Assignment-2

1 CS171 - Winter 2022

1.0.1 Instructor: Vagelis Papalexakis

In this assignment we will implement two different supervised learning models: 1) linear regression (using gradient descent), and 2) k-nearest neighbor classification. As we did in Assignment 1, here we will also use the Iris dataset. Below are some useful imports and some data bookkeeping:

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
import random as rand
from sklearn.model_selection import train_test_split
from collections import Counter
data_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',␣
 ↪'label']
data = pd.read_csv('iris.data',
                   names = data_names)
```

1.1 Question 1: Linear Regression [50%]

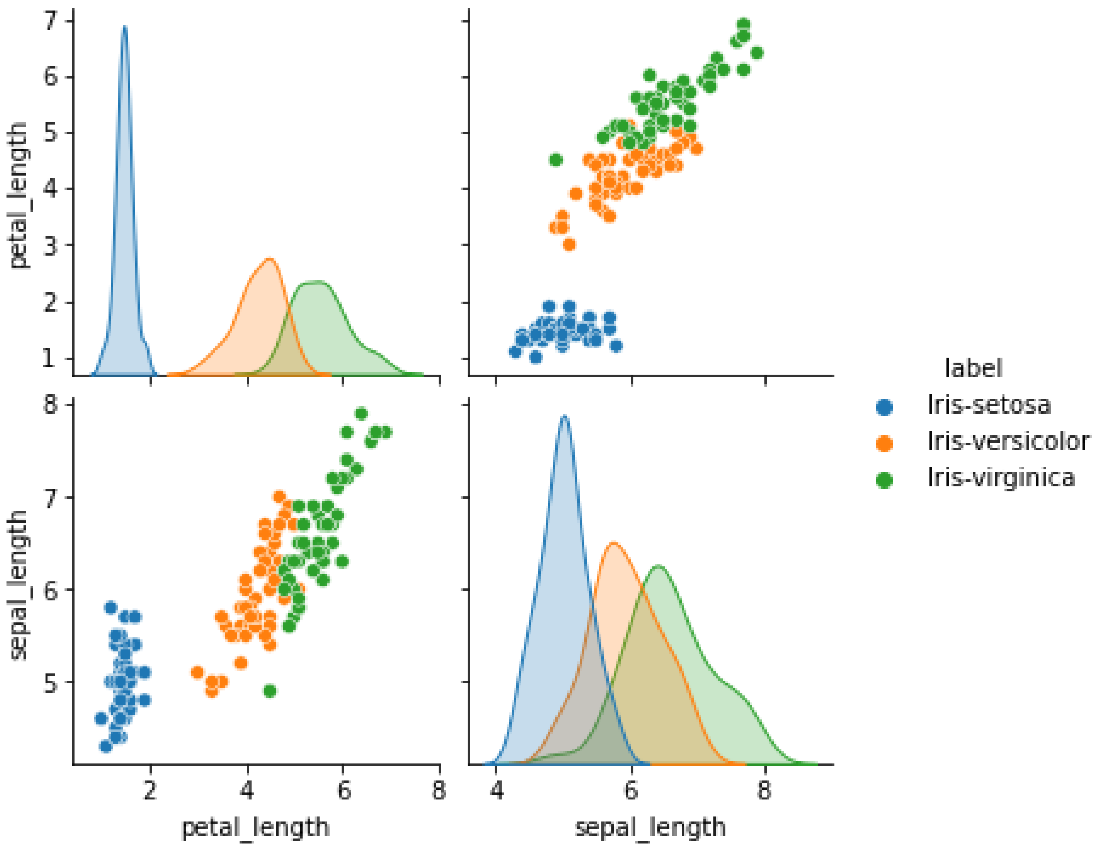
The first model we will implement is Linear Regression using Gradient Descent.

1.1.1 Getting data

In order to properly test linear regression, we first need to find a set of correlated variables, so that we use one to predict the other. Consider the following scatterplots:

[3]:
```
sb.pairplot(data[['petal_length','sepal_length','label']], hue = 'label')
```

[3]:
```
<seaborn.axisgrid.PairGrid at 0x123c08c40>
```



We observe that sepal length and petal width for Iris-versicolor and Iris-virginica are reasonably correlated, so we are going to take those two variables for those two classes and use one to regress on the other.

[4]:
```python
sub_data = data.loc[data['label'] != 'Iris-setosa', :]
y = sub_data['petal_length'].values
x = sub_data['sepal_length'].values
x = x.reshape(-1, 1)
```

1.1.2 Question 1a: Gradient descent for linear regression [40%]

As we saw in class, here we will implement the gradient descent version of linear regression. In particular, the function implemented should follow the following format:

```python
def linear_regression_gd(x,y,learning_rate = 0.00001,max_iter=10000,tol=pow(10,-5)):
```

Where 'x' is the training data feature(s), 'y' is the variable to be predicted, 'learning_rate' is the learning rate used, 'max_iter' defines the maximum number of iterations that gradient descent is allowed to run, and 'tol' is defining the tolerance for convergence (which we'll discuss next).

The return values for the above function should be (at the least) 1) 'theta' which are the regression parameters, 2) 'all_cost' which is an array where each position contains the value of the objective function $J(\theta)$ for a given iteration, 3) 'iters' which counts how many iterations did the algorithm need in order to converge to a solution.

Gradient descent is an iterative algorithm; it keeps updating the variables until a convergence criterion is met. In our case, our convergence criterion is whichever of the following two criteria happens first:

- The maximum number of iterations is met
- The relative improvement in the cost is not greater than the tolerance we have specified. For this criterion, you may use the following snippet into your code:

```
np.absolute(all_cost[it] - all_cost[it-1])/all_cost[it-1] <= tol
```

[5]:
```
#your code here

#function to compute b and m
def get_gradient(x,y,b,m,learning_rate):

    #initializing variables
    grad_b, grad_m = 0, 0
    n = float(len(x))

    #formula for calculating gradient descent, source notes
    for i in range (len(x)):

        grad_b += -(2/n) * (y[i] - ((m * x[i]) + b))
        grad_m += -(2/n) * x[i] * (y[i] - ((m * x[i]) + b))

    #updating the gradient values for b and m
    update_grad_b = b - (learning_rate * grad_b)
    updated_grad_m = m - (learning_rate * grad_m)

    #returning the new gradient values
    return [update_grad_b, updated_grad_m]

#calculateing the error loss for a given x, y, and set of parameters
def compute_cost(x, y, b, m):

    #initializing variables
    num_of_errors, total = 0,0
    n = float(len(x))

    #formula for calculating total number of errors
    for i in range (len(x)):
        num_of_errors += (y[i] - (m * x[i] + b)) **2

    total=num_of_errors/n

    #returning average of error
    return total

#function computing gradient descent version of linear regression
def linear_regression_gd(x,y,learning_rate,max_iter,tol):

    #initializing all_cost array
    all_cost = []

    #initializing variables
    b, m, iters = 0, 0, 0

    #calculating intercept, slope, all_cost array, by iterating till the result␣
 ↪is computed or till the max number of iterations is reached
    for i in range (max_iter):

        #incrementing number of iterations
        iters=iters+1

        #calling get_gradient function to compute intercept and slope
        b, m = get_gradient(x,y,b,m,learning_rate)

        #storing the value that was returned by compute_cost() in all_cost array
        all_cost.append(compute_cost(x, y, b, m))

        if (i>=1):
            #The relative improvement in the cost is not greater than the␣
 ↪tolerance we have specified
            if(np.absolute(all_cost[i] - all_cost[i-1])/all_cost[i-1] <= tol):
                break
    #returning slope, intercept,  all_cost, and iters
    return b, m, all_cost, iters


b, m, all_cost, iters = linear_regression_gd(x,y,0.00001,10000,pow(10,-5))
#Calling linear_regression_gd function which returns intercept, slope, all_cost␣
 ↪array, and iters which is the number of iterations
```

1.1.3 Question 1b: Convergence plots [10%]

After implementing gradient descent for linear regression, we would like to test that indeed our algorithm converges to a solution. In order see this, we are going to look at the value of the objective/loss function $J(\theta)$ as a function of the number of iterations, and ideally, what we would like to see is $J(\theta)$ drops as we run more iterations, and eventually it stabilizes.

As we discussed in class, the learning rate plays a big role in how fast our algorithm converges: a larger learning rate means that the algorithm is making faster strides to the solution, whereas a smaller learning rate implies slower steps. In this question we are going to test two different values for the learning rate: - 0.00001 - 0.000001 while keeping the default values for the max number of iterations and the tolerance.

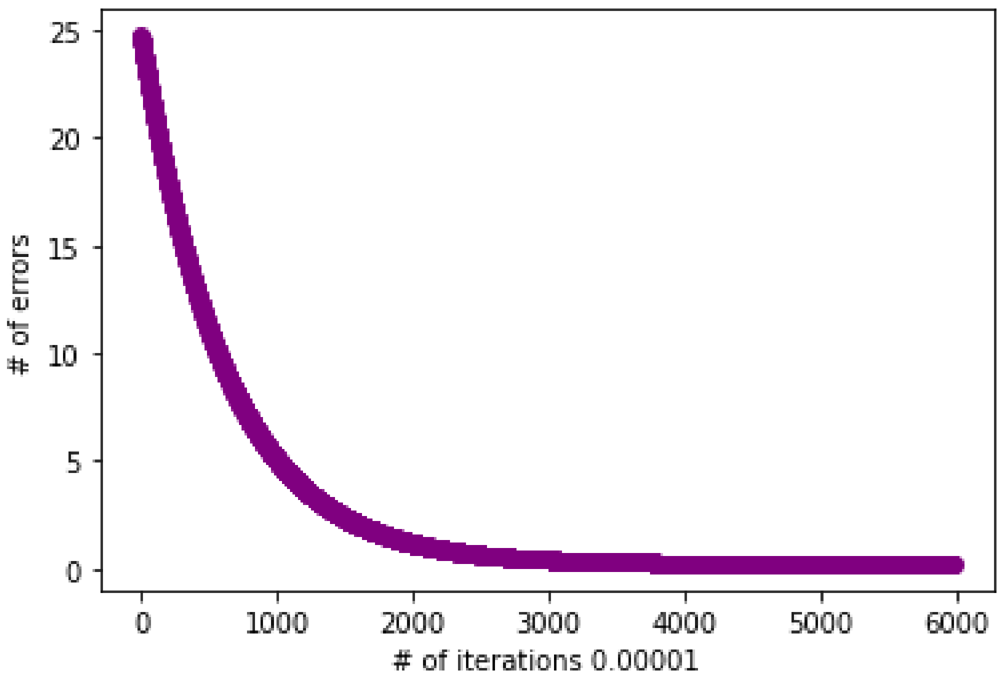
- Plot the two convergence plots (cost vs. iterations) [5%]
- What do you observe? [5%]

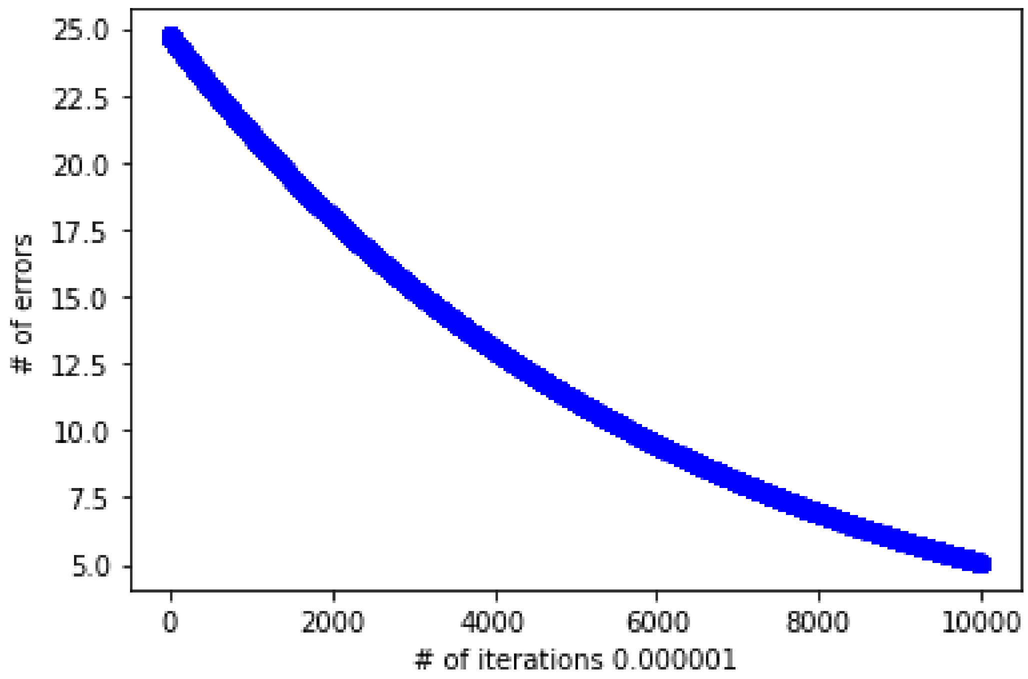
Important: Remember that as we discussed in class, in reality, when we are running gradient descent, we should be checking convergence based on the validation error (i.e., we would have to split our training set into a e.g., 70/30 training'/validation subsets, use the new training' set to calculate the gradient descent updates and evaluate the error both on the training' set and the validation set, and as soon as the validation loss stops improving, we stop training. In order to keep things simple, in this assignment we are only looking at the training loss, but as long as you have a function

```
def compute_cost(x,theta,y):
```

that calculates the loss for a given x, y, and set of parameters you have, you can always compute it on the validation portion of x and y (that are not used for the updates).

```
[6]: #your code here

     #computing values with learning rate 1: 0.00001
     b_r1, m_r1, all_cost_r1, iters_r1 = linear_regression_gd(x,y,0.
      ↪00001,10000,pow(10,-5))

     #plot for learning rate 1: 0.00001
     fig,plot1 = plt.subplots()
     #creating an array for number of iterations with equally spaced elements +1␣
      ↪(numerical sequences for iters)
     updated_iters_r1 = np.arange(0,iters_r1,1)
     plot1.scatter(updated_iters_r1, all_cost_r1, color='purple')
     #plot labels
     plt.ylabel("# of errors")
     plt.xlabel("# of iterations 0.00001")

     #computing values with learning rate 2: 0.000001
     b_r2, m_r2, all_cost_r2, iters_r2 = linear_regression_gd(x,y,0.
      ↪000001,10000,pow(10,-5))

     #plot for learning rate 2: 0.000001
     fig,plot2 = plt.subplots()
     #creating an array for number of iterations with equally spaced elements +1␣
      ↪(numerical sequences for iters)
     updated_iters_r2 = np.arange(0,iters_r2,1)
     plot2.scatter(updated_iters_r2, all_cost_r2, color='b')
     #plot labels
     plt.ylabel("# of errors")
     plt.xlabel("# of iterations 0.000001")

     plt.show()
```





Your answer here

1.2 Question 2: K-Nearest Neighbors Classifier [50%]

The K-Nearest Neighbors Classifier is one of the most popular instance-based (and in general) classification models. In this question, we will implement our own version and test in different scenarios.

1.2.1 Question 2a: Implement the K-NN Classifier [30%]

For the implementation, your function should have the format:

```
def knnclassify(test_data,training_data, training_labels, K=1):
```

where 'test_data' contains test data points, 'training_data' contains training data points, 'training_labels' holds the training labels, and 'K' is the number of neighbors.

The output of this function should be 'pred_labels' which contains the predicted label for each test data point (it should, therefore, have the same number of rows as 'test_data').

The piece of code below prepares the Iris dataset by converting the labels from strings to integers (which is quite easier to move around and do calculations with):

[7]:
```
all_vals = data[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].
 ↪values
all_labels = data['label'].values
unique_labels = np.unique(all_labels)
#change string labels to numbers
new_labels = np.zeros(len(all_labels))
for i in range(0,len(unique_labels)):
    new_labels[all_labels == unique_labels[i]] = i
all_labels = new_labels
```

[8]:
```
#your code here

#function computes euclidean distance and returns it back to nearest neighbor␣
 ↪classifeir
def get_euclidean(i,j,test_data,training_data):

    euclidean_value = ((test_data[i][0]-training_data[j][0])**2 +
                      (test_data[i][1]-training_data[j][1])**2 +
                      (test_data[i][2]-training_data[j][2])**2 +
                      (test_data[i][3]-training_data[j][3])**2)**(1/2)

    return euclidean_value

#this function gets label for the test data when k is more than one
def get_label(i,K,nearest_label,label):

    virginica = 0
    versicolor = 0
    setosa = 0
    ##labeling the test data after computing euclidean distance

    for j in range(K):
        if (nearest_label[i][j] == 1.):
            versicolor +=1
        elif(nearest_label[i][j] == 0.):
            setosa +=1
        else:
            virginica +=1

        #here we set the default label to one of the three,
        #here we selected setosa and then we iterate untill the correct label is␣
↪found

        label[i] = 0.
        if (versicolor > setosa):
            label[i] = 1.
            if(virginica > versicolor):
                label[i] = 2.
        if (virginica > setosa):
            label[i] = 2.
            if (versicolor > virginica):
                label[i] = 1.
    return label

#function for nearest neighbor classifier
def knnclassify(test_data,training_data, training_labels, K):

    ##initializing
    width, height = len(test_data), len(training_data)

    if(K!=1):
        ##initializing
        nearest, nearest_label = np.zeros((width, K)), np.zeros((width, K))
        label = [0 for i in range(width)]

        for i in range (width):

            for j in range(height):
                #getting euclidean distance
                euclidean_value=get_euclidean(i,j,test_data,training_data)
                if (j < K):
                    nearest_label[i] = training_labels[j]
                    nearest[i][j] = euclidean_value
                else:
                    for n in range (K):
                        if (euclidean_value < nearest[i][n]):
                            nearest_label[i][n] = training_labels[j]
                            nearest[i][n] = euclidean_value
                            break
            #after computing euclidean distance we label the object of the test␣
↪data
            label=get_label(i,K,nearest_label,label)

        return label

    ##when k=1
    else:
        ##initializing
        nearest, nearest_label = [0 for i in range(width)], [0 for i in␣
↪range(width)]

        for i in range (width):

            for j in range(height):
                #getting euclidean distance
                euclidean_value=get_euclidean(i,j,test_data,training_data)

                if (j!=0):
                    if (euclidean_value <  nearest[i]):
                        nearest_label[i] = training_labels[j]
                        nearest[i] = euclidean_value

                else:
                    nearest_label[i] = training_labels[j]
                    nearest[i] = euclidean_value

        return nearest_label
```

1.2.2 Question 2b: Measuring performance [10%]

In this question you will have to evaluate the average performance of your classifier for different values of K. In particular, K will range in $\{1, \cdots, 8\}$. We are going to measure the performance using classification accuracy. For computing the accuracy, you may use

`accuracy = sum(test_labels == pred_labels)/len(test_labels)`

where 'test_labels' are the actual class labels and 'pred_labels' are the predicted labels

In order to get a proper estimate for the accuracy for every K, we need to run multiple iterations where for each iteration we get a different randomized split of our data into train and test. In this question, we are going to run 100 iterations for every K, and for every random splitting, you may use:

```
    (training_data, test_data, training_labels, test_labels) = train_test_split(all_vals, all_la
```

where the train/test ratio is 70/30.

After computing the accuracy for every K for every iteration, you will have 100 accuracies per K. The best way to store those accuracies is in a matrix that has as many rows as values for K and 100 columns, each one for each iteration.

Compute average accuracy as a function of K. Because we have a randomized process, we also need to compute how certain/uncertain our estimation for the accuracy per K is. For that reason, we also need to compute the standard deviation of the accuracy for every K. Having computed both average accuracy and standard deviation, make a figure that shows the average accuracy as a function of K with each point of the figure being surrounded by an error-bar encoding the standard deviation. You may find

`plt.errorbar()`

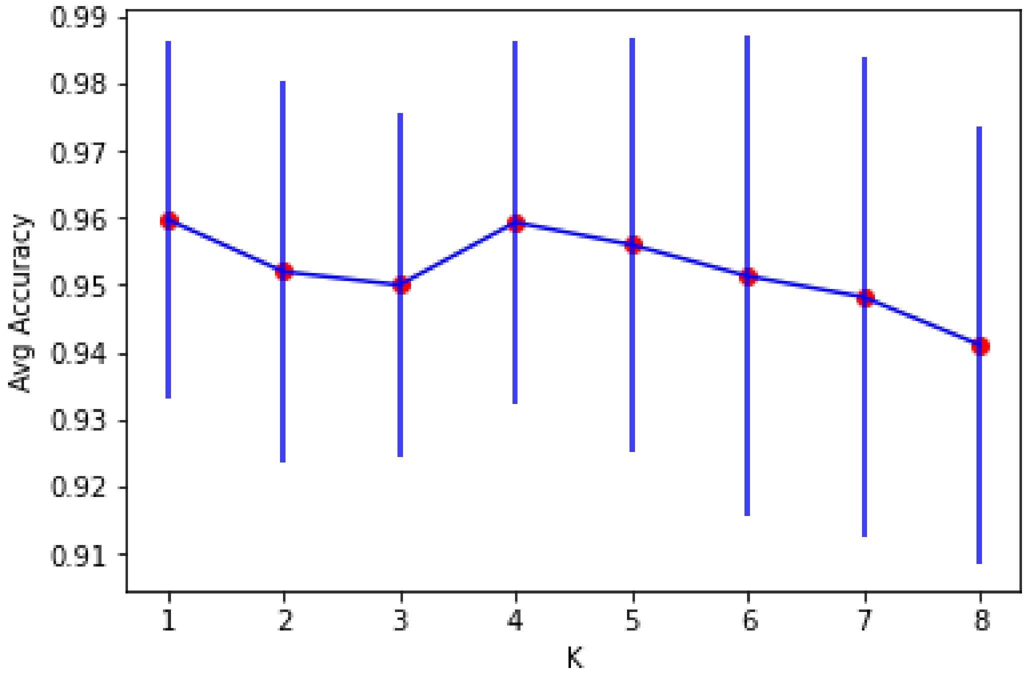
useful for this plot.

[9]:
```
#your code here
#running for k in range 1,8

#initializing data_accuracy to zero
data_accuracy = np.zeros((8, 100))

#iterationg through k=1 to k=8
for i in range (1,9):

    #going through 100 iterations for each K
    for j in range(100):

        #randomize split of our data
        (training_data, test_data, training_labels, test_labels) =␣
 ↪train_test_split(all_vals, all_labels, test_size=0.3)

        #getting the predicted label from knn classifier function
        pred_labels = knnclassify(test_data, training_data, training_labels, i)

        #computing accuracy and storing the value
        accuracy = sum(test_labels == pred_labels)/len(test_labels)
        data_accuracy[i-1][j] = accuracy

##print(data_accuracy)

#computing average accuracy for each k 1,8
average_accuracy = [0 for x in range(8)]
for i in range(8):
    total = 0
    for j in range(100):
        total += data_accuracy[i][j]
    average_accuracy[i] = total/100
```

```
#computing standard deviation for each k 1,8
stdev = [0 for x in range(8)]
for i in range(8):
    deviation_total = 0
    for j in range(100):
        deviation_total += (average_accuracy[i] - data_accuracy[i][j])**2
    deviation = deviation_total/99   #100-1
    stdev[i] = (deviation)**(1/2)
print("devvv", stdev)
##print("avg acc:", average_accuracy, "stdev:", stdev)
```

```
devvv [0.02674418213769568, 0.028552823828775362, 0.02553822669198847,
0.026996648953990204, 0.030782293120750297, 0.03569207796528503,
0.03560252229078045, 0.03257651884023365]
```

[10]:
```
fig,kplot = plt.subplots()
#array with k: 1,8
all_k = [x for x in range(1,9)]
#plot of k vs avg accuracy
kplot.scatter(all_k , average_accuracy,color='r')

#using plt.errorbar() method to show error-bar around the points that encode the␣
 ↪standard deviation
kplot.errorbar(all_k , average_accuracy, yerr=stdev, color='b')
#plot labels
plt.ylabel("Avg Accuracy")
plt.xlabel("K")
plt.show()
```



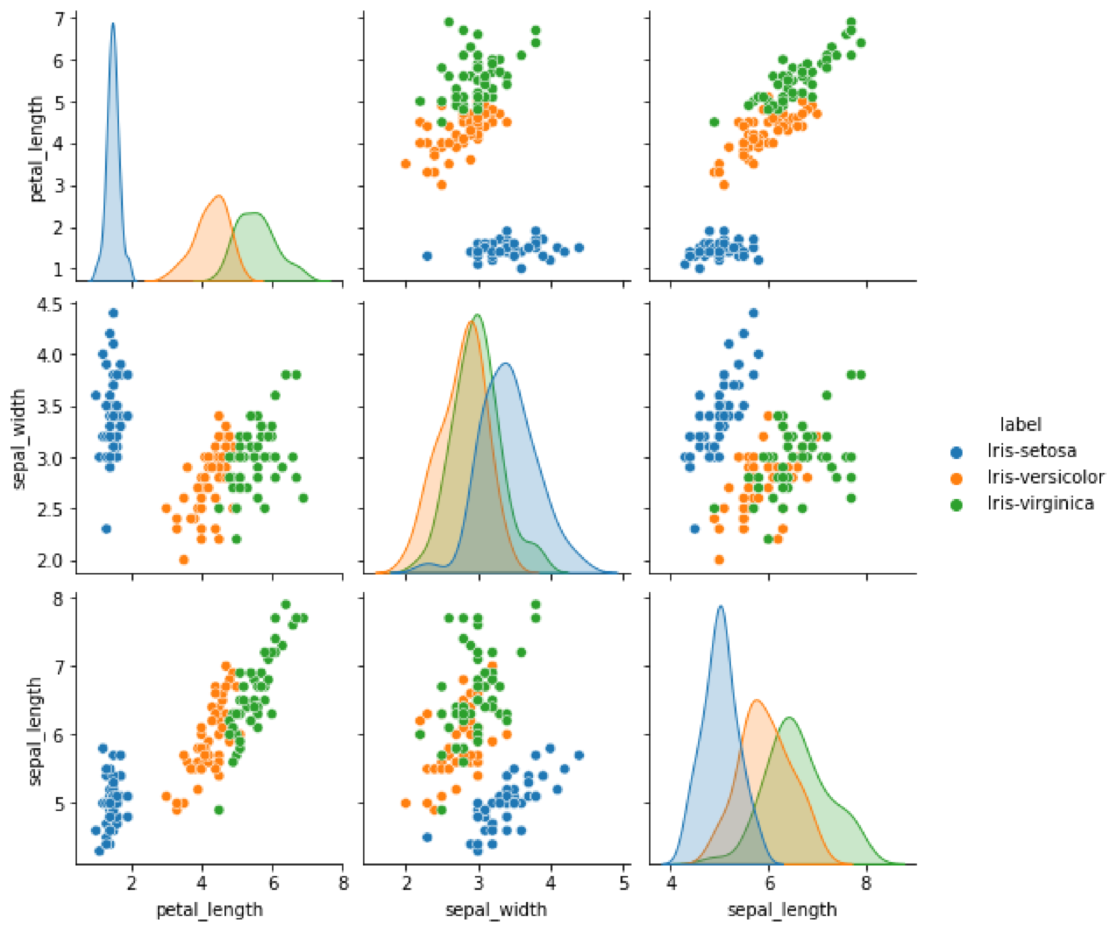
1.2.3 Question 2c: Feature selection [10%]

We have extensively discussed in class the fact that a good or bad set of features can make or break our model! Here we will see what happens when we operate on a subset of the features, and in particular in - a subset that has good separability of classes - a subset that has poor separability of classes

Recall from Assignment 1 where we did the scatterplots of the Iris dataset that a pair of features with high visual separability is (petal length, sepal width), whereas a set that confuses at least two classes is (sepal length, sepal width).

```
[11]: sb.pairplot(data[['petal_length','sepal_width','sepal_length','label']], hue =␣
      ↪'label')
```

```
[11]: <seaborn.axisgrid.PairGrid at 0x12652ebe0>
```



Apply K-NN classification with K = 1 on two datasets (using the same train/test split for both datasets, and the same method you used to split as above) and measure the classification accuracy for: - Only (petal length, sepal width) [2.5%] - Only (sepal length, sepal width) [2.5%]

What do you observe regarding the classification accuracy? [5%]

[12]:
```
#your code here
#minor change to euclidean formula to compare only two feautres k=1
#function computes euclidean distance and returns it back to nearest neighbor␣
 ↪classifeir
def get_euclidean(i,j,test_data,training_data):

    euclidean_value = ((test_data[i][0]-training_data[j][0])**2 +
                       (test_data[i][1]-training_data[j][1])**2)**(1/2)

    return euclidean_value

#this function gets label for the test data when k is more than one
def get_label(i,K,nearest_label,label):
    virginica = 0
    versicolor = 0
    setosa = 0

    ##labeling the test data after computing euclidean distance

    for j in range(K):
        if (nearest_label[i][j] == 1.):
            versicolor +=1
        elif(nearest_label[i][j] == 0.):
            setosa +=1
        else:
            virginica +=1

        #here we set the default label to one of the three,
        #here we selected setosa and then we iterate untill the correct label is␣
 ↪found

        label[i] = 0.
        if (versicolor > setosa):
            label[i] = 1.
            if(virginica > versicolor):
                label[i] = 2.
        if (virginica > setosa):
            label[i] = 2.
            if (versicolor > virginica):
                label[i] = 1.
    return label

#function for nearest neighbor classifier
def knnclassify2c(test_data,training_data, training_labels, K):

    ##initializing
    width, height = len(test_data), len(training_data)

    if(K!=1):
        ##initializing
        nearest, nearest_label = np.zeros((width, K)), np.zeros((width, K))
        label = [0 for i in range(width)]

        for i in range (width):

            for j in range(height):
                #getting euclidean distance
                euclidean_value=get_euclidean(i,j,test_data,training_data)
                if (j < K):
                    nearest_label[i] = training_labels[j]
                    nearest[i][j] = euclidean_value
                else:
                    for n in range (K):
                        if (euclidean_value < nearest[i][n]):
                            nearest_label[i][n] = training_labels[j]
                            nearest[i][n] = euclidean_value
                            break
            #after computing euclidean distance we label the object of the test␣
↪data
            label=get_label(i,K,nearest_label,label)

        return label

    ##when k=1
    else:
        ##initializing
        nearest, nearest_label = [0 for i in range(width)], [0 for i in␣
↪range(width)]

        for i in range (width):

            for j in range(height):
                #getting euclidean distance
                euclidean_value=get_euclidean(i,j,test_data,training_data)

                if (j!=0):
                    if (euclidean_value <  nearest[i]):
                        nearest_label[i] = training_labels[j]
                        nearest[i] = euclidean_value

                else:
                    nearest_label[i] = training_labels[j]
                    nearest[i] = euclidean_value

        return nearest_label
```

[13]:
```
#your code here

##petal_length vs sepal_width_accuracy
petal_vs_sepal = data[['sepal_width', 'petal_length']].values
#initializing to zero
petal_vs_sepal_accuracy = 0
#iterating 100 times
for i in range(0,100):
    #for k=1
    #randomize split of our data
    (training_data, test_data, training_labels, test_labels) =␣
 ↪train_test_split(petal_vs_sepal, all_labels, test_size=0.3)
    pred_labels = knnclassify2c(test_data, training_data, training_labels, 1)
    accuracy = sum(test_labels == pred_labels)/len(test_labels)
    petal_vs_sepal_accuracy += accuracy

petal_vs_sepal_average_accuracy = petal_vs_sepal_accuracy / 100

##sepal_vs_sepal_average_accuracy
sepal_vs_sepal = data[['sepal_length', 'sepal_width']].values
#initializing to zero
sepal_vs_sepal_accuracy = 0
#iterating 100 times
for i in range(0,100):

    #for k=1
    #randomize split of our data
    (training_data, test_data, training_labels, test_labels) =␣
 ↪train_test_split(sepal_vs_sepal, all_labels, test_size=0.3)
    pred_labels = knnclassify2c(test_data, training_data, training_labels, 1)
    accuracy = sum(test_labels == pred_labels)/len(test_labels)
    sepal_vs_sepal_accuracy += accuracy

sepal_vs_sepal_average_accuracy = sepal_vs_sepal_accuracy / 100

print(" avg accuracy for petal_length vs. sepal_width: ",␣
 ↪petal_vs_sepal_average_accuracy,"\n",
      "avg accuracy for sepal_length vs. sepal_width: ",␣
 ↪sepal_vs_sepal_average_accuracy)
```

```
avg accuracy for petal_length vs. sepal_width:  0.9153333333333326
avg accuracy for sepal_length vs. sepal_width:  0.7126666666666662
```