Machine Learning
Homework #2

Daniel Ho Kwan Leung
104360098
CSIE, Year 3

Question 1(i):

To cope with the overfitting problem, there are many ways to solve. We can use cross-validation, drop-out and regularization etc. The overfitting problem refer to the situation that a complex model can somehow overfit the training data. That is, the model is so complex that it's parameters can perform a complicated hypothesis which fits too well to the training data. For example, if we use a deep neural networks containing millions of parameters (weights and bias) to fit the training data of Iris data from UC Irvine Machine Learning Repository, since the data is quite simple in terms of the number of features and number of data, this complicated model ought to fit the training data very well.

If we want to keep the complex model but remedy the overfitting problem, we can use regularization. There are many kinds of regularization, one of the regularization method that the question 1 stated is called L2-regularization.

Assume that each weight w_i belongs to the set W which contains all the weights of the machine learning model. λ is a constant hyperparameter.

$$L2\ \ Regularization = \lambda \sum_{i=1}^{|W|} w_i^2$$

$$L1\ \ Regularization = \lambda \sum_{i=1}^{|W|} |w_i|$$

Let's define the dataset be a set X, each tuple of data be $x^{[j]}$, the ground-truth label be $y^{[j]}$, the hypothesis be $\hat{y}^{[j]} = h\left(x^{[j]}|w_1, w_2, \dots, w_{|W|}\right)$, the loss be Mean Square Error and the cost be the mean of losses of the whole data among the dataset. That is,

$$\because \hat{y}^{[j]} = h\left(x^{[j]}|w_1, w_2, \dots, w_{|W|}\right)$$

$$\mathcal{L}_j\left(w_1, w_2, \dots, w_{|W|}|\hat{y}^{[j]}, y^{[j]}\right) = \left(y^{[j]} - \hat{y}^{[j]}\right)^2$$

$$Total\ \ Loss = \sum_{j=1}^{|X|} \mathcal{L}_j$$

$$\mathcal{C}(\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_{|X|}) = \frac{1}{|X|}\sum_{j=1}^{|X|}\mathcal{L}_j$$

The cost function is what we would like to reduce so that we can have a model of better performance. When we add the regularization term to the total loss, such as L2 regularization, they become

$$Total\ Loss_{L2\ Regularized} = \sum_{j=1}^{|X|}\mathcal{L}_j + \lambda\sum_{i=1}^{|W|}w_i^2$$

$$\begin{aligned}\mathcal{C}(\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_{|X|}, \lambda)_{L2\ Regularized} &= \frac{1}{|X|}\left[\sum_{j=1}^{|X|}\mathcal{L}_j + \lambda\sum_{i=1}^{|W|}w_i^2\right] \\ &= \frac{1}{|X|}\left[\sum_{j=1}^{|X|}\left(y^{[j]} - \hat{y}^{[j]}\right)^2 + \lambda\sum_{i=1}^{|W|}w_i^2\right] \\ &= \frac{1}{|X|}\sum_{j=1}^{|X|}\left(y^{[j]} - \hat{y}^{[j]}\right)^2 + \frac{\lambda}{|X|}\sum_{i=1}^{|W|}w_i^2\end{aligned}$$

The objective is to minimize the cost. That is,

$$\arg\min_{W}\mathcal{C}(\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_{|X|}, \lambda)_{L2\ Regularized}$$

When we are minimizing the cost, we are also minimizing the regularization term $\frac{\lambda}{|X|}\sum_{i=1}^{|W|}w_i^2$ or $\lambda\sum_{i=1}^{|W|}w_i^2$. Intuitively, minimizing the regularization term means that we are reducing the magnitude of each weight. Generally, smaller magnitude of weights will then give us a smooth regression curve which can vastly generalize the training data.

When we are using gradient search method such as gradient descent to minimize the cost, we have to compute the gradient of the cost function. Generally, to update the weights for each iterative training using gradient search, we are performing,

$$W_{t+1} = w_t - \alpha\nabla\mathcal{C}_{W_t}$$

$$w_{k,t+1} = w_{k,t} - \alpha\frac{\partial\mathcal{C}}{\partial w_{k,t}}$$

Considering the term $\frac{\partial\mathcal{C}}{\partial w_{k,t}}$.

$$\frac{\partial\mathcal{C}}{\partial w_{k,t}} = \frac{\partial\frac{1}{|X|}\sum_{j=1}^{|X|}\mathcal{L}_j}{\partial w_{k,t}} + \frac{\partial\mathcal{C}\frac{\lambda}{|X|}\sum_{i=1}^{|W|}w_i^2}{\partial w_{k,t}}$$

$$= \frac{1}{|X|}\sum_{j=1}^{|X|}\frac{\partial \mathcal{L}_j}{\partial w_{k,t}} + \frac{\lambda}{|X|}\sum_{i=1}^{|W|}\frac{\partial w_i^2}{\partial w_{k,t}}$$

Considering the term $\frac{\lambda}{|X|}\sum_{i=1}^{|W|}\frac{\partial w_i^2}{\partial w_{k,t}}$. We can get

$$\frac{\lambda}{|X|}\sum_{i=1}^{|W|}\frac{\partial w_i^2}{\partial w_{k,t}} = \frac{2\lambda}{|X|}w_{k,t}$$

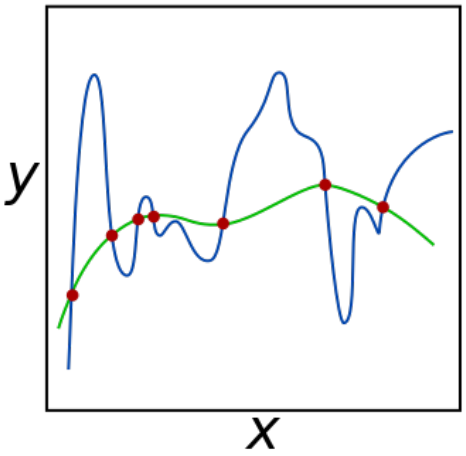
So we can have

$$w_{k,t+1} = w_{k,t} - \alpha\left[\frac{1}{|X|}\sum_{j=1}^{|X|}\frac{\partial \mathcal{L}_j}{\partial w_{k,t}} + \frac{2\lambda}{|X|}w_{k,t}\right]$$

$$= w_{k,t} - \frac{\alpha}{|X|}\sum_{j=1}^{|X|}\frac{\partial \mathcal{L}_j}{\partial w_{k,t}} - \frac{2\alpha\lambda}{|X|}w_{k,t}$$

$$= \left[w_{k,t} - \frac{2\alpha\lambda}{|X|}w_{k,t}\right] - \frac{\alpha}{|X|}\sum_{j=1}^{|X|}\frac{\partial \mathcal{L}_j}{\partial w_{k,t}}$$

The term $\left[w_{k,t} - \frac{2\alpha\lambda}{|X|}w_{k,t}\right]$ actually implies that there is a decay of weights if $0 < \frac{2\alpha\lambda}{|X|} < 1$. That is the reason why regularization is also called "weight decay". Throughout the whole optimizing process, the weights are diminished. Thus, it can reduce the overfitting issue.



Regularization (Source: Wikipedia)

Question 1(ii):

It is actually a L-p Regularization problem. Consider L2-regularization,

if we are going to minimize the cost function,

$$\mathcal{C}(\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_{|X|}, \lambda)_{L2\ Regularized} = \frac{1}{|X|}\left[\sum_{j=1}^{|X|} \mathcal{L}_j + \lambda \sum_{i=1}^{|W|} w_i^2\right]$$

minimizing the latter regularization term is required. When we are minimizing the regularization term, we are actually focusing on the slightly bigger weights. Since a bigger weight will have a bigger square of that weight, the optimization will focus on penalizing those weights of larger value so that it can efficiently minimize the cost function.

If we are now using $\lambda \sum_{i=1}^{|W|} w_i^{10}$ as regularization term, the optimization is penalizing the large weights more in order to efficiently reduce the cost function. Since the weights that between 0 and 1, $0 < \mathrm{w_i} < 1$, will be smaller for the regularization term, the penalty for smaller weights won't be significant but vast for larger weights.

Intuitively, if we want to minimize a function, $g(k_1, k_2) = k_1^{10} + k_2^{10}$ where $0 < |k_1| < 1\ and\ 1 < |k_2|$, reducing the magnitude of k_2 should prior to that of k_1. In short, using the regularization term $\lambda \sum_{i=1}^{|W|} w_i^{10}$ is focusing more on the penalty of larger weights than using $\lambda \sum_{i=1}^{|W|} w_i^2$.

We can, thus, expect that using $\lambda \sum_{i=1}^{|W|} w_i^{10}$ will give us a lower variance but higher bias which produce an underfitting predictions and using $\lambda \sum_{i=1}^{|W|} w_i^2$ will give us a lower bias and higher variance comparing to the L10 regularization.

The related simulation can be found in my GitHub:
https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW2_Q1ii.ipynb

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
data = np.random.rand(150, 1) * 2 - 1
data = np.append(data, np.power(data[:, 0], 2).reshape(-1, 1), axis=1)
data = np.append(data, np.power(data[:, 0], 3).reshape(-1, 1), axis=1)
label = (0.5 * data[:, 0] + 0.75 * data[:, 1] + 3 * data[:, 2] + 4).reshape(-1, 1)
sortarg = np.argsort(data[:, 0])
data = data[sortarg]
label = label[sortarg]
data_input = tf.placeholder(tf.float32, (None, 3), name='data_input')
data_label = tf.placeholder(tf.float32, (None, 1), name='data_label')
A1 = tf.contrib.layers.fully_connected(data_input, 1, activation_fn=None,
weights_initializer=tf.zeros_initializer(), scope='A1')
with tf.variable_scope('A1', reuse=True):
```

```
    W = tf.get_variable('weights')
    B = tf.get_variable('biases')
cost_no_reg = tf.losses.mean_squared_error(label, A1)
cost_reg = tf.losses.mean_squared_error(label, A1) + 0.01 * tf.reduce_sum(tf.square(W))
cost_reg_10 = tf.losses.mean_squared_error(label, A1) + 0.01 * tf.reduce_sum(tf.pow(W, 10))
opt_minimize_no_reg =
tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost_no_reg)
opt_minimize_reg = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost_reg)
opt_minimize_reg_10 =
tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost_reg_10)
sess = tf.Session()
```

```
sess.run(tf.global_variables_initializer())
for i in range(10000):
    sess.run(opt_minimize_no_reg, feed_dict={data_input:data, data_label:label})
WW = sess.run(W)
BB = sess.run(B)
AA = np.squeeze(np.dot(data, WW) + BB)
print(WW, BB)
```

```
sess.run(tf.global_variables_initializer())
for i in range(10000):
    sess.run(opt_minimize_reg, feed_dict={data_input:data, data_label:label})
WW_reg = sess.run(W)
BB_reg = sess.run(B)
AA_reg = np.squeeze(np.dot(data, WW_reg) + BB_reg)
print(WW_reg, BB_reg)
```

```
sess.run(tf.global_variables_initializer())
for i in range(10000):
    sess.run(opt_minimize_reg_10, feed_dict={data_input:data, data_label:label})
WW_reg_10 = sess.run(W)
BB_reg_10 = sess.run(B)
AA_reg_10 = np.squeeze(np.dot(data, WW_reg_10) + BB_reg_10)
print(WW_reg_10, BB_reg_10)
```

```
plt.plot(data[:, 0], label, 'bx')
plt.plot(data[:, 0], AA, 'r-')
plt.plot(data[:, 0], AA_reg, 'g-')
plt.plot(data[:, 0], AA_reg_10, 'y-')
plt.show()
```

The ground-truth function is $0.5x_1 + 0.75x_1^2 + 3x_1^3 + 4$.

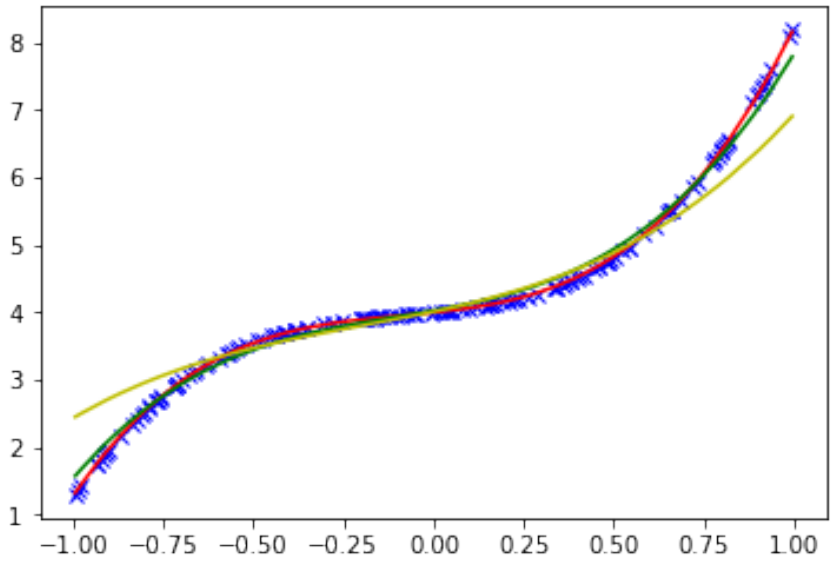
After training, the weights:

\	W1	W2	W3	bias

Without Regularization	0.5536899	0.74895734	2.9137404	4.0003247
L2 Regularization	0.9394344	0.66210777	**2.2088406**	4.0127576
L10 Regularization	1.1365346	0.6738426	**1.1156877**	4.0126557

We can see that the weights without regularized are close to the ground-truth weights. The weights with L2 regularized and with L10 regularized are what we would like to compare. Consider the W3 (in bold), the weight of larger magnitude, we can see that the one with L10 regularized is heavily penalized comparing with the one with L2 regularized.

The following graph shows the data point, the trained curve without regularization (red), the trained curve with L2 regularization (green) and the trained curve with L10 regularization (yellow).



Question 2:

This question is computed by using python. Code can be found in my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW2_Q2.ipynb

```
import numpy as np
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)
```

```
"""
Compute the condition number
Return: condition number
"""
def get_condnum(M):
    eigvals = np.linalg.eigvals(M)
    return np.fabs(np.amax(eigvals) / np.amin(eigvals))
```

```
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
features, labels = load_data(classes)
covs = []
condnums = []
for i in range(3):
    target_features = features[np.squeeze(labels == float(i))]
    covs.append(np.cov(target_features.T))
    condnums.append(get_condnum(covs[i]))
print(condnums)
```

The computed condition number according to classes,

Class	Condition Number
Iris-setosa (Class 0)	25.373914506239237
Iris-versicolor (Class 1)	49.832063916410355
Iris-virginica (Class 2)	20.29001629283431

Question 3:

Done by using python, code can be found in my GitHub: https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW2_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)
```

```
"""
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 Principle Component Analysis
Return: Principle Component matrix according to num_pc
"""
def pca(features, num_pc, colvar=True):
    features = features.T if not colvar else features
    sigma = np.cov(features, rowvar=False)
    u, s, v = np.linalg.svd(sigma)
    return u[:, 0:num_pc]
```

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

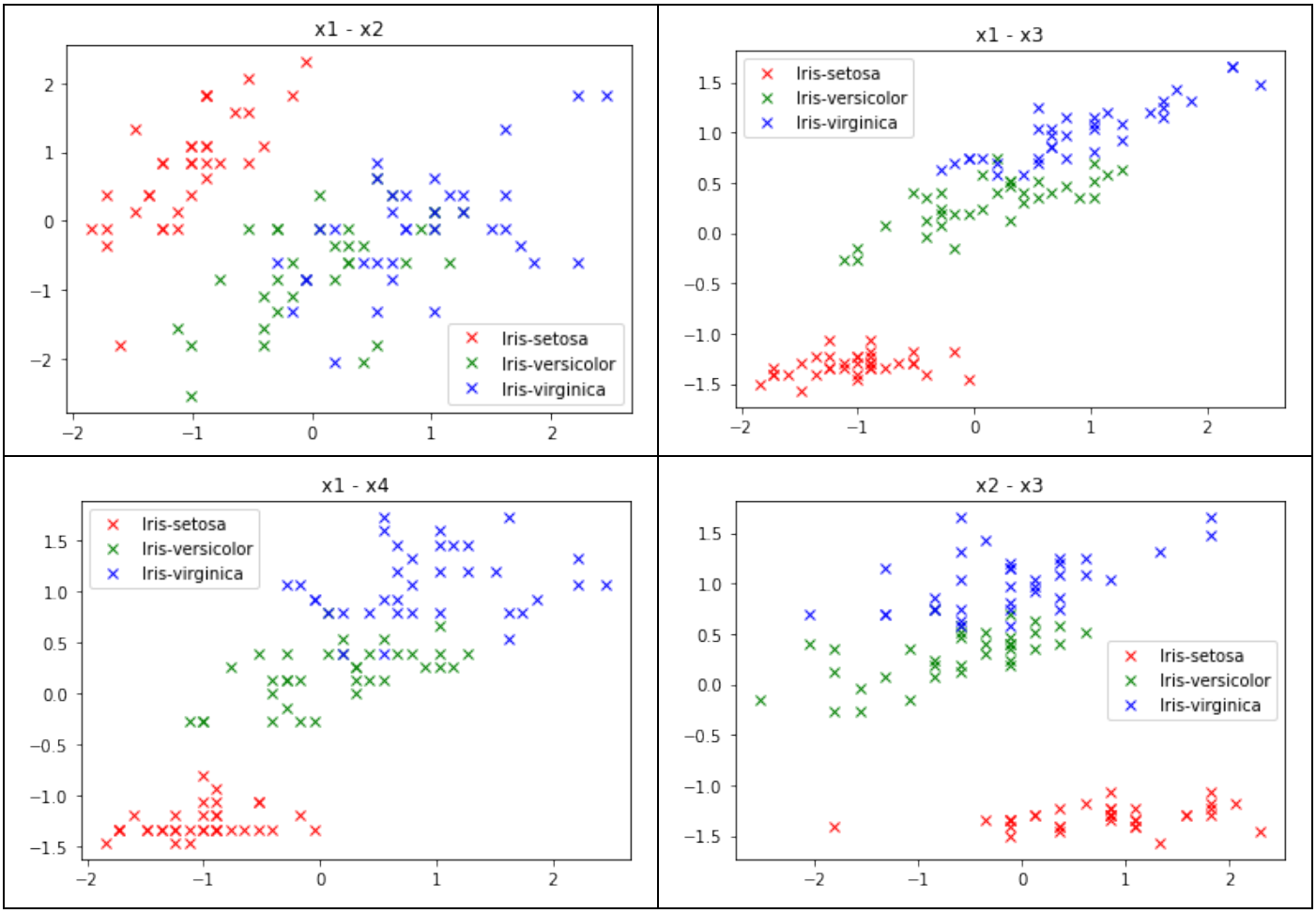
```
    #Properties#
    train_data_size = train_features.shape[0]
    train_features_size = train_features.shape[1]
    inf_data_size = inf_features.shape[0]

    #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
```

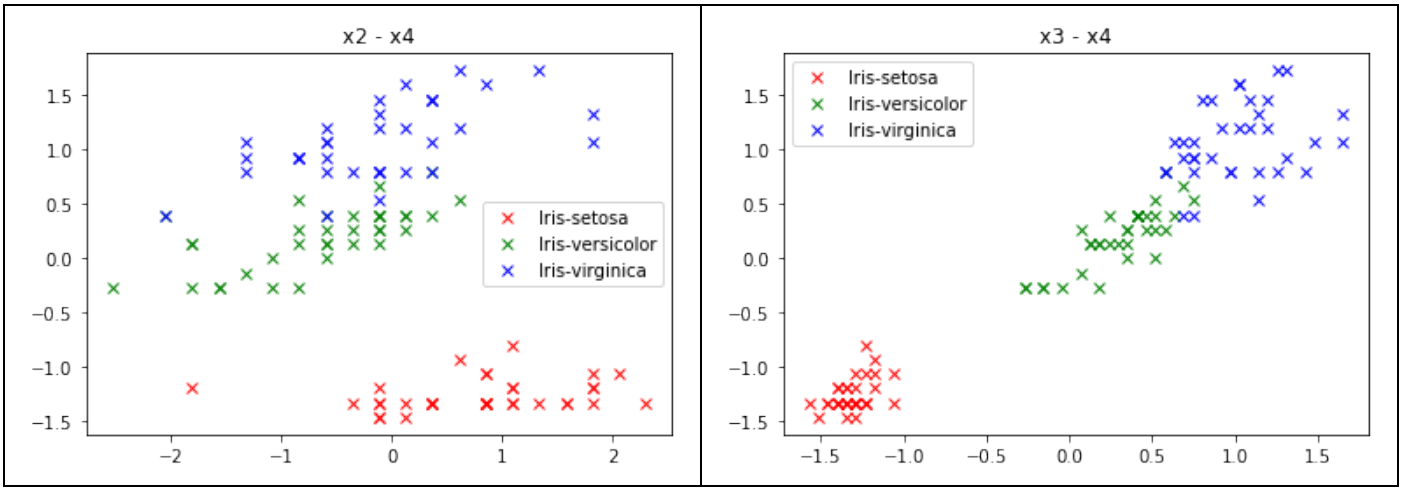
```
"""
Plot the scatter graphs
"""
def plot_scatter(features, labels):
    num_features = features.shape[1]
    for i in range(num_features - 1):
        for j in range(i + 1, num_features):
            plt.plot(features[np.squeeze(labels == float(0))][:, i],
features[np.squeeze(labels == float(0))][:, 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
```

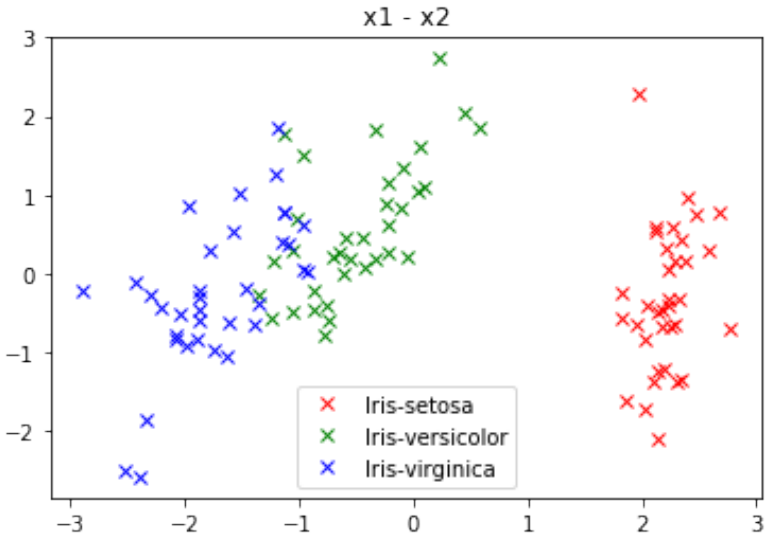
```
features, labels = load_data(classes)
avg_accuracy = 0.
for i in range(cross_val_times):
    train_features, train_labels, test_features, test_labels = data_preparation(features,
labels)
    w = pca(train_features, 2)
    train_features_reduced = np.dot(train_features, w)
    test_features_reduced = np.dot(test_features, w)
    predicts = kNN_inference(train_features_reduced, train_labels, test_features_reduced)
    avg_accuracy += np.mean((predicts == test_labels).astype(np.float32)) / cross_val_times
print('Scatter Plots of Original Data: ')
plot_scatter(train_features, train_labels)
print('Scatter Plots of PCA-transformed Data: ')
plot_scatter(train_features_reduced, train_labels)
print('Average Accuracy ({} times): {}'.format(cross_val_times, avg_accuracy))
```

Scatter Plots of Original Data,





Scatter Plots of PCA-transformed Data,



The average accuracy of 10 times is 0.9133333384990693.

The average accuracy of 10000 times is 0.9007533392489185.

Question 4:

Done by using python, code can be found in my GitHub: https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW2_Q4.ipynb

Nearly all the functions used are as the same as the functions used in Question 3. The following are specific for this question.

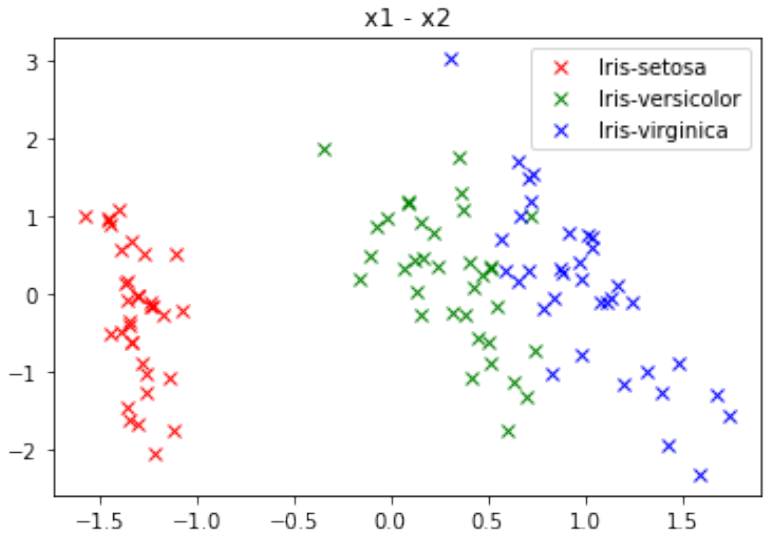
```
"""
Perform Factor Analysis
Return: Dimension-reduced features
"""
def fa(features, num_component, colvar=True):
    features = features.T if not colvar else features
    factor_analysis = decomposition.FactorAnalysis(n_components=num_component)
```

```
    features_reduced = factor_analysis.fit_transform(features)
    return features_reduced
```

```
features, labels = load_data(classes)
avg_accuracy = 0.
for i in range(cross_val_times):
    train_features, train_labels, test_features, test_labels = data_preparation(features,
labels)
    train_features_reduced = fa(train_features, 2)
    test_features_reduced = fa(test_features, 2)
    predicts = kNN_inference(train_features_reduced, train_labels, test_features_reduced)
    avg_accuracy += np.mean((predicts == test_labels).astype(np.float32)) / cross_val_times
print('Scatter Plots of Original Data: ')
plot_scatter(train_features, train_labels)
print('Scatter Plots of FA-transformed Data: ')
plot_scatter(train_features_reduced, train_labels)
print('Average Accuracy ({} times): {}'.format(cross_val_times, avg_accuracy))
```

Scatter Plots of Original Data are nearly the same as the plots in Question 3.

Scatter Plots of FA-transformed Data,



The average accuracy of 10 times is 0.9311111152172088.

Question 5:

Done by using python, code can be found in my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW2_Q5.ipynb

Nearly all the functions used are as the same as the functions used in Question 3. The following are specific for this question.

```
"""
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
```

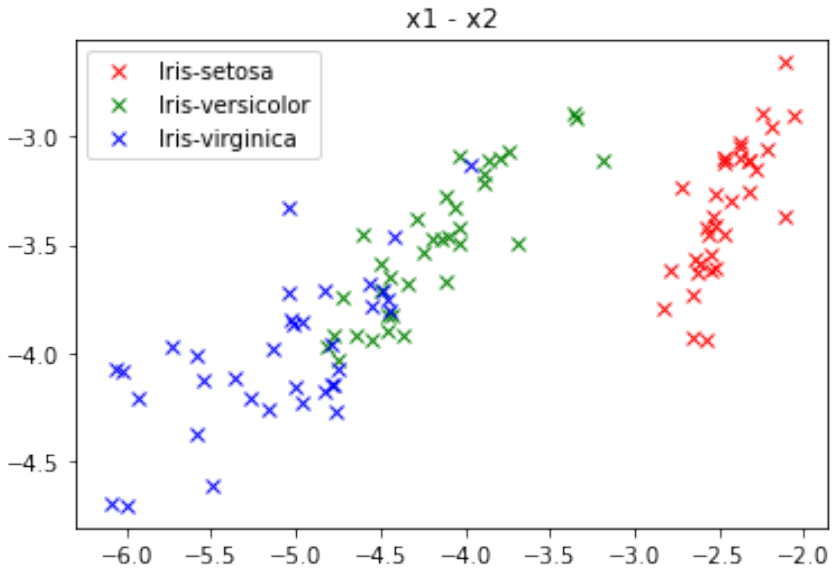
```
"""
Perform Linear Discriminant Analysis
Return: Eigenvalues and eigenvectors
"""
def lda(features, labels, colvar=True):
    features = features.T if not colvar else features

    #Compute means among features for each class#
    mean_c = np.empty((0, features.shape[1]))
    for each_class in classes:
        mean_c = np.append(mean_c, np.mean(features[np.squeeze(labels ==
float(each_class[1]))], axis=0, keepdims=True), axis=0)

    #Compute the scatter matrices#
    S = np.zeros((features.shape[1], features.shape[1]))
    for each_class in classes:
```

```
        f_class = features[np.squeeze(labels == float(each_class[1]))]
        S += np.dot((f_class - mean_c[each_class[1]].reshape(1, -1)).T, (f_class -
mean_c[each_class[1]].reshape(1, -1)))

    #Compute the between-class scatter matrix#
    mean = np.mean(features, axis=0, keepdims=True)
    S_b = np.zeros((features.shape[1], features.shape[1]))
    for each_class in classes:
        n_c = features[np.squeeze(labels == float(each_class[1]))].shape[0]
        S_b += n_c * np.dot((mean_c[each_class[1]].reshape(1, -1) - mean).T,
(mean_c[each_class[1]].reshape(1, -1) - mean))

    #Solving the generalized eigenvalues and eigenvectors#
    eigen_vals, eigen_vects = np.linalg.eig(np.dot(np.linalg.inv(S), S_b))

    #Sort the eigenvalues corresponding to their eigenvectors#
    sort_arg = np.argsort(np.abs(eigen_vals))[::-1]
    eigen_vals = eigen_vals[sort_arg]
    eigen_vects = eigen_vects[sort_arg]
    return eigen_vals, eigen_vects
```

```
features, labels = load_data(classes)
avg_accuracy = 0.
for i in range(cross_val_times):
    train_features, train_labels, test_features, test_labels = data_preparation(features,
labels)
    eigen_vals, eigen_vects = lda(train_features, train_labels)
    train_features_reduced = np.dot(train_features, eigen_vects[0:2].T)
    test_features_reduced = np.dot(test_features, eigen_vects[0:2].T)
    predicts = kNN_inference(train_features, train_labels, test_features)
    avg_accuracy += np.mean((predicts == test_labels).astype(np.float32)) / cross_val_times
print('Scatter Plots of Original Data: ')
plot_scatter(train_features, train_labels)
print('Scatter Plots of LDA-transformed Data: ')
plot_scatter(train_features_reduced, train_labels)
print('Average Accuracy ({} times): {}'.format(cross_val_times, avg_accuracy))
print('Variance Restored: {}'.format(np.sum(np.abs(eigen_vals[0:2])) /
np.sum(np.abs(eigen_vals))))
```

Scatter Plots of Original Data are nearly the same as the plots in Question 3.

Scatter Plots of LDA-transformed Data,



The average accuracy of 10 times is 0.9711111128330231.
The variance Restored is 0.9999999999999998.

Comparing the three results from using PCA, FA, and LDA with several experiment repeated. PCA provides generally fair performance, FA, in this case, provides an unstable performance. The accuracy vibrates between 87%~93% after testing for several trials of cross validation. The LDA gives us the best performance with the fact that accuracy stably lies between 95~97%. In addition, the first two bases calculated by using LDA gives us a variance restored of 99.999%. From the scatter graphs, we can see that LDA gives us a clear and better transformation.