```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
cd drive/MyDrive/hw1
```

```
[Errno 2] No such file or directory: 'drive/MyDrive/hw1'
/content
```

Problem 1: Python & Data Exploration

```
import numpy as np
import matplotlib.pyplot as plt

iris = np.genfromtxt("data/iris.txt",delimiter=None) # load the text file
Y = iris[:,-1] # target value is the last column
X = iris[:,0:-1] # features are the other columns
```

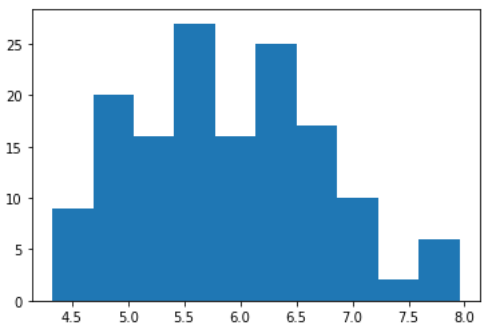
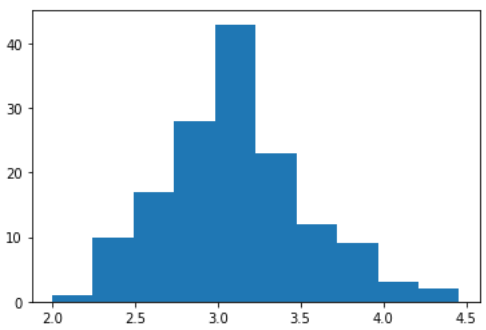
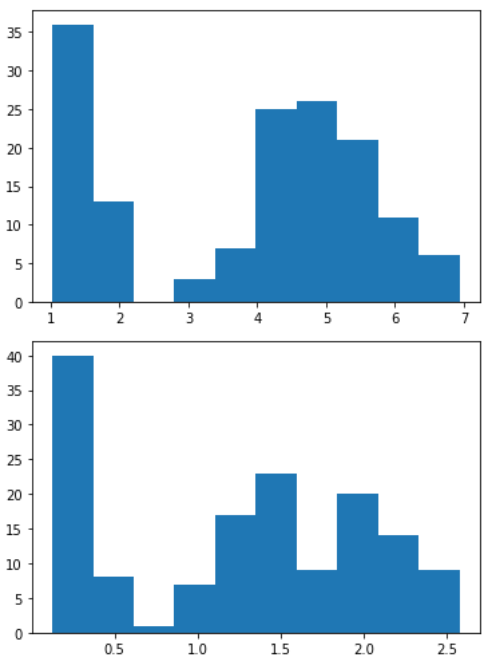
1.1 number of data points: 148
number of features: 4

```
X.shape
```

```
(148, 4)
```

1.2 histogram for data values

```
plt.hist(X[:,0])
plt.show()
plt.hist(X[:,1])
plt.show()
plt.hist(X[:,2])
plt.show()
plt.hist(X[:,3])
plt.show()
```

1.3 mean & standard deviation for each feature

```
print("mean and std for feature 0:")
print(np.mean(X[:,0]))
print(np.std(X[:,0]))
print("mean and std for feature 1:")
print(np.mean(X[:,1]))
print(np.std(X[:,1]))
print("mean and std for feature 2:")
print(np.mean(X[:,2]))
print(np.std(X[:,2]))
print("mean and std for feature 3:")
print(np.mean(X[:,3]))
print(np.std(X[:,3]))
```

```
mean and std for feature 0:
5.900103764189188
```

```
0.833402066774894
mean and std for feature 1:
3.098930916891892
0.43629183800107685
mean and std for feature 2:
3.8195548405405404
1.7540571093439352
mean and std for feature 3:
1.2525554845945945
0.7587724570263247
```

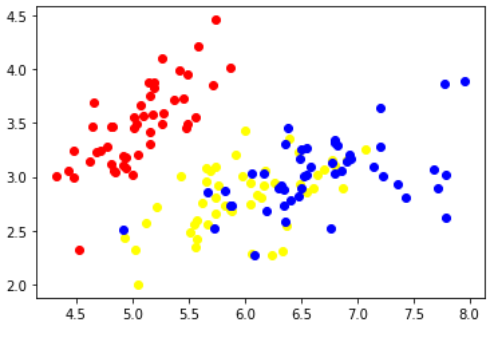
1.4 plot a scatterplot

Y=0: 0~48

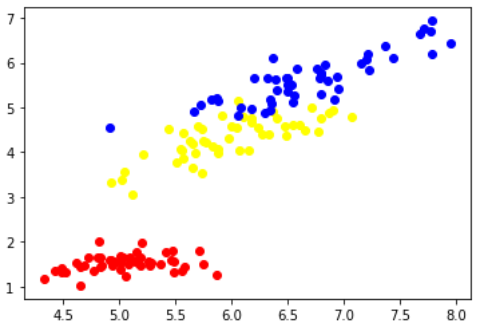
Y=1: 49~98

Y=2: 99~147

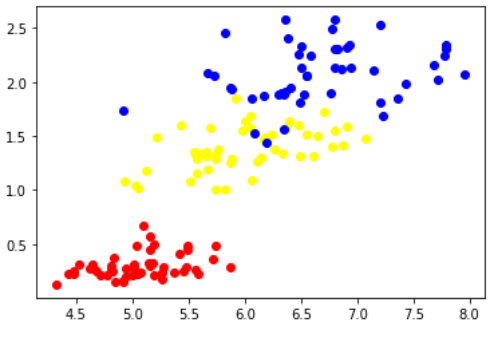
```
#  feature  1  and  2
plt.scatter(X[0:48,0],X[0:48,1],  color='red')
plt.scatter(X[49:98,0],X[49:98,1],  color='yellow')
plt.scatter(X[99:147,0],X[99:147,1],  color='blue')
plt.show()
```



```
#  feature  1  and  3
plt.scatter(X[0:48,0],X[0:48,2],  color='red')
plt.scatter(X[49:98,0],X[49:98,2],  color='yellow')
plt.scatter(X[99:147,0],X[99:147,2],  color='blue')
plt.show()
```



```
#  feature  1  and  4
plt.scatter(X[0:48,0],X[0:48,3],  color='red')
plt.scatter(X[49:98,0],X[49:98,3],  color='yellow')
plt.scatter(X[99:147,0],X[99:147,3],  color='blue')
plt.show()
```



Problem 2: k-Nearest Neighbor (kNN) exercise

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

```
#  Run  some  setup  code  for  this  notebook.

import  random
import  numpy  as  np
from  cs273p.data_utils  import  load_CIFAR10
import  matplotlib.pyplot  as  plt

#  This  is  a  bit  of  magic  to  make  matplotlib  figures  appear  inline  in  the  notebook
#  rather  than  in  a  new  window.
%matplotlib  inline
plt.rcParams['figure.figsize']  =  (10.0,  8.0)  #  set  default  size  of  plots
plt.rcParams['image.interpolation']  =  'nearest'
plt.rcParams['image.cmap']  =  'gray'

#  Some  more  magic  so  that  the  notebook  will  reload  external  python  modules;
#  see  http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
#  %load_ext  autoreload
#  %autoreload  2


%cd  cs273p/datasets
!source  get_datasets.sh
```

```
/content/drive/MyDrive/hw1/cs273p/datasets
--2023-01-19 07:08:00--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: 'cifar-10-python.tar.gz'

cifar-10-python.tar 100%[===================>] 162.60M  29.7MB/s    in 6.2s

2023-01-19 07:08:06 (26.4 MB/s) - 'cifar-10-python.tar.gz' saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
```

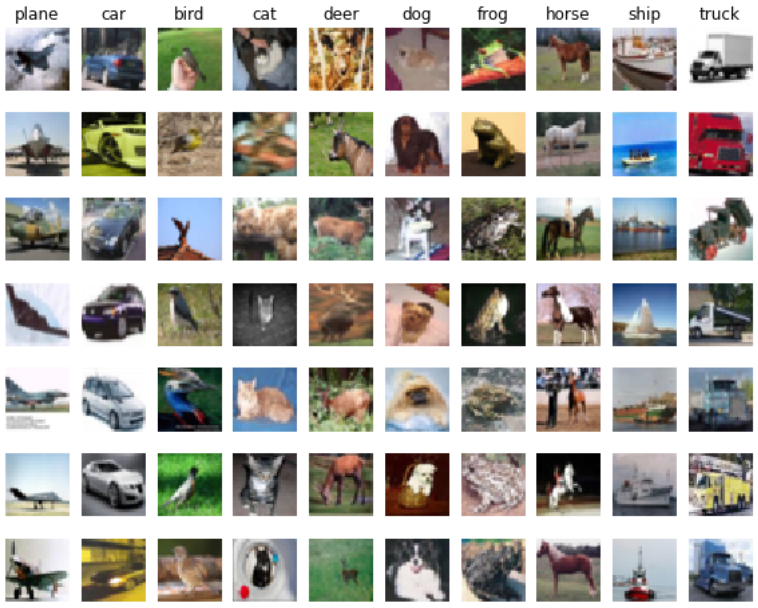
```
%cd ../..
```

```
/content/drive/MyDrive/hw1
```

```
# Load the raw CIFAR-10 data.
cifar10_dir = './cs273p/datasets/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```




```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]


# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

```
from cs273p.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

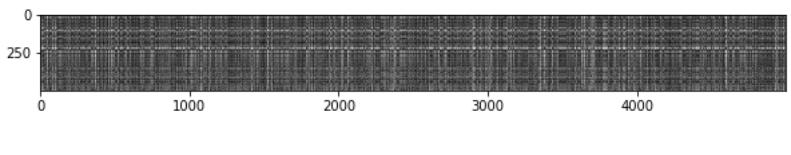
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs273p/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

```
# Open cs273p/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
print(dists)
```

```
(500, 5000)
[[3803.92350081 4210.59603857 5504.0544147  ... 4007.64756434
  4203.28086142 4354.20256764]
 [6336.83367306 5270.28006846 4040.63608854 ... 4829.15334194
  4694.09767687 7768.33347636]
 [5224.83913628 4250.64289255 3773.94581307 ... 3766.81549853
  4464.99921613 6353.57190878]
 ...
 [5366.93534524 5062.8772452  6361.85774755 ... 5126.56824786
  4537.30613911 5920.94156364]
 [3671.92919322 3858.60765044 4846.88157479 ... 3521.04515734
  3182.3673578  4448.65305458]
 [6960.92443573 6083.71366848 6338.13442584 ... 6083.55504619
  4128.24744898 8041.05223214]]
```

```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question #1: Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer:

- 1. The distinctly bright rows are test cases that are different to most of the training examples, so the columns are brighter, which means they have high distances with other training examples.
- 2. The distinctly bright columns are, oppositely, training examples that are different to most of the test cases, so they also have high distances with other test cases.

```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words, reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
        print('Good! The distance matrices are the same')
else:
        print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
        print('Good! The distance matrices are the same')
else:
        print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# you should see significantly faster performance with the fully vectorized implementation
```

```
Two loop version took 28.266758 seconds
One loop version took 30.440825 seconds
No loop version took 0.589230 seconds
```

Cross-validation

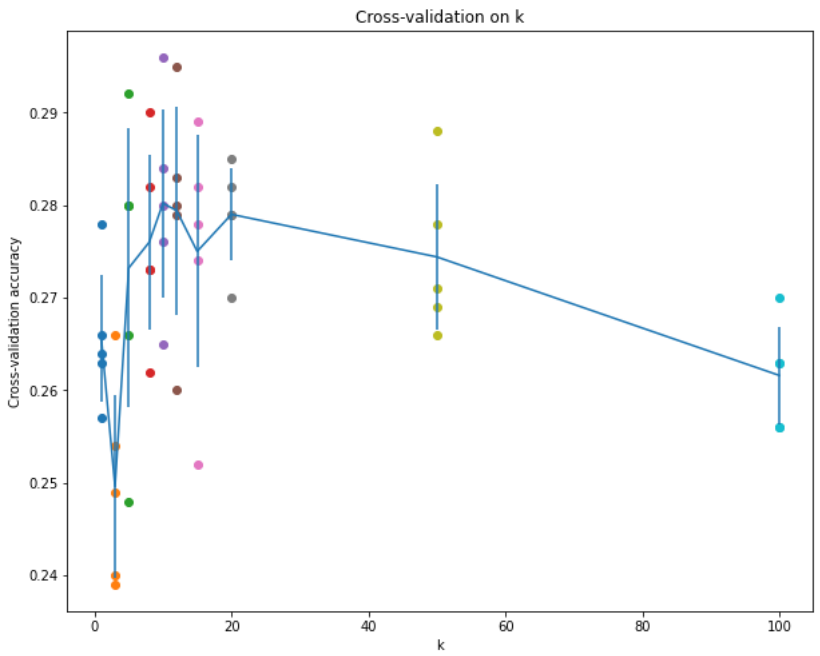
We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################

'''split them into 5 folds with eq sizes'''
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}

################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
for k in k_choices:
    print(k)
    k_to_accuracies[k] = list()
    for i in range(num_folds):
        #print(np.array(X_train_folds[0:i]).shape, np.array(X_train_folds[i:num_folds]).shape)
        #print(X_train_folds[0:i])
        #X_Train_Data = np.concatenate((X_train_folds[0:i], X_train_folds[i:num_folds]))
        #y_Train_Data = np.concatenate((y_train_folds[0:i], y_train_folds[i:num_folds]))
        #print(X_train_folds[0:i])

        X_Train_Data = np.concatenate((X_train_folds[0:i]+ X_train_folds[i+1:num_folds]))
        y_Train_Data = np.concatenate((y_train_folds[0:i]+ y_train_folds[i+1:num_folds]))

        X_Valid_Data = X_train_folds[i]
        y_Valid_Data = y_train_folds[i]

        classifier.train(X_Train_Data, y_Train_Data)
        dists_val = classifier.compute_distances_no_loops(X_Valid_Data)
        y_val_pred = classifier.predict_labels(dists_val, k)
        val_num_correct = np.sum(y_val_pred == y_Valid_Data)
        accuracy = float(val_num_correct) / X_Valid_Data.shape[0]
        k_to_accuracies[k].append(accuracy)


################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
5
8
10
12
15
20
50
100
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
```

```
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```




```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

Problem 3: Naïve Bayes Classifiers

You don't need to code this question. You can either type your answer or attach an image of hand written solution here.

Q1.

p(y=1) = 4/10, P(y=-1) = 6/10

P(xi|yj)	y=1	y=-1
X1=1	3/4	3/6
X1=0	1/4	3/6
X2=1	0/4	5/6
X2=0	4/4	1/6
X3=1	3/4	4/6
X3=0	1/4	2/6
X4=1	2/4	5/6
X4=0	2/4	1/6
X5=1	1/4	2/6
X5=0	3/4	4/6

Q2.

1. x=(0,0,0,0,0)

P(x=(0,0,0,0,0)|y=1) = 1/4 * 4/4 * 1/4 * 2/4 * 3/4 = 3/128

P(x=(0,0,0,0,0)|y=-1) = 3/6 * 1/6 * 2/6 * 1/6 * 4/6 = 1/324

P(x=(0,0,0,0,0)|y=1) * p(y=1) = 3/128 * 4/10 = 3/320

P(x=(0,0,0,0,0)|y=-1) * p(y=-1) = 1/324 * 6/10 = 1/540

3/320 > 1/540 => predict x=(0,0,0,0,0) as y=1

2. x=(1,1,0,1,0)

P(x=(1,1,0,1,0)|y=1) = 3/4 * 0/4 * 1/4 * 2/4 * 3/4 = 0

P(x=(1,1,0,1,0)|y=-1) = 3/6 * 5/6 * 2/6* 5/6 * 4/6 = 25/324

P(x=(1,1,0,1,0)|y=1) * p(y=1) = 0 * 4/10 = 0

P(x=(1,1,0,1,0)|y=-1) * p(y=-1) = 25/324 * 6/10 = 5/108

0<5/108 => predict x=(1,1,0,1,0) as -1

Q3. posterior probability = P(x=(1,1,0,1,0)|y=1) / P(x=(1,1,0,1,0)) = 0/ P(x=(1,1,0,1,0)) = 0

4.

because we don't have enough data to compute p(x1,x2,x3,x4,x5|y=1) and p(x1,x2,x3,x4,x5|y=-1)

the dataset has only 10 rows, which is much fewer than 2^5 parameters.

many combinations will be seen as "impossible" if we do so, because we can't find same data in the dataset.

5.

we can use the same model, for all of the x features are independent in naïve bayes, but the algorithm should exclude x1.

because if the model misses x1 feature, it should compute and compare between

p1 = p(x2|y=1)*p(x3|y=1)* p(x4|y=1)*p(x5|y=1) / p(y=1)

and

p2 = p(x2|y=-1)*p(x3|y=-1)* p(x4|y=-1)*p(x5|y=-1) / p(y=-1)

if we assume all authors are unknown, the probability will be

p1' = p1 * p(x1=0|y=1)

and

p2' = p2 * p(x1=0|y=-1)

this changes the result because p(x1=0|y=1) != p(x1=0|y=-1)

hence, our algorithm should exclude x1 feature and compute the rest of them.

Statement of Collaboration:

Yinfeng Cong:

- we discussed about implementation of 2-loop, 1-loop and no-loop KNN function.

✓ 24 秒 完成時間：下午5:14