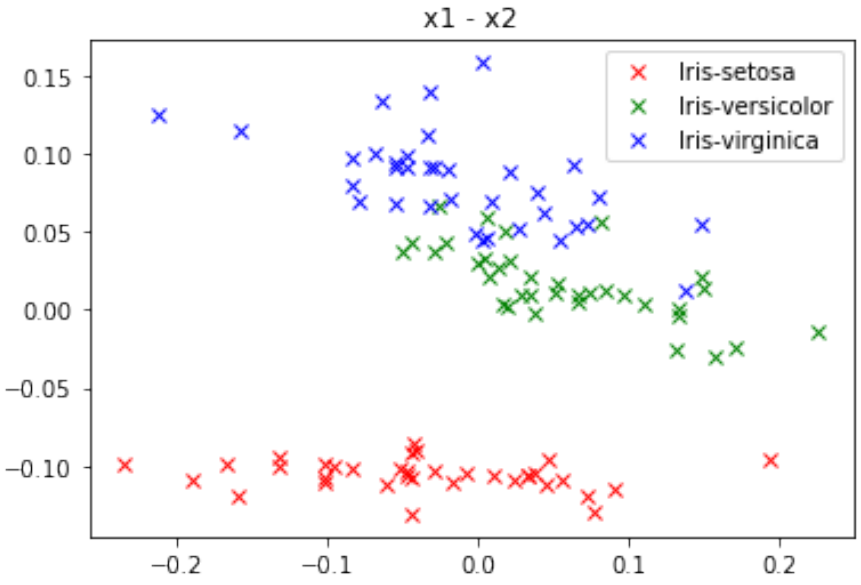
```
features, labels = load_data(classes)
accuracy_history = list()
for i in range(cross_val_times):
    train_features, train_labels, test_features, test_labels = data_preparation(features,
labels)
    train_num = train_features.shape[0]
    test_num = test_features.shape[0]
    features_reduced = ica(np.append(train_features, test_features, axis=0))
    train_features_reduced = features_reduced[:train_num]
    test_features_reduced = features_reduced[train_num:train_num + test_num]
    predicts = kNN_inference(train_features, train_labels, test_features)
    accuracy_history.append(np.mean((predicts == test_labels).astype(np.float32)))
print('Scatter Plots of Original Data: ')
plot_scatter(train_features, train_labels)
print('Scatter Plots of ICA-transformed Data: ')
plot_scatter(train_features_reduced, train_labels)
accuracy_history = np.array(accuracy_history)
print('Average Accuracy ({} times): {}'.format(cross_val_times, np.mean(accuracy_history)))
print('Accuracy Variance: {}'.format(np.var(accuracy_history)))
```

The following are the plots without using any means of dimensionality reduction,





The following plot is executed with ICA,



The average accuracy of 10 times is 0.9555555582046509.

The accuracy variance of 10 times is 0.0004938271595165133.

Question 2:

According to the question requirement, the initialization of clusters' centres is the first data sample of each class without shuffling. Let the clusters be "0, 1, 2" since $k = 3$.

The initial coordinates of clusters are,

Initial coordinate of Cluster 0: [5.1 3.5 1.4 0.2]

Initial coordinate of Cluster 1: [7. 3.2 4.7 1.4]

Initial coordinate of Cluster 2: [6.3 3.3 6. 2.5]

The question was solved by using Python3. The code is uploaded to GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW3_Q2.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
"""
Load UCI ML Iris data
Return: data(shape = (150, 4)) and labels(shape = (150, 1)) in numpy array(rank 2)
"""
def load_data(classes):
    data = pd.read_csv('Iris.csv', index_col=0).as_matrix()
    features = data[:, :-1]
    labels = data[:, -1].reshape(-1, 1)
    for class_id in classes:
        labels[labels == class_id[0]] = class_id[1]
    return features.astype(np.float32), labels.astype(np.float32)
```

```
"""
Initialize the clusters' coordinate with the first sample from each class
Return: initialized clusters' coordinate
"""
def cluster_init(features, labels, k):
    clusters = np.empty((k, features.shape[1]), dtype=np.float32)
    for i in range(k):
        clusters[i] = features[np.squeeze(labels == float(i))][0]
    return clusters
```

```
"""
Perform k-means
"""
def k_means(features, clusters, k, converge_diff):
    diff_history = list()
    num_data = features.shape[0]
    dim = features.shape[1]
    diff = converge_diff + 10e-6
    clus_feat_diff = np.zeros((k, num_data), dtype=np.float32)
    epoch_counter = 0
    while diff > converge_diff:
        pre_clusters = np.copy(clusters)
        for i in range(k):
            clus_feat_diff[i] = np.linalg.norm(np.tile(clusters[i].reshape(1, dim),
(num_data, 1)) - features, axis=1)
        nearest_cluster = np.argmin(clus_feat_diff, axis=0)
        for i in range(k):
            clusters[i] = np.average(features[nearest_cluster == i], axis=0)
        diff = np.average(np.linalg.norm(clusters - pre_clusters, axis=1))
        diff_history.append(diff)
```

```
        epoch_counter += 1
        print('Epoch {}, Average Distance Difference(with previous clusters):
{}'.format(epoch_counter, diff))
    return clusters, diff_history, nearest_cluster
"""
Plot the history of average distance difference
"""
def plot_avg_diff(diff_history):
    plt.plot(list(range(1, len(diff_history) + 1)), diff_history, label='avg distance
difference')
    plt.title('Average Distance Difference of Clusters - Epoch')
    plt.legend(loc='best')
    plt.show()
"""
Plot the scatter graphs with clusters
"""
def plot_scatter(features, labels, clusters, nearest_cluster):
    num_features = features.shape[1]
    for i in range(num_features - 1):
        for j in range(i + 1, num_features):
            plt.plot(clusters[0][i], clusters[0][j], 'ro', label='Cluster 0(C0)')
            plt.plot(clusters[1][i], clusters[1][j], 'go', label='Cluster 1(C1)')
            plt.plot(clusters[2][i], clusters[2][j], 'bo', label='Cluster 2(C2)')
            plt.plot(features[nearest_cluster == 0][:, i], features[nearest_cluster
== 0][:, j], 'r+', label='Samples near C0')
            plt.plot(features[nearest_cluster == 1][:, i], features[nearest_cluster
== 1][:, j], 'g+', label='Samples near C1')
            plt.plot(features[nearest_cluster == 2][:, i], features[nearest_cluster
== 2][:, j], 'b+', label='Samples near C2')
            plt.title('x{} - x{} with clusters'.format(i + 1, j + 1))
            plt.legend(loc='best')
            plt.show()
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
converge_diff = 0.01
k = 3
features, labels = load_data(classes)
clusters = cluster_init(features, labels, k)
for i in range(k):
    print('Initial coordinate of Cluster {}: {}'.format(i, clusters[i]))
print()
clusters, diff_history, nearest_cluster = k_means(features, clusters, k,
converge_diff)
print()
for i in range(k):
    print('Final coordinate of Cluster {}: {}'.format(i, clusters[i]))
print()
plot_avg_diff(diff_history)
plot_scatter(features, labels, clusters, nearest_cluster)
for i in range(k):
    print('Data points in Cluster {}: {}'.format(i, np.sum((nearest_cluster ==
i).astype(int))))
print()
labeled_clusters = np.zeros_like(clusters)
for i in range(k):
    possible_label, majority_vote = np.unique(labels[nearest_cluster == i],
return_counts=True)
    cluster_label = int(possible_label[np.argmax(majority_vote)])
    labeled_clusters[cluster_label] = clusters[i]
    print('Votes for Cluster {}:'.format(i))
    for j in range(possible_label.shape[0]):
        print('Label Class {}: {}'.format(int(possible_label[j]), majority_vote[j]))
```

```
    print('Cluster {} belongs to the Label Class {} with majority vote
{}/{}'.format(i, cluster_label, np.amax(majority_vote), np.sum(majority_vote)))
    print()
for i in range(k):
    print('Final coordinate of Class Labeled Cluster {}: {}'.format(i,
labeled_clusters[i]))
print()
print('{} of data samples are placed in wrong clusters'.format(np.sum(1 -
(nearest_cluster == np.squeeze(labels)).astype(int))))
```

The training process of k-means,

Epoch 1, Average Distance Difference(with previous clusters): 0.6643375754356384

Epoch 2, Average Distance Difference(with previous clusters): 0.1411137729883194

Epoch 3, Average Distance Difference(with previous clusters): 0.020745761692523956

Epoch 4, Average Distance Difference(with previous clusters): 0.0

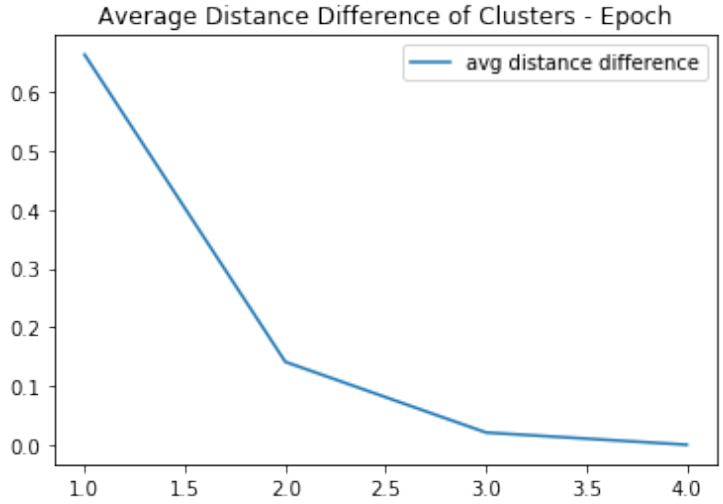
The final coordinates are,

Final coordinate of Cluster 0: [5.0059996 3.4180002 1.464 0.24399997]

Final coordinate of Cluster 1: [5.901613 2.7483873 4.393549 1.4338712]

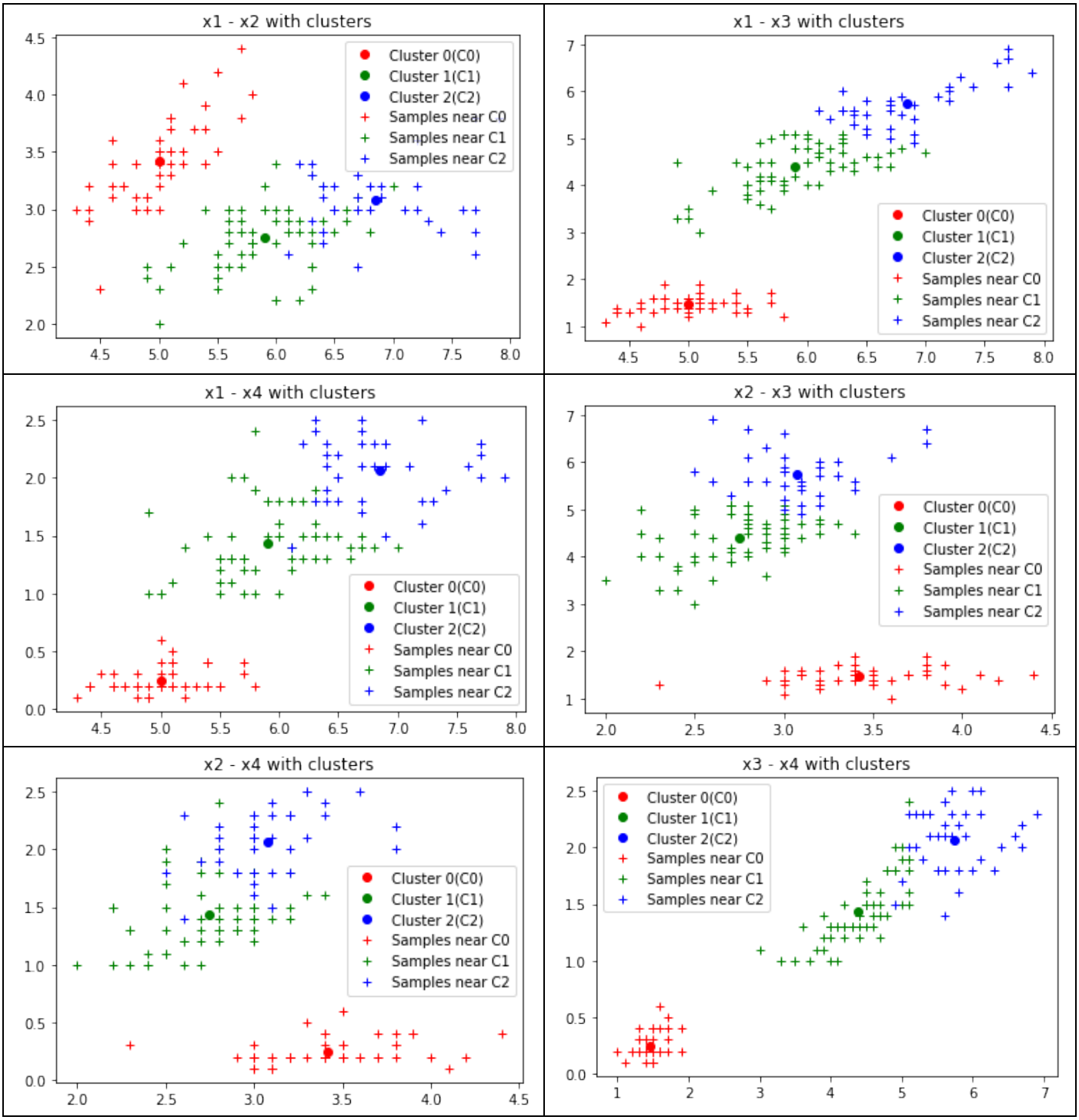
Final coordinate of Cluster 2: [6.849999 3.0736842 5.742105 2.0710528]

The average distance differences of clusters – epoch are plotted in the following graph,



We can see that the model converges to 0 average distance difference with the previous clusters' coordinates. The clusters', then, stop moving.

The scatter graphs for the clusters and the samples coloured according to the nearest cluster are plotted as following,



The total number of data points in Cluster 0 is 50, in Cluster 1 is 62 and 38 in Cluster 2.

The votes are recorded as follow,

Votes for Cluster 0: Label Class 0: 50 Cluster 0 belongs to the Label Class 0 with majority vote 50/50
Votes for Cluster 1: Label Class 1: 48 Label Class 2: 14 Cluster 1 belongs to the Label Class 1 with majority vote 48/62
Votes for Cluster 2: Label Class 1: 2 Label Class 2: 36 Cluster 2 belongs to the Label Class 2 with majority vote 36/38

According to the majority vote, the class-labeled clusters are computed as follow,

Final coordinate of Class Labeled Cluster 0: [5.0059996 3.4180002 1.464 0.24399997]

Final coordinate of Class Labeled Cluster 1: [5.901613 2.7483873 4.393549 1.4338712]

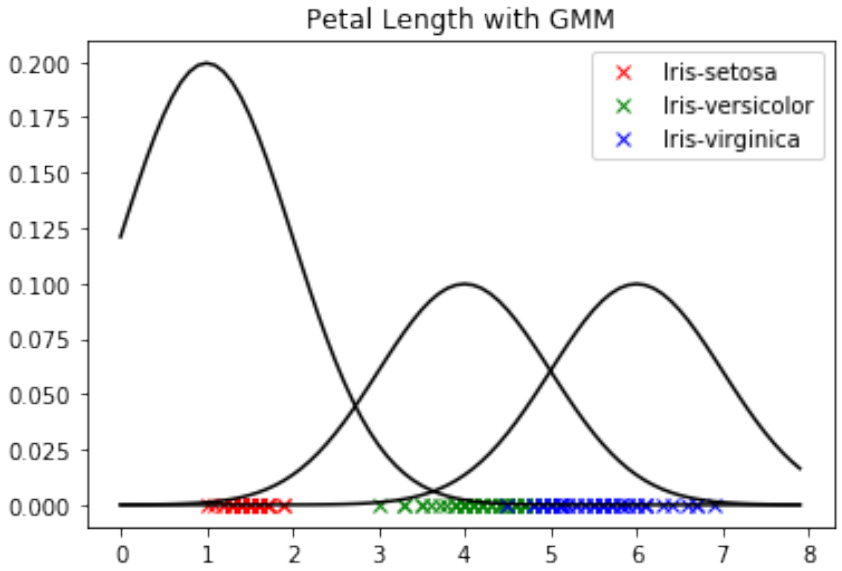
Final coordinate of Class Labeled Cluster 2: [6.849999 3.0736842 5.742105 2.0710528]

which are coincidentally the same as the original trained cluster.

By computing the misplaced data samples, it was found that 16 of data samples are placed in wrong clusters.

Question 3:

According to the requirements given in this question, the weights, means and variances are initialized with the given values. By plotting the graphs for the initial state,



The classes of the data are actually of no use in this question since we are doing unsupervised clustering. However, we can still see that the Iris-setosa are distinct and far from the other two group while the Iris-

versicolor and Iris-virginica are just near each other. If we just implement the Gaussian Mixture Model and Expectation-Maximization algorithm directly, we will be suffering from the "divided by zero" problem since one of the cluster will be updated with an uncertain mean, a gigantic variance and tiny weight.

As a result, when it comes to the Maximization step (or the Expectation step if someone would like to update the weights in Expectation step), the numerator of the update of the weight,

$$numerator\ of\ the\ update\ term\ of\ weight = \sum_{i=1}^{n=150} \frac{\alpha_j \mathcal{G}_j(x|\mu_j, \sigma_j^2)}{\sum_{k=1}^{c=3} \alpha_k \mathcal{G}_k(x|\mu_k, \sigma_k^2)}$$

will be extremely close to zero. Thus, when it comes to the update of mean and variance of the specific cluster, it suffers from the "divided by zero" problem and NaN is occurred.

In my solution, I add a $\epsilon = 1 \times 10^{-6}$ to the term above so that we can successfully pass the numerical issue.

The code is uploaded to GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW3_Q3.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
"""
Load UCI ML Iris data
Return: data(shape = (150, 4)) and labels(shape = (150, 1)) in numpy array(rank 2)
"""
def load_data(classes):
    data = pd.read_csv('Iris.csv', index_col=0).as_matrix()
    features = data[:, :-1]
    labels = data[:, -1].reshape(-1, 1)
    for class_id in classes:
        labels[labels == class_id[0]] = class_id[1]
    return features.astype(np.float32), labels.astype(np.float32)
```

```
"""
Initialize the univariate GMM parameters
"""
def params_init():
    means = np.array([1., 4., 6.], dtype=np.float32)
    variances = np.array([1., 1., 1.], dtype=np.float32)
    alphas = np.array([0.5, 0.25, 0.25], dtype=np.float32)
    return means, variances, alphas
```

```
"""
Get the probability density from univariate gaussian distribution
"""
def get_gauss(x, mean, sigma):
    gauss_x = (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(-(np.square(x - mean) / (2 *
sigma)))
    return gauss_x.astype(np.float32)
```

```
"""
Univariate Gaussian Mixture Model
"""
def uni_gmm(features, means, variances, alphas, epoch, k, epsilon=10e-6):
    log_likelihood_history = list()
```

```
    #Define the function for calculating weighted probability densities for all data
according to clusters#
    def get_probden():
        return np.multiply(alphas, get_gauss(features, means, np.sqrt(variances)))

    #Compute the log-likelihood for the initial state#
    total_prob_den = get_probden()
    log_likelihood_history.append(np.sum(np.log(np.sum(total_prob_den, axis=1)), axis=0))

    #Start EM Algorithm#
    for i in range(epoch):
        #Expectation step#
        p = np.divide(total_prob_den, np.sum(total_prob_den, axis=1, keepdims=True))

        #Maximization step#
        alphas = (np.sum(p, axis=0) + epsilon) / features.shape[0]
        means = np.multiply((1 / (np.sum(p, axis=0) + epsilon)), np.sum(np.multiply(p,
features), axis=0))
        variances = np.multiply((1 / (np.sum(p, axis=0) + epsilon)), np.sum(np.multiply(p,
np.square(features - means))))
        total_prob_den = get_probden()
        log_likelihood_history.append(np.sum(np.log(np.sum(total_prob_den, axis=1)), axis=0))
    return means, variances, alphas, log_likelihood_history
```

```
"""
Return number of members in each cluster if hard clustering
"""
def hard_cluster_num_members(features, alphas, means, variances):
    p = np.multiply(alphas, get_gauss(features, means, np.sqrt(variances)))
    belong = np.argmax(p, axis=1)
    clusters_name, num_members = np.unique(belong, return_counts=True)
    return clusters_name, num_members
```

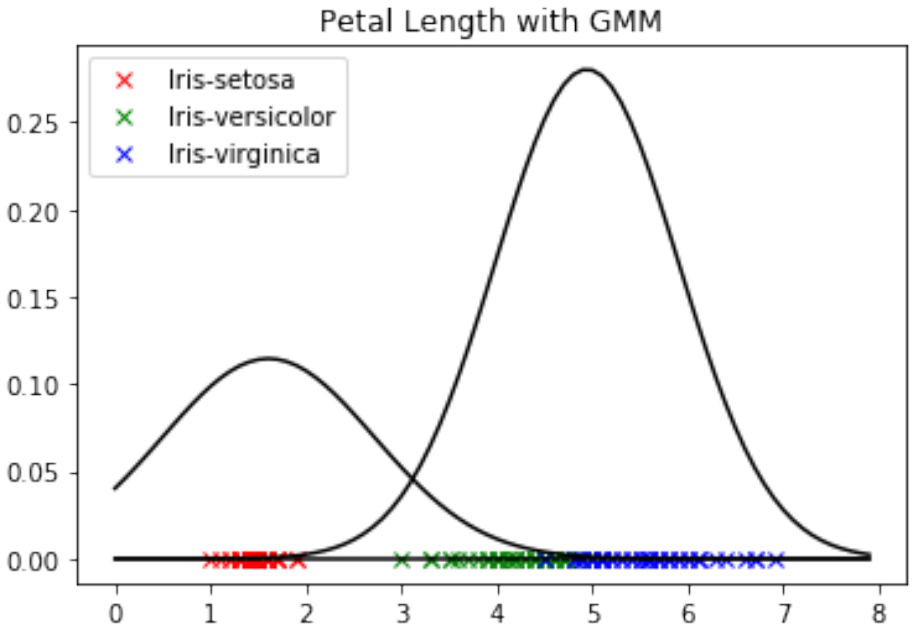
```
"""
Plot the scatter graphs with univariate Gaussian Mixture Model
"""
def plot_scatter(features, labels, plot_GMM=False, GMM_mean=None, GMM_var=None,
GMM_alpha=None):
    plt.plot(features[labels == float(0)], np.zeros_like(features[labels == float(0)]), 'rx',
label='Iris-setosa')
    plt.plot(features[labels == float(1)], np.zeros_like(features[labels == float(1)]), 'gx',
label='Iris-versicolor')
    plt.plot(features[labels == float(2)], np.zeros_like(features[labels == float(2)]), 'bx',
label='Iris-virginica')
    if plot_GMM:
        x = np.arange(np.amin(features) - 1., np.amax(features) + 1., 0.1)
        plt.plot(x, alphas[0] * get_gauss(x, means[0], np.sqrt(variances[0])), 'k')
        plt.plot(x, alphas[1] * get_gauss(x, means[1], np.sqrt(variances[1])), 'k')
        plt.plot(x, alphas[2] * get_gauss(x, means[2], np.sqrt(variances[2])), 'k')
    plt.title('Petal Length {}'.format(('with GMM' if plot_GMM else '')))
    plt.legend(loc='best')
    plt.show()
```

```
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
epoch = 3000
k = 3
```

```
features, labels = load_data(classes)
features = features[:, 2].reshape(-1, 1)
means, variances, alphas = params_init()
print('The initial state:')
plot_scatter(features, labels, plot_GMM=True, GMM_mean=means, GMM_var=variances,
GMM_alpha=alphas)
```

```
means, variances, alphas, log_likelihood_history = uni_gmm(features, means, variances,
alphas, epoch, k)
print('State after performing GMM:')
plot_scatter(features, labels, plot_GMM=True, GMM_mean=means, GMM_var=variances,
GMM_alpha=alphas)
print('After {} epochs, the parameters are as follow:'.format(epoch))
print('Means: {}'.format(means))
print('Variances: {}'.format(variances))
print('Alphas: {}'.format(alphas))
print()
clusters_name, num_members = hard_cluster_num_members(features, alphas, means, variances)
no_member_cluster = [x for x in list(np.arange(k)) if x not in list(clusters_name)]
print('If we use hard clustering, the number of members in each cluster:')
for i in range(clusters_name.shape[0]):
    print('Cluster {}: {}'.format(clusters_name[i], num_members[i]))
print('There is no member in Cluster {}'.format(no_member_cluster))
```

The state after performing Gaussian Mixture Model,



As mentioned, one of the cluster is of some mean, gigantic variance and extremely small weight. The black “straight line” at the bottom is that cluster. The reason is that the distribution of data in this single dimension is showing significantly two group of data since the distribution of Iris-versicolor (green) and Iris-virginica (blue) are very similar to each other. They are considered to be the same group of distribution when being handled by unsupervised clustering.

After 3000 epochs, the parameters are,

	Cluster 0	Cluster 1	Cluster 2
Mean	1.6074963e+00	4.9435172e+00	3.2133581e-03
Variance	1.5305580e+00	8.4302324e-01	8.1465505e+06
Alpha	3.551694e-01	6.448308e-01	6.672853e-08

If we want to convert the soft clustering results to hard clustering, we can directly compute the three values (since we are using three clusters) of product of alphas and probability density of Gaussian using the trained

means and variances. After that, we can pick the cluster that gives us the largest value among the three values. Then we can say that that data point belongs to the mentioned cluster.

The results of number of members in each cluster are as follow,

Cluster 0	51
Cluster 1	99
Cluster 2	0

Question 4:

In my program, the data are encoded in integers as follow,

Attribute	Values	Encoded
outlook	sunny	1
Outlook	Overcast	2
Outlook	Rain	3
Temperature	≥ 80	3
Temperature	$\geq 70\ and \leq 79$	2
Temperature	< 70	1
Humidity	≥ 76	2
Humidity	< 76	1
Windy	True	1
Windy	False	0

and labels are encoded as follow,

play	1
no play	0

The program was upload to my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW3_Q4.ipynb

```
import numpy as np
"""
Create data for decision tree
"""
def create_data(discrete=False):
    feature_name = np.array(['outlook', 'temperature', 'humidity', 'windy'])
    data = np.array([
        [1., 85., 85., 0.],
        [1., 80., 90., 1.],
        [2., 83., 78., 0.],
        [3., 70., 96., 0.],
        [3., 68., 80., 0.],
        [3., 65., 70., 1.],
        [2., 64., 65., 1.],
        [1., 72., 95., 0.],
        [1., 69., 70., 0.],
        [3., 75., 80., 0.],
        [1., 75., 70., 1.],
        [2., 72., 90., 1.],
        [2., 81., 75., 0.],
```

```
        [3., 71., 80., 1.]
    ])
    if discrete:
        data[:, 1][data[:, 1] < 70] = 1.
        data[:, 1][np.logical_and(70. <= data[:, 1],data[:, 1] <= 79.)] = 2.
        data[:, 1][data[:, 1] >= 80.] = 3.
        data[:, 2][data[:, 2] < 76] = 1.
        data[:, 2][data[:, 2] >= 76] = 2.
    label = np.array([
        [0.],
        [0.],
        [1.],
        [1.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [1.],
        [1.],
        [0.]
    ])
    return data.astype(np.float32), label.astype(np.float32), feature_name
"""
Compute the entropy
"""
def get_entropy(label):
    distinct_label, count = np.unique(np.squeeze(label), return_counts=True)
    p = np.divide(count, np.sum(count))
    entropy = -np.sum(np.multiply(p, np.log2(p)))
    return entropy
"""
Select the best feature for splitting
Return: the index of the best features
"""
def get_best_feature(data, label):
    num_data = data.shape[0]
    num_feature = data.shape[1]
    base_entropy = get_entropy(label)
    information_gains_ratio = np.zeros((num_feature))
    for i in range(num_feature):
        unique_feature, unique_count = np.unique(data[:, i], return_counts=True)
        feature_entropy = 0.
        for j in range(unique_feature.shape[0]):
            temp_label = label[data[:, i] == unique_feature[j]]
            p_feature = temp_label.shape[0] / num_data
            feature_entropy += p_feature * get_entropy(temp_label)
        information_gains_ratio[i] = (base_entropy - feature_entropy) / get_entropy(data[:,
i])
    return np.argmax(information_gains_ratio)
"""
Create decision tree
"""
def create_tree(tree, data, label, feature_name, pre_feature='Root', pre_value='Root'):
    #Check if the data subset of the current node is pure#
    if np.unique(label).shape[0] == 1:
        return np.squeeze(label)[0]

    #Get the best feature for splitting#
    index_best_feature = get_best_feature(data, label)
    values = np.unique(data[:, index_best_feature])
```

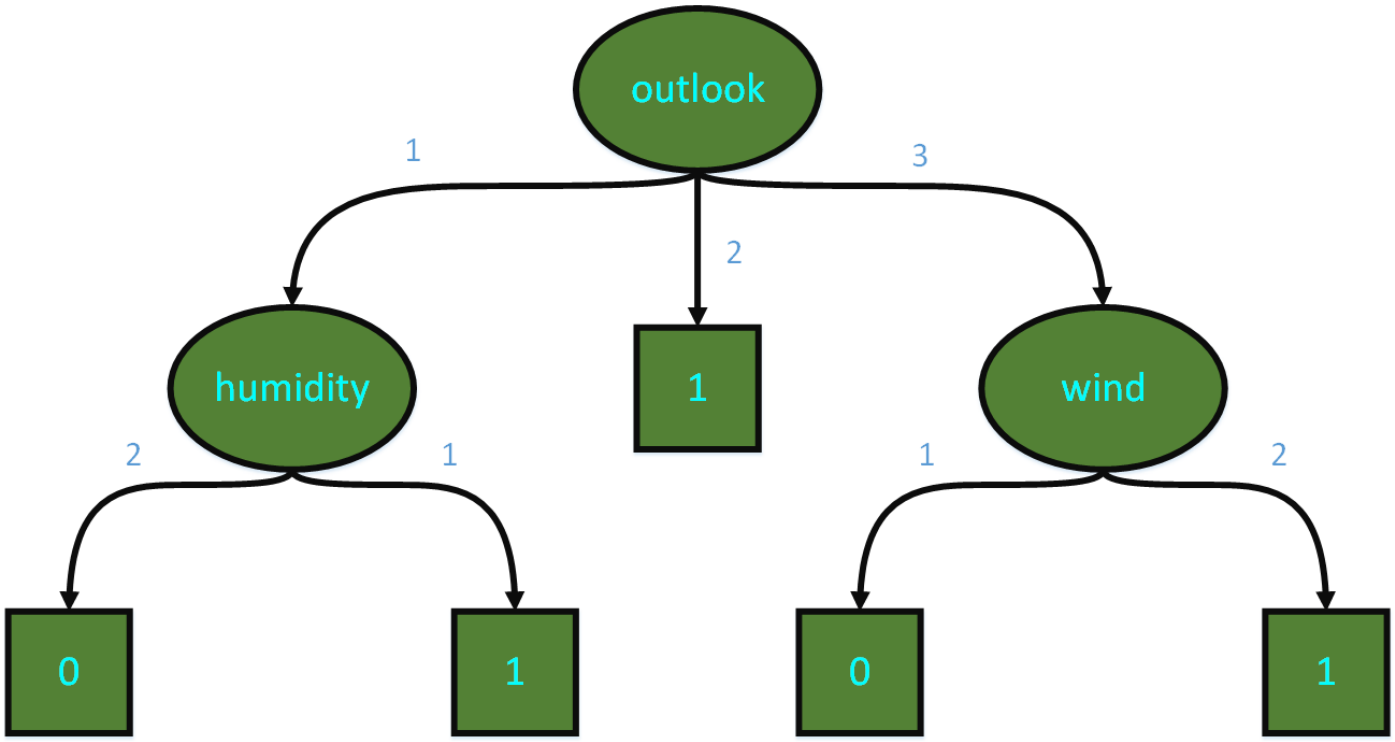
```
    #Create the tree layer in depth first#
    for value in values:
        split_feature_name = feature_name[index_best_feature]
        temp = data[data[:, index_best_feature] == value]
        new_data = np.append(temp[:, :index_best_feature], temp[:, index_best_feature + 1:],
axis=1)
        new_label = label[data[:, index_best_feature] == value]
        new_feature_name = np.append(feature_name[0:index_best_feature],
feature_name[index_best_feature + 1:])
        subtree = create_tree(list(), new_data, new_label, new_feature_name,
pre_feature=split_feature_name, pre_value=str(value))
        tree.append(((split_feature_name, value), subtree))
    return tree
```

```
data, label, feature_name = create_data(discrete=True)
tree = create_tree(list(), data, label, feature_name)
tree
```

The decision tree computed are formatted as

```
[(('outlook', 1.0), [(('humidity', 1.0), 1.0), (('humidity', 2.0), 0.0)]),
 (('outlook', 2.0), 1.0),
 (('outlook', 3.0), [(('windy', 0.0), 1.0), (('windy', 1.0), 0.0)])]
```

which denotes



Question 5:

In my program, the data are encoded in integers as follow,

Attribute	Values	Encoded
outlook	sunny	1
Outlook	Overcast	2
Outlook	Rain	3

Windy	True	1
Windy	False	0

and labels are encoded as follow,

play	1
no play	0

In order to accept continuous values, we should use the way stated in PPT. The algorithm be like:

First, we should sort values of the selected feature, then we have to find the distinct values of this list of values. If we want to divide the continuous values into 3 sections, we can try the first threshold from the smallest value to the largest value in the list. Figure out the one with the largest information gain. Secondly, we can try the second threshold from the value slightly larger than the first threshold and repeat the action done previously. So that we can have three sections.

The program was upload to my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW3_Q5.ipynb

```
import numpy as np
"""
Create data for decision tree
"""
def create_data(discrete=False):
    feature_name = np.array(['outlook', 'temperature', 'humidity', 'windy'])
    data = np.array([
        [1., 85., 85., 0.],
        [1., 80., 90., 1.],
        [2., 83., 78., 0.],
        [3., 70., 96., 0.],
        [3., 68., 80., 0.],
        [3., 65., 70., 1.],
        [2., 64., 65., 1.],
        [1., 72., 95., 0.],
        [1., 69., 70., 0.],
        [3., 75., 80., 0.],
        [1., 75., 70., 1.],
        [2., 72., 90., 1.],
        [2., 81., 75., 0.],
        [3., 71., 80., 1.]
    ])
    if discrete:
        data[:, 1][data[:, 1] < 70] = 1.
        data[:, 1][np.logical_and(70. <= data[:, 1],data[:, 1] <= 79.)] = 2.
        data[:, 1][data[:, 1] >= 80.] = 3.
        data[:, 2][data[:, 2] < 76] = 1.
        data[:, 2][data[:, 2] >= 76] = 2.
    label = np.array([
        [0.],
        [0.],
        [1.],
        [1.],
        [1.],
```

```
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [1.],
        [1.],
        [0.]
    ])
    return data.astype(np.float32), label.astype(np.float32), feature_name
```

```
"""
Compute the entropy
"""
def get_entropy(label):
    distinct_label, count = np.unique(np.squeeze(label), return_counts=True)
    p = np.divide(count, np.sum(count))
    entropy = -np.sum(np.multiply(p, np.log2(p)))
    return entropy
```

```
"""
Select the best feature for splitting
Return: the index of the best features
"""
def get_best_feature(data, label, feature_name):
    num_data = data.shape[0]
    num_feature = data.shape[1]
    base_entropy = get_entropy(label)
    information_gains_ratio = np.zeros((num_feature))
    for i in range(num_feature):
        unique_feature, unique_count = np.unique(data[:, i], return_counts=True)
        feature_entropy = 0.
        if feature_name[i] != 'temperature' and feature_name[i] != 'humidity':
            for j in range(unique_feature.shape[0]):
                temp_label = label[data[:, i] == unique_feature[j]]
                p_feature = temp_label.shape[0] / num_data
                feature_entropy += p_feature * get_entropy(temp_label)
            information_gain = base_entropy - feature_entropy
        else:
            unique_feature = np.sort(unique_feature)
            continuous_gains = np.zeros_like(unique_feature)
            for index in range(unique_feature.shape[0]):
                temp_label = label[data[:, i] <= unique_feature[index]]
                p_feature = temp_label.shape[0] / num_data
                feature_entropy += p_feature * get_entropy(temp_label)
                temp_label = label[data[:, i] > unique_feature[index]]
                p_feature = temp_label.shape[0] / num_data
                feature_entropy += p_feature * get_entropy(temp_label)
                continuous_gains[index] = base_entropy - feature_entropy
            threshold = np.argmax(continuous_gains)
            information_gain = np.amax(continuous_gains)
        information_gains_ratio[i] = information_gain / get_entropy(data[:, i])
    return np.argmax(information_gains_ratio)
```

```
"""
Create decision tree
"""
def create_tree(tree, data, label, feature_name, pre_feature='Root', pre_value='Root'):
    #Check if the data subset of the current node is pure#
    if np.unique(label).shape[0] == 1:
        return label.reshape(-1)[0]

    #Get the best feature for splitting#
    index_best_feature = get_best_feature(data, label, feature_name)
    values = np.unique(data[:, index_best_feature])
```

```
    #Create the tree layer in depth first#
    for value in values:
        split_feature_name = feature_name[index_best_feature]
        temp = data[data[:, index_best_feature] == value]
        new_data = np.append(temp[:, :index_best_feature], temp[:, index_best_feature + 1:],
axis=1)
        new_label = label[data[:, index_best_feature] == value]
        new_feature_name = np.append(feature_name[0:index_best_feature],
feature_name[index_best_feature + 1:])
        subtree = create_tree(list(), new_data, new_label, new_feature_name,
pre_feature=split_feature_name, pre_value=str(value))
        tree.append(((split_feature_name, value), subtree))
    return tree
```

```
data, label, feature_name = create_data()
tree = create_tree(list(), data, label, feature_name)
tree
```

The decision tree computed are formatted as

```
[(('outlook', 1.0),
  [(('humidity', 70.0), 1.0),
   (('humidity', 95.0), 0.0)]),
 (('outlook', 2.0), 1.0),
 (('outlook', 3.0), [(('windy', 0.0), 1.0), (('windy', 1.0), 0.0)])]
```

the above "(('humidity', 95.0), 0.0)])" means > 75 is 0. The humidity ≤ 70 represents "low" and humidity > 70 represnets "high".

The tree is like,