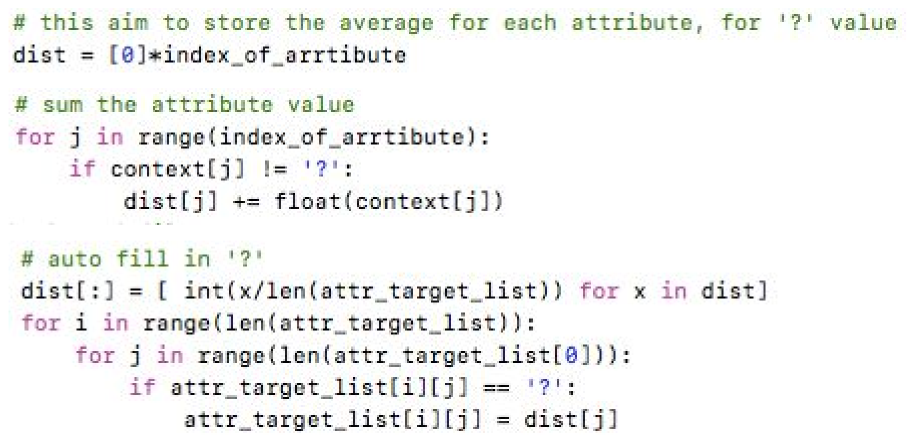
Name: Chenyang Yu   SUID: 862052273

Question 0. Get data and handle missing value

As we can see, there are totally 16 missing value in *breast-cancer-wisconsin* data set. I use “auto filling” to replace missing value (value = ‘?’) with the mean value of each attribute. Although the mean value might not reflect the properties of the data point we modify, it is a good method to keep the whole data point without affect the feature we modify, instead of ignoring the whole data point.

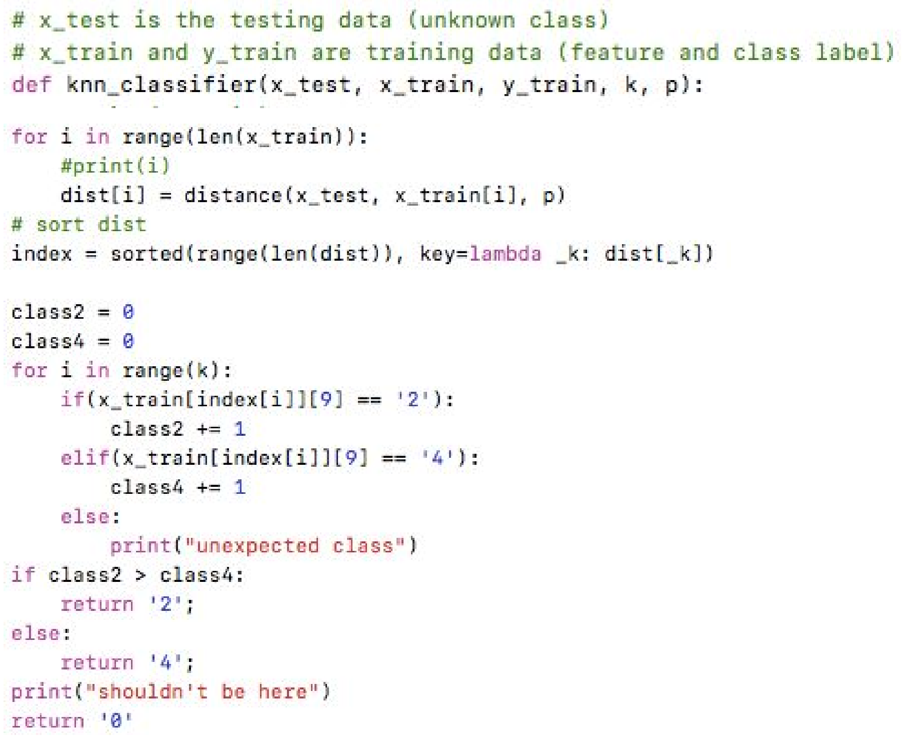


```
# this aim to store the average for each attribute, for '?' value
dist = [0]*index_of_arrtibute

# sum the attribute value
for j in range(index_of_arrtibute):
    if context[j] != '?':
        dist[j] += float(context[j])

# auto fill in '?'
dist[:] = [ int(x/len(attr_target_list)) for x in dist]
for i in range(len(attr_target_list)):
    for j in range(len(attr_target_list[0])):
        if attr_target_list[i][j] == '?':
            attr_target_list[i][j] = dist[j]
```

Question 1. K-NN classifier

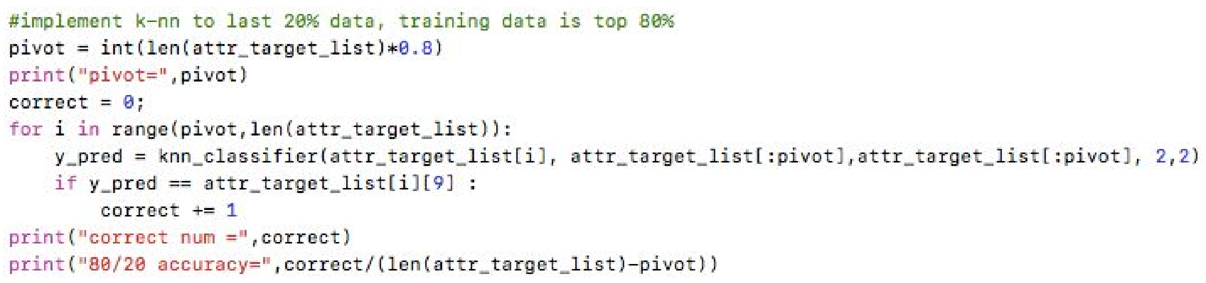
This is lazy learner, which means we don’t train a model to predict a testing data. Instead, when a testing data is given, we calculate the distance between this testing data and all training data. Sort the training data depending on the distance and choose the top k data points (the closest k data points). Use these k data points to decide the label of testing data. The idea is: Fnd the k-closest data points, the more you close a specific class, the more possible you belongs to this class. We use knn_classifier(x_test, x_train, y_train, k, p), where x_test is the testing data, x_train and y_train is the training data set.

The context is loop through training data the calculate distance one by one. The distance function is Lp-norm, the parameter p is assigned by input parameter



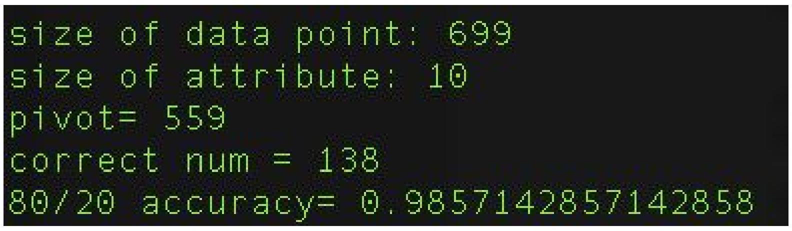
```
# x_test is the testing data (unknown class)
# x_train and y_train are training data (feature and class label)
def knn_classifier(x_test, x_train, y_train, k, p):

for i in range(len(x_train)):
    #print(i)
    dist[i] = distance(x_test, x_train[i], p)
# sort dist
index = sorted(range(len(dist)), key=lambda _k: dist[_k])

class2 = 0
class4 = 0
for i in range(k):
    if(x_train[index[i]][9] == '2'):
        class2 += 1
    elif(x_train[index[i]][9] == '4'):
        class4 += 1
    else:
        print("unexpected class")
if class2 > class4:
    return '2';
else:
    return '4';
print("shouldn't be here")
return '0'
```

80/20 cross validation is used for checking performance of k-NN here, further evaluation will be given in the question 3. First we choose pivot as 0.8*total_size and partition data set into 80:20



```
#implement k-nn to last 20% data, training data is top 80%
pivot = int(len(attr_target_list)*0.8)
print("pivot=",pivot)
correct = 0;
for i in range(pivot,len(attr_target_list)):
    y_pred = knn_classifier(attr_target_list[i], attr_target_list[:pivot],attr_target_list[:pivot], 2,2)
    if y_pred == attr_target_list[i][9] :
        correct += 1
print("correct num =",correct)
print("80/20 accuracy=",correct/(len(attr_target_list)-pivot))
```

Result:

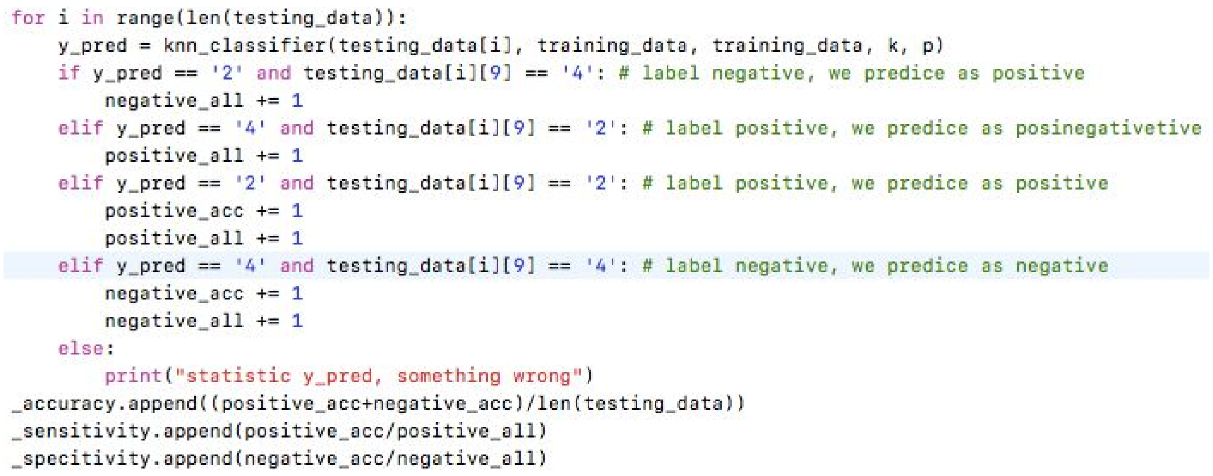


```
size of data point: 699
size of attribute: 10
pivot= 559
correct num = 138
80/20 accuracy= 0.9857142857142858
```

The accuracy is 98.6%. At first glance k-NN classifier seems to be suitable for this data set.

Question 2.

We use 10-fold cross-validation to measure the accuracy of k-NN classifier. Partition the original data set into 10 fold, use one proportion as testing data and the rest ninie proportions as training data. Three metric are used to measure the performance:

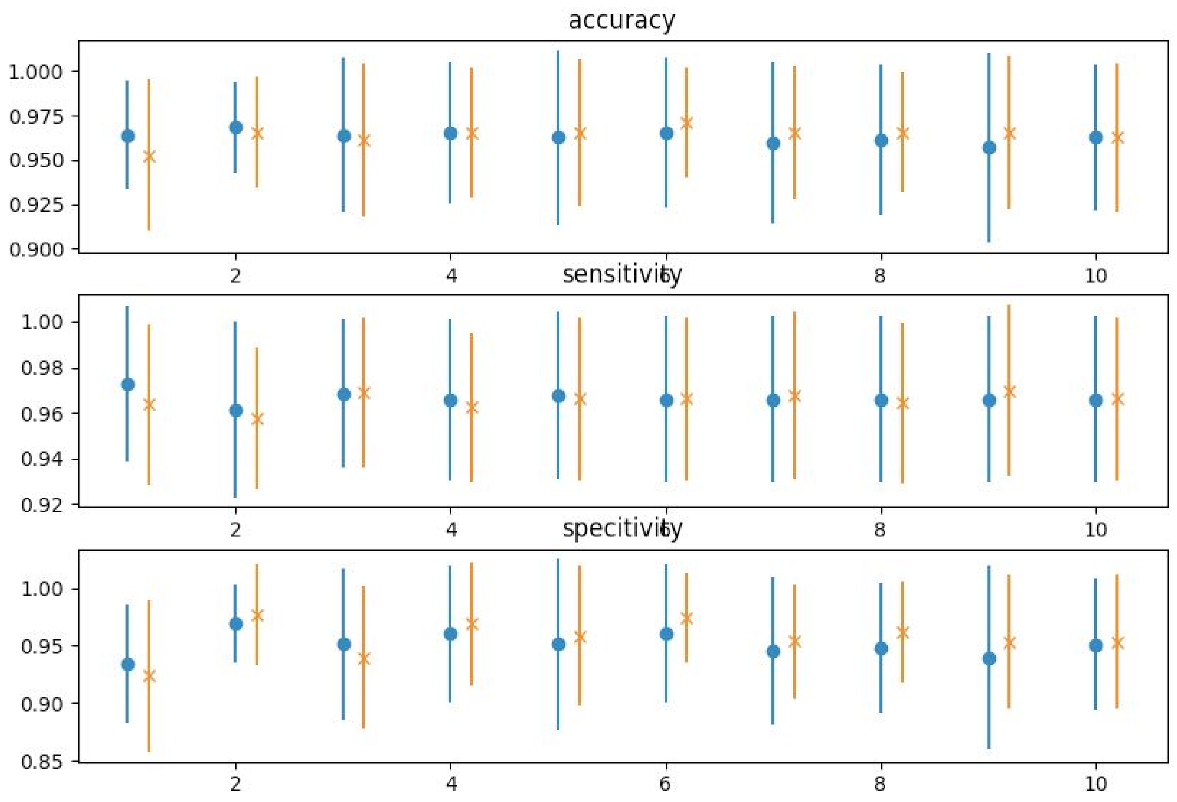


```
for i in range(len(testing_data)):
    y_pred = knn_classifier(testing_data[i], training_data, training_data, k, p)
    if y_pred == '2' and testing_data[i][9] == '4': # label negative, we predice as positive
        negative_all += 1
    elif y_pred == '4' and testing_data[i][9] == '2': # label positive, we predice as posinegativetive
        positive_all += 1
    elif y_pred == '2' and testing_data[i][9] == '2': # label positive, we predice as positive
        positive_acc += 1
        positive_all += 1
    elif y_pred == '4' and testing_data[i][9] == '4': # label negative, we predice as negative
        negative_acc += 1
        negative_all += 1
    else:
        print("statistic y_pred, something wrong")
_accuracy.append((positive_acc+negative_acc)/len(testing_data))
_sensitivity.append(positive_acc/positive_all)
_specitivity.append(negative_acc/negative_all)
```

(1) Accuracy: Among all testing data, how many samples we can predict correctly (label ‘2’ and we predict as ‘2’, or label ‘4’ and we predict as ‘4’)
(2) Sensitivity: Among all positive testing data (class ‘2’ as positive class), how many samples we can predict correctly
(3) Specitivity: Among all positive testing data (class ‘4’ as negative class), how many samples we can predict correctly

We repeat 10-fold and use 10 different fold as training set. Report the final result after 10 times average. The comparison between different k-value (for k nearest neighbor) is also record.

Result:



The blue error bar is p=1 and the orange error bar is p=2. All three evaluations are pretty high (over 95%). Standard deviant is little larger when k-value is larger in most case. But overall, there is no obvious pattern in our result.

Accuracy:

	k=1	2	3	4	5	6	7	8	9	10
p=1	0.96 (0.03)	0.97 (0.03)	0.96 (0.04)	0.97 (0.04)	0.96 (0.05)	0.97 (0.04)	0.96 (0.05)	0.96 (0.04)	0.96 (0.05)	0.96 (0.04)
p=2	0.95 (0.04)	0.97 (0.03)	0.96 (0.04)	0.97 (0.04)	0.97 (0.04)	0.97 (0.03)	0.97 (0.04)	0.97 (0.03)	0.97 (0.04)	0.96 (0.04)

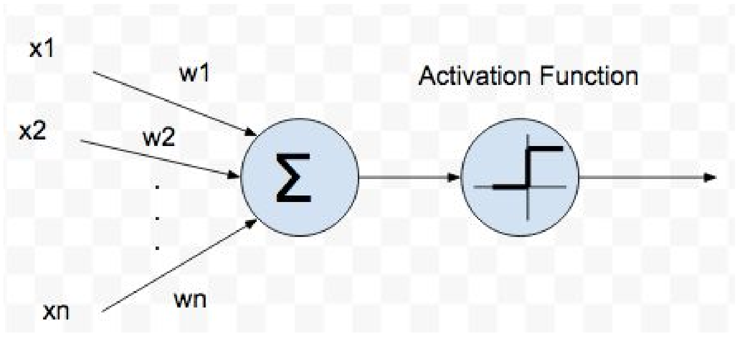
Sensitivity:

	k=1	2	3	4	5	6	7	8	9	10
p=1	0.97 (0.03)	0.96 (0.04)	0.97 (0.03)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)
p=2	0.96 (0.04)	0.96 (0.03)	0.97 (0.03)	0.96 (0.03)	0.97 (0.04)	0.97 (0.04)	0.97 (0.04)	0.96 (0.04)	0.97 (0.04)	0.97 (0.04)

Specitivity:

	k=1	2	3	4	5	6	7	8	9	10
p=1	0.93 (0.05)	0.97 (0.03)	0.95 (0.07)	0.96 (0.06)	0.95 (0.07)	0.96 (0.06)	0.95 (0.06)	0.95 (0.06)	0.94 (0.08)	0.96 (0.06)
p=2	0.92 (0.07)	0.98 (0.04)	0.94 (0.06)	0.97 (0.05)	0.96 (0.06)	0.97 (0.04)	0.95 (0.05)	0.96 (0.04)	0.95 (0.06)	0.95 (0.06)

Question 3. Implement a perceptron



As figure showing above, the perceptron predict out by $\Sigma w_i x_i$, where i in 1:n, n is the number of attribute. The activation function is sign function, return positive if $\Sigma w_i x_i \geq 0$, else return negative (we define '4' as positive result here)

```
# input_x is sinle data point
def single_perceptron(input_x, w):
    sigma = 0
    for i in range(len(input_x)-1):
        sigma += float(input_x[i])*w[i]
    # we put the bias at the end of w vector
    sigma += w[len(w)-1]
    if sigma >= 0:
        return '4'
    else:
        return '2'
    print("shouldn't be here")
    return '0'
```

When training our model, we repeat modify the weight until it can perfectly predict the data point (or stop training if we think it cannot go any better). Notice that we put the bias (w_0) in w[len(w)-1], this bias will not be modified in the process of training. For each data point, we modify the weight as:

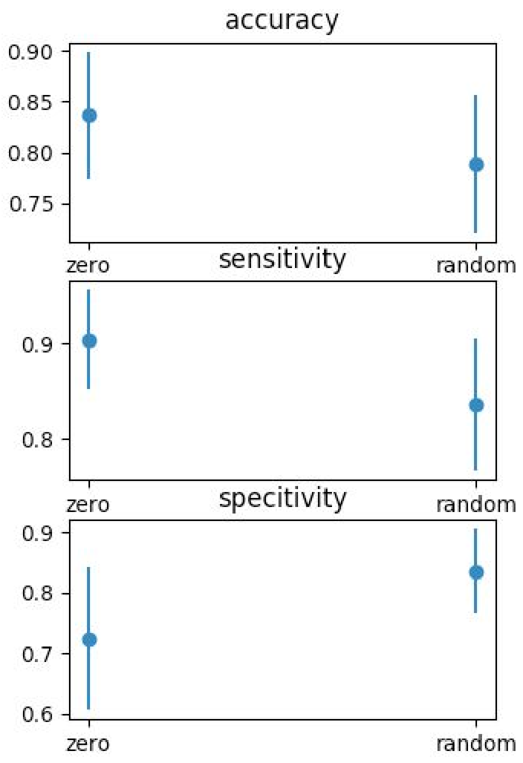
$w = w + \alpha \times \Delta y \times x$, where Δy =y_label - y_predict, α is learning rate

It is convenient for us to calculate Δy since there is only two possible outcome in sign function (our activation function). When the label is positive and we predict it as negative, the Δy is +1, vice versa. When the label is the same as we predict, Δy is zero and no need to modify weight. We set learning as 0.1 here

```
# go through every data point and adjust the weight
for i in range(len(input_x)):
    y_pred = single_perceptron(input_x[i], w)
    if y_pred == '4' and output_y[i][9] == '2': # y-y' < 0
        error = -1
        total_error += 1
    #tmp_error += -1
    elif y_pred == '2' and output_y[i][9] == '4': # y-y' > 0
        error = 1
        total_error += 1
    #tmp_error += 1
    elif y_pred == '2' and output_y[i][9] == '2': # y-y' = 0
        error = 0
    elif y_pred == '4' and output_y[i][9] == '4': # y-y' = 0
        error = 0
    else:
        print("in perceptron, error calculate unexpected")
    for j in range(len(w)-1):
        w[j] += 0.1*error*float(input_x[i][j])
# next while loop if there is still error
```

In order to get the better model (or get the model faster), two different approach is used here. Initialize weight with all zero, or initialize weight with random number. Same with question 2, we use 10-fold to evaluate the accuracy, sensitivity, and specitivity for both approach in 10 different fold and average them.

Result:



The average accuracy is 83.7%, which is lower than k-NN classifier. There is no obvious relation between initializing weight with all zero, or initializing weight with random number (zero initialization is better in accuracy and sensitivity, but worse in specitivity)

	Accuray	Sensitivity	Specitivity
Zero init	0.837(0.062)	0.905(0.052)	0.725(0.118)
Random init	0.789(0.068)	0.835(0.0695)	0.7307(0.1339)