Machine Learning
Homework #1

Daniel Ho Kwan Leung
104360098
CSIE, Year 3

Question 1:

Let H be the activity of being infected HIV, H^{C} be the complement of H. Let $+$ be the activity of being tested as positive, $-$ be the complement of $+$. According to the question, it stated that $\mathrm{P}(-|\mathrm{H}) = 0.001$, $\mathrm{P}(+|\mathrm{H}^{\mathrm{C}}) = 0.001$, $\mathrm{P}(\mathrm{H}) = 0.0001$. We can derive that $\mathrm{P}(+|\mathrm{H}) = 0.999$, $\mathrm{P}(-|\mathrm{H}^{\mathrm{C}}) = 0.999$ and $\mathrm{P}(\mathrm{H}^{\mathrm{C}}) = 0.9999$.

By conditional probability and Bayes' Theorem,

$$P(H|+) = \frac{P(H \cap +)}{P(+)}$$

$$P(H|+) = \frac{P(+|H)P(H)}{P(+ \cap H) + P(+ \cap H^C)}$$

$$P(H|+) = \frac{P(+|H)P(H)}{P(+|H)P(H) + P(+|H^C)P(H^C)}$$

$$P(H|+) = \frac{0.999 \cdot 0.0001}{0.999 \cdot 0.0001 + 0.001 \cdot 0.9999}$$

$$P(H|+) = 0.0908346972176759$$

The probability of whether the person is really infected is 0.0908346972176759.

Common code used in Question 2, 3, 5:

All the code written in this homework are in Python.

The whole code for assignments in this class are uploaded to the repository:

https://github.com/DHKLeung/NTUT_Machine_Learning

```
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
```

Question 2:

The following code are functions that are used to build the k-NN model as well as inferencing.

```
"""
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
```

The following code are built for answering question 2.

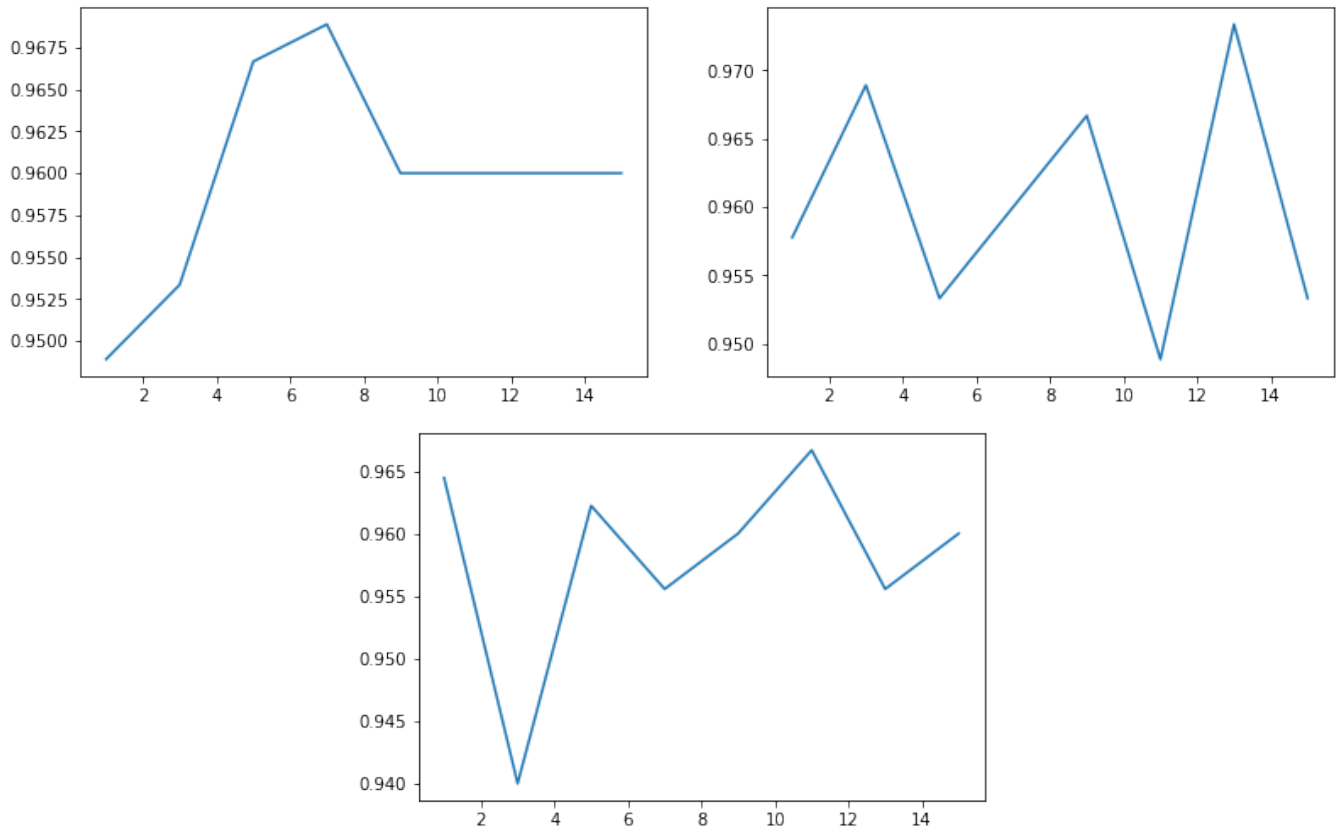
```
"""
Settings
```

```
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
cross_val_times = 10
ks = np.arange(1, 16, 2)
```

```
features, labels = load_data(classes)
records = []
for k in ks:
    accuracy = 0.
    for i in range(cross_val_times):
        features, labels = data_shuffling(features, labels)
        train_features, train_labels, test_features, test_labels = data_split(features,
labels)
        predictions = kNN_inference(train_features, train_labels, test_features)
        accuracy += np.mean((predictions == test_labels).astype(np.float32)) /
cross_val_times
    records.append((k, accuracy))
records = np.array(records)
plt.plot(records[:, 0], records[:, 1])
plt.show()
```

All the code are uploaded to my GitHub with Jupyter Notebook used so a tidy view can be shown. The code are in

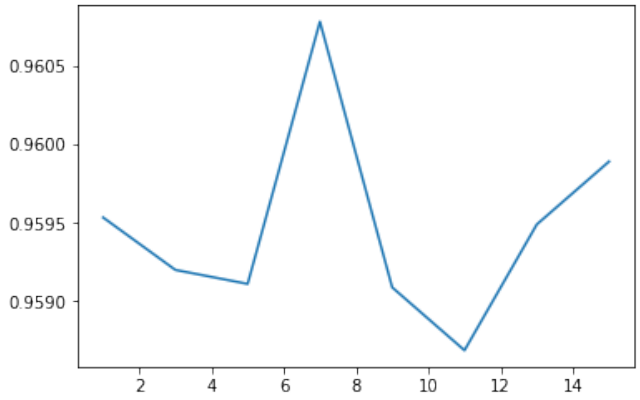
https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW1_Q2.ipynb

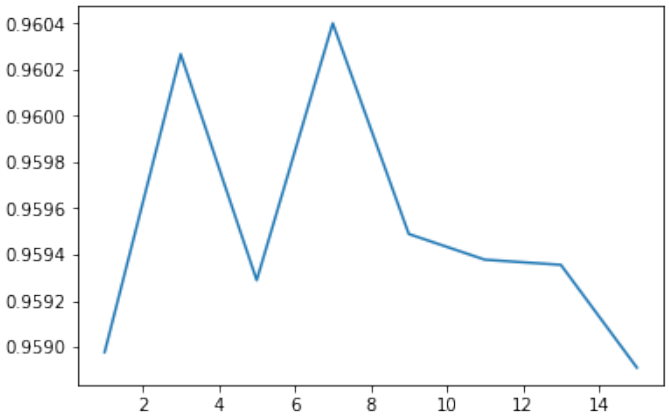


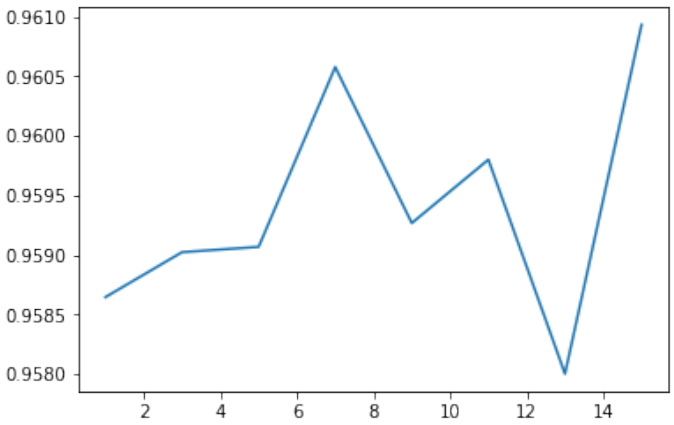
The above graphs show k-accuracy graph for 3 complete trials where the accuracies are computed by averaging the inference result of 10 times for each k chosen with reference to the question 2. The horizontal axis represents the k chosen and the vertical axis represents the accuracy.

We can't determine the best k from the above experiment because the result and graph generated in each trial are not consistent with the previous. Sometimes, larger value of k performs better but sometimes not. However, despite the unsteady behaviour, we can conclude that the performance of accuracies are always about 96%.

The following graphs show 3 complete trials where the accuracies are computed by averaging the inference result of 1000 times for each k chosen with reference to the question 2.







We can figure out that when $k = 7$, we can always get a satisfactory performance which achieving about 96.05% accuracy.

Question 3:

Assume that we don't have the test set. So what we have is the original train set of 105 records. To figure out the optimal k, we split the original train set into train set and validation set which is of the ratio of 7:3.

As a result, the new train set is of 74 records and validation set is of 31 records. We still run and average 1000 times of cross validation for each k. And repeat the whole process for 3 trial.

The k-NN architecture is the same as the code shown in question 2. Only the part of inference and settings are altered. The altered parts are shown below:

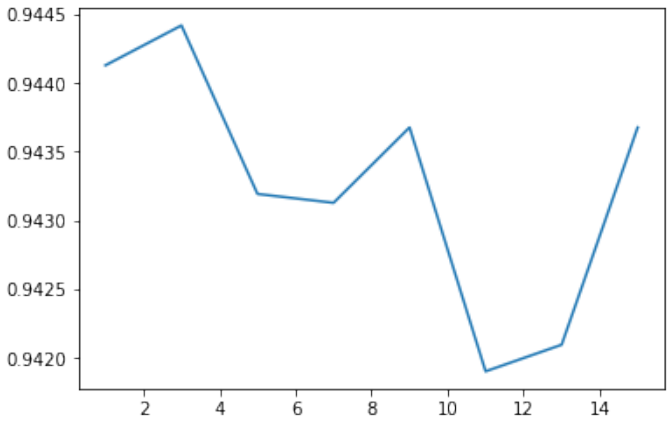
```
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
cross_val_times = 1000
ks = np.arange(1, 16, 2)
```

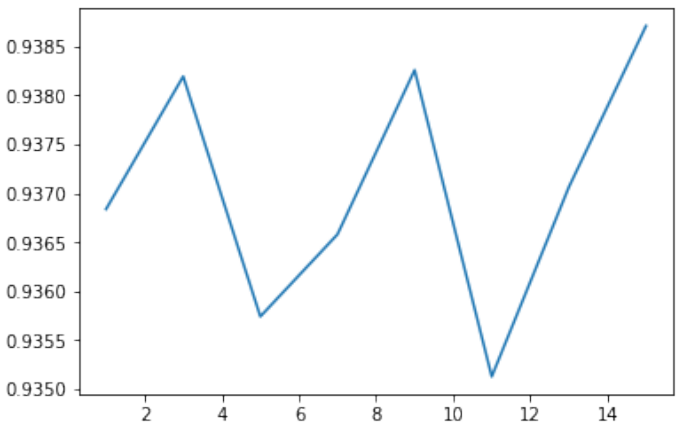
```
features, labels = load_data(classes)
features, labels = data_shuffling(features, labels)
features, labels, _, _ = data_split(features, labels)
records = []
for k in ks:
    accuracy = 0.
    for i in range(cross_val_times):
        features, labels = data_shuffling(features, labels)
        train_features, train_labels, validation_features, validation_labels =
data_split(features, labels)
        predictions = kNN_inference(train_features, train_labels, validation_features)
        accuracy += np.mean((predictions == validation_labels).astype(np.float32)) /
cross_val_times
    records.append((k, accuracy))
records = np.array(records)
plt.plot(records[:, 0], records[:, 1])
plt.show()
```

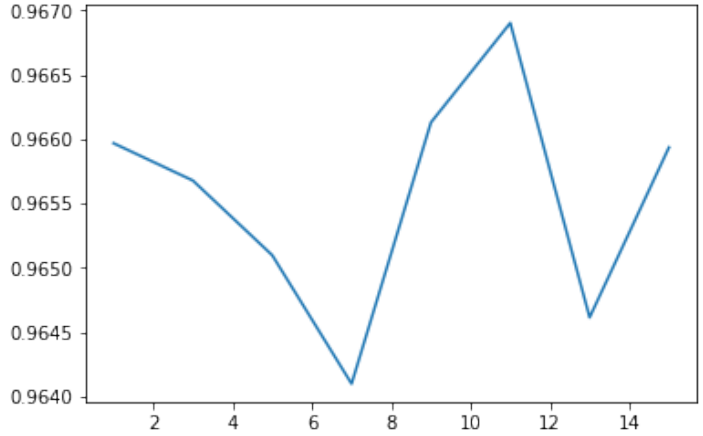
The above code are uploaded to my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW1_Q3.ipynb

The graphs of the three trials are shown below:







The above graphs show 3 complete trials where the accuracies are computed by averaging the inference result of 1000 times for each k chosen with reference to the question 3.

In this case, we can figure out that $k = 9$ should be the optimal value since for three trials where each trial runs and averages the accuracies of 10 times for each k shows that $k = 9$ achieves satisfactory performance which its accuracy is about 95%.

Question 4:

If the features are all of discrete-type, the calculation of Naïve Bayes Classifier is very easy by just counting. However, if the features are of continuous-type, there are two ways to cope with it. Discretisation and Using probability density functions.

It is much easier to just discretise the continuous numeric features. For instance, if we are discretising sepal length of irises. We may have a partition of the fact that $4.0 \sim 4.9$ belongs to "small", $5.0 \sim 5.9$ belongs to "fair", $6.0 \sim 6.9$ belongs to "big". As a result, we can categorise those sepal lengths in to the three types of discrete values and continue to construct our Naïve Bayes Classifier. However, the discretisation process is quite subjective, thus, it may cause serious loss of information.

Another way is to use probability density functions to compute the probability density of the class but obviously not the probability. The reason is that the calculation of probability via using probability density function is to compute the integration of the probability density function of certain range of random variables. So the probability density function itself is not probability. However, in the case of using Naïve Bayes Classifier for continuous-type features, the features are just a single numerical values, there is not such "range" for us to perform integration. So we are using the probability density to compare within the classes but not the probability.

Concretely, the forms of using Naïve Bayes Classifier for continuous-type features are actually the same as the one in discrete-type features. We can still derive that till the following,

$$P(C|x_1, x_2, \dots, x_n) = \frac{P(C \cap x_1, x_2, \dots, x_n)}{P(x_1, x_2, \dots, x_n)}$$

$$P(C|x_1, x_2, \dots, x_n) = \frac{P(x_1 \cap x_2 \cap \dots \cap x_n|C)}{P(x_1 \cap x_2 \cap \dots \cap x_n)}$$

Due to the independent assumption of random variables in Naïve Bayes Classifier. We can derive that

$$P(C|x_1, x_2, \dots, x_n) = \frac{P(x_1|C)P(x_2|C) \dots P(x_n|C)P(C)}{P(x_1 \cap x_2 \cap \dots \cap x_n)}$$

$$P(C|x_1, x_2, \dots, x_n) = \frac{P(C)\prod_{i=1}^{n} P(x_i|C)}{\prod_{i=1}^{n} P(x_i)}$$

Coming to this part, the denominator $\prod_{i=1}^{n} P(x_i)$ is not so significant that we can just ignore it because it is a constant.

We actually don't compute the probability but the probability density function as mentioned above paragraph. We can first assume our dataset are of Gaussian distribution (or normal distribution), thus, we can firstly calculate the standard deviation $\sigma_{i,C}$ and the mean $\mu_{i,C}$ for our dataset by class and by feature.

In short, what we are actually calculating is,

$$P(C)\prod_{i=1}^{n} pdf(x_i|C)$$

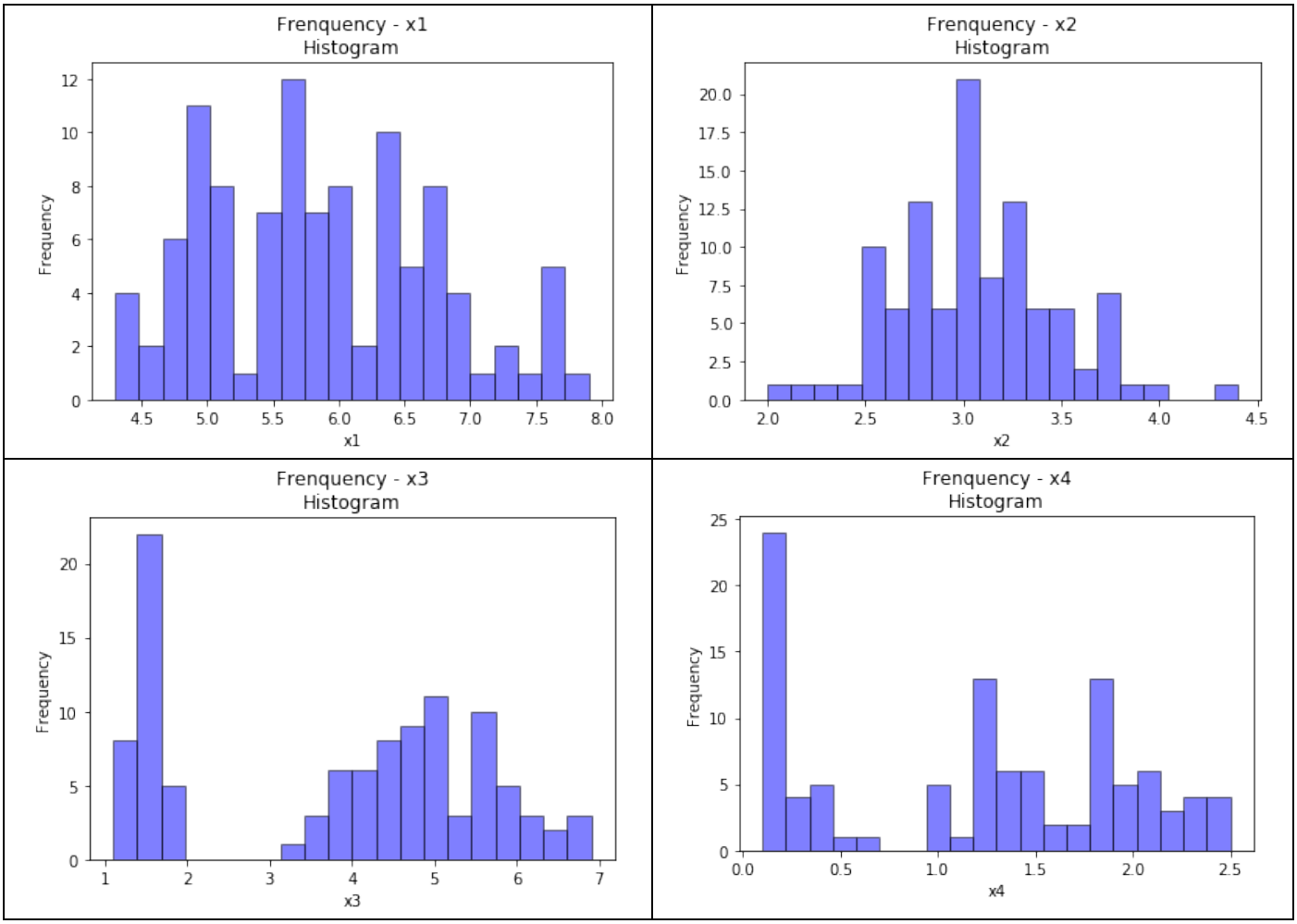
$$pdf(x_i|C) = \frac{1}{\sigma_{i,C}\sqrt{2\pi}} e^{-\frac{(x_i-\mu_{i,C})^2}{2\sigma_{i,C}{}^2}}$$

Thus, we can obtain the probability density and by comparing them. The class with highest probability density is the class predicted.

Question 5:

As mentioned in the answer of question 4, Naïve Bayes Classifier can be dealing with continuous-type features in two ways. However, the discretisation method is too simple and general and is like what have been taught in class, thus, I would like to implement the way of using probability density function.

We can first observe the data distribution. The following are the data distribution plotted in histogram. x1, x2, x3, x4 are the features of the iris dataset.



Theoretically, we should implement this by finding probability distribution which fit the distribution of features. However, in most cases, we directly use the Gaussian distribution instead.

The code related to this part:

```
"""
Compute the probability density using gaussian distribution
"""
def get_prob_den(x, sample_var, mean):
    prob_den = (1 / np.sqrt(2 * np.pi * sample_var)) * np.exp(-np.square(x - mean)/(2 *
sample_var))
    return prob_den
```

```
"""
Build Naive Bayes Classifier and preform inferencing
Return: predictions to test set
"""
def naive_bayes_classifier(train_features, train_labels, test_features):
    p_c = {}
    var = {}
    mean = {}
    for i in range(3):
        p_c[str(i)] = np.sum((train_labels[:, 0] == np.float(i)).astype(np.int32)) /
train_labels.shape[0]
        for j in range(4):
            var['c{}x{}'.format(i, j + 1)] = np.var(train_features[train_labels[:, 0] ==
np.float(i)][:, j]) * 4 / 3 #sample variance
```

```
            mean['c{}x{}'.format(i, j + 1)] = np.mean(train_features[train_labels[:, 0] ==
np.float(i)][:, j])
    predictions = []
    for k in range(test_features.shape[0]):
        probs = []
        for i in range(3):
            prob = p_c[str(i)]
            for j in range(4):
                prob *= get_prob_den(test_features[k, j], var['c{}x{}'.format(i, j + 1)],
mean['c{}x{}'.format(i, j + 1)])
            probs.append(prob)
        predict = np.argmax(probs)
        predictions.append((predict))
    predictions = np.array(predictions).reshape(-1, 1)
    return predictions
```

```
"""
Settings
"""
classes = [('Iris-setosa', 0), ('Iris-versicolor', 1), ('Iris-virginica', 2)]
cross_val_times = 1000
```

```
features, labels = load_data(classes)
accuracy = 0.
for i in range(cross_val_times):
    features, labels = data_shuffling(features, labels)
    train_features, train_labels, test_features, test_labels = data_split(features, labels)
    predictions = naive_bayes_classifier(train_features, train_labels, test_features)
    accuracy += np.mean((predictions == test_labels).astype(np.float32)) / cross_val_times
print('Accuracy ({} times average): {}'.format(cross_val_times, accuracy))
```

The above code are uploaded to my GitHub:

https://github.com/DHKLeung/NTUT_Machine_Learning/blob/master/HW1_Q5.ipynb

As what have been done in question 2 and 3, I ran a total of 3 trials. Each trial averages the accuracies of 1000 times. The results are shown below:

```
Accuracy (1000 times average): 0.9543111138343897
Accuracy (1000 times average): 0.9551555582285002
Accuracy (1000 times average): 0.954644447147855
```

As the above image, the accuracy of using Naïve Bayes Classifier is very stable. We can always achieve about 95% accuracy, which is much stable and relative easier than k-NN implemented in question 2 and 3. k-NN requires the tuning of hyperparameters (meta-parameters) k. However, as a parametric model, Naïve Bayes Classifier doesn't need to be tuned. So the implementation of Naïve Bayes Classifier is much efficient and the performance is of fair satisfaction.