Machine Learning HW1

Department of Computer Science and Engineering

Name: Chan-Wei Hu

UIN: 231003486

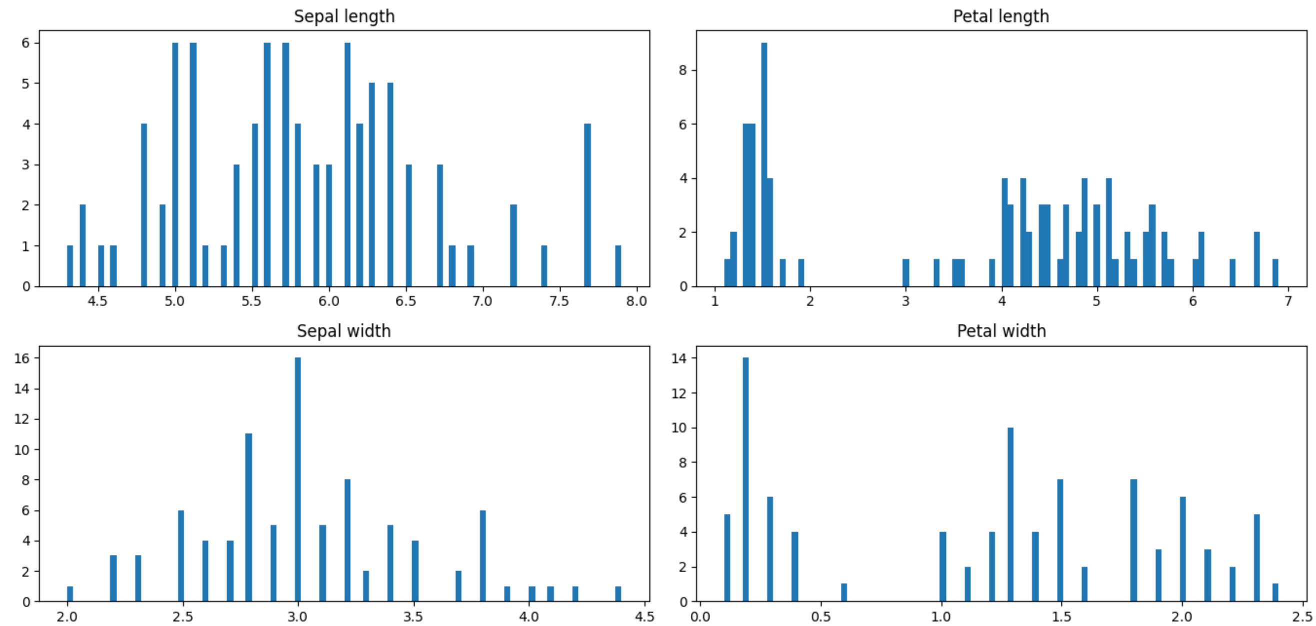
Question 2

(a.i)

The numbers of samples belonging to each class are all **30**, indicating that the classes are equally distributed!

(a.ii)

The histogram is plotted in following figure.



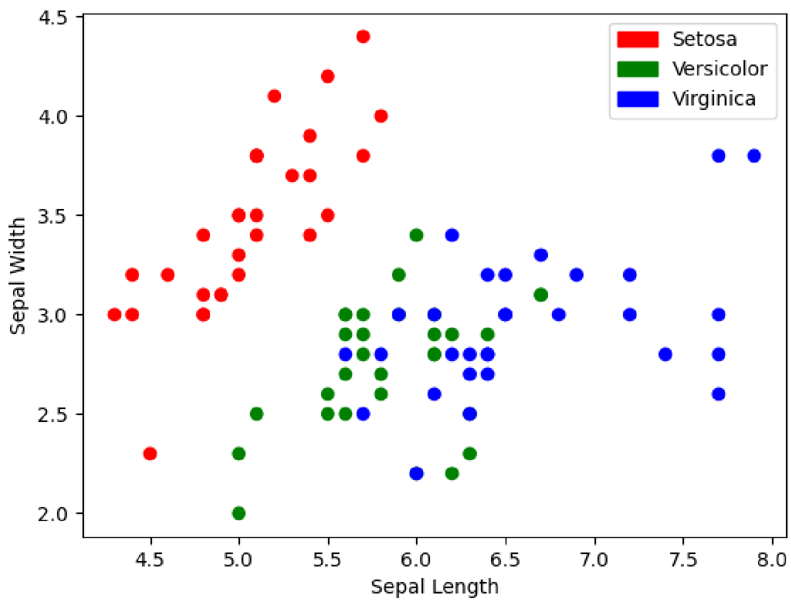
I observed that:

- <u>Sepal Length:</u> It is not obviously belonging to any distribution, but basically the length lies in the range from 4.8 to 6.4.
- <u>Sepal Width</u>: It is apparently an unimodal distribution because a clear peak with width 3.0 is presented. (Almost normal distribution)
- <u>Petal Length:</u> It is a bimodal distribution because two peaks are observed, but the right one doesn't have a clear peak.
- <u>Petal Width</u>: It is a trimodal distribution but two distribution on the right don't have a clear peak.

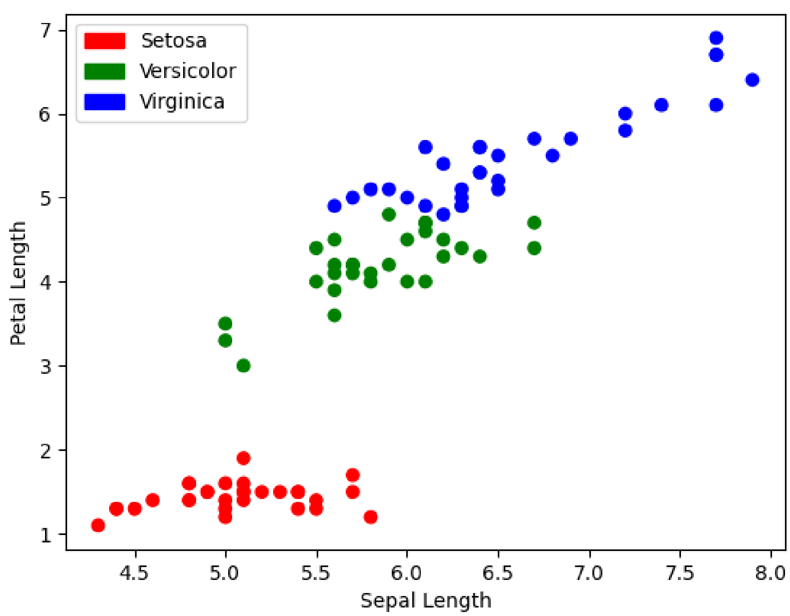
(a.iii)

In this sub-problem, I use red for Setosa, green for Versicolor, and blue for Virginica.

➔ Pair 1: (sepal length, sepal width)
In this pair, it is hard to distinguish Versicolor and Virginica because the features are not separable. However, Setosa can be separated based on these two features.

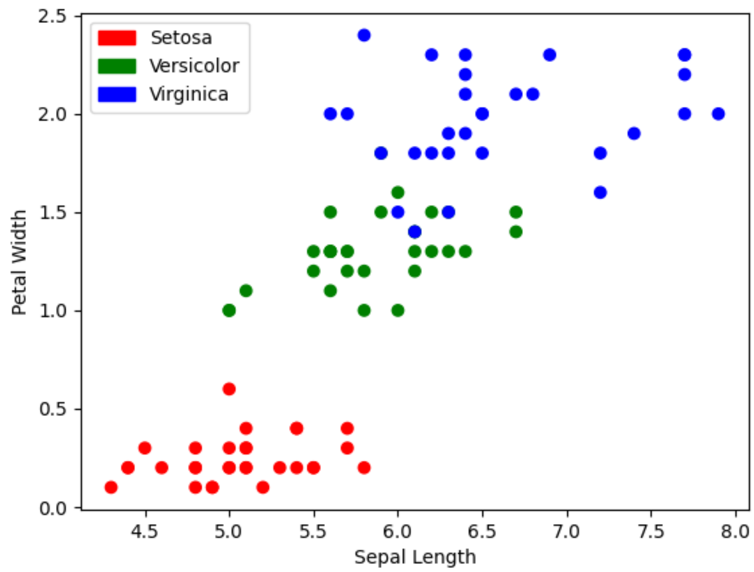


➔ Pair 2: (sepal length, petal length)
Same as in pair 1, Setosa can be separated easily. Also Versicolor and Virginica are divided but really close to each other. This may lead to mis-classification if KNN is used.



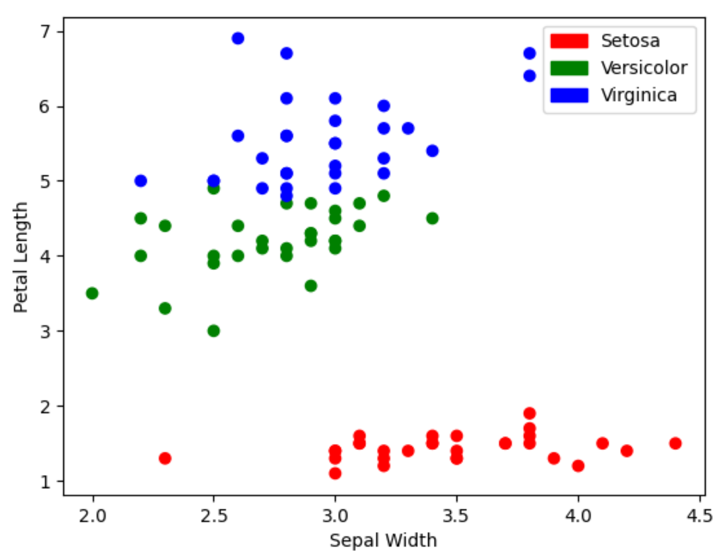
➔ Pair 3: (sepal length, petal width)

This pair is similar with pair 1. Setosa is separated from the other two classes but Versicolor and Virginica are close, which may cause classification error using KNN.



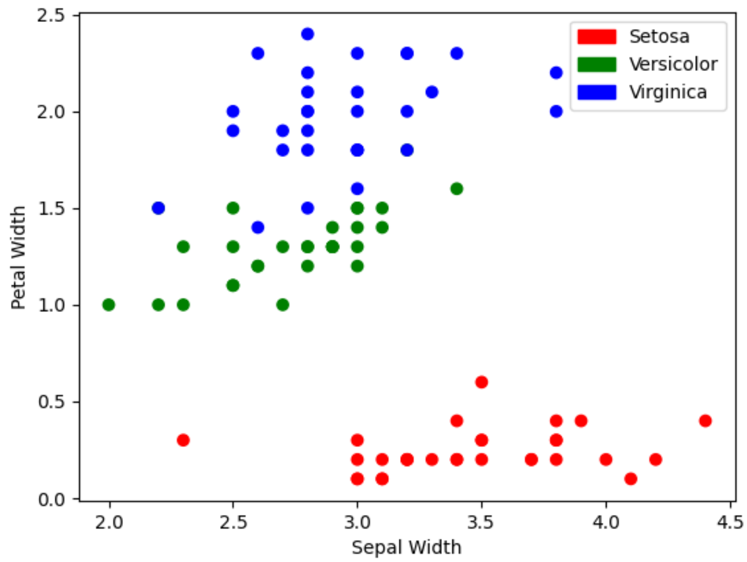
➔ Pair 4: (sepal width, petal length)

Same as pair 2, Setosa is clearly separated from other two.

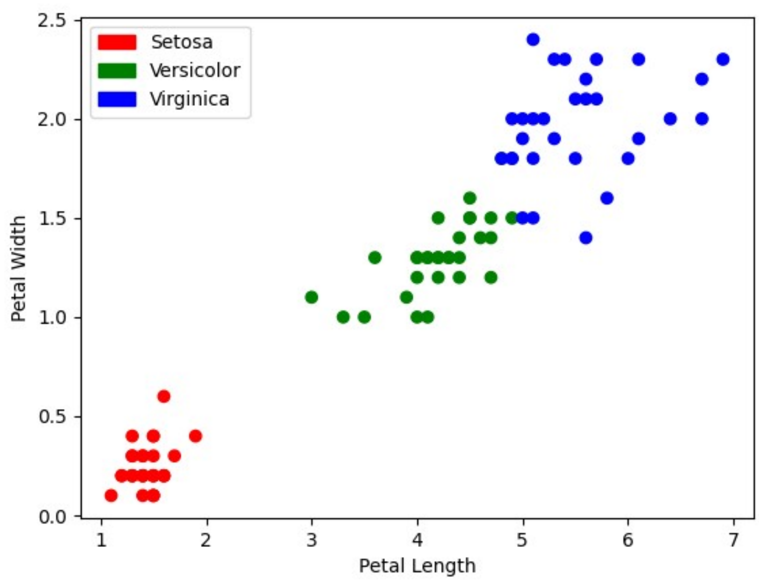


➔ Pair 5: (sepal width, petal width)

Versicolor and Viriginica are not clearly divided, while Setosa is apparently far from those.



➔ Pair 6: (petal length, petal width)



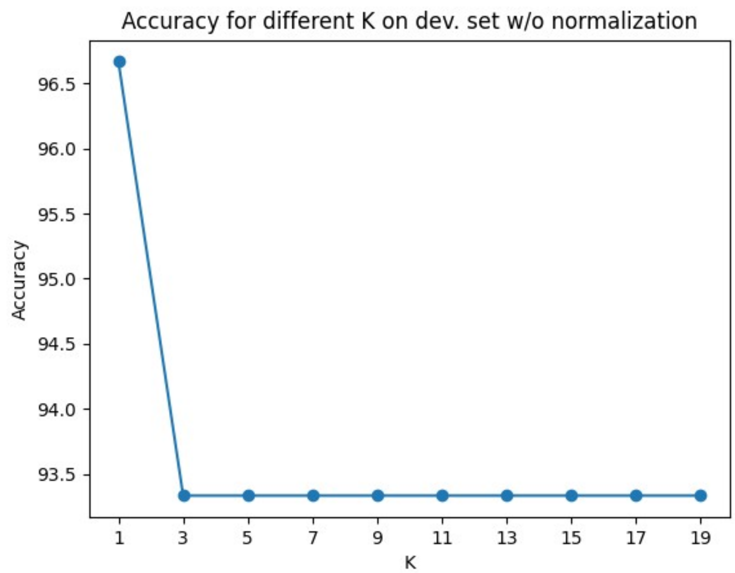
To sum up, pair 2 and 6 are the most separable features that can be used for classification because there are clearly the division between three classes. However, Versicolor and Virginica might be misclassification if using KNN.

(b.i)

Please refer to code in Appendix (class KNN) I implemented a class for KNN algorithm.

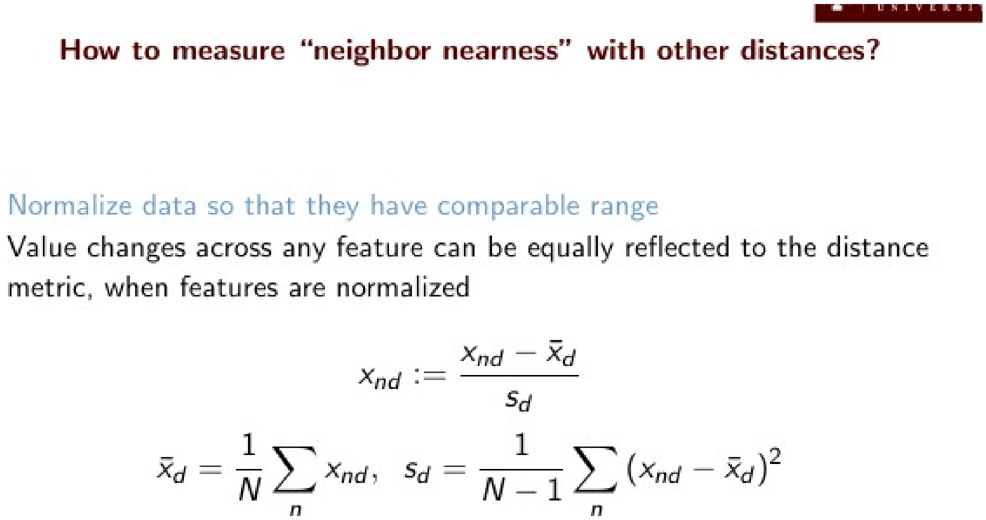
(b.ii)

I have ran the KNN algorithm for K=1, 3, ..., 19. The result is as follows.

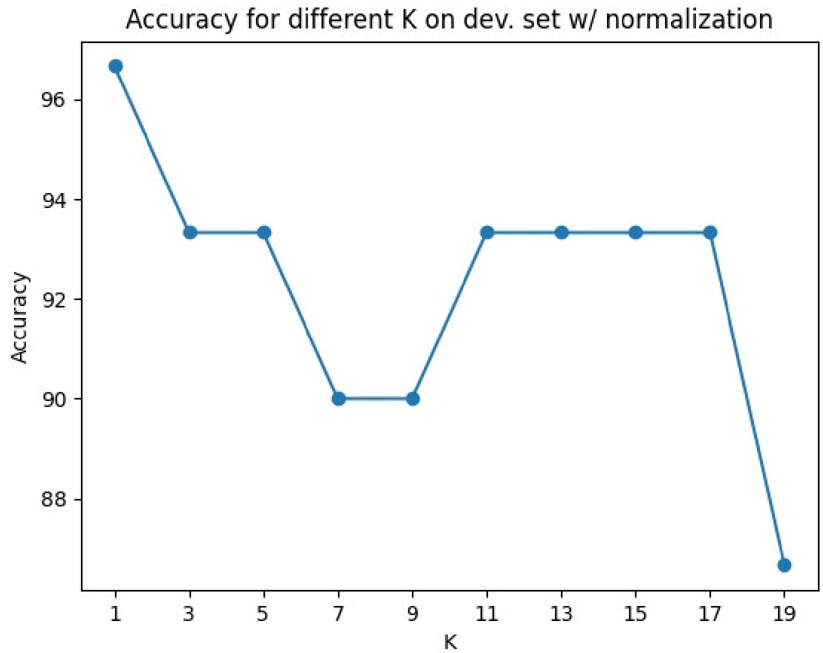


As the result shows, the accuracy with K=1 is the highest (Acc = 96.67%). Thus, the best hyper-parameter K* is 1.

However, since the features are in different scales, I performed normalization on Iris dataset to get comparable range. As slide shows.



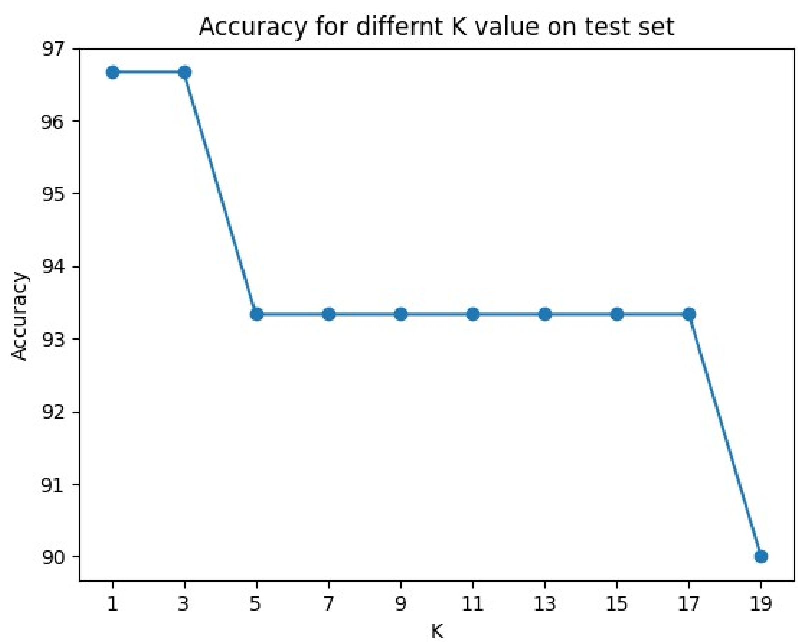
The result is shown as below.



K=1 is also the best! Thus, I concluded that K*=1, with accuracy = 96.67%.

(b.iii)

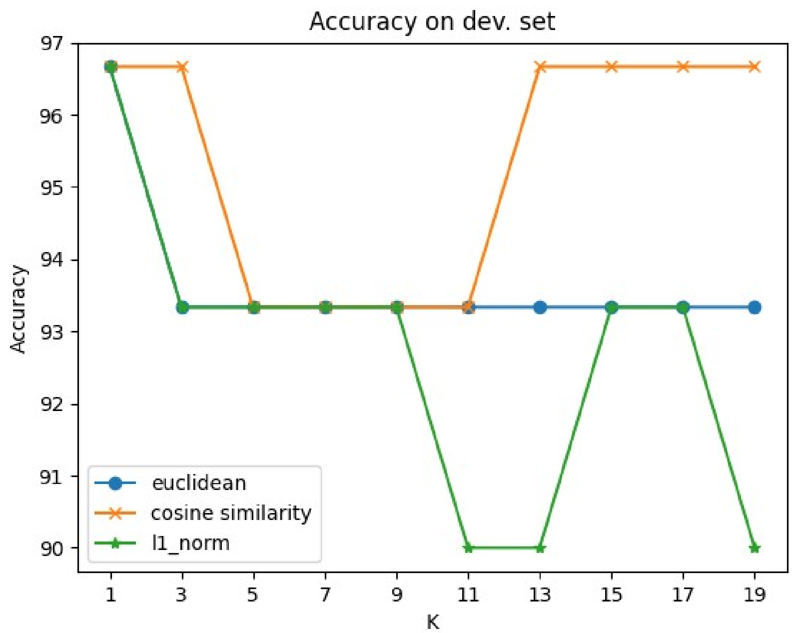
The testing result is below.

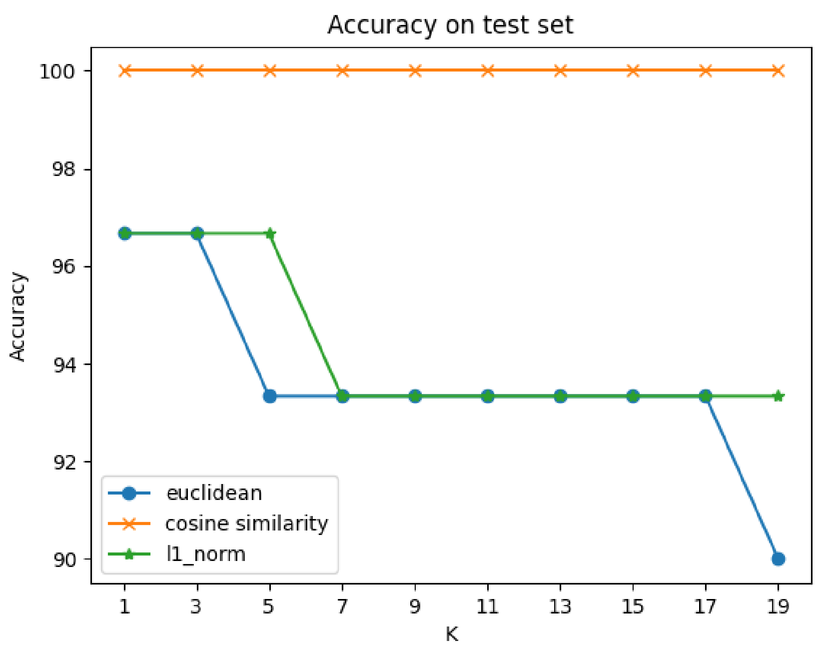


From the result, both K=1 and 3 achieve the best accuracy.

(b.iv)

I compare Euclidean distance with cosine similarity and l1-norm. The result is as follows.





As the result shows, using cosine similarity as distance measurement achieves the best result on test data!

Appendix (full code)

```
import pdb  # debugging tool :)
import math
import argparse
import numpy as np
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
from itertools import combinations

# this is set as a global map
label_map = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-
virginica': 2}

class DataPoint(object):
    def __init__(self, feats):
        self.s_l = feats['sepal_l']
        self.s_w = feats['sepal_w']
        self.p_l = feats['petal_l']
        self.p_w = feats['petal_w']
        self.label = feats['label']

    def get_feature(self, bias_flag):
        if bias_flag is True:
            return np.array([self.s_l, self.s_w, self.p_l, self.p_w])
        return np.array([self.s_l, self.s_w, self.p_l, self.p_w],
1.0)

def data_parser(filename):
    data_file = open(filename, 'r') # Open file
    dataset = []
    for index, line in enumerate(data_file):
        if index == 0:
            continue
        sepal_l, sepal_w, petal_l, petal_w, label =
line.strip().split(',')
        dataset.append(DataPoint({'sepal_l': float(sepal_l),
'sepal_w': float(sepal_w), 'petal_l': float(petal_l), 'petal_w':
float(petal_w), 'label': label_map[label]}))
    return dataset

# KNN
class KNN(object):
    def __init__(self, K, is_norm, dist_type, train_feat,
train_label, test_data, test_label):
        self.K = K
        self.train_feat = train_feat
        self.train_label = train_label
        self.test_feat = test_feat
        self.test_label = test_label
```

```
        self.len_test_data = len(test_data)
        self.is_norm = is_norm  # shall you normalize data?
        self.dist_type = dist_type  # what type of distance you want
to use as measurement

    def calDistance(self, feat1, feat2):
        # check feature length
        assert len(feat1) == len(feat2)

        if self.dist_type == "euclidean":
            square_sum = 0
            for i in range(len(feat1)):
                square_sum += math.pow(feat1[i]-feat2[i], 2)
            return math.sqrt(square_sum)
        elif self.dist_type == "cosine":
            dot_product = np.dot(feat1, feat2)
            magnitude = np.linalg.norm(feat1)*np.linalg.norm(feat2)
            return 1-(dot_product/magnitude)
        elif self.dist_type == "l1_norm":
            return np.sum(np.abs(feat1-feat2))
        else:
            raise ValueError("Distance type doesn't exist!")

    def Normalize_data(self):
        mean = np.mean(self.train_feat, axis=0)
        std = np.std(self.train_feat, axis=0)
        norm_train_feat = (self.train_feat-mean)/std
        norm_test_feat = (self.test_feat-mean)/std
        return norm_train_feat, norm_test_feat

    def get_KNeighbor(self):
        if self.is_norm is True:
            train_feat, test_feat = self.Normalize_data()
        else:
            train_feat = self.train_feat
            test_feat = self.test_feat
        # calculate the distance
        sorted_dist = []
        k_neighbor = []
        for i in range(len(test_feat)): # For each row feature
            distance = []
            for j in range(len(train_feat)):
                distance.append([self.calDistance(test_feat[i],
train_feat[j]), self.train_label[j]])
            distance.sort(key = lambda x: x[0])
            # Now we can retrieve k neighbors
            sorted_dist.append(distance[:self.K])
            k_neighbor.append([distance[i][1] for i in
range(self.K)])
```

```
        return k_neighbor

    def get_Accuracy(self):
        k_neighbor = self.get_KNeighbor()
        correct_classified = 0
        for i in range(len(test_label)):
            predict = max(k_neighbor[i], key=k_neighbor[i].count)
            ground_truth = test_label[i]
            if predict == ground_truth:
                correct_classified = correct_classified + 1

        return correct_classified/len(test_label)*100

if __name__ == "__main__":
    # use argument parser to control the input
    parser = argparse.ArgumentParser(description='HW1 arguments')
    parser.add_argument("-train", "--traindata", help="training data
file name")
    parser.add_argument("-test", "--testdata", help="testing data
file name")
    parser.add_argument("-K", "--K", default=21, help="parameter K
for KNN")
    parser.add_argument("-data_norm", "--norm", default=False,
action='store_true', help="Do you want to do data normalization
first?")
    parser.add_argument("-which_dist", "--which_dist",
choices=["euclidean", "cosine", "l1_norm"], default="euclidean",
help="What kind of distance you want for KNN")
    args = parser.parse_args()

    # load the training data first
    train_dataset = data_parser(args.traindata)
    test_dataset = data_parser(args.testdata)
    len_dataset = len(train_dataset)

    # Start answering questions :)
    # (a.i) First calculate the number of each label
    label_list = [train_dataset[idx].label for idx in
range(len_dataset)] # get the list of label
    print("Number of Iris Setosa:
{}".format(label_list.count(label_map['Iris-setosa'])))
    print("Number of Iris Versicolor:
{}".format(label_list.count(label_map['Iris-versicolor'])))
    print("Number of Iris Virginica: {}\
n".format(label_list.count(label_map['Iris-virginica'])))

    # (a.ii) First gather the feature information
    s_l_list = [train_dataset[idx].s_l for idx in range(len_dataset)]
    s_w_list = [train_dataset[idx].s_w for idx in range(len_dataset)]
```

```
    p_l_list = [train_dataset[idx].p_l for idx in range(len_dataset)]
    p_w_list = [train_dataset[idx].p_w for idx in range(len_dataset)]

    # plot the histogram
    n_bins = len_dataset
    fig, axs = plt.subplots(2, 2, figsize=(20, 20), sharex=False,
sharey=False, tight_layout=True)
    axs[0][0].hist(s_l_list, n_bins, density=False)
    axs[0][0].set_title('Sepal length')
    axs[1][0].hist(s_w_list, n_bins, density=False)
    axs[1][0].set_title('Sepal width')
    axs[0][1].hist(p_l_list, n_bins, density=False)
    axs[0][1].set_title('Petal length')
    axs[1][1].hist(p_w_list, n_bins, density=False)
    axs[1][1].set_title('Petal width')
    plt.show()

    # (a.iii)
    # First, get the combination of the feature
    comb = combinations([[s_l_list, "Sepal Length"], [s_w_list,
"Sepal Width"], \
                         [p_l_list, "Petal Length"], [p_w_list,
"Petal Width"]], 2)
    colormap = np.array(['r', 'g', 'b'])
    for _pair in list(comb):
        plt.xlabel(_pair[0][1])
        plt.ylabel(_pair[1][1])
        print(len(_pair[0][0]))
        plt.scatter(_pair[0][0], _pair[1][0], c=colormap[label_list])
        class_0 = mpatches.Patch(color='r', label='Setosa')
        class_1 = mpatches.Patch(color='g', label='Versicolor')
        class_2 = mpatches.Patch(color='b', label='Virginica')
        plt.legend(handles=[class_0, class_1, class_2])
        plt.show()
    # (b.i)
    # Implement KNN, convert list to numpy array
    # Training data
    train_feat = [[train_dataset[idx].s_l, train_dataset[idx].s_w,
train_dataset[idx].p_l, \
                   train_dataset[idx].p_w] for idx in
range(len_dataset)]
    train_feat = np.asarray(train_feat, dtype=np.float32)
    train_label = np.asarray(label_list)

    # Testing/Validation data
    len_test_dataset = len(test_dataset)
    test_feat = [[test_dataset[idx].s_l, test_dataset[idx].s_w,
test_dataset[idx].p_l, \
```

```
                  test_dataset[idx].p_w] for idx in
range(len_test_dataset)]
    test_label = [test_dataset[idx].label for idx in
range(len_test_dataset)]
    test_feat = np.asarray(test_feat, dtype=np.float32)
    test_label = np.asarray(test_label)

    # start
    accu_list = []
    for k in range(1, args.K, 2):
        knn = KNN(k, args.norm, args.which_dist, train_feat,
train_label, test_feat, test_label)
        accu = knn.get_Accuracy()
        accu_list.append(accu)
        print("Accuracy for K = {} is {}\n".format(k, accu))

    # (b_iii): plot the result
    plt.plot(range(1, args.K, 2), accu_list, marker='o')
    plt.xticks(range(1, args.K, 2))
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.title('Accuracy for differnt K value on ' + args.testdata)
    plt.show()

    # (b.iv)
    accu_list = []
    for d in ["euclidean","cosine","l1_norm"]:
        tmp_accu = []
        for k in range(1, args.K, 2):
            knn = KNN(k, args.norm, d, train_feat, train_label,
test_feat, test_label)
            accu = knn.get_Accuracy()
            tmp_accu.append(accu)

        accu_list.append(tmp_accu)

    plt.plot(range(1, args.K, 2), accu_list[0], marker='o')
    plt.plot(range(1, args.K, 2), accu_list[1], marker='x')
    plt.plot(range(1, args.K, 2), accu_list[2], marker='*')
    plt.xticks(range(1, args.K, 2))
    plt.xlabel('K')
    plt.ylabel('Accuracy')
    plt.legend(["euclidean","cosine similarity", "l1_norm"])
    plt.title('Accuracy on test set')
    plt.show()
```