Assignment2:Non-parametric Classifiers

SCHOOL OF ADVANCED TECHNOLOGY

INT410: Advanced Pattern Recognition

Mengfan Li

ID number： 2032048

2020 Fall

Abstract

The main purpose of this experiment is to use Python language and related libraries to realize two none-parameter estimations: Parzen Windows and KNN, and to find out the most appropriate H and K by using the five-fold cross validation, so as to improve the accuracy of classification.

Declaration

Content

1 Objectives

Implement the Parzen window and KNN algorithm

In this experiment, we will use the publicly dataset to verify our algorithm. Download the UCI Iris dataset: https://archive.ics.uci.edu/ml/datasets/Iris

Warnings: For KNN and Parzen algorithms, there is no training stag

2 Materials

Pycharm

3 Methods, result and discussion

3.1 Estimation of Classification Methods

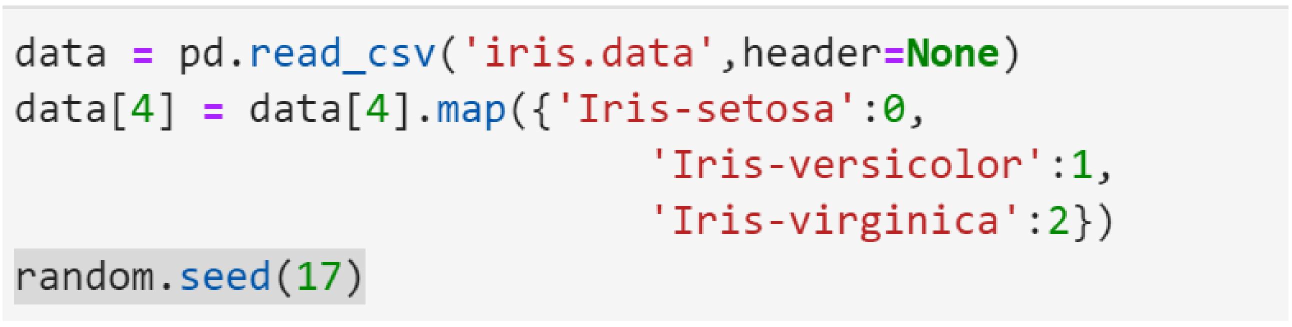
3.1.1

Read the Iris dataset into a list and shuffle it with the random.shuffle method. Hint: fix the random seed (e.g. random.seed(17)) before calling random.shuffle

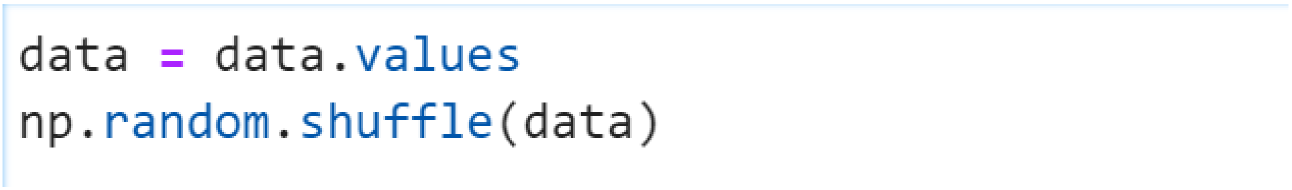
Paezrn Windows:

In Parzen's program, I used random.shuffle to let the data random and set seed = 17

Code:



```
data = pd.read_csv('iris.data',header=None)
data[4] = data[4].map({'Iris-setosa':0,
                          'Iris-versicolor':1,
                          'Iris-virginica':2})
random.seed(17)
```

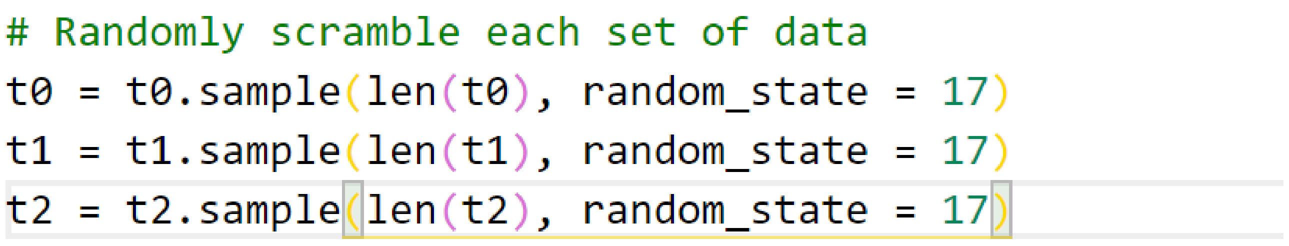


```
data = data.values
np.random.shuffle(data)
```

KNN

In the KNN experiment, I chose to use SAMPLE to shuffle the data, because in KNN, I read the data in the form of list, so it is more convenient to use sample. The comparison between Random. Shuffle and SAMPLE will be in the fourth chapter.

Code:



```
# Randomly scramble each set of data
t0 = t0.sample(len(t0), random_state = 17)
t1 = t1.sample(len(t1), random_state = 17)
t2 = t2.sample(len(t2), random_state = 17)
```

3.1.2

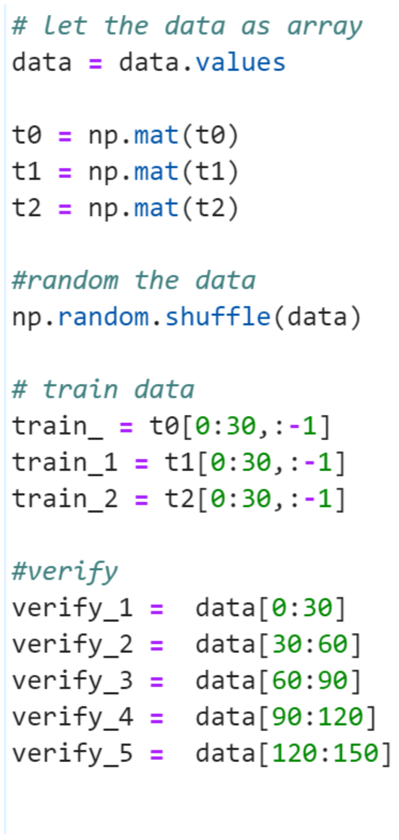
Split the dataset as five parts to do cross-fold validation: Each of 5 subsets was used as test set and the remaining data was used for training. The 5 subsets were used for testing rotationally for evaluating the classification accuracy.

In this experiment, Instead of using sklearn's built-in cross-validation function, I chose to manually group and cross-validate. First, I divided the data into 5 points. Then, in each validation, I selected one part as the test set and the remaining four parts as the training set. Then, I averaged the results of the five cross-validation tests to obtain the experimental results.

Code:

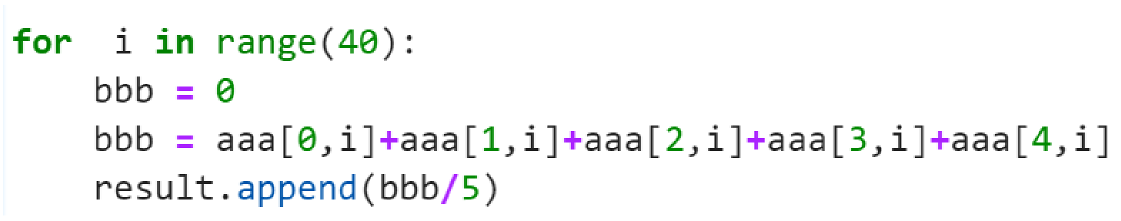
Parzen:

Manual data grouping



```
# Let the data as array
data = data.values

t0 = np.mat(t0)
t1 = np.mat(t1)
t2 = np.mat(t2)

#random the data
np.random.shuffle(data)

# train data
train_ = t0[0:30,:-1]
train_1 = t1[0:30,:-1]
train_2 = t2[0:30,:-1]

#verify
verify_1 =  data[0:30]
verify_2 =  data[30:60]
verify_3 =  data[60:90]
verify_4 =  data[90:120]
verify_5 =  data[120:150]
```

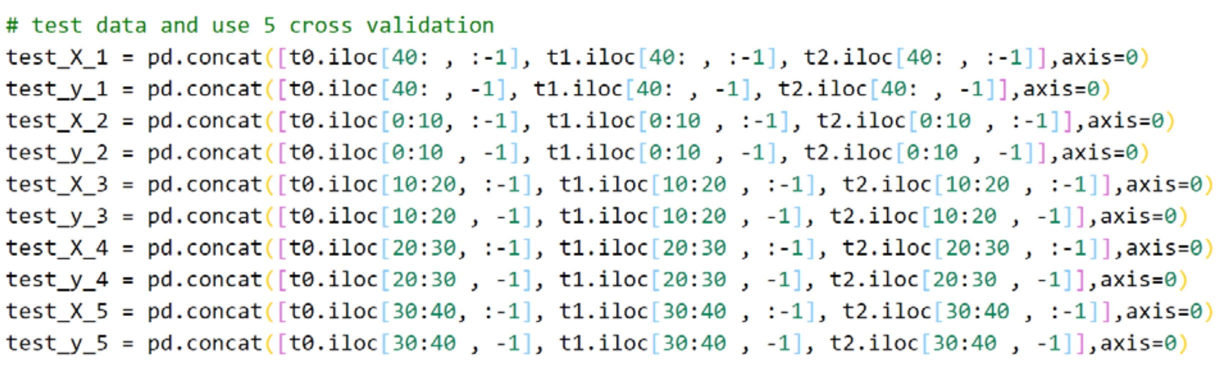
Go five times on average



```
for  i in range(40):
    bbb = 0
    bbb = aaa[0,i]+aaa[1,i]+aaa[2,i]+aaa[3,i]+aaa[4,i]
    result.append(bbb/5)
```

KNN

Manual data grouping



```
# test data and use 5 cross validation
test_X_1 = pd.concat([t0.iloc[40: , :-1], t1.iloc[40: , :-1], t2.iloc[40: , :-1]],axis=0)
test_y_1 = pd.concat([t0.iloc[40: , -1], t1.iloc[40: , -1], t2.iloc[40: , -1]],axis=0)
test_X_2 = pd.concat([t0.iloc[0:10, :-1], t1.iloc[0:10 , :-1], t2.iloc[0:10 , :-1]],axis=0)
test_y_2 = pd.concat([t0.iloc[0:10 , -1], t1.iloc[0:10 , -1], t2.iloc[0:10 , -1]],axis=0)
test_X_3 = pd.concat([t0.iloc[10:20, :-1], t1.iloc[10:20 , :-1], t2.iloc[10:20 , :-1]],axis=0)
test_y_3 = pd.concat([t0.iloc[10:20 , -1], t1.iloc[10:20 , -1], t2.iloc[10:20 , -1]],axis=0)
test_X_4 = pd.concat([t0.iloc[20:30, :-1], t1.iloc[20:30 , :-1], t2.iloc[20:30 , :-1]],axis=0)
test_y_4 = pd.concat([t0.iloc[20:30 , -1], t1.iloc[20:30 , -1], t2.iloc[20:30 , -1]],axis=0)
test_X_5 = pd.concat([t0.iloc[30:40, :-1], t1.iloc[30:40 , :-1], t2.iloc[30:40 , :-1]],axis=0)
test_y_5 = pd.concat([t0.iloc[30:40 , -1], t1.iloc[30:40 , -1], t2.iloc[30:40 , -1]],axis=0)
```

Go five times on average

```
#Computational accuracy
r_1 = np.sum(result_1 == test_y_1) / len(result_1)
r_2 = np.sum(result_2 == test_y_2) / len(result_2)
r_3 = np.sum(result_3 == test_y_3) / len(result_3)
r_4 = np.sum(result_4 == test_y_4) / len(result_4)
r_5 = np.sum(result_5 == test_y_5) / len(result_5)

avg = (r_1 + r_2 + r_3 + r_4 + r_5) / 5
result_re.append(avg)
```

3.2 Parzen Window Method

3.2.1

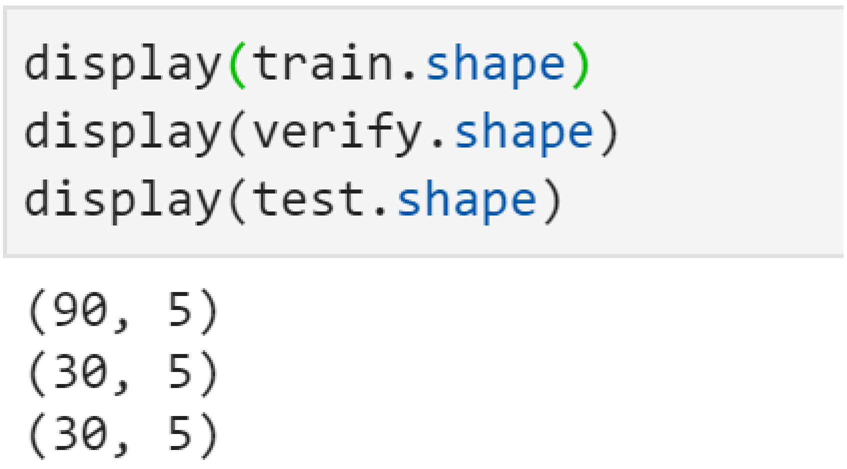
Separate the training dataset into two groups by their labels

The data is divided into two groups, that is, one part is to self-verify and then find out the best h value, the other part is to use the best H value to test. Here, the data is divided into 90:30:30, 90 is train, 30 is verify, and 30 is test

Code:

```
data = pd.read_csv('iris.data',header=None)
data[4] = data[4].map({'Iris-setosa':0,
                       'Iris-versicolor':1,
                       'Iris-virginica':2})

random.seed(17)
t0 = data[data[4] == 0]
t1 = data[data[4] == 1]
t2 = data[data[4] == 2]

data = data.values
train = data[0:90]
verify = data[90:120]
test = data[120:150]
```

Result:



```
display(train.shape)
display(verify.shape)
display(test.shape)
```

```
(90, 5)
(30, 5)
(30, 5)
```

3.2.2

Estimate the prior class probability $P(\omega_k)$

$$P(\omega_k) = \frac{n_k}{\sum_{i=1}^{K} n_i}$$

where n_k is the number of examples from the class ω_k .

Method:

In the Parzen Windows program, I manually grouped the calculated record matrix, so the proportion of each label is one third, so the prior probability here is one third for each

3.2.3

For any test example x , the conditional probability $P(x|\omega_k)$ are computed as

$$P(x|\omega_k) = \frac{1}{n_k} \sum_{x_i \in \omega_k} \frac{1}{h^d} \emptyset(\frac{x - x_i}{h})$$

where h is hyperparameter (or user-defined parameter) of model and $\emptyset(u)$ is defined as

$$\emptyset(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$$

Method:

Here, I choose the shape of Parzen Windows as cube, so d = 3. And the $\frac{1}{\sqrt{2\pi}}$ in it is a constant, so it doesn't affect the final result, so we can discard the $\frac{1}{\sqrt{2\pi}}$ here.

Let

$$\varphi\left(\frac{x-x_i}{h}\right) = Hexp[-(x-x_i)^t(x-x_i)/(2h^2)]$$

Where H is a positive integer. We have

$$P(x|\omega_k) = \frac{1}{n_k}\sum_{x_i\in\omega_k}\frac{1}{h^d}\emptyset(\frac{x-x_i}{h})$$

Therefore, the value change of H can only enlarge or shrink in the same proportion, without affecting its change trend and size relationship. Therefore, we have

$$\varphi\left(\frac{x-x_i}{h}\right) = exp[-(x-x_i)^t(x-x_i)/(2h^2)]$$

Code:

$\emptyset(u)$:

```
def get_phi(x, xi, h):
    phi = 0
    x = np.mat(x)

    xi = np.mat(xi)
    phi =  np.exp(-(x - xi) * (x - xi).T / (2 * h * h))

    return phi
```

$P(x|\omega_k)$:

```
def get_px(x, xi, h):
    px = 0
    phi = 0

    for T_t in xi:
        phi += get_phi(x, T_t, h)

    px = phi  / ( len(xi) * np.power(h, 3))

    return px
```

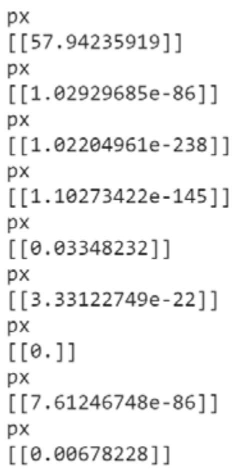
Result:

```
def get_phi(x, xi, h):
    phi = 0
    x = np.mat(x)

    xi = np.mat(xi)
    phi =  np.exp(-(x - xi) * (x - xi).T / (2 * h * h))
    print("phi")
    print(phi)
    return phi
```

```
phi
[[5.04347663e-07]]
phi
[[0.04978707]]
phi
[[0.00408677]]
phi
[[0.082085]]
phi
[[6.82560338e-08]]
phi
[[1.58321429e-23]]
phi
[[0.00012341]]
phi
[[0.00091188]]
phi
[[6.14421235e-06]]
phi
[[0.22313016]]
```

In the printout results, every three are in a group, which are the phi distance calculated by and train_0, train_1 and train_2 respectively

```
def get_px(x, xi, h):
    px = 0
    phi = 0

    for T_t in xi:
        phi += get_phi(x, T_t, h)

    px = phi  / ( len(xi) * np.power(h, 3))
    print("px")
    print(px)
    return px
```



```
px
[[57.94235919]]
px
[[1.02929685e-86]]
px
[[1.02204961e-238]]
px
[[1.10273422e-145]]
px
[[0.03348232]]
px
[[3.33122749e-22]]
px
[[0.]]
px
[[7.61246748e-86]]
px
[[0.00678228]]
```

In the printout results, every three are in a group, which are the px distance calculated by and train_0, train_1 and train_2 respectively

3.2.4

According to the Bayesian rule, we know that

$$P(x|\omega_k) \propto P(\omega_k)P(x|\omega_k)$$

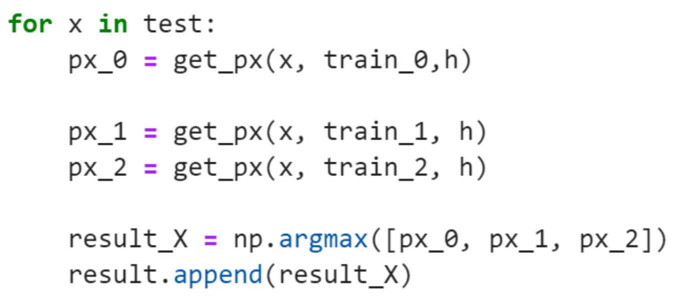
The example x is assigned to the label with the maximum $P(x|\omega_k)$

Method:

Since my prior probability is fixed, the prior probability can be ignored here and the accuracy can be obtained by directly comparing the calculated label with the correct label.

Take a row of data from the array, and then find $P(x|\omega_k)$ with train_0, train_1 and train_2 respectively. Then find the largest one, whose index is the category corresponding to this set of data.

Code:



```
for x in test:
    px_0 = get_px(x, train_0,h)

    px_1 = get_px(x, train_1, h)
    px_2 = get_px(x, train_2, h)

    result_X = np.argmax([px_0, px_1, px_2])
    result.append(result_X)
```

Result:

```
    for x in test:
        px_0 = get_px(x, train_0,h)

        px_1 = get_px(x, train_1, h)
        px_2 = get_px(x, train_2, h)

        result_X = np.argmax([px_0, px_1, px_2])
        result.append(result_X)


    print("result")
    print(result)
    return result
```

```
result
[0, 1, 2, 1, 0, 1, 2, 1, 2, 2, 0, 0, 1, 0, 0, 2, 1, 0, 2, 1, 0, 0, 2, 2, 1, 2, 0, 2, 1, 1]
```

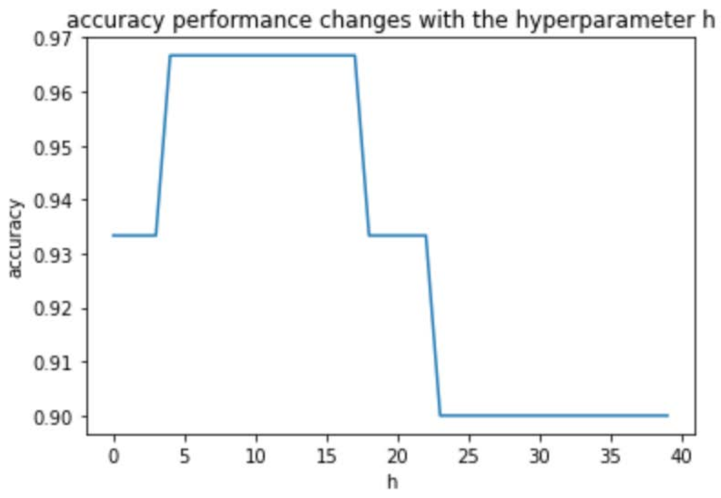
3.2.5

Show the figure that the accuracy performance changes with the hyperparameter *h* (real number)

Code:

```
plt.title('accuracy performance changes with the hyperparameter h')
plt.plot(result_re)
plt.xlabel("h")
plt.ylabel("accuracy")
plt.show
```

Result:



Since h ranges from 0.1 to 0.5 and is a step size of 0.01, the x-coordinate 0 corresponds to 0.01, and 10 corresponds to 0.2

3.3 KNN Method

3.3.1

For each test example, we fink K (K is a hyperparameter of KNN) nearest neighbor examples from n labeled training examples, where the distance is measured by the Euclidean distance(with the norm $||\cdot||_2$).

Euclidean distance formula：

$$dist(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Methods:

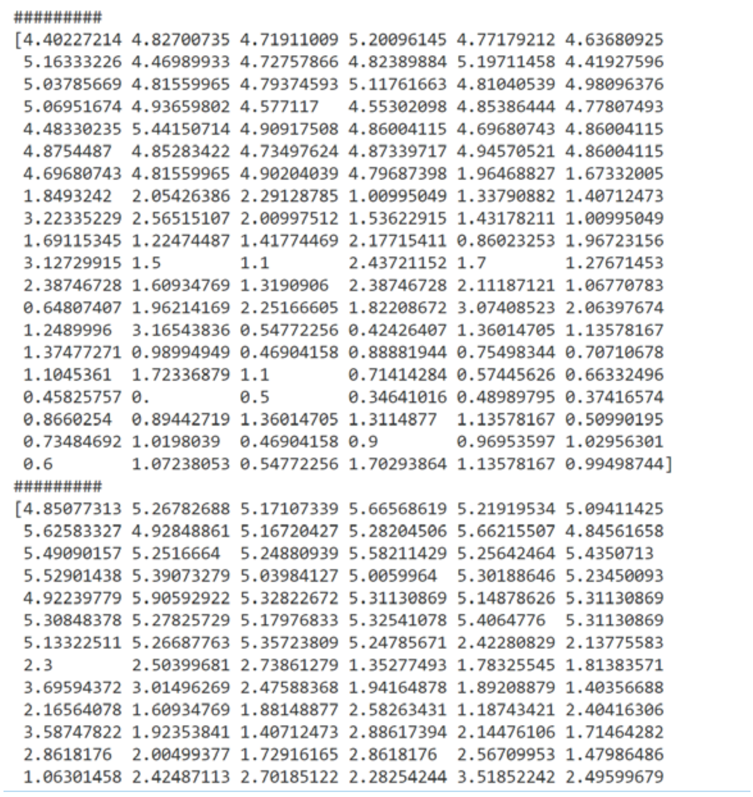
The idea is to take one data out of the array at a time and figure out the distance from the other data

Code:

```
distance = np.sqrt(np.sum((x - self.X) ** 2, axis = 1))
```

Result:

```
#Compute the Euclidean distance
distance = np.sqrt(np.sum((x - self.X) ** 2, axis = 1))
print("#########")
 print(distance)
```



```
#########
[4.40227214 4.82700735 4.71911009 5.20096145 4.77179212 4.63680925
 5.16333226 4.46989933 4.72757866 4.82389884 5.19711458 4.41927596
 5.03785669 4.81559965 4.79374593 5.11761663 4.81040539 4.98096376
 5.06951674 4.93659802 4.577117   4.55302098 4.85386444 4.77807493
 4.48330235 5.44150714 4.90917508 4.86004115 4.69680743 4.86004115
 4.8754487  4.85283422 4.73497624 4.87339717 4.94570521 4.86004115
 4.69680743 4.81559965 4.90204039 4.79687398 1.96468827 1.67332005
 1.8493242  2.05426386 2.29128785 1.00995049 1.33790882 1.40712473
 3.22335229 2.56515107 2.00997512 1.53622915 1.43178211 1.00995049
 1.69115345 1.22474487 1.41774469 2.17715411 0.86023253 1.96723156
 3.12729915 1.5        1.1        2.43721152 1.7        1.27671453
 2.38746728 1.60934769 1.3190906  2.38746728 2.11187121 1.06770783
 0.64807407 1.96214169 2.25166605 1.82208672 3.07408523 2.06397674
 1.2489996  3.16543836 0.54772256 0.42426407 1.36014705 1.13578167
 1.37477271 0.98994949 0.46904158 0.88881944 0.75498344 0.70710678
 1.1045361  1.72336879 1.1        0.71414284 0.57445626 0.66332496
 0.45825757 0.         0.5        0.34641016 0.48989795 0.37416574
 0.8660254  0.89442719 1.36014705 1.3114877  1.13578167 0.50990195
 0.73484692 1.0198039  0.46904158 0.9        0.96953597 1.02956301
 0.6        1.07238053 0.54772256 1.70293864 1.13578167 0.99498744]
#########
[4.85077313 5.26782688 5.17107339 5.66568619 5.21919534 5.09411425
 5.62583327 4.92848861 5.16720427 5.28204506 5.66215507 4.84561658
 5.49090157 5.2516664  5.24880939 5.58211429 5.25642464 5.4350713
 5.52901438 5.39073279 5.03984127 5.0059964  5.30188646 5.23450093
 4.92239779 5.90592922 5.32822672 5.31130869 5.14878626 5.31130869
 5.30848378 5.27825729 5.17976833 5.32541078 5.4064776  5.31130869
 5.13322511 5.26687763 5.35723809 5.24785671 2.42280829 2.13775583
 2.3        2.50399681 2.73861279 1.35277493 1.78325545 1.81383571
 3.69594372 3.01496269 2.47588368 1.94164878 1.89208879 1.40356688
 2.16564078 1.60934769 1.88148877 2.58263431 1.18743421 2.40416306
 3.58747822 1.92353841 1.40712473 2.88617394 2.14476106 1.71464282
 2.8618176  2.00499377 1.72916165 2.8618176  2.56709953 1.47986486
 1.06301458 2.42487113 2.70185122 2.28254244 3.51852242 2.49599679
```

We can see directly how far we're going to go each time

3.3.2

Estimate the posterior probability $P(x|\omega_k)$ as

$$P(x|\omega_k) = \frac{K_i}{K}$$

where K_i is the number of the example from K nearest neighbor examples.

Methods:

after sorting by index, intercept the first K elements

Code:

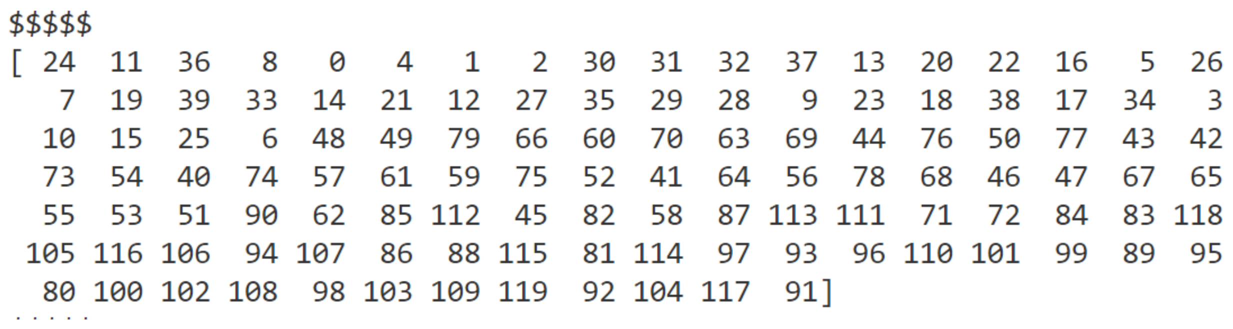
```
 """
After sorting the distance array,
returns the index of each element in the original array (the array
before sorting)
"""
index = distance.argsort()
```

```
        """
        Truncate the distance index array a
        take the index of the first K elements
        """
        index = index[:self.k]

        """
        Count the number of occurrences of each element in the index array
        """
        count = np.bincount(self.y[index])
```

Result:

index of each element in the original array

```
        """
        After sorting the distance array,
        returns the index of each element in the original array
        (the array before sorting)
        """
        index = distance.argsort()
        print("$$$$$")
         print(index)
```



```
$$$$$
[ 24  11  36   8   0   4   1   2  30  31  32  37  13  20  22  16   5  26
   7  19  39  33  14  21  12  27  35  29  28   9  23  18  38  17  34   3
  10  15  25   6  48  49  79  66  60  70  63  69  44  76  50  77  43  42
  73  54  40  74  57  61  59  75  52  41  64  56  78  68  46  47  67  65
  55  53  51  90  62  85 112  45  82  58  87 113 111  71  72  84  83 118
 105 116 106  94 107  86  88 115  81 114  97  93  96 110 101  99  89  95
  80 100 102 108  98 103 109 119  92 104 117  91]
```

We can see the sorted list of index values

The index list after the first K elements

```
        """
        Truncate the distance index array a
        take the index of the first K elements
        """
        index = index[:self.k]
        print("$$$$$")
       print(index)
```

```
$$$$$
[117 104 109  92 108 119  91 103  98 100  89 101  80  99 102  97  93  95
  81  86 107 110  96  88 114 116  72  58  94  62 115  53 111  45]
$$$$$
[ 83 118 105  82 113  71 112  90 106  85  87  94 111  84  65 114  56 115
  41  96  46 107 110  64  47 116  81  75  88  51  72  52  54  59]
$$$$$
[119 103 108 104  98  89  92 100  95  91 101  97 109  99  81 117  80  96
 110 107 114 102  94  86  72  93 115 116  58  88 111  45  62  87]
$$$$$
[32 16 36  8 13 22  7 39 37  5 11 33  2 24  4 28  0 19 20 14 35 27 29 21
  1 12 23 38  9 30 17 18 31 26 34]
```

In this result, there are the cases of K=34 and K=35, which well reflects the interception of the previous K indexes according to the K value

3.3.3

Decision rule: The example x is assigned to the label with the maximum $P(x|\omega_k)$

Method:

Find the index that appears most frequently among them, and the corresponding label is the category to which the element belongs

Code:

```
"""
Returns the index corresponding to the element
 with the most occurrences
which is the label corresponding to this set of data
"""
result.append(count.argmax())
```

Result:

```
"""
Returns the index corresponding to the element with
the most occurrences
which is the label corresponding to this set of data
"""
result.append(count.argmax())

print("$$$$$")
 print(result)
```

```
$$$$$
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2]
$$$$$
[0]
```

As you can see, when the loop ends, the result list stores 30 predictions

3.3.4

Show the figure that the accuracy performance changes with the hyperparameter *K* (natural number)

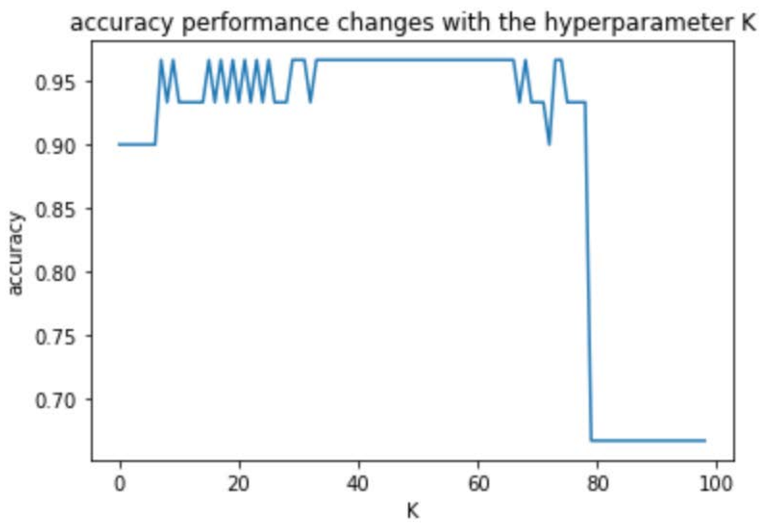
Here, I set the value range of K from 1 to 100 to observe the change curve of K and accuracy

Code:

```
for i in range(1,100):

    knn = KNN(k = i)
    # train
    knn.train_KNN(train_X, train_y)

    # tsst

    result_1 = knn.Predict(test_X_1)

    #Computational accuracy
    r_1 = np.sum(result_1 == test_y_1) / len(result_1)

#     avg = (r_1 + r_2 + r_3 + r_4 + r_5) / 5
    result_re.append(r_1)
```

```
# Draw a graph to show the change of accuracy with K value
plt.title('accuracy performance changes with the hyperparameter K')
plt.plot(result_re)
plt.xlabel("K")
plt.ylabel("accuracy")
plt.show
```

Result:



4 Comparison and improvement

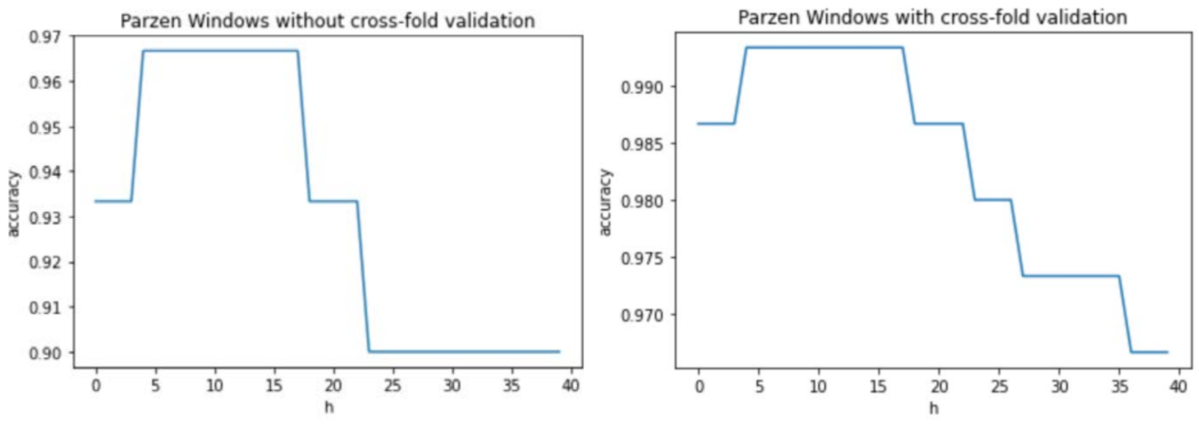
4.1 5 cross-fold validation

4.1.1 Parzen Windows

Using a cross-fold validation in the Parzen window, you average the five sets of data at the end to get a more accurate prediction. Code is at the 3.1.2

Result:

The results here are reflected by the line graph of accuracy



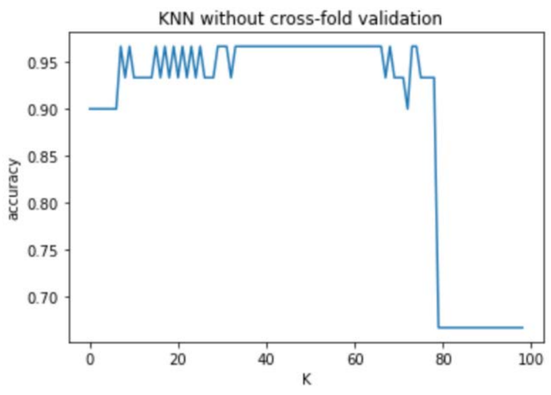
For example, the line graph can well reflect the difference between cross-validation and non-cross-validation. The accuracy of cross-validation is higher.

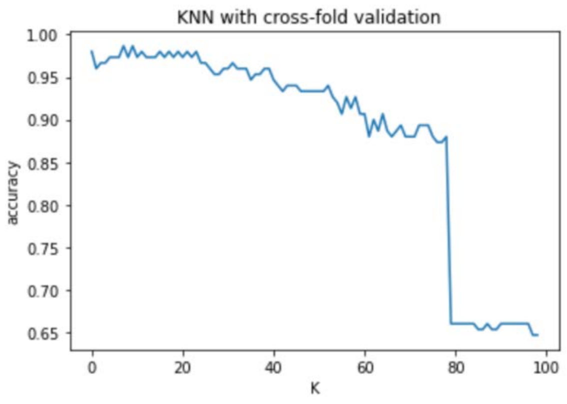
4.1.2 KNN

Using a cross-fold validation in the Parzen window, you average the five sets of data at the end to get a more accurate prediction. Code is at the 3.1.2

Result:

The results here are reflected by the line graph of accuracy





For example, line graph can well reflect the difference between cross-validation and non-cross-validation, and cross-validation is more accurate.

4.2 Seed and Sample

There is essentially no difference between the two, except that one is for array elements and one is for list elements, so here we compare the results of the experiment by doing two operations on A.

Code:

```python
import numpy as np
import pandas as pd
import random



# read the dataser iris
data = pd.read_csv('iris.data',header=None)

# set the seed
random.seed(17)

# mapping dataset
data[4] = data[4].map({'Iris-setosa':0,
                           'Iris-versicolor':1,
                           'Iris-virginica':2})
A = data[0:10]
print(A)
```

```
     0    1    2    3  4
0  5.1  3.5  1.4  0.2  0
1  4.9  3.0  1.4  0.2  0
2  4.7  3.2  1.3  0.2  0
3  4.6  3.1  1.5  0.2  0
4  5.0  3.6  1.4  0.2  0
5  5.4  3.9  1.7  0.4  0
6  4.6  3.4  1.4  0.3  0
7  5.0  3.4  1.5  0.2  0
8  4.4  2.9  1.4  0.2  0
9  4.9  3.1  1.5  0.1  0
```

Use sample:

```python
A = A.sample(len(A), random_state = 17)
```

	0	**1**	**2**	**3**	**4**
6	4.6	3.4	1.4	0.3	0
0	5.1	3.5	1.4	0.2	0
7	5.0	3.4	1.5	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
8	4.4	2.9	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
9	4.9	3.1	1.5	0.1	0
1	4.9	3.0	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0

Use random

```
A = A.values
random.shuffle(A)
print(A)
```

```
array([[4.6, 3.4, 1.4, 0.3, 0. ],
       [4.6, 3.4, 1.4, 0.3, 0. ],
       [4.6, 3.4, 1.4, 0.3, 0. ],
       [5. , 3.4, 1.5, 0.2, 0. ],
       [5.1, 3.5, 1.4, 0.2, 0. ],
       [5. , 3.4, 1.5, 0.2, 0. ],
       [5. , 3.4, 1.5, 0.2, 0. ],
       [5. , 3.6, 1.4, 0.2, 0. ],
       [4.7, 3.2, 1.3, 0.2, 0. ],
       [4.9, 3. , 1.4, 0.2, 0. ]])
```

You can clearly see that both SAMPLE and Random mix up the data so well that there is no difference

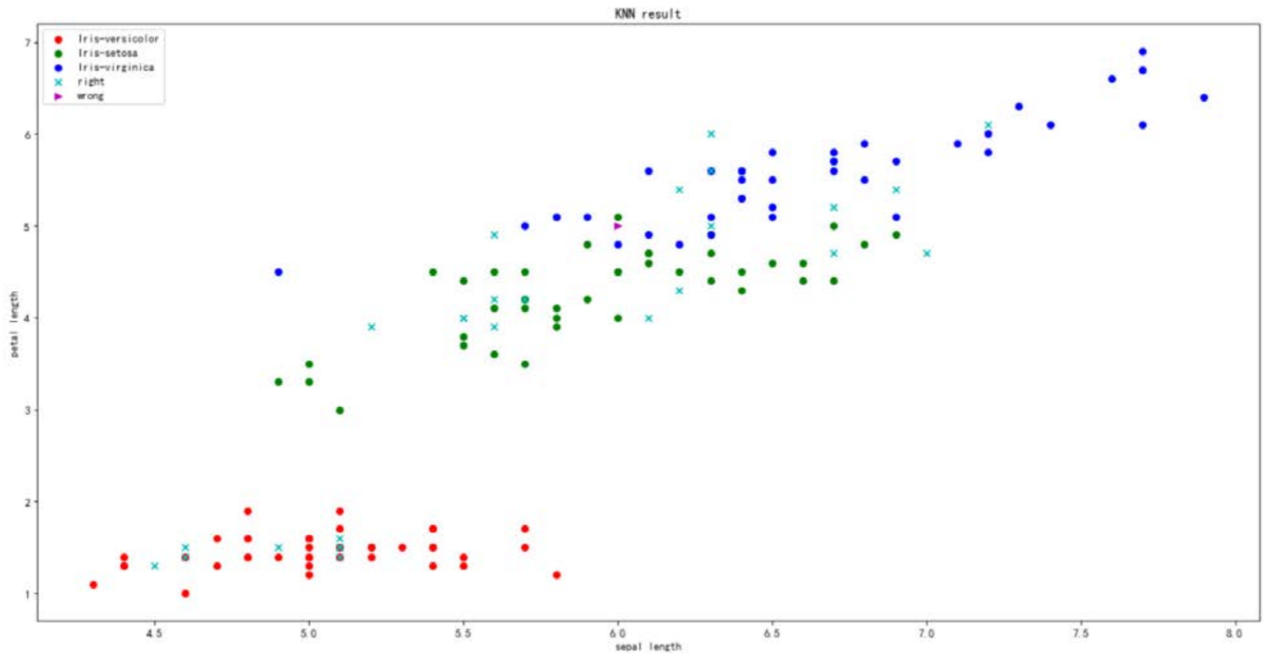
4.3 A visual reflection of the results

For KNN algorithm, we use the image to reflect the prediction results, making the observation of the results more intuitive

Code:

```
# "Iris-versicolor":0,"Iris-setosa":1,"Iris-virginica":2
import matplotlib as mpl
import matplotlib.pyplot as plt
plt.figure(figsize=(20,10))

plt.scatter(x=t0[0][:40], y=t0[2][:40], color='r', label="Iris-versicolor")
plt.scatter(x=t1[0][:40], y=t1[2][:40], color='g', label="Iris-setosa")
plt.scatter(x=t2[0][:40], y=t2[2][:40], color='b', label="Iris-virginica")
right = test_X[result == test_y]
wrong = test_X[result != test_y]
plt.scatter(x=right[0], y=right[2], color='c', label="right", marker="x")
plt.scatter(x=wrong[0], y=wrong[2], color='m', label="wrong", marker=">")
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('KNN result')
plt.legend(loc='best')
plt.show()
```

Result：



We can see that Wrong appears in the handover of the two categories, where KNN tends to confuse label

5 conclusion

In Parzen Windows algorithm, the formula is used to first calculate the distance, and then conduct index sorting according to the sum of the distances. The largest index is the category corresponding to this data. Based on this method, we get that the best H appears near 0.15, and the accuracy is about 0.96 at this time.

In KNN algorithm, index sorting is carried out according to the Euclidean distance obtained, and then the first K elements are intercepted to find the category with the most occurrences, that is, the category corresponding to this data. Based on this method, the accuracy can reach 0.99 when K is appropriate.